

UNIVERSITÉ JOSEPH FOURIER DE GRENOBLE

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER DE GRENOBLE

Spécialité : "Informatique : Systèmes et Communications"

PRÉPARÉE AU LABORATOIRE D'INFORMATIQUE DE GRENOBLE
DANS LE CADRE DE *l'École Doctorale "Mathématiques, Sciences et
Technologies de l'Information, Informatique"*

PRÉSENTÉE ET SOUTENUE PUBLIQUEMENT

PAR

LUCAS NUSSBAUM

LE 4 DÉCEMBRE 2008

CONTRIBUTIONS À L'EXPÉRIMENTATION
SUR LES SYSTÈMES DISTRIBUÉS
DE GRANDE TAILLE

Directeurs de thèse :

M. Jean-François Méhaut

M. Olivier Richard

JURY

M. Didier DONSEZ

M. Franck CAPPELLO

M. Jean-François MÉHAUT

M. Olivier RICHARD

M. Pierre SENS

Mme Pascale VICAT-BLANC PRIMET

Président du jury

Rapporteur

Directeur de thèse

Co-encadrant

Rapporteur

Examinateur

*In theory, there is no difference between theory and practice.
But, in practice, there is.*

Yogi Berra¹

¹Ancien joueur et manager de baseball, aussi connu pour ses nombreux *Yogiisms*, comme « *It's tough making predictions, especially about the future.* » ou « *You can observe a lot by watching.* ».

Remerciements

De mon master 2 à l'écriture de ces remerciements, le travail autour de cette thèse aura duré presque quatre ans. Ces travaux n'auraient pas été possibles sans l'aide et le soutien de nombreuses personnes.

Tout d'abord, je tiens à remercier les membres du jury pour avoir accepté d'évaluer ces travaux au milieu d'une période dense en soutenances de thèse. Ils ont fait preuve de souplesse dans les semaines qui ont précédé la soutenance pour permettre son organisation, et je leur en suis très reconnaissant. Malheureusement (ou heureusement?), on ne soutient sa thèse qu'une fois, donc les membres du jury sont toujours contraints d'essayer les plâtres d'une organisation un peu cahotique...

Je tiens aussi à remercier Olivier Richard, qui m'a proposé de travailler sur cette thématique, et m'a fait découvrir les différentes facettes du métier d'enseignant-chercheur tout en me laissant beaucoup de liberté (dont, comme beaucoup de thésards, j'ai peut-être un peu trop profité). Même si je n'ai que peu travaillé avec lui, Jean-François Méhaut, mon directeur de thèse "officiel", a toujours sû être là quand j'en avais besoin.

Mais plus généralement, il me faut remercier toutes les personnes que j'ai eu l'occasion de côtoyer au cours de ces quatre années, que ce soit au département Informatique de l'IUT2, où j'enseignais, ou au laboratoire ID-IMAG, devenu pendant ma thèse l'antenne de Montbonnot du Laboratoire d'Informatique de Grenoble. Votre bonheur de travailler dans l'enseignement et la recherche ont fini, malgré quelques hésitations, de me convaincre de continuer dans cette voie. Plus particulièrement, j'ai partagé bien plus qu'un bureau avec mes co-bureaux du bureau 118 : Adrien au début de ma thèse, puis Hamza et notre futur ministre des NTIC, Yanik.

Enfin, merci à ma famille, de m'avoir donné la chance d'avoir pu faire de si longues études. Et merci à toi, Cécile, pour avoir sû me soutenir et me supporter, même dans les passages difficiles de la rédaction.

TABLE DES MATIÈRES

1	Introduction	11
1.1	Contexte	11
1.2	Positionnement	13
1.3	Organisation du manuscrit	13
2	État de l'art	15
2.1	Introduction	15
2.2	Simulateurs	16
2.2.1	Simulateurs de grille et de systèmes pair-à-pair	16
2.2.2	Network Simulator - ns-2	16
2.3	Plates-formes d'expérimentation	17
2.3.1	Emulab	17
2.3.2	Grid'5000	19
2.3.3	PlanetLab et OneLab	20
2.3.4	DSLlab	22
2.3.5	Conclusion	22
2.4	Émulateurs	22
2.4.1	Émulateurs de liens réseaux	22
2.4.2	MicroGrid	23
2.4.3	VINI	23
2.4.4	Modelnet	24
2.4.5	Wrekavoc	26
2.4.6	V-DS	26
2.4.7	DieCast	26
2.4.8	Autres émulateurs de topologies réseaux	27
2.5	Conclusion	28
3	Étude comparative des émulateurs réseaux	29

3.1	Introduction	29
3.2	Émulateurs réseaux	30
3.3	Fonctionnalités	31
3.4	Évaluation des performances	31
3.4.1	Configuration expérimentale	31
3.4.2	Source de temps, précision et émulation de latence	33
3.4.3	Limitation de la bande passante	37
3.5	Différences fonctionnelles	41
3.5.1	Interfaces utilisateur	41
3.5.2	Point d'interception des paquets avec TC	41
3.6	Conclusion	42
4	Conduite d'expériences avec un émulateur	45
4.1	Introduction	45
4.2	Latence et shells distants	45
4.3	Performance d'outils de transfert de fichiers	47
4.4	Émulation de topologies réseaux	47
4.5	Conclusion	51
5	Tunnel IP sur DNS robuste	53
5.1	Introduction	53
5.2	Canaux cachés dans le DNS	54
5.2.1	Principe général	54
5.2.2	Implémentations existantes	55
5.2.3	Conclusion	56
5.3	TUNS	57
5.4	Évaluation des performances	59
5.4.1	Influence de la latence	60
5.4.2	Performances sous conditions réseaux dégradées	63
5.5	Adaptation des paramètres du tunnel	63
5.6	Perspectives	66
5.7	Conclusion	67

6	P2PLab	69
6.1	Introduction	69
6.2	P2PLab : présentation générale	70
6.3	Virtualisation	71
6.3.1	Exécution concurrente d'un nombre important de processus par machine	73
6.3.2	Virtualisation de l'identité réseau des processus	77
6.4	Émulation de topologies réseaux	80
6.5	Conclusion	83
7	P2PLab : Validation et Expérimentations	85
7.1	Valider une plate-forme d'émulation	85
7.2	Validation de P2PLab à l'aide d'expériences sur BitTorrent	86
7.2.1	Rapport de virtualisation	87
7.2.2	Passage à l'échelle	89
7.3	Comparaison de différentes implantations de BitTorrent	91
7.4	Conclusion	92
8	Conclusion et perspectives	93
8.1	Contexte d'étude et difficultés	93
8.2	Contributions	94
8.3	Perspectives	95

INTRODUCTION

1

1.1	Contexte	11
1.2	Positionnement	13
1.3	Organisation du manuscrit	13

1.1 Contexte

A chaque évolution technologique en informatique, il est nécessaire de reconcevoir l'ensemble des outils utilisés lors du développement : débogage, traçage, environnements de test et d'expérimentation, notamment pour la mise au point et la mesure de performances. Les systèmes et applications distribués, conçus pour s'exécuter sur plusieurs machines éventuellement hétérogènes et réparties géographiquement, ne font pas exception à la règle, et apportent leur lot de difficultés.

Depuis la démocratisation de l'accès à Internet, au milieu des années 90, le nombre de ressources connectées entre-elles via le réseau a évolué de manière très impressionnante. Les équipements terminaux sont de plus en plus nombreux, de plus en plus performants, mais aussi de plus en plus différents, allant du téléphone portable au super-calculateur, en passant par la console de jeu. Les connexions réseaux elles-mêmes se sont diversifiées : les utilisateurs utilisent des réseaux filaires comme l'ADSL, le câble et la fibre optique, mais aussi aux technologies sans-fil comme la 3G, le Wifi ou le Wimax. Tous ces équipements terminaux et réseaux ont leurs propres spécificités, avantages et inconvénients, rendant leur utilisation coinjointe, par exemple dans le cadre d'une application distribuée, très complexe.

Dans le monde plus conservateur, en comparaison, du calcul à hautes performances, les choses évoluent aussi. Si le paradigme de la grille de calcul [1] semble ne jamais avoir vraiment percé hors des sphères académiques, les problématiques de consommation énergétique des centres de calcul traditionnels semblent convaincre qu'il est nécessaire d'évoluer [2]. Les infrastructures de *Cloud Computing* [3], en dématérialisant les ressources de calcul, et en les rendant plus souples, semblent apporter une réponse intéressante. Mais l'exploitation de ces infrastructures par des applications à grande échelle est loin d'être simple : il faut pouvoir combiner des ressources éventuellement distribuées géographiquement, éventuellement différentes. Ces problèmes sont finalement proches de ceux ren-

contrôlés par les applications distribuées s'exécutants sur les terminaux de consommateurs.

D'autant plus que dans l'idéal, on souhaite viser une convergence entre les ressources dispersées chez des utilisateurs d'internet, et des ressources plus stables et contrôlées situées dans des centres de calcul traditionnels. Cela permettrait de bénéficier de l'extraordinaire puissance de calcul disponible sur des ressources non-contrôlées, tout en obtenant des garanties sur le rendement réel de l'application grâce aux ressources contrôlées.

Mais pour développer de telles applications, il faut être capable d'apporter des réponses à des problèmes très durs. Si beaucoup de problèmes en informatique sont simplement résolus grâce à l'évolution quasi-naturelle des performances (cf Loi de Moore [4]), on rencontre déjà dans le cadre des applications distribuées des limites physiques, notamment en réseau : il n'y a pas réellement de limite à l'augmentation de la bande passante, mais la latence est souvent déjà très proche du minimum imposé par la vitesse de propagation de la lumière. Puisqu'aucune réduction de la latence n'est à attendre, la seule voie possible est de la prendre en compte, telle qu'elle est, dans le développement des applications, en utilisant des techniques complexes pour la masquer.

Devant la complexité de telles infrastructures et applications, la nécessité d'avoir des outils pour développer, comprendre et tester les applications s'exécutant sur de telles plate-formes est évidente : avant de déployer une application sur des milliers de noeuds, il est nécessaire de pouvoir s'assurer que le fonctionnement de l'application répondra aux attentes, sous peine de se retrouver face à de très nombreuses pannes [5]. Ces outils doivent être capables de prendre en compte l'échelle des systèmes étudiés (une étude sur une dizaine de noeuds, d'une application qui devra s'exécuter sur des milliers de noeuds, n'apporte pas grand chose), et la diversité matérielle, logicielle et réseau des noeuds. On pourra ainsi reproduire des problèmes qui ne se produisent que sous une charge ou à une échelle importante [6], ou lorsque plusieurs éléments tombent en panne simultanément [7].

