**UNIVERSITE Paris DIDEROT**
**ORANGE LABS**

**Rapport de stage** M2 IMPAIR 2016-2017

---

# Etude approfondie sur la performance des technologies unikernel

---

**Auteur :**
ROUX Maximilien
*maximilien.roux@informatique.univ-paris-diderot.fr*

**Encadrants :**
Dr. DANG TRAN Frédéric
Dr. SANGNIER Arnaud

Stage effectué du 27 mars 2017 au 26 septembre 2017

**Orange Labs Products and Services**
**44 avenue de la République, 92326 CHATILLON**

# Table des matières

# 1 Introduction

## 1.1 Contexte

Au sein de l'industrie des télécommunications, les entreprises s'intéressent grandement aux environnements virtualisés qui offrent agilité, fléxibilité et qui permettent une réduction des coûts. Tandis que les data centers et les architectes réseaux se tournent vers des solutions basées sur les Software Defined Network (SDN) qui offrent une meilleure gestion des réseaux par l'abstraction de leurs fonctionnalités, les fournisseurs de services font évoluer leurs réseaux vers une architecture basée sur le principe des Network Function Virtualization (NFV). Cela consiste à virtualiser les services et fonctions réseau en lieu et place d'un matériel dédié et propriétaire.

Le concept de la virtualisation n'est pas nouveau dans l'industrie mais le déploiement d'applications et de services virtualisés est en plein essor avec pour but la performance à moindre coût. Nous traversons un ère où le cycle de vie et le déploiement de service doit être rapide et simple. C'est pourquoi, passer par des environnement virtualisés qui ne dépendent pas du matériel ou d'un système d'exploitation hôte spécifique est une nécessité. Il faut être capable de s'adapter à la demande et déployer ou stopper des services en fonction du traffic. Ceci n'est pas faisable avec du matériel dédié, alors que les NFV encouragent le développement de solutions logicielles s'exécutant sur des machines génériques. Ce sont donc ces NFV qui rendent le réseau plus flexible et rendent les infrastructures réutilisables dans le sens où elle peuvent être utilisées pour n'importe quelles NFV, il n'y a pas d'infrastructure liée à un service spécifique.

## 1.2 Environnement de travail

J'ai effectué mon stage chez Orange, qui est une des entreprises leader en Europe dans le domaine des télécommunications. La structure Orange Lab que j'ai rejoint travaille sur le développement de nouveaux services réseau pour les entreprises axés autour de trois domaines principaux : l'amélioration du réseau, le cloud et la live-box destinés aux entreprises. J'ai travaillé dans l'équipe LOcation Virtualization (LOV) qui elle s'intéresse à l'informatique du cloud et plus particulièrement à tout ce qui est infrastructure en tant que service (IaaS) et plateforme en tant que service (PaaS) qui sont des solutions de gestion d'infrastructures virtualisées.

## 1.3 Remerciements

Je souhaiterais remercier Orange et toute l'équipe d'Orange Lab qui m'a accueilli pendant 6 mois pour ce stage. Je remercie plus particulièrement Frédéric, mon tuteur de stage, et Ruby qui m'ont conseillé pour l'écriture de ce rapport mais qui m'ont surtout encadré, fait confiance et soutenu dans mon travail. Le contenu du stage était intéressant et conforme à mes attentes, j'ai pris plaisir à travailler sur différentes problématiques autour des unikernels et cela a été très enrichissant. Je remercie également Xuecan qui était aussi stagiaire et avec qui j'ai beaucoup apprécié travailler.

## 1.4 Motivations du stage

Il apparaît que les techniques de virtualisation actuelles ne sont pas adaptées à certaines classes de fonctions réseau. Parmi ces fonctions, on retrouve :

- Les routeurs, qui n'ont pas simplement pour but de diriger sur le réseaux des paquets mais qui peuvent aussi agir sur ces paquets (contrôle parental, filtrage d'adresses, firewall).
- Les middleboxes, qui regroupent toutes les applications réseau qui sont implémentées grâce à du matériel dédié et que l'on souhaiterait mettre sous forme logiciel.
- Le cloud Radio Access Network (cloud RAN). Actuellement, l'accès à un réseaux mobile se fait par un accès radio dont les fonctionnalités sont placées dans des machines spécifiques (que l'on trouve souvent au pieds des antennes) et qui couvrent une petite zone. Le cloud RAN est une alternative centralisée où les fonctionnalités de ses machines se trouveraient dans le cloud.

Ces fonctions réseau nécessitent de la performance, elles sont aussi sensibles à la latence et sont des fonctions temps réel bien que non critiques. Elles ont besoin de performances pour récupérer les paquets sur le réseau et pour les traiter, ce qui peut passer par une accélération matérielle. Elles ont aussi besoin d'isolation (ce qui est traditionnellement une des raisons principales de la virtualisation) car l'exécution d'une application ne doit pas perturber celle des autres. De plus les applications sensibles à la latence ont besoin d'un temps de réponse le plus rapide possible pour être le moins possible impacté par la latence ou elles doivent être au moins exécutées avant une certaine échéance.

La virtualisation est aujourd'hui largement utilisée au travers du cloud avec des machines virtuelles classiques ou des containers. Mais cela n'est pas suffisant pour répondre au besoin de regrouper sur une même machine et mettre sous forme de logiciel des applications et services aujourd'hui réalisés en grande partie par du matériel spécialisé et coûteux. Si on regarde l'état actuel des choses des améliorations sont nécessaires, notamment au niveau des performances, de la consommation des ressources et de la sécurité. Les unikernels sont une technologie émergente et peuvent servir de support d'exécution pour les fonctions réseau présentées plus haut. C'est sur cette technologie que j'ai travaillé pendant mon stage et plus particulièrement sur leurs performances au niveau réseau et du transfert de paquets (accès aux paquets via la carte réseau, transmission de paquets entre des services d'un même serveur...). L'intitulé de mon sujet de stage était: "Etude approfondie sur la performance des technologies unikernel".

# 2 Etat de l'art des unikernels[1]

Dans la partie précédente, j'ai introduit les unikernels comme une technologie permettant de répondre à certaines problématiques qui peuvent se présenter aujourd'hui et pour lesquelles nous n'avons pas encore de réponses suffisamment satisfaisantes. Durant le début de mon stage, j'ai été amené à effectuer un état de l'art sur les unikernels et à en étudier certains plus précisément dans l'optique d'en sélectionner un ou plusieurs correspondant à l'utilisation voulue par les motivations du stage. Cet état de l'art à été réalisé en collaboration avec mon co-stagiaire qui travaillait lui sur l'ordonnancement des unikernels. Je vais donc, dans cette partie, retranscrire l'état de l'art effectué en tentant de définir ce que sont les unikernels, ce qui les caractérise et les différencie des solutions utilisées actuellement comme les machines virtuelles classiques et les containers et en quoi ils sont mieux adaptés. Je vais aussi parler des différents types d'unikernels et présenter quatre unikernels autour desquels nos réflexions se sont portées. Et enfin j'expliquerai sur quel(s) unikernel(s) notre choix s'est finalement arrêté et en donnerai les raisons.

## 2.1 Définition

*Un unikernel est une image spécialisée avec un unique espace d'adresse, construite à partir de systèmes d'exploitation dit "systèmes d'exploitation librairies".*

Ceci est la définition qu'en donne le site spécialisé sur les technologies unikernels [1]. D'un point de vue fonctionnel, les unikernels sont donc des images de tailles réduites (parfois très reduites), offrant un système à déploiement rapide et sécurisé. Toutes ces caractéristiques pouvant parfois faire défaut aux solutions utilisées aujourd'hui.

Les images spécialisées obtenues en compilant des langages de haut niveau seront ensuite exécutées sur un hyperviseur ou directement sur le matériel. Il existe deux types d'hyperviseurs, les hyperviseurs de type I et les hyperviseurs de type II. Les hyperviseurs de type I sont des hyperviseurs natifs, c'est à dire qu'ils s'exécutent directement sur le matériel contrairement à ceux de type II qui sont "hosted", c'est à dire qu'ils vont s'exécuter à l'intérieur d'un système d'exploitation.

Sur la figure 1, on peut voir le schéma des différentes couches de la pile d'exécution du matériel à l'applicatif pour les deux types d'hyperviseurs. On remarque la couche supplémentaire induite par les hyperviseurs de type II du fait qu'ils aient besoin d'un hôte pour s'exécuter. Cela introduit une perte au niveau de l'isolation et moins de sécurité. C'est pourquoi, les principaux supports à l'exécution des unikernels sont les hyperviseurs de type I ou le matériel.

[1] Références: [1, 2, 3, 4, 5, 6, 7, 8, 9]

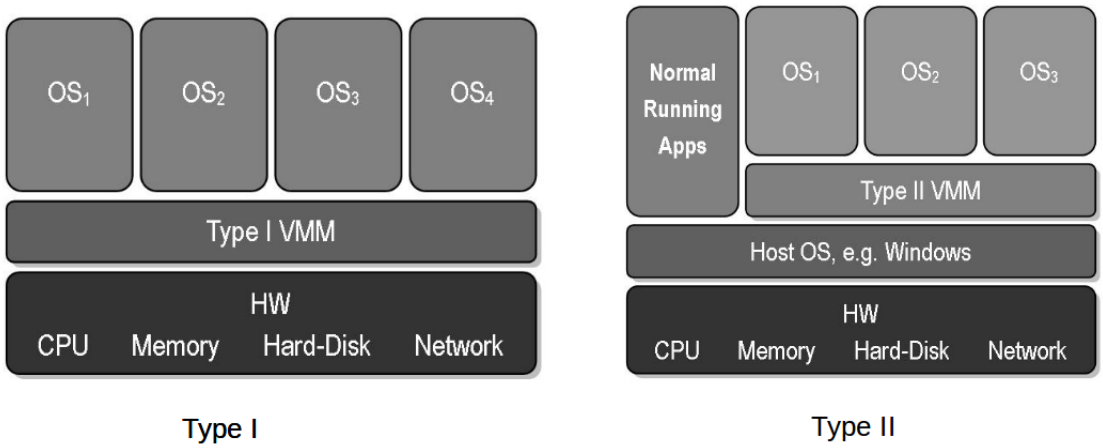


Figure 1: Les deux types d'hyperviseurs.

## 2.2 Les machines virtuelles classiques

En premier lieu, on retrouve les machines virtuelles s'exécutant un système d'exploi- tation traditionnel tels que Linux ou Windows pour fournir des services. Ces machines virtuelles font tourner quelques applications spécifiques sur un système d'exploitation généraliste fournissant un large panel d'outils souvent non utilisés par ces machines. Il en résulte que l'exécution de ces images occupent plusieurs gigaoctet de ressource mémoire et de stockage. Avec ce type de machines virtuelles, on arrive vite à un besoin accru en serveurs et donc en électricité conséquent. En effet, la multiplication de ces machines requiert une accumulation conséquente de serveurs pour fournir les ressources nécessaires. Et par transitivité, cela entraîne un besoin plus important d'énergie pour refroidir les machines physiques et donc entraîne une augmentation significative des coûts. On pourrait mettre plus d'applications sur une même machine virtuelle pour avoir moins de machines mais cela se ferait au dépend de la qualité du service. Regrouper sur une même machine sans aucune isolation des applications indépendantes n'est pas envisageable d'où multiplication des machines virtuelles et de tout ce qui va avec. Ces machines sont aussi lentes à démarrer, il leur faut parfois plusieurs minutes. Cela induit donc qu'elles restent constamment allumées.Ainsi, elle surchargent et ralentissent le cloud. De plus, elles induisent souvent un ralentissement des performances et elles ne sont pas très bien isolées en terme de ressources (CPU, réseau...).

Mais en plus de cela, ces machines virtuelles classiques sont confrontées à un autre problème majeur et à enjeu: la sécurité. La surface d'attaque engendrée par une machine virtuelle classique est conséquente. Ces dernières années, ont été recensé de plus en plus de failles de sécurité et d'intrusions chez de grosses compagnies. Des gouvernements ont même reporté des attaques réussies sur leurs infrastructures. La surface d'attaque offerte par les machines virtuelles classique est conséquente de par les outils qu'elles embarquent (par exemple le shell) et qui ne sont parfois

pas utilisés mais qui sont présents car cela fait partie intégrante du système. La sécurité est devenue une issue importante ces dernières années et beaucoup de choses ont été mises en place pour prévenir et empêcher des attaques. Les données sont confinées derrière plusieurs anneaux de sécurité entourés par des firewall. Cela est une très bonne avancée mais le problème n'est que repoussé et il y a toujours des personnes capables de hacker les systèmes de sécurité et d'acceder aux données. Ce type de protection oblige donc à fournir des efforts conséquent pour mettre à jour et patcher tous les outils régulièrement et cela sans garantie. Le problème vient de la nature même de la pile d'application traditionnelle. Elle repose sur un système complexe basé sur des systèmes d'exploitation multi-utilisateurs et multi-utilisations qui disposent d'une pléthore d'outils par lequel une attaque peut subvenir.

## 2.3 Les containers

Pour tenter de pallier les problèmes inhérents aux machines virtuelles, est apparu il y a plusieurs année la technologie des containers. Ils ont très vite gagné en succès et sont aujourd'hui très utilisés, notamment la solution de container proposée par Docker.

Les containers repose sur le mécanisme Linux des *control groups* (cgroup) et s'exécutent sur un système d'exploitation hôte. Un cgroup est un ensemble de processus liés par un critère spécifique. Il peut y avoir une relation de filiation entre différents groupes. Cela permet donc une isolation des processus présents au sein d'un même cgroup et de les gérer séparément du reste des processus (ressources utilisées, priorisation des processus du groupe par le système, ect...).

Les containers utilisent les cgroup avec une isolation par espace de noms sur les PIDs des processus. Les espaces de noms sur Linux permettent une isolation de certaines ressources comme les points de montage, les PIDs des processus, les piles réseau ou les user IDs pour virtualiser des ressources et les isoler. Ici, le premier processus créé dans ce cgroup aura un PID de 1 et tous les autres ensuite incrémenteront cette valeur, leur numérotation ne suivra pas celle de l'espace de noms du parent. Depuis la version 4.6 du noyau linux, il existe un nouvel espace de noms lié au cgroups pour cacher à un processus le cgroup des autres processus. Un processus verra le chemin d'autre processus relativement au chemin du cgroup de ce dernier et défini à la création, cachant ainsi la vrai position et identité de son cgroup. Tous les containers partagent le noyau du système d'exploitation sur lequel ils s'exécutent. Ils ont aussi la possibilité de partager des utilitaires ou des logiciel présent sur leur système hôte. Ceci est déjà un pas en avant par rapport aux machine virtuelle classiques ou chaque image contiendra une copie de tout. La taille d'un container est donc bien moindre puisqu'il peut compter sur son hôte. L'isolation apportée par les cgroups fait qu'on peut lancer plusieurs containers sur le même hôte qui partageront les outils de l'hôte.

D'autre part, les containers reposent sur les processus de support de leur hôte. Un containers ne va exécuter que l'application pour laquelle il a été lancer alors qu'une machine virtuelle complète aura un nombre significatif de processus qui s'exécutent, lancés pour la plupart pendant le démarrage

pour assurer les services disponibles à l'intérieur du système d'exploitation. Un ensemble de containers s'exécutant sur un hôte consommeront beaucoup moins de ressources qu'un ensemble équivalent de machines virtuelles complètes. Le fait, pour un container, de pouvoir utiliser les services du noyau de son hôte fait que son démarrage sera beaucoup plus rapide.

Une taille réduite et un démarrage rapide explique le succès rencontré par les containers ces derniers temps. Cependant, ce n'est pas encore optimal. Les images restent conséquentes, il est par exemple difficile d'envoyer ses images en utilisant des connexions distantes. On peut bien sur utiliser des petites distributions pour construire son image telles que TinyCore mais ce n'est pas encore une pratique répendue (par exemple la base recommandée pour les images Docker est Debian).
A cela s'ajoute un autre aspect négatif. Il y a toujours un problème de sécurité. L'isolation des containers n'est qu'une isolation fonctionnelle pour faire en sorte que sont exécution soit indépendante et qu'il ne soit pas perturbé par tout ce qui ce passe sur son hôte. Mais, comme dit précédemment, tous les containers partagent le noyau de leur hôte, ce qui fait que leur isolation se trouve au niveau applicatif qui est au sommet de la pile d'exécution, le noyau a accès à tous les containers . De plus, les brèches de sécurités sont toujours présentes. On retrouve dans les images des containers des binaires exécutables ou des exécutables de shell qui peuvent être accessibles au travers de failles. Cela peut même devenir plus problématique car quelqu'un qui arriverait à accéder à un container et à compromettre l'hôte (par exemple crash) sur lequel il se trouve pourrait compromettre tous les autres containers présents sur cet hôte.

## 2.4 Les unikernels

Les unikernels sont des images spécialisées immuables. Autrement dit toutes modifications voulant être apportées à un unikernel après sa compilation est impossible. Cela demandera la recompilation et la formation d'une nouvelle image. Il est impossible de faire une quelconque mise à jour ou de faire des changements de configuration pendant la vie d'une instance d'un unikernel. De plus, l' image obtenue réalisera uniquement la tâche pour laquelle elle a été prévue et rien de plus. Il est impossible comme sur un système plus classique de lancer plusieurs applications, de rajouter des modules ect...

Un unikernel est construit de telle sorte que le noyau et l'application qui formeront l'image sont compilés ensembles. Plus exactement, noyau, applications et librairies sont compilés ensembles en une seule image qui sera exécutée comme un seul programme et donc comme un seul processus. Il n'y a qu'un seul fil d'exécution dans les unikernels, il est donc impossible d'avoir accès à des appels comme *fork* ou *exec*; seul des threads sont utilisables.

Il en résulte qu'il n'existe qu'un seul espace d'adresses au sein d'un unikernel. Cela apporte un gain de performance par rapport aux machines virtuelles classiques et aux containers. En effet, un seul espace d'adresse implique qu'il n'y a pas de séparation entre un espace utilisateur et un espace noyau. Ce qui fait que dans un unikernel, il n'y a pas de changement de contexte ni d'appels

système coûteux. Si un unikernel s'exécute sur un hyperviseur, il y aura bien évidement des hypercalls vers celui-ci qui sont le pendant des appels système des hyperviseurs. Nonobstant, une machine virtuelle Linux qui s'exécuterait qur un hyperviseur aurait aussi besoin de ces hypercalls en plus des appels système qui continueraient de se faire au sein du système d'exploitation virtualisé.

Cette spécialisation des unikernels fait qu'ils ne doivent contenir que ce qu'ils vont utiliser. C'est pourquoi, la plupart des unikernels proposent un système de compilation qui permet au programmeur de choisir les fonctions bas niveau, habituellement fournies par l'espace noyau, qui seront présentes dans l'image finale. Ces fonctions seront ensuite directement intégrées à l'exécutable via un "système d'exploitation librairie" qui est une collection de librairies apportant les fonctions requises par un système dans un format compilable.

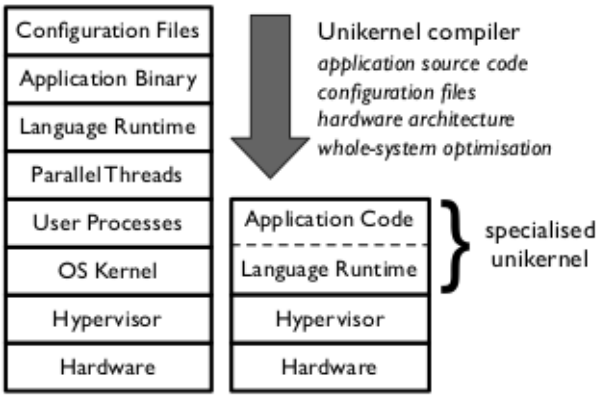


Figure 2: Comparaison de l'architecture verticale d'une VM classique et d'un unikernel.

