Mémoire d'ingénieur

Mise en place d'une infrastructure d'expérimentation pour l'évaluation de performances de solutions de virtualisation de fonctions réseaux appliquées au déploiement du protocole Named-Data-Networking

Xavier MARCHAL

Année 2014–2015

Stage de fin d'études réalisé dans l'entreprise LORIA

en vue de l'obtention du diplôme d'ingénieur de TELECOM Nancy

Maître de stage : Thibault CHOLEZ

Encadrant universitaire : Dominique MERY

Déclaration sur l'honneur de non-plagiat

Je soussigné(e),

Nom, prénom : MARCHAL, Xavier

Élève-ingénieur(e) régulièrement inscrit(e) en 3e année à TELECOM Nancy

Numéro de carte de l'étudiant(e) : 1202007632k

Année universitaire : 2014–2015

Auteur(e) du document, mémoire, rapport ou code informatique intitulé :

Mise en place d'une infrastructure d'expérimentation pour l'évaluation de performances de solutions de virtualisation de fonctions réseaux appliquées au déploiement du protocole Named-Data-Networking

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Villers-lès-Nancy, le 5 septembre 2015

Signature :

Mémoire d'ingénieur

Mise en place d'une infrastructure d'expérimentation pour l'évaluation de performances de solutions de virtualisation de fonctions réseaux appliquées au déploiement du protocole Named-Data-Networking

Xavier MARCHAL

Année 2014–2015

Stage de fin d'études réalisé dans l'entreprise LORIA

en vue de l'obtention du diplôme d'ingénieur de TELECOM Nancy

Xavier MARCHAL
2, allée des roses
54230, Chaligny
06 80 38 43 94
xavier.marchal@telecomnancy.eu

TELECOM Nancy
193 avenue Paul Muller,
CS 90172, VILLERS-LÈS-NANCY
+33 (0)3 83 68 26 00
contact@telecomnancy.eu

LORIA
615, Rue du Jardin botanique
54600, Villers-lès-Nancy
03 83 58 17 50

Maître de stage : Thibault CHOLEZ

Encadrant universitaire : Dominique MERY

Remerciements

Je souhaite tout d'abord remercier mon tuteur de stage, Monsieur Thibault CHOLEZ, enseignant-chercheur au LORIA, pour m'avoir accepté en tant que stagiaire et m'avoir guidé tout au long de ce stage.

Je tiens aussi remercier le personnel de l'equipe MADYNES pour avoir pu répondre a certaines de mes interrogations et plus spécialement Monsieur Jérôme FRANCOIS, chercheur à l'Inria, pour m'avoir mis a ma disposition son matériel informatique en attendant la réception de celui nécessaire au bon déroulement du projet à mener durant mon stage.

Table des matières

1 Introduction

1.1 Contexte

Au cours de ces dernières années la demande des utilisateurs sur l'Internet a considérablement augmenté. Avec toujours plus de services et les volumes de données toujours plus grands, les opérateurs de réseaux sont contraints de faire évoluer en permanence leurs réseaux. Mais les investissements de ces réseaux sont confrontés à de nombreuses contraintes. Premièrement, les matériels réseau vendus par les équipementiers, comme par exemple CISCO, sont souvent conçus pour une utilisation spécifique d'un matériel spécifique ce qui rend très coûteux leurs intégration dans les réseaux. Ensuite, avant de déployer de nouvelles technologies pour répondre à cette demande croissante, les opérateurs doivent examiner attentivement leurs possibilités et les revenus qu'ils pourraient en tirer mais ce processus peut prendre du temps. Cependant une nouvelle tendance dans le domaine des réseaux a émergé au cours des dernières années : NFV (Network Functions Virtualization).

L'European Telecommunications Standards Institute (ETSI) a posé les bases du standard début 2013 et celui ci est encore en cours de spécification (actuellement en phase 2). NFV est une approche qui mise sur le concept de la mise en œuvre de fonctions réseau sous forme de logiciels qui peut fonctionner sur tous types de serveurs. Cette technologie apporte de nombreux avantages pour les opérateurs de réseaux comme une réduction des coûts d'installation et de maintenance ou encore permettre une plus grande rapidité de déploiement.

1.2 le projet DOCTOR

C'est dans ce contexte que le projet DOCTOR [8][15] a lieu. DOCTOR, DeplOyment and seCurisaTion of new functiOnalities in virtualized networking enviRonnements, est un projet de recherche visant à favoriser le déploiement de nouvelles solutions réseaux dans des infrastructures virtualisées en définissant des solutions de sécurité et supervision des équipements de réseaux virtualisés et en réalisant un PoC (Proof-of-Concept). DOCTOR est composé de 4 acteurs majeurs :

- le LORIA (aspect recherche)
- l'UTT (aspect recherche)
- Montimage (solutions de monitoring)
- Orange (fournisseur d'accès)
- Thales (solutions de sécurité)

qui forment le consortium du projet et chaque partenaire industriel a des attentes différentes en ce qui concerne la finalité du projet :

— Pour Orange avoir une infrastructure réseau virtualisée permettra de déployer plus facilement de nouvelles solutions réseaux innovantes et ainsi offrir de nouvelles opportunités à ses clients.
— Montimage étendra sa solution de supervision MMT avec les résultats du projet relatifs à la supervision, la détection d'attaques ou encore les analyses de performances, permettant d'offrir des solutions personnalisées dans la supervision des environnements virtualisés.
— Thales intégrera les résultats du projet dans leur offre Cyber Operational Centers (CYBELS), comme les études sur les nouvelles vulnérabilités relatives aux environnements virtualisées.

L'équipe du projet se concentrera sur les nouvelles technologies réseaux, comme NFV et les réseaux centrés sur le contenu (ICN), qui se veulent révolutionnaires sur la manière de penser des réseaux d'aujourd'hui notamment ceux de l'Internet. Les réseaux ICN n'étant pas encore implantés dans les réseaux actuels ceux ci seront utilisé sous forme d'*usecases* de nouveaux services bas niveau et leurs implémentations sera facilité par la virtualisation. L'étude de ces technologies se fera via la mise en place d'un testbed réel mis à disposition à des utilisateurs réels (étudiants des partenaires universitaires). C'est ce testbed qui va rythmer les différentes phases du projet car celui-ci a plusieurs objectifs :

— déployer un réseau virtuel
— déployer une couche protocolaire ICN
— étudier la coexistence IP/ICN
— collecter des traces d'utilisation
— éprouver les solutions de monitoring et de sécurité des partenaires

Le but de mon stage de fin d'étude se concentre sur la mise en place d'un réseau virtuel et de la mise en place de la couche protocolaire ICN mais aussi sur le dimensionnement et la mise en place du serveur qui servira au testbed du côté du LORIA. Ainsi dans ce rapport je développerai dans une première partie l'état de l'art sur le protocole NDN, qui fait partie de la familles des ICN et a été sélectionné par le consortium du projet DOCTOR pour la mise en place du testbed, ainsi que de NFV. Dans une seconde partie, je ferai une étude comparative des différentes solutions de virtualisation sélectionnées pour le testbed, puis continuerai sur des tests de performances du protocole NDN. Ensuite j'expliquerai mes choix de dimensionnement du serveur accueillant le testbed ainsi que la démarche de la mise en place de son réseau virtuel.

2 Présentation de l'entreprise

2.1 Inria

L'Inria [10], Institut national de recherche en informatique et en automatique, a été créé en 1967 suite au Plan Calcul lancé par le général de Gaulle en 1966. Le but du Pan Calcul est d'assurer à la France son indépendance en matière de super-ordinateurs. L'Inria est rattaché au ministère de l'enseignement supérieur et de la recherche et au ministère en charge de l'industrie. L'Inria est composé de 8 centres de recherche et est présent sur tout le territoire français :

- Rocquencourt (siège social)
- Rennes
- Sophia Antipolis
- Grenoble
- Nancy
- Bordeaux
- Lille
- Saclay

L'Inria est composée de 173 équipes-projets regroupant 2700 collaborateur dont une grande majorité de scientifiques et de doctorant, avec un budget initial de plus de 200 millions d'euros par an. Ces équipes sont à l'origine de plus de 4500 publications scientifique par an et sont amenées à travailler conjointement avec des entreprises du privé et des instances étrangères. Le domaine de recherche de l'Inria s'etend autour de cinq domaines :

- Santé, biologie et planète numérique
- Mathématique appliquées, calcul et simulation
- Perception, cognition, interaction
- Réseaux, Systèmes et services, calcul distribué
- Algorithmique, programmation, logiciel et architecture

2.2 LORIA

Le LORIA [13], Laboratoire Lorrain de Recherche en Informatique et ses Applications, est situé au 615 rue du jardin botanique à Villers-lès-Nancy en Lorraine. le LORIA est une unité mixte de recherche créé en 1997 qui est commune à trois établissements :

- Le CNRS
- l'Université de Lorraine
- L'Inria

Le domaine d'application du LORIA est la recherche fondamentale et appliquée des sciences informatiques. Cette recherche est effectuée par 30 équipes, dont une partie est commune à l'Inria et regroupe plus de 450 personnes. Ces équipes sont réparties en cinq grand départements :

- Algorithmique, calcul, image et géométrie
- Méthodes formelles
- Réseaux, systèmes et services
- Traitement automatique des langues et des connaissances
- Systèmes complexes et Intelligence artificielle

Pour donner une idée de la répartition du personnel du LORIA voici les chiffres donnés en fin d'année 2013 :

- 114 enseignants-chercheurs
- 60 chercheurs
- 14 personnels administratifs
- 102 doctorants
- 23 post-doctorants
- 4 ATER
- 2 chercheurs contractuels
- 27 ingénieurs scientifiques

Pour plus d'information sur l'organisation du LORIA, un organigramme est prensent en annexe.

2.3 L'équipe MADYNES

Lors de mon stage, j'ai intégré l'équipe MADYNES [11][12][14]. Cette équipe est commune à l'Inria et au LORIA et est actuellement dirigée par Mme. Isabelle CHRISMENT. L'équipe fait partie du département Réseaux, systèmes et services, et ce concentre plus spécifiquement sur la conception et la mise en place de nouveaux paradigmes et de méthodes de supervision et de sécurité des communications. Cette recherche se fait principalement sur des réseaux dynamiques de grande envergure :

- Les réseaux de capteurs
- Les réseaux sans-fil auto organisés
- Les réseaux IPv6
- Les réseaux centrés sur le contenu
- Les réseaux pair-à-pair à grande échelle

3 État de l'art

3.1 Named Data Networking

3.1.1 Principes généraux

Named Data Networking (NDN) [1][3][4], fait partie de la famille des réseaux ICN (information-centric networking). Ce nouveau paradigme est porté par Van Jacobson dont les premières publications datent de 2006 [5]. Les réseaux ICN ont une approche différente de celles que l'on utilise aujourd'hui (principalement IP). Ces types de réseaux ont comme différence essentielle de mettre l'information au centre des communications (content-centric) en lieu et place des machines (host-centric) avec leurs notions de connections point-à-point. NDN a été pensé avant tout pour un usage Internet et se veut une solution pour réduire l'engorgement de ce type de réseau car la majorité du trafic Internet actuel est dû à des données répliquées et à leur routage entre serveurs et clients. En brisant la barrière des connexions point-à-point NDN permet de répondre en une seule fois à plusieurs demandes d'une même information et utilise des stratégies de cache sur les routeurs pour permettre non seulement la réémission de ces mêmes données lors d'une demande ultérieure mais aussi pour garder ces données au plus proche des utilisateurs.

Le protocole NDN se place au niveau de la couche 3 (réseau) et 4 (transport) du modèle OSI, car il n'y a plus de notion de connexion point-à-point (couche 3 du modèle OSI). Ici la résolution se fait via une chaîne de caractères sous forme d'URI qui contient des NDO (Name Data Object) espacés par des "/". Un nom NDN est composé d'un préfixe (qui décrit la ressource) suivi de champs optionnels comme le numéro de version ou d'un numéro de segment (voir figure 3.1) dans le cas de données plus grandes que la taille maximale d'un paquet NDN (8800 octets). Pour les reconnaître ces champs sont respectivement précédés par les codes hexadécimal 0xFD et 0x00.

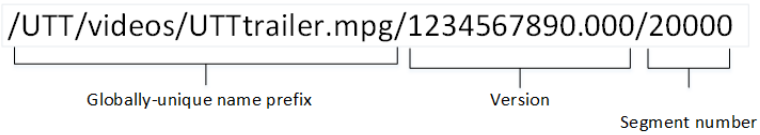


Figure 3.1 – Exemple de nom NDN [4]

Un NDO fonctionne de manière arborescente et peut aussi bien servir à router les paquets NDN (car chaque nom NDN peut avoir une entrée dans la table de routage avec comme racine "/" qui fait office de route par défaut) qu'à spécifier le comportement d'un programme, en déclarant un serveur avec un NDO donné (exemple : "/youtube") et par exemple le faire suivre de "watch" ("/youtube/watch/intitulé_de_la_vidéo") pour l'envoi de vidéos aux clients ou encore "upload"

("/youtube/upload/intitulé_de_la_vidéo") pour la réception de vidéos sur le serveur. Comme le nom NDN est hiérarchique, il est possible de répondre à une demande avec un nom NDN plus précis. Par exemple pour un streaming live en cours, une requête sur le nom "/twitch/stream/kanemochi" aurait une réponse comme celle-ci "/twitch/stream/kanemochi/20150919120000/15672".

Pour transmettre des données le protocole NDN se base sur deux types de paquets (figure 3.2) : les paquets *Interest* et les paquets *Data*. Les paquets *Interest*, qui permettent d'effectuer une requête, doivent être au minimum composés d'un nom NDN et d'une nonce. La nonce est un nombre de 32 bits qui permet pour un nom donné d'avoir un paquet *Interest* unique. Le principal intérêt de cette propriété est d'éviter au paquets *Interest* d'effectuer des boucles. Il existe aussi, en plus du nom et de la nonce, deux champs supplémentaires optionnels : les sélecteurs et les guides. Les sélecteurs permettent de préciser la requête avec notamment la variable *MustBeFresh* qui précise lorsque la réponse à cette requête a déjà été envoyée et est encore présente dans le cache des routeurs si l'on veut accepter que la donnée soit expirée ou non. Les guides servent à spécifier le comportement des routeurs avec ces paquets, par exemple la variable *InterestLifeTime* spécifie combien de temps le router va garder l'entrée de ce paquet avant de la supprimer.

Pour répondre aux paquets *Interest* le protocole prévoit un autre paquet nommé *Data*. Celui-ci reprend le même champ de nom NDN pour permettre l'envoie de la donnée au client, les autres champs lui étant propres. Le champ *MetaInfo* regroupe des variables permettant de décrire le contenu présent dans le paquet dont, par exemple, *FreshnessPeriod* qui indique combien de temps, en millisecondes, la donnée et considérée comme valide, le champ *Content* contient les données et le champ Signature contient la signature qui peut être pour le moment soit RSA(-2048 par défaut) soit SHA-256 ou ECDSA ainsi que des informations supplémentaires sur cette signature comme où aller chercher la clé public, etc... Ainsi en signant chaque paquet plutôt que de créer un tunnel sécurisé, on ne cherche plus à avoir une connexion de confiance mais un contenu de confiance.

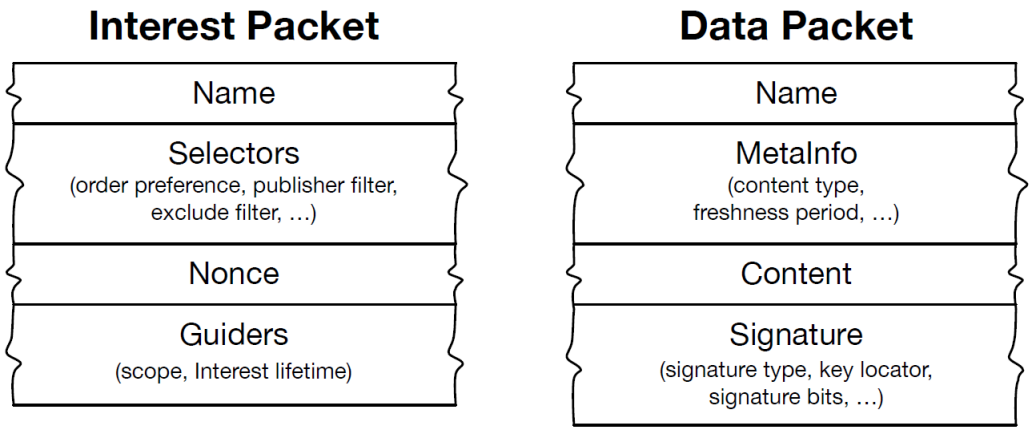


FIGURE 3.2 – Spécification des paquets NDN [3]

3.1.2 Fonctionnement d'un nœud NFD

Pour router les noms NDN, les développeurs ont mis à disposition un programme sous forme de daemon, nommé NFD (NDN Forwarding Daemon), qui va se charger du routage NDN. Celui-ci est constitué de 3 tables (voir figure 3.3)) :

— le *Content Store* (CS), C'est dans cette table que toutes les réponses qui ont pu satisfaire une requête sont stockées. Elle peut être gérée à l aide de différentes stratégies de cache

comme par exemple Least Recently Used (LRU). De plus, ce cache est présent dans chaque nœud NDN.

— la *Pending Interest Table* (PIT), C'est dans cette table que sont regroupées toutes les requêtes en attente de réponse. Il existe une entrée pour chaque préfixe en attente ainsi si deux requêtes sur le même préfixe sont reçues, NFD va enregistrer dans une liste toutes les faces par lesquelles ces requêtes sont arrivées pour pouvoir renvoyer par la suite la réponse par ces même faces. Une face (virtuelle), à ne pas confondre avec interface (physique), est une route entre deux programmes (serveur-NFD, client-NFD ou NFD-NFD). Un préfixe peut avoir plusieurs faces qui peuvent pointer sur une ou plusieurs interfaces. Le nombre de face n'est pas limité par interface.

— la *Forwarding Information Base* (FIB). C'est dans cette table que le daemon stock les différentes routes disponibles. De la même manière que la PIT, la FIB utilise comme clé le préfixe :

1. enregistré par le serveur via la fonction *registerPrefix*,
2. manuellement
3. ou via le protocole de routage NDN, NLRS (*Named Data Link State Routing Protocol*)

et comme valeur une liste des faces disponibles pour ce préfixe. Les préfixes peuvent être concaténés, à l'instar des sous-réseaux dans le monde IP, pour réduire la taille de la table. NFD gère chaque préfixe indépendamment grâce a une stratégie de routage qui lui permettra de choisir parmi les faces disponibles la ou lesquelles celui-ci va forwarder le paquet *Interest*.

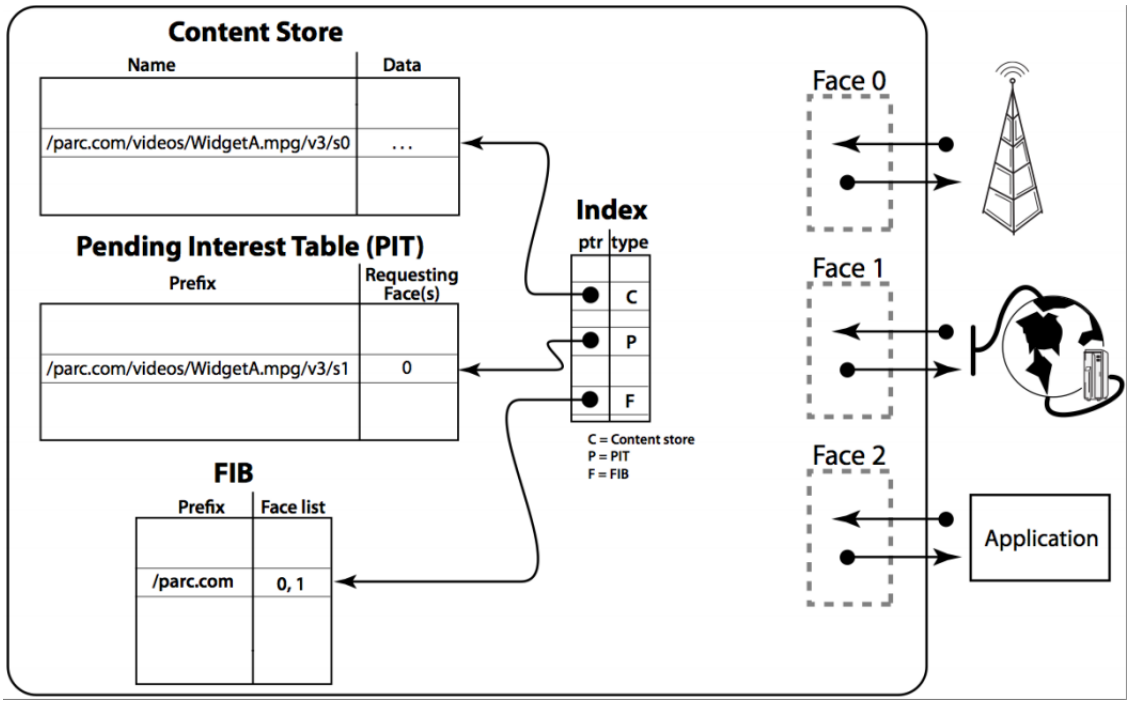


FIGURE 3.3 – Routage NDN [3]

Chacune de ces tables joue un rôle précis durant le processus de routage (voir figure 3.4) :

1. lors de la réception d'une requête (*Interest*), NFD va d'abord regarder dans le CS pour voir si il n'y a pas déjà eu une réponse à cette requête. Si c'est le cas alors il renvoie la donnée en cache, sinon il continue le processus de routage.
2. NFD va enregistrer le préfixe de la requête dans la PIT puis continue le routage.

3. NFD va regarder dans la FIB si une entrée existe pour ce préfixe. Si c'est le cas alors il va forwarder la requête au(x) serveur(s) ou prochain(s) routeur(s).
4. après être arrivé au serveur, celui-ci forge une réponse (*Data*) et l'envoie au daemon.
5. le daemon va ensuite vérifier si le préfixe de la réponse correspond à une entrée de la PIT. Si au moins une entrée correspond alors le daemon cachera la réponse puis la forwardera à toutes les faces par où sont venues les requêtes.

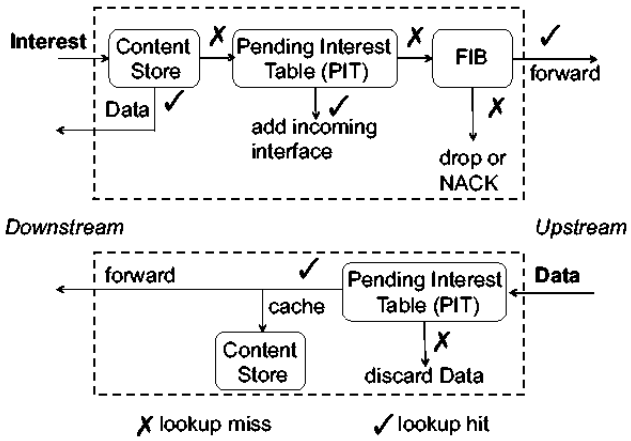


FIGURE 3.4 – Forwarding NDN [9]

3.2 Network Functions Virtualisation

NFV [17] est un paradigme, paru en 2013, qui se veut une alternative aux équipementiers de matériel réseau, car il existe de nombreux équipements réseaux et chacun a un usage spécifique. Le principe de NFV est de pouvoir chainer des composants, appelés VFN, qui regroupent une ou plusieurs fonctions réseau de base pour pouvoir créer un ensemble formant un service. En fait, il existe déjà des applications capablent de reproduire des fonctions réseaux comme les bridges ou l'ip forwarding couplé avec la table de routage de la machine mais cela reste très basique et il faudra encore quelques années pour que des entreprises développent des logiciels aussi personnalisables que ce que l'on peut trouver dans les matériels comme, par exemple, ceux de CISCO. De plus NFV ne s'arrête pas à l'application, il apporte tout un principe basé sur la virtualisation de ces composants ainsi que leur orchestration. Le fait de tout regrouper sur une même machine physique a l'avantage d'être plus efficace dans la gestion des ressources et ainsi permettre une réduction des coûts matériels et de maintenance (CAPEX et OPEX). De plus le fait de pouvoir chaîner facilement des fonctions réseau permet aux entreprises de pouvoir proposer des services plus rapidement sur le marché et de répondre plus vite aux variations de charge. En suivant les technologies candidates identifiées par le projet DOCTOR, qui cherche à étudier les solutions libres et open-source de virtualisation, nous nous concentrerons sur deux des solutions les plus utilisées à savoir OpenStack et Docker.

3.2.1 OpenStack

Openstak a été créé en 2010 par la NASA et Rackspace Hosting. C'est un logiciel de virtualisation gratuit et open-source spécialisé dans le *Cloud Computing*. OpenStack est un agrégat de

plusieurs modules, chacun étant dans un projet indépendant et chaque année des conférences sont organisées pour pouvoir définir les futures lignes directrices de l'ensemble du programme. Chaque module a sa fonction, par exemple le module Nova gère les machines virtuelles, Neutron le réseau, Cinder le stokage, etc... l'ensemble forme donc un hyperviseur de type 1 aussi appelé *"bare metal"* (voir figure 3.5) principalement basé sur QEMU + KVM. OpenStack implémente déjà quelques agents NFV comme des routeurs, des DHCPs, loadbalancer, firewalls etc.. mais ceux-ci ne laissent pas encore beaucoup de place à la personnalisation. OpenStack est une API REST, ce qui veut dire que toutes les actions que l'on peut faire sont représentés par des lien HTTP ce qui permet de facilement créer des applications tierces. Même si OpenStack est une solution très complète de *Cloud Computing* celle-ci est relativement complexe et demande un niveau d'expertise important pour bien pouvoir la prendre en main.

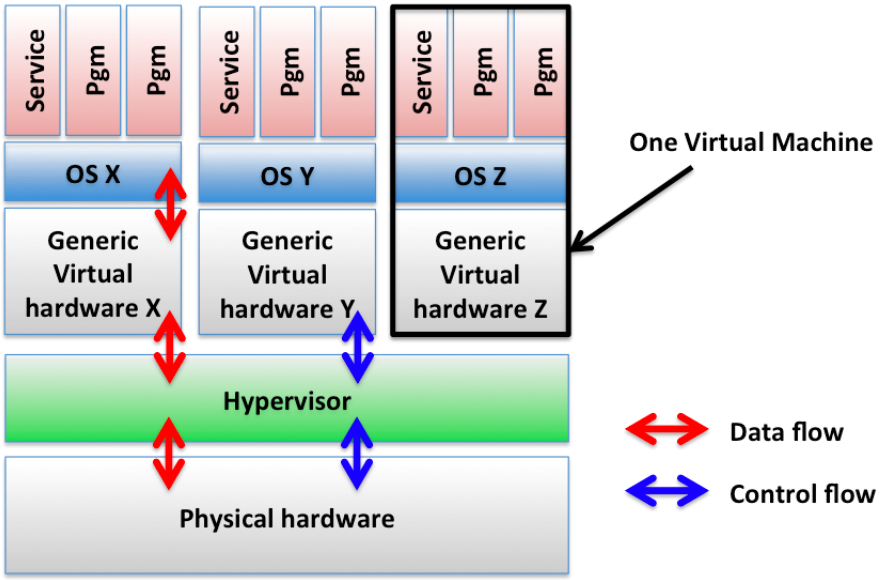


Figure 3.5 – Type-1 hypervisor virtualization [4]

3.2.2 Docker

Docker est une plate-forme qui permet le déploiement d'applications dans des conteneurs virtuels et fait partie de la catégorie des hyperviseurs de type 0 (voir la figure 3.6), il fournit une API de haut niveau qui offre une solution de virtualisation pour exécuter des applications isolées les unes des autres. Docker utilise des conteneurs qui utilisent directement le noyau et appels système pour fonctionner ainsi tout les programmes exécutés dans les conteneurs sont visibles par le système hôte mais le contraire n'est pas vrai. En effet le conteneur est placé à l'intérieur d'un *namespace* qui va restreindre la visibilité du conteur et celui-ci ne pourra accéder qu'aux ressources associées à ce *namespace*.