Ces outils sont forcés de faire des compromis entre le réalisme et le *coût* (au sens large) de cette évaluation préalable : si elle était trop coûteuse (en temps, en machines), elle ne serait pas *rentable*. Toutefois, il est important de maîtriser ce compromis, afin que l'utilisateur connaisse clairement la portée et les limites des résultats qu'il obtient.

On peut classer ces outils en trois catégories :

Les solutions reposant sur une modélisation mathématique suivie d'une approche analytique et/ou de simulations. Il s'agit ici de modéliser l'application étudiée, puis, soit d'utiliser une approche analytique à l'aide d'outils mathématiques, soit d'utiliser des simulations pour étudier expérimentalement son comportement. Cette solution a d'énormes avantages, et est très largement utilisée : elle permet d'arriver rapidement à un résultat, ne nécessite

pas de ressources très importantes (même si le temps de calcul d'une simulation peut poser problème), et ne nécessite pas d'implémenter l'application à étudier, puisqu'elle fait abstraction des problèmes d'implémentations pour se concentrer sur des aspects algorithmiques, de plus haut niveau. Mais son positionnement loin de l'application finale, et l'obligation de réaliser des compromis assez importants lors de la modélisation, fait que le domaine d'application (et le réalisme) des résultats obtenus reste limité.

Les plateformes d'expérimentation consistent en l'utilisation de ressources bien réelles, de préférence nombreuses, pour fournir aux utilisateurs la possibilité d'exécuter de manière simple leur application sur un nombre important de ressources, aussi proche que possible des ressources (matériel, réseau) sur lesquelles l'application finale sera exécutée. Toutefois, ces plateformes sont très chères (coût d'achat, consommation énergétique) et très difficiles à administrer (notamment à cause de la nécessité de distribuer les ressources géographiquement). De plus, les résultats obtenus sur une plateforme sont difficiles à généraliser : les plateformes d'expérimentation sont en général de taille bien inférieure à celle des plateformes cibles, et elles ne proposent qu'une seule configuration, pas forcément tout à fait représentative de celles des plateformes cibles.

L'émulation propose une approche intermédiaire. Elle consiste à utiliser des ressources réelles, mais à altérer leurs caractéristiques pour permettre de réaliser des expériences dans des conditions différentes, plus proches de celles que l'application finale rencontrera. Elle est souvent couplée avec la virtualisation pour permettre d'augmenter artificiellement le nombre de ressources disponibles.

1.2 Positionnement

Cette thèse se focalise sur l'émulation. En effet, cette approche propose un compromis intéressant entre les approches mathématiques, qui proposent une vue abstraite et incomplète de la réalité, et les expérimentations *in vivo*, coûteuses financièrement et en temps.

L'émulation a l'avantage d'utiliser l'application réelle, et pourrait permettre, à long terme, de limiter le besoin en plates-formes d'expérimentation, en permettant de les adapter dynamiquement en fonction de l'expérience, afin qu'elles correspondent aux différentes conditions d'expériences souhaitées.

1.3 Organisation du manuscrit

Ce manuscrit s'articule en 8 chapitres. Le chapitre 2 présente un état de l'art des différentes solutions permettant d'évaluer un système distribué : les simula-

teurs, les plates-formes réelles, et finalement les émulateurs existants.

Le chapitre 3 s'intéresse aux émulateurs de liens réseaux. Nous y contribuons une étude comparative des principaux outils existants, puis, au chapitre 4, nous illustrons leur utilisation.

Au chapitre 5, nous nous intéressons à une application particulière, les tunnels IP sur DNS. Nous utilisons intensivement les émulateurs réseaux pour les évaluer.

Le chapitre 6 présente P2PLab, notre plate-forme pour l'émulation des systèmes pair-à-pair, tandis que le chapitre 7 en propose une validation expérimentale.

Enfin, le chapitre 8 tire un bilan général, et évoque les différentes perspectives ouvertes par ce travail.

2.1	Introduction	15
2.2	Simulateurs	16
2.2.1	Simulateurs de grille et de systèmes pair-à-pair	16
2.2.2	Network Simulator - ns-2	16
2.3	Plates-formes d'expérimentation	17
2.3.1	Emulab	17
2.3.2	Grid'5000	19
2.3.3	PlanetLab et OneLab	20
2.3.4	DSLlab	22
2.3.5	Conclusion	22
2.4	Émulateurs	22
2.4.1	Émulateurs de liens réseaux	22
2.4.2	MicroGrid	23
2.4.3	VINI	23
2.4.4	Modelnet	24
2.4.5	Wrekavoc	26
2.4.6	V-DS	26
2.4.7	DieCast	26
2.4.8	Autres émulateurs de topologies réseaux	27
2.5	Conclusion	28

2.1 Introduction

Dans ce chapitre, nous donnons un aperçu des différents outils disponibles et utilisés pour étudier les systèmes distribués. Nous présentons d'abord quelques simulateurs, qui sont la manière la plus classique d'étudier le comportement à grande échelle des applications distribuées. Puis nous présentons les plates-formes d'expérimentation, puis, pour finir, nous nous intéressons aux émulateurs, qui visent à permettre de s'abstraire d'une plate-forme spécifique, tout en facilitant la reproduction des résultats par d'autres chercheurs.

2.2 Simulateurs

Les systèmes distribués sont naturellement largement étudiés en utilisant la modélisation mathématique, suivie généralement de simulations. Nous présentons ici un éventail des simulateurs existants, sans toutefois chercher à être exhaustif : il existe des dizaines, voire des centaines de simulateurs pour les systèmes distribués.

2.2.1 Simulateurs de grille et de systèmes pair-à-pair

De nombreux simulateurs permettent de simuler une grille de calcul ou un système pair-à-pair. Mais ces simulateurs sont en général dédiés à la simulation d'un type d'application précis, voire à l'étude d'un aspect précis de ce type d'application.

Pour les grilles, on peut citer par exemple SimGrid [8], dont le développement s'effectue en France, ou GridSim [9], qui se focalisent sur l'ordonnancement des tâches et des communications. D'autres simulateurs, comme OptorSim [10], se focalisent sur les stratégies de répllication et de placement des données.

De même, il existe de nombreux simulateurs pour les systèmes pair-à-pair [11]. On peut citer par exemple P2Psim [12], PlanetSim [13], Overlay Weaver [14], et PeerSim [15], qui ciblent tous la simulation d'*overlays* structurés ou non. Tous proposent une interface différente, d'où la proposition d'une interface de programmation commune, comme par exemple dans Macedon [16].

Dans de nombreux travaux, un simulateur est développé spécifiquement pour simuler une application précise, comme dans [17] où BitTorrent est l'objet de la simulation, ou FreePastry [18]. Ces simulateurs peuvent parfois ensuite être généralisés à d'autres applications similaires.

2.2.2 Network Simulator - ns-2

NS2 [19, 20] est un simulateur *Open Source* à événements discrets destiné à la simulation de protocoles réseaux. Il permet la simulation du routage et de protocoles IP comme UDP ou TCP sur des réseaux câblés, sans-fil ou satellites. NS2 est très utilisé par la communauté scientifique.

NS2 permet de choisir le niveau d'abstraction désiré en définissant la granularité de la simulation, ce qui permet aussi de contrôler la durée de la simulation.

L'utilisation de NS2 se fait via l'écriture d'un script en TCL décrivant la topologie à simuler ainsi que le déroulement de l'expérience.

Network Simulator inclut également une interface d'émulation [21] permettant d'échanger du trafic réel entre un réseau réel et le simulateur.

Un remplaçant de NS2, NS3 [22], est en cours de développement, mais ses premières versions "stables" ne sont sorties qu'en 2008 : il est difficile d'évaluer pour l'instant s'il va être adopté par la communauté.

2.3 Plates-formes d'expérimentation

A l'opposé de la simulation, plusieurs plates-formes d'expérimentation permettent de réaliser des expériences sur les systèmes distribués. Chaque plate-forme se distingue par plusieurs aspects :

- **Interface utilisateur** : avec quels outils l'utilisateur accède-t-il à la plate-forme ? Quels sont les prérequis ?
- **Modèle de partage de ressources et d'isolation** : comment les ressources sont-elles partagées entre les utilisateurs ? Comment deux expériences concurrentes influent-elles l'une sur l'autre.
- **Caractéristiques topologiques de la plate-forme** : quels types de réseaux la plate-forme propose-t-elle ?
- **Taille de la plate-forme** : de combien de noeuds la plate-forme est-elle composée ? Un nombre trop faible de noeuds empêchera de réaliser des expériences à très grande échelle.

Dans la suite de cette section, nous présentons les principales plates-formes expérimentales en tentant de répondre à ces questions.

2.3.1 Emulab

Emulab [23] désigne à la fois un ensemble de logiciels utilisés pour construire une plate-forme d'expérimentation réseau, et les différentes installations de cette plate-forme. La principale installation d'Emulab est située à l'université de l'Utah, également à l'origine du projet, et contient environ 450 noeuds. Le code source et les informations nécessaires pour installer sa propre instance d'Emulab étant disponibles, il existe une vingtaine d'autres installations.

La définition et le contrôle d'une expérience avec Emulab se fait à l'aide d'une interface similaire à celle de Network Simulator (section 2.2.2) : l'utilisateur décrit son expérience avec un script TCL.

En plus d'un ensemble de noeuds utilisant de l'émulation réseau pour fournir une topologie réseau arbitraire, Emulab fournit également un accès à d'autres plates-formes expérimentales, comme :

- Un accès aux plates-formes RON et PlanetLab ;
- Des noeuds connectés à des réseaux sans-fil (802.11) ;
- Des réseaux de capteurs ;
- Des réseaux simulés à l'aide de Network Simulator, reliés aux réseaux réels avec la couche d'émulation de Network Simulator.