La figure 2 présente un comparatif entre l'architecture verticale qui va être parcourue lors de l'exécution d'une application dans une machine virtuelle ou un unikernel. On remarque bien qu'un unikernel est beaucoup plus compact qu'une machine virtuelle ou un container avec une pile logicielle traditionnelle. Tout est réduit au simple minimum. On retrouve tout en haut le code de l'application et entre l'hyperviseur et cette application se trouve le "langage runtime" qui va servir de lien entre les deux. Cela permet un gain de performance et de sécurité.

L'isolation au niveau des unikernels est plus forte que pour les containers. Elle est faite à un niveau plus bas : au niveau de l'hyperviseur. Alors que chaque container partage le noyau de son hôte et s'exécute par dessus, chaque unikernel à son propre noyau et s'exécute sur l'hyperviseur.

En ce qui concerne la taille de l'image et la rapidité de boot, les unikernels se distinguent largement. Beaucoup d'unikernels ont une taille d'image inférieure au méga byte et ne font que quelques centaines de kilo octets. Il serait donc possible de lancer des centaines d'instance sur une même machine sans perte de performance sur l'exécution d'une instance. De plus, du fait de leur petite

taille et de leurs outils embarqués restreints au stricte minimum, un unikernel ne va exécuter au démarrage que les fonctions nécessaires au fonctionnement de son application. Ainsi, le temps de démarrage se compte en millisecondes et cela permet d'envisager le déploiement sur le cloud de service à la volée. On pourrait envisager de lancer un unikernel que lorsque c'est nécessaire puis de le stopper lorsque qu'il a fini son travail.

La principale force des unikernels est qu'ils permettent de réduire la surface d'attaque et la proportions des ressources utilisées dans les services proposés à travers le cloud. Ils offrent à la fois une isolation d'un point de vue performance (pas d'appels système, même s'il y a des hypercalls) et d'un point de vue sécurité.

La sécurité est inhérente à la construction d'un unikernel, il n'y a pas d'effort supplémentaire à fournir pour la mettre en place . Sa spécialisation fait qu'il n'emporte pas les outils traditionnellement utilisés pour attaquer des systèmes :

- Les unikernels n'intègrent pas de terminal donc il n'y a pas de ligne de commande.
- Il n'y a pas d'utilitaires dont on pourrait prendre le contrôle.
- Il n'y a pas de pilotes ou de librairies non utilisés à attaquer.
- On ne trouve pas de fichier de mots de passe ou quelconques informations d'identification (plus généralement le concept d'utilisateurs n'existe pas dans les unikernels).

Ce n'est bien sur pas une solution miracle non plus garantissant 100% de sécurité. Mais le degré de sécurité apporté est supérieur et ne demande pas de contraintes supplémentaires. Il ne faut pas non plus confondre la sécurité intrinsèque d'une solution et les actions de l'utilisateur. Un utiisateur à toujours le moyen de laisser ou des créer des portes dérobées malvenues.

Il faut bien noter que les unikernels ne sont pas voués à remplacer les machines virtuelles ou les containers. Ils sont plus intéressant pour certaines classes d'application mais ne sont pas la réponse à tout. C'est même d'ailleurs le contraire, le futur pourrait voir une cohabitation des unikernels et des containers. En effet, Docker s'interesse de prêt aux unikernel, ils ont racheté début 2016 la société Unikernel Systems et ils souhaitent associer unikernels et containers.

## 2.5 Classification et présentation de certains unikernels

Les unikernels sont actuellement en plein essor, c'est pourquoi on trouve de nombreux unikernels. Ils tentent parfois de répondre à des problèmes spécifiques ou essayent d'apporter des solutions plus larges, pour un plus grand nombre d'utilisations. De nouveaux unikernels continuent d'apparaître encore assez fréquemment.

Le premier critère de classification qui apparaît est le(s) langage(s) que l'unikernel va proposer et dans le(s)quel(s) il faudra programmer ses applications si on veut les faire fonctionner sur cet

unikernel. Cela peut déjà apparaître comme une première restriction majeure dans l'utilisation ou non d'un unikernel donné. En effet si on veut simplement utiliser une application déjà existante et l'intégrer dans un unikernel, il faudra se tourner vers un unikernel supportant le langage dans lequel est écrit l'application où alors il faudra porter cette application en la recodant dans un autre langage. Aujourd'hui, la plupart des langages de haut niveau utilisés ont au moins un unikernel permettant leur utilisation. Mais le langage de programmation est aussi important du fait du concept "de système d'exploitation librairie". La taille de l'image, les dépendances à la compilation et la sécurité à l'exécution dépendent tous en partie du langage sous jacent.

Un deuxième critère de classification est le support sur lequel va s'exécuter un unikernel. Chaque unikernel va fournir la possibilité de s'exécuter sur un ou plusieurs hyperviseurs ou encore la possibilité de s'exécuter directement sur le matériel (bare metal). Les différents supports d'exécution peuvent aussi s'avérer être un frein et cela a notamment été le cas au cours de mon stage. Cependant j'y reviendrai un peu plus loin dans mon rapport.

Au cours de ce stage, nous nous sommes intéressé à quatre unikernels en particulier qui sont : MirageOS, ClickOS, Rumprun et OSv. Pour chacun de ses unikernels, nous nous sommes documenté sur leurs utilisation, leur fonctionnement, leurs performances et leurs dépendances. Nous avons aussi essayé de voir comment fonctionnait les trois premiers unikernels en construisant une image avec une application la plus simple possible et de les exécuter sur Xen. Le but était simplement d'avoir une première prise en main des différents unikernels avec leurs outils et de tester les difficultés à construire une image. Cela n'a pas été effectué pour OSvs car nous avons rapidement abandonné cet unikernel puisqu'il ne correspondait pas aux utilisations que nous voulions, c'est à dire principalement pour des NFVs.
La plupart des unikernels rencontrés ne s'exécutent que sur un seul CPU virtuel. Cependant ce CPU virtuel peut être lié à plusieurs CPU physique. Mais si la performance rentre en jeu, le mieux est d'avoir un seul CPU virtuel lié à un seul CPU physique: cela permet d'optimiser les accès aux caches hardware.

### 2.5.1 MirageOS

MirageOS est un unikernel basé le langage Ocaml. Ce langage a été choisi car les développeurs considèrent que l'Ocaml est un langage d'avenir pour ce genre d'application et qu'en plus de la sécurité intrinsèque des unikernels présentée auparavant, le langage apportait une sécurité supplémentaire par le biais de son typage statique fort. Ce dernier permet de garantir à la compilation que le code compilé ne produira pas certains types d'erreurs de ségmentation ou d'erreurs à l'exécution qui sont difficiles à détecter sans typage fort et qui passeraient la compilation dans d'autres langages (e.g C). Cela fourni une assurance supplémentaire sur l'image obtenue. MirageOS utilise comme base le noyau mini-os qui est un noyau open source fourni avec les sources de Xen. Ce noyau mini-os ce trouvera dans la partie Mirage runtime que l'on peut voir sur la figure ci-dessous et c'est lui qui servira de glue entre l'hyperviseur sous-jacent et l'application. C'est notamment lui qui fera

les hypercalls et qui fourni les librairies dont à besoin l'application. Cependant, il a été réécris en ocaml pour pouvoir être inclus dans MirageOS. La concurrence est possible sur mirageOS et tout est basé sur les événements, il n'y a aucun support pour un ordonnancement préemptif : le model d'ordonnancement suit le principe de "run to completion". De plus, chaque instance de mirageOS s'exécute au plus sur un seul CPU virtuel. Les unikernels mirageOS peuvent s'exécuter sur Xen, KVM ou sur Linux comme un fichier binaire.

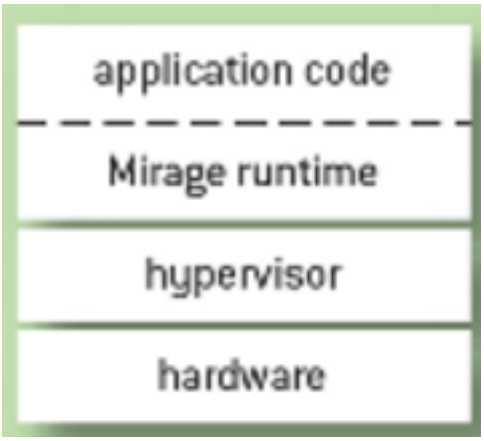


Figure 3: Architecture de MirageOS.

### 2.5.2 Rumprun

Rumprun est un unikernel qui a pour base un noyau rump. C'est un noyau dérivé de netBSD. Le but de Rumprun n'est pas d'offrir un support pour un langage particulier mais de permettre de mettre sous forme d'unikernel n'importe quel programme qui respecte la norme POSIX et de réutiliser le plus de code déjà existant possible (une grande partie du code du noyau est du code netBSD non modifié). Il est donc en théorie possible de compiler la plupart des programmes fonctionnant sur des systèmes d'exploitation de type Linux en un unikernel rumprun. Tout comme mirageOS chaque instance rumprun utilise un seul CPU virtuel. De plus, il n'y a pas de scheduler dans rumprun, il utilise la politique d'ordonnancement de la plateforme sur laquelle il s'exécute. C'est pourquoi, lorsqu'un unikernel Rumprun s'exécute sur un hyperviseur, les mécanismes de synchronisation passent par des hypercalls à l'hyperviseur car ils demandent l'accès à un ordonnanceur. La politique concernant la mémoire virtuelle est la même et Rumprun est transparent concerant la mémoire virtuelle: il s'exécute dans l'espace mémoire fournie par la plateforme sur laquelle il se trouve, que cette mémoire soit virtuelle ou pas. Cet unikernel s'exécute soit directement sur le matériel, soit sur Xen ou soit en mode utilisateur sur Linux.

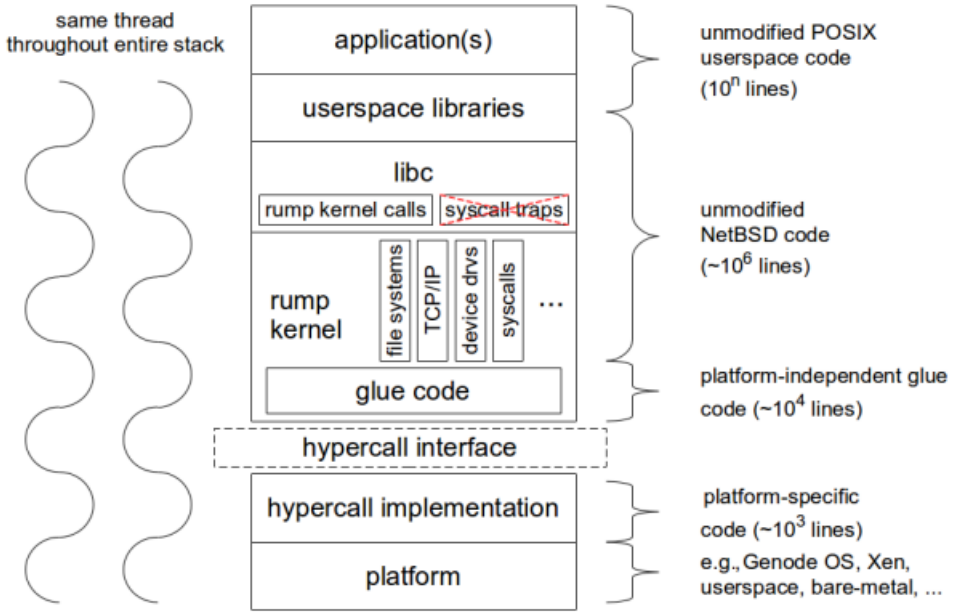


Figure 4: Architecture de Rumprun.

### 2.5.3 ClickOS

ClickOS est un unikernel, comme MirageOS, qui utilise un mini-os modifié. La particularité de ClickOS vient du fait qu'il ne permet pas d'exécuter n'importe quelles applications mais qu'il repose sur Click modular router qui est à la base une application fonctionnant sur Linux soit en mode utilisateur soit en temps que module. Cette application permet d'implémenter bon nombre de fonctions réseau au travers d'un fichier de configuration écrit dans un pseudo langage de scipt spécifique. Click modular router est codé en C++ et est extensible par n'importe quel utilisateur si des outils lui manque dans la version de base. J'expliquerai plus en détail ce qu'est Click modular router plus loin dans le rapport. Le côté applicatif dans l'unikernel sera donc apporté par le fichier de configuration et la runtime library correspondra au fonctionnement interne de Click modular router. Etant donné que click modular router nécessite un fichier de configuration, une instance de clickOS en a aussi besoin. Cependant, il n'y a pas de système de fichier dans clickOS il est donc impossible d'accéder à un fichier sur le disque depuis une instance. Pour cela, clickOS apporte son propre outil : Cosmos. Ce dernier se lance depuis le domain principal de Xen (qui est un système d'exploitation de type Unix), il va accéder au fichier de configuration puis envoyer son contenu à l'instance de ClickOS souhaitée via un mécanisme de Xen. Tout comme les unikernels précédant, chaque instance de ClickOS utilise un seul CPU virtuel. ClickOS est utilisable uniquement sur Xen.

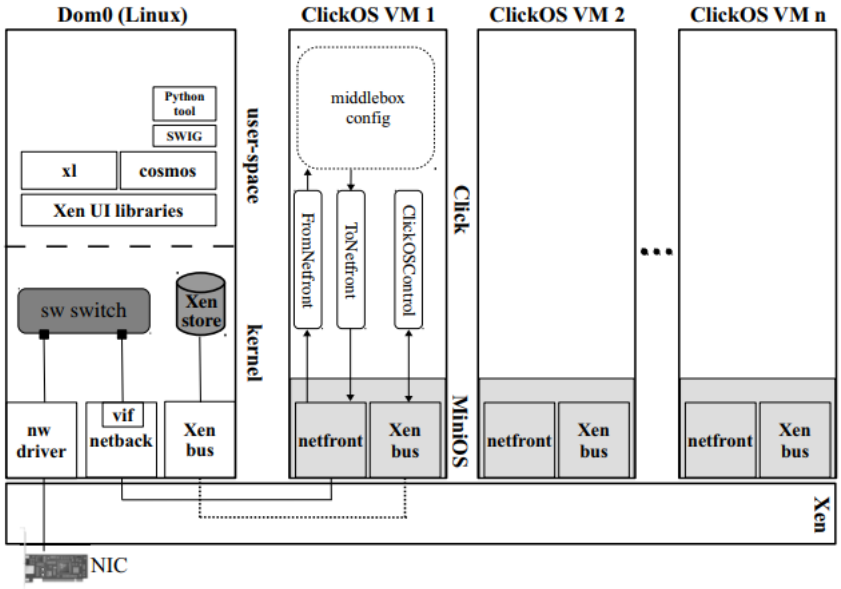


Figure 5: Architecture de clickOS sur l'hyperviseur Xen.

### 2.5.4 OSv

OSv est à la base un unikernel créé pour exécuter des machines virtuelles Java grâce à l'utilisation d'un fichier WAR. Cependant, son architecture lui permet en réalité d'exécuter quasiment n'importe quelle application mono processus. De fait, beaucoup de langages peuvent être utilisés comme le python, le ruby, le pearl, le C, le C++ ect... Le but de cet unikernel et de pouvoir exécuter n'importe quelle application qui pourrait s'exécuter de manière mono processus (même si à la base l'application ne l'est pas et qu'il faille réécrire le code). C'est pourquoi les unikernels OVs ont la particularité d'être des unikernels moins légers et ont plus souvent une taille se comptant en méga-octets qu'en kilo-octets. OSv, contrairement à Rumprun, utilise un mécanisme de mémoire virtuelle et apporte son propre ordonnanceur pour les thread. Ce dernier est lock-free et préemptif. OSv a vocation à être utilisé pour des applications qui doivent être déployées sur le cloud, c'est pourquoi il fournit une pile TCP/IP de base. Cet unikernel est un de ceux qui supporte le plus d'hyperviseur : Xen, VirtualBox, VMware et KVM.

### 2.5.5 Conclusion

Après étude de ces 4 unikernels nous avons décidé de retenir ClickOS. De par son but affiché d'être utilisé pour mettre des fonctions réseaux sous la forme d'unikernels, il était naturellement le plus approprié. Nous avons décidé de ne pas poursuivre avec mirageOS à cause du langage ocaml. En effet MirageOS fait appel à une extension du langage ocaml et personne dans l'équipe n'était familier avec. Nous n'avons pas totalement abandonné Rumprun. Je ne m'en suis pas servi pendant mon stage par la suite mais mon costagiaire a utilisé Rumprun car il n'avait pas

besoin des applications spécifiques de ClickOS mais simplement de pouvoir tester son travail avec les unikernels. Cela était donc plus pratique pour lui de travailler avec Rumprun. De plus, le fait que Rumprun puisse être utilisé avec différents langages et puisse fonctionner avec des applications qui respectent la norme POSIX reste quand même très intéressant.
Ci-dessous se trouve un tableau récapitulatif des avantages et inconvénients des unikernels par rapport au machines virtuelles et aux containers:

| Avantages | Inconvénients |
|---|---|
| • temps de démarrage rapide (quelques millisecondes)<br>• taille d'image réduite (quelques centaines de kilo-octets)<br>• unique espace d'adresse, pas de distinction noyau/utilisateur<br>• spécialisés et donc surface d'attaque réduite<br>• isolation, chaque unikernel à son propre noyau | • toutes modifications doivent être faites statiquement et suivies d'une recompilation: il faut couper le service et le redéployer pour mettre à jour<br>• certains unikernels utilisent un langage spécifique: il faut écrire toutes les applications dans ce langage ce qui limite la fléxibilité<br>• ne sont pas adaptés à toutes les classes d'application<br>• la plupart ont une contrainte liée aux choix possibles pour la plateforme d'exécution sous-jacente qui imposent des restrictions |

# 3 Projets utilisés et travaux existants[2]

Le but du stage était l'étude de la performance des unikernels et notamment des performances au niveau réseau. Une fois le choix de Click0S effectué, deux axes se sont présentés :

- Comment une instance de ClickOS pourrait-elle accéder à une carte réseau de manière optimale? C'est à dire récupérer les paquets dans un unikernel depuis la carte sans copie ou encore utiliser un mode polling sur la carte pour gérer la notification de réception de paquets plutôt que le système d'interruption utilisé par défaut?
- Comment avoir un mécanisme de communication inter unikernel optimisé (sans copie à chaque passage d'un unikernel à l'autre et lock-free) sans passer par le réseaux pour des unikernels colocalisés sur la même machine?

Pour travailler autour de ses deux problématiques, je me suis appuyé sur des logiciels et technologies déjà existants, soit en les utilisant, soit en m'en inspirant pour mon travail. Dans cette partie, je vais présenter les principales technologies auxquelles j'ai été confrontées pour une meilleure compréhension lors de l'explication de mon travail.

## 3.1 Data Plane Development Kit (DPDK)

DPDK offre un ensemble de librairies pour accélérer le traitement des paquets, il permet, entre autre, de recevoir et d'envoyer des paquets en un minimum de cycle CPU (moins de 80). Ces librairies sont spécifiques à du matériel mais la liste des cartes réseau supportées est relativement longue et s'allonge régulièrement. De plus, DPDK est utilisable avec n'importe quel CPU. Pour utiliser DPDK dans une application, il suffit de compiler DPDK pour générer une librairie puis de compiler son application en incluant la librairie obtenue.

Sur les systèmes d'exploitation classiques, lors de la réception d'un paquet, il y a une première copie pour passer le paquet du buffer de la carte réseau au noyau et une deuxième copie pour passer du noyau à l'espace l'utilisateur. Le plus souvent, la notification de la réception de paquets par la carte se fait via un mécanisme d'interruption. Les accélérateurs de traitement de paquets comme DPDK offre des librairies pour effectuer un polling sur la carte réseau et récupérer les paquets directement depuis la carte réseau sans copie grâce à de la mémoire partagée.