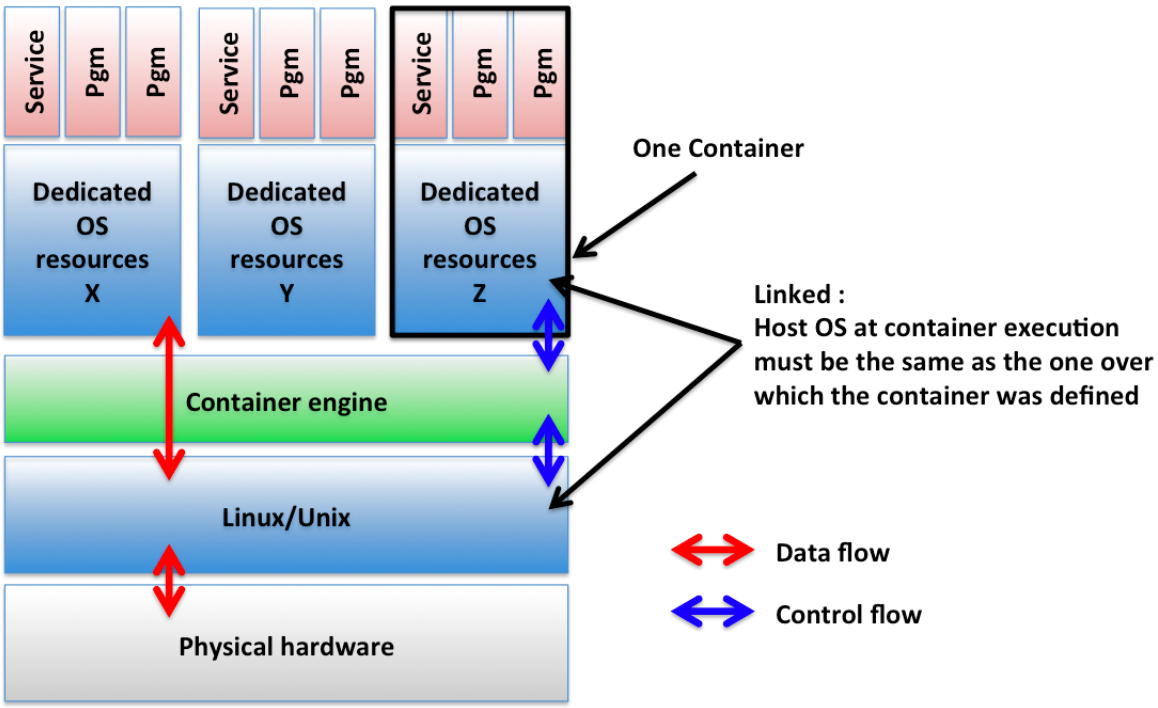


FIGURE 3.6 – Type-0 hypervisor virtualization [4]

Le principal avantage de cette technologie est qu'elle est légère car il n'a pas besoin de système d'exploitation supplémentaire pour la virtualisation du fait que les conteneurs sont basés sur le noyau de la machine hôte. Cela permet d'obtenir des performances encore plus proche du système native que les solutions de virtualisation *"bare metal"*.

3.3 Synthèse et constat

NDN est un protocole prometteur, centré sur l'information qui peut réduire la congestion dans les infrastructures réseaux, principalement celle de l'Internet, en regroupant tous les mêmes demandes des clients (*Interest*) dans une même entrée. De cette façon, les serveurs n'enverront qu'une seule fois la réponse (*Data*) à ces demandes et les routeurs se chargerons d'envoyer cette réponse aux différents clients ayant émis la demande mais aussi mettront en cache cette réponse, avec une certaine durée de vie donnée par le serveur, pour qu'ils puissent directement répondre aux futures demandes qui correspondraient à cette réponse à la place du serveur. De cette façon, les données, principalement les contenus statiques (image, vidéo, etc.), seront toujours au plus proche des clients et n'auront plus besoin d'être envoyés de multiples fois par les serveurs. le protocole NDN met en avant l'aspect sécurité des données car celui ci veut dans sa spécification que le paquet *Data*, qui contient les données, soit signé ; ce qui permettra de renforcer la sécurité des connexions. Mais le protocole est encore jeune car celui-ci est encore en cours de développement (nécessite un programme tiers pour le routage des paquets) et ses spécifications peuvent encore changer.

NFV apporte un concept révolutionnaire dans le domaine des réseaux. Ce standard est encore en phase de spécification (actuellement phase 2) mais les bases du concept on déjà été définies et des logiciels basés sur ce concept ont déjà pu voir le jour comme par exemple Open vSwitch. Du coté des grands acteurs de la virtualisation, le développement de solutions NFV est déjà en cours avec par exemple VMware qui à sorti récemment un programme additionnel à sa solution vSphere

(ESXi) [16] ou encore Microsoft qui devrait proposer prochainement sa solution NFV avec la sortie du prochain hyperV (windows server 2016) [7]. Mais le monde du libre n'est pas en reste par exemple OpenStack, une des solution de virtualisation ayant la plus grande communauté, de son côté propose déjà des VNF comme des routeurs, firewalls, load-balancers, etc... Pour le moment Docker semble plus attiré par l'ajout de fonctions d'orchestration (Docker Machine, Swarm and Compose). Cependant il existe tout de même quelques conteneurs spécialisés dans les fonctions réseaux comme le conteneur "socketplane/openvswitch" qui permet d'avoir un switch virtuel. Mais comment savoir quelle technologie de virtualisation est la plus adaptée pour implanter des Virtualized Network Functions ?

4 Comparaison des solutions OpenStack et de Docker

4.1 Analyse du problème

Lors de ce chapitre nous allons comparer les deux solutions de virtualisation énoncées précédemment à savoir OpenStack et Docker. Le but est de les comparer autant au niveau des performances que de leur mise en place.

Avant toute chose voici une présentation des différentes machines qui ont été mises à ma disposition pour mes tests car, dans le cas d'évaluation de performances, il est important de connaître le matériel utilisé pour pouvoir reproduire et/ou comparer les résultats :

1. Ma station de travail que l'on nommera *workstation* :
   - CPU : Intel core i5 4590 @ 3.3GHz
   - RAM : 16GB (4x4) DDR3 @ 1600MHz
   - Ethernet : Broadcom BCM5722 PCIe
   - OS : Linux Mint 17 Qiana
   - Kernel : 3.13
   - Software :
     - Docker v1.6.0
     - OpenStack via Devstack
     - iperf
2. Le premier serveur, emprunté à M. Jérôme FRANÇOIS, que l'on nommera *server1* :
   - CPU : 2x Intel xeon E5-2420v2 @ 2.2GHz
   - RAM : 48GB (6x8) DDR3 @ 1600MHz
   - Ethernet : 4x Broadcom BCM5720 PCIe (1Gbps)
   - OS : Ubuntu server 15.04
   - Kernel : 3.19
   - Software :
     - Docker v1.7.0
     - iperf
3. Le second serveur, emprunté à M. Jérôme FRANÇOIS, que l'on nommera *server2* :
   - CPU : Intel xeon E5-2420v2 @ 2.2GHz
   - RAM : 32BG (4x8) DDR3 @ 1600MHz
   - Ethernet : 4x Broadcom BCM5720 PCIe (1Gbps)
   - OS : Ubuntu server 15.04
   - Kernel : 3.19
   - Software :
     - Docker v1.7.0
     - iperf

Dans les deux cas il nous est offert la possibilité d'installer les solutions de virtualisation via des scripts. Au niveau d'OpenStack, nous avons utilisé le script devstack (https://github.com/OpenStack-dev/devstack) qui va installer tout ce dont nous avons besoin pour faire fonctionner un environnement OpenStack. Par défaut ce script n'a pas besoin de fichier de configuration mais il est quand même recommandé d'en faire un car l'installation par défaut ressemble plus à ce que l'on pourrait trouver dans les solutions de virtualisation *desktop* comme VirtualBox (Oracle). L'installation d'OpenStack est aussi très longue avec une vingtaine de minutes d'installation. De plus l'installation est très sensible il suffit d'oublier une chose ou avoir un conflit (qui peut venir de l'installation elle même) pour tout recommencer. Cette façon d'installer OpenStack n'est pas faite pour un environnement de production car il est conseillé pour certains modules de les installer sur des serveurs dédiés pour avoir une installation optimale mais dans notre cas, cela compliquerait vraiment l'installation au vu de ce pourquoi nous le faisons à savoir faire des tests de performance réseaux. De plus, cette installation n'est pas très encline au redémarrage et lorsque cela arrive c'est un peu "la roulette russe" pour que tout fonctionne de nouveau correctement, souvent une réinstallation est de mise.

Dans le cas du script d'installation de Docker le script n'a besoin d'aucun fichier de configuration et s'installe toujours vite et bien. Dans le cas de la *workstation*, mint 17 n'avait pas le paquet apparmor qu'il nous a fallu installer avant d'utiliser Docker.

4.2 Déploiement d'un réseau

4.2.1 OpenStack

Comme OpenStack inclut beaucoup de modules pour fonctionner correctement et que ceux-ci sont tous exécutés sur la même machine, on se retrouve avec une consommation CPU non négligeable juste en lançant l'hyperviseur (voir figure 4.1). On peut voir que l'hyperviseur consomme environ 14% du temps CPU de notre *workstation*.

FIGURE 4.1 – Coût CPU initial d'OpenStack

Avec le module Horizon (une interface web), OpenStack est relativement simple à prendre en main car il fait la liaison avec l'API à notre place. Ainsi nous avons pu construire notre premier réseau virtuel grâce à cette interface (voir figure 4.2) :

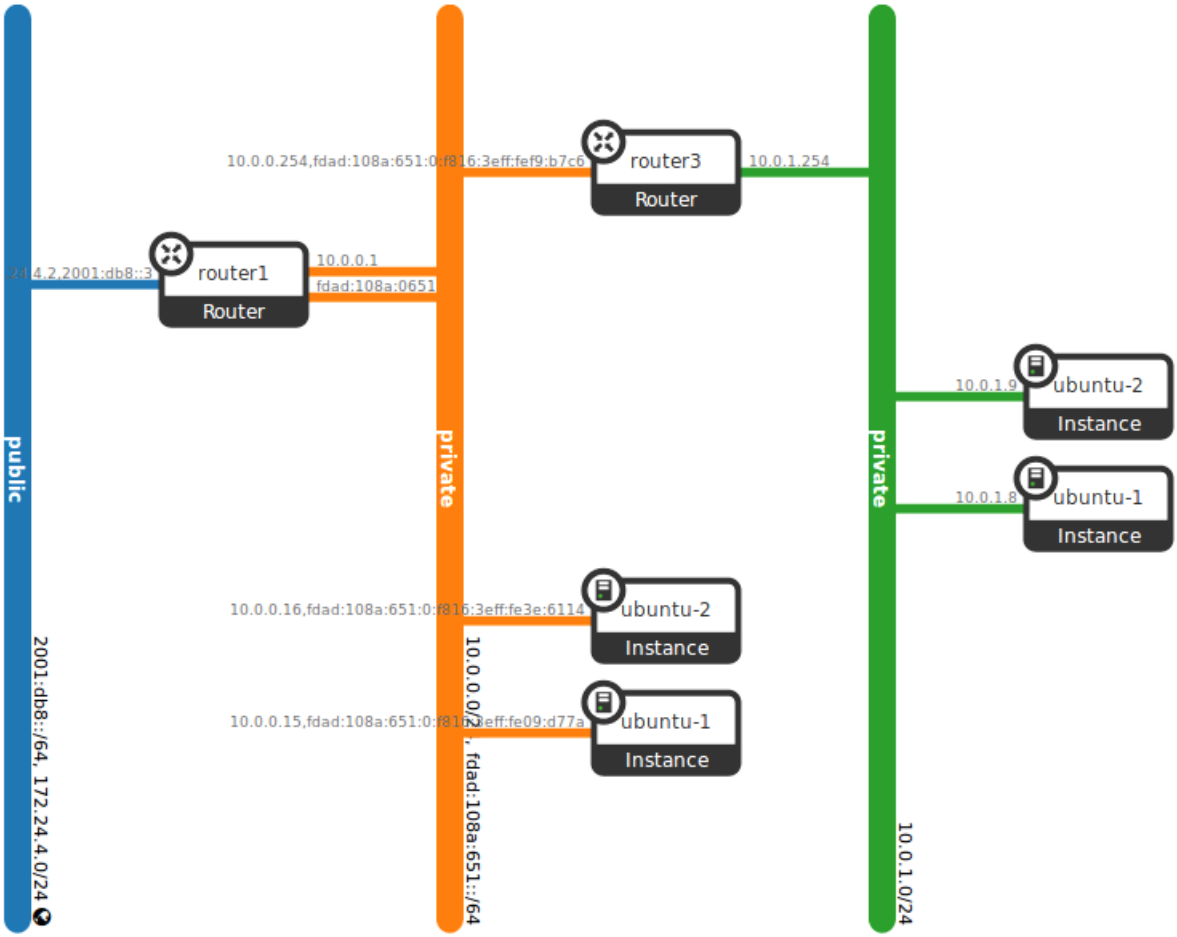


FIGURE 4.2 – Réseau virtuel sous OpenStack

Ce réseau se veut simple avec deux machines virtuelles par sous réseau reliés entre elles par un routeur. Pour pouvoir installer les machines virtuelles, il m'a fallu ajouter une règle NAT pour pouvoir communiquer avec l'extérieur car je n'ai pas pu avoir de plages d'adresses pour faire mes tests et OpenStack est prévu pour avoir un sous-réseau disponible pour celui nommé "public". Les sous-réseaux et les routeurs sont gérés par OpenStack et sont ce que l'on appelle des agents ; ils sont plus légers que des machines virtuelles. Les sous-réseaux ont la particularité de définir les règles DHCP, DNS et les routes à annoncer. La gestion des routeurs est un peu particulière car l'interface web ne permet pas de modifier les routes des routeurs et par défaut ceux-ci ne savent router que les réseaux reliés à leurs interfaces. Pour pouvoir modifier les règles de routage, il faut le faire directement via l'API en lignes de commande ou par requêtes HTTP mais ces deux méthodes sont bien plus complexes pour la configuration des composants de l'hyperviseur.

4.2.2 Docker

Contrairement à OpenStack Docker n'a pas d'interface de configuration mais son contrôleur est assez léger en paramètres, ce qui permet de mieux s'y retrouver. Pour créer un conteneur avec Docker il faut avoir une image de conteneur. Ces images sont un peu comme des clichés instantanés d'un système de fichiers et l'on peut soit la créer, soit télécharger l'image sur une sorte de

git d'image Docker avec la commande : $ docker pull nom_d'image[:tag_de_version].

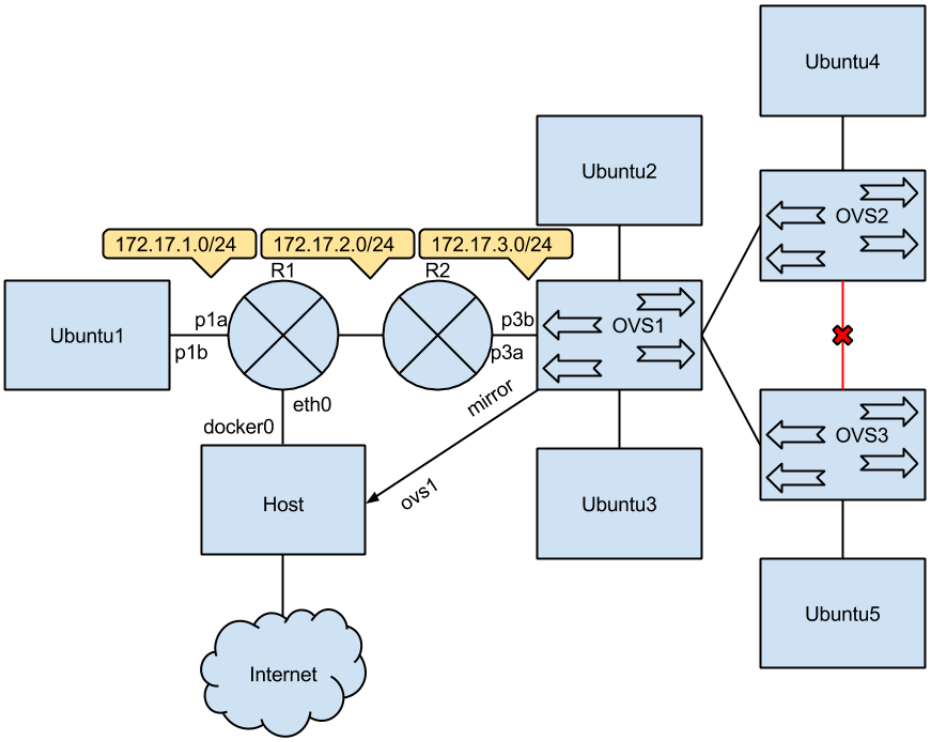


FIGURE 4.3 – Réseau virtuel sous Docker

La figure 4.3 montre un des réseaux que j'ai eu l'occasion de mettre en place durant ma phase d'expérimentation de cette solution de virtualisation. Lorsque l'on crée un conteneur celui-ci est par défaut relié au bridge docker0 qui permet aux conteneurs de communiquer entre eux, avec l'hôte et l'extérieur via un NAT. Mais avoir un seul grand sous-réseau n'est pas forcément très intéressant ne serait ce que pour isoler certains conteneurs d'autres. Docker utilise les *namespaces* qui permettent de restreindre la visibilité qu'à un processus à son *namespace* au lieu du système global. Ainsi il est par exemple possible de créer des interfaces virtuelles (veth) et de lier ses extrémités à deux *namespaces* pour qu'ils puissent communiquer entre eux (exemple de p1a/p1b et p3a/p3b). Créer ces interfaces virtuelles peut prendre pas mal de temps mais les commandes nécessaire pour les créer, les attribuer et les configurer peut facilement être scriptées car celles-ci ne sont que des commandes système (voir Annexe D).

Ces interfaces virtuelles nous permettent d'avoir une grande liberté sur la conception de nos réseaux car il n'y a théoriquement pas de limite sur le nombre d'interfaces virtuelles que peut avoir un système, par contre les conteneurs présentent un gros défaut : lorsqu'un conteneur est arrêté (arrêt du processus) le *namespace* est perdu, les ressources allouées n'y seront plus et les interfaces virtuelles, qui ne sont pas persistantes, seront détruites les autres seront restituées au système global (*namespace* 0).

Dans la figure 4.3 j'ai voulu reconstituer les différents types de connections que l'on peut retrouver dans un réseau simple à savoir des connexions routeur-routeur, routeur-switch, switch-switch, machine-routeur et machine-switch. Pour créer des routeurs j'ai utilisé directement la capacité qu'offre Linux à router les paquets IP en activant la variable *ip_forwarding* du noyau Linux et en utilisant la table de routage pour pouvoir donner les règles de routage. Pour créer un switch j'ai utilisé un logiciel supplémentaire nommé Open vSwtich qui réutilise les bridges Linux et est une surcouche à ceux-ci permettant de les manipuler avec plus de facilité comme par exemple

pour cloner les paquets traversant le conteneur et les rediriger vers un port miroir spécifique dans l'objectif de faire du monitoring (exemple Wireshark).

4.3 Évaluation des performances des deux solutions

Une interface virtuelle fonctionne de la même façon en interne que l'interface de Loopback. Pour donner une idée de la capacité maximum que peut avoir un lien virtuel de notre *workstation* nous avons conduit dans un premier temps des tests de performance sur l'interface de Loopback. Ces tests serviront de référence lors de mes prochains tests sur les deux solutions de virtualisation pour constater les différences de performances avec une solution native. lors de cette section les tests sont effectués avec le logiciel iperf. Le but de ce logiciel est d'atteindre le débit le plus élevé possible sur un réseau IP, pour ce faire ce logiciel n'escalade pas toute la pile protocolaire du modèle OSI et se limite à la couche 4 (transport) pour minimiser les éventuelles pertes dues aux couches supérieures. Dans cette partie nos tests réalisés avec le logiciel iperf durent 30 seconds et sont effectués dix fois chacun avant de calculer la moyenne des résultats. Le premier test consiste à créer une connexion iperf sur la Loopback :

- serveur : $ iperf -s 127.0.0.1
- client : $ iperf -c 127.0.0.1 -t 30 -i 10

Le résultat de ce premier test (voir figure 4.4) nous montre que pour une connexion l'interface de Loopback est capable de fournir environ 60 Gbps de données. Il faut savoir que iperf n'alloue qu'un seul thread par connexion donc, dans ce test nous n'utilisons pas toute la capacité de notre processeur. En effet, lors de test faisant varier le nombre de connexions simultanées la *workstation* a pu atteindre un débit moyen d'environ 83 Gbps avec 3 connexions simultanées. Au delà nous observons une diminution des performances de l'interface de Loopback.

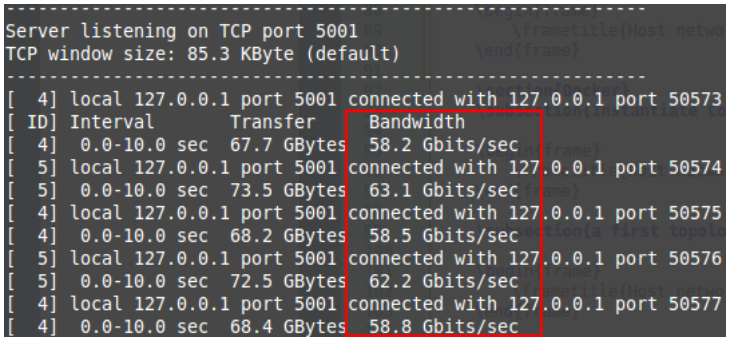


FIGURE 4.4 – Simple test de performance sur l'interface de Loopback

Pour en apprendre un peu plus sur le comportement interne de l'interface de Loopback, nous avons effectué un deuxième un test de débit symétrique :

- serveur : $ iperf -s 127.0.0.1
- client : $ iperf -c 127.0.0.1 -t 30 -i 10 -d

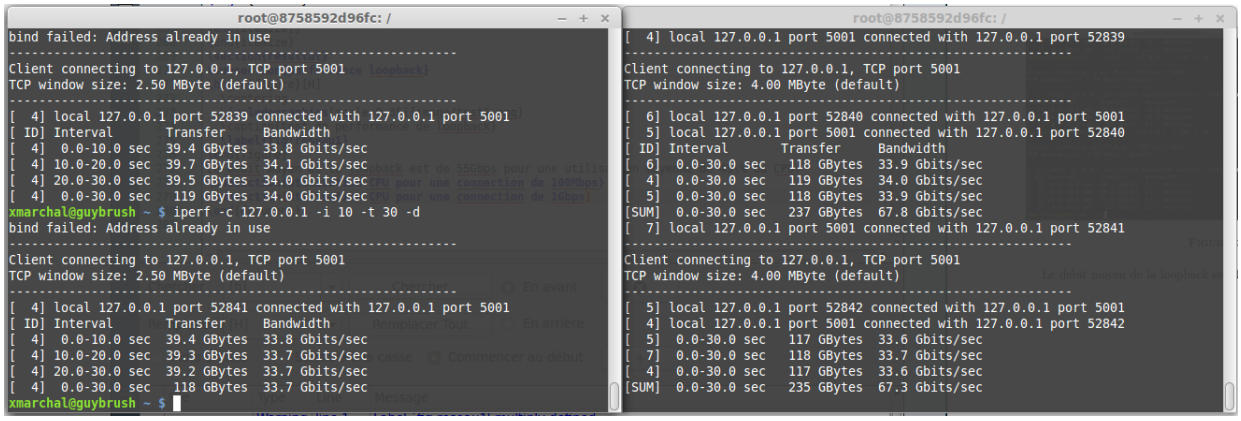


FIGURE 4.5 – Test de performance d'une connexion symétrique

les résultats de ce test (voir figure 4.5) ont montré que l'interface de Loopback de notre *workstation* avec une connexion symétrique est capable d'atteindre 34 Gbps dans les deux sens pour un total de 68 Gbps. Mais ce test ne peut pas être comparé directement avec le premier car pour réaliser ce test le logiciel iperf a ouvert une connexion dans chaque sens il faut donc comparer ce résultat à un dernier test de la Loopback pour lequel on créera deux connexion simultanées dans le même sens :

— serveur : $ iperf -s 127.0.0.1
— client : $ iperf -c 127.0.0.1 -t 30 -i 10 -P 2

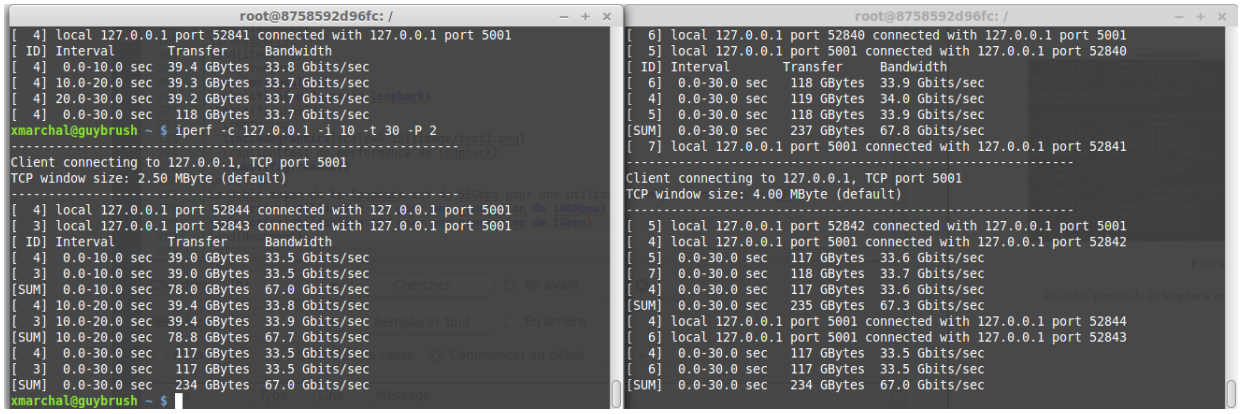


FIGURE 4.6 – 2 connections loop-back performance test

Lors de ce dernier test (voir figure 4.6) avec deux connexions simultanés, on retrouve le même débit que le résultat précédent à savoir 34 Gbps pour chaque connexion. Cela permet de mettre en évidence une chose : c'est que l'interface de Loopback n'est pas géré comme une interface full-duplex mais plus comme une interface half-duplex avec probablement une implémentation proche d'une queue FIFO.

Maintenant que nous avons eu un premier aperçu des performances de l'interface de Loopback nous allons pouvoir commencer les tests de performances des différentes solutions abordées précédemment à savoir OpenStack et Docker. Nous avons construit pour chacune des solutions un réseau linéaire (voir figure 4.8). On peut comparer ce type d'implémentation aux polymères linéaire que l'on peut retrouver en chimie et qui est composé d'une brique de base (monomère) que l'on va répéter plusieurs fois (voir figure 4.7).

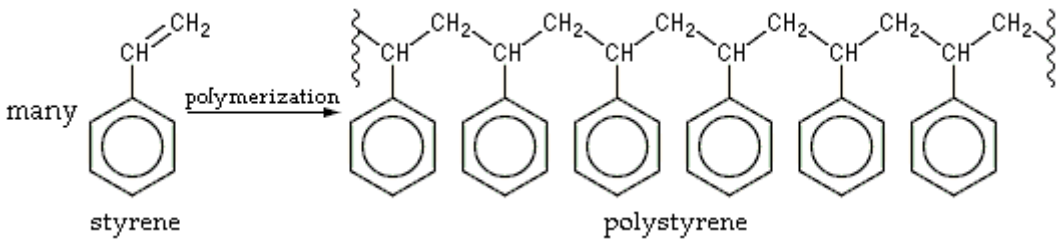


FIGURE 4.7 – Exemple d'un polymère (droite) et de son monomère (gauche) [2]

De la même façon, pour réalisé nos tests, nous allons commencer avec deux conteneurs ou machines virtuelles côte à côte puis nous allons les espacer de plus en plus en rajoutant ce que l'on va appeler un *NETWORK BLOCK* dont la structure interne varie selon la solution. Pour Docker il existe plusieurs façon de relier des conteneurs selon les capacités que l'on leur a attribué. Ici nous avons testé deux cas :

1. le premier avec des conteur faisant office de switches, même ci cela n'est pas fondamentalement utile car un conteneur peut fournir tout un sous réseau comme le fait OpenStack mais il est toujours bon de voir si il existe une différence de performance entre un routage L2 et L3.
2. Le deuxième avec des conteurs faisant office de routeurs (comme des switches L3).