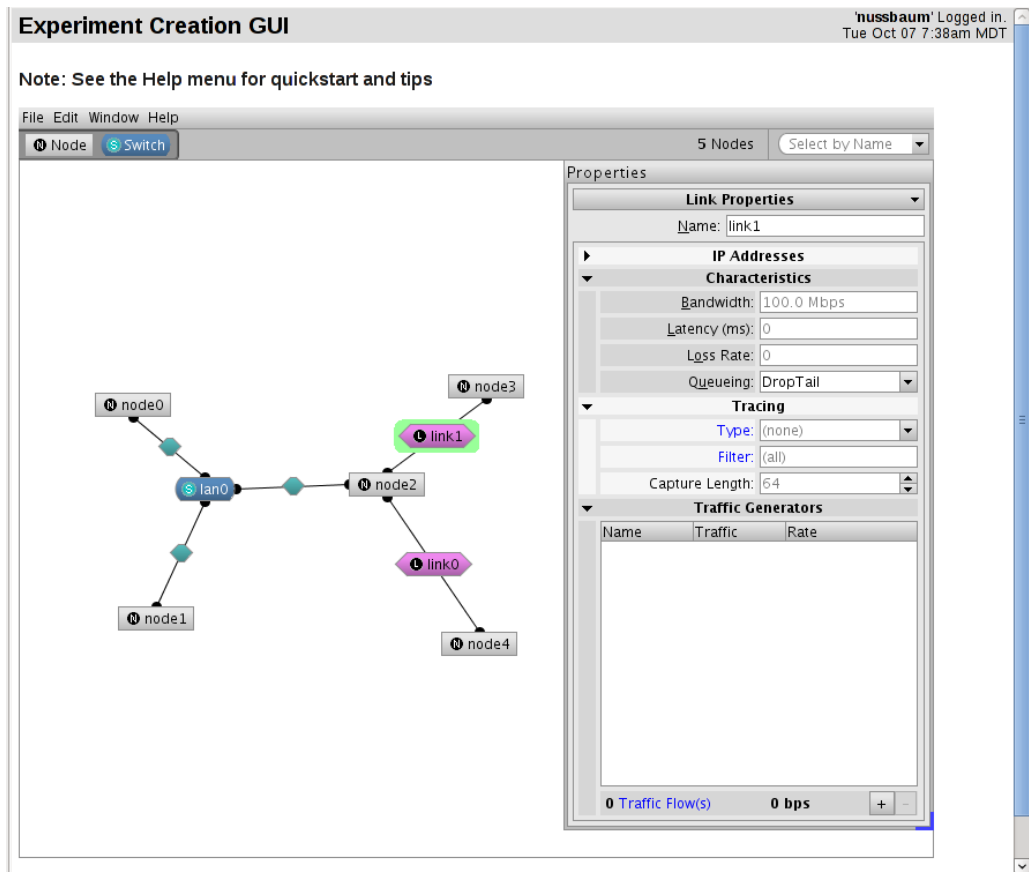


FIG. 2.1: Emulab : Interface graphique de définition de topologie.

Concernant la partie émulation, qui est la principale raison d'être d'Emulab, Emulab utilise Dummynet sur des noeuds FreeBSD, et TC/Netem sur des noeuds Linux, afin d'émuler la topologie définie par l'utilisateur à l'aide d'un script NS ou d'une interface graphique (figure 2.1).

Les expériences sont exécutées sur des noeuds dédiés, et peuvent être *swapped out*, c'est-à-dire arrêtées temporairement pour être reprises plus tard. Le réseau est lui partagé à l'aide de VLANs, mais cela ne garantit pas que les résultats des expériences ne sont pas perturbés par du trafic circulant dans les mêmes switches, si le *fond de panier* du switch est insuffisant. Un travail a également été fait sur la manière dont les topologies virtuelles sont implantées sur l'ensemble de ressources physiques [24].

Emulab est encore actuellement l'objet d'améliorations : Flexlab [25] consiste à améliorer les conditions réseaux émulées par Emulab en utilisant des sondes sur Planetlab pour capturer des conditions réalistes à reproduire. Et pour remédier au manque de ressources (machines physiques) dans Emulab, une infrastructure de virtualisation [26] basée sur les *FreeBSD jails* est proposée. Afin d'éviter un biais dans les résultats introduit par la virtualisation, un système de *feedback* est utilisé : tant que des artéfacts sont remarqués pendant l'expérience, on reproduit l'expérience avec une répartition différente des noeuds virtuels.

2.3.2 Grid'5000

Grid'5000 [27] est une plate-forme expérimentale pour l'étude des systèmes distribués. Elle rassemble plus de 2000 noeuds, répartis dans une quinzaine de clusters, eux-mêmes répartis dans 9 sites géographiques en France.

Les noeuds de Grid'5000 sont des noeuds de clusters de calcul classiques, éventuellement reliés par des réseaux rapides au sein d'un cluster. Au niveau de la grille, tous les noeuds sont directement joignables par le réseau inter-sites dédié (à 10 Gbps entre la plupart des sites).

Pendant une réservation, les ressources sont attribuées de manière dédiée à l'utilisateur, au niveau souhaité par l'utilisateur : noeud, processeur ou coeur. Le réseau est par contre partagé entre les expériences, ce qui peut induire des perturbations.

Une particularité importante de Grid'5000 est sa capacité à être reconfiguré pour les besoins d'une expérience : un utilisateur peut facilement, à l'aide de Ka-deploy [28], déployer son propre système d'exploitation sur les noeuds qu'il va utiliser. Cette propriété sera largement utilisée dans ces travaux de thèse, notamment pour déployer FreeBSD sur les noeuds (chapitre 6).

2.3.3 PlanetLab et OneLab

PlanetLab [29] est une plate-forme mondiale qui rassemble environ 800 machines localisés dans 472 sites différents. Son objectif est de servir de support à l'expérimentation de nouveaux protocoles et services réseaux. Les machines sont hébergées par les laboratoires et entreprises participant, et la virtualisation (basée sur Linux-VServer) est utilisée pour permettre aux utilisateurs d'avoir les droits *root* sur une machine virtuelle (*slice*). Les machines sont gérées de manière centralisée par l'université de Princeton. Toutefois, afin de minimiser l'importance des éléments centralisés dans PlanetLab, un système de fédération est prôné pour permettre à d'autres entités administratives de gérer un sous-ensemble de ressources (soit privées - accessibles uniquement par un sous-ensemble des utilisateurs, soit publiques - accessibles par tous les utilisateurs de PlanetLab). Ce système de fédération est utilisé par de nombreuses institutions, notamment au sein de EverLab [30] ou de OneLab [31].

OneLab [31] est un projet européen ayant pour objectif de gérer la partie européenne de PlanetLab (PlanetLab Europe), et d'ajouter des services de surveillance (*monitoring*) réseau. OneLab ajoute aussi des passerelles entre le réseau de PlanetLab (constitué majoritairement de machines connectées aux réseaux académiques) et des noeuds connectés à des réseaux moins classiques (WiMAX, UMTS), ainsi que des noeuds utilisant de l'émulation réseau, pour permettre des expériences exploitant ces connexions réseaux.

Disponibilité des machines sur PlanetLab

Nous avons utilisé PlanetLab dans le cadre de l'expérience "IDHAL", visant à vérifier la faisabilité d'une plate-forme de calcul globalisée, combinant des noeuds de Grid'5000, de clusters extérieurs à Grid'5000, de PlanetLab, de DSL-Lab (section 2.3.4), et des machines virtuelles hébergées chez des particuliers. Dans ce cadre, nous avons cherché à réduire les noeuds de PlanetLab à un ensemble réellement utilisable pour notre expérience. Nous nous sommes alors rendu compte que, sur les 891 noeuds proposés par PlanetLab, un nombre important de noeuds (figure 2.2) étaient indisponibles :

- 215 noeuds étaient listés comme non démarrés ;
- 11 noeuds étaient enregistrés avec une adresse DNS incorrecte ;
- sur 178 noeuds, nous n'avons pas pu nous connecter en SSH ;
- 45 noeuds n'offraient pas une connectivité réseau vers 2 serveurs différents localisés en France.

Finalement, seulement 442 noeuds, soit moins de la moitié de l'ensemble des noeuds, étaient réellement utilisables à un instant donné pour notre expérience. Ces résultats confirment ceux déjà obtenus en 2006 par l'équipe de PlanetLab [32].

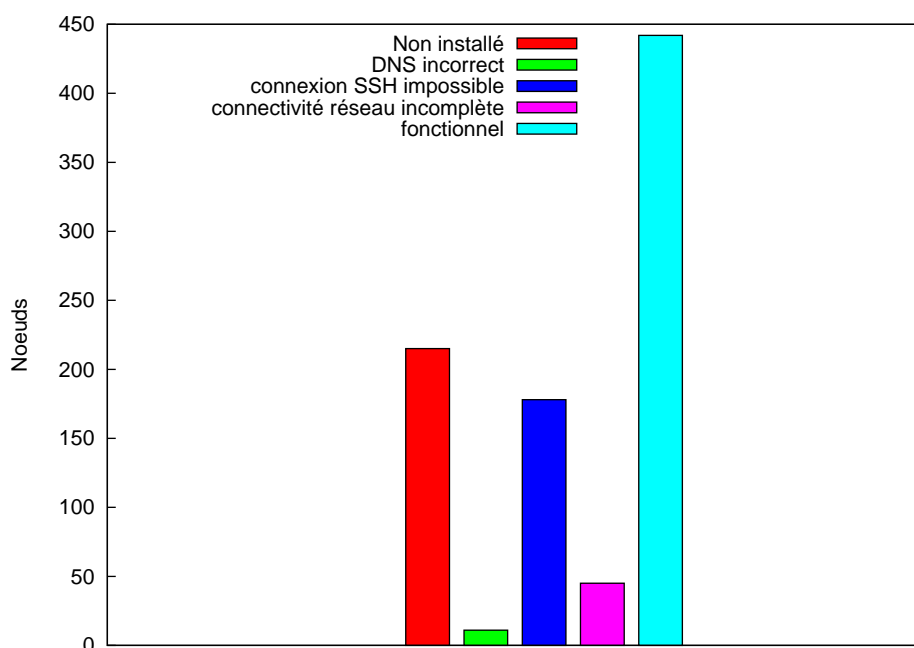


FIG. 2.2: Disponibilité des 891 noeuds de PlanetLab au 30 septembre 2008.

Malgré le choix d'un mode d'administration centralisé, on constate qu'il est difficile d'assurer une bonne disponibilité des ressources, et que cela réduit fortement la taille de la plate-forme.