Pour cela DPDK a besoin de prendre le contrôle sur la carte réseau. Il faut donc détacher les interfaces que l'on veut utiliser de Linux. Les paquets seront donc entièrement gérer par DPDK et ne passeront plus par Linux. Pour ce faire, il suffit de charger les drivers réseau que l'on veut utiliser pour DPDK (avec la commande modprobe par exemple) puis d'utiliser une commande fournie par DPDK (via un script python). Cette commande permet aussi de faire le chemin inverse et de

[2]Références: [10, 11, 12, 13, 14, 15, 16, 17]

remettre une interface sur Linux.

DPDK offre aussi une librairie pour gérer des grosses pages de mémoire (hugepages). Ces grosses pages de mémoire sont nécessaires au fonctionnement de DPDK qui requiert la réservation d'un certain nombre de pages par son système hôte. Le mécanisme de hugepages (page de mémoire physique de taille supérieure 4k) est présent sur Linux. Selon la taille voulue pour ces pages, il faut les allouées soit au moment du boot soit le plus rapidement possible après le boot. Cela est du au fait que plus on tarde à les allouées, plus ces pages seront fragmentées. La plupart du temps, sur les CPU intel, les tailles possibles pour ses pages est 2Mo ou 1Go (les tailles de pages supportées dépendent du CPU). On ne peut pas allouer des pages d'1G après le boot mais cela est possible pour les pages de 2M via le fichier */sys/kernel/hugepages/hugepages-2048kb/nr_hugepages*. Une fois la mémoire allouée, il faut la rendre accessible à DPDK via un point de montage. Ces pages seront utilisées par DPDK comme un pool de mémoire pour les paquets qu'il va traiter.

L'existence de telles pages est un gain de performance pour les processus qui utilisent beaucoup de mémoire. Traditionnellement, la RAM est découpée en page de 4096 octets et lorsqu'un processus utilise de la RAM, il lui est attribué un certain nombre de pages mémoires selon ses besoins. Pour des applications demandant une consommation de mémoire importante, un découpage en page de 4ko est préjudiciable. En effet, la mémoire virtuelle pour les processus étant de plus en plus courante, le CPU et le système d'exploitation doivent garder une trace de quelle page appartient à quel processus et où cette page se trouve. Et évidemment, plus un processus aura de pages plus ce sera coûteux de chercher où cette mémoire est mappée.

## 3.2 Netmap

Netmap est un framework qui, comme DPDK, a pour but la rapidité du traitement des paquets et qui est utilisable en mode utilisateur ou comme module noyau. Netmap inclus VALE, un switch logiciel sous forme de module noyau, et les netmap pipes, qui sont des canaux de transport de paquets utilisant de la mémoire partagée. Tout est utilisable avec la même API. Un port réseau (interface), virtuel ou non, passé en mode netmap permet un trafic rapide des paquets entre processus, marchines virtuelles, cartes réseau ou la pile réseau de l'hôte. Pour cela, il suffit de connecter chacun d'eux à un port virtuel lié à un port physique grâce au switch VALE. Netmap supporte à la fois des entrées/sorties non bloquantes et des entrées/sorties et synchronisation bloquantes.

La figure 6 présente l'architecture de Netmap. Le traitement des paquets ce fait à travers des ports, physiques d'une carte réseau ou virtuels de VALE. Les ports ont à leur disposition des buffers en anneaux pré-alloués d'une region mémoire mappée par Netmap. Chaque port possède un anneau pour sa file de transmission de paquets (Tx) et pour sa file de réception de paquets (Rx) et il y a une paire d'anneau supplémentaire qui est reliée à la pile réseau de l'hôte. Toutes les cartes réseau en mode Netmap utilisent la même région de mémoire.

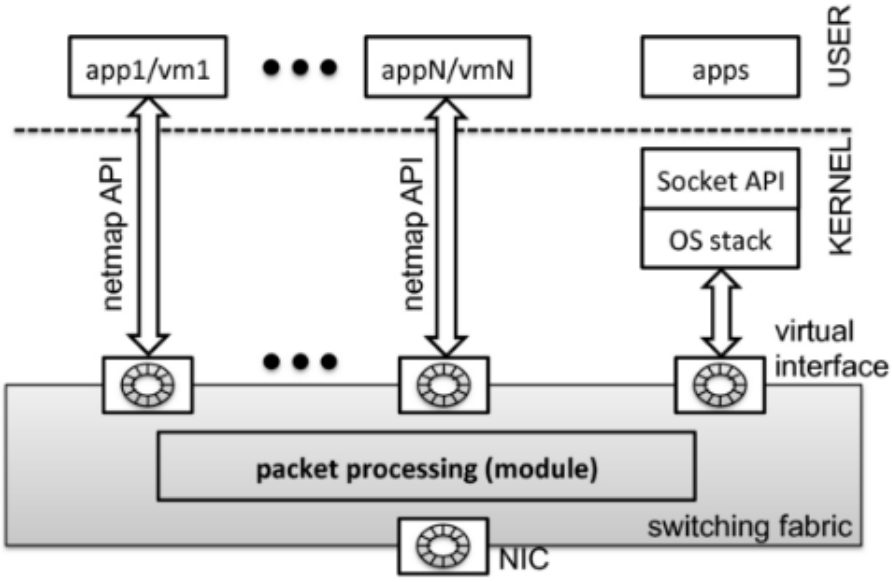


Figure 6: Architecture de Netmap.

## 3.3 Click modular router

Click est un logiciel qui permet de construire des routeurs flexibles et configurables. Aujourd'hui, on attend d'un routeur qu'il puisse faire plus que de simplement router des paquets. Click offre la possibilité de créer des routeurs aux fonctions diverses et interchangeables facilement. Un routeur Click est constitué à partir de modules servants au traitement des paquets : les éléments. Chaque élément implémente une action simple qui se fera sur les paquets qui le traverseront. Cela peut aller de la classification de paquets, en passant par l'ordonnancement, la mise en file d'attente ou encore faire office d'interface avec le hardware réseau. Une configuration d'un routeur Click peut être vu comme un graphe orienté avec pour sommets les éléments et où les paquets vont passer d'élément en élément en suivant les arêtes. On peut facilement modifier un routeur Click en rajoutant ou enlevant des éléments qui sont des briques plus ou moins indépendantes.

Les éléments sont donc les unités de base d'un routeur Click et réalisent des actions très simples. La connection de deux éléments représente une arête possible pour le parcours des paquets au sein du routeur. Toutes les actions effectuées par un routeur sont encapsulées dans un élément. Le choix des éléments et leurs interconnexions se font lors de la configuration. Chaque élément est un objet C++, chaque connexion est représentée comme un pointeur vers des éléments et l'envoie d'un paquet le long d'un connexion entre deux éléments se fait par un appel à une fonction virtuelle (au sens de la programmation objet). Les propriétés les plus importantes d'un élements sont :

- La classe de l'élément. Chaque élément appartient à une unique classe dérivée de la classe mère de tous les éléments: *Element*. Cela spécifie le code qui sera exécuté quand l'élément

traitera un paquet et permet l'initialisation et la configuration de l'élément.

- Les ports de l'élément. Un élément peut posséder plusieurs ports d'entrée et plusieurs ports de sortie. Le parcours des paquets le long d'une connexion peut être initié soit par la source (push), soit par la destination (pull). C'est pourquoi les ports d'un élément click suivent ce principe et peuvent être répartis en trois catégories : les ports push, pull et agnostiques. Un port push (respectivement pull) ne peut être connecté qu'à un port push (respectivement pull). Les connections entre un port push et un port pull sont invalides. Les ports agnostiques deviennent push (respectivement pull) s'ils sont connectés à un port push (respectivement pull). Deux ports agnostiques ne peuvent pas être connectés. Lors de l'initialisation d'un routeur, les contraintes sont propagées jusqu'à ce que chaque port agnostiques soit assigné pull ou push.
- Le string de configuration de l'élément. Dans le fichier de configuration, chaque élément peut prendre un string en argument qui correspondra aux entrées qui seront passées en arguments à cet élément lors de l'initialisation du routeur. Un élément peut avoir des arguments obligatoires et/ou facultatifs ou pas d'arguments.
- Les handlers d'un élément. Chaque éléments peut posséder ou non des handlers. Ce sont des variables internes à un élément qui gardent un état d'une caractéristique/propriété pendant la vie d'un élément. Par exemple cela peut être un compteur pour savoir le nombre de paquets passés dans un élément. Les handlers sont accessibles (en lecture et/ou en écriture) pendant l'exécution d'un routeur via des lignes de commande et permettent soit de récupérer une information ou soit de changer la valeur d'un attribut d'un élément dynamiquement.

La figure 7 montre le code d'une configuration Click simple qui ne fait que compter les paquets reçus puis les jette.

```
// Declare three elements ...
src :: FromDevice(eth0);
ctr :: Counter;
sink :: Discard;
// ... and connect them together
src -> ctr;
ctr -> sink;

// Alternate definition using syntactic sugar
FromDevice(eth0) -> Counter -> Discard;
```

Figure 7: Configuration d'un routeur Click modular router.

## 3.4 L'hyperviseur Xen

Xen est un hyperviseur open source de type I qui rend possible l'exécution en parallèle d'un grand nombre d'instance de machines virtuelles sur une seule machine. Xen s'exécute directement sur le matériel et sera le responsable du management du CPU, de la mémoire et des interruptions. C'est le premier programme qui s'exécute après la sortie du bootloader. L'hyperviseur en lui même n'a pas connaissance de tout ce qui est réseau ou stockage.

Les machines virtuelles s'exécutant sur Xen sont appelées domaines et sont réparties en deux groupes : le domaine principal appelé domaine 0 et les domaines invités. Chaque domaine à un identifiant entier unique, le domaine principal ayant toujours le numéro 0: c'est un domaine spécial, il est privilégié. Il sera le premier lancé après Xen et s'exécutera tout le temps que Xen s'exécute, ils sont indissociables. Les systèmes d'exploitation supportés par Xen comme domaine principal sont Linux, NetBSD et OpenSolaris. Ce domaine comporte les outils pour gérer Xen, les domaines invités (les lancer, les configurer,, y accéder via console ect...) et les pilotes pour le matériel. Il est privilégié et a donc l'accès direct au matériel, aux entrées/sorties et peut interagir avec les domaines invités. Chaque domaine invité aura ses propres pilotes front-end qui seront lié aux pilotes back-end de Xen présents de le domaine 0. Ces domaines invités n'ont aucun privilège d'écriture ou de lecture sur le matériel, c'est pourquoi ils sont aussi appelés domaines non privilégiés (domU: unpriviledged domain).

Xen autorise aussi une isolation des pilotes. Il est possible de dissocier les pilotes du domaine 0 et de les mettre dans un domaine à part qui n'aura que pour fonction de fournir les pilotes back-end. Cela permet, en cas de crash ou de compromission des pilotes de simplement reboot la machine virtuelle les contenant et de relancer les pilotes sans affecter tout le reste du système.

Xen est un hyperviseur optimisé pour exécuter des machines virtuelles dites para-virtualisées (PV). C'est à dire que Xen peut s'exécuter sur des CPUs qui n'ont pas d'extension de virtualisation et que les noyaux des machines virtuelles ont été modifiés pour pouvoir s'exécuter sur Xen (notamment pour intégrer les hypercalls vers Xen dans leurs appels système) sans avoir besoin d'émulation virtuelle du matériel. Cependant, Xen supporte aussi les machines complètement virtualisées (HVM) qui requièrent une émulation du matériel pour pouvoir s'exécuter et qui n'ont pas un noyau modifié. Du fait de cet émulation, ces machines virtuelles sont moins performantes. Les distributions Linux, NetBSD et OpenSolaris contiennent dans leur noyau les modifications nécessaires mais pas Windows qui devra être exécuté comme une HVM.
Je vais maintenant présenter trois mécanismes du fonctionnement interne de Xen qui se rapportent à la communication inter domaines et qui m'ont été indispensables pendant mon stage : le mécanisme des *grant tables*, des *event channels* et du *Xenstore*.

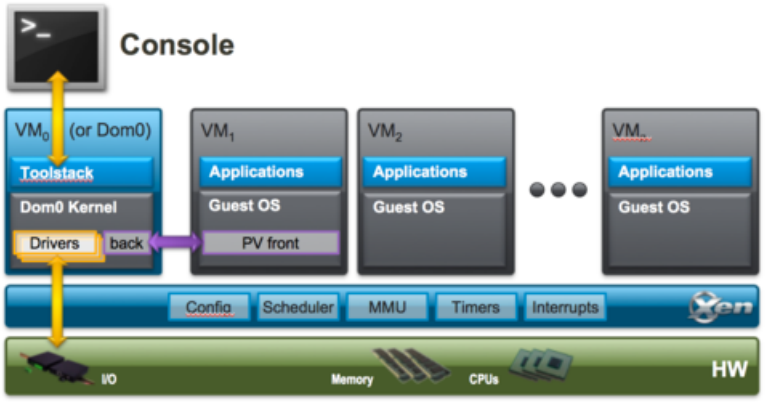


Figure 8: Architecture de Xen.

### 3.4.1 Grant tables

Il existe dans Xen un mécanisme permettant le partage ou le transfère de pages en mémoire entre domaines de tous types (privilégié ou non). Il permet de donner des droits d'accès en lecture et/ou écriture sur des pages allouées par un domaine ou alors de transférer la possession d'une page d'un domaine à un autre.

Chaque domaine possède sa table de droits pour ses pages en mémoire initialisées à la construction du domaine. Elle est implémentée comme une structure de donnée partagée avec Xen. Cela permet au domaine de directement partager avec Xen les droits qu'ont les autres domaines sur ses pages. Chaque entrée dans la table est référencée par un entier qui sert d'index dans la table. Grâce à cet entier et l'identifiant du domaine possédant la page, un autre domaine pourra demander l'accès ou le transfert de la page à Xen qui ira vérifier dans la table du domaine d'où viennent les droits si cela lui est effectivement permis ou non.

De plus, la référence d'un droit encapsule les détails liés à la page partagée. Ainsi, les domaines n'ont pas besoin de connaître les adresses physiques des pages à partager ou à transférer, rendant possible le partage de mémoire entre domaines qui s'exécutent dans un environnement où ils n'ont accès qu'à de la mémoire virtuelle.

### 3.4.2 Event channels

Ce mécanisme est le mécanisme asynchrone de base pour la notification d'évènements dans Xen. Il est notamment utilisé pour la notification entre le domaine possédant les pilotes matériels back-end et les autres domaines pour le bon fonctionnement des pilotes. Ces canaux de notification sont l'équivalent Xen des interruptions matériels.

Un canal est représenté par un port dans chacun des deux domaines partageant ce canal. C'est Xen qui garde les informations permettant d'identifier un canal entre deux domaines. Chaque canal correspond à un bit qui sera positionné à 0 ou 1 pour signifier une notification sur le canal à un domaine. Les notifications sont reçues par un domaine via un upcall de la part de Xen. Les notification sur ce canal seront masquées tant que le bit n'aura pas été réinitialisé. C'est pourquoi un domaine devra tester de nouveau la valeur du bit après avoir démasqué l'évènement pour être sur de n'avoir manqué aucune notification car Xen mettra le bit à jour pour dire qu'une (ou plusieurs) notification ont été reçu pendant le masquage mais le masquage aura empêché tout upcall. Si plusieurs notifications sont reçues pendant le masquage du canal, le domaine n'en recevra qu'une seule au démasquage, les autres seront perdues. Dans chaque domaine, les évènements sont gardés dans une bitmap partagée entre le domaine et Xen.

Quatre type d'évènement sont gérés sous forme de canal de notification dans Xen :

- les notifications inter domaines (notamment pilotes)
- les VIRQs (virtual interrupt requests)
- IPIs (inter-processor interrupt)
- PIRQs (physical interrupt requests)

Un domaine possède deux moyens de recevoir des notifications via un canal. Ce choix se fait lors de la création du canal qui prend en argument un pointeur de fonction vers un handler. Si le handler n'est pas null, lors de la réception d'un évènement un callback de Xen va exécuter ce handler. Cela fonctionne comme les handlers de signaux sur Linux. Il est aussi possible de mettre ce handler à null et de bloquer sur l'attente d'une notification via un polling sur le port du canal. Le polling peut se faire sur plusieurs canaux à la fois. L'hypercall de polling retournera lorsque un ou plusieurs canaux auront reçu une notification et le domaine devra se charger de trouver lesquels auront reçu une notification en vérifiant le bit de chaque canal. Pour cela, il faut au préalable masquer les canaux que l'on souhaite poller.

### 3.4.3 Xenstore

Le Xenstore est une sorte de pseudo file système fourni par Xen pour permettre aux domaines et aux pilotes de communiquer des informations de configuration ou des statut. Il faut éviter de l'utiliser comme un mécanisme de transfert de grosses quantités de données. Chaque domaine possède son propre chemin dans le Xenstore (*/local/domain/id/data*) et il est possible de créer des chemins sous cette racine. Le Xenstore est une table partagée entre un domaine et Xen avec comme clés les chemins et comme valeurs les données que l'on souhaite stocker. N'importe quel domaine peut lire ou écrire où il le souhaite dans le Xenstore à condtion qu'il en ait les droits. De base, chaque domaine non privilégié à uniquement accès en lecture et écriture aux chemins sous sa racine */local/domain/id/data* peut donner les droits à d'autres domaines s'il le souhaite. Quant

au domaine 0, il à accès en lecture et écriture à tout le Xenstore. Il est naturellement possible de lire et d'écrire dans le Xenstore mais il est aussi possible de suivre les modifications apportées aux valeurs stockées dans un chemin en le demandant. Si une demande à été faite, toutes modifications de données sur ce chemin ou les chemins dont il est la racine seront notifiées via un callback de Xen.

La figure 9 ci dessous représente le fonctionnement au niveau réseau des machines virtuelles sur Xen en explicitant l'utilisation des trois mécanismes précédemment décris. C'est le domaine 0 qui reçoit tous les paquets et qui les dispatche dans leur domaine de destination en utilisant de la mémoire partagée et en les notifiant via un canal.

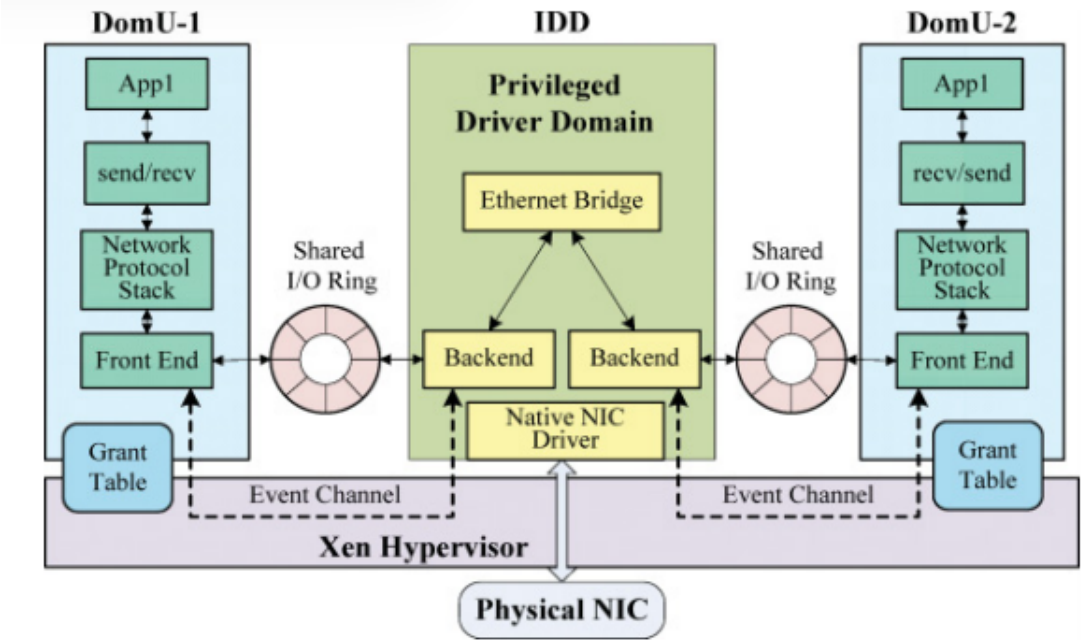


Figure 9: Architecture du fonctionnement réseau de machines virtuelles s'exécutant sur Xen.