Pour OpenStack, l'équivalent des switch n'existe pas vraiment car un sous réseau gère toute la plage d'IP comme une seul entité et il n'est pas possible de les chaîner. Pour ce qui est du routage pour OpenStack, un *NETWORK BLOCK* correspondra à un agent routeur plus un sous-réseau.

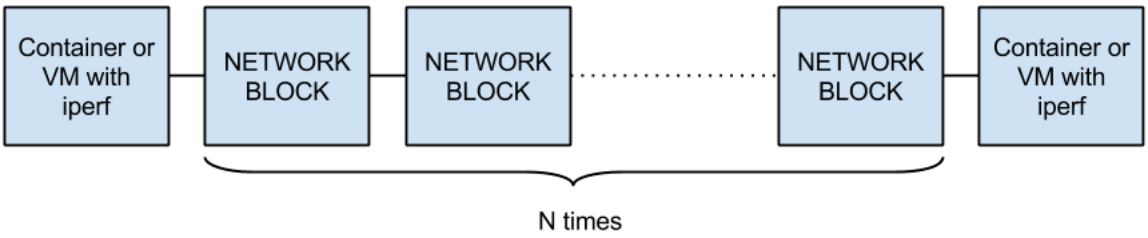


FIGURE 4.8 – Architecture du réseau en chaine

Pour ce test nous avons gardé les mêmes façons de router que précédemment, pour Docker à savoir le routage L2 est réalisé par Open vSwitch 2.0.2 et le routage L3 par l'activation de l'ip forwarding du noyau Linux couplé avec sa table de routage. Pour OpenStack nous utilisons les agents fournis par le module Neutron. la figure 4.9 montre les différents relevés de débit dans les trois situations présentées précédemment en faisant varier le nombre de *NETWORK BLOCK* de 0 à 4. on peut voir que dans le cas d'une connexion directe entre deux conteneurs le débit enregistré est presque le même que dans le cas de la solution native. Ensuite, que ce soit avec des conteneurs switches ou des conteneurs routeurs, on observe une perte de performance non linéaire. Cela pourrait s'expliquer par le fait d'ajouter des intermédiaires augmente la latence et cette latence a une influence inversement proportionnelle sur le débit ($dbit\_max = \frac{taille\_de\_la\_fenetre\ (en\ bits)}{RTT\ (en\ seconds)}$). Concernant la différence entre les débits L2 et L3 cela peut s'expliquer par le fait que par rapport au modèle OSI traiter la couche L3 nécessite d'avoir déjà traité les couches précédentes. Pour OpenStack on retrouve un débit trois fois inférieur par rapport à Docker lors d'une communication de deux machines virtuelles côte à côte. Cette différence se réduit au fur et à mesure

que l'on rajoute des *NETWORK BLOCK* et l'on atteint une différence de 2 fois avec 4 *NETWORK BLOCK*. Nous avons poussé un peu plus la courbe de débit pour les conteneurs de type routeur et le constat est qu'il faut 10 *NETWORK BLOCK* pour obtenir les performances d'OpenStack avec deux machines virtuelles côte à côte avec des débits de l'ordre de 20 Gbps.

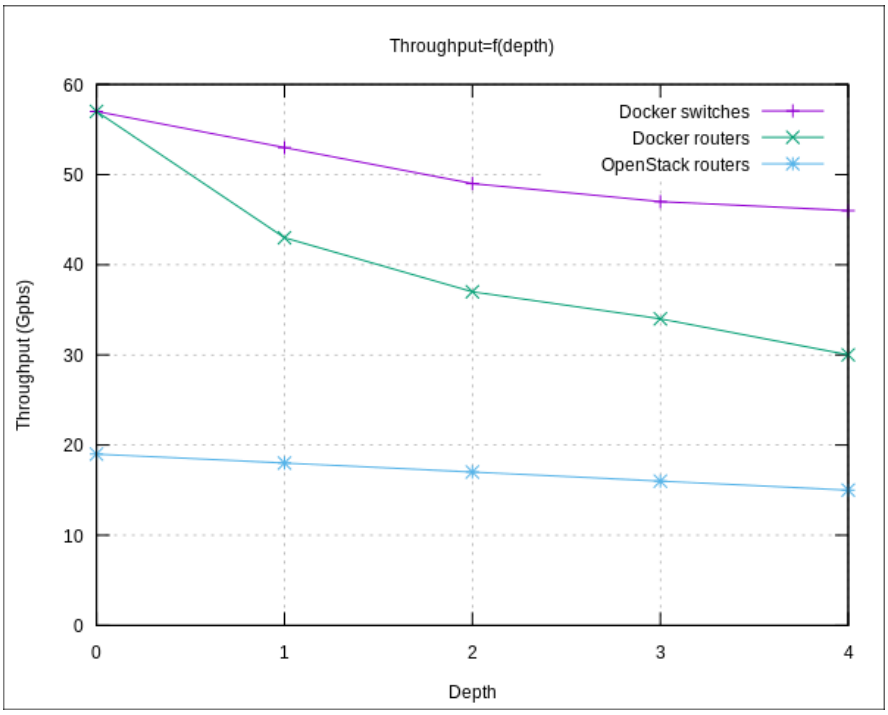


FIGURE 4.9 – Débits atteint pas les différentes solution de virtualisation

1 connexion :

	Docker	OpenStack
Throughput (Gbps)	~58	~20
CPU usage	2c@100%	4c@80%

2 connexions :

	Docker	OpenStack
Total Throughput (Gbps)	~73	~20
CPU usage	4c@90%	4c@88%

3 connexions :

	Docker	OpenStack
Total Throughput (Gbps)	~75	~20
CPU usage	4c@95%	4c@93%

TABLE 4.1 – Débit et coût des solutions durant des tests iperf en parallèle avec 0 *NETWORK BLOCK*

La table 4.1 nous montre la charge du système lorsque que une ou plusieurs connexions sont actives. On peut voir dans le cas d'OpenStack que, pour une seule connexion, que le système à

déjà une charge très importante. Cela est en grande partie dû au module Neutron et c'est pour cela qu'il est normalement conseillé de l'installer sur un serveur dédié séparé des autres modules. Au niveau du débit total Docker reste proche des mesures que l'on avait fait sur l'interface de Loopback mais pour OpenStack, étant donné que le system éprouve déjà une forte charge, le débit global reste inchangé. Ce test a été refait sur le *server2* et j'ai pu constater que le débit global augmentait mais que le débit maximum par connexion était toujours limité au même ordre de grandeur que dans la figure 4.9.

Pour finir sur cette comparaison de performance entre Docker et OpenStack, j'ai réalisé un dernier test sur le coût relatif des solutions lorsque celles-ci ont besoin d'effectuer des communications avec le monde extérieur à la machine sur laquelle les conteneurs/machines virtuelles sont hébergé(e)s via une interface Gigabit Ethernet. La figure 4.10 nous montre le résultat de ces tests. On peut voir que lorsqu'un conteneur directement relié avec l'hôte communique avec l'extérieur la consommation relative du processeur est quasiment la même que si c'est l'hôte qui avait effectué cette communication. Si ce même conteneur est derrière quatre switches ce coût est 20% plus important que la solution native et 60% plus important si celui ci est derrière 4 routeurs. Pour OpenStack si la machine virtuelle est directement rattachées a l'hôte, le coût en temps processeur de la communication est 3 fois plus importante et ce coût augmente à 3,2 fois si celui-ci est derrière 4 routeurs.

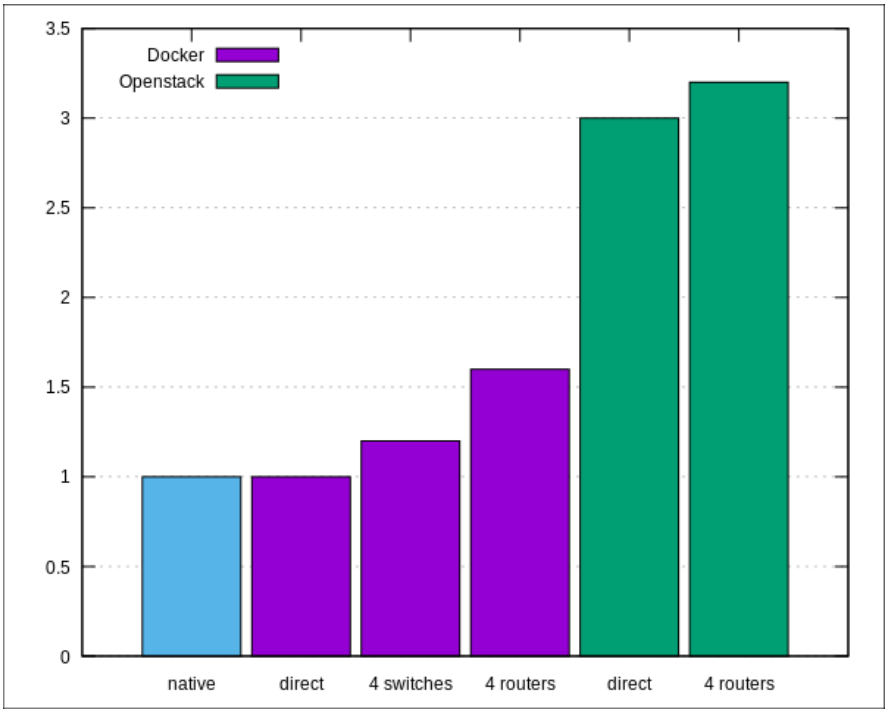


FIGURE 4.10 – Coût en temps processeur relatif des différentes solution a travers l'interface de l'hôte

4.4 Conclusion

Pour conclure sur cette étude comparative entre OpenStack et Docker, sur le plan de la mise en place et le maintien du réseau les deux solutions ont leurs atouts. D'un côté OpenStack est très complet et propose tout ce qu'un administrateur système a besoin pour gérer un service de *Cloud Computing* et est facile à prendre en main pour les taches de base grâce a l'interface web proposé

par le module Horizon mais celle ci ne gère pas encore tout ce dont est capable OpenStack ce qui veut dire que dès que l'on veut effectuer des modifications plus spécifiques les actions se feront via des commandes passées à l'API et cela demande un certain niveau d'expertise si l'on veut pourvoir faire cette configuration sans trop de problème. De l'autre côté Docker propose beaucoup moins de personnalisation au niveau de l'API mais laisse une plus grande ouverture pour la configuration de ces conteneurs, dont la majorité des actions se reposent sur des fonctions du noyau Linux, ce qui nous permet, selon moi, d'offrir un plus grand niveau de personnalisation qu'OpenStack. Mais cette personnalisation a un prix car il faut tout faire à la main ce qui peut être long. Il faut donc prévoir l'écriture de scripts de configuration pour simplifier cette tâche. Au final le domaine d'application de ces deux solutions ne sont, selon moi, pas les même de part leur façon de faire. Je vois plus OpenStack pour un environnement de production car la solution est robuste et quelle contient divers modules capables par exemple de faire de l'orchestration, etc... Docker serait plus adapté à un environnement de développement car on peut assez rapidement mettre en place des réseaux plus complexes qu'OpenStack mais sera plus limité dans un domaine de production car Docker n'est pas encore vraiment prévu pour tout ce qui est variation de charge, n'a pas d'orchestrateur avancé et demandera donc un travail supplémentaire à un administrateur système pour maintenir cette solution.

Au niveau des performances Docker a un net avantage de part le fait qu'il soit un hyperviseur de type 0. Cela lui permet d'utiliser le noyau du système hôte, ce qui limite le nombre intermédiaires entre les applications et le matériel et ne pas dépendre d'un système virtualisé avec des pilotes génériques. Ainsi Docker obtient des performances très proches de l'hôte pour un coût en ressources réduit car il ne consomme que ce que ces applications et services ont besoin. Par rapport aux solutions traditionnelles, Docker permet de faire de la virtualisation sur des machines moins imposantes et permet ainsi une réduction des coûts matériels par rapport aux autres types d'hyperviseurs.

5 Etude de performance d'NDN

5.1 Analyse du problème

Cette étude de performance du protocole NDN découle de la mise en place d'un réseau NDN virtualisé servant à la mise en place de la première version de la *gateway* NDN/HTTP d'Orange où j'ai remarqué que les performances (le débit dans le réseau NDN) de la *gateway* n'étaient pas optimales. j'ai donc entrepris de chercher les causes de ce défaut de performances et proposé des améliorations.

La *gateway* NDN/HTTP, développée par Orange, est codé en Java et son but est de fournir au projet des relevés d'usage réel fait par de vrais utilisateurs, principalement des étudiants (TELECOM Nancy et UTT). Cette *gateway* est codée en deux parties distinctes : une partie, nommée *Ingress Gateway*, va se charger d'écouter les requêtes HTTP des utilisateurs, un peu comme un proxy web, et va transformer ces requêtes en nouvelles requêtes (le monde NDN) et va les transmettre à la deuxième partie, nommée *Egress Gateway*, qui va récupérer ces requêtes pour recréer la requête originale et l'envoyer au serveur HTTP dans le monde IP. Une fois la réponse retournée à la *Egress Gateway*, la réponse va parcourir le chemin inverse de la requête NDN jusqu'à la *Ingress gateway* qui à son tour recréera la réponse HTTP pour pouvoir l'envoyer par la suite à l'utilisateur.

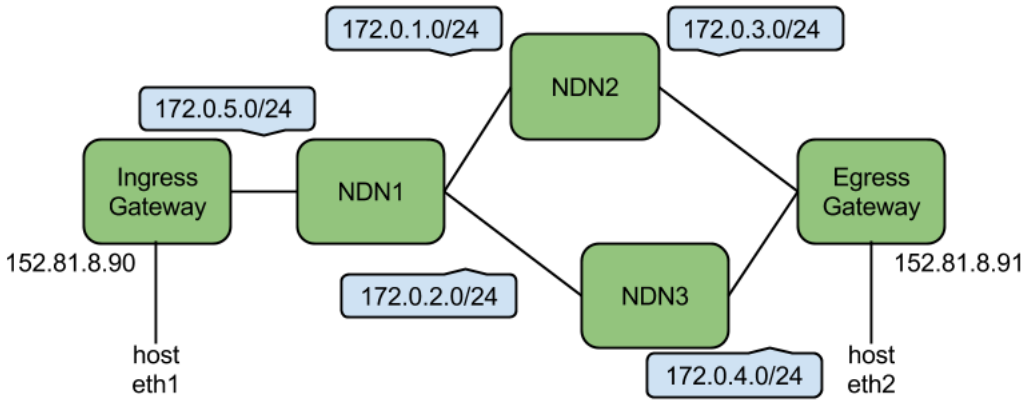


FIGURE 5.1 – Implémentation des gateway d'Orange sous Docker

La figure 5.1 représente l'implémentation qui a été faite pour tester les *gateway* d'Orange. Il faut savoir que les *gateway* sont encore en développement et que les performances affichées dans la

figure 5.2 ne sont pas les performances finales. De plus celles-ci dépendent aussi de l'évolution du code du protocole NDN. La figure 5.2 Montre le débit IP et NDN de chaque *gateway* pour un téléchargement d'un fichier de 20 mégaoctets. Ce qu'il faut retenir dans cette figure :

1. en rouge : la requête HTTP de l'utilisateur qu'a transformé la *Ingress Gateway* dans le monde NDN.
2. en vert : le téléchargement du fichier par la *Engress Gateway* dans le monde IP.
3. en jaune : le transfert du fichier entre les deux *gateway* dans le monde NDN.
4. en orange : l'envoi du ficher à l'utilisateur dans le monde IP.

FIGURE 5.2 – Historique du transfert d'un fichier de 20 mégaoctets

On peut voir que le débit dans le monde IP est bon mais que le débit entre les deux *gateway* n'est pas très important (~370 KB/s) et c'est pour cette raison que j'ai fait une étude plus poussée sur le protocole NDN pour aider ainsi au développement des *gateway*. Le but est de savoir comment on pourrait augmenter le débit NDN de façon globale ?

5.2 Solution testées

Comme il n'existait pas d'outil pour faire des tests de performance dans le protocole NDN, j'ai décidé de coder un outil qui va me permettre d'obtenir le débit maximum en faisant varier plusieurs paramètres. Pour rester dans le thème des *gateway*, j'ai commencé par créer un couple client/serveur très simple qui va reproduire le processus Interest/Data grâce à la librairie jNDN. Le principe de fonctionnement est très simple (voir figure 5.3) :

1. le client va envoyer un paquet Interest au serveur, le plus simple possible : un nom, une nonce, et le sélecteur *MustBeFresh* actif.
2. le serveur va retourner au client un paquet *Data* avec le même nom, l'information *Freshness* à 0 ms pour ne pas utiliser le cache et un contenu fixe issu d'une chaîne de caractères de la taille maximum d'un paquet (8800 octets), dont une partie sera extraite et définira la taille du contenu envoyé dans chaque paquet.
3. après réception du paquet Data le client l'ignore et renvoie le paquets *Interest* ayant permis de recevoir le paquets *Data*.

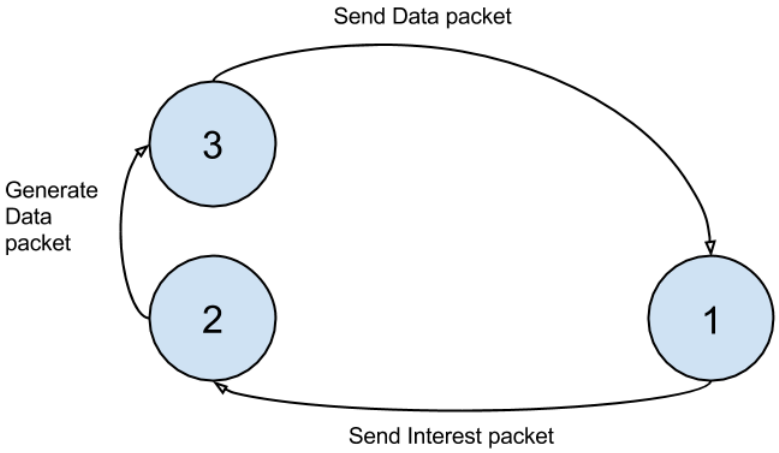


FIGURE 5.3 – Fonctionnement du client/serveur Java

Cette façon de faire essaye de minimiser au maximum le nombre d'actions nécessaires entre l'envoi et la réception des deux cotés pour réduire au maximum l'impact sur le RTT (Round-Tirp Time, c'est le temps passé entre l'émission d'une requête et sa réponse) des applications et ainsi obtenir le débit maximum. Mais on peut voir que cette façon de faire n'envoie qu'un paquet à la fois et attend la réponse avant d'envoyer le prochain paquet *Interest*. Cette façon de faire n'est pas optimale car le débit va être inversement proportionnel au RTT ($dbit\_max = \frac{taille\_de\_la\_fenetre\ (en\ bits)}{RTT\ (en\ seconds)}$) mais c'est de cette façon que fonctionne la *gateway* pour le moment. De ce fait, une première optimisation serait d'envoyer plusieurs paquets *Interest* au même moment un peu comme la fenêtre d'émission de TCP/IP. Ainsi l'objet de notre premier test est de faire varier cette fenêtre d'émission mais aussi la taille du contenu envoyé par le serveur car dans le code des *gateway* cette valeur est fixée à 4500 octets, la signature est la même que pour la *gateway*, à savoir une signature RSA-2048 et le test est effectué sur l'interface de Loopback de la *workstation*.

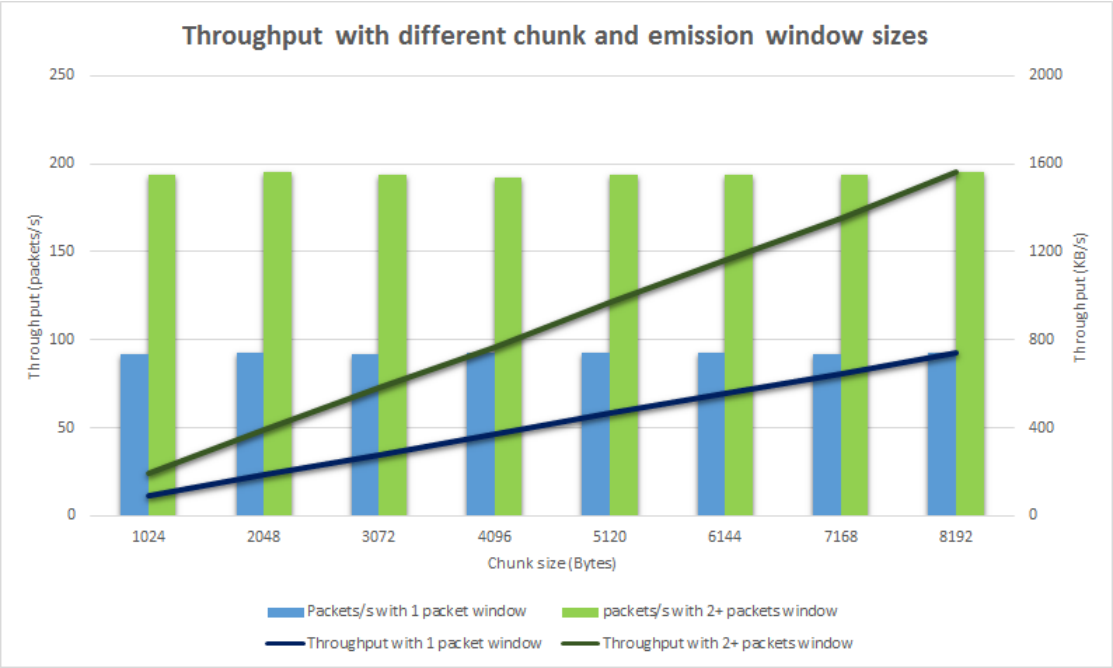


FIGURE 5.4 – Débit atteint pas le couple client/serveur en fonction de la fenêtre d'émission et de la taille du contenu

la figure 5.4 présente deux relevés, l'un en histogramme qui représente le débit en paquets par seconde et l'autre en courbe qui représente le débit en kilooctets par seconde. Pour l'envoi d'un paquet à la fois, on peut voir que le fait de changer la taille du contenu envoyé ne change pas le débit en paquets par seconde (histogrammes). Il en résulte une augmentation linéaire du débit avec la taille du contenu (courbes). Dans ces conditions l'occupation du cœur du processeur qui exécute le serveur s'élève a $\sim$ 50%. C'est pour cela que le fait d'envoyer plus de deux paquets à la fois n'a aucun impact sur les performances car à partir de deux paquets le cœur du processeur traitant l'application est constamment à 100%. Mais malgré cela augmenter le nombre de paquets envoyés simultanément a permis de doubler le débit atteint précédemment. Une chose importante à retenir de ce test est qu'il est plus efficace d'envoyer le maximum de données possible dans le paquet *Data* pour maximiser le débit.

Comme le serveur sature très vite le cœur sur lequel il est exécuté, j'ai décidé de réécrire le code du serveur pour qu'il soit cette fois-ci multi-thread. Le principe reste le même sauf que le thread principale va réceptionner les paquets *Interest*, via une boucle sur la fonction processEvent, et mettre leur nom NDN dans une queue FIFO bloquante. Ensuite des threads de calculs iront récupérer le nom NDN et forgeront les paquets *Data* pour ensuite les mettre dans la file d'envois (exécuter dans la boucle du thread principale car la même fonction traite l'envoi et la réception). Ce fonctionnement suit le concept de producteur/consommateur. Ainsi le serveur pourra exploiter toute la puissance du processeur de la *workstation*. Dans ce deuxième test on fixe la taille du contenu à 8192 octets et le nombre de thread de calculs à 4. La signature est toujours RSA-2048.

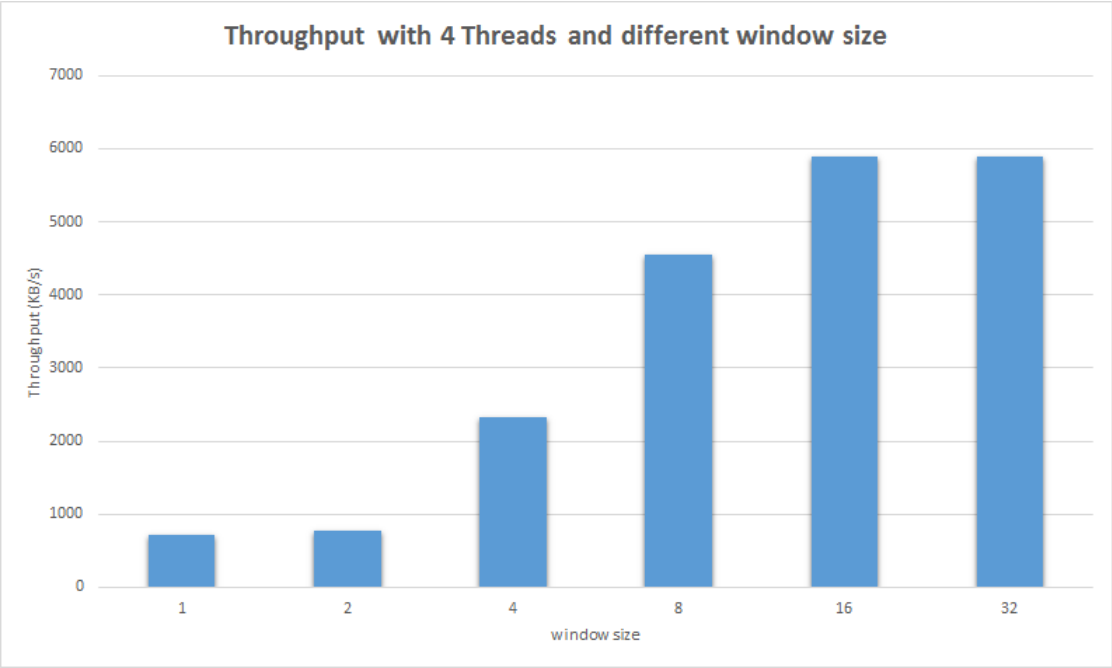


FIGURE 5.5 – Débit atteint par le serveur multi-thread avec 4 threads de calculs

La figure 5.5 montre l'augmentation de performance obtenu avec la version multi-thread du serveur. Le débit maximum atteint est aux alentours de 6 MB/s, ce qui est une bonne progression mais il faut noter que ces performances ne sont pas tout a fait quatre fois supérieure (3.75 fois). Cela est du aux contraintes de synchronisation de la queue FIFO bloquante. De plus un effet pour lequel je n'ai pas encore trouvé d'explication est que pour une fenêtre d'émission de 2 paquets les performances sont les mêmes que pour une fenêtre d'émission de 1 paquet, Une explication logique serait que cela est dû à la synchronisation des threads. Comme le serveur est maintenant multi-thread j'ai voulu expérimenter son utilisation sur le serveur *server2* qui possède 12 cœurs physiques. Cela m'a permis de tester les deux façons de router le protocole NDN "over IP" car le routage NDN n'est pas encore implémenté "over Ethernet". Ce test est le même que précédemment seule la machine change (*server2*) et le routage entre machines devient nécessaire. ainsi ce test permettra de voir si il existe des différences entre un routage NDN "over TCP" et NDN "over UDP".