Limites de PlanetLab

Une critique récurrente sur PlanetLab est le manque de diversité et de représentativité de ses noeuds [33, 34, 35] (la majorité des noeuds étant connectés à des réseaux académiques, eux-mêmes connectés entre eux à travers un réseau parfois désigné sous l'acronyme GREN - *Global Research and Educational Network*). Trois réponses à cette critique sont apportées :

- La nécessité de combiner des expériences sur plusieurs plates-formes, et de développer des outils et des bonnes pratiques pour évaluer les systèmes distribués [36]. Le rapprochement de PlanetLab et Emulab au sein de GENI¹ devrait faciliter ces démarches ;
- À l'intérieur même de PlanetLab, le développement de VINI (section 2.4.3), une infrastructure réseau virtuelle ;
- Le développement de plateformes satellites de PlanetLab, comme Satelli-

¹Global Environment for Network Innovations, un projet financé par la National Science Foundation visant à améliorer la recherche expérimentale en réseaux et systèmes distribués

teLab [37], ajoutant des noeuds connectés à des réseaux différents (pour SatelliteLab, des noeuds connectés à l'ADSL, au câble, etc.).

2.3.4 DSLLab

DSLlab [38] est un projet soutenu par l'Agence Nationale de la Recherche visant à contruire et exploiter une plate-forme d'expérience autour des systèmes distribués sur l'Internet haut-débit. La plate-forme est constituée de 40 noeuds hébergés chez des particuliers, utilisant leurs connexions à Internet ADSL ou câble.

Les hébergeurs n'étant pas dédommagés, les noeuds ont été choisis pour être silencieux et peu consommateurs en électricité. De plus, un mécanisme d'endormissement et de réveil, couplé avec le gestionnaire de ressource OAR [39], permet de n'allumer les noeuds que lorsqu'ils sont réellement utilisés.

Cette plate-forme permet de réaliser des expériences sur les *Desktop Grids*, ou à un plus bas niveau, sur les connexions réseaux des hébergeurs. Nous avons participé au développement de la plate-forme, et l'avons utilisée dans le cadre de l'expérience IDHAL (section 2.3.3).

2.3.5 Conclusion

Les différentes plates-formes expérimentales sont complémentaires, et permettent toutes de faire des expériences dans des environnements différents. Idéalement, il devrait être possible, facilement, de réaliser des expériences sur la même application, sur l'ensemble des plates-formes. En pratique, chaque plate-forme nécessite une expertise spécifique, et la plupart des travaux de recherche n'utilisent qu'une ou deux plates-formes. La convergence entre Emulab et Planet-Lab est clairement une voie intéressante, mais elle n'est pour l'instant pas étendue aux autres plates-formes.

2.4 Émulateurs

2.4.1 Émulateurs de liens réseaux

De nombreux outils permettent de modifier les caractéristiques d'un lien réseau, afin d'y ajouter de la latence, de limiter sa bande passante, ou d'injecter des fautes. Au niveau logiciel, on peut citer Dummynet [40, 41] (FreeBSD), NIST-Net [42] (Linux) et TC/Netem [43]. Des émulateurs matériels existent également, comme GtrcNET-1 [44] et les produits de la société Anué [45]. Toutes ces solutions sont présentées en détail au chapitre 3.

2.4.2 MicroGrid

MicroGrid [46] est un émulateur de grilles développé à l'université de Californie - San Diego, permettant d'émuler des applications Globus sans modification. Il combine émulation et virtualisation.

Concernant la virtualisation, le principal problème est de virtualiser l'identité de la machine. Les appels de bibliothèque comme `gethostbyname`, `bind`, `send`, `receive` sont donc interceptés. La simulation des ressources processeur se fait à l'aide d'un ordonnanceur *round-robin*, les tranches de temps étant calculées en fonction de la vitesse de simulation désirée.

Concernant l'émulation réseau, deux approches ont été explorées. Tout d'abord, NS2 (voir 2.2.2) a été essayé, mais il ne permettait pas de passer à l'échelle correctement. MaSSF [47] a alors été développé. MaSSF est un simulateur de réseaux à événements discrets. Pour permettre un bon passage à l'échelle, il utilise un moteur de simulation distribué tournant sur une grappe. A partir d'une topologie de réseau à simuler et d'un ensemble de noeuds, MaSSF partitionne le réseau virtuel en plusieurs blocs, assigne les blocs à des noeuds de la grappe, et simule en parallèle. Chaque noeud de la grappe utilise un simulateur à événements discrets et échange des événements avec les autres noeuds. Pour conserver une simulation réaliste, les noeuds se synchronisent périodiquement.

D'une manière générale, MicroGrid se distingue des autres solutions d'émulation par son souci de conserver un réalisme très important au détriment du coût de l'émulation. Il s'agit d'une approche à mi-chemin entre simulation et émulation.

MicroGrid n'est plus développé ni maintenu depuis 2004, et n'est pas utilisable sur un système Linux plus récent sans effectuer un important travail de portage.

2.4.3 VINI

VINI [48] (*Virtual Network Infrastructure*) est un système de topologies réseaux virtuelles au-dessus des réseaux *Abilene* et *National LambdaRail*, rassemblant actuellement 37 noeuds dans 22 sites différents. Les noeuds de VINI font également partie de PlanetLab, et il est donc très simple de "déplacer" une expérience de PlanetLab vers VINI.

L'objectif de VINI est de proposer un réseau d'expérimentation différent de PlanetLab, en permettant à l'utilisateur de définir sa propre topologie réseau et d'injecter des fautes.

2.4.4 Modelnet

Modelnet [49] a été initialement développé à l'université de Duke, mais le travail sur cet outil s'effectue maintenant à l'université de Californie, San Diego. Modelnet est un émulateur de topologies réseaux. Pour fonctionner, il sépare les noeuds en deux ensembles :

- les *edge nodes*, sur lesquels l'application à étudier est exécutée (éventuellement, en exécutant plusieurs instances de l'application par *edge nodes* pour économiser des ressources) ;
- les *core nodes*, qui exécutent l'émulateur réseau. Ils utilisent FreeBSD avec un module noyau spécifique.

Le point crucial avec Modelnet est la manière dont est construit le réseau de *core nodes*, afin qu'il émule correctement la topologie réseau fournie en entrée. Une première approche consiste à utiliser un *core node* par noeud de la topologie fournie en entrée. Mais cette approche est peu pratique, car elle nécessite un nombre très important de *core nodes*.

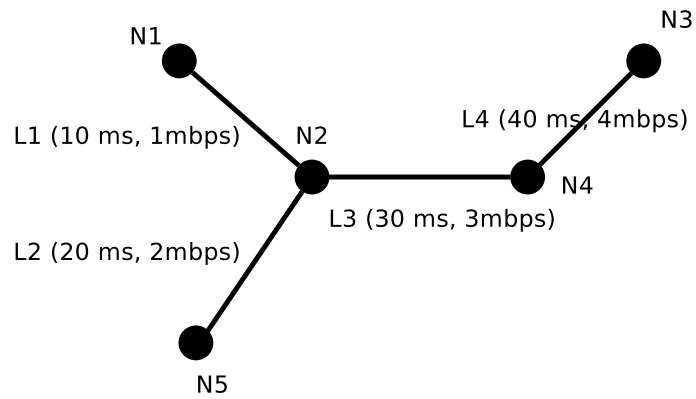
Modelnet propose donc une phase de *distillation*, qui a pour but de simplifier la topologie réseau en entrée pour limiter le nombre de noeuds nécessaires, en calculant directement la latence totale et le débit minimum entre deux noeuds de la topologie (figure 2.3). Cette approche, poussée à l'extrême, supprime tous les liens intérieurs de la topologie, mais supprime également les informations sur la congestion des liens intérieurs. Les auteurs justifient cette approche en expliquant que c'est un compromis acceptable, le coeur de l'Internet étant surprovisionné, et la bande passante étant principalement limitée par les noeuds situés en bordure.

Des améliorations ultérieures [50] visent à améliorer le passage à l'échelle de Modelnet et des *core nodes* :

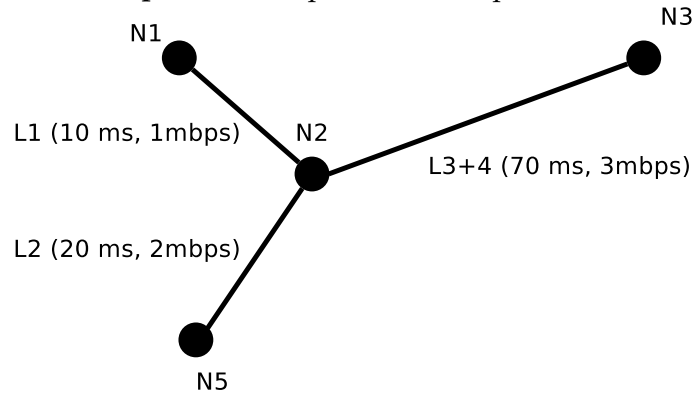
- Le partitionnement de la topologie pour regrouper des sous-graphes sur un seul *core node* sans perdre la possibilité d'émuler la congestion ;
- Pour éviter de saturer le réseau entre les *core nodes*, la conservation des données des paquets sur le premier *core node* (seules les informations réellement utiles pour l'émulation étant transférées aux autres *core nodes*), avec une solution basée sur [51] ;
- Le remplacement de la table de routage en $O(n^2)$ sur chaque *core nodes* par le stockage d'un arbre de recouvrement, et un calcul pour chaque paquet du *core node* auquel il faut envoyer le paquet [52].

D'autres émulateurs sont basés sur Modelnet. Mobinet [53] est un émulateur de réseaux sans-fil, ajoutant à Modelnet l'émulation de la couche physique de réseaux sans-fil, ainsi que la modification, pendant l'expérience, de la topologie, afin de prendre en compte la mobilité des noeuds.

Modelnet est disponible publiquement et gratuitement.