## 3.5 libvchan

Libvchan est une API fournie dans les sources de Xen (tools) pour la communication inter domaines au travers de canaux de mémoire partagée. Cette librairie est complètement implémentée pour Linux mais pas pour mini-os qui est le noyau sur lequel j'ai travaillé pendant mon stage. Cependant, les mécanismes et principes utilisés par libvchan ont été une source pour mon travail. Cette librairie permet de créer deux buffers en anneau partagés par deux domaines : un buffer pour copier des données du domaine A au domaine B et inversement (chaque anneau est unidirectionnel). L'accès à ces buffers se fait de mainière lock-free par les domaines. La notification de lecture et d'écriture se fera via un event channel entre les deux domaines. Un des deux domaines sera à

l'initiative de la communication et sera considéré comme le serveur et l'autre domaine sera le client.

Le serveur va allouer les pages mémoires nécessaires pour les buffers, il va ensuite, via un hypercall, donner les droits en lecture et écriture sur ces pages au domaine avec qui il veut communiquer. En retour, il va recevoir un entier par page attestant de ces droits qu'il devra fournir à l'autre domaine. Le couple (serveur id, droit) est unique pour chaque page et sera utiliser par le client pour mapper chacune des pages via un hypercall. Pour transmettre ces droits, le serveur va les écrire dans le Xenstore à un chemin connu de lui et du client : */local/domain/serverID/data/clientID/ring* et il donnera les droits en lecture au client sur ce chemin. Le serveur va aussi créer, via un hypercall, un event channel pour les notifications. Il va recevoir un port qui lui servira à utiliser ce canal. Pour initialiser le canal, le client aura besoin du port du serveur. Ainsi, le serveur écrit ce port dans le Xenstore sur le chemin */local/domain/serverID/data/clientID/channel*.

De son côté, le client va récupérer, depuis le Xenstore, les droits et le port du serveur. Il va ensuite mapper les pages de mémoire partagée correspondant aux buffers et initialiser son côté du canal de notification pour recevoir un port attaché au port du serveur.

Une fois cela fait, les deux domaines peuvent communiquer. Il y a deux manières possibles de communiquer : bloquante ou non bloquante. Dans le cas non bloquant, un écrivain écrira ce qu'il peut écrire dans le buffer à chaque appel et le lecteur lira ce qu'il peut lire dans le buffer à chaque appel. Dans le cas bloquant, l'écrivain écrira ce qu'il peut écrire dans le buffer et s'il ne peux pas écrire tout ce qu'il veut, il mettra un entier à jour pour signifier au lecteur qu'il souhaite être notifié à la prochaine lecture. Il vérifiera ensuite un autre entier pour savoir si le lecteur souhaite être notifier. Si oui, il enverra une notification sur leur canal dédié. Et enfin, s'il n'a pas pu tout écrire, il bloquera en attendant une notification du lecteur. Le lecteur, lui, va lire dans le buffer tout ce qu'il peut lire (au plus la valeur souhaitée en argument). Il va ensuite vérifier l'entier permettant de savoir si l'écrivain souhaite une notification. Si c'est le cas, il lui envoie. Finalement, s'il a lu moins que ce qu'il souhaitait, il va mettre l'entier signifiant qu'il souhaite une notification à la prochaine écriture à jour puis il va bloquer en attendant d'être notifié. La schématisation de l'architecture de la librairie libvchan est présentée dans la figure ci dessous.

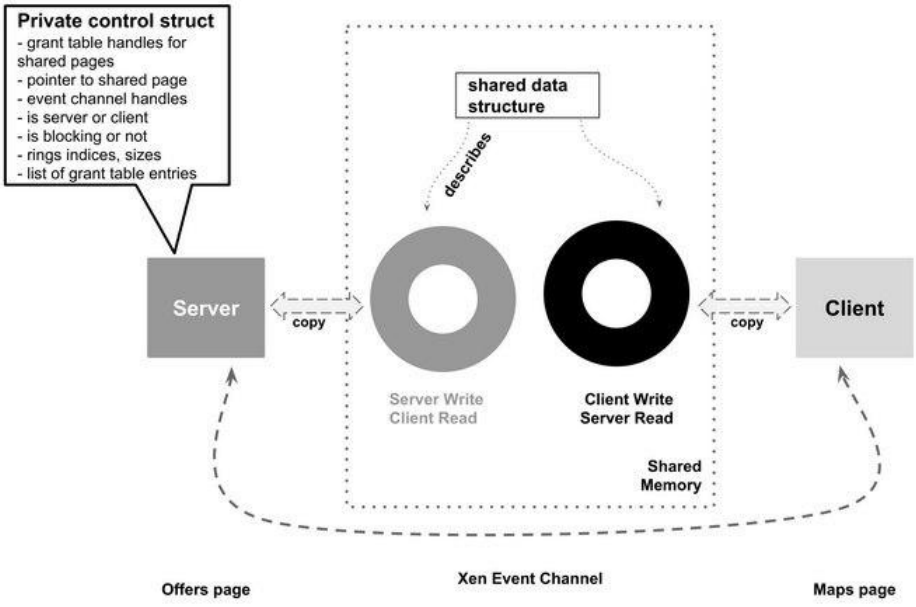


Figure 10: Architecture de la libvchan.

# 4 Tests de performance: click et DPDK

Dans cette partie, je vais présenter la première partie de mon travail qui a consisté a faire des tests de performance sur l'utilisation de click et de DPDK en mode utilisateur sur Linux. En effet, nous avons décidé, dans un premier temps, de nous pencher sur les performances de la réception de paquets dans un unikernel par une carte réseaux. Nous nous sommes intéressé à deux frameworks qui permettent d'améliorer les performances de réception de paquets : Netmap et DPDK.

Initialement, je me suis intéressé à Netmap car ce framework est compatible avec les pilotes front-end de clikcOS. Netmap va venir remplacer les pilotes back-end du domaine 0 (pour cela, le domaine 0 doit forcément être une distribution Linux). Ainsi, les pilotes front-end de clickOS vont venir se brancher sur les pilotes back-end de Netmap qui sera lui directement relié à la carte réseau et partagera la mémoire pour les paquets avec elle. Il n'y aura plus de copie de paquets entre la carte réseau et les pilotes back-end, il y aura seulement une copie entre le front-end et le back-end. De plus, Netmap réalisera un polling sur la carte réseau: il n'y aura plus d'interruptions asynchrones à la réception des paquets.

Cependant, je n'ai jamais réussi à installer Netmap dans le domaine 0. Les sources de netmap n'étant pas mises à jour depuis trois ans il y avait plein d'erreur à la compilation. J'ai donc tenté de passé à une version de noyau Linux inférieur à 4.0 pour voir si cela fonctionnait. Malheureusement, cela n'a rien donné et de toute manière, être obligé de travailler avec de vieilles versions de

Linux n'est pas forcément judicieux. Nous avons donc décidé d'abandonner Netmap et de nous concentrer sur DPDK.

DPDK étant une initiative à l'origine d'Intel, il est constamment mis à jour, amélioré et supporté par de plus en plus de matériel. Nonobstant, le support de DPDK est surtout concentré sur l'hyperviseur KVM et non sur Xen. L'intégration de DPDK dans Xen n'est pas encore satisfaisante et il reste beaucoup de travail à faire. De plus, même si DPDK est supporté par click en mode utilisateur sur Linux, il ne l'est pas sur clickOS. C'est pourquoi, avant de nous intéresser à integrer DPDK dans clickOS, nous avons décidé de nous intéresser à ses performances sur Linux.

J'ai donc du mettre en place deux test sur Linux : un test de ping et un test de retransmission (forwarding) de paquets à l'aide d'iperf3. Pour les deux tests, des mesures ont été prises pour comparer 3 cas :

- Linux : c'est le noyau qui gère le ping ou le forwarding de paquets.
- Click modular router : Click est utilisé comme application sur Linux et gère le ping et le forwarding de paquets. Linux quant à lui s'occupe des paquets qui arrivent sur la carte réseau.
- Click modular router et DPDK : c'est la même configuration de tests que pour Click à ceci prêt que les interfaces réseaux ont été déconnectées de Linux et passées en mode DPDK. Les paquets arrivant sur ces interfaces (ports sur la carte) ne passent donc plus par le noyau et c'est Click qui doit donc traiter les paquets.

Le but était d'essayer de voir l'apport de performance de DPDK sur un traitement simple de paquets et d'essayer de le quantifier.

## 4.1 Test de performance: ping

### 4.1.1 Présentation

Pour le test de ping, présenté sur la figure 11, j'ai utilisé deux machines sur le même réseau. Une machine servant de client va simplement lancer une commande ping classique vers la deuxième machine. Cette dernière m'a servi de serveur ping qui va répondre à la requête envoyée par la première machine.

Un ping est simplement la mesure du round-trip time entre l'envoie d'une requête ICMP (Internet Control Message Protocol) et sa réponse ie le temps d'aller retour entre la requête et la réponse. ICMP est un protocole internet qui permet de savoir si une machine distante est disponible et permet de fournir des messages de contrôle et d'erreur pour la suite des protocoles Internet (tcp/ip par exemple) dont il fait parti.

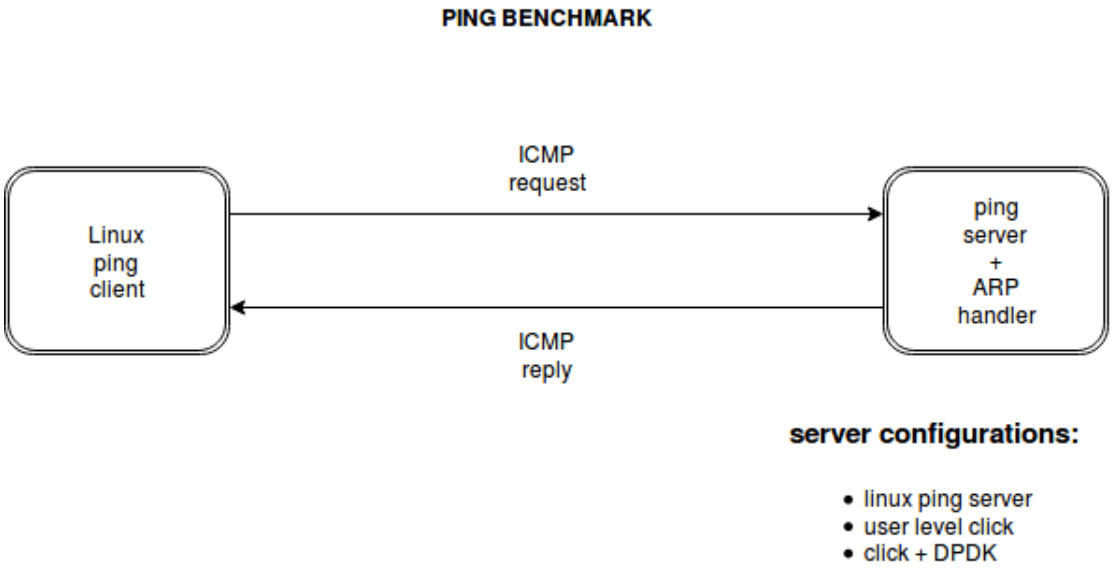


Figure 11: Configuration du test de ping.

Le client qui veut effectuer un ping sur une machine devra connaître une adresse IP liée à cette machine. Cependant pour que la requête arrive à destination, le client a aussi besoin de l'adresse physique (MAC) de la machine qu'il veut contacter. Avant d'envoyer la requête ICMP, il va donc envoyer en diffusion sur tout son sous-réseau une requête ARP (Address Resolution protocol) en demandant à ce quon lui donne l'adresse MAC de la machine correspondant à l'IP qu'il envoie avec la requête. Seule la machine correspondant à cette adresse IP répondra à cette requête même si d'autres machines ont déjà ce couple (IP, MAC) dans leur table. Elle va répondre à la requête ARP en renvoyant un message disant je suis l'adresse IP demandée et mon adresse MAC est ceci.

C'est pourquoi le serveur devra être capable, en plus de répondre aux requêtes ICMP, de répondre aux requêtes ARP pour son couple (IP, MAC). Pour le cas où le serveur est géré par le noyau Linux il n'y a pas de problèmes, il suffit de lancer la commande ping depuis le client. Par contre pour les cas Click et Click + DPDK il faudra écrire une configuration qui répond à la fois aux requêtes icmp et ARP pour un couple choisi (IP, MAC). En effet, dans le cas Click + DPDK cela est logique puisque les paquets ne passent plus par le noyau Linux du fait que les ports de la carte réseau ne soient plus liés à Linux. Mais pour le cas de Click sans DPDK l'explication est différente. Dans ce cas là, on veut que la réponse à la requête ICMP soit faite par le routeur Click et pas par Linux. Or, Linux voit quand même les paquets passer. Il a donc fallu ping une adresse IP quelconque ne correspondant pas à une adresse des adresses liée à une interface de Linux (et n'étant pas déjà prise sur le réseau où on se trouve). Ainsi, Linux ne fera rien sur ces paquets et il faudra gérer la requête ARP et la requête ICMP. Cela est faisable car la carte réseau est de base en mode *promiscuous*,

c'est à dire qu'elle reçoit tous les paquets, même ceux qui ne sont pas à destination des adresses des interfaces de la machine. Les paquets seront donc bien reçus et remontés à Click qui les traitera.

```
//ip address of the machine to ping
define($IP 192.168.55.22);

//MAC address of the machine to ping
define($MAC a0:36:9f:c7:67:98);

source :: FromDevice(ens4f0)
sink :: ToDevice(ens4f0);
c :: Classifier(
            12/0806 20/0001,
            12/0800,
     -);
arpr :: ARPResponder($IP $MAC);
q :: Queue;

source -> c
c[0] -> arpr -> q;
c[1] -> CheckIPHeader(14)
     -> ICMPPingResponder()
     -> EtherMirror()
     -> q;
c[2] -> Discard;
q -> sink;
```

Figure 12: Configuration pour un serveur ping Click.

```
//ip address of the machine to ping
define($IP 192.168.55.22);

//ip address of the machine to ping
define($MAC a0:36:9f:c7:67:98);

source :: FromDPDKDevice(0000:01:00.0)
sink :: ToDPDKDevice(0000:01:00.0);
c :: Classifier(
            12/0806 20/0001,
            12/0800,
     -);
arpr :: ARPResponder($IP $MAC);

source -> c
c[0] -> arpr -> sink;
c[1] -> CheckIPHeader(14)
     -> ICMPPingResponder()
     -> EtherMirror()
     -> sink;
c[2] -> Discard;
```

Figure 13: Configuration pour un serveur ping Click avec DPDK.

La figure 12 présente la configuration pour créer un routeur Click permettant de répondre aux pings envoyés à une adresse IP et à une adresse MAC définies par des variables. Cette configuration utilise les interfaces réseau de Linux. Elle définit un élément source pour dire depuis quelle interface on veut lire les paquets et un élément de sortie du routeur pour dire par quelle interface doivent repartir les paquets traités. Les paquets entrant vont d'abord passé par un élément qui sert de classificateur pour trier les éléments et les envoyer vers l'élément suivant approprié selon leur type. L'ordre donné pour le type des paquets en argument de l'élément classificateur correspondra au numéro du port de sortie de l'élément qu'ils emprunteront. Par exemple, ici, les paquets 12/0806 20/0001 correspondant à des requêtes ARP seront envoyés sur la sortie 0 du classificateur, les paquets 12/0800 correspondant à des paquets IP seront envoyés sur la sortie 1 et tous les autres type de paquets passeront par la sortie 2. On définit ensuite un élément qui prendra en entrée des requêtes ARP et qui, si elles sont destinées à l'adresse IP définie dans le routeur, créera et enverra sur sa sortie un paquet correspondant à la réponse avec l'adresse MAC souhaitée. On définit aussi un élément queue qui sert juste de tampon pour les paquets et qui n'est pas obligatoire pour le test mais qui l'est pour la configuration car les ports des éléments encadrant la file ne sont pas compatibles (push/pull). Finalement le flux des paquets dans le routeur sera le suivant :

- Les requêtes ARP seront envoyées par le classificateur à l'élément qui gère ces requêtes, ses réponses repartiront sur le réseau.

- Les paquets IP sortant du classificateur verront leur header testé pour vérifier que se sont des paquets IP valides. Ces paquets passeront ensuite dans un élément qui répond aux requêtes ICMP. Si un paquet entrant est une requête ICMP, cet élément crée une réponse en recopiant les informations du paquets entrant et envoie cette réponse sur sa sortie 0. Si le paquet entrant n'est pas une requête ICMP, il est simplement envoyé sur la sortie 1 de l'élément ou jeté si cette sortie n'existe pas (ici il sera donc jeté). Une chose qui m'a surprise est que dans la réponse créée pour la requête ICMP, la source et la destination n'ont pas changé par rapport au contenu de la requête initiale. Donc de base, la source de la réponse est l'émetteur initial de la requête et la destination est l'adresse du serveur. C'est pourquoi il faut faire passer la réponse créée dans un élément qui va inverser la source et la destination du paquet pour bien que la réponse arrive à l'émetteur du ping. La réponse est ensuite envoyée sur le réseau.
- Tous les autres paquets reçus sont jetés.

La figure 13 présente la configuration dans le cas Click + DPDK. Cette configuration est la même à ceci près qu'il faut juste changer les éléments d'entré et de sorti du routeur pour spécifier qu'on utilise des ports en mode DPDK.

### 4.1.2 Résultats

Les caractéristiques de la machine utilisée comme serveur ping étaient les suivantes :

- CPU intel xeon 3.4GHz avec 8 threads (1 socket, 4 cores, 2 threads par core)
- 8Go de RAM
- carte réseau 10Gb/s

En ce qui concerne les résultats, de manière logique au vu de ce que DPDK est censé apporter comme amélioration lors du traitement des paquets, nous nous attendions à ce que la configuration avec DPDK soit la plus performante.

La figure 14 présente le temps que prend un ping entre le client et le serveur en fonction de la taille des paquets de la requête du client pour les trois cas de configuration du serveur. La première chose à noter est que pour les réponses ICMP, Click tronque les réponses à partir d'une taille de paquets égale a 1480 octets. C'est pourquoi, après cette taille on voit peu, voir pas, de variation pour Click et Click + DPDK au contraire de Linux qui lui n'effectue pas cette troncature. L'enseignement principal que le peut tirer en regardant ces résultats est que la configuration utilisant Click et DPDK est entre une fois et demi et deux fois plus rapide que Linux. Cela répond donc en partie aux attentes que nous avions sur l'apport de performance de DPDK. Quant à Click seul, il performe un peu moins bien que Linux (qu'il finit par rattraper lorsqu'il tronque les paquets) et la courbe n'est pas aussi nette que pour Linux et DPDK, on aperçoit beaucoup plus d'oscillations. Cela n'est pas illogique, Click ayant été utilisé en mode utilisateur, les paquets devaient remonter de la carte réseau à Click en passant par le noyau ce qui induit une latence aléatoire supplémentaire(en fonction de ce que le noyau a à traiter).