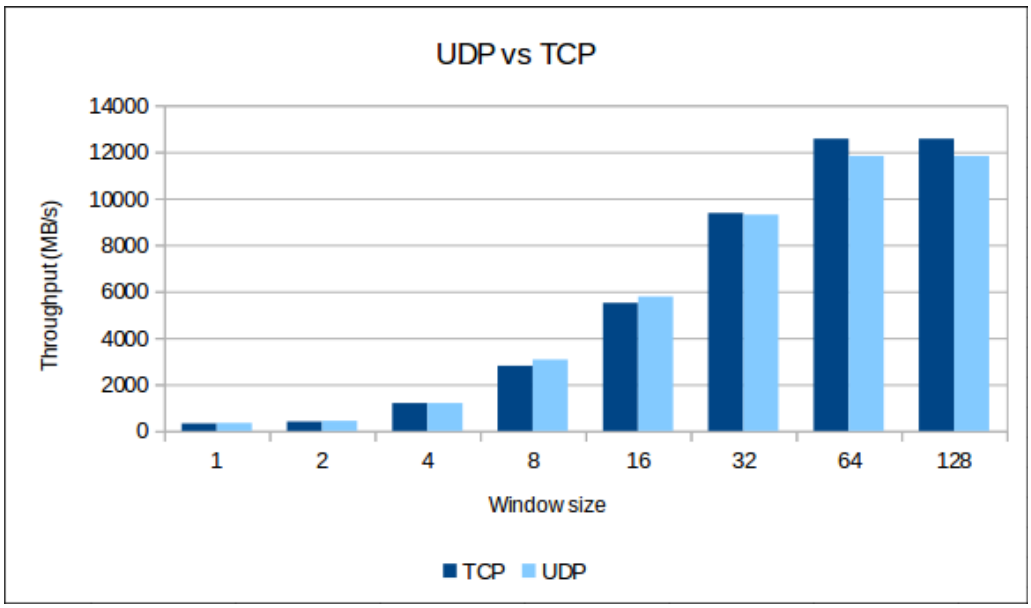


Figure 5.6 – UDP vs TCP

La figure 5.6 nous montre 2 histogrammes, l'un pour TCP (foncé) et l'autre pour UDP (clair). La première chose que l'on peut remarquer par rapport au test précèdent est que le débit maximum atteint est un peu plus du double avec 12.3 MB/s en moyenne avec une fenêtre d'émission de 64 paquets. Il faut noter que le débit est moins important que notre *workstation* pour une fenêtre donnée car la fréquence de fonctionnement est plus faible (2.2 GHz contre 3.3 GHz) mais en fréquence cumulée on arrive à deux fois plus de cycle par seconde (26,4 GHz cumulés contre 13.2 GHz) ce qui explique le débit maximum 2 fois plus important. Ensuite pour ce qui est des protocoles TCP et UDP, les performances sont assez similaires avec toutefois une nuance : UDP est plus rapide pour des petites fenêtre d'émission mais ce fait dépasser par TCP au-delà d'une fenêtre de 32 paquets car UDP commence à perdre des paquets ce qui nécessite que le client attende le timeout du paquet (4 secondes par défaut) avant de pouvoir le relancer.

J'ai voulu réaliser un dernier type de test afin de mesurer l'impact qu'a la signature sur les performances. Pour le moment il existe trois façons de signer un paquet : RSA, SHA-256 et ECDSA (courbe elliptique). ECDSA est plus expérimentale que les 2 autres car elle n'apparait pas encore dans l'API officielle [6] et est donc donnée à titre indicatif. Ainsi nous comparerons principalement SHA-256 et RSA avec différentes tailles de clés. Les constante sont une taille de contenu de 8192 octets avec 4 threads de calculs sur l'interface de Loopback de la *worstation*. Durant ce test la fenêtre est réglée pour chaque signature ainsi par exemple pour SHA-256, il faut 256 paquets pour atteindre le débit maximal alors qu'il n'en faut que 16 pour RSA-2048.

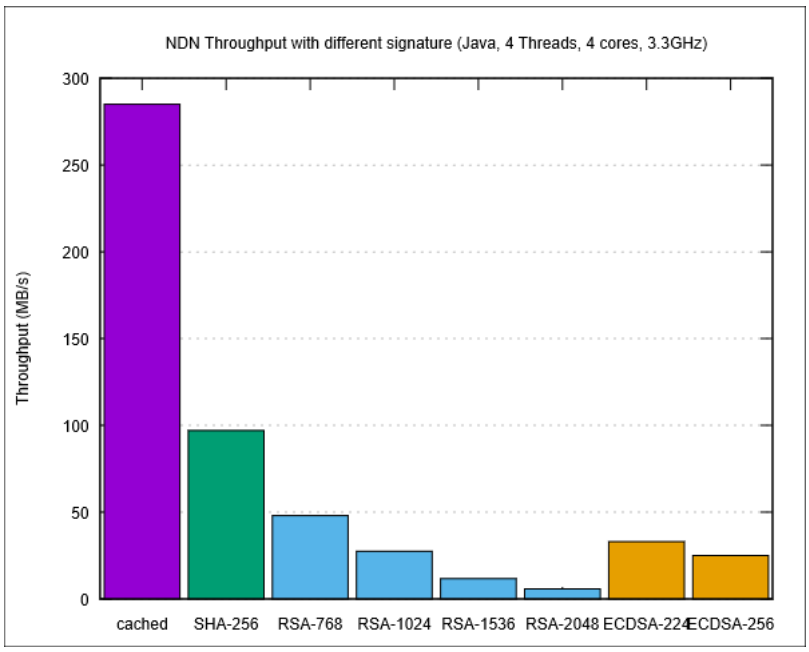


FIGURE 5.7 – Débit atteint pour différentes signatures avec le serveur Java

Sur la figure 5.7 on peut voir le débit maximum que l'on peut atteindre avec chaque signature comparé à celui du cache. On peut voir clairement que le cache est la solution la plus performante pour transmettre les données et c'est un bon point car le protocole se repose beaucoup sur l'utilisation du cache pour mettre en avant son intérêt par rapport à IP mais pour pouvoir cacher des données il faut déjà les générer. On peut voir que SHA-256 est une façon efficace de signer du contenu mais celui-ci est limité par le deamon NFD à environ 100 MB/s. De plus, SHA-256 assure seulement l'intégrité des données alors que, même si RSA est plus lent, celui ci nous assure l'intégrité mais aussi l'authentification des données grâce aux certificats. Les performances de RSA dépendent de la taille de sa clé ; plus elle est grande et plus cela met de temps pour signer un contenu. Ainsi on peut voir grâce à ces quatre valeurs, qu'en Java avoir une clé deux fois plus grande divise par quatre le débit du protocole et donc on peut en déduire une relation $O(n^2)$. Il faut savoir que RSA-768 a pu être factorisé le 12 décembre 2009 et n'est donc, dans la théorie, plus sûr. Pour ce qui est de RSA-1024 sa fiabilité devrait encore tenir quelque années.

Pour le cas de ECDSA les débits sont prometteurs avec environ 25 Mo/s pour une signature ECDSA-256 (équivalent à RSA-3072) et environ 33 Mo/s pour une signature ECDSA-224 (équivalent à RSA-2048). ECDSA a l'avantage par rapport à RSA d'utiliser des clés de chiffrement plus petites et prend moins de temps pour signer un contenu mais la vérification prend plus de temps que pour RSA se qui impacte le débit maximum que peut atteindre NFD (lors de tests avec des clés plus petites que 224 bit le débit maximum ne dépasse pas 37 Mo/s).

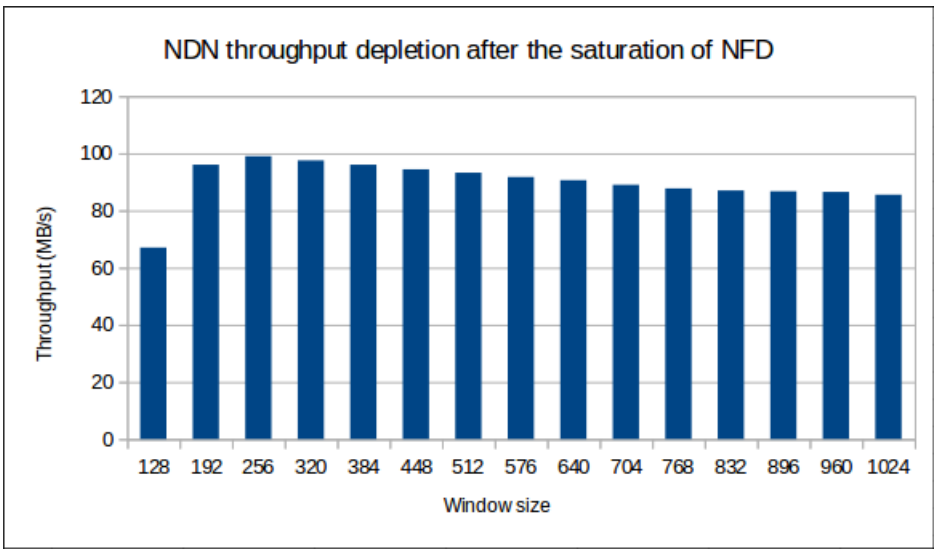


FIGURE 5.8 – Diminution des performances du daemon NFD avec le nombre de paquet Interest

Un autre point important qui ne peut se voir qu'en atteignant la limite de routage du daemon NFD est que le débit maximum lors du routage de paquets diminue avec le nombre de paquets en attente dans la PIT (voir figure 5.8).

A la suite de ces résultats, il m'a été demandé de porter mon code en C++ pour avoir un véritable outil de relevé de performances car Java n'est pas connu pour ses performances (surtout en réseau). Je développerai cette partie plus rapidement car les tests sont souvent les même que pour Java. Vous pouvez retrouver les sources de mon outil sur le dépôt suivant : https://github.com/Kanemochi/ndnperf

5.3 ndnperf

Le but de ce logiciel en ligne de commande est de pouvoir donner une estimation du débit maximal atteignable avec les paramètres suivants :

- signature
- taille de la clé RSA
- taille du contenu
- fenêtre d'émission
- nombre de thread
- fraîcheur de la donnée

Pour le premier test nous reprendrons le premier test du client/serveur simple en Java (voir figure 5.4) que nous avions fait, a savoir faire varier la taille du contenu et la taille de la fenêtre d'émission avec une constante une fraîcheur de 0 ms et une signature de type RSA-2048. Ce test est effectué avec une version modifiée de ndnping codé par les développeurs du protocole NDN car sa structure est proche et que cela m'a permis d'avoir un premier aperçu de comment coder en C++ mais aussi un aperçu des performances du protocole NDN pour le dimensionnement du serveur (car chronologiquement ce test a été réalisé avant d'avoir accès au code des *gateway* d'Orange).

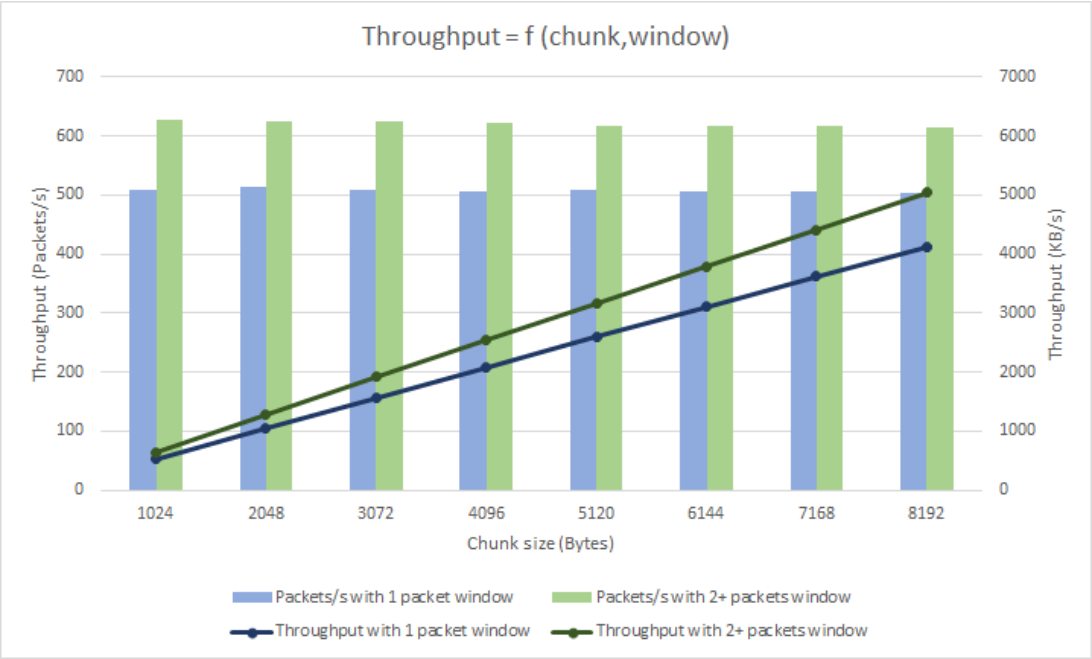


FIGURE 5.9 – Débit atteint par la version modifiée d'ndnping

Si l'on compare les figures 5.9 et 5.4, on peut voir que l'implémentation en C++ dépasse de loin celle en Java (avec un débit plus de 6 fois plus important avec une fenêtre de 1 paquet et près de 4 fois supérieur avec deux paquets et plus) et fait aussi bien que la version multi-threaded en Java avec une fenêtre d'émission de 2 paquets. Cela s'explique par un RTT beaucoup plus faible de une à deux millisecondes alors que Java met presque 10 millisecondes pour faire la même chose. C'est cette différence de performance qui a poussé le développement du serveur multi-thread en C++.

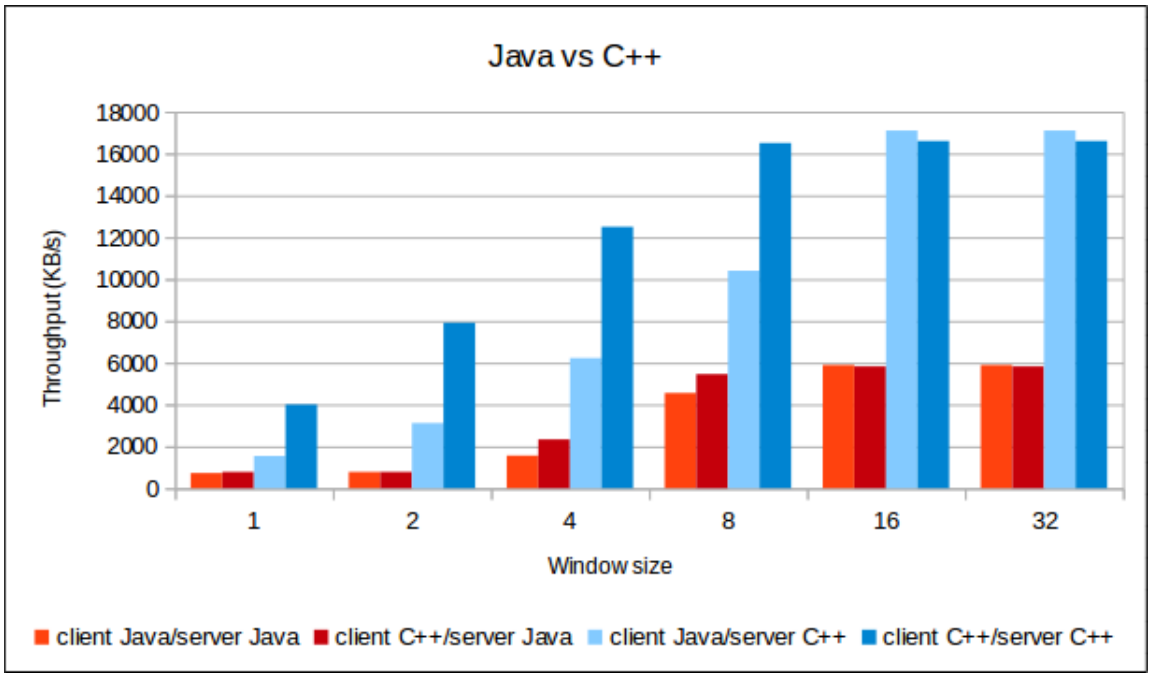


FIGURE 5.10 – Java vs C++

La figure 5.10 montre les débits atteignables par les différents couples possibles Java et C++. On remarque encore la dominance du serveur C++ par rapport au serveur Java avec un débit près de trois fois supérieur. Le constat pour la partie cliente est un peu différent, avec une petite fenêtre d'émission le client C++ fait beaucoup mieux que le client Java sauf lorsque le serveur, toutes implémentations confondues, atteint son débit maximum. A ce moment, le client Java devient plus performant car celui ci nécessite moins de cycles processeur pour s'exécuter que pour le client C++ (on peut constater la même chose pour les deux serveurs lorsqu'ils utilisent SHA-256 et sature NFD). On pourra aussi noter que du fait de sa grande réactivité, C++ atteint le débit maximum du serveur avec des fenêtres deux fois plus petites que pour Java. Pour les prochains tests mettant en évidence le débit atteignable des signature, nous utiliserons le couple client Java et serveur C++ car il offre les meilleurs performances en terme de débit.

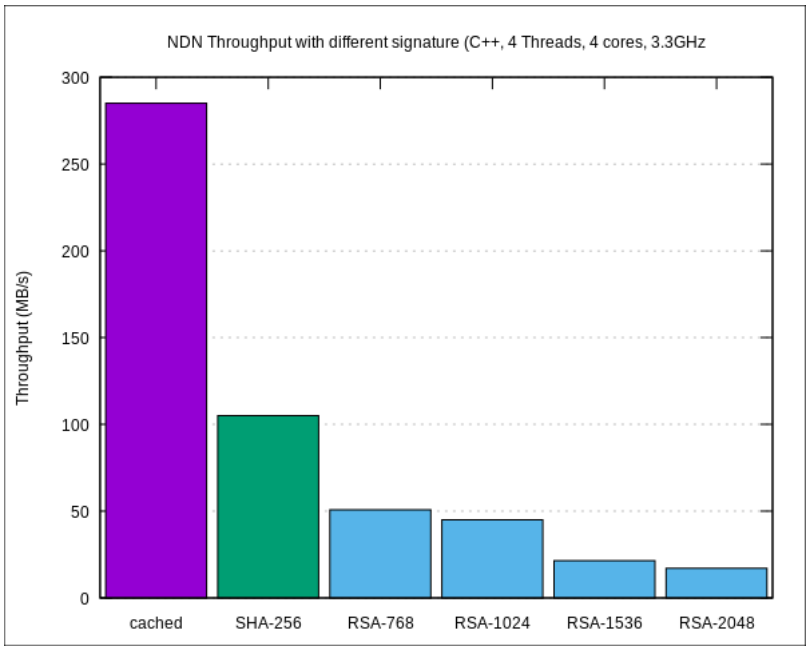


Figure 5.11 – Débit atteint pour différentes signatures avec le serveur C++

Sur la figure 5.11 on peut voir que le cache n'a pas changé étant donné qu'il ne dépend que du client et du daemon NFD. Par contre on observe une légère amélioration du débit en SHA-256 avec ~105 MB/s qui est du au fait qu'il faut envoyer moins de paquets simultanément pour atteindre le débit maximum (relevé fait avec 128 paquets). La plus grande différence est au niveau des débits avec RSA. en effet on observe que pour RSA-2048, RSA-1536 et RSA-1024 l'implémentation en C++ est plus rapide que celle en Java mais obtient le même débit pour RSA-768 (comparaison avec la figure 5.7). Contrairement à Java, C++ manque de linéarité entre les résultats de la signature RSA et ne permet pas de donner une tendance comme cela a été le cas pour Java (du à la présence d'une marche entre RSA-1024 et RSA-1536). Ainsi j'ai décide de faire un test plus précis couvrant les longueurs de clé RSA de 512 bits (minimum) jusqu'à 2240 bits (pour avoir quelque valeurs après 2048 bits) par incrément de 64 bits. le résultat se trouve sur la figure 5.12.

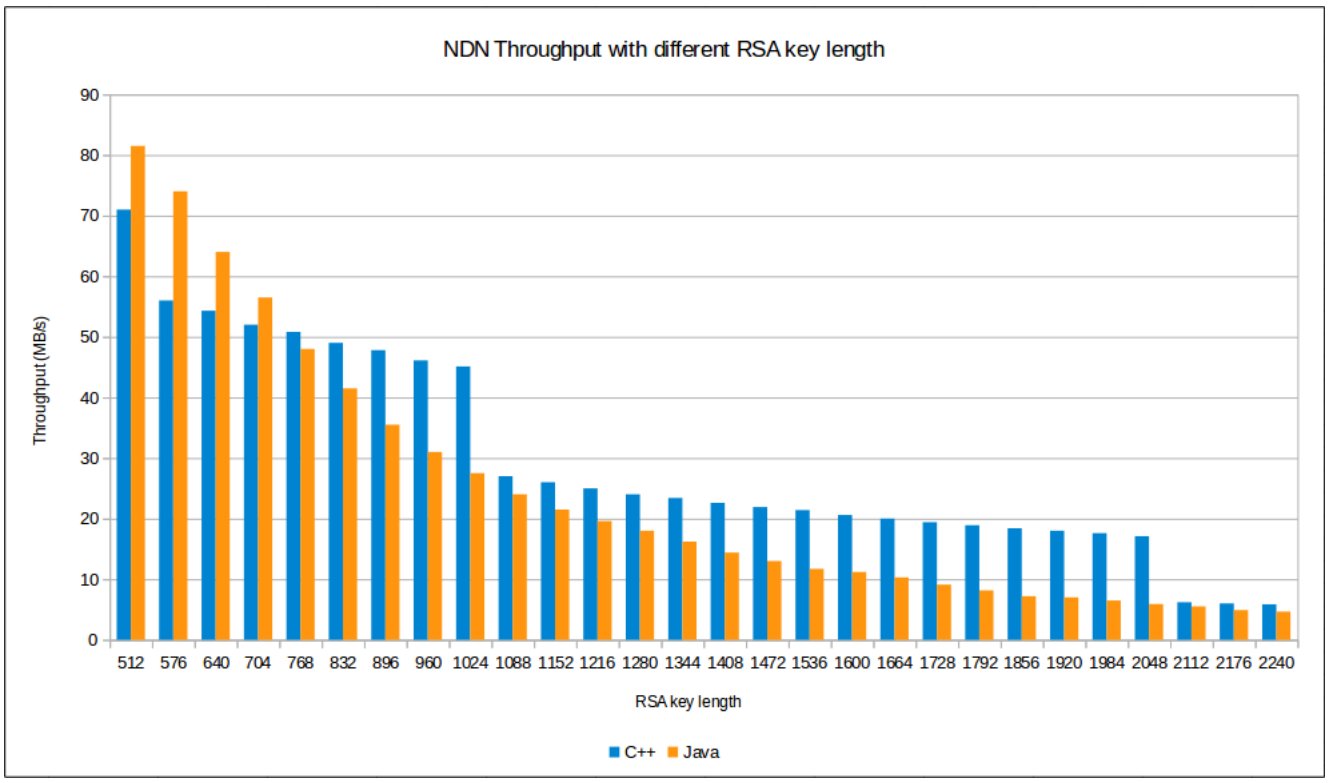


FIGURE 5.12 – Débit atteint avec différentes valeurs de longueur de clé RSA pour les serveurs Java et C++

Cette figure nous permet d'obtenir des résultats plus détaillés concernant les différents débits atteignables pour une taille de clé donnée en Java et C++ et nous permet de comprendre pourquoi en C++ il était difficile de donner une tendance avec nos quatre valeurs. En effet en C++ l'histogramme ne suit pas une tendance logarithmique comme Java mais présente des marches décroissantes. Cela est probablement dû au fait que la librairie qui permet de signer les paquets (crypto++) à été optimisée pour l'utilisation de clé RSA de taille puissance de 2 ce qui expliquerai la baise importante de performance immédiatement après ces valeurs (512, 1024 et 2048).

5.4 Repousser les limites matérielles et logicielles

Le but de cette section n'est plus d'essayer d'atteindre le débit le plus important possible mais de montrer, avec résultats à l'appui, que l'on peut encore améliorer le débit NDN.

5.4.1 Transformer les threads en workers

Une des questions que je me suis posé a été : est il possible d'aller encore plus loin en terme de débit ? Car dans le cas de la signature RSA le coût processeur est très important et les machines que j'ai eues à disposition n'arrivaient pas à dépasser 12.5 MB/s (*server2*). La solution trouvée consiste en une nouvelle approche : déporter les threads sur d'autres machines pour qu'elles puissent effectuer une partie du travail à la place du server NDN. En suivant le concept de producteur/consommateur pour l'implémentation du serveur, cela m'a permis de l'adapter à des technologies

que l'on retrouve plus dans les HPC comme par exemple les *message brokers*. Ici nous utiliserons le *message broker* RabbitMQ comme nous avons eu l'occasion d'avoir quelque cours sur le sujet durant notre 3e année. J'ai décidé de modifier l'implémentation en Java car il est beaucoup plus rapide de coder avec ce langage qu'avec C++. Cette nouvelle implémentation est répartie en deux parties (voir figure 5.13) :

1. le Front : c'est cette partie qui va se charger de récupérer les paquets *Interest*. Il va ensuite forger les paquets *Data* mais ne va pas les signer puis les envoyer au *message broker*. Envoyer des données implique que celles-ci soit sérialisables ce qui malheureusement n'est pas le cas dans les librairies NDN (au moins C++ et Java). J'ai donc créé un objet qui hérite de la class Data et qui implémente deux fonctions pour la sérialisation/déserialisation, cela à pour avantage de ne pas à avoir à modifier le code d'origine. Lorsque la seconde partie du programme retourne le paquet *Data* signé le Front peut ensuite l'envoyer au client.

2. le Worker : c'est lui qui va recevoir les paquets *Data* du Front et son travail va être de signer ces paquets ; c'est la partie la plus coûteuse. Quand il a fini, il va retourner le paquet *Data* au Front et attendre de nouveau paquets. Les Workers n'ont pas besoin de NFD pour s'exécuter mais dépendent du Front qui lui en a besoin pour le routage des paquets NDN.

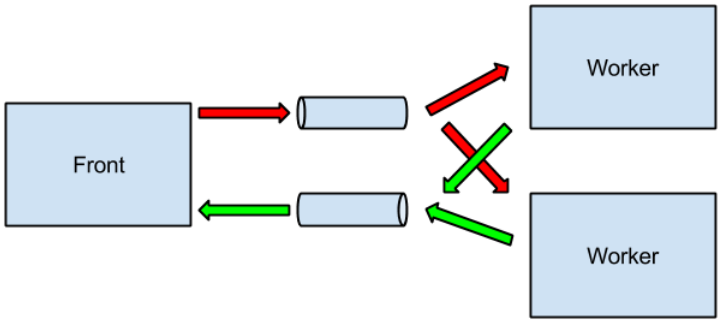


FIGURE 5.13 – Schema d'implémentation du serveur avec le *message broker*

Pour vérifier l'Intérêt de mon implémentation, j'ai conduit un test avec les trois machines que j'avais à ma disposition ce qui m'a permis de faire un test avec 19 Workers (voir figure 5.14). Ce test a été réalisé avec une taille de contenu de 8192 octets et avec une signature RSA-2048. Il faut savoir individuellement ces ordinateurs sont capables d'un débit cumulé de 21 MB/s avec cette répartition de worker (~6MB/s pour la *workstation*, ~5MB/s pour le *server1* and ~10MB/s pour le *server2*). Le résultat de ce test est présenté sur la figure 5.15.