Etape 1 : on remplace L3 et L4 par L3+4



Etape 2 : on remplace tous les liens intérieurs par des liens directs entre *edge nodes*

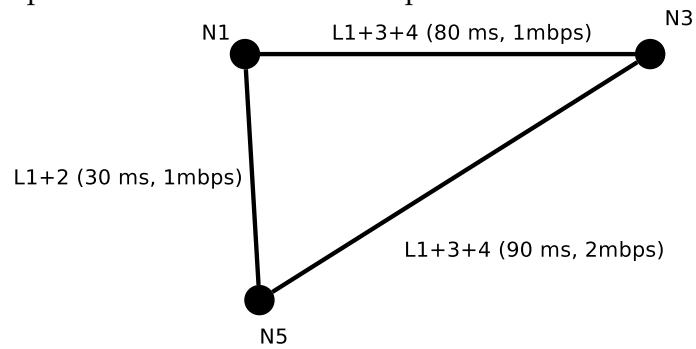


FIG. 2.3: Processus de distillation de Modelnet.

2.4.5 Wrekavoc

Wrekavoc [54] est un ensemble d'outils destinés à introduire de l'hétérogénéité dans un cluster, développé par Louis-Claude Canon et Emmanuel Jeannot, au LORIA. Wrekavoc permet de :

- dégrader les performances du CPU (soit en utilisant CPUFreq, soit en exécutant conjointement un processus utilisant beaucoup de temps processeur, soit en utilisant un processus-ordonnanceur qui va arrêter et redémarrer le processus étudié à l'aide de SIGSTOP et SIGCONT) ;
- ajouter de la latence et limiter la bande passante du réseau en utilisant Linux TC ;
- limiter l'utilisation mémoire en utilisant PAM.

2.4.6 V-DS

V-DS [55, 56] est un émulateur développé par Benjamin Quétier au cours de sa thèse au LRI. V-DS utilise Xen [57] pour virtualiser l'ensemble du système d'exploitation sur les noeuds virtuels (Benjamin Quétier avait précédemment comparé les différentes approches de virtualisation dans [58]). Pour l'émulation réseau, V-DS utilise un ensemble de noeuds sous FreeBSD, avec Dummynet [40]. V-DS ne permet pas d'émuler une topologie : il permet uniquement d'intercaler un noeud Dummynet entre chaque couple de noeuds, et d'y appliquer des paramètres de latence et de bande passante en émission et réception.

2.4.7 DieCast

DieCast [59] a été développé par le même groupe que Modelnet (*Systems and Networking, Computer Science and Engineering, University of California - San Diego*). DieCast permet l'étude des systèmes distribués en utilisant la dilatation temporelle [60]. Le principe est de modifier l'échelle du temps pour permettre :

- d'utiliser des machines virtuelles pour augmenter le nombre de noeuds, tout en masquant les effets de la virtualisation (*si on utilise 10 machines virtuelles par machine physique, et qu'on ralentit le temps 10 fois, on retrouve les performances d'une machine physique*) ;
- de se placer dans des conditions expérimentales impossibles à mettre en place avec une plateforme réelle, comme par exemple un réseau 100 Gbps.

DieCast fonctionne en utilisant des machines virtuelles Xen, et en modifiant la fréquence des interruptions horloge envoyées aux machines virtuelles. Ainsi, pour la machine virtuelle, le temps s'écoule plus lentement qu'il ne s'écoule réellement.

L'ordonnanceur de l'hyperviseur Xen a également été modifié, afin qu'il n'attribue pas plus d'un pourcentage spécifié de temps à chaque machine virtuelle

(*non-work conserving scheduler*). En effet, si on utilise 10 machines virtuelles par machine physique, et qu'on divise la fréquence des interruptions horloge par 10, on souhaiterait que les machines virtuelles aient la même performance (perçue par les applications) qu'une machine physique. Si une machine virtuelle est exécutée plus fréquemment que 10 % du temps par l'ordonnanceur de Xen, les performances seront supérieures à celles d'une machine physique.

Les autres composants de la plate-forme doivent également être ralentis : pour le réseau, de l'émulation réseau est utilisée (avec Dummynet [40], Modelnet [49] ou Linux TC/Netem [43]). Et pour le disque, Disksim [61] permet de simuler les performances d'un disque déterminé.

Si cette approche permet a priori des résultats sans biais liés à la virtualisation, elle a toutefois l'inconvénient de ralentir les expériences du facteur de virtualisation choisi : avec un rapport de virtualisation de 10, l'expérience prendra 10 fois plus de temps. De plus, même si l'idée est très séduisante, il faut noter qu'une machine virtuelle ralentie n'est pas tout à fait équivalente à une machine physique :

- Le processeur n'est pas ralenti ; les machines virtuelles sont seulement ordonnancées moins souvent. Pendant qu'une machine virtuelle dispose du processeur, elle calculera à vitesse normale.
- Certains composants ne peuvent pas être ralentis. Par exemple, une application utilisant très intensivement la mémoire pourra produire des résultats erronés.

DieCast n'est pour l'instant pas diffusé publiquement.

2.4.8 Autres émulateurs de topologies réseaux

EMPOWER [62] est un émulateur développé par Pei Zheng pendant sa thèse. Il n'est pas disponible publiquement. EMPOWER utilise un module noyau spécifique pour Linux chargé de l'émulation réseau. L'émulation de topologies se fait toujours au sein d'un seul noeud-émulateur, et ne permet donc pas un bon passage à l'échelle.

IMUNES [63] est un émulateur développé à l'université de Zagreb. Il virtualise la pile réseau du noyau FreeBSD pour permettre à plusieurs applications de disposer chacune de sa propre pile. Comme dans EMPOWER, les différents noeuds émulés s'exécutent sur la même machine physique, limitant le passage à l'échelle de l'outil.

V-eM [64] est un émulateur réseau développé en Grèce (ICS-FORTH). Il utilise Xen pour héberger les noeuds virtuels, et NISTNet pour réaliser l'émulation réseau. Il ne permet pas de créer une topologie réseau.

eWAN [65] est un émulateur développé par l'ENS Lyon et le National Institute of Advanced Industrial Science and Technology. Il utilise Linux TC et netem,

avec le module PSpacer [66] pour permettre un espacement correct des paquets envoyés, indépendant de l'horloge système. eWAN émule une topologie réseau en transformant le nuage réseau en réseau en étoile, donc le coeur est géré par un ensemble de noeuds.

2.5 Conclusion

Les solutions que nous avons présenté répondent à un large éventail de problèmes, mais certains besoins restent sans réponse. Notamment, pour les plates-formes d'émulation, on peut insister sur les besoins suivants :

Passage à l'échelle : la plupart des solutions présentées ne permettent pas de dépasser quelques centaines de noeuds. Or le comportement d'un système peut être modifié de manière très importante lorsqu'il sera déployé sur plusieurs milliers ou dizaines de milliers de noeuds.

Topologies de réseaux proposées : La plupart des émulateurs utilisent des topologies décrivant de manière précise le coeur du réseau. Or il apparaît que la plupart des problèmes se situent sur les liens les plus extérieurs (dans le cadre des systèmes pair-à-pair, par exemple, sur les liens reliant l'utilisateur à son fournisseur d'accès). S'il reste intéressant, dans certains cas, d'avoir une modélisation précise du coeur du réseau, cela rajoute de la complexité, et n'est pas forcément souhaitable dans tous les cas.

Validation : la validation de la plupart des solutions d'émulation présentées n'a pas été très poussée. Certaines briques de bases des plates-formes (systèmes de virtualisation, émulateurs réseaux) ont originellement été développées avec d'autres objectifs, et n'ont pas été validés dans ce contexte là. Il est possible que cela provoque des biais non détectés dans les résultats obtenus.

Dans la suite du manuscrit, nous nous attachons à proposer une réponse à ces besoins en abordant cette problématique avec une démarche incrémentale. Nous commençons par valider les briques de base que nous allons utiliser (émulateurs de liens réseaux) et par illustrer leur utilisation (chapitres 3, 4 et 5), avant de s'intéresser à une plate-forme d'émulation complète (chapitres 6 et 7).

ÉTUDE COMPARATIVE DES ÉMULATEURS DE LIENS RÉSEAUX 3

3.1	Introduction	29
3.2	Émulateurs réseaux	30
3.3	Fonctionnalités	31
3.4	Évaluation des performances	31
3.4.1	Configuration expérimentale	31
3.4.2	Source de temps, précision et émulation de latence	33
3.4.3	Limitation de la bande passante	37
3.5	Différences fonctionnelles	41
3.5.1	Interfaces utilisateur	41
3.5.2	Point d'interception des paquets avec TC	41
3.6	Conclusion	42

3.1 Introduction

Le développement et l'évaluation d'applications distribuées nécessitent de prendre en compte la variété des conditions environnementales qui peuvent être rencontrées. Ainsi, la performance des communications d'une application est très fortement influencée par les caractéristiques du réseau utilisé, et il est donc nécessaire de vérifier le bon fonctionnement d'une application dans une grande variété de conditions réseaux.

Malheureusement, il est en général très difficile de modifier la configuration du réseau des plates-formes expérimentales classiques, puisque cela nécessite en général de manipuler des câbles réseaux, des switches ou des routeurs, domaines réservés aux administrateurs réseaux. Une autre solution est d'utiliser plusieurs plates-formes expérimentales différentes, comme Grid'5000, Emulab ou Planet-Lab. Mais obtenir l'accès à ces plates-formes n'est pas toujours évident, et leurs spécificités font qu'il est difficile de reproduire une expérience identique sur des plates-formes différentes.

Les méthodes basées sur l'émulation (qui consiste à modifier l'environnement expérimental de manière artificielle) sont une alternative intéressante : elles permettent de reproduire des conditions expérimentales différentes d'une manière simple, sans modifier le reste de l'environnement.

Il existe de nombreux outils permettant d'émuler des caractéristiques réseaux différentes. Dans ce chapitre, nous proposons une étude comparative d'un sous-ensemble des émulateurs réseaux, les émulateurs de liens réseaux. Ces émulateurs ont déjà été utilisés pour réaliser de nombreuses expériences et pour construire des plates-formes comme Emulab [23], mais n'ont jamais été validés ou comparés.

3.2 Émulateurs réseaux

L'idée de modifier de manière logicielle les caractéristiques d'un réseau n'est pas récente : en 1995, un émulateur de réseaux longue distance a été utilisé pour évaluer la variante TCP de Vegas [67].