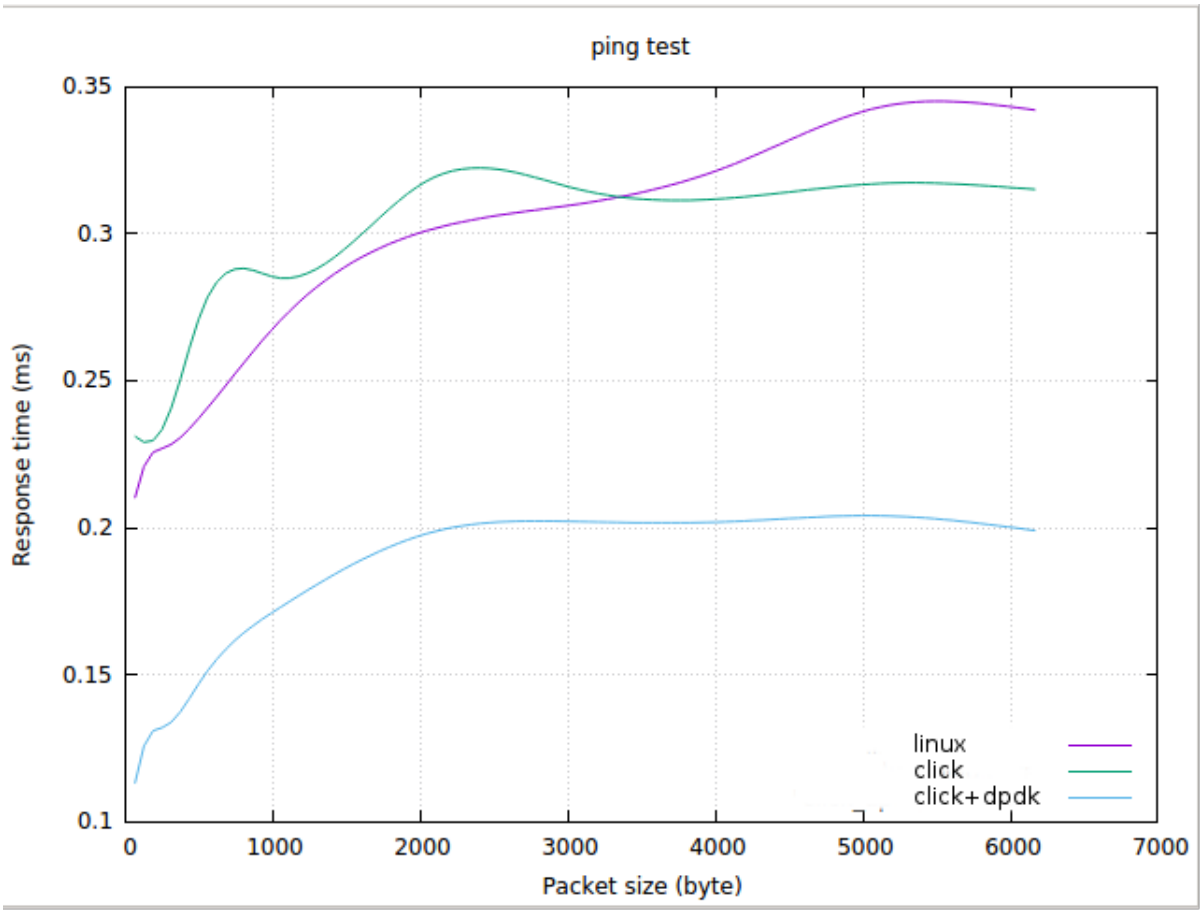


Figure 14: Temps de réponse à un ping en fonction de la taille des paquets.

Nous avons décider d'effectuer un autre test de comparaison pour confirmer ses résultats et tester DPDK dans une autre utilisation. J'ai donc ensuite effectué un test de retransmission de paquets entre trois mahines.

## 4.2 Test de performance: retransmission de paquets

### 4.2.1 Présentation

Pour ce test de retransmission de paquets présenté sur la figure 15, j'ai utilisé le logiciel iperf3. Il a été utilisé comme client sur une machine pour servir en tant que générateur de charge de paquets UDP. Il a été utilisé comme serveur pour recevoir les paquets générés et donne tout un tas de statistiques sur une session avec un client. Les statistiques qui m'ont intéressé pour ce test sont :

- la bande passante (bandwidth)
- la gigue : variation de la latence au cours du temps (jitter)

- la quantité de données transférées
- la quantité de paquets perdus

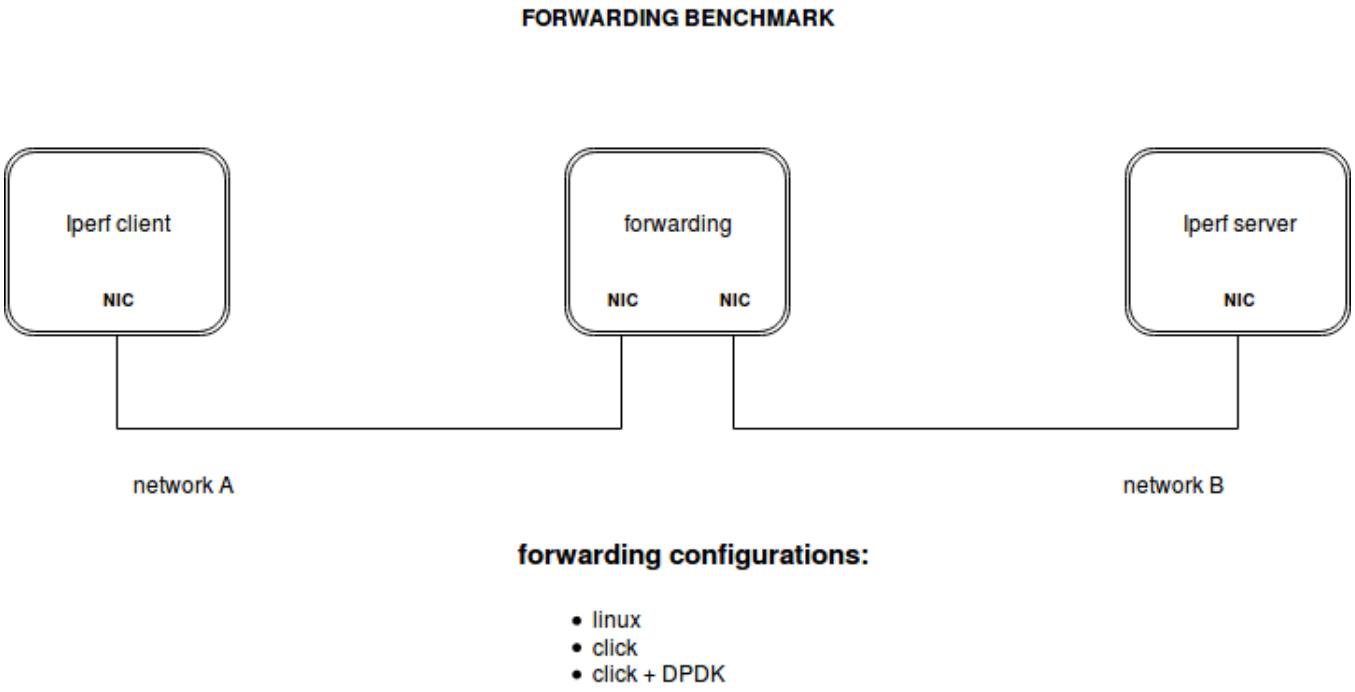


Figure 15: Configuration du test de retransmission de paquets.

Ce test s'est donc réalisé entre trois machines. Le client iperf sur un réseau A envoyait des paquets au serveur iperf sur un réseau B, Ces deux machines n'étant pas sur le même réseau, pour que les paquets arrivent à destination, il fallait une troisième machine entre les deux premières ayant accès aux deux réseaux pour faire le lien. C'est donc cette machine centrale qui effectue la retransmission des paquets via un switch qui inter-connecte les deux réseaux.

Pour pouvoir effectuer cette retransmission, il y a des manipulations à faire sur la machine où se trouve le client iperf et sur la machine qui effectue la retransmission selon le cas que l'on teste. Dans tous les cas, il faut modifier les routes des paquets sortant du client iperf. En effet, on veut que les paquets à destination du serveur iperf passe par la machine de forwarding et sans ce routage de toute manière les paquets ne pourraient pas être envoyé car la destination est sur un réseaux auquel le client n'a pas accès. L'adresse IP de la destination était 192.168.55.23, l'adresse de la source était 10.192.177.232 et les IP de la machine centrales étaient 10.192.177.77 et 192.168.55.44. Il faut donc taper la commande suivante sur la machine du client iperf :

```
route add -net 192.168.55.0 netmask 255.255.255.0 gw 10.192.177.77 dev eno1
```

Cette commande permet de spécifier à Linux que tous les paquets destinés à des machines du réseau 192.168.55 doivent être envoyés à la machine possédant l'adresse 10.192.177.77 en sortant par l'interface eno1 à laquelle est liée l'adresse IP 10.192.177.232 (cela permet de fixer l'adresse IP source des paquets). Ainsi, lorsque le client voudra envoyer un paquet IP (TCP, UDP...) au serveur, l'adresse IP de destination sera celle du serveur mais l'adresse MAC de destination sera celle de la machine de retransmission. Le paquet ira donc vers la machine de retransmission qui devra elle mettre l'adresse MAC du serveur dans le paquet à la place.

Pour le cas du test de retransmission de paquets qui se fait par Linux, il faut rendre la retransmission de paquet par Linux possible, ce qui ne l'est pas forcement par défaut. Pour cela, il faut d'abord écrire 1 dans le fichier */proc/sys/net/ipv4/ip_forward*. Il faut ensuite modifier les tables IP en tapant les commandes suivantes et permettre une retransmission de paquet enstre les interfaces eno1 et ens4f0 de notre machine :

```
sudo iptables -t nat -A POSTROUTING -o eno1 -j MASQUERADE
sudo iptables -A FORWARD -i eno1 -o ens4f0 -m state
              --state RELATED, ESTABLISHED -j ACCEPT
sudo iptables -A FORWARD -i ens4f0 -o eno1 -j ACCEPT
```

Pour le test avec les outils de Linux il n'y a rien d'autre à faire, il suffit de lancer le serveur iperf (avec les options –server et –udp) puis d'y connecter le client (avec les options –client –udp et -t pour spécifier le temps en seconde de la génération de charge).

Dans le cas du test avec Click il faut bien sur désactiver le forwarding de Linux (cela n'est pas nécessaire lors du test en mode DPDK car les paquets ne passe pas par Linux). Il faut aussi écrire une configuration qui créera un routeur qui retransmet les paquets. Le test porte uniquement sur la retransmission de paquets, j'ai donc décidé de laisser Linux traiter les requêtes ARP du client iperf pour obtenir l'adresse MAC de la machine de retransmission. Cela est ici faisable car contrairement au test du ping les adresses IP utilisées sont celles des interfaces réseau de Linux ce qui n'est pas gênant car le forwarding est coupé sur Linux pendant le test avec click donc Linux ne peut pas le faire.

Comme on peut le voir sur la figure 16, le routeur va simplement prendre tous les paquets arrivant sur l'adresse IP vers laquelle on a dit au client d'envoyer les paquets destinés au serveur (avec la commande route add) qui est sur le même réseau que le client iperf. Il va mettre l'adresse MAC de la source des paquets à l'adresse du client et la source à l'adresse du serveur. J'ai choisi de mettre les adresses MAC à la main car il n'y a pas d'élément Click qui permette de trier les paquets en fonction d'une adresse IP de source ou de destination. Donc tous les paquets sortant verront leurs adresses MAC de source et de destination changer même ceux n'ont rien à voir avec le test. Cela n'est pas gênant car le plus important était que les paquets du client soit bien transférés au serveur. Ensuite, les paquets sont renvoyés sur un autre réseau que celui de leur entrée, ce nouveau réseau étant celui du serveur de destination.

```
FromDevice(eno1, PROMISC true) -> EtherRewrite(18:03:73:d7:f2:0a, a0:36:9f:c7:66:b4)
                               -> Queue
                               -> ToDevice(ens4f0);
```

Figure 16: Fichier de configuration Click pour le forwarding.

Dans le cas du test avec Click et DPDK, il faudra par contre ici tout gérer. Donc, en plus de transférer les paquets au serveur iperf, le routeur Click devra répondre aux requêtes ARP du client iperf.

Sur la figure 17 on peut voir la configuration Click qui va servir à l'initiation du routeur pour le test avec Click et DPDK. On y définit l'adresse MAC de la machine qui exécute le routeur et l'adresse IP de l'interface qui reçoit les paquets du client iperf: c'est le couple pour lequel il faut répondre au requête ARP. On définit ensuite un éléments source pour récupérer les paquets sur le port connecté au réseau du client iperf. On définit deux sorties pour les paquets : une pour le port connecté au réseau du client pour envoyer les réponses ARP et une pour le port connecté au réseau du serveur pour lui retransmettre les paquets. Il y a aussi un classificateur, comme pour le test précédent, qui va renvoyer sur sa sortie 0 les requêtes ARP, sur sa sortie 1 les paquets IP et sur sa sortie 2 tous les autres paquets. Le flow des paquets dans le routeur sera le suivant :

- Les paquets arrivant du réseau du client par la source seront envoyés au classificateur.
- Le classificateur enverra les requêtes ARP à un élément qui créera une réponse appropriée et qui enverra cette réponse sur le réseau du client.
- Le classificateur enverra les paquets IP vers un éléments qui vérifiera s'ils sont valident, un autre élément supprimera l'en-tête Ethernet du paquet, puis un dernier élément remettra une nouvelle en-tête Ethernet en spécifiant que c'est un paquet Ethernet et en mettant l'adresse MAC source à l'adresse du client et l'adresse MAC de destination à l'adresse du serveur. Finalement les paquets seront envoyés sur le réseau du serveur.
- Le classificateur jettera tous les autres paquets.

### 4.2.2 Résultats

Les caractéristiques de l'environnement des tests étaient les suivants :

- La machine servant pour le forwarding avait un CPU intel xeon 3.4GHz avec 8 threads (1 socket, 4 cores, 2 threads par core), 8Go RAM et carte réseau 10Gb/s
- Hub/switch 1Gb/s pour relier les deux réseaux A et B

```
define($IP 10.192.177.77);
define($MAC 70:5a:0f:43:6e:86);

source :: FromDPDKDevice(0000:00:19.0)
sink0  :: ToDPDKDevice(0000:01:00.0);
sink1  :: ToDPDKDevice(0000:00:19.0);

c      :: Classifier(
            12/0806 20/0001,
            12/0800,
            -);

arpr   :: ARPResponder($IP $MAC);

source    -> c;
c[0]      -> arpr
          -> sink1;
c[1]      -> CheckIPHeader(14)
          -> Strip(14)
          -> EtherEncap(0x0800, 18:03:73:d7:f2:0a, a0:36:9f:c7:66:b4)
          -> sink0
c[2]      -> Discard;
```

Figure 17: Fichier de configuration Click pour le forwarding avec ports réseau en mode DPDK.

Les résultats obtenus pour ce test ne sont pas ceux auquel on aurait pu penser et la configuration de test avec Clikc et DPDK ne montre pas des performances meilleures que Linux ou Click seul.

Sur la figure 18, on peut voir le résultat obtenu pour la perte de paquets. Pour les trois configurations de tests, le pourcentage de paquets perdus en fonction de leur taille est très similaire et évolue de la même manière.

Ce test à été effectué en deux parties. Dans un premier temps, j'ai effectué ce test sur une grande plage de taille de paquets puis j'ai refait le même test en me concentrant sur des tailles de paquets plus petites pour avoir un résultat plus local (taille de paquets jusqu'à 3ko). J'ai effectué ce test de manière plus précise sur les petites de paquets car ce sont elles qui nous intéresse le plus. Je vais d'abord présenter les résultats pour la grande plage de taille de paquets.

Sur la figure 19, on trouve les résultats du test pour la latence. Sur de petites tailles de paquets (jusqu'à environ 3ko), la latence est à peu près la même pour les trois configurations de tests. On remarque qu'ensuite les configurations Click et Click avec DPDK ont une latence un peu plus faible mais qu'a partir d'environ 25ko, la latence sur Linux se stabilise alors qu'elle continue de monter dans les autres configurations. Cependant, l'écart entre les latences est d'au plus 1ms et on

atteint cette valeur pour de grandes tailles de paquets donc on ne peut pas vraiment dire que telle configuration du test est meilleure que les autres.

Les figures 20 et 21 montrent les résultats du test pour la bande passante et la quantité de données transférées pendant le test. Ces deux caractéristiques sont bien évidement liées et on voit bien que les deux graphes se ressemblent fortement. On ne distingue pas très bien ce qu'il se passe pour des petites tailles de paquets mais il ressort deux parties sur les graphes avec une séparation à environ 25ko. En dessous de cette taille de paquets, les configurations Click et Click avec DPDK sont globalement au dessus de Linux mais leurs courbes font des oscillations autour de la courbe de Linux qui est plus stable. Je n'explique pas trop ces grandes variations pour Click et Click avec DPDK. Parmi les deux réseaux utilisés, celui entre la machine qui retransmet les paquets et le serveur était un réseau isolé avec simplement 2 machines donc sans trafic indésirable. Par contre, celui entre le client et la machine de retransmission est un réseau interne à orange avec du trafic non lié au test et non maitrisable. Il se peut que ce trafic ait joué sur le test et que Click et Click avec DPDK soient plus sensibles au trafic que Linux (due à mes configurations de routeurs) ce qui expliquerait en partie ces variations. Au dessus de 25ko, les trois courbes se rejoignent et suivent ensuite la même courbes et les grandes oscillations ont disparues. On peut aussi voir que les valeurs de bande passante et de quantité de données transférées ne font pas qu'augmentées avec la taille des paquets mais qu'il y a un maximum entre 15ko et 25ko selon les configurations du test.

J'ai ensuite refait le même test en me concentrant plus précisément sur des tailles de paquets inférieures à 3ko. Les résultats de latence et pertes de paquets sont toujours très similaires pour les trois configuration et il n'apparaît pas d'informations importantes à retirer.
Sur les figures 22 et 23 on aperçoit le résultat pour la bande passante et la quantité de données transmises pour un zoom sur de petites tailles de paquets. On voit de nouveau une cassure sur les courbes, cette fois-ci à 1,5ko. Avant, les trois courbes sont plutôt régulières et après on retrouve des oscillations qui marquent des variations dans les valeurs des résultats et cette fois leur intensitée est la même pour les trois configurations de test. J'ai dit précédemment que le trafic sur le réseau pourrait expliquer en partie ses variations pour la cassure précédente mais la valeur observée ici de 1500 octets comme apparition des variations nous donne une autre raison pour cette cassure. En effet, cette valeur correspond à la valeur de base du Maximum Transmission Unit (MTU) d'une carte réseau. Le MTU est la taille maximale d'un paquet qui peut être transmis par la carte en une seule fois, si un paquet est plus gros il sera fragmenté. Les oscillations dans les valeurs seraient donc aussi expliquées par la fragmentation des paquets reçus carte notre carte réseau avait un MTU à 1500 (information observable via ifconfig pour chaque interface). Sinon, une nouvelle fois les trois courbes présentent des valeurs assez proches et aucune configuration ne ressort comme plus performante.

En conclusion de ce test, le seul enseignement que l'on peut tiré est que DPDK est au moins aussi performant que Linux. Il se peut aussi que ces résultats non concluant par rapport aux attentes que l'on avait pour DPDK aient une autre explication. Ils pourraient s'expliquer par les

caractéristiques de l'environnement de test. Comme dit plus haut, les réseaux sont reliés par un switch 1G/s et il se peut que cela ne permette pas d'atteindre des débits qui mettraient en avant les avantages de DPDK.

Après avoir effectué ces tests, l'étape suivante aurait été de se tourner vers ClickOS pour intégrer le support de DPDK dans celui-ci et par la même occasion améliorer celui de Xen. Cependant, cela aurait demandé trop de travail pour être effectuer avant la fin du stage car c'est un travail conséquent. Nous avons donc décidé de nous tourner vers la deuxième problématique évoquer pour mon stage, à savoir l'implémentation d'une communication inter unikernel pour des instances localisées sur la même machine.