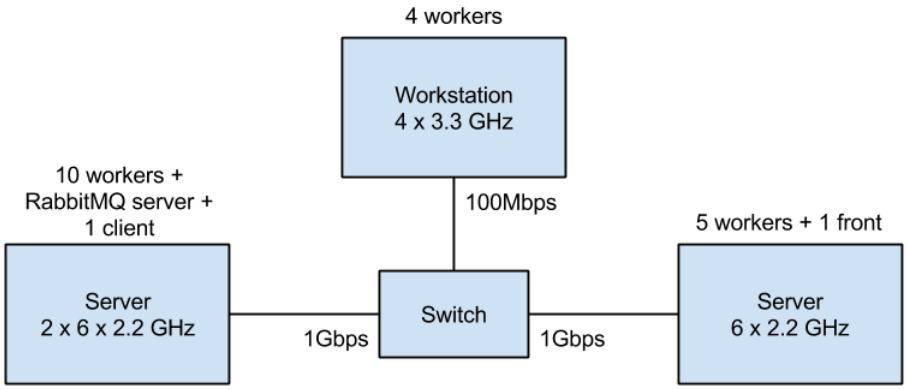


FIGURE 5.14 – Implémentation du serveur RabbitMQ avec 19 Workers

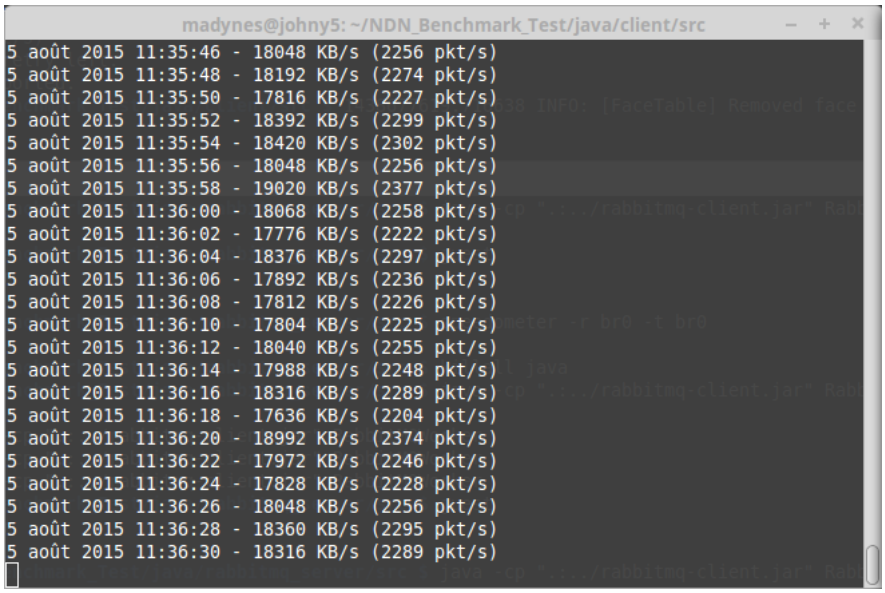


FIGURE 5.15 – Performance de implémentation du serveur RabbitMQ avec 19 Workers

Ainsi on peut voir que le débit affiché (environ 18.3 Mo/s) est très correct malgré l'ajout d'un programme tiers (RabbitMQ) et que ce débit est proche du débit théorique calculé (~87%) ce qui est un bon point car cela montre que cette solution peut être viable. Mais elle présente toutefois quelques désavantages en plus de la perte de performance :

1. Du fait du *message broker*, on augmente le RTT. Il va donc falloir une plus grande fenêtre de réception pour obtenir un débit similaire au serveur multi-thread.
2. A cause de la séparation du serveur et du *message broker* l'information doit traverser le réseau quatre fois avant de pouvoir être transmise au client et comme nous envoyons le paquet *Data* complet cela revient à multiplier par environ 5 le débit réseau nécessaire pour répondre au client.

Mais une solution est envisageable pour limiter cet augmentation du débit réseau nécessaire : créer un disque réseau partagé par les différentes machines pour qu'elles puissent avoir en commun toutes les données nécessaires à leur fonctionnement (fichier de données, clé publique, clé privée, etc ...). Ainsi grâce à cela il ne sera plus nécessaire de faire transiter d'une partie a l'autre tout le paquet *Data* mais uniquement de quoi l'identifier et quelques informations utiles. Le Front

enverra ainsi seulement le nom NDN et la taille du contenu pour ce paquet et le Worker lui renverra le nom NDN avec la signature. Chacun forgera le même paquet mais ne le transmettera pas. Ainsi l'application, dans le cas général, transmettra quatre fois le nom, deux fois la signature et ira chercher deux fois la donnée sur le disque partagé. On se retrouve donc plus qu'avec une utilisation du réseau environ 3 fois supérieure au serveur multi-thread. Et dans notre cas qui est plus spécifique, car la donnée est toujours la même, ce ratio s'abaisse encore pour atteindre une utilisation du réseau environ 1.1 fois supérieur au serveur multi-thread car le contenu est stocké dans une variable et n'est donc appelé qu'une seule fois par entité (figure 5.16). Dans cette figure le débit nécessaire pour envoyer des paquets *Data* est la somme des deux graphiques du haut et de celui en bas à droite soit $\frac{732 + 516 + 10900}{10900} = 1.115$

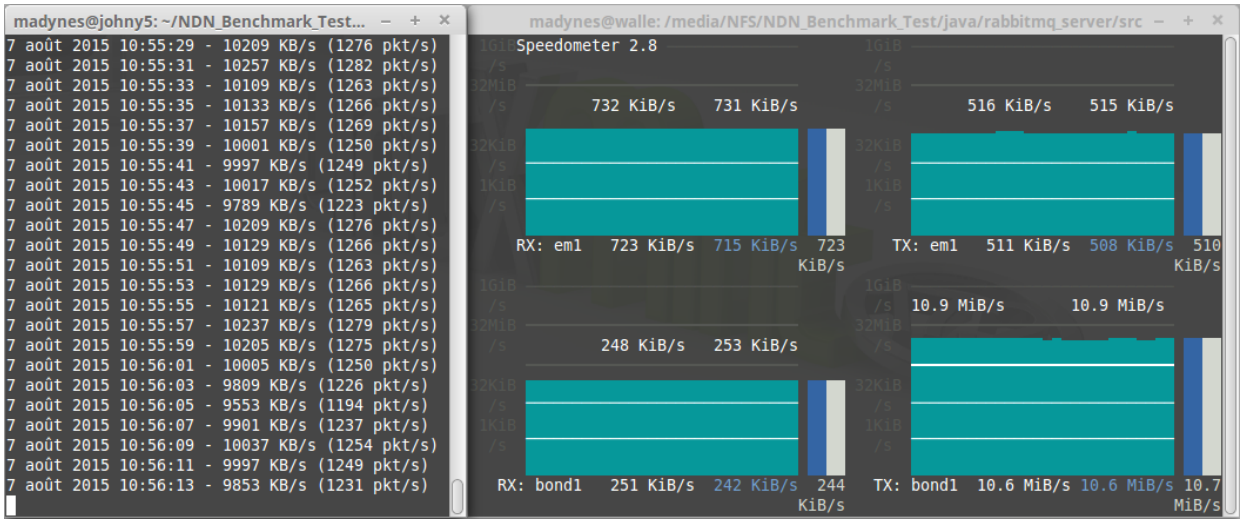


FIGURE 5.16 – Impact sur le réseau de l'implémentation RabbitMQ après optimisation

[RX :em1 = data reçu de RabbitMQ, TX :em1 = data envoyer a RabbitMQ, RX :bond1 = paquets *Interest*, TX :bond1 = paquets *Data*]

5.4.2 repartir la charge de NFD

Dans nos précédents tests on a pu voir que pour le cas de SHA-256 nous étions limités par le deamon NFD dans les alentour de 100MB/s lors du routage. La question est de savoir comment outrepasser cette limite ? Le daemon NFD est un programme mono-thread qui est nécessaire au client et au serveur car c'est lui qui permet le routage des paquets NDN mais il ne peut y en avoir qu'un par système. Le problème est que le nombre maximum de paquets qu'il peut router dépend donc de la fréquence du processeur mais un serveur a, le plus souvent, un grand nombre de cœurs processeurs mais ayant une fréquence moins importante contrairement à ce que l'on peut voir dans le secteur grand public. Ainsi pour mieux utiliser le potentiel d'un serveur, on peut essayer de virtualiser plusieurs instances du couple NFD/serveur et répartir la charge entre ces instances. Actuellement NFD implante deux grandes stratégies :

1. Best-route : choisi la face qui a le coût le plus faible (basé sur la métrique de chaque face). Équivalent à ce qu'on retrouve dans le monde IP lors du choix d'une route lorsque plusieurs pointent vers un même réseau.
2. Broadcast : envoie le paquet sur toutes les faces connues du préfixe, elle ressemble plus a du multicast et peut servir pour faire du fail-over.

Ces stratégies ont le même principe : elles envoient toutes les requêtes sur un ou plusieurs serveurs ce qui ne permet pas vraiment de répartir la charge entre eux. Néanmoins NFD propose la possibilité d'inclure ses propres stratégies et je me suis donc essayé a écrire une stratégie d'équilibrage de charge simple. Son principe est d'alterner entre chaque face connue les paquets *Interest*, de cette façon chaque serveur recevra uniquement 1/N paquets, N étant le nombre de face. Avec cette stratégie on peut observer que :

1. Cela ne change pas le débit maximum du serveur.

2. Cela ne change pas le débit maximum du client avec une seule face sinon on observe une petite perte de performance.

3. La charge est équitablement répartie sur chaque serveur.

4. Le débit maximum cumulé est égal à N fois le débit minimum. Cela veut dire que si un serveur est moins puissant que les autres il va limiter leur débit.

Ci-dessous vous pouvez voir un test qui met en évidence ce quatrième point (l'implémentation sur la figure 5.17 et le résultat sur la figure 5.18). Ce test a été effectué avec une taille de contenu de 8192 octets, une signature RSA-2048 et les trois instances sont virtualisées avec Docker. Ainsi on peut voir que le premier serveur limite le deuxième car, dans la théorie, celui-ci devrait être capable de traiter deux fois plus d'informations que le premier.

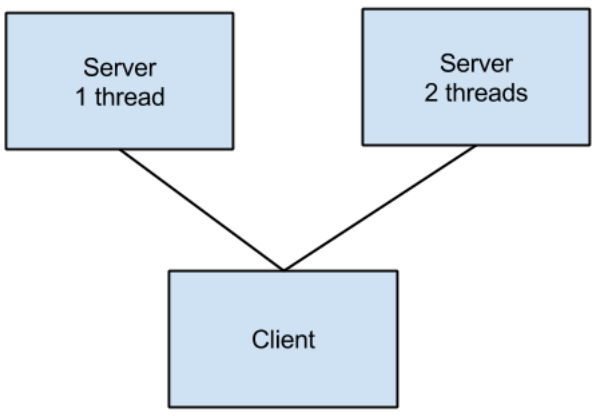


Figure 5.17 – Implémentation du test montrant la limitation du débit par le plus lent des serveurs

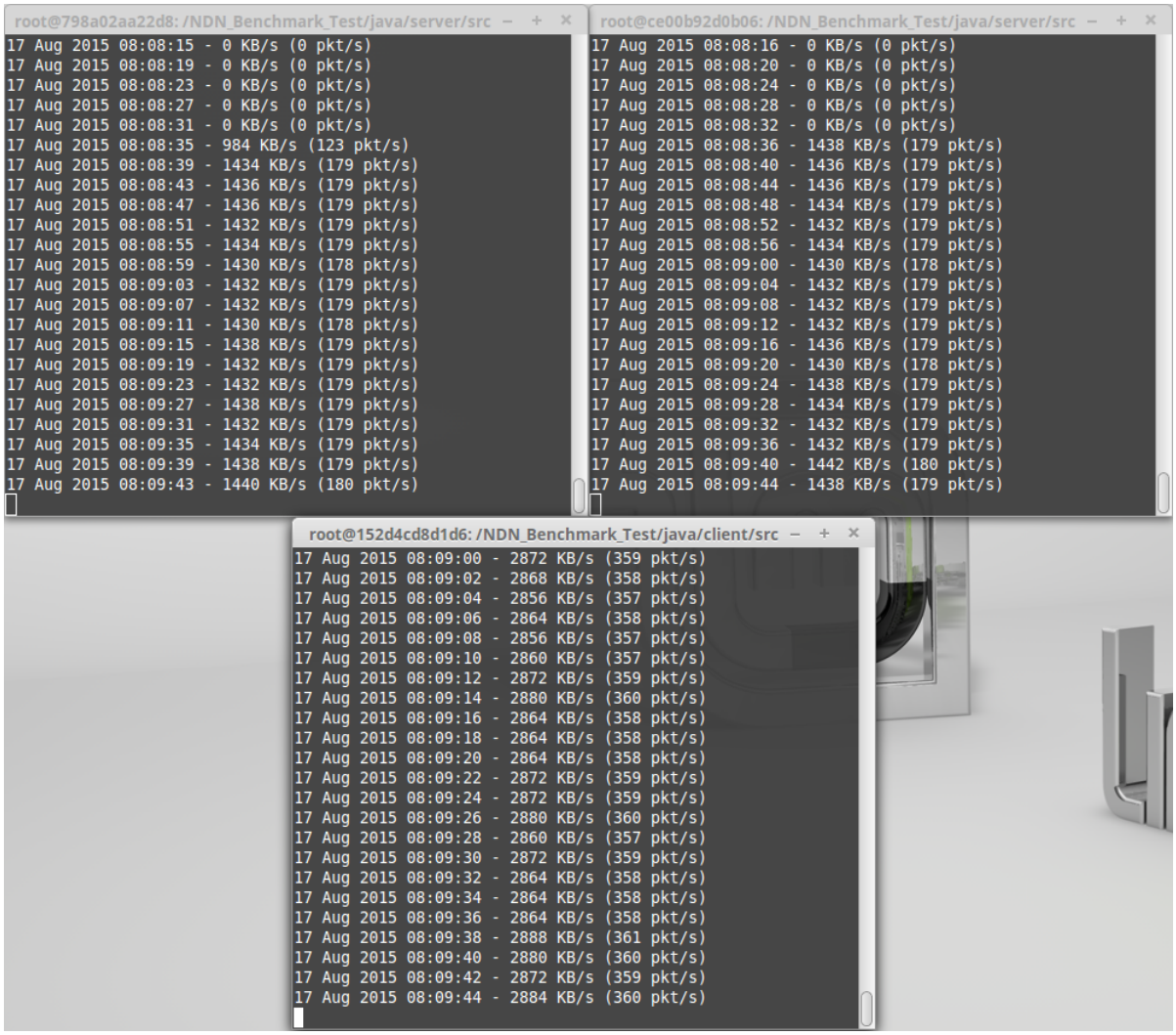


FIGURE 5.18 – Résultat de l'implémentation du test montrant la limitation du débit par le plus lent des serveurs

Ce résultat peut aussi être montré de manière théorique (voir figure 5.19). Vous trouverez d'autres démonstrations en annexe B, ou l'on peut voir que cette limitation est due au fait qu'avec cette répartition de paquets le serveur le plus lent va stocker des paquets et ainsi limiter le nombre de paquets que le client va pouvoir répartir. On peut voir que le client ne pourrait jamais distribué plus de N fois le nombre de paquets que peut traiter le plus lent des serveurs car tous les paquets supplémentaires vont finir par être mis en fil d'attente sur ce serveur.

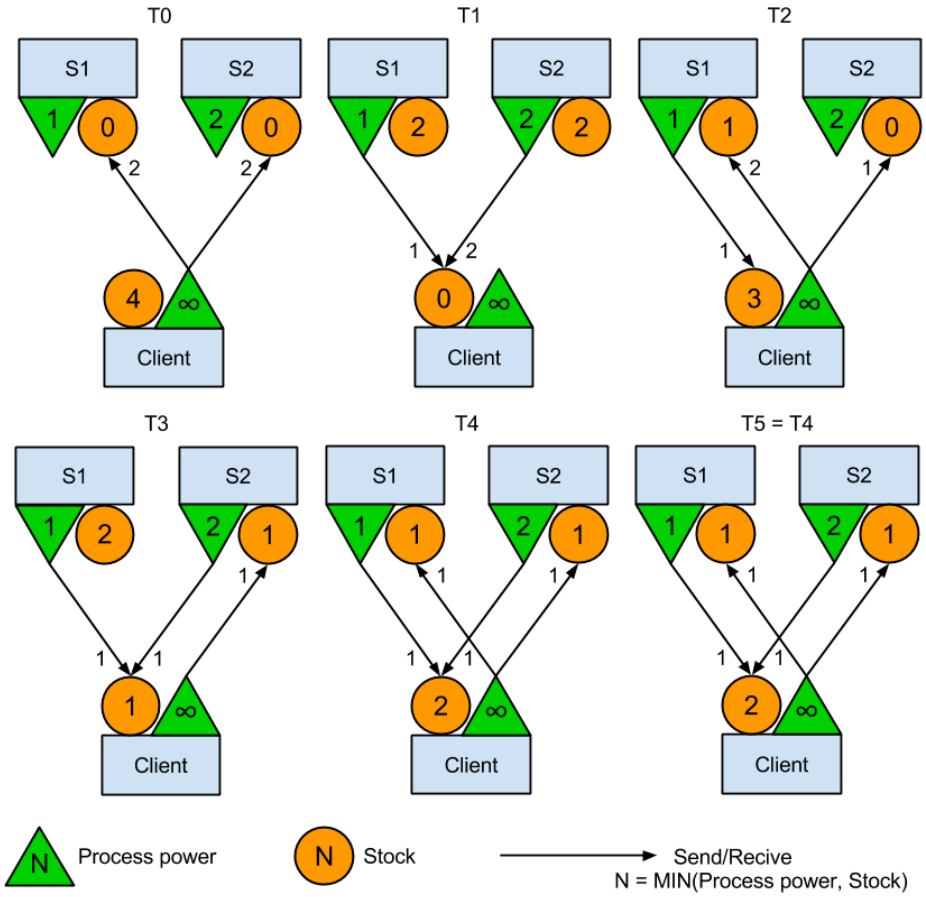


FIGURE 5.19 – Démonstration de la convergence de la stratégie d'équilibrage de charge

Ensuite je vais montrer que l'on peut atteindre un débit global supérieur à ce que l'on aurait eu avec un seul serveur. Pour ce test nous allons utiliser le *server2* car la *workstation* ne dispose pas d'assez de cœurs processeurs. En effet ce test consiste a virtualiser trois serveurs et deux clients qui vont donc au minimum utiliser cinq cœurs processeurs, car nous voulons saturer les deamon NFD avec des paquets signés avec SHA-256, et la *workstation* n'en possède que quatre. Pourquoi trois et deux ? Car ce sont des nombres premiers entre eux ce qui permettra de montrer qu'il n'y a pas de triche dans les résultats et que cette répartition est bien dû à la stratégie que j'ai implémentée car par exemple :

- avec le même nombre de clients est de serveurs on ne verrait pas bien la différence avec la stratégie *Best-route* (dans une configuration 1 client pour 1 serveur)
- avoir moins de deux clients n'a pas d'intérêt réel à cause de la limitation du deamon NFD du client
- et plus d'instances (7 conteneurs et plus) risquerait de saturer le serveur.

Dans un premier temps il faut voir où se situe la limite en débit de NFD sur un serveur virtualisé, pour cela il suffit de faire un test avec la stratégie *Best-route* entre un client et un serveur (voir figure 5.20). On peut voir sur cette figure que le débit maximum moyen est d'environ 50 MB/s. Sur la figure 5.21 on peut voir les débits atteints avec la stratégie d'équilibrage de charge. Chaque client affiche un débit d'environ 40 MB/s ce qui est moins que la stratégie *Best-route* mais la somme des deux clients est supérieure à celle-ci avec environ 80 MB/s. Il faut aussi savoir que dans cette configuration, pour chaque serveur, le daemon NFD consomme environ 60% du temps processeur d'un cœur ce qui permettrait d'ajouter un autre client sans impacter le débit des deux autres. Vous retrouvez les autres tests réalisés pour ces différentes stratégies en annexe C à savoir, *Best-route* avec 2 clients, *Broadcast* avec 1 et 2 clients et la stratégie d'équilibrage de charge avec 1 client.

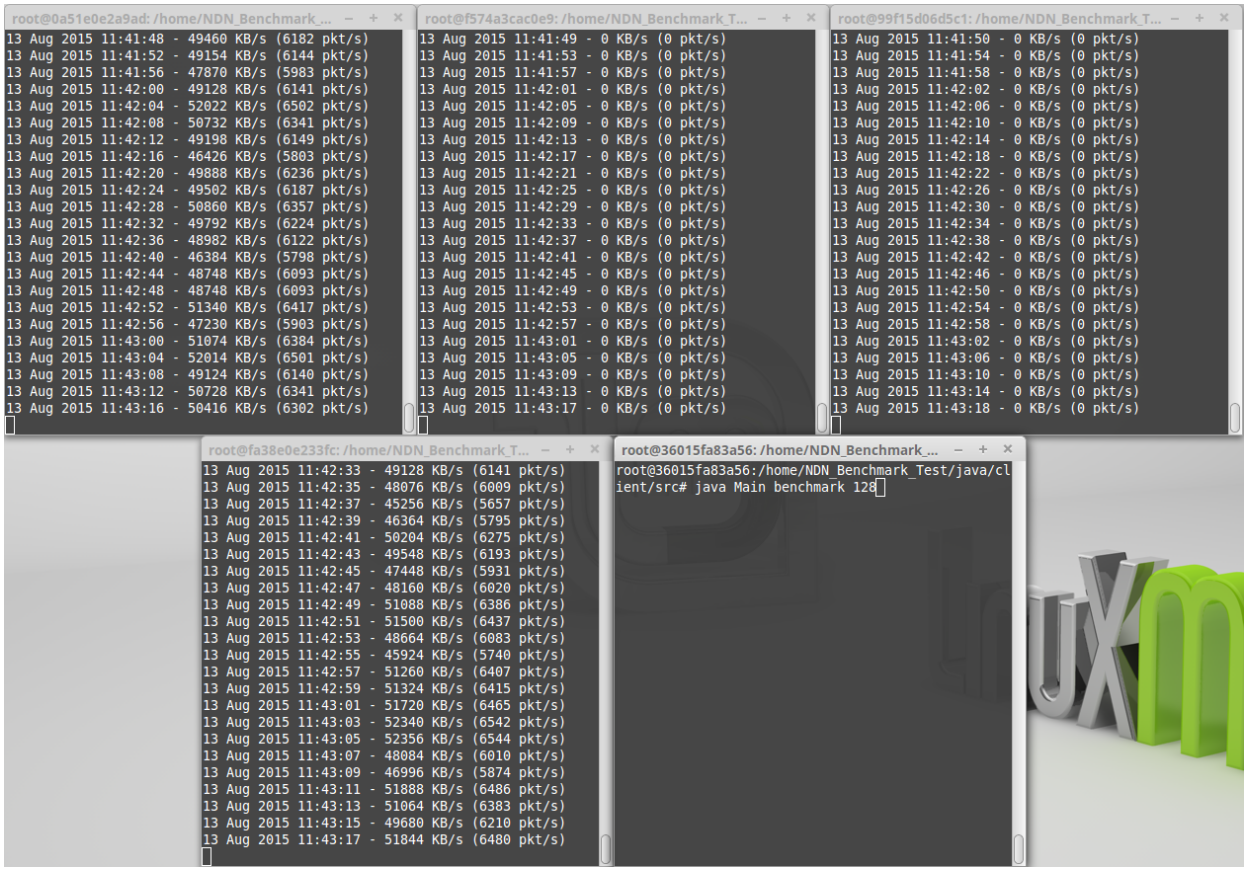


FIGURE 5.20 – Performance de la stratégie de routage *Best-route* avec 3 serveurs et 1 client

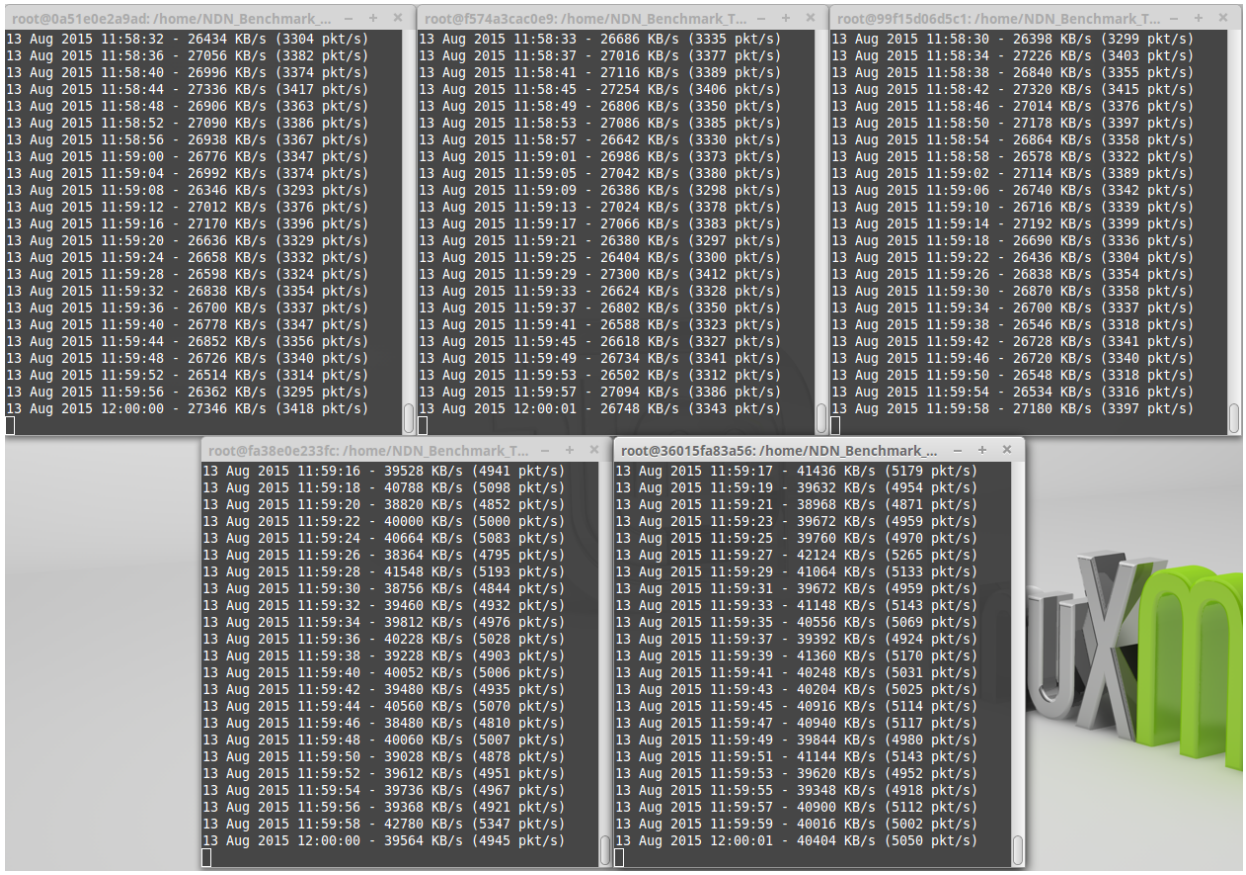


FIGURE 5.21 – Performance de la stratégie d'équilibrage de charge avec 3 serveurs et 2 clients

5.5 Conclusion

Pour conclure sur le protocole NDN, nous sommes en face d'un protocole prometteur grâce à sa nouvelle façon de "penser" qui met l'information au centre des communications plutôt que les machines qui les hébergent. Cela permet de réduire la congestion des réseaux notamment ceux où des clients demandent souvent les mêmes informations du fait que le protocole est capable de factoriser les requêtes vers le serveur pour qu'il n'ait à répondre qu'une seule fois à ces requêtes. Cela va donc réduire à terme la puissance nécessaire des serveurs qui fournissent du contenu car le nombre de requêtes qu'ils devront traiter sera grandement réduit. De plus l'utilisation de caches distribués est bénéfique puisqu'elle permet de réduire la congestion des réseaux en gardant au plus près des clients les informations qu'ils demandent le plus.

Mais pour le moment le protocole est encore jeune car les spécifications ne sont pas encore définitives et les libraires permettant d'utiliser ce protocole sont encore en développement (version 0.3.3). Comme nous avons pu le voir lors de nos tests le protocole applique une charge très importante lors que l'on doit générer des paquets car chacun d'entre eux doit être signé notamment avec RSA car SHA-256 n'assure que l'intégrité des données et un attaquant pourrait facilement tirer parti de cette signature. Mais un troisième type de signature existe dans la spécification du protocole, ce sont les courbes elliptiques (ECDSA) mais celles-ci ne sont pas encore très bien gérées par le protocole mais sont une bonne alternative à RSA car elle offre le même niveau de sécurité et coûte moins de temps processeur pour signer le même contenu. Ainsi les débits ne sont peut être pas extraordinaires pour le moment car il faut quand même de bonnes machines (12 coeurs a 2.2GHz) pour fournir un débit de l'ordre de la centaine de méga-bits par seconde avec RSA-2048, mais l'on reste optimiste car cela ne peut que s'améliorer (par exemple une amélioration hardware serait la décharge de la signature sur la carte réseau comme se qui existe déjà en IP avec la décharge du checksum). Pour le moment la seule réelle limitation est le fait que le daemon NDN est mono-thread ce qui limite le débit maximum que peut envoyer un serveur. Cette limite disparaîtra très probablement dans le future lorsque le protocole sera plus mature.