Parmi la multitude de solutions développées depuis, on peut distinguer 2 catégories :

Les émulateurs de topologies réseaux visent à émuler un nuage réseau complet : la description d'une topologie réseau est fournie à l'émulateur, qui va typiquement utiliser une grappe de calcul pour émuler l'ensemble du réseau. On peut citer par exemple MicroGrid [46], Modelnet [49], Netbed et Emulab [23], EMPOWER [62], IMUNES [63], V-em [64] et eWAN [65]. Toutefois, ces émulateurs sont en général complexes à maîtriser, et leur utilisation passe par une longue phase de configuration : leur diffusion en dehors de leur laboratoire d'origine est en général limitée.

Les émulateurs de lien réseau sont plus simples : ils retardent ou suppriment (*drop*) les paquets entrant ou sortant d'une interface réseau particulière, en fonction des paramètres souhaités par l'utilisateur (bande passante, latence, taux de pertes). Delayline [68] est une bibliothèque en mode utilisateur qui fournit ces fonctionnalités. Ohio Network Emulator [69] fonctionnait sur Solaris, mais n'est plus maintenu. Dummynet [40, 41] fonctionne sur FreeBSD, et est intégré dans le pare-feu de FreeBSD (IPFW). NISTNet [42] a été initialement développé pour Linux 2.4, et a été récemment porté pour Linux 2.6, mais son développement semble arrêté. Linux 2.6 fournit aussi Netem [43], une solution d'émulation réseau intégrée dans le sous-système Linux Traffic Control (TC). Solaris propose enfin son propre système d'émulation réseau, appelé hxbt [70]. Enfin, des solutions matérielles existent, comme GtrcNET-1 [44] (à base de FPGA), ou les produits de la société Anué [45].

Dans cette étude, nous nous limitons à Dummynet, NISTNet et TC/Netem, pour plusieurs raisons :

- tout d'abord, ces trois solutions sont de qualité suffisante pour être utilisées sans trop de difficultés pour tester des applications, et sont réellement conçues dans l'objectif d'être diffusées et utilisées ;

- ensuite, ces trois solutions sont disponibles gratuitement, et fonctionnent avec les systèmes d’exploitation couramment utilisés sur les grappes de calcul ;
- enfin, ce sont les solutions les plus couramment utilisées actuellement par la communauté. Certains émulateurs réseaux sont également utilisés comme briques de bases dans des émulateurs de topologie, comme Dummynet (dans Emulab et V-DS), NISTNet (dans V-eM) et Linux TC/Netem (dans Emulab, Wrekavoc et eWAN).

3.3 Fonctionnalités

Le tableau 3.1 présente les fonctionnalités de Dummynet, NISTNet et TC/Netem. NISTNet et Netem ont des fonctionnalités très proches, car ils partagent du code, mais leur architecture est complètement différente : alors que NISTNet est construit comme un module noyau, et s’appuie sur l’horloge temps réel de la machine (*RTC*), Netem est intégré dans le sous-système Linux Traffic Control (habituellement utilisé pour paramétrer de la qualité de service à l’intérieur de réseaux). De plus, Netem est distribué avec Linux, tandis que NISTNet est distribué séparément. NISTNet n’est actuellement disponible que pour les versions du noyau antérieures au noyau 2.6.14 à cause d’un problème dans la méthode utilisée par NISTNet pour intercepter les paquets. Il est toutefois relativement simple de modifier NISTNet afin de le faire fonctionner sur des noyaux plus récents. Nous l’avons ainsi modifié pour l’utiliser sur un noyau 2.6.26¹.

Dummynet a été développé indépendamment de NISTNet et Netem, et est intégré dans FreeBSD depuis FreeBSD 4. Son principal avantage comparé à ces derniers est qu’il permet d’intercepter à la fois les paquets entrants et sortants. Toutefois, il propose moins de fonctionnalités que ses alternatives.

3.4 Évaluation des performances

3.4.1 Configuration expérimentale

Les expériences qui suivent utilisent toutes la configuration système et réseau décrite dans la figure 3.1 : 3 nœuds (Bi-Opteron 2.0 Ghz avec 2 Go de RAM) du cluster GridExplorer sont utilisés. Le nœud *routeur* est configuré pour router les paquets entre *Nœud 1* et *Nœud 2*. Les nœuds 1 et 2 utilisent Debian avec un noyau Linux 2.6.26, tandis que le routeur utilise Linux 2.6.22 ou 2.6.26 (pour TC/Netem), Linux 2.6.26 (pour NISTNet), ou FreeBSD 7.0 (pour Dummynet). Les interfaces réseaux sont des cartes Broadcom BCM5780 bi-ports, intégrées aux cartes mères des nœuds. Sans configurer d’émulation réseau sur le routeur, nous

¹Notre patch est disponible sur <http://www-id.imag.fr/~nussbaum/#nistnet>

	Dummynet	NISTNet	TC/Netem
Disponibilité	Inclus dans FreeBSD	Disponible pour Linux 2.4 et 2.6 (< 2.6.14)	Inclus dans Linux 2.6
Résolution temporelle	horloge système (HZ)	Horloge temps réel (RTC)	Horloge système (HZ) ou <i>timer</i> haute résolution
Sens d'interception des paquets	Entrée et sortie	Entrée seulement	Sortie seulement
Latence	Oui, valeur constante	Oui, avec gigue corrélée suivant une distribution uniforme, normale, de Pareto, ou normal+Pareto	Oui, avec gigue corrélée suivant une distribution uniforme, normale, de Pareto, ou normal+Pareto
Limitation de bande passante	Oui, le délai à rajouter aux paquets est calculé lors de leur entrée dans Dummynet	Oui, le délai à rajouter aux paquets est calculé lors de leur entrée dans NISTNet	Oui, pas directement dans Netem, mais en utilisant le Token Bucket Filter de TC
Perte de paquets	Oui, mais sans corrélation	Oui, éventuellement corrélées	Oui, éventuellement corrélées
Réordonnement de paquets	Non	Oui, éventuellement corrélé	Oui, éventuellement corrélé
Duplication de paquets	Non	Oui, éventuellement corrélé	Oui, éventuellement corrélé
Corruption de paquets	Non	Oui, éventuellement corrélé	Oui, éventuellement corrélé

TAB. 3.1: Fonctionnalités de Dummynet, NISTNet, and TC/Netem

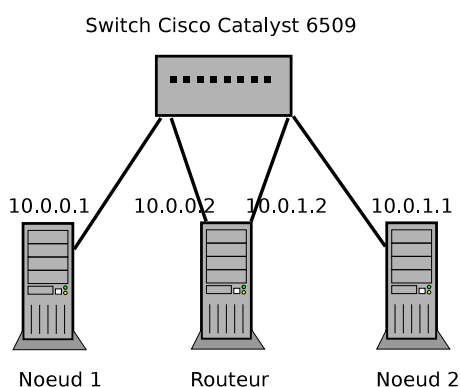


FIG. 3.1: Configuration expérimentale

avons mesuré un débit maximum de 943 Mbps et un *Round Trip Time* (RTT) d'environ $180 \mu s$ entre les nœuds 1 et 2 (en traversant le routeur).

3.4.2 Source de temps, précision et émulation de latence

*The entire focus of the industry is on bandwidth,
but the true killer is latency.*

Prof. M. Satyanarayanan
Keynote à ACM Mobicom '96

La précision de l'émulation dépend fortement de la source de temps utilisée par l'émulateur. NISTNet utilise directement l'horloge temps réel (*Real Time Clock*) configurée à 8192 Hz, tandis que Dummynet et Netem utilisent la même source d'interruptions d'horloge que le reste du noyau. Pour FreeBSD, la fréquence de ces interruptions est configurée dans la variable `HZ`, dont la valeur par défaut est de 100 Hz.

La situation dans Linux est plus complexe. Dans les versions anciennes du noyau (jusqu'au noyau 2.6.22 sur `i386`, 2.6.24 sur `x86_64`), Netem utilisait les interruptions d'horloge, comme Dummynet sur FreeBSD. Mais en plus des interruptions d'horloge classiques, la file d'attente de Netem est également examinée à chaque fois qu'un paquet entre dans Netem, ce qui peut, si le trafic est important, masquer les imprécisions des interruptions d'horloge.

Dans les versions ultérieures du noyau, Netem utilise un nouveau sous-système (*High Resolution Timers* [71]) permettant d'obtenir une bien meilleure précision.

Nous mesurons ici l'influence de ces différentes solutions sur la précision de l'émulation de la latence.

Les implémentations classiques de *ping*, utilisant `gettimeofday()`, ne fournissent pas une précision de mesure suffisante pour ces expériences. Nous avons

donc modifié une version de *ping* pour utiliser le *Time-Stamp Counter* du processeur (instruction assembleur `RDTSC`), afin d'obtenir une fréquence de mesure très élevée (10-20 KHz) et une précision de l'ordre de la microseconde. Nous avons mesuré l'évolution de la latence entre les nœuds 1 et 2 lorsque les émulateurs étaient configurés pour retarder les paquets allant du nœud 1 vers le nœud 2 pendant 10 ms.

Nous avons évalué les configurations suivantes pour le routeur :

- Linux 2.6.22 sur `x86_64`, utilisant donc les interruptions d'horloge, avec une fréquence de 100 Hz, 250 Hz (la valeur par défaut) et 1000 Hz ;
- Linux 2.6.26, utilisant les *High Resolution Timers*. Par acquis de conscience, nous avons vérifié que modifier la fréquence des interruptions d'horloge (100 Hz, 250 Hz ou 1000 Hz) ne changeait pas les résultats ;
- Linux 2.6.26 avec NISTNet ;
- FreeBSD 7.0, avec une fréquence de 100 Hz, 1 kHz et 10 kHz. Pour certaines expériences, nous avons aussi comparé les résultats obtenus avec FreeBSD 6.1.

Les figures 3.2 et 3.3 présentent les résultats pour l'ensemble de ces configurations. Les configurations ont été séparées en trois groupes, fournissant des résultats de qualité similaire, afin de faciliter les comparaisons. Pour chaque groupe, le graphique de la figure 3.2 montre l'évolution de la latence au cours du temps, mesurée à l'aide de *pings* envoyés à une fréquence très importante, tandis que celui de la figure 3.3 donne la fonction de répartition de la latence, mesurée avec des *pings* envoyés avec un intervalle aléatoire.

Avec Linux 2.6.22 et FreeBSD 7.0, on constate l'influence de la fréquence de l'horloge : plus la fréquence de l'horloge est élevée, plus l'émulation est précise. Avec une fréquence d'horloge faible (100 Hz ou 250 Hz), on assiste à des variations de latence très importantes. Par exemple, avec Linux 2.6.22 cadencé à 100 Hz, la latence varie entre 13 ms et 30 ms lorsque l'utilisateur demande une émulation de 10 ms de latence.