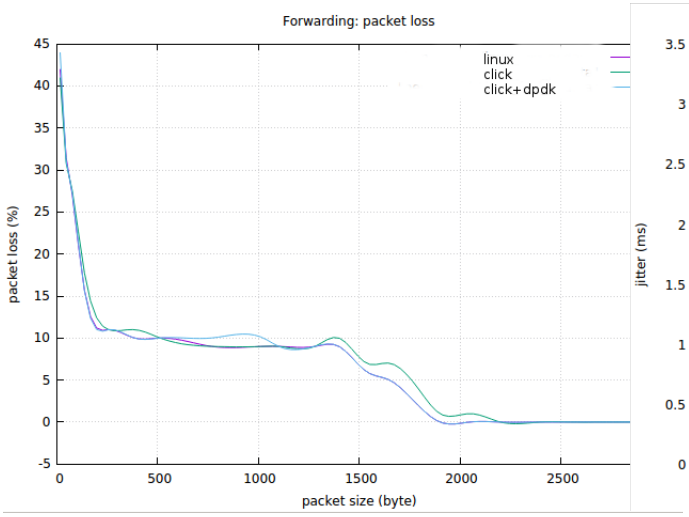


Figure 18: Résultats pour la perte de paquets.

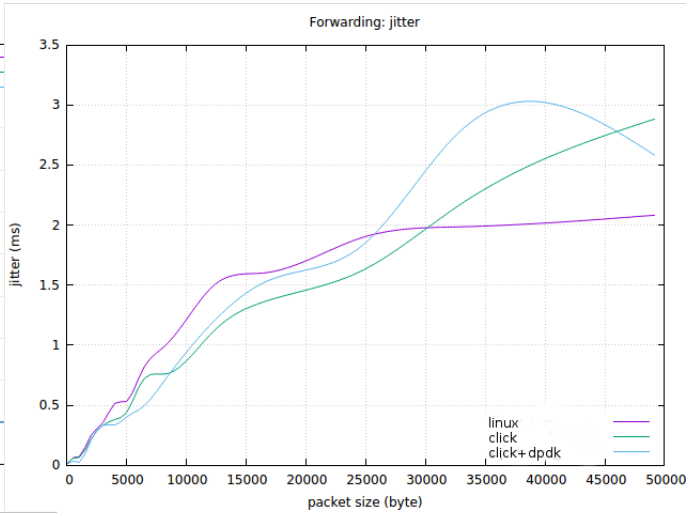


Figure 19: Résultats pour la latence.

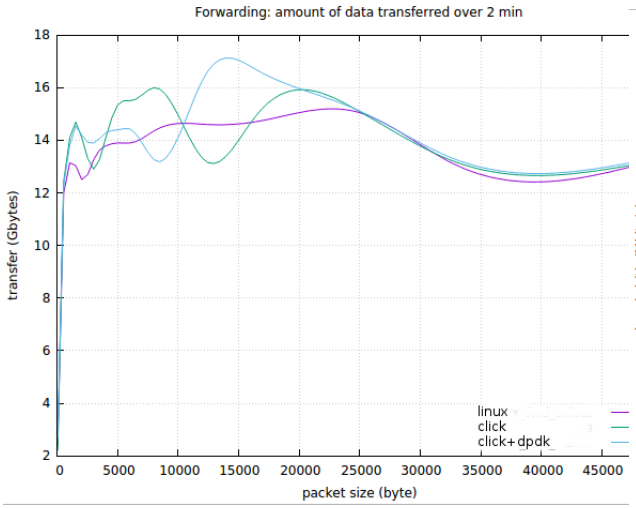


Figure 20: Résultats pour la quantité de données sur une grande plage de taille des paquets.

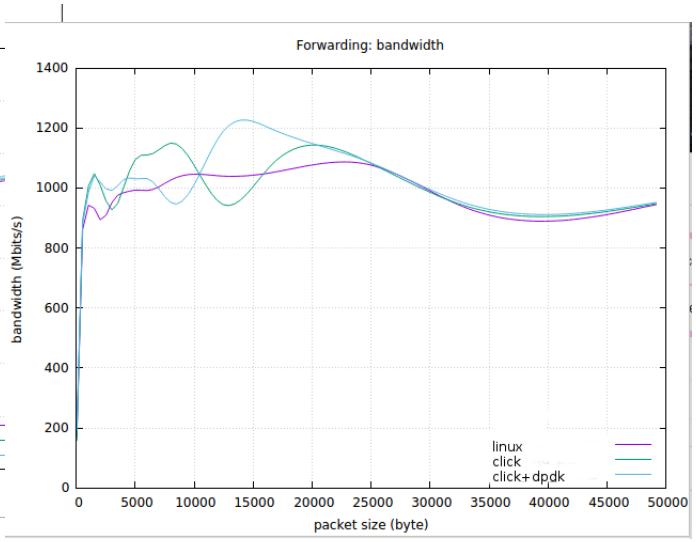


Figure 21: Résultats pour la bande passante sur une grande plage de taille des paquets.

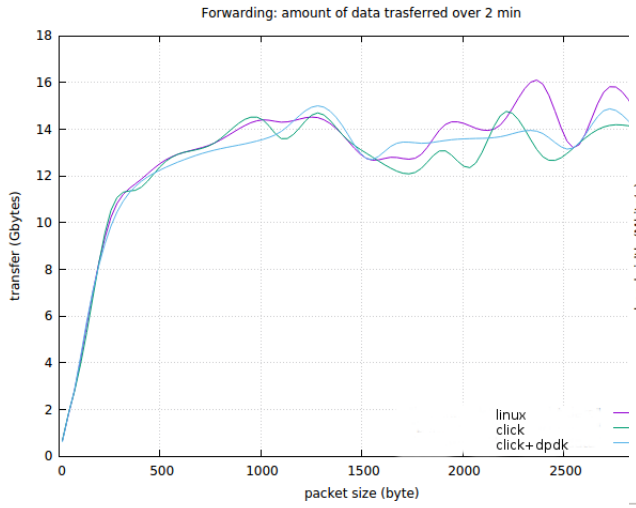


Figure 22: Résultats pour la quantité de données sur des petites tailles de paquet.

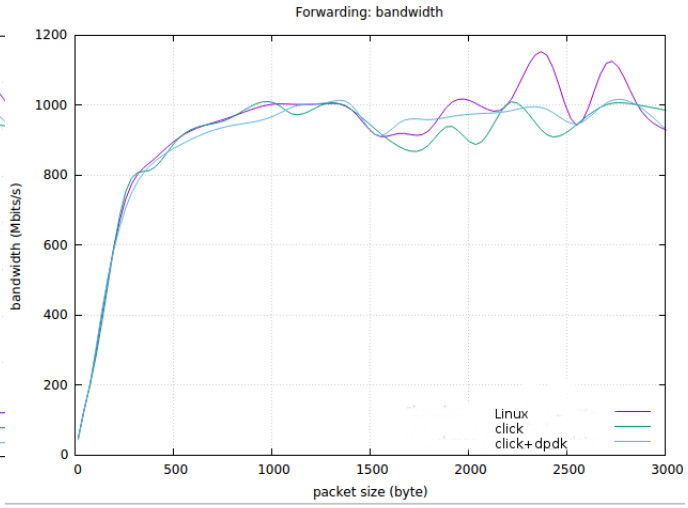


Figure 23: Résultats pour la bande passante sur des petites tailles de paquet.

# 5 Communication inter unikernels: implémentation[3]

Rappelons ici, que Click est une application qui permet de créer des routeurs composés d'un chaînage d'éléments qui vont traiter les paquets chacun à leur tour. Les routeurs communiqueront avec l'extérieur grâce à des éléments d'entrée et de sortie. Rappelons aussi que ClickOS est un unikernel qui embarque les outils nécessaire pour exécuter des configurations Click (Click sur mini-os). Dans un routeur Click (et donc aussi dans ClickOS), les paquets ne sont pas nécéssairement copiés d'un élément à l'autre et souvent ce sont les mêmes paquets qui parcourent tout le routeur. Il peut être intéressant de prendre la configuration d'un routeur Click monolithique dans un unikernel ClickOS et de la scinder pour former des configurations Click qui seront chacunes dans un unikernel ClickOS différent. Il y a plusieurs raisons à cela:

- Un unikernel clickOS ne posède qu'un seul CPU virtuel, ce qui est une bonne chose car il n'a pas à se soucier des contraintes apportées par une exécution multi-processeurs et ne requiert donc pas d'ordonnanceur au sein de l'unikernel. Ce CPU virtuel va nécessiter une certaine charge d'exécution sur les CPUs physiques. Etant donné que le CPU virtuel ne peut fonctioner que sur un mode d'exécution mono-coeur, il pourra, à un moment donné, soliciter au plus la totalité d'un CPU physique (il ne pourra pas utiliser deux CPUs physiques en même temps car cela ne serait plus mono-core). Si on normalise les choses et qu'on dit que la charge délivrable par un CPU physique est de 1 (normalement c'est le nombre d'instructions faisables par seconde), alors si l'unikernel a besoin d'une charge inférieure à 1 alors il n'y a pas de problèmes. Mais il se peut que la charge demandée par l'unikernel soit supérieure à ce que peut fournir un CPU physique. Dans ce cas là, l'exécution de l'unikernel va voir son nombre de deadline miss augmenter et on va perdre en performance. En effectuant un découpage de l'unikernel précédant, on répartirait la charge d'exécution initiale sur plusieurs CPU virtuels et donc on aurait plus de chance d'avoir une charge demandée par chacun de ses CPU virtuel inférieur à 1 et dont la somme serait la charge initiale. Avec ce découpage on pourrait tenir la charge demandée par l"excution de l'unikernel.
- On a vu que dans un unikernel clickOS il y a de la concurrence grâce à des threads légers. Cependant, chaque thread sera exécuté à tour de rôle du fait de l'execution mono-coeur requise par l'unique CPU virtuel: il n'y a pas de parallélisme. En découpant une configuration de routeur en plusieurs parties plus petites, on pourra exécuter chaque partie en parallèle sur différents processeurs pysiques en déployant chacune des partie de l'application Click dans un unikernel ClickOS. On obtiendrait ainsi du parallélisme.
- Le cloud aujourd'hui nécessite de la modularité et de l'élasticité. On veut pouvoir s'adapter rapidement et dynamiquement à la charge du traffic sur le réseau en utilisant des unikernels effectuant des fonctions simple et pouvant se brancher ou débrancher d'un chaînage.

Une fois que l'unikernel ClickOS de départ a été scindé en plusieurs unikernels, pour que l'on garde la fonctionnalité de ce premier routeur, on introduit la nécessité pour les unikernels qui résultent

[3]Références: [18]

de la scissions de pouvoir communiquer et se transmettre les paquets: c'est là qu'un mécanisme de communication devient nécessaire. Nous avons décider de faire une communication par canal de mémoire partagée. De plus, pour garder cet aspect d'un routeur Click monolithique où les paquets traversent le chaînage des éléments, nous allons utiliser un pool de mémoire partagée entre tous les unikernels et ils pourront s'échanger des paquets mis ce pool plutôt que de copier les paquets à chaque point de communication. Un paquet traversera donc toute la chaîne d'unikernels comme il le ferait avec une chaîne d'éléments d'un même routeur Click. Ainsi, au lieu d'avoir un seul unikernel ClickOS qui fait tourner une configuration Click complexe on aurait plusieurs unikernels ClickOS qui communiquent par de la mémoire partagée (canal) et avec de la mémoire partagée (pool de mémoire).

Grâce à une communication rapide par canal de mémoire partagée, on pourrait compenser la perte de performances résultant de la scission d'une configuration pour un gain de modularité et d'élasticité. En effet, la nécessité d'ajouter un mécanisme de communication entre plusieurs systèmes à de grandes chances de faire diminuer les performances globales par rapport à une exécution complète interne à un unique système et ce malgrè les points positifs apportés par une solution de découpage. Le choix du type de la communication et de son implémentation est donc primordial. On espère qu'avec une communication par canal de mémoire partagée efficace on arrivera à des performances similaires à l'exécution d'un seul unikernel mais en ayant gagné en modularité et élasticité. La suite de mon stage s'est donc portée sur l'implémentation d'un outil de communication par canal entre unikernels colocalisés sur une même machine en s'appuyant sur de la mémoire partagée.

## 5.1 Caractéristiques

La première chose qu'il a fallu faire, c'est de définir les caractéristiques que nous voulions pour ce module de communication pour qu'il soit le plus performant possible et qu'il devrait respecter. Les caractéristiques sont les suivantes :

- Nous voulions éviter le plus possible les copies lors de communication, il ne fallait pas qu'il y ait, par exemple, une copie des données de l'écrivain à un buffer puis du buffer au lecteur.
- Il fallait, si c'était faisable, que l'accès à toutes les structures partagées soit fait sans verrou (lock-less).
- La communication doit être point à point: un seul écrivain et un seul lecteur.
- Le module doit facilement être intégrable au framework Click donc il faut donner la priorité à un mode non bloquant.

## 5.2 Structure et architecture du module

Il a ensuite fallu définir l'architecture de notre module. Comme dit précédemment dans le rapport, ClickOS est constitué d'un noyau (mini-os) et d'un partie applicative (Click modular rou-

teur). Nous avons donc décidé d'intégrer à mini-os une API pour permettre l'allocation d'un pool de mémoire partagée et permettre l'envoi et la réception de données via une structure de mémoire paragée. Le fait d'intégrer une API dans mini-os permet de pouvoir facilement porter ce travail sur n'importe quel unikernel utilisant mini-os comme noyau. Il suffit de l'utiliser dans la partie applicative de l'unikernel. Notre cible étant ClickOS, il faudra créer de nouveaux éléments Click qui l'utilisent pour inclure la communication dans les routeurs et connecté des instances de ClickOS. Le volume de code de l'API dans mini-os est d'un peu plus de mille lignes de code et le volume de code des éléments click est compris entre 200 et 300 lignes de code.

Pour chercher la structure à adopter pour ce module de communication par mémoire partagée j'ai d'abord cherché et regarder parmi les solutions existantes. Celle dont je me suis le plus inspiré à été la librairie libvchan décrite précédemment dans ce rapport. Dans un premier temps je me suis intéressé au fonctionnement de cette librairie en lisant le code source et en essayant de le comprendre. Une fois cela fait je m'en suis inspiré pour mon travail à quelques exceptions prêt. La principale différence entre la librairie libvchan et ce que nous souhaitions faire est que la librairie libvchan utilise de la mémoire partagée pour transmettre les données d'un domaine à l'autre mais dans les deux domaines les données ne partagent pas la même mémoire. Ainsi, il y a une copie des données du premier domaine au buffer et du buffer au 2e domaine. Pour palier cela, ma solution, en plus d'une mémoire partagée pour la structure de transmission de données entre les domaines, possède aussi un pool de mémoire partagée où sera stocké les données qui seront partagées entre tous les domaines. Ainsi, il ne sera transmis entre les domaines que des adresses vers ce pool de mémoire. La deuxième différence est qu'il n'y aura pas de distinction entre deux domaines qui communiquent avec d'un coté un domaine serveur qui allouerait la mémoire et de l'autre un domaine client qui mapperait cette mémoire. L'allocation de mémoire se fera par un domaine tierce. Dans un deuxième temps, j'ai essayé de manipuler cette librairie et de faire communiquer deux domaines Linux en l'utilisant. Pour cela, j'ai pris deux images Debian que j'ai mises sur une partion vg0 et que j'ai montées avec la commande *sudo mount -o loop,rw /dev/vg0/vm /home/maxoux/* dans le domaine 0 de Xen. L'argument rw donne les droits en lecture et en écriture sur l'image montée, */dev/vg0/vm* est le chemin de l'image et */home/maxoux/VM* le point de montage dans le domaine 0. J'ai mis un exécutable utilisant la libvchan et les .a de la librairie statique dans les deux images montées puis j'ai lancé ces deux images en tant que machine virtuelle sur Xen. Ce programme de test de la librairie libvchan est déjà présent dans ses sources, il suffit de compiler la librairie pour avoir un exécutable lié avec la librairie statique pour faire un test. Et enfin, j'ai exécuté le programme sur les deux machines et ce que j'écrivais dans la console de l'une était affiché dans la console de l'autre.

La structure visible sur la figure 24 est la solution vers laquelle je me suis tourné après cette étude de la libvchan et après discussion avec mes encadrants. Elle comporte trois parties: un contrôleur, un pool de mémoire et des buffers en anneau. L'API fournie des fonctions pour gérer ces trois parties.

Le tableau ci-dessous présente rapidement les fonctions publiques fournies par l'API :

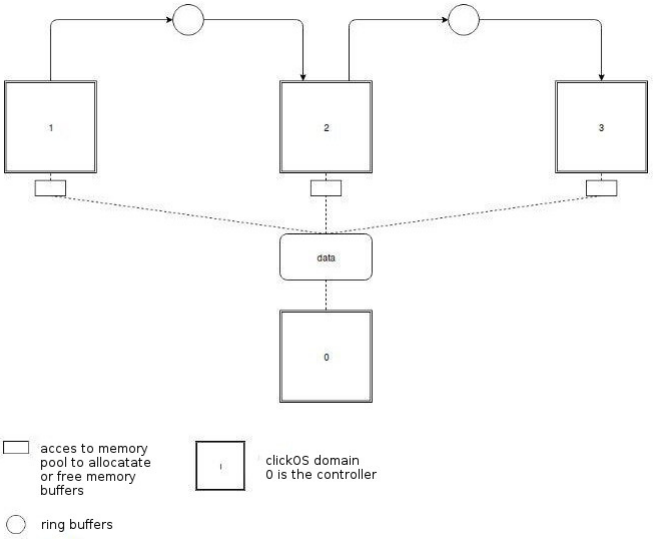


Figure 24: Architecture du module de communication dans mini-os.

| Fonction | description |
|---|---|
| `shmp init_shm(int shm_order, int block_order, domid_t *ids, int nbr_ids)` | Alloue la mémoire pour le pool de mémoire partagée et donne les droits en lecture/écriture à tous les domaines passés en argument |
| `void free_shm(shmp m)` | Libère la mémoire allouer pour le pool de mémoire partagée |
| `channel init_ring_buffer(int ring_order, domid_t *ids)` | Alloue la mémoire pour un canal de communication et donne les droits en lecture/écriture aux deux domaines passés en argument |
| `channel init_ring_buffer(int ring_order, domid_t *ids)` | Libère la mémoire allouée pour un canal de communication |
| `int map_shm(domid_t self_domid, domid_t controller)` | Mappe le pool de mémoire partagée dans un domaine donné |
| `int unmap_shm(void)` | Unmappe le pool de mémoire partagée |
| `packet_buf_t *alloc_mbuf(int nbr_buf, int err)` | Alloue exactement nbr_buf buffers dans le pool de mémoire partagée ou échoue |
| `packet_buf_t * free_mbuf(packet_buf_t *packet_buf, int nbr_buf)` | Libère les buffer du pool de mémoire partagée passés en argument |
| `channel map_ring_buffer(domid_t *ids, char function)` | Mappe un canal de communication dans un domaine donné en spécifiant s'il va lire ou écrire |
| `int unmap_ring_buffer(channel chn)` | Unmappe un canal de communication |
| `int send_mbuf(channel chn, packet_buf_t *packet_buf, int nbr_buf);` | Envoie des buffers du pool de mémoire partagée sur un canal donné |
| `int recv_mbuf(channel chn,` | Reçoit des buffers du pool de mémoire partagée |

Un utilisateur, grâce à l'API dans mini-os, aura donc accès à des fonctions pour :

- Ecrire le code d'un contrôleur pour allouer ou libérer de la mémoire partagée.
- Mapper ou unmapper un pool de mémoire et allouer et libérer des buffers dedans.
- Mapper ou unmapper des canaux pour envoyer et recevoir des données venant du pool de mémoire entre deux unikernels.