6 Dimensionnement des serveurs

6.1 Analyse du problème

Pour les besoins du projet un budget de 17000€ été réservé pour l'achat de matériel et autres besoins pour la réalisation du projet. L'université technologique de Troyes possédant déjà une grande quantité de serveurs, leur administrateurs vont donc en utiliser une partie pour les besoins du projet. Du coté de l'équipe MADYNES, qui ne dispose pas de beaucoup de matériel, une partie de ce budget servira donc à l'équiper en serveurs. Pour rappel, les serveurs sur lesquels j'ai effectué mes tests sont des serveurs existant dans l'équipe mais que ceux-ci seront dédiés au projet DOCTOR.

6.2 Solution

Pour le dimensionnement du serveur, une limite de 50% maximum du budget alloué a été définie par le responsable du projet pour l'achat de serveurs et me suis basé sur les tests précédents. Le choix le plus difficile a été de choisir le/les processeurs car ceux-ci représentent une grande proportion du prix d'un serveur et leur prix augmente assez vite avec les gammes mais aussi car c'est eux qui vont définir le débit que l'on pourra atteindre avec NDN. Pour la commande, nous sommes obligés de suivre le marché public et donc de passer par le fabriquant DELL. DELL ne propose que des serveurs à base d'Intel Xeon et ce fabriquant de processeurs a tendance à facturer plus cher la montée en fréquence que le nombre de cœurs pour une même architecture par exemple un processeur quatre cœurs à 3.5 GHz coûte 996€ alors qu'un processeur 6 cœurs à 2.4 GHz coûte 417€ alors qu'ils ont une fréquence cumulée presque identique. Cela vient du fait qu'il est plus compliqué de certifier une fréquence de fonctionnement qu'un nombre de transistors fonctionnels.

6.2.1 Dimensionnement du processeur

Pour dimensionner le processeur je me suis donné comme contrainte d'avoir approximativement 50 MB/s en C++ car au moment de la commande je n'avais pu testé que la version modifiée de ndnping car à ce moment là je n'avais pas encore accès à la *gateway* de Orange qui n'avait pas encore diffusé la première version de leur *gateway*. De plus, cette version de ndnping avais été faite pour ne pas trop perdre de temps a comprendre comment faire une application pour le protocole NDN. La raison pour laquelle ces serveurs n'apparaissent pas dans les tests de performance du protocole NDN est que DELL avait oublié de traiter la commande et donc il a fallu un mois et

trois semaines pour les recevoir. Pour la capacité du processeur j'ai appliqué une règle de trois avec le résultat de la *workstation*. Cela donne donc :

- $\frac{5\ Mo/s}{3.7\ GHz} = 1.35\ Mo.s^{-1}.GHz^{-1}$, ici j'utilise la fréquence dite de turbo car le processeur, lorsque il n'utilise qu'un ou deux cœurs, a une fréquence de fonctionnement de 3.7 GHz au lieu de 3.3 GHz.
- $\frac{50\ Mo/s}{1.35\ Mo.s^{-1}.GHz^{-1}} = 37\ GHz$

Ainsi on peut voir qu'il faut environ 37 GHz de fréquence cumulée ce qui correspond à un processeur de 14 a 16 cœurs selon la gamme actuelle de Intel. Mais le problème est que ces processeurs coûtent au minimum 2400€ (pour ceux qui pourrait correspondre) ce qui est beaucoup trop cher. Mais DELL propose la possibilité d'équiper la plupart des serveurs de 2 processeurs. Avant de choisir cette option j'ai voulu vérifier le ratio du gain d'un deuxième processeur grâce aux deux serveurs de M. Jérôme FRANÇOIS qui on les même processeurs (voir figure 6.1 et 6.2).

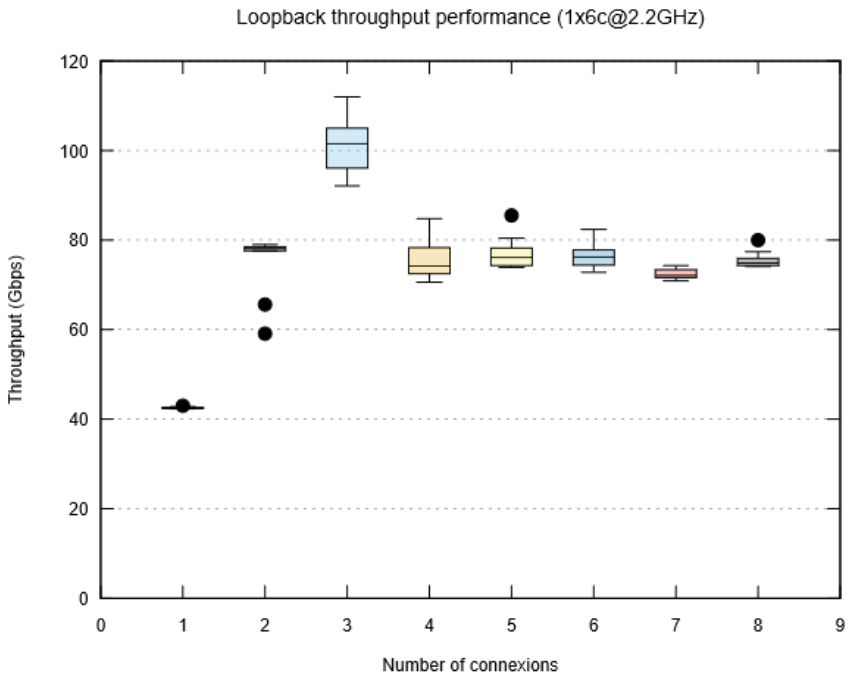


FIGURE 6.1 – Performance de l'interface de Loopback avec 1 processeur

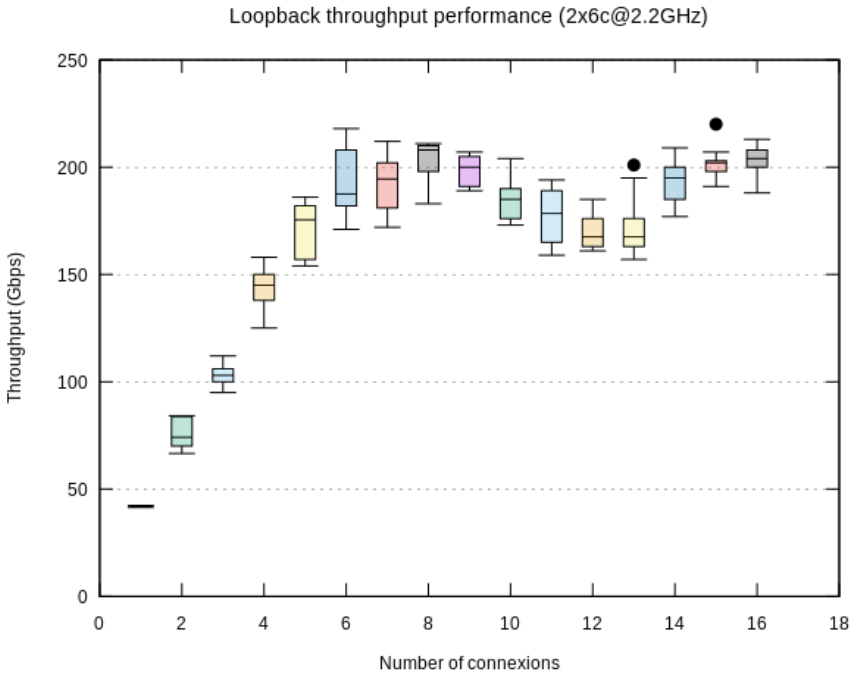


FIGURE 6.2 – Performance de l'interface de Loopback avec 2 processeurs

Sur figure 6.1 et 6.2 on peut voir que le facteur de mise à l'échelle est très bon ce qui me permet de valider ce type de configuration. Ainsi le choix s'est porté sur deux Xeon E5-2630v3 (8 cœurs et 2.4GHz) à 667€ chacun. Ce sont aussi ces processeurs qui ont le meilleur ratio performance/-prix car, en se renseignant sur les architectures des processeurs, si l'on veut plus de cœurs les processeurs passent d'une architecture 1 anneau à 2 anneaux ce qui explique pourquoi les prix augmente si vite entre 8 et 10 cœurs et plus (démarre à 1166€ pour 10 cœurs a 2.3 GHz). De même une version 2.6 GHz existe mais est vendu 40% plus cher pour moins de 10% de performance n'est pas très intéressant pour notre gamme de prix.

6.2.2 Dimensionnement de la mémoire vive

Pour le dimensionnement de la mémoire vive je suis parti sur une base de 64 GB. Cette valeur est assez empirique car il n'est pas simple de savoir la quantité exacte de mémoire nécessaire. Il existe 2 paramètres qui m'ont poussé à ce choix, le premier est le fait que l'on va faire de la virtualisation (la solution de virtualisation n'avait pas encore été définie) du coup la quantité est prévue pour de la virtualisation d'une quinzaine de machines virtuelles avec une répartition d'au moins 2GB par cœurs . Le second paramètre est le fait que NFD met en cache les données NDN directement dans la mémoire vive il faut donc prévoir une portion pour cette usage.

6.2.3 Dimensionnement des interfaces réseaux

Au niveau de la connectique réseau j'ai opté pour une carte additionnelle 10 Gbps en RJ45 pour pouvoir exploiter au mieux le cache NDN car le débit atteignable est, d'après nos tests, d'environ

280 MB/s ce qui est beaucoup plus que ce que permet les carte réseau 1 Gbps. De plus le serveur, d'après le projet, doit à terme être placé à TELECOM Nancy et le cœur de réseau de l'école est aussi en 10 Gbps RJ45. Cette carte est une Intel X540, ce modèle a été choisi car qu'elle est compatible avec la solution 6Wind qui permet une optimisations des performances réseau sous OpenStack si cette solution de virtualisation était retenue.

6.2.4 Dimensionnement du stockage

Pour le stockage j'ai opté pour deux SSD de 400 Go qui une fois mis en RAID 0 me permettront d'atteindre des débits proches de ce que peut fournir la carte réseau additionnelle. De plus les disques SSD sont capables de supporter un grand nombre d'opérations par seconde contrairement aux disques durs avec des ordre de grandeur de 10^4 à 10^7 opérations par seconde contre 10^2 pour les disques durs. Ainsi ces disques serviront pour deux choses : la première est étendre le cache d'NDN car dans ces spécifications celui-ci prévoir de pouvoir être stocker en disque en plus de la mémoire vive il faut donc un disque véloce pour cette utilisation ; la seconde est de stocker les fichiers système ainsi que ceux des machines virtuelles car en cas d'accès multiples un disque dur s'essoufflerait vite à cause du nombre limité d'opérations par seconde qu'il peut fournir. En plus des SSD, j'ai ajouté un disque dur de 4To qui servira pour tout ce qui est stockage de fichier pour éviter de remplir les SSD qui n'ont pas une grande capacité de stockage et dont le prix au Go est encore très élevé par rapport au disque dur.

6.3 Évaluation

Si l'on résume la configuration du serveur nous avons :

- Facteur de forme : R730
- CPU : Xeon E5-2630 v3 (8c@2.4GHz)
- RAM : 64GB DDR4@2133MHz
- Stockage :
  - 2x400Go SAS SSD
  - 4To SATA HDD
- Ethernet :
  - Intel X540 2x10Gbps
  - Intel i350 2x1Gbps

Le coût total de cette configuration est de 4262€ ce qui représente environ 25% du budget alloué et cela laisse la possibilité d'en commander deux si besoin. Au cour de mon stage j'ai eu l'occasion de participé à la troisième réunion de brainstorming du projet DOCTOR qui a eu lieu le 10 et 11 juin 2015, c'est lors de cette réunion que j'ai pu présenter mes premiers résultats sur les différentes solutions de virtualisation ainsi que mes choix pour la configuration du serveur. Ces choix ont tous été validés par l'équipe du projet et il a été décidé de prendre deux serveurs car cela permettrait de faire des configurations plus complexes et qui ressemblent plus à ce que l'on retrouverait dans les réseaux d'Orange notamment avec la découpe en régions géographiques du réseau. C'est aussi à ce moment là que j'ai pu avoir la première version de la *gateway* d'Orange, qui m'a permis d'entreprendre plus en profondeur les tests du protocole NDN.

Après réception des serveurs j'ai pu entrevoir la phase d'installations de ces serveurs en les déclarant au service informatique du LORIA, en les montant (voir figure 6.3, les deux serveurs du haut sont ceux que j'ai commandés et les deux serveurs du bas sont ceux de M. Jérôme FRANÇOIS) dans leurs racks et en les reliant au réseau recherche du bâtiment.

Figure 6.3 – Les serveurs commandés une fois montés

Après l'installation du système d'exploitation, une distribution Ubuntu 15.04, j'en ai profité pour faire quelques tests pour vérifier mes choix de dimensionnement.

Pour le processeur, le test figure 6.4 a été effectué avec la version C++ du serveur multi-thread avec les paramètres suivant :

- taille du contenu : 8192 octets
- freshness : 0 ms
- nombre de threads de calculs : 32
- signature : RSA-2048

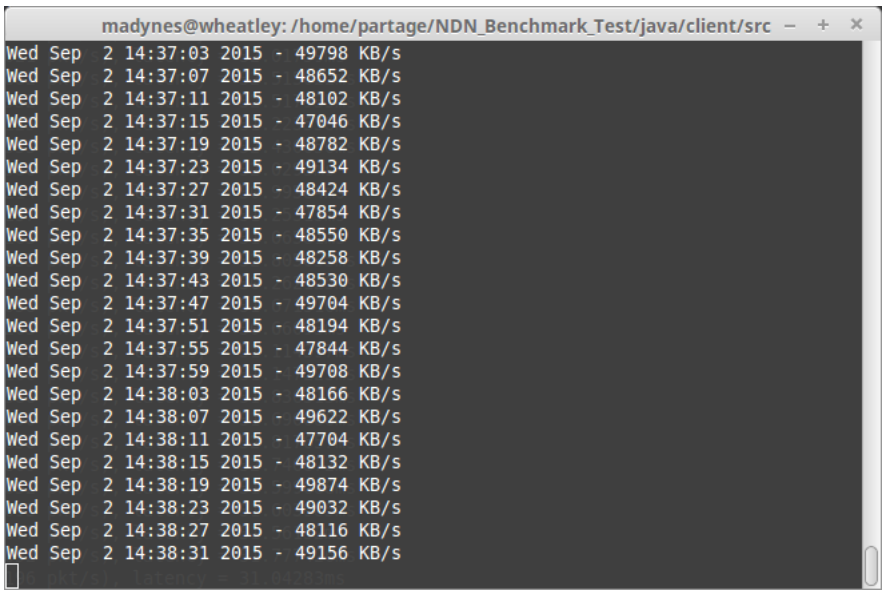


FIGURE 6.4 – Performance du protocole NDN avec les serveurs commandés

On peut remarquer que le débit affiché est proche de celui estimé avec un peu plus de 48Mo/s de moyenne.

Pour les SSD, j'ai voulu faire deux tests (voir figure 6.5 et 6.6). Le premier sert à montrer le débit en lecture aléatoire par block de 8 kilobits pour une utilisation classique du SSD (16 queues et 16 threads) par plusieurs applications optimisant leurs accès au disque avec un comportement comme NFD. Le fait que la lecture soit aléatoire et par block de 8 kilobits représente la recherche d'un paquet NDN de 8192 octets. On peut voir que le débit atteint est de 875 Mo/s (environ 110000 opérations par seconde) parce qui représente 70% d'une connexion 10Gbps. Cela est un peu plus faible que ce que j'attendais (environ 1000 Mo/s) mais reste tout a fait correct. Le deuxième test représente toujours le débit en lecture aléatoire par block de 8 kilobits mais cas d'une application dont les accès au disque ne serait pas optimisés (1 queue et 1 thread), cela représente d'une certaine manière le pire cas. On observe ainsi un débit de 105 Mo/s (environ 14000 opérations par seconde).

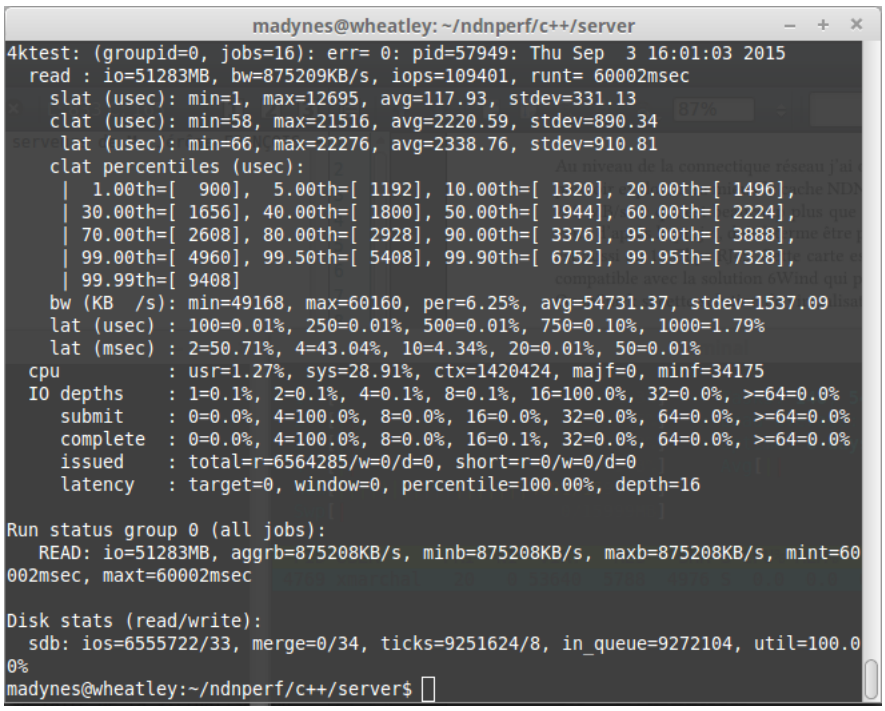


FIGURE 6.5 – Performance du raid0 de SSD dans un cas classique

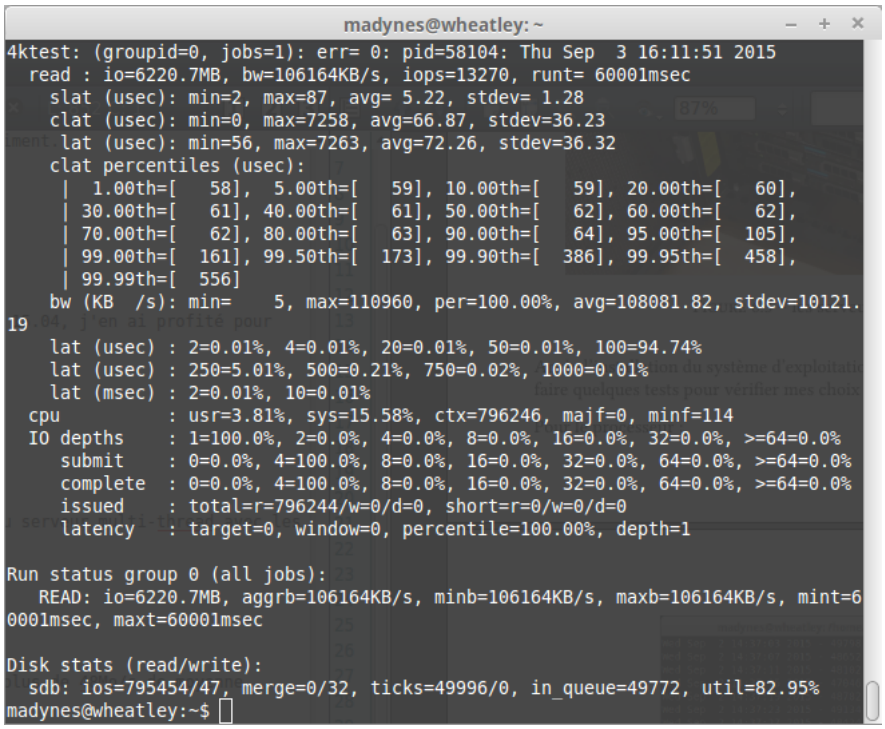


FIGURE 6.6 – Performance du raid0 de SSD dans le pire cas sans charge

7 Mise en place du réseau virtuel

7.1 Contexte

Suite a la troisième réunion de brainstorming du projet DOCTOR qui à eu lieu le 10 et 11 juin 2015, il a été décidé d'une première architecture de réseau à mettre en place. Celle-ci est la même que l'architecture qui se trouve sur la figure 5.1. Cette architecture est pensée pour tester deux choses : comment NDN et son protocole de routage gèrent le changement de route et/ou la perte d'une route. Pour le moment Docker est la solution qui tend à être la plus appréciée dans le consortium du projet. Ainsi la question est de savoir comment mettre en place cette architecture de manière simple avec cette solution de virtualisation ?

7.2 Solution proposée

Après avoir instancié une première fois l'architecture du réseau NDN pour mes tests de performance j'ai pris conscience que ça mise en place demandais pas mal de temps principalement à cause de la phase d'installation des applications. Cette phase demande plus d'opérations pour avoir un système fonctionnel que la machine hôte car l'image "ubuntu" de départ se veut légère et ne contient que le strict minimum pour pouvoir exécuter des applications (l'image ne fait que 188Mo) ainsi il faut installer beaucoup plus de programmes car une partie n'est pas présente à l'origine mais aussi faire face à des problèmes d'incompatibilité, par exemple à cause de mauvaises versions de librairies, dont il est parfois difficile de trouver la solution. Ainsi Docker offrant la possibilité de faire ses propres images, j'ai donc décidé de créer ma propre image (voir le listing "Code du déploiement de l'image Docker" plus bas) regroupant tout ce qui est nécessaire au fonctionnement de NDN mais aussi les applications nécessaires pour nos tests. ainsi cette image permettra de déployer plus facilement notre réseau mais aussi permettre par exemple une mise a échelle simple en ajoutant d'autre conteneurs à l'architecture en n'ayant qu'à les relier aux autres conteneurs et éditer les éventuels fichiers de configuration.

Code du déploiement de l'image Docker :

```
1 FROM ubuntu
2 MAINTAINER Xavier MARCHAL <xavier.marchal@telecomnancy.net>
3 RUN apt-get update && apt-get -y install software-properties-common && add-apt
    -repository ppa:openjdk-r/ppa -y && add-apt-repository ppa:ubuntu-
    toolchain-r/test -y && add-apt-repository ppa:named-data/ppa -y
4 RUN apt-get update && apt-get -y install g++ make git openjdk-8-jdk libpcap-
    dev pkg-config libsqlite3-dev libcrypto++-dev libboost-all-dev
    liblog4cxx10-dev libprotobuf-dev protobuf-compiler libssl-dev at && apt-
    get -y dist-upgrade
5 RUN cd /home && git clone --recursive https://github.com/named-data/ndn-cxx &&
```

```
    git clone --recursive https://github.com/named-data/NFD && git clone --
    recursive https://github.com/Kanemochi/ndnperf && git clone --recursive
    https://github.com/named-data/NLSR
6   RUN cd /home/ndn-cxx && ./waf configure && ./waf && ./waf install && ./waf
    clean
7   COPY rr-strategy.hpp rr-strategy.cpp /NFD/daemon/fw/
8   RUN cd /home/NFD && ./waf configure && ./waf && ./waf install && ./waf clean
    && cp /usr/local/etc/ndn/nfd.conf.sample /usr/local/etc/ndn/nfd.conf
9   RUN cd /home/NLSR && ./waf configure && ./waf && ./waf install && ./waf clean
    && mkdir /var/log/nlsr && mkdir /var/lib/nlsr
10  COPY nlsr.1 nlsr.2 nlsr.3 /home/
11  COPY Gateway_HTTP_NDN /home/Gateway_HTTP_NDN
```

Pour créer une image il faut avant tout se baser sur une image précédente. Ici j'ai choisi l'image "ubuntu" car celle ci propose une base de système d'exploitation avec un shell. Ensuite je demande à Docker d'exécuter des actions dans cette image grâce à la commande RUN. vous pourrez remarquer que l'on peut exécuter plusieurs actions à la suite de la même manière qu'en shell mais la présence de plusieurs lignes RUN a un avantage de taille car en exécutant RUN, les modification sur l'image sont enregistrer dans une nouvelle image qui est l'équivalent d'un commit ainsi lorsque l'on modifie le fichier de configuration de l'image, Docker recommence le déroulement à partir de la dernière ligne non modifiée ce qui permet de ne pas avoir à refaire toutes les étapes.

Docker offrant aussi la possibilité d'interagir avec ces conteneurs assez facilement par l'intermédiaire de l'API Docker et les fonctions système comme iproute2 pour la gestion des *namespaces*. Il semble ainsi relativement simple de pouvoir scripter la mise en place de l'architecture réseau (voir le listing "Code du script de déploiement du réseau NDN" en annexe E). Ce script est prévu pour permettre le (re)déploiement du réseau NDN énoncé précédemment mais aussi permettre de le remettre en état si il y a eu des problèmes au niveau d'un ou plusieurs conteneurs (exemple d'un conteneur qui s'arrête).

8 Conclusion

Durant mon stage, j'ai été associé au projet ANR DOCTOR qui vise à favoriser le déploiement de nouvelles solutions réseaux dans des infrastructures virtualisées. Après une étude de l'état de l'art sur les différentes technologies sélectionnées par le consortium du projet, j'ai effectué des tests de performances sur les différentes technologies de virtualisation (OpenStack/Docker) ainsi que du protocole NDN. Ces tests ont permis au consortium d'avoir une meilleure appréciation des performances réseaux de ces technologies ainsi que diverses idées d'amélioration pour le développement de leurs *gateway* NDN/HTTP. Ces tests mon aussi permis de dimensionner les serveurs qui représenteront la partie du testbed du LORIA. Après réception des serveurs j'ai pu procéder à leurs installation ainsi qu'à la mise en place d'un premier réseau NDN virtuel dans Docker à l'aide d'un script que j'ai codé qui pourra servir de base pour d'autres architectures de réseaux plus complexes.

Pour ce qui est de l'organisation et de la méthode de travail, celles-ci se décomposent en deux parties. La première est la relation que j'ai eu avec mon encadrant de stage qui se rapproche de l'encadrement de thèse ou l'encadré a une quasi total liberté de ce qu'il fait avec une réunion de temps en temps pour faire le bilan de l'avancement du projet et décider si il faut continuer dans cette voie ou redéfinir la ligne directrice. En plus de cela j'ai, à mon initiative, fait des rapports réguliers de mes travaux, problèmes et ce que je comptais faire tous les deux ou trois jours, un peu à l'instar de la méthode AGILE. J'ai aussi pris quelques liberté sur le projet notamment avec la proposition d'amélioration pour les *gateway* et le logiciel qui permet de les mettre en évidence.

La deuxième partie est la communication entre les acteurs du projet DOCTOR. Cette communication se fait de trois façons selon l'importance :

- par mail pour la diffusion de contenu, la mise en place de réunions et toutes autres formes d' expressions non prioritaires qui n'impactent pas l'avenir du projet.
- par audioconférence pour tout ce qui touche à la synchronisation entre deux parties et les décisions mineures sur le projet.
- par réunion en présentiel, en général tout les trois mois d'une durée d'environ deux jours, ces réunions servent à présenter au consortium tout ce qui a été fait durant cette période et permet de se mettre d'accord sur des choix techniques et pour faire diverses démonstrations.

Avant de démarrer ce stage j'avais encore des interrogations sur ce que j'allais faire après la fin de mes études : continuer sur une thèse ou chercher un emploi. Ce stage m'a permis d'avoir un premier aperçu de ce qu'est la recherche en informatique et de répondre à une grande partie de mes interrogations ce qui m'a conforté dans le choix de poursuivre mes études avec une thèse. Je ressors de ce stage avec la sensation d'avoir été utile au projet car :

1. Les différents études sur les performances donnent au consortium du projet DOCTOR des éléments concrets qui leur permettront de choisir une solution de virutalisation.
2. La mise en place de mon outil d'évaluation de performance ayant servi pour les tests de

performance pourra être utile a l'ensemble de la communauté NDN mais aussi au projet DOCTOR via les diverses propositions d'améliorations qu'il implémente.