Avec FreeBSD 7.0, on constate aussi que la précision ne change pas lorsqu'on passe la fréquence de l'horloge de 1 kHz à 10 kHz. Avec FreeBSD 6.1 (voir figure 3.2, groupe 3), cela n'est pas le cas : une fréquence de 10 kHz fournit une émulation 10 fois plus précise qu'une horloge à 1 kHz.

On remarque aussi qu'à cause de différences algorithmiques dans la manière dont est émulée la latence, la latence émulée n'est jamais inférieure à celle configurée avec Linux, alors que la latence émulée est rarement supérieure à celle émulée avec FreeBSD : avec Linux, on définit une borne inférieure, tandis qu'avec FreeBSD on définit quasiment une borne supérieure.

Enfin, 3 solutions fournissent de bonnes performances (la différence entre la latence souhaitée et celle mesurée s'expliquant en grande partie par la latence physique du réseau) :

- NISTNet, car il n'utilise pas les mêmes interruptions d'horloge que le reste

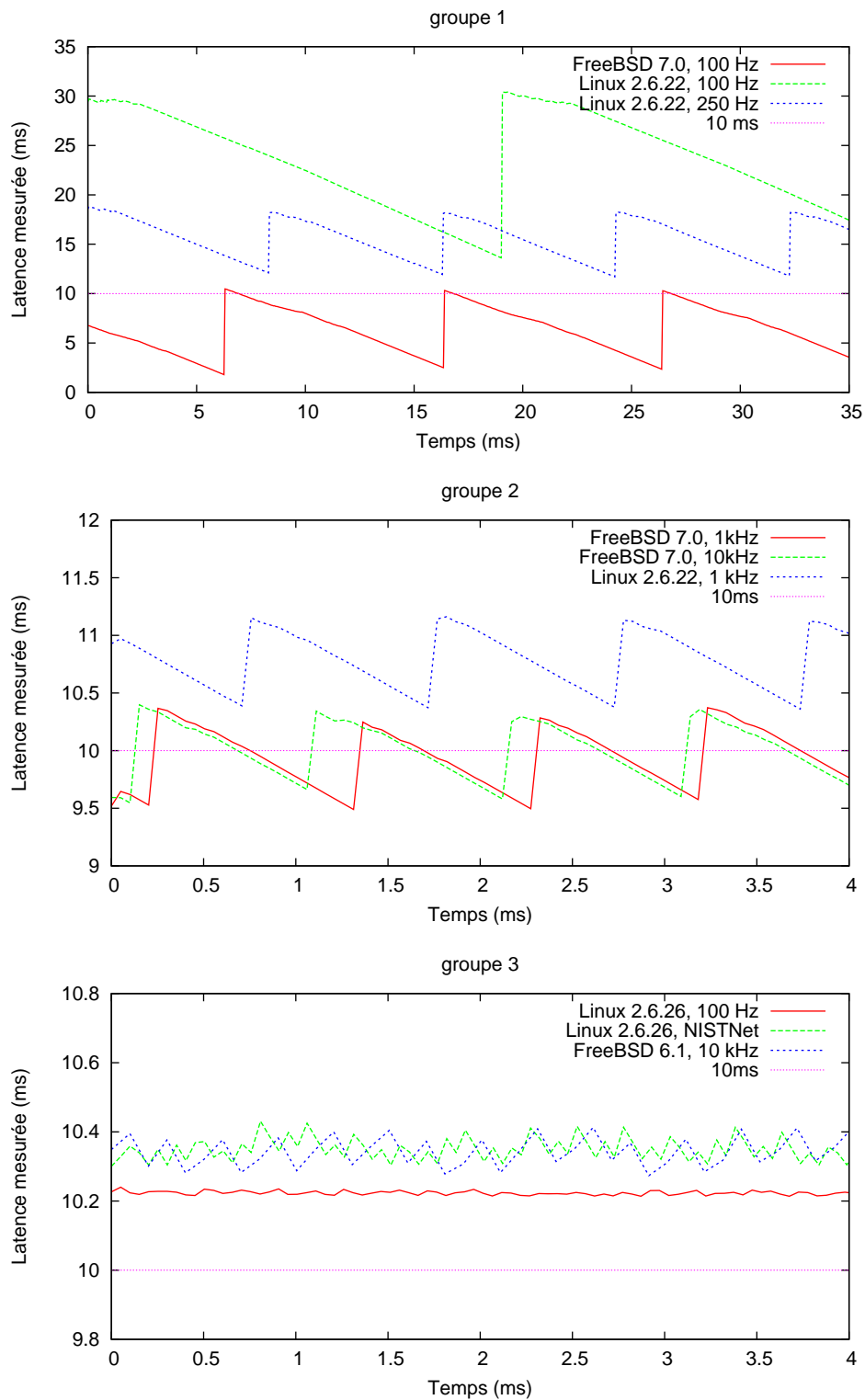


FIG. 3.2: Émulation de latence : évolution au cours du temps de la latence appliquée au paquets.

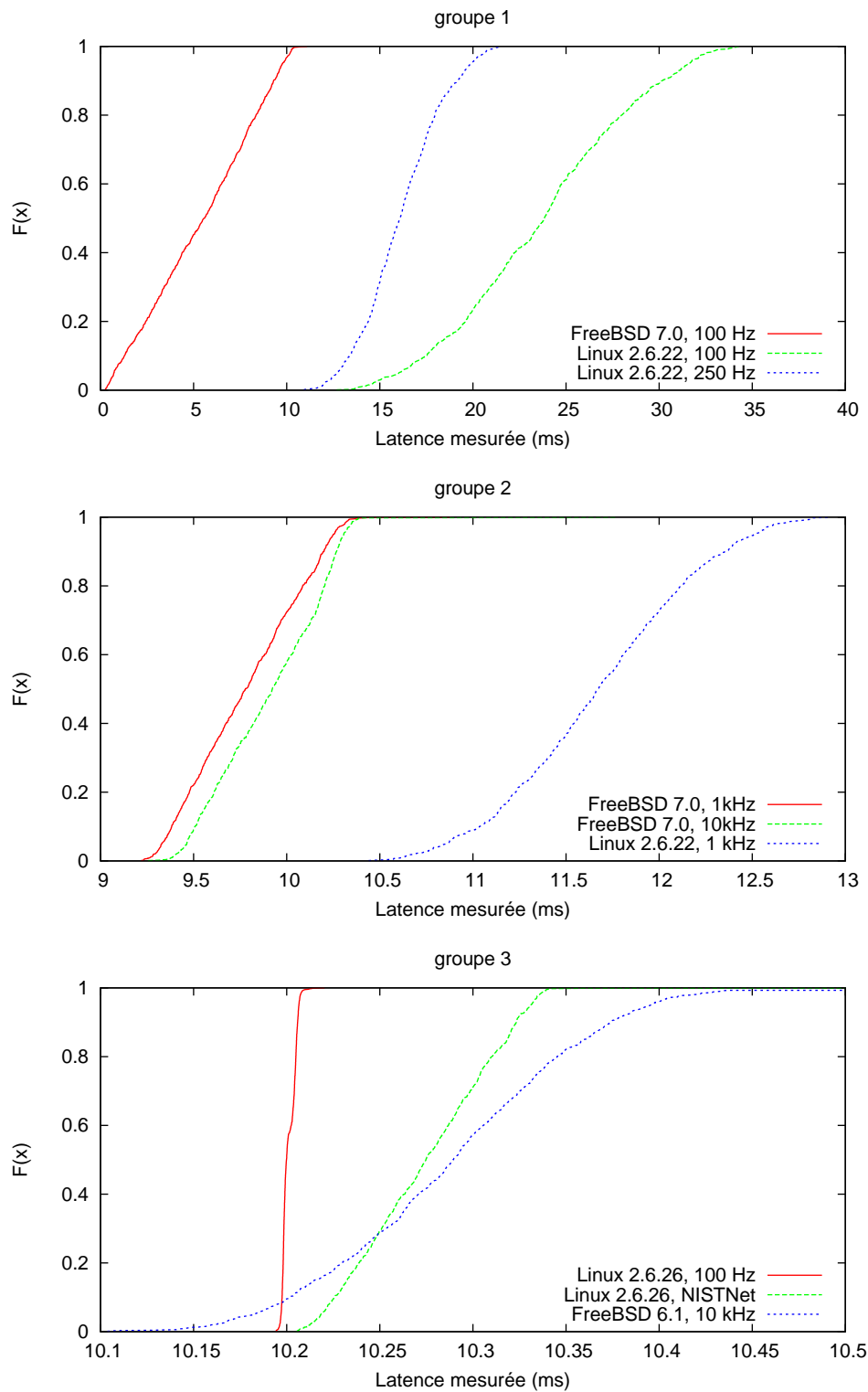


FIG. 3.3: Émulation de latence : fonction de répartition des latences émuloées, pour des paquets envoyés avec un intervalle aléatoire.

- du système, mais une horloge différente cadencée à 8192 Hz ;
- FreeBSD 6.1 cadencé à 10 kHz ;
- Linux, lorsque les *High-Resolution Timers* sont utilisés (ici avec Linux 2.6.26).

Toutefois, l’augmentation du nombre d’interruptions avec NISTNet et FreeBSD 6.1 va provoquer un ralentissement général du système (puisque’il passera plus de temps à traiter ces interruptions, provoquant à chaque fois des changements de contexte et des défauts de cache).

Pour mesurer ce surcoût, nous avons mesuré le temps d’exécution d’un programme utilisant très intensivement le processeur sur FreeBSD 6.1, en faisant varier la fréquence de l’horloge de 100 Hz à 10 kHz (tableau 3.2). On constate qu’à 10 kHz, l’exécution de notre programme de test est 3.3% plus lente, ce qui pourrait causer un problème pour certaines expériences.

Fréquence d’horloge	Temps d’exécution	Surcoût
100 Hz	81.5 s	-
1000 Hz	81.7 s	0.2%
10000 Hz	84.2 s	3.3%

TAB. 3.2: Temps moyen d’exécution d’un processus utilisant le CPU sur FreeBSD, en utilisant différentes fréquences d’horloge

Il est important de remarquer que NISTNet souffre du même problème : une fois le module noyau de NISTNet chargé, l’exécution du même programme a pris 86.8 s (surcoût de 6.3%).