## 5.3 Implémentation

### 5.3.1 Le contrôleur

La première partie du module est un contrôleur qui sera simplement présent pour allouer et désallouer toute la mémoire partagée que ce soit pour les buffers en anneau ou le pool de mémoire. Ce contrôleur pourrait être théoriquement n'importe quelle machine virtuelle s'exécutant sur Xen y compris le domaine 0. Cependant j'ai décidé d'intégrer les outils permettant de l'utiliser dans mini-os donc pour nous le contrôleur sera une machine virtuelle clickOS qui n'exécutera pas de routeur mais dans le main de laquelle on exécutera le code du contrôleur. Pour l'utiliser dans une autre machine virtuelle n'ayant pas pour noyau mini-os, il faudrait porter les fonctions de l'API dont il a besoin. Ce contrôleur va donc utiliser l'API pour allouer la mémoire pour le pool de mémoire partagée et pour chaque buffer en anneau servant au transfert des données qui vont être utilisées par les domaines ClickOS. Il va aussi donner les droits aux domaines pour leur permettre de mapper cette mémoire.

Le passage des droits aux domaines allant mapper la mémoire se fait via le Xenstore. Par exemple, les droits pour le pool de mémoire pour un domaine donné seront mis dans le chemin */local/domain/controllerID/data/domainID/mem*. Ou encore, les droits pour les buffers en anneaux pour un domaine donné seront écrits dans le chemin */local/domain/controllerID/data/domainID/char/remoteDomainID/ring*. Le caractère char étant soit 'w' si le domaine en question sera lecteur soit 'r' s'il sera lecteur. Cela permet de pouvoir avoir deux buffers en anneau entre deux domaines s'ils veulent communiquer dans les deux sens (un anneau étant unidirectionnel). Comme il faut un entier représentant les droits pour chaque page à partager, les valeurs des chemins sont sous la forme « grant0;grant1;...;grantN » et il faudra parser ce string pour les récupérer.

Pour le pool de mémoire partagée, lorsque celui ci est grand (par exemple 1mo correspond a 256 pages de 4ko en mémoire), il a fallu contourner une limitation du Xenstore qui n'est pas fait pour transmettre beaucoup de deonnées et qui se retrouve saturé si le nombre de droits à écrire est trop élevé. Pour cela, j'ai décidé que lorsque la taille du pool de mémoire dépasse 7 pages en mémoire je n'écrirai pas dans le Xenstore les droits concernant ces pages mais j'écrirai les droits concernant des pages mémoires où seront écrits les droits pour les pages de départ. Il faut donc allouer des

pages de mémoire partagée en plus. Par exemple, si le pool de mémoire fait 2mo, il faut 512 pages de mémoire partagée. Au lieu d'écrire dans le Xenstore un string contenant les 512 droits plus les séparateurs, je vais allouer 2 pages mémoires dans lesquelles je vais écrire ces droits. En effet, j'ai monté le nombre de pages qu'un domaine peut partager simultanément de 2048 de base à 8192. Un droit sous forme de string fera donc au plus quatre octets et un string de n droits fera au plus (n*5)-1 octet si on compte les ';' séparateurs. Donc pour, écrire 512 droits, il faut deux pages mémoire de 4ko. Je vais ensuite récupérer des droits pour partager ces deux pages avec le domaine qui souhaite mapper la mémoire et je vais écrire dans le Xenstore les droits pour ces pages. Ainsi, le domaine qui voudra récupérer les 512 droits lira dans le Xenstore les droits de deux pages qu'il devra mapper et dans lesquelles il lira les 512 droits. Une fois cela, fait il n'aura plus qu'a unmapper les deux pages ayant servies d'intermédiaire.

### 5.3.2 Le pool de mémoire

La deuxième partie du module est un pool de mémoire partagée qui sera mappée par tous les unikernels cliCkOS. L'API sera utilisée pour allouer et désallouer des buffers dans cette mémoire ou l'unmapper. Les paquets qui circuleront d'un routeur ClickOS à l'autre seront donc mis dans des buffers tirés de cette mémoire et seront partagés par tous les unikernels.

Le pool de mémoire partagée est représenté par la structure suivante dont le contenu est partagé entre tous les unikernels mappant le pool:

```
typedef struct shm_t {
        shm_info_t *info;
        data_t     *mem;
        uint8_t    *tabe;
} shm_t;
```

Cette structure est déclarée de manière statique dans chaque unikernel et elle est initialisée dans une fonction qui mappe la mémoire partagée (où l'alloue si c'est le contrôleur) et qui indique seulement la réussite ou l'échec de l'initialisation. La déclaration statique fait que la structure ne sera accessible que dans le fichier où elle est définie, l'utilisateur n'aura donc pas accès à cette structure et au pool de mémoire partagée, il aura seulement accès aux fonctions de l'API qui permettent d'allouer ou de libérer des buffers dans le pool de mémoire partagée.

Le pointeur info pointe sur une structure contenant des informations sur le pool de mémoire : la taille de la mémoire, la taille des buffers et la taille de la table d'allocation (le nombre de buffers). La taille du pool de mémoire et la taille des buffers est forcément une puissance de deux. Ces informations sont remplies par le contrôleur à l'allocation et elle sont passées au contrôleur en argument par l'utilisateur. Les unikernels mappant la mémoire n'ont donc pas besoin de ces informations en argument et les récupèrent en mappant la mémoire.

Le pointeur mem est un pointeur sur le pool de mémoire partagée.

Le pointeur table est un pointeur vers un tableau qui sert de table d'allocation pour les buffers de la mémoire. Chaque case du tableau correspond à un buffer dans le pool de mémoire. Une valeur de 0 dans une case indique que le buffer est libre et une valeur de 1 indique que le buffer est alloué. La case i de cette table correspond au buffer à l'adresse *(mem + i * buffer_size)*.

Les unikernels qui auront besoin de buffers utiliseront cette table de manière concurrente pour en allouer. Ces accès doivent se faire sans verrou et il ne faut pas qu'un buffer soit alloué plusieurs fois. L'allocation d'un buffer s'effectue donc avec une opération atomique. Un unikernel qui veut allouer un buffer parcourra linéairement la table d'allocation jusqu'à trouver une case à 0. Il va récupérer l'indice de cette case et tenter d'écrire dedans de manière atomique. J'ai trouvé deux possibilités, soit écrire avec une opération de *test and set*, soit écrire avec une opération de *compare and swap*. Le test and set est une opération qui va écrire dans une variable la valeur passée en argument et qui va renvoyer la valeur qui était contenue précédemment dans la variable. Le compare and swap, lui, va comparer la valeur de la variable avec une valeur passée en argument. S'il y a égalité, il va écrire dans cette variable la deuxième valeur passée en argument et dans tous les cas il renvoie la valeur initiale de la variable. J'ai décidé d'utiliser un compare and swap plutôt qu'un test and set car le compare and swap ne va écrire dans la mémoire que lorsque c'est nécessaire. Ainsi, lorsqu'un unikernel voudra écrire dans une case de la table d'allocation pour allouer un buffer il va faire un compare and swap en comparant la valeur de la case à 0 et écrira 1 dans cette case si sa valeur était 0. Le retour de ce compare and swap permet de savoir si l'allocation a réussi ou non. Si la valeur de retour est 0, cela veut dire que cet unikernel a été le premier à écrire dans cette case et que par atomicité de l'opération d'écriture aucun autre unikernel ne pourra écrire avec succès dans cette même case. Si la valeur de retour est 1, cela veut dire qu'entre la lecture de la case qui a donné 0 et l'écriture dans cette case il y a un autre unikernel qui a déjà écrit dans la case et qui a alloué le buffer. Dans ce cas, l'allocation a échoué et recommence au début en recherchant un nouvel indice libre dans la table d'allocation. Ce mécanisme recommencera jusqu'à la réussite de l'allocation ou jusqu'à ce que tous les buffers aient été alloués et qu'il n'y en ait plus de disponibles ou jusqu'à atteindre le nombre limite d'essais qu'il est possible de spécifier en argument de la fonction d'allocation. Cette allocation est donc linéaire en la taille de la table d'allocation (ie le nombre total de buffers disponibles).

Pour la désallocation, on prend en argument l'adresse du buffer à libérer, on calcule l'indice dans la table d'allocation correspondant à ce buffer et on effectue une opération de compare and swap pour passer la valeur de cette case à 1. Si valeur de retour 1 c'est que le buffer était bien alloué, si c'est 0 qu'il y a une erreur.

### 5.3.3 Les buffers en anneau

La troisième partie du module correspond aux buffers en anneau qui serviront de canal de communication partagé entre deux unikernels. Chaque canal est unidirectionnel : on écrit depuis un unikernel et on lit dans l'autre. Il faudra donc spécifié à l'initialisation qui est l'écrivain et qui est le lecteur.

Un canal est représenté par la structure suivante dont le contenu est partagé entre deux unikernels l'utilisant pour communiquer :

```
typedef struct ring_info_t {
        unsigned long w_cpt;
        unsigned long r_cpt;
        int           order;
} ring_info_t;

typedef struct ring {
        ring_info_t  *info;
        packet_buf_t *buf;
} channel_t;
```

Tout comme pour le pool de mémoire partagé, chacune des structures pour un canal sont déclarées comme statique ne laissant l'accès qu'aux fonctions de l'API pour envoyer ou recevoir des données via un canal. Chaque unikernel possède un tableau de canaux dont la taille est définie dans le code par une macro. A chaque initialisation d'un canal, s'il y a de la place dans le tableau et si l'initialisation s'est bien déroulée, l'utilisateur récupère un entier correspondant à l'indice de ce canal dans le tableau et qui devra être fourni lorsque l'utilisateur voudra l'utiliser. Cet indice sera libéré lorsque l'utilisateur unmappera le canal correspondant.
Le pointeur info pointe vers une structure qui contient des informations concernant un canal : les offsets d'écriture et de lecture dans le buffer du canal et la taille du buffer (qui est une puissance de deux).

Le pointeur buf est un pointeur vers le buffer qui sert pour transférer les données. Nous souhaitions qu'il n'y ait pas de multiples copies des données lors du passage d'un unikernel à l'autre. C'est pourquoi, le buffer d'un canal ne contient pas directement des données mais des adresses qui sont des pointeurs vers des buffers alloués dans le pool de mémoire partagée qui contiennent eux les données. Lors de l'envoie et de la réception, il n'y aura donc que des copies d'adresses et pas une copie complète des données.

Un canal de communication entre deux unikernels est définie par un buffer en anneau. Ce canal est unidirectionnel et il y a un unique écrivain et un unique lecteur. Pour connaître la place disponible dans le buffer, la quantité de données déjà présente dans le buffer et à partir de quel indice il faut

lire où écrire dans le buffer, j'ai simplement utilisé deux compteurs : un pour l'offset d'écriture et un pour l'offset de lecture. Ces deux compteurs seront utilisés pour calculer toutes les informations nécessaires pour la lecture ou l'écriture et il n'y a pas besoin de verrou pour accéder à ces compteurs ou au buffer du canal.

Pour que cela fonctionne, il faut que le type de ses deux compteurs puisse contenir des valeurs supérieures ou égales à la taille du buffer de l'anneau pour pouvoir couvrir tous les indices du buffer. Ici, un anneau ne pourra pas avoir une taille supérieur à un long non signé. Tant que la valeur d'un compteur est inférieure ou égale à la taille du buffeur l'offset réel sera la valeur du compteur. Dans le cas où la valeur du compteur dépasse la taille du buffer, l'offest réel dans le buffer est obtenu en effectuant un ET logique entre la valeur du compteur et la taille du buffer moins un. Le buffer ayant une taille correspondant à une puissance de deux, si on soustrait un à la taille du buffer on obtient une valeur en binaire qui ne contient que des uns. Un ET logique avec la valeur du compteur va mettre à 0 tous les bits qui dépasse la taille du buffer et ne garder que les bits de poids faible qui correspondent à l'offset réel. Par exemple, si la taille du buffer est 16 et que le compteur vaut 18, l'indice réel dans le buffer est 2 et il est bien obtenu en faisant 10010 & 01111 = 00010.

La quantité de données disponible à la lecture dans le buffeur est toujours obtenue en soustrayant au compteur d'écriture le compteur de lecture et la place disponible en écriture dans le buffeur est la taille du buffer moins ce qui est disponible à la lecture. Cela reste vrai même lorsque le compteur d'écriture à overflow et celui de lecture pas encore. Car même si le compteur d'écriture avance avant celui de lecture, il ne peut overflow qu'une seule fois avant que celui de lecture n'overflow. Cela ne serait plus vrai si le compteur d'écriture pouvait overflow deux fois avant que celui de lecture n'overflow une fois. Pour que le compteur d'écriture overflow deux fois de plus que celui de lecture, il faudrait au minimum pouvoir écrire un nombre de données équivalent à la taille du buffer plus un avant une lecture, ce qui est impossible. Par exemple, si on stocke les compteur sur huit bits (non signé) et que le compteur d'écriture vaut 2 et que le compteur de lecture vaut 252 la soustraction donnerait -250. Ce qui en binaire serait 10000110. Or les compteur sont sur huit bits donc cette valeur serait tronquée à 0000110, ce qui fait 6 en décimale. Et on a bien que 252+6 = 2 sur huit bits non signés.

L'utilisation de verrou pour la manipulation des compteurs est inutile. Premièrement, il n'y a qu'un seul écrivain par compteur (l'écrivain met à jour l'offset d'écriture et le lecteur met à jour l'offset de lecture) donc on aura jamais le cas ou deux entités veulent écrire en même temps dans un offset ce qui pourrait écraser une mise à jour et au lieu d'avoir une valeur d'offset à offset+a+b on aurait une valeur à offset+a ou offset+b. Deuxièmement, une lecture d'un offset depuis la mémoire partagée retournera toujours une valeur cohérente, qui a existé ou qui existe. On ne pourra jamais lire dans la mémoire partagée un offset qui contiendrait une valeur à la fois avec des bits d'une vieille valeur et des bits d'une mise à jour en cours. On lira soit la valeur avant l'écriture de la mise à jour soit la valeur après. Et cela n'est pas gênant, si la valeur de l'offset est lue avant la mise à jour, la mis à jour sera vue à la prochaine lecture de l'offset.

En ce qui concerne l'envoi sur le canal, dans un premier temps, l'écrivain passait en argument les adresses à envoyer et le nombre d'adresses à envoyer. Le maximum d'adresses étaient écrites dans le buffer du canal et le lecteur était notifié via un canal de notification de Xen. Si ce maximum était inférieur au nombre souhaité l'écrivain bloquait en attendant une notification du lecteur. Une fois cette notification reçue, il envoyait la suite des adresses sur le canal de communication et re-bloquait s'il n'avait toujours pas pu tout envoyer. Ce mécanisme se répétait jusqu'à ce que toutes les adresses passées en argument aient été envoyé. Pour la lecture, le lecteur lisait tout ce qui était disponible à la lecture au moment de l'appel de la fonction et si rien n'était disponible il bloquait en attendant une notification de l'écrivain. Une fois qu'il avait lu, il notifiait l'écrivain. Mais étant donné qu'il y a un ordonnancement au niveau de l'exécution des éléments dans Click, ces blocages au niveau de mini-os auraient pu être gênant lors de l'exécution du routeur Click. C'est pourquoi j'ai supprimé ces blocages dans les fonctions d'envoi et de réception de l'API et que ce mécanisme a été déplacé dans l'implémentation d'éléments Click l'utilisant.

Finalement, l'envoi sur le canal prend toujours en argument un pointeur vers les adresses à envoyer et le nombre d'adresses à envoyer mais il ne sera envoyé que ce qui est possible et la fonction retournera un pointeur vers la première adresse non envoyée ou NULL si tout à pu être envoyé. La lecture, elle, récupère toutes les adresses lues et renvoie un pointeur les contenant ou NULL s'il n'y avait rien à lire.

Plus précisément, la fonction d'envoi suit cet algorithme :

- Elle calcule la place disponible pour l'écriture dans le canal de communication, s'il y a moins de place que ce qui veut être écrit elle change la variable passée en argument pour la taille à ecrire par cette valeur calculée sinon cette variable reste inchangée.
- Elle calcule l'offset réel d'écriture dans le buffer du canal à partir du compteur d'écriture.
- Elle calcule la place contiguë disponible pour l'écriture pour savoir si elle peut tout ecrire à la suite dans le buffer où si l'écriture dans le buffer se fera à cheval entre la fin du buffer et le début du buffer.
- Elle écrit les adresses dans le buffer, met à jour le compteur d'écriture et retourne

La fonction de lecture suit cet algorithme :

- Elle calcule le nombre de données qu'il y à lire dans le canal, s'il n'y en a pas elle retourne.
- Elle calcule l'offset réel de lecture dans le canal.
- Elle calcule le nombre de données contiguë qu'il est possible de lire pour savoir les données chevauchent la fin et le début du buffer.
- Elle lit les données disponibles, met à jour le compteur de lecture et retourne.

Cependant, dans ces deux fonctions, il faut rajouter l'utilisation d'instructions servant de barrières mémoire pour empêcher une optimisation faite par les CPUs modernes. Ces derniers emploient le plus souvent un modèle d'exécution appelé out-of-order execution. Dans ce modèle d'exécution, un CPU n'exécutera pas les instructions dans l'ordre dans lequel elles se trouvent dans le programme mais en fonction de la disponibilité des données en entrée. Cela permet de ne pas gâcher de cycle CPU en attendant la fin d'une instruction avant de lancer les suivantes qui ne dépendent pas ce qui est entrain de s'exécuter. Dans mon cas d'accès à une mémoire partagée par deux unikernels cela peut poser des problèmes. Par exemple, la fonction d'envoi de données calcule le nombre de données qu'elle peut écrire et ensuite écrit ces données dans le canal. Cependant avec le modèle d'exécution du CPU décrit précédemment, lorsque l'écriture dans le canal se fait, le CPU à déjà toutes les données nécessaire pour mettre à jour l'offset d'écriture. Il ne va donc pas attendre que les données soient écrites dans le canal et il va lancer la mise à jour de l'offset d'écriture immédiatement. Seulement, si à ce moment là l'autre unikernel lance une lecture dans le canal, il va lire un offset d'écriture qui ne correspond pas à la réalité et il va considérer comme à lire une partie du canal avec des données en cours d'écriture. Il lirait donc des données corrompues. C'est la que les barrières mémoire entrent en jeu. Ce sont des routines assembleur qui sont là pour que tout le code avant une barrière s'exécute avant le code après la barrière. Ainsi, il faut mettre une barrière entre l'écriture des données dans le canal et la mise à jour de l'offset d'écriture et une barrière entre la lecture des données et la mise à jour de l'offset de lecture. Ces barrières mémoire ont uniquement un effet sur l'ordonnancement des instructions au niveau matériel. Or, les compilateurs peuvent eux aussi effectuer des optimisations et réordonner les instructions d'un programme. Cependant, par exemple, le compilateur gcc ne réordonnera pas d'instructions autour d'un code assembleur en ligne contenant les tags "volatile" ou "memory". C'est pourquoi souvent les barrières mémoire sont écrites avec le tag "memory".

Au moment de l'écriture de ce rapport, il reste un problème que je n'ai pas réussi à régler. Il se situe au niveau du mapping des pages de mémoire partagé. Je n'arrive pas à mapper plus de 495 pages mémoire simultanément dans un même unikernel, une fois cette limite atteinte, l'hypercall vers Xen pour mapper une page renvoie un statut d'erreur générale indéfinie. Cette limite est sans doute de 512 pages, car il faut ajouter aussi les pages mappée par ClickOS pour les utilitaires partagés avec Xen tel que le Xenstore ou les Grant tables. Cela limite donc la taille du pool de mémoire à 1mo alors que dans l'idéale, nous aimerions au moins pouvoir disposer de 2mo. Pour résoudre ce problème j'ai essayé de chercher des constantes dans le code source de Xen qui pourrait être à l'origine de cette limitation et j'ai essayé de modifier leur valeur en passant la valeur souhaité en argument à Xen lors de son boot via le fichier de configuration grub mais cela n'a rien changé.