3. Les serveurs du projet étant choisis, installés, configurés et fourni avec un premier script de mise en place d'un réseau virtuel, les premières expériences du projet DOCTOR pourront être réalisées à l'issue de mon stage.

Bibliographie / Webographie

[1] Van Jacobson, Diana K. Smetters, Nicholas H. Briggs, James D. Thornton, Michael F. Plass, Rebecca L. Braynard. Networking named content. Technical report, Palo Alto Research Center, CA, USA, 2009. 5

[2] Polystyrène, exemple de polymère. https://en.wikipedia.org/wiki/Polystyrene. 19, 57

[3] NDN details. http://named-data.net/project/archoverview/. 5, 6, 7, 57

[4] DOCTOR. Doctor-d1.1. Technical report, ANR, 2014. 5, 9, 10, 57

[5] Van Jacobson. A new way to look at networking, google tech talk. https://www.youtube.com/watch?v=oCZMoY3q2uM, 2006. 5

[6] API NDN. http://named-data.net/doc/ndn-ccl-api/. 28

[7] Windows Server 2016 Technical Preview 3 Networking Overview. https://technet.microsoft.com/en-us/library/mt412879.aspx. 11

[8] presentation du projet ANR DOCTOR. http://www.agence-nationale-recherche.fr/?Projet=ANR-14-CE28-0001. 1

[9] Named Data Networking Next Phase (NDN-NP) Proposal. http://www.caida.org/funding/ndn-np/ndn-np_proposal.xml. 8, 57

[10] présentation de l'Inria. http://www.inria.fr/institut/inria-en-bref. 3

[11] présentation de MADYNES sur le site de l'Inria. http://www.inria.fr/equipes/madynes. 4

[12] présentation de MADYNES sur le site du LORIA. http://www.loria.fr/la-recherche/equipes/madynes. 4

[13] présentation du LORIA. http://www.loria.fr/le-loria-1. 3

[14] site web de l'équipe MADYNES. http://madynes.loria.fr/. 4

[15] site web du projet DOCTOR. http://doctor-project.utt.fr/project_description.htm. 1

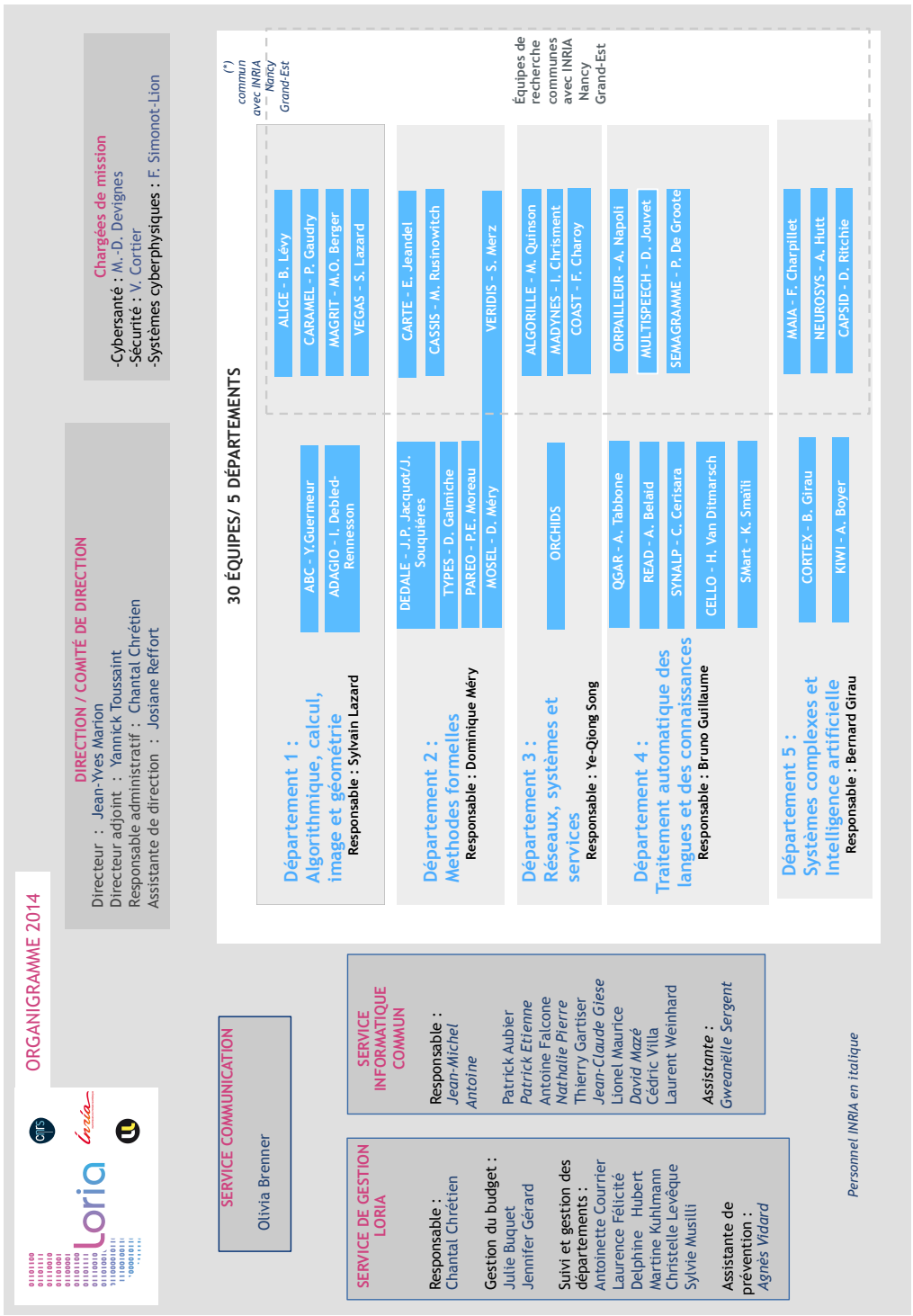
[16] vCloud for NFV with VMware Integrated OpenStack. http://www.vmware.com/files/microsites/vcloud_nfv/details.html. 11

[17] Network Functions Virtualisation. http://www.etsi.org/technologies-clusters/technologies/nfv. 8

Liste des illustrations

Annexes

A Organigramme du LORIA



B Suite de la démonstration de la stratégie de répartition de charge

t0	Is pending	process power	Will send	Will receive
client	10		10	0
Server 1	0	1	0	5
Server 2	0	2	0	5
t1				
client	0		0	3
Server 1	5	1	1	0
Server 2	5	2	2	0
t2				
client	3		3	3
Server 1	4	1	1	2
Server 2	3	2	2	1
t3				
client	3		3	3
Server 1	5	1	1	1
Server 2	2	2	2	2
t4				
client	3		3	3
Server 1	5	1	1	2
Server 2	2	2	2	1
t5				
client	3		3	2
Server 1	6	1	1	1
Server 2	1	2	1	2
t6				
client	2		2	3
Server 1	6	1	1	1
Server 2	2	2	2	1
t7				
client	3		3	2
Server 1	6	1	1	2
Server 2	1	2	1	1
t8				
client	2		2	2
Server 1	7	1	1	1
Server 2	1	2	1	1
t9 = t8				
client	2		2	2
Server 1	7	1	1	1
Server 2	1	2	1	1

FIGURE B.1 – Convergence théorique de la stratégie d'équilibrage de charge avec 2 serveurs et 1 client

t0	Is pending	process power	Will send	Will receive
client	12		12	0
Server 1	0	3	0	6
Server 2	0	7	0	6
t1				
client	0		0	9
Server 1	6	3	3	0
Server 2	6	7	6	0
t2				
client	9		9	3
Server 1	3	3	3	5
Server 2	0	7	0	4
t3				
client	3		3	7
Server 1	5	3	3	1
Server 2	4	7	4	2
t4				
client	7		7	5
Server 1	3	3	3	4
Server 2	2	7	2	3
t5				
client	5		5	6
Server 1	4	3	3	2
Server 2	3	7	3	3
t6				
client	6		6	6
Server 1	3	3	3	3
Server 2	3	7	3	3
t7 = t6				
client	6		6	6
Server 1	3	3	3	3
Server 2	3	7	3	3

FIGURE B.2 – Convergence théorique de la stratégie d'équilibrage de charge avec 2 serveurs et 1 client (vitesses de traitement différentes)

t0	Is pending	process power	Will send	Will receive
client	10		10	0
Server 1	0	1	0	4
Server 2	0	2	0	3
Server 3	0	2	0	3
t1				
client	0		0	5
Server 1	4	1	1	0
Server 2	3	2	2	0
Server 3	3	2	2	0
t2				
client	5		5	3
Server 1	3	1	1	1
Server 2	1	2	1	2
Server 3	1	2	1	2
t3				
client	3		3	5
Server 1	3	1	1	1
Server 2	2	2	2	1
Server 3	2	2	2	1
t4				
client	5		5	3
Server 1	3	1	1	2
Server 2	1	2	1	2
Server 3	1	2	1	1
t5				
client	3		3	4
Server 1	4	1	1	1
Server 2	2	2	2	1
Server 3	1	2	1	1
t6				
client	4		4	3
Server 1	4	1	1	1
Server 2	1	2	1	1
Server 3	1	2	1	2
t7				
client	3		3	4
Server 1	4	1	1	1
Server 2	1	2	1	1
Server 3	2	2	2	1
t8				
client	4		4	3
Server 1	4	1	1	2
Server 2	1	2	1	1
Server 3	1	2	1	1
t9				
client	3		3	3
Server 1	5	1	1	1
Server 2	1	2	1	1
Server 3	1	2	1	1
t10 = t9				
client	3		3	3
Server 1	5	1	1	1
Server 2	1	2	1	1
Server 3	1	2	1	1

FIGURE B.3 – Convergence théorique de la stratégie d'équilibrage de charge avec 3 serveurs et 1 client

Avec ces trois figure, nous pouvons voir que le nombre de paquets envoyés par les serveurs sont les mêmes pour tous les serveurs est vaut la valeur du plus lent des serveurs. Le système tend à stocker les paquets *Interest* ou se trouve le plus lent des serveurs car celui-ci reçoit le même nombre de paquets que les autres mais n'est pas capable de les traiter aussi vite que les autres. De plus cette stratégie n'a pas de gestion de la QoS ni de gestion de poids. Dans la figure B.2 on peut voir que le client commence avec 12 unité de paquets *Interest* au lieu de 10, cela est du au fait que si l'on a pas au moins ce nombre le pool de serveurs ne pourra pas atteint une vitesse de traitement de (nombre de serveurs) fois le serveur le plus lent. ainsi le client doit avoir un nombre d'unité de paquets *Interest* d'au moins (nombre de serveurs) fois la vitesse de traitement du serveur le plus lent plus deux fois la vitesse de traitement du serveur le plus lent (émission/réception) ce qui donne $2 * 3 + 3 + 3 = 12$.

C Suite des tests pratiques des stratégies de NFD

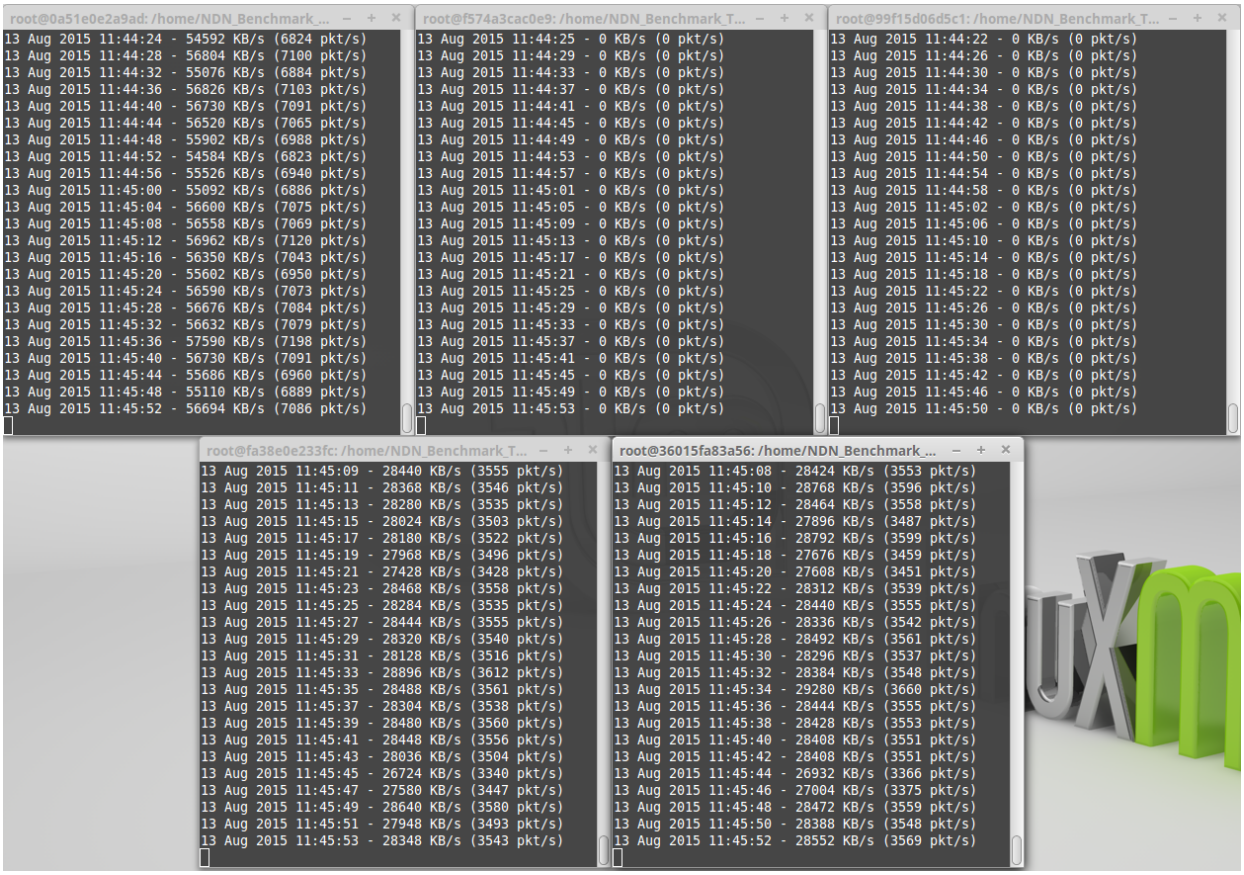


FIGURE C.1 – Test de débit de la stratégie *Best-route* avec 3 serveurs et 2 clients

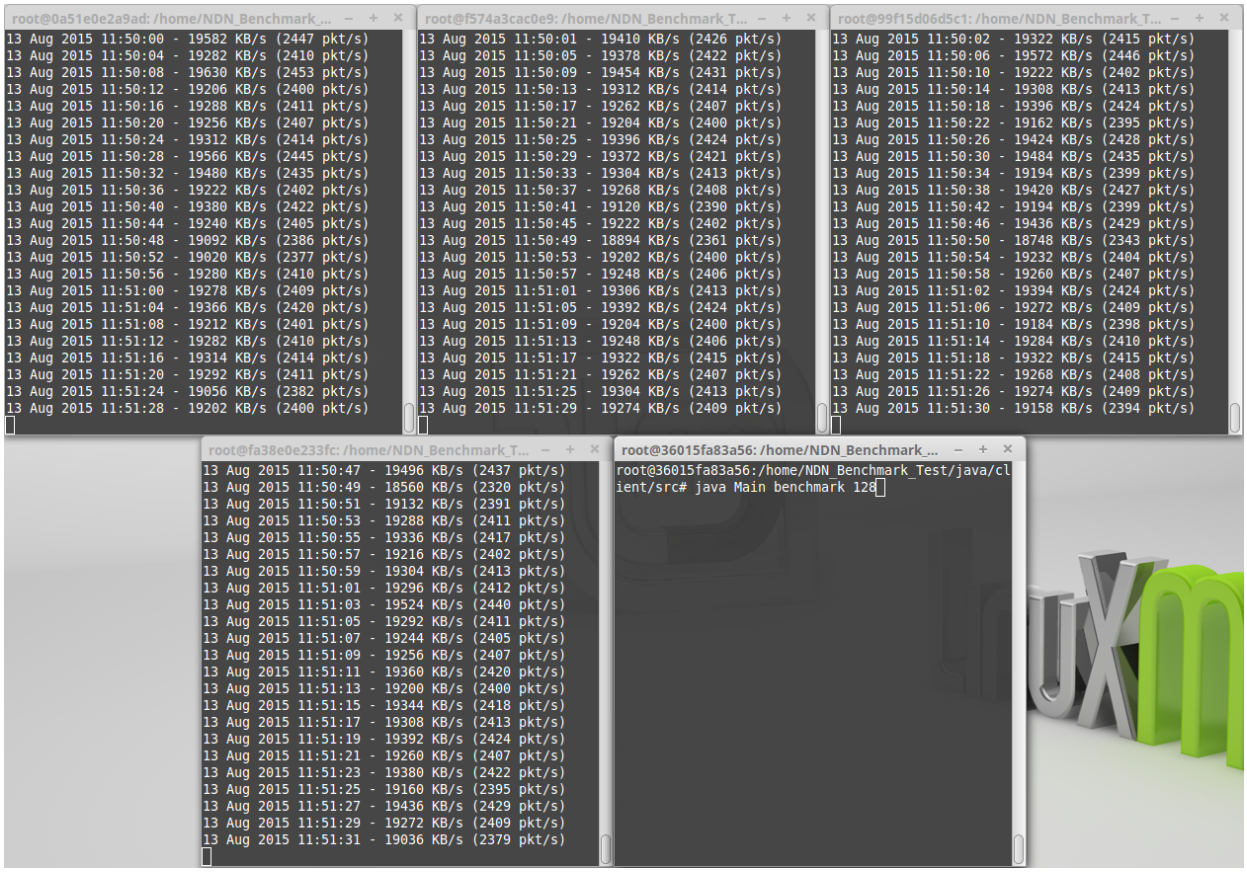


FIGURE C.2 – Test de débit de la stratégie *Broadcast* avec 3 serveurs et 1 client

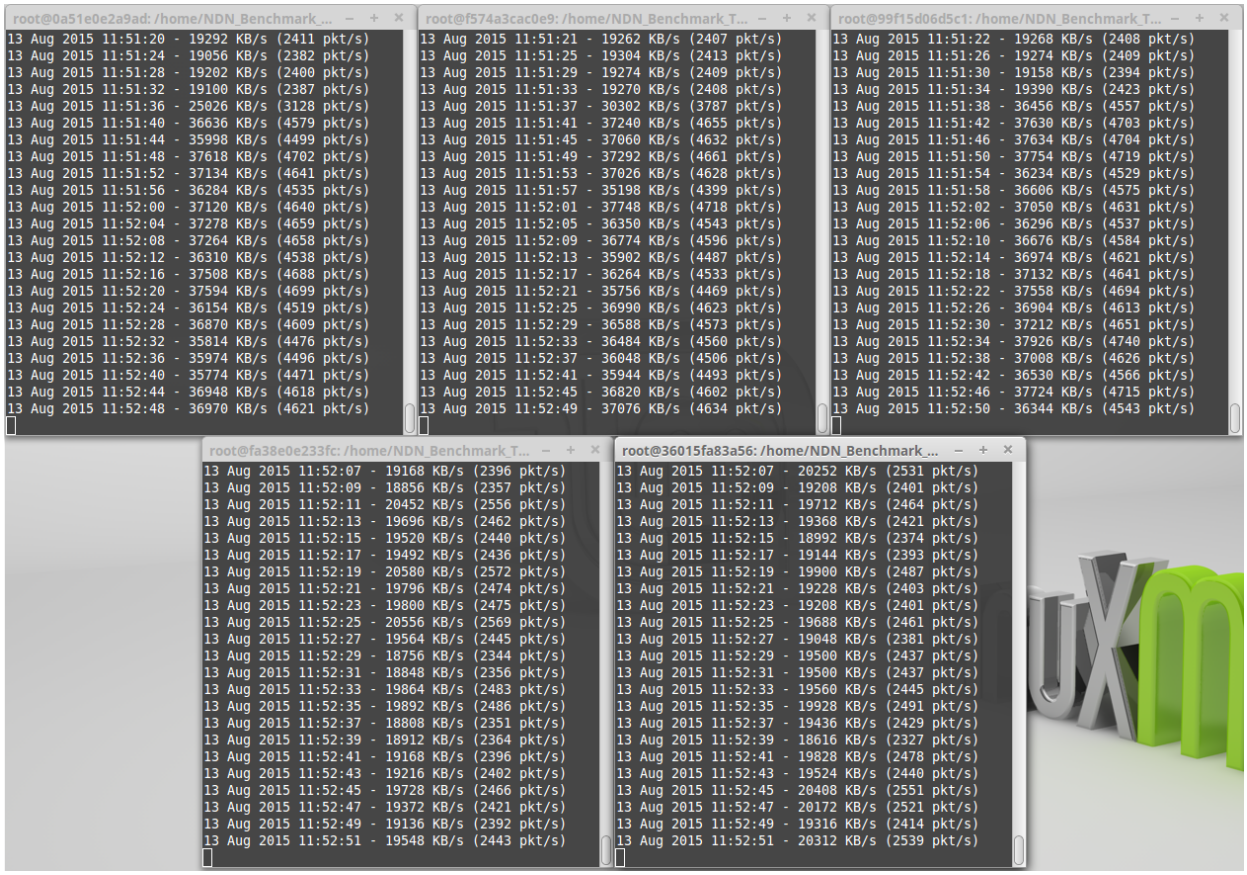


FIGURE C.3 – Test de débit de la stratégie *Broadcast* avec 3 serveurs et 2 clients

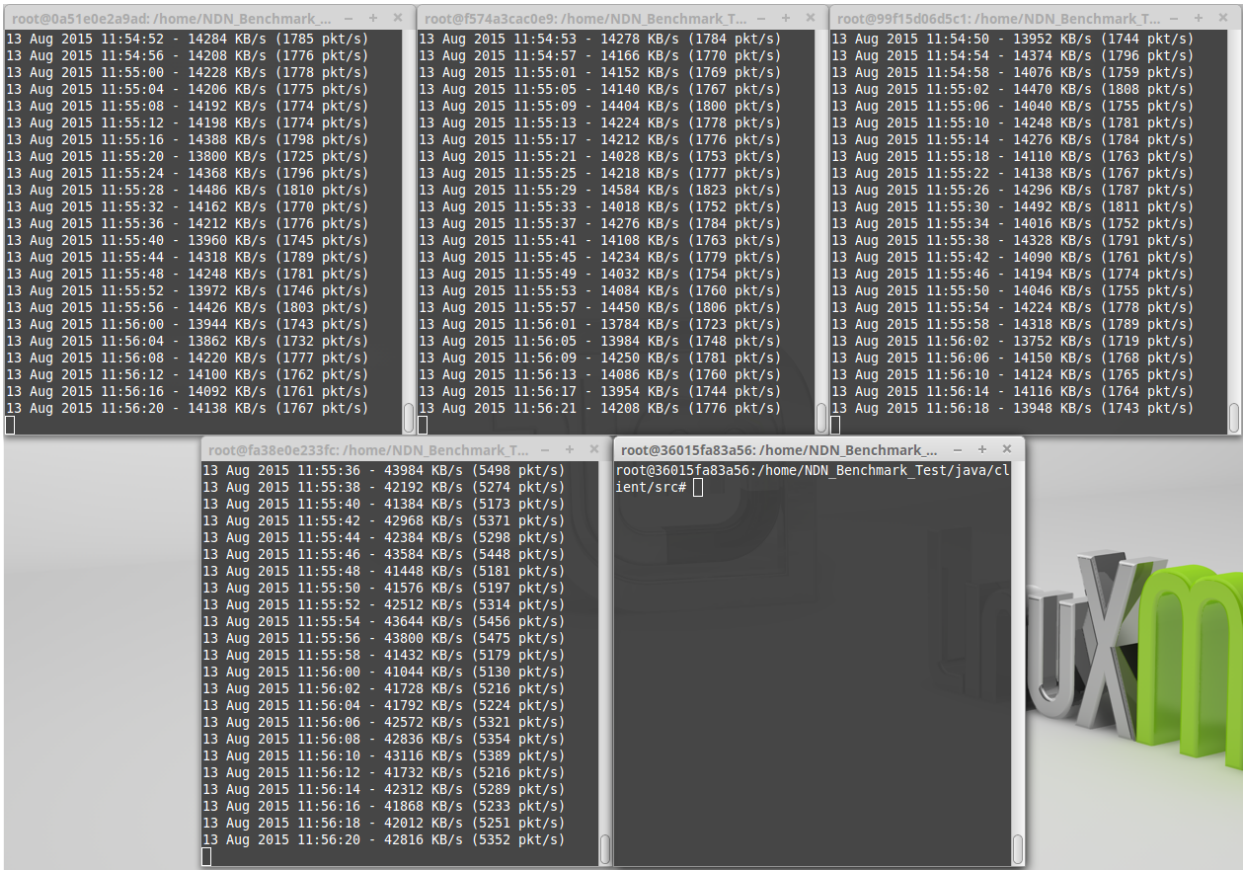


FIGURE C.4 – Test de débit de la stratégie d'équilibrage de charger avec 3 serveurs et 1 client

D Code du script pour la création d'un nouveau veth entre 2 conteneurs

Listing D.1 – Code du script pour la création d'un nouveau veth entre 2 conteneurs

```
#! /bin/bash

if [[ $(/usr/bin/id -u) -ne 0 ]]; then
    echo "please run this as root"
    exit
fi

if [ "$#" -ge 5 ]
then
        if [ "$3" -gt 16 ]
        then
                #recuperation des pid des containers
                pid1=$(docker inspect -f '{{.State.Pid}}' "$1")
                pid2=$(docker inspect -f '{{.State.Pid}}' "$2")

                if [ ! -d /var/run/netns ]
                then
                        mkdir /var/run/netns
                fi

                #creation des namespaces
                ln -s "/proc/$pid1/ns/net" "/var/run/netns/$pid1" &> /dev/null
                ln -s "/proc/$pid2/ns/net" "/var/run/netns/$pid2" &> /dev/null

                #calcul du sous reseau
                if [ "$3" -gt 30 ]
                then
                        c=30
                else
                        c=$3
                fi
                z=4
                for (( i=$(( 30 - $c )); i > 0; i-- ))
                do
                        z=$(( $z * 2 ))
                done
                m=$z
                z=$(( $z * $4 ))
                m=$(( $m + $z - 1 ))
                y=$(( $z / 256 ))
                x=$(( $z % 256 ))
                my=$(( $m / 256 ))
                mx=$(( $m % 256 ))
                echo "in subnet 172.17.$y.$x/$c:"

```

```
46                  #creation du lien
47                  if [ "$#" -ge 6 ]
48                  then
49                          B=$6
50                  else
51                          B=$5
52                  fi
53
54                  ip link add "A$c-$4-$5-$B" type veth peer name "B$c-$4-$5-$B"
55
56                  #definition des points
57                  ax=$(( $x + $5 ))
58                  ay=$y
59                  if [ "$ax" -ge 256 ]
60                  then
61                          ay=$(( $ay + $ax / 256 ))
62                          ax=$(( $ax % 256 ))
63                  fi
64
65                  ip link set "A$c-$4-$5-$B" netns "$pid1"
66                  ip netns exec "$pid1" ip addr add "172.17.$ay.$ax/$c"
                        broadcast "172.17.$my.$mx" dev "A$c-$4-$5-$B"
67                  ip netns exec "$pid1" ip link set "A$c-$4-$5-$B" up
68                  echo -e "\tset IP = 172.17.$ay.$ax/$c to $1"
69                  bx=$(( $x + $6 ))
70                  by=$y
71                  if [ "$bx" -ge 256 ]
72                  then
73                          by=$(( $by + $bx / 256 ))
74                          bx=$(( $bx % 256 ))
75                  fi
76
77                  ip link set "B$c-$4-$5-$B" netns "$pid2"
78                  ip netns exec "$pid2" ip link set "B$c-$4-$5-$B" up
79                  if [ "$B" -ne "$5" ]
80                  then
81                          bx=$(( $x + $6 ))
82                          by=$y
83                          if [ "$bx" -ge 256 ]
84                          then
85                                  by=$(( $by + $bx / 256 ))
86                                  bx=$(( $bx % 256 ))
87                          fi
88                          ip netns exec "$pid2" ip addr add "172.17.$ay.$bx/$c"
                                broadcast "172.17.$my.$mx" dev "B$c-$4-$5-$B"
89                          echo -e "\tset IP = 172.17.$by.$bx/$c to $2"
90                  fi
91          fi
92  else
93          echo "usage ./ppp_link_docker.sh <container_name/id> <container_name/
                id> <CIDR (17 to 30)> <subnet_ID> <container_subnet_rank> [<
                container_subnet_rank> (don't fill for switchs)]"
94  fi
```