### 5.3.4 Intégration dans Click

Une fois l'API finie dans mini-os, il a fallu l'utiliser pour écrire des éléments Click pour pouvoir lancer des instances d'unikernels ClickOS pouvant communiquer pour se transmettre des paquets. Pour cela, il fallait deux éléments qui seraient reliés par un canal de communication. Un élément

ToSHM qui se placerait en fin de chaîne de traitement de paquets d'un unikernel et qui écrirait dans le canal et un élément FromSHM qui se placerait en début de chaîne d'un autre unikernel et qui lirait depuis le canal. Ces éléments prennent en arguments les éléments nécessaires au mapping du pool de mémoire partagée et du canal de communication qu'ils vont utiliser. Ce mapping sera fait lors de l'appel à la fonction de configuration d'un élément. Ces deux éléments possèdent un port d'entrée et un port de sortie même si l'élément ToSHM n'utilisera pas son port de sortie et l'élément FromSHM n'utilisera pas son port d'entrée. Le port d'entrée d'un élément ToSHM et le port de sortie d'un élément FromSHM seront de type push. Pour réaliser ces éléments je me suis inspiré du fonctionnement de Click lorsqu'il fonctionne avec DPDK. Cependant, DPDK modifie la structure de paquets de Click pour s'y intégrer. Je n'ai pas incorporé à l'implémentation des paquets de Click l'utilisation de la mémoire partagée, les éléments de base ne savent donc pas qu'ils ont à faire à des paquets avec données partagées et ne sont pas capables de gérer la partie mémoire partagée. Cela explique notre limitation à certain type d'application et seuls mes éléments relatifs à la communication pourront créer des paquets et en supprimer en dehors bien sur des éléments qui prennent et rendent les paquets au réseau. Modifier l'implémentation des paquets de Click serait un travail futur à réaliser pour avoir une intégration complète, il n'a pas été fait car le temps que cela prendrait est trop conséquent par rapport aux six mois du stage.

Un élément ToSHM traitera les paquets qui lui arrivent et sera simplement ordonnancé par le flot des paquets. A chaque fois qu'un élément ToSHM sera ordonnancé, une fonction virtuelle C++ du code de l'élément sera appelée avec en argument un paquet venant de l'élément précédent. Cette fonction réalisera le travail qu'est censé effectuer l'élément au sein du routeur. Dans le cas d'un élément ToSHM cette fonction effectue les actions suivantes :

- Tout d'abord, elle vérifie le type du paquets qu'elle reçoit. Click définie un type `Packet` pour les paquets, mais pour la communication par mémoire partagée, il faut que les données soient soient stockées dans cette mémoire. Or, les paquets qui arrivent initialement dans le routeur par une source sont des paquets allouées par l'élément source donc les données ne sont pas forcément dans notre mémoire partagée (sauf si la source est un élément FromSHM). Pour palier cela, chaque élément ToSHM d'une chaine de communication entre unikernel vérifiera si le paquet contient déjà des données dans notre mémoire partagée ou non. Si ce n'est pas le cas, il allouera un buffer pour ces données et les copiera dedans. Dans click, lors de la création d'un paquet, il est possible de passer en argument une fonction qui sera appelée à la destruction du paquet pur libérer le pointeur de données. Ainsi, un élément ToSHM a juste à regarder si la fonction de destruction du pointeur de données correspond à la fonction que j'ai écrite et qui permet de libérer un buffer de mémoire partagée et il saura si c'est le premier élément ToSHM que ce paquet rencontre ou non.
- Ensuite, la fonction étant appelée à chaque paquet transféré, elle ne va pas directement envoyer les données dans le canal. Un élément ToSHM contient une file dont la taille est définissable en argument à la configuration de l'élément. Chaque paquet sera donc mis dans la file en attendant d'être envoyer sur le canal. Si la file est pleine, le paquet sera jeté si

l'élément à été défini comme non bloquant, sinon on ne sortira pas de cette fonction tant qu'il y aura de la congestion au niveau du canal et que la file sera pleine.

- Un élément ToSHM contient aussi un attribut qui lui indique par groupe de combien il doit envoyer les données dans le canal. Cet attribut est aussi paramétrable à la configuration et il doit être inférieur à la taille de file. Si le nombre en attente dans la file est supérieur ou égal à cet attribut ou si il y a congestion, on va écrire les données de la file dans le canal. On va écrire des données tant que l'écriture sur le canal écrit tout ce qu'on lui à donné en argument et que le nombre de données en attente dans la file est supérieur à 0.
- Finalement, comme le paquet en entrée ne va pas ressortir de l'élément, il est détruit avant de sortir de la fonction. Si le paquet en entrée était déjà dans la mémoire partagé, la fonction de destruction des données à été mise à NULL à la vérification pour ne pas libérer le buffer. De plus, juste avant d'appeler la destruction du paquet il faut mettre le pointeur de données à NULL dans le paquet car par défaut, si la fonction de destruction de données est NULL c'est l'opérateur delete[] qui est appelé et on ne le souhaite pas non plus.

Un élément FromSHM va lui effectuer son travail au sein d'une tâche car il n'y a personne pour lui donner des paquets, il doit les lire lui même depuis un canal de communication. A chaque exécution de cette tâche, l'élément va lire tout ce qui est disponible à la lecture et il va créer un paquet par donnée récupérée dans le canal. La fonction de destruction du pointeur de données du paquet que l'élément donnera en argument de la création du paquet indiquera que c'est déjà un paquet qui utilise de la mémoire partagée. Il va ensuite pousser chacun des paquets vers l'élément suivant du routeur.

# 6 Comunnication inter unikernel: test et benchmark

Les tests et benchmarks s'effectueront sur une machine avec les cartéristiques suivantes :

- CPU intel xeon 2.20Ghz avec 48 threads (2 sockets, 12 cores par sockect, 2 threads par core)
- 8go de RAM
- Xen 4.6.5

## 6.1 Test fonctionel

Au moment de l'écriture du rapport, l'intégration dans click n'est pas tout à fait terminée. Mais une fois cela fait, il faudra tester que tout fonctionne bien. J'ai donc déjà un test de prévu pour vérifier que des unikernels ClikcOS peuvent communiquer.

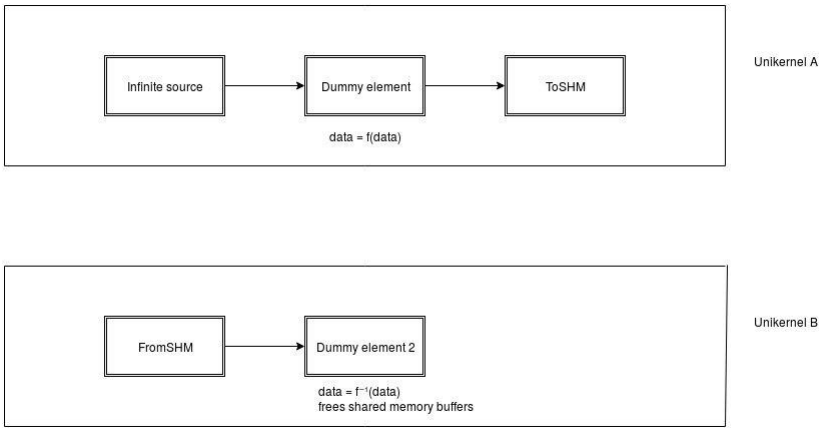


Figure 25: configuration du test fonctionel de l'implémentation.

Comme le montre la figure 25 ci-dessus, dans un premier unikernel, un premier routeur aura comme source un élément générateur de paquets et ces paquets auront comme données le string passé en argument dans le fichier de configuration à l'élément. Ces paquets seront passés à un élément fait uniquement pour ce test qui concaténera au string contenu dans les données du paquets la valeur d'un compteur qui comptera le nombre de paquets traités et le renversera. Il passera ensuite le paquet à un élément ToSHM qui l'enverra à un deuxième unikernel. Par exemple si la source génère des paquets avec le string "Hello" le premier paquet envoyé au deuxième unikernel contiendra le string "0olleH". Le deuxième unikernel exécutera un routeur qui va lire des paquets depuis un canal de communication avec un élément FromSHM et qui avec un deuxième élément uniquement défini pour ce test va renverser le string qu'il reçoit dans chaque paquet et l'afficher. Si tout se passe bien on affiche des strings contenant la donnée de départ passée à la source infinie avec la valeur du compteur et dans l'ordre dans lequel ils ont été envoyé (vérifiable grâce au compteur).

## 6.2 Benchmarks

Une fois l'implémentation terminée et testée, il faudra bien évidement faire des benchmark pour avoir une idée réelle des performances de l'implémentation par rapport à ce qui existe déjà.

### 6.2.1 Plateformes

Indépendamment des scénarios du test, ces benchmarks serviront à comparer les performance sur trois plateformes différentes :

- Deux unikernels ClickOS sur la même machine qui utilisent les éléments standard ToDevice et FromDevice pour communiquer.
- Deux routeurs Click utilisant DPDK et s'exécutant sur le même Linux en mode utilisateur et utilisant donc les éléments FromDPDKDevice et ToDPDKDevice pour la source et la sortie de paquets des routeurs.
- Deux unikernels ClickOS sur la même machine qui utilisent les éléments ToSHM et FromSHM que j'ai implémenté.

Le but étant de comparer les performances pour une communication sur une même machine, on ne veut donc pas passer par le réseau. C'est pourquoi pour le cas des tests sur ClickOS utilisant FromDevice et ToDevice il faudra mettre par exemple un bridge entre les deux interfaces sondées par les éléments pour court-circuiter le réseau. Il faudra faire de même pour le cas avec DPDK mais nous ne savons pas si cela est possible donc dans le pire des cas, les paquets passeront par le réseau même si cela ne donnera pas un comparatif totalement juste.

### 6.2.2 Scénarios

Pour l'instant, nous avons défini deux scénarios pour ces benchmark. La figure 26 présente la configuration d'un premier scénario de benchmark.

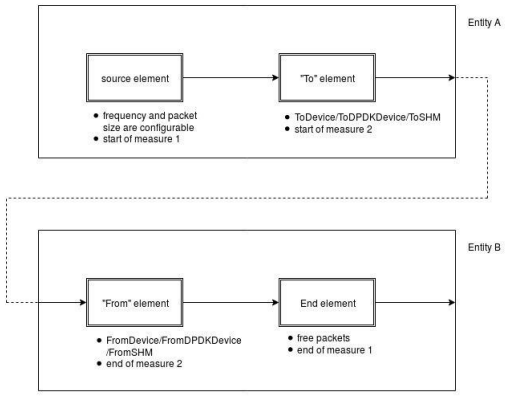


Figure 26: configuration du premier scénario.

Dans les trois cas du test, dans le premier routeur Click, un élément que j'aurais écrit sera utilisé comme générateur de paquets. Ils seront ensuite envoyés à l'élément qui les passera au prochain routeur: ToDevice, ToDPDKDevice ou ToSHM selon la plateforme utilisée pour le test. Dans la deuxième entité, les paquets seront récupérés par l'élément adéquat selon la plateforme et seront envoyés à un élément que j'aurais écrit et qui lira le contenu des paquets puis les libérera.

Deux mesures seraient prises pendant ce test : une mesure du temps d'envoi et de réception des paquets et une mesure du temps qui contiendra le temps d'envoi/reception plus le temps de lecture/écriture dans les paquets. La première mesure sera donc prise pendant le test entre l'élément To de la première entité et l'élément From de la deuxième entité et permettra de mesurer les performances pures de la communication. La deuxième mesure sera prise entre l'élément source de la première entité et l'élément de libération des paquets de la deuxième entité et permettra en plus d'avoir des indications sur l'influence des caches sur la lecture/écriture dans les paquets.

Un deuxième scénario de benchmark serait d'effectuer le même test entre trois entités. On peut voir sa configuration sur la figure 27.

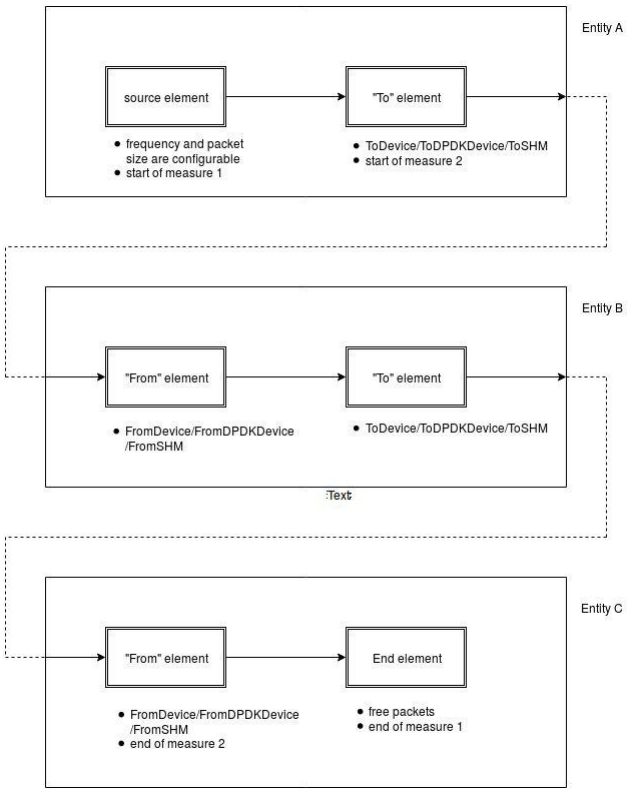


Figure 27: configuration du second scénario.

Dans cette configuration, on rajoute simplement une réception et un envoie en plus en intercalant une entité supplémentaire entre les deux du test précédent qui ne fera que From en entrée et To en sortie. Les mêmes mesures seront effectuées pour ce benchmark: cette fois-ci, la première mesure sera prise entre l'élément To de la première entité et l'élément From de la troisième entité et la deuxième mesure, elle, sera prise entre l'élément source de la première entité et l'élément de libération des paquets de la troisième entité.

Dans chacun des scénarios, pour chacune des mesures, je prendrai la valeur moyenne, le minimum, le maximum et la variance.

### 6.2.3 Variables

Les benchmark comporteront deux variables mais pour avoir un apercu clair des choses il sera sans doute mieux d'en faire varier une seule à la fois. La première variable sera la taille des paquets. En effet c'est une caractéristique primordiale d'un paquet et on veut bien sur savoir quel impact cette taille peut avoir lors de la transmission de paquets sur une même machine. La deuxième variable sera la fréquence d'émission des paquets de la source qui simule l'arrivée de paquet dans notre fonction réseau. En effet, selon le type de service que l'on va gérer, la fréquence de réception de paquets ne sera pas la même et nous voulons donc couvrir plusieurs cas d'application. Par exemple dans les cas d'application de type voix sur IP (voIP) ou de streaming, les paquets sont reçus à des fréquences plus ou moins régulières et linéaires, alors que dans le cas d'application comme du cloud Radio Acces Network (cloud RAN) la fréquence de réception des paquets suit plutôt une loi du type loi de Poisson.

### 6.2.4 Répartition matérielle

Il faudra aussi réaliser ces tests dans différentes conditions d'exécution sur la machine pour voir l'influence des CPUs (changement de contexte, cache) sur la communication entre les entités. Pour cela il faudrait effectuer chaque test dans chacune des trois conditions suivantes :

- Les deux entités communicantes sont sur la même socket et sur le même core.
- Les deux entités communicantes sont sur la même socket mais sur un core différent.
- Les deux entités communicantes sont sur une socket différente (et sur un core différent mais si elles ne sont pas sur la même socket cela va de soit)

# 7 Conclusion

Au moment de l'écriture de ce rapport, il me reste encore du travail à effectuer afin de pouvoir faire les benchmarks présentés pour la communication par mémoire partagée. Pour ce qui est des travaux qui pourraient s'effectuer dans la continuité de mon travail, il y a tout d'abord la modification de la structure des paquets click dans clickOS pour y intégrer la mémoire partagée. Ensuite, mon implémentation concerne uniquement le passage de paquets d'un unikernel à l'autre mais il faudrait aussi amélioré la la partie lecture/envoie des paquets sur le réseau en incluant un support pour DPDK dans Xen et dans ClickOS (à la fois dans mini-os et dans Click).

Dans l'équipe ou j'ai effectué mon stage, leur travail s'est porté sur l'utilisation de ClickOS pour une implémentation de cloud RAN. La configuration d'un routeur Click effectuant le même travail sur les paquets qu'une implémentation matérielle à été écrite. Le but sera de scinder cette configuration en plusieurs configurations plus petites pour les intégrer chacune dans un unikernel, d'utiliser mon travail de communication par mémoire partagée pour connecter les unikernels et d'utiliser le travail de mon co-stagiaire sur l'ordonnancement des unikernels pour le binding des instances sur les CPUs de la machine. Si tout se passe bien, une démonstration interne à Orange devrait être effectuée.

Durant mon stage, j'ai pu participer à un projet ambitieux et d'envergure beaucoup plus importante que tout ce que j'ai pu avoir à faire lors de mon cursus informatique. En plus des connaissances que j'ai pu acquérir d'un point de vue informatique, cela m'a permis de découvrir l'évolution du développement d'un projet voué à terme à prendre une place importante dans la gestion du réseau dans le monde des télécommunications et d'acquérir une certaine méthodologie pour la gestion d'un projet à échelle plus grande que ce que j'avais pu rencontrer jusqu'à présent. Ce stage m'a permis de voir plusieurs aspects dans la construction d'un projet : une partie de réflexion et d'étude des technologies du domaine lié au projet pour trouver une voie à suivre et à creuser ; mais aussi la partie d'implémentation d'une solution puis la réalisation de benchmarks pour en vérifier les performances. J'ai bien évidemment pu approfondir les connaissances que j'avais sur la virtualisation, les hyperviseurs et tout ce qui est gestion et fonctionnement des systèmes bas niveau (au niveau du CPU, de la mémoire, ect...) en étant en contact avec des outils existants dans ces domaines, en essayant de les comprendre, en essayant de les utiliser et d'y intégrer de nouvelles fonctionnalités. J'ai également acquis des connaissances sur tout ce qui tourne autour des réseaux notamment sur le transport des paquets sur le réseau et le traitement des paquets au plus bas niveau dans le matériel (carte réseau) et dans le noyau d'un système d'exploitation. Lors de l'étude du code sources des projets que j'ai utilisé ou lors de mon implémentation de mon API de canal de mémoire partagée j'ai aussi acquis des connaissances nouvelles et découvert des mécanismes nouveau en programmation, surtout pour le langage C.

Ce stage m'a conforté dans mon envie de poursuivre ma carrière professionnelle dans l'informatique bas niveau et les systèmes et à renforcé mon goût pour la programmation en C/C++.

# References


[1] site de référence sur les unikernels. www.unikernel.org.

[2] multiples auteurs. Unikernels: Library operating systems for the cloud. March 2016.

[3] Russel Pavlicek. *Unikernels: Beyond Containers to the Next Generation of Cloud*. O'Reilly, October 2016.

[4] site du projet mirageos. https://mirage.io.

[5] site du projet rumprun. www.rumpkernel.org.

[6] Antti Kante et Justin Cormack. Rump kernels: No os? no problem! October 2014.

[7] site du projet clickos. http://cnp.neclab.eu/clickos/.

[8] multiples auteurs. Clickos and the art of network function virtualization. April 2014.

[9] multiples auteurs. Osv: Optimizing the operating system for virtual machines. June 2014.

[10] site du dpdk. www.dpdk.org.

[11] site de référence pour netmap. http://info.iet.unipi.it/ luigi/netmap/.

[12] netmap dans clickos. http://cnp.neclab.eu/vale.

[13] site du projet click modular router. http://read.cs.ucla.edu/click/.

[14] multiples auteurs. The click modular router. July 2000.

[15] site du projet xen, avec le wiki dédié. https://www.xenproject.org.

[16] documentation sur xen. https://xenbits.xen.org/docs/4.4-testing/.

[17] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall, November 2007.

[18] multiples auteurs. Shared-memory optimizations for inter-virtual-machine communication. November 2015.