E Code du script pour déployer le réseau virtuel

Listing E.1 – Code du script pour déployer le réseau virtuel

```
#! /bin/bash

if [[ $(/usr/bin/id -u) -ne 0 ]]; then
    echo "please run this as root"
    exit
fi

image="ndn"
version="v1"
image_file=$image$version"_Dockerfile"

# check si l'image existe
if [[ -z $(docker images | awk "\$1 == \"$image\" && \$2 == \"$version\" { print \"ok\" }") ]]
then
        echo -n "-> building the needed docker image..."
        docker build --force-rm=true -t $image:$version -f $image_file . &> /dev/null || (echo -e "\\033[1;31mFAIL\\033[0;39m" && echo "please verify the config file: $image_file" && exit)
        echo -e "\\033[1;32mDONE\\033[0;39m"
fi

# instanciation des conteneurs
containers=("DOCTOR_network_1_igw" "DOCTOR_network_1_egw" "DOCTOR_network_1_ndn1" "DOCTOR_network_1_ndn2" "DOCTOR_network_1_ndn3")
echo "-> checking the containers state..."
for container in ${containers[@]}
do
        # check si le conteneur est deja actif
        if [[ -z $(docker ps | awk "\$NF == \"$container\" { print \"ok\" }") ]]
        then
                # check si le conteneur est deja instancie
                if [[ -z $(docker ps -a | awk "\$NF == \"$container\" { print \"ok\" }") ]]
                then
                        echo -e "\tinstanciate a new container with the name $container"
                        docker run  -itd --name $container $image:$version > /dev/null
                else
                        echo -e "\ta container with the name $container exist, startup of this container"
                        docker start $container > /dev/null
                fi
        else
                echo -e "\t$container is already running"
```

```
39          fi
40  done
41
42  # creation des connections
43  echo "-> checking the containers network interfaces..."
44  # igw
45  if [[ -z $(docker exec ${containers[0]} ifconfig -a | awk '$1 == "A30-64-1-2"
        { print "ok" }') ]]
46  then
47          ./new_docker_veth.sh ${containers[0]} ${containers[2]} 30 64 1 2
48  else
49          if [[ -z $(docker exec ${containers[0]} ifconfig | awk '$1 == "A30
                -64-1-2" { print "ok" }') ]]
50          then
51                  pid=$(docker inspect -f '{{.State.Pid}}' "${containers[0]}")
52                  ip netns exec "$pid" ip link set A30-64-1-2 up
53                  ip netns exec "$pid" ip addr add 172.17.1.1/30 broadcast
                        172.17.1.3 dev A30-64-1-2
54          else
55                  if [[ -z $(docker exec ${containers[0]} ifconfig A30-64-1-2 |
                        grep 172.17.1.1) ]]
56                  then
57                          pid=$(docker inspect -f '{{.State.Pid}}' "${containers
                                [0]}")
58                          ip netns exec "$pid" ip addr add 172.17.1.1/30
                                broadcast 172.17.1.3 dev A30-64-1-2
59                  fi
60          fi
61  fi
62  echo -n -e "verify needed routes for ${containers[0]}..."
63  pid=$(docker inspect -f '{{.State.Pid}}' "${containers[0]}")
64  if [[ -z $(docker exec ${containers[0]} route -n | awk '$1 == "172.17.1.4" {
        print "ok" }') ]]
65  then
66          ip netns exec "$pid" ip route add 172.17.1.4/30 via 172.17.1.2
67  fi
68  if [[ -z $(docker exec ${containers[0]} route -n | awk '$1 == "172.17.1.8" {
        print "ok" }') ]]
69  then
70  ip netns exec "$pid" ip route add 172.17.1.8/30 via 172.17.1.2
71  fi
72  if [[ -z $(docker exec ${containers[0]} route -n | awk '$1 == "172.17.1.12" {
        print "ok" }') ]]
73  then
74  ip netns exec "$pid" ip route add 172.17.1.12/30 via 172.17.1.2
75  fi
76  if [[ -z $(docker exec ${containers[0]} route -n | awk '$1 == "172.17.1.16" {
        print "ok" }') ]]
77  then
78  ip netns exec "$pid" ip route add 172.17.1.16/30 via 172.17.1.2
79  fi
80  echo -e "\\033[1;32mDONE\\033[0;39m"
81
82  # ndn1
83  if [[ -z $(docker exec ${containers[2]} ifconfig -a | awk '$1 == "A30-65-1-2"
        { print "ok" }') ]]
84  then
85          ./new_docker_veth.sh ${containers[2]} ${containers[3]} 30 65 1 2
86  else
```

```
87          if [[ -z $(docker exec ${containers[2]} ifconfig | awk '$1 == "A30
                -65-1-2" { print "ok" }') ]]
88          then
89                  pid=$(docker inspect -f '{{.State.Pid}}' "${containers[2]}")
90                  ip netns exec "$pid" ip link set A30-65-1-2 up
91                  ip netns exec "$pid" ip addr add 172.17.1.5/30 broadcast
                        172.17.1.7 dev A30-65-1-2
92          else
93                  if [[ -z $(docker exec ${containers[2]} ifconfig A30-65-1-2 |
                        grep 172.17.1.5) ]]
94                  then
95                          pid=$(docker inspect -f '{{.State.Pid}}' "${containers
                                [2]}")
96                          ip netns exec "$pid" ip addr add 172.17.1.5/30
                                broadcast 172.17.1.7 dev A30-65-1-2
97                  fi
98          fi
99  fi
100
101 if [[ -z $(docker exec ${containers[2]} ifconfig -a | awk '$1 == "A30-66-1-2"
        { print "ok" }') ]]
102 then
103         ./new_docker_veth.sh ${containers[2]} ${containers[4]} 30 66 1 2
104 else
105         if [[ -z $(docker exec ${containers[2]} ifconfig | awk '$1 == "A30
                -66-1-2" { print "ok" }') ]]
106         then
107                 pid=$(docker inspect -f '{{.State.Pid}}' "${containers[2]}")
108                 ip netns exec "$pid" ip link set A30-66-1-2 up
109                 ip netns exec "$pid" ip addr add 172.17.1.9/30 broadcast
                        172.17.1.11 dev A30-66-1-2
110         else
111                 if [[ -z $(docker exec ${containers[2]} ifconfig A30-66-1-2 |
                        grep 172.17.1.9) ]]
112                 then
113                         pid=$(docker inspect -f '{{.State.Pid}}' "${containers
                                [2]}")
114                         ip netns exec "$pid" ip addr add 172.17.1.9/30
                                broadcast 172.17.1.11 dev A30-66-1-2
115                 fi
116         fi
117 fi
118 echo -n -e "verify needed routes for ${containers[2]}..."
119 pid=$(docker inspect -f '{{.State.Pid}}' "${containers[2]}")
120 if [[ -z $(docker exec ${containers[2]} route -n | awk '$1 == "172.17.1.12" {
        print "ok" }') ]]
121 then
122 ip netns exec "$pid" ip route add 172.17.1.12/30 via 172.17.1.6
123 fi
124 if [[ -z $(docker exec ${containers[2]} route -n | awk '$1 == "172.17.1.16" {
        print "ok" }') ]]
125 then
126 ip netns exec "$pid" ip route add 172.17.1.16/30 via 172.17.1.10
127 fi
128 echo -e "\\033[1;32mDONE\\033[0;39m"
129
130 if [[ -z $(docker exec ${containers[2]} ifconfig | awk '$1 == "B30-64-1-2" {
        print "ok" }') ]]
131 then
132         pid=$(docker inspect -f '{{.State.Pid}}' "${containers[2]}")
```

```
133         ip netns exec "$pid" ip link set B30-64-1-2 up
134         ip netns exec "$pid" ip addr add 172.17.1.2/30 broadcast 172.17.1.3
                dev B30-64-1-2
135 else
136         if [[ -z $(docker exec ${containers[2]} ifconfig B30-64-1-2 | grep
                172.17.1.2) ]]
137         then
138                 pid=$(docker inspect -f '{{.State.Pid}}' "${containers[2]}")
139                 ip netns exec "$pid" ip addr add 172.17.1.2/30 broadcast
                        172.17.1.3 dev B30-64-1-2
140         fi
141 fi
142
143 # ndn2
144 if [[ -z $(docker exec ${containers[3]} ifconfig -a | awk '$1 == "A30-67-1-2"
        { print "ok" }') ]]
145 then
146         ./new_docker_veth.sh ${containers[3]} ${containers[1]} 30 67 1 2
147 else
148         if [[ -z $(docker exec ${containers[3]} ifconfig | awk '$1 == "A30
                -67-1-2" { print "ok" }') ]]
149         then
150                 pid=$(docker inspect -f '{{.State.Pid}}' "${containers[3]}")
151                 ip netns exec "$pid" ip link set A30-67-1-2 up
152                 ip netns exec "$pid" ip addr add 172.17.1.13/30 broadcast
                        172.17.1.15 dev A30-67-1-2
153         else
154                 if [[ -z $(docker exec ${containers[3]} ifconfig A30-67-1-2 |
                        grep 172.17.1.13) ]]
155                 then
156                         pid=$(docker inspect -f '{{.State.Pid}}' "${containers
                                [3]}")
157                         ip netns exec "$pid" ip addr add 172.17.1.13/30
                                broadcast 172.17.1.15 dev A30-67-1-2
158                 fi
159         fi
160 fi
161
162 if [[ -z $(docker exec ${containers[3]} ifconfig | awk '$1 == "B30-65-1-2" {
        print "ok" }') ]]
163 then
164         pid=$(docker inspect -f '{{.State.Pid}}' "${containers[3]}")
165         ip netns exec "$pid" ip link set "B30-65-1-2" up
166         ip netns exec "$pid" ip addr add 172.17.1.6/30 broadcast 172.17.1.7
                dev B30-65-1-2
167 else
168         if [[ -z $(docker exec ${containers[3]} ifconfig B30-65-1-2 | grep
                172.17.1.6) ]]
169         then
170                 pid=$(docker inspect -f '{{.State.Pid}}' "${containers[3]}")
171                 ip netns exec "$pid" ip addr add 172.17.1.6/30 broadcast
                        172.17.1.7 dev B30-65-1-2
172         fi
173 fi
174 echo -n -e "verify needed routes for ${containers[3]}..."
175 pid=$(docker inspect -f '{{.State.Pid}}' "${containers[3]}")
176 if [[ -z $(docker exec ${containers[3]} route -n | awk '$1 == "172.17.1.0" {
        print "ok" }') ]]
177 then
178 ip netns exec "$pid" ip route add 172.17.1.0/30 via 172.17.1.5
```

```
179 fi
180 if [[ -z $(docker exec ${containers[3]} route -n | awk '$1 == "172.17.1.8" {
    print "ok" }') ]]
181 then
182 ip netns exec "$pid" ip route add 172.17.1.8/30 via 172.17.1.5
183 fi
184 if [[ -z $(docker exec ${containers[3]} route -n | awk '$1 == "172.17.1.16" {
    print "ok" }') ]]
185 then
186 ip netns exec "$pid" ip route add 172.17.1.16/30 via 172.17.1.5
187 fi
188 echo -e "\\033[1;32mDONE\\033[0;39m"
189
190 # ndn3
191 if [[ -z $(docker exec ${containers[4]} ifconfig -a | awk '$1 == "A30-68-1-2"
    { print "ok" }') ]]
192 then
193         ./new_docker_veth.sh ${containers[4]} ${containers[1]} 30 68 1 2
194 else
195         if [[ -z $(docker exec ${containers[4]} ifconfig | awk '$1 == "A30
            -68-1-2" { print "ok" }') ]]
196         then
197                 pid=$(docker inspect -f '{{.State.Pid}}' "${containers[4]}")
198                 ip netns exec "$pid" ip link set A30-68-1-2 up
199                 ip netns exec "$pid" ip addr add 172.17.1.17/30 broadcast
                    172.17.1.19 dev A30-68-1-2
200         else
201                 if [[ -z $(docker exec ${containers[4]} ifconfig A30-68-1-2 |
                    grep 172.17.1.17) ]]
202                 then
203                         pid=$(docker inspect -f '{{.State.Pid}}' "${containers
                            [4]}")
204                         ip netns exec "$pid" ip addr add 172.17.1.17/30
                            broadcast 172.17.1.19 dev A30-68-1-2
205                 fi
206         fi
207 fi
208
209 if [[ -z $(docker exec ${containers[4]} ifconfig | awk '$1 == "B30-66-1-2" {
    print "ok" }') ]]
210 then
211         pid=$(docker inspect -f '{{.State.Pid}}' "${containers[4]}")
212         ip netns exec "$pid" ip link set B30-66-1-2 up
213         ip netns exec "$pid" ip addr add 172.17.1.10/30 broadcast 172.17.1.11
            dev B30-66-1-2
214
215 else
216         if [[ -z $(docker exec ${containers[4]} ifconfig B30-66-1-2 | grep
            172.17.1.10) ]]
217         then
218                 pid=$(docker inspect -f '{{.State.Pid}}' "${containers[4]}")
219                 ip netns exec "$pid" ip addr add 172.17.1.10/30 broadcast
                    172.17.1.11 dev B30-66-1-2
220         fi
221 fi
222 echo -n -e "verify needed routes for ${containers[4]}..."
223 pid=$(docker inspect -f '{{.State.Pid}}' "${containers[4]}")
224 if [[ -z $(docker exec ${containers[4]} route -n | awk '$1 == "172.17.1.0" {
    print "ok" }') ]]
225 then
```

```
ip netns exec "$pid" ip route add 172.17.1.0/30 via 172.17.1.9
fi
if [[ -z $(docker exec ${containers[4]} route -n | awk '$1 == "172.17.1.4" { print "ok" }') ]]
then
ip netns exec "$pid" ip route add 172.17.1.4/30 via 172.17.1.9
fi
if [[ -z $(docker exec ${containers[4]} route -n | awk '$1 == "172.17.1.12" { print "ok" }') ]]
then
ip netns exec "$pid" ip route add 172.17.1.12/30 via 172.17.1.9
fi
echo -e "\\033[1;32mDONE\\033[0;39m"

# egw
if [[ -z $(docker exec ${containers[1]} ifconfig | awk '$1 == "B30-67-1-2" { print "ok" }') ]]
then
        pid=$(docker inspect -f '{{.State.Pid}}' "${containers[1]}")
        ip netns exec "$pid" ip link set B30-67-1-2 up
        ip netns exec "$pid" ip addr add 172.17.1.14/30 broadcast 172.17.1.15 dev B30-67-1-2

else
        if [[ -z $(docker exec ${containers[1]} ifconfig B30-67-1-2 | grep 172.17.1.14) ]]
        then
                pid=$(docker inspect -f '{{.State.Pid}}' "${containers[1]}")
                ip netns exec "$pid" ip addr add 172.17.1.14/30 broadcast 172.17.1.15 dev B30-67-1-2
        fi
fi

if [[ -z $(docker exec ${containers[1]} ifconfig | awk '$1 == "B30-68-1-2" { print "ok" }') ]]
then
        pid=$(docker inspect -f '{{.State.Pid}}' "${containers[1]}")
        ip netns exec "$pid" ip link set B30-68-1-2 up
        ip netns exec "$pid" ip addr add 172.17.1.18/30 broadcast 172.17.1.19 dev B30-68-1-2
else
        if [[ -z $(docker exec ${containers[1]} ifconfig B30-68-1-2 | grep 172.17.1.18) ]]
        then
                pid=$(docker inspect -f '{{.State.Pid}}' "${containers[1]}")
                ip netns exec "$pid" ip addr add 172.17.1.18/30 broadcast 172.17.1.19 dev B30-68-1-2
        fi
fi
echo -n -e "verify needed routes for ${containers[1]}..."
pid=$(docker inspect -f '{{.State.Pid}}' "${containers[1]}")
if [[ -z $(docker exec ${containers[1]} route -n | awk '$1 == "172.17.1.0" && $2 == "172.17.1.13" { print "ok" }') ]]
then
        ip netns exec "$pid" ip route add 172.17.1.0/30 via 172.17.1.13 metric 1
fi
if [[ -z $(docker exec ${containers[1]} route -n | awk '$1 == "172.17.1.0" && $2 == "172.17.1.17" { print "ok" }') ]]
then
```

```
273         ip netns exec "$pid" ip route add 172.17.1.0/30 via 172.17.1.17 metric
                2
274 fi
275 if [[ -z $(docker exec ${containers[1]} route -n | awk '$1 == "172.17.1.4" &&
        $2 == "172.17.1.13" { print "ok" }') ]]
276 then
277         ip netns exec "$pid" ip route add 172.17.1.4/30 via 172.17.1.13 metric
                1
278 fi
279 if [[ -z $(docker exec ${containers[1]} route -n | awk '$1 == "172.17.1.4" &&
        $2 == "172.17.1.17" { print "ok" }') ]]
280 then
281         ip netns exec "$pid" ip route add 172.17.1.4/30 via 172.17.1.17 metric
                2
282 fi
283 if [[ -z $(docker exec ${containers[1]} route -n | awk '$1 == "172.17.1.8" &&
        $2 == "172.17.1.13" { print "ok" }') ]]
284 then
285         ip netns exec "$pid" ip route add 172.17.1.8/30 via 172.17.1.13 metric
                2
286 fi
287 if [[ -z $(docker exec ${containers[1]} route -n | awk '$1 == "172.17.1.8" &&
        $2 == "172.17.1.17" { print "ok" }') ]]
288 then
289         ip netns exec "$pid" ip route add 172.17.1.8/30 via 172.17.1.17 metric
                1
290 fi
291 echo -e "\\033[1;32mDONE\\033[0;39m"
292
293 # demarrage des services NDN
294 echo -n "-> configuring NLRS..."
295 docker exec ${containers[0]} sh -c "sed 's/_name/igw/g' /home/nlsr.1 > /home/
        nlsr.conf"
296 docker exec ${containers[0]} sh -c "sed 's/_neighbor/ndn1/g' /home/nlsr.2 |
        sed 's/_ip/172.17.1.2/g' | sed 's/_cost/10/g' >> /home/nlsr.conf"
297 docker exec ${containers[0]} sh -c "sed 's/_prefix/prefix \/iGW/g' /home/nlsr
        .3 >> /home/nlsr.conf"
298
299 docker exec ${containers[1]} sh -c "sed 's/_name/egw/g' /home/nlsr.1 > /home/
        nlsr.conf"
300 docker exec ${containers[1]} sh -c "sed 's/_neighbor/ndn2/g' /home/nlsr.2 |
        sed 's/_ip/172.17.1.13/g' | sed 's/_cost/10/g' >> /home/nlsr.conf"
301 docker exec ${containers[1]} sh -c "sed 's/_neighbor/ndn3/g' /home/nlsr.2 |
        sed 's/_ip/172.17.1.17/g' | sed 's/_cost/10/g' >> /home/nlsr.conf"
302 docker exec ${containers[1]} sh -c "sed 's/_prefix/prefix \/eGW/g' /home/nlsr
        .3 >> /home/nlsr.conf"
303
304 docker exec ${containers[2]} sh -c "sed 's/_name/ndn1/g' /home/nlsr.1 > /home/
        nlsr.conf"
305 docker exec ${containers[2]} sh -c "sed 's/_neighbor/igw/g' /home/nlsr.2 | sed
         's/_ip/172.17.1.1/g' | sed 's/_cost/10/g' >> /home/nlsr.conf"
306 docker exec ${containers[2]} sh -c "sed 's/_neighbor/ndn2/g' /home/nlsr.2 |
        sed 's/_ip/172.17.1.6/g' | sed 's/_cost/10/g' >> /home/nlsr.conf"
307 docker exec ${containers[2]} sh -c "sed 's/_neighbor/ndn3/g' /home/nlsr.2 |
        sed 's/_ip/172.17.1.10/g' | sed 's/_cost/10/g' >> /home/nlsr.conf"
308 docker exec ${containers[2]} sh -c "sed 's/_prefix/ /g' /home/nlsr.3 >> /home/
        nlsr.conf"
309
310 docker exec ${containers[3]} sh -c "sed 's/_name/ndn2/g' /home/nlsr.1 > /home/
        nlsr.conf"
```

```
311 docker exec ${containers[3]} sh -c "sed 's/_neighbor/ndn1/g' /home/nlsr.2 |
    sed 's/_ip/172.17.1.5/g' | sed 's/_cost/10/g' >> /home/nlsr.conf"
312 docker exec ${containers[3]} sh -c "sed 's/_neighbor/egw/g' /home/nlsr.2 | sed
    's/_ip/172.17.1.14/g' | sed 's/_cost/10/g' >> /home/nlsr.conf"
313 docker exec ${containers[3]} sh -c "sed 's/_prefix/ /g' /home/nlsr.3 >> /home/
    nlsr.conf"
314
315 docker exec ${containers[4]} sh -c "sed 's/_name/ndn3/g' /home/nlsr.1 > /home/
    nlsr.conf"
316 docker exec ${containers[4]} sh -c "sed 's/_neighbor/ndn1/g' /home/nlsr.2 |
    sed 's/_ip/172.17.1.9/g' | sed 's/_cost/10/g' >> /home/nlsr.conf"
317 docker exec ${containers[3]} sh -c "sed 's/_neighbor/egw/g' /home/nlsr.2 | sed
    's/_ip/172.17.1.18/g' | sed 's/_cost/10/g' >> /home/nlsr.conf"
318 docker exec ${containers[4]} sh -c "sed 's/_prefix/prefix \/eGW/g' /home/nlsr
    .3 >> /home/nlsr.conf"
319 echo -e "\\033[1;32mDONE\\033[0;39m"
320 echo "-> starting the containers services..."
321 for container in ${containers[@]}
322 do
323         if [[ -z $(docker exec $container ps -e | grep atd) ]]
324         then
325                 docker exec $container service atd start &> /dev/null
326         fi
327         if [[ -z $(docker exec $container ps -e | grep nfd) ]]
328         then
329                 echo -n -e "\tstarting NFD in $container..."
330                 docker exec $container apt-get install -y ndn-cxx &> /dev/null
                        #fix nfd: error while loading shared libraries: libndn-
                    cxx.so.0.3.3
331                 docker exec $container apt-get remove -y ndn-cxx &> /dev/null
                        #fix nfd: error while loading shared libraries: libndn-
                    cxx.so.0.3.3
332                 docker exec $container bash -c 'echo "nfd-start" | at now &> /
                    dev/null'
333                 sleep 3
334                 if [[ -n $(docker exec $container ps -e | grep nfd) ]]
335                 then
336                         echo -e "\\033[1;32mDONE\\033[0;39m"
337                 else
338                         echo -e "\\033[1;31mFAIL\\033[0;39m"
339                 fi
340         fi
341         if [[ -z $(docker exec $container ps -e | grep nlsr) ]]
342         then
343                 echo -n -e "\tstarting NLSR in $container..."
344                 docker exec $container bash -c 'echo "nlsr -f /home/nlsr.conf"
                        | at now &> /dev/null'
345                 sleep 2
346                 if [[ -n $(docker exec $container ps -e | grep nlsr) ]]
347                 then
348                         echo -e "\\033[1;32mDONE\\033[0;39m"
349                 else
350                         echo -e "\\033[1;31mFAIL\\033[0;39m"
351                 fi
352         fi
353 done
354 if [[ -z $(docker exec ${containers[0]} ps -e | grep java) ]]
355 then
356         echo -e "please manually start the gateway in ${containers[0]} with \"
            chmod +x /home/Gateway_HTTP_NDN/vert.x-3.0.0-milestone4/bin/vertx;
```

```
            export LC_ALL=C.UTF-8; /home/Gateway_HTTP_NDN/vert.x-3.0.0-
            milestone4/bin/vertx run Ingressgw/src/ingressgw/Ingressgw.java\""
#       docker exec ${containers[0]} sh -c "cd /home/Gateway_HTTP_NDN; chmod +
    x /home/Gateway_HTTP_NDN/vert.x-3.0.0-milestone4/bin/vertx; export LC_ALL=
    C.UTF-8; /home/Gateway_HTTP_NDN/vert.x-3.0.0-milestone4/bin/vertx run
    Ingressgw/src/ingressgw/Ingressgw.java | at now &> /dev/null"
#       sleep 3
#       if [[ -n $(docker exec ${containers[0]} ps -e | grep java) ]]
#       then
#               echo -e "\\033[1;32mDONE\\033[0;39m"
#       else
#               echo -e "\\033[1;31mFAIL\\033[0;39m"
#       fi
fi
if [[ -z $(docker exec ${containers[1]} ps -e | grep java) ]]
then
        echo -e "please manually start the gateway in ${containers[1]} with \"
            chmod +x /home/Gateway_HTTP_NDN/vert.x-3.0.0-milestone4/bin/vertx;
            export LC_ALL=C.UTF-8; /home/Gateway_HTTP_NDN/vert.x-3.0.0-
            milestone4/bin/vertx run Egressgw/src/egressgw/Egressgw.java\""
#       docker exec ${containers[1]} sh -c "cd /home/Gateway_HTTP_NDN; chmod +
    x /home/Gateway_HTTP_NDN/vert.x-3.0.0-milestone4/bin/vertx; export LC_ALL=
    C.UTF-8; vert.x-3.0.0-milestone4/bin/vertx run Egressgw/src/egressgw/
    Egressgw.java  | at now &> /dev/null"
#       sleep 3
#       if [[ -n $(docker exec ${containers[1]} ps -e | grep java) ]]
#       then
#               echo -e "\\033[1;32mDONE\\033[0;39m"
#       else
#               echo -e "\\033[1;31mFAIL\\033[0;39m"
#       fi
fi
echo -e "\\033[1;32mALL DONE\\033[0;39m The network is now ready!"
```

Résumé

Durant mon stage, j'ai été associé au projet ANR DOCTOR qui vise à favoriser le déploiement de nouvelles solutions réseaux dans des infrastructures virtualisées. Après une étude de l'état de l'art sur les différentes technologies sélectionnées par le consortium du projet, j'ai effectué des tests de performances sur les différentes technologies de virtualisation (OpenStack/Docker) ainsi que du protocole NDN. Ces tests ont permis au consortium d'avoir une meilleure appréciation des performances réseaux de ces technologies ainsi que diverses idées d'amélioration pour le développement de leurs *gateway* NDN/HTTP présentent dans l'outil que j'ai développé pour l'étude de performances du protocole NDN. Cette outil est aussi disponible pour le reste de la communauté via un dépôt github. Ces tests mon aussi permis de dimensionner les serveurs qui représenteront la partie du testbed du LORIA. Après réception des serveurs j'ai pu procéder à leurs installation ainsi qu'à la mise en place d'un premier réseau NDN virtuel dans Docker à l'aide d'un script que j'ai codé qui pourra servir de base pour d'autres architectures de réseaux plus complexes.

Mots-clés : évaluation de performances, NFV, NDN

Abstract

During my internship, I was associated with the ANR project DOCTOR that aims to foster the deployment of new network solutions in virtualized environment. After a study of the state of the art on the technologies selected by the consortium of the project, I conducted some performance evaluations on different virtualization technologies (OpenStack/Docker) and the NDN protocol. The results off these evaluations enabled the consortium to have a better appreciation of the performance of these solutions and provide them some code improvement ideas for the development of their NDN/HTTP gateway that are present in the tool I developed for the performance evaluation of the NDN protocol. This tool is also available for the rest of the community via a github repository. These tests also allowed me to size the servers of the LORIA testbed. After receipt of the servers, I proceeded to the installation and configuration these servers but also deploy a first virtual NDN network in a Docker environment by using a script I coded. This script can serve as a basis for other and more network architectures.

Keywords : performance evaluation, NFV, NDN