Les *High Resolution Timers* n’ont pas ces désavantages : lorsqu’ils ne sont pas utilisés, ils ne ralentissent pas le reste du système. Par contre, lorsqu’ils sont utilisés, ils peuvent provoquer plus d’interruptions : puisqu’ils sont plus précis, les paquets ne seront pas envoyés en groupe, mais séparément, provoquant une interruption pour chaque paquet.

3.4.3 Limitation de la bande passante

La limitation de bande passante est l’autre aspect important de l’émulation réseau. Tandis que NISTNet et Dummysnet calculent simplement le délai à rajouter à un paquet donné à l’aide de la bande passante souhaitée, TC utilise un seau à jetons (*Token-Bucket*) pour réguler le trafic.

Dans cette expérience, nous comparons le débit souhaité avec celui mesuré en utilisant *iperf*. Utiliser *iperf* cause un léger biais dans la mesure, car *iperf* mesure la bande passante disponible en utilisant un flux TCP, tandis que les paramètres de l’émulation affectent la bande passante disponible pour les paquets IP. L’expérience a été réalisée sur un réseau Ethernet, et le MTU de l’interface était donc défini à 1500 octets. Nous avons donc corrigé la bande passante mesurée de

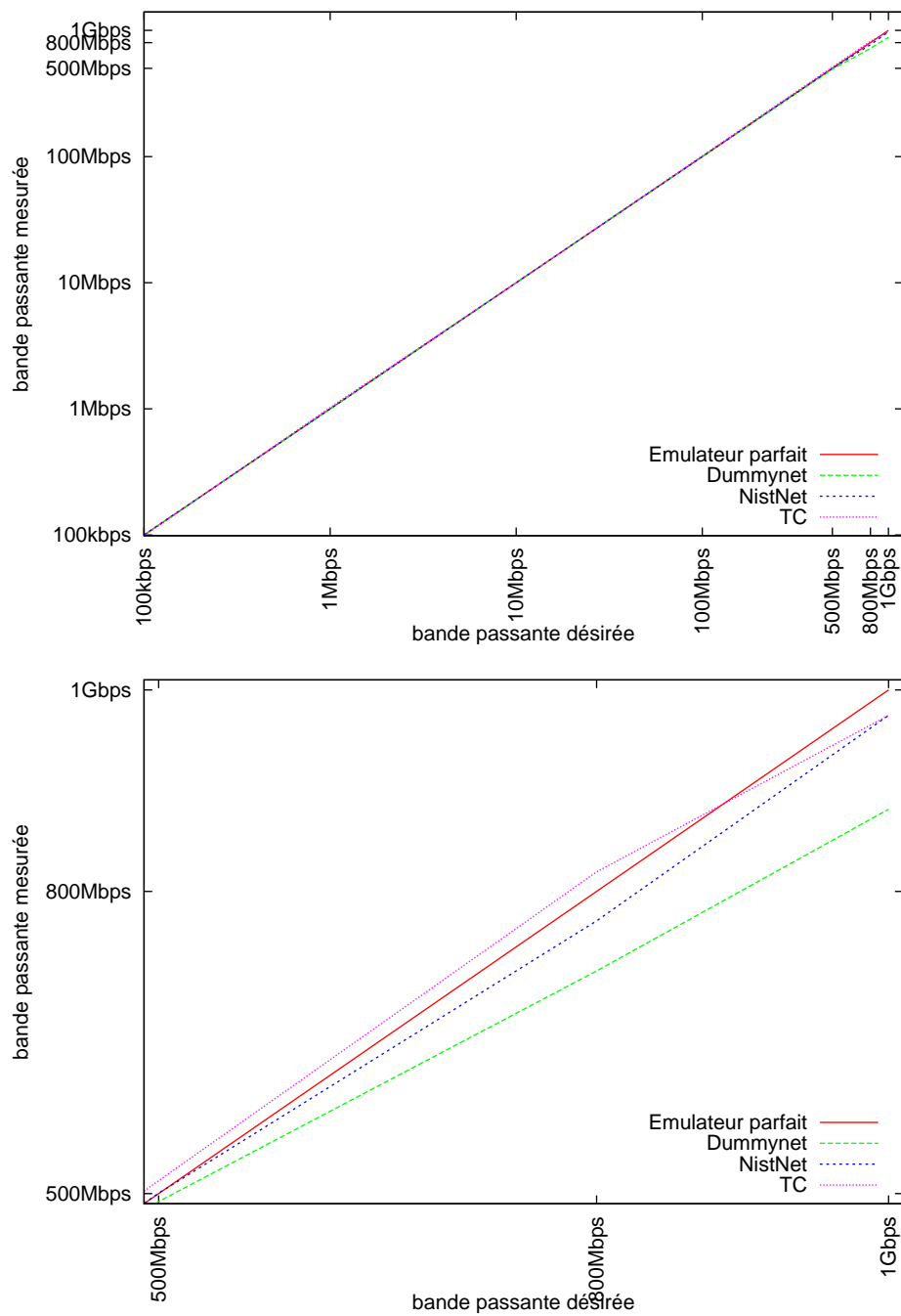


FIG. 3.4: Bande passante mesurée lorsque la bande passante paramétrée dans l'émulateur varie entre 100 kbps et 1 Gbps (échelle logarithmique), puis entre 500 Mbps et 1 Gbps

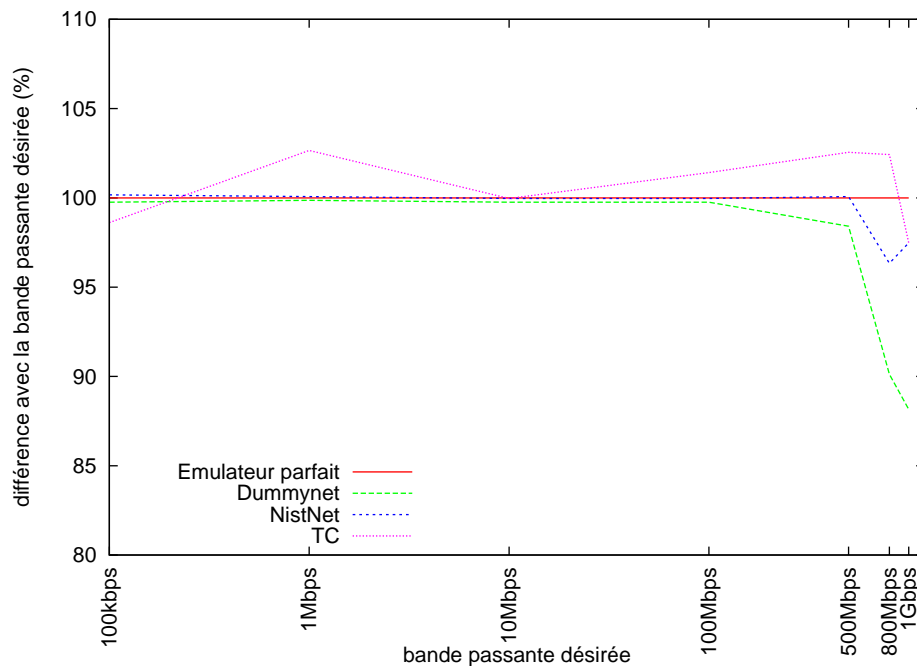


FIG. 3.5: Différence entre la bande passante paramétrée dans l'émulateur et celle mesurée

3.6% pour inclure les en-têtes IP et TCP (52 octets).

Les figures 3.4 et 3.5 comparent la bande passante mesurée en utilisant Dumynet (avec une fréquence d'horloge de 10 kHz), NISTNet, et TC (sous Linux 2.6.26). Les différences entre les bandes passantes désirées et mesurées sont faibles, mais on peut remarquer que lorsque la bande passante désirée est importante, Dumynet ne permet pas d'atteindre la bande passante souhaitée.

La limitation de bande passante souffre aussi du problème décrit dans la section 3.4.2. La fréquence des interruptions provoque un trafic en rafale : les paquets ne sont pas transmis de manière continue, mais uniquement lors d'une interruption. Lorsque le trafic est important, cela provoque l'envoi de plusieurs paquets lors d'une même interruption, ce qui peut être gênant lors de certaines expériences, selon l'application étudiée, par exemple en modifiant le comportement de la gestion de la congestion sur d'autres équipements. Ce problème n'est pas présent avec les *High Resolution Timers*.

Dimensionnement de la file d'attente

Un autre problème lié à la fréquence des interruptions est le dimensionnement de la file d'attente de l'émulateur. Ainsi, fin de pouvoir atteindre le débit souhaité, la taille de la file d'attente doit être au moins de :

$$queue_size \geq emulated_bandwidth * interrupt_frequency$$

Mais même avec une file d'attente définie ainsi, l'émulateur ne permettra pas de recevoir un nombre important de paquets, puis de les écouter petit à petit, lors de plusieurs interruptions. Il est donc important de surdimensionner la file d'attente, sans toutefois atteindre des limites irréalistes. Avec le *Token Bucket Filter* de TC, la taille du seau à jetons peut être configurée en spécifiant le temps maximal (latence) que les paquets sont autorisés à passer dans le seau. La taille de la queue est alors :

$$queue_size = emulated_bandwidth * latency$$

Ce problème est particulièrement important avec FreeBSD, car même la taille maximale de la file d'attente ne permet pas d'émuler des débits importants avec la fréquence d'horloge par défaut (100 Hz). Ainsi, la taille maximale de la file d'attente est de 100 paquets, limitant le débit émulé à 120 Mbit/s dans le cas le plus favorable (où tous les paquets de la file sont des paquets de 1500 octets). Ce problème disparaît en changeant la fréquence des interruptions d'horloge.

Paramétrage du seau à jetons de TC

Le seau à jetons utilisé par TC pour limiter la bande passante est aussi une source d'écarts entre le débit configuré et celui mesuré : comme il a été développé pour fournir un mécanisme de gestion de Qualité de Service (QoS), il utilise un algorithme complexe décrit dans la page de manuel de *tc-tbf* (`man tc-tbf`). Cet algorithme permet de transmettre une rafale de paquets sans les réguler. L'idée est que dans le cas où le réseau est silencieux, il est préférable de ne pas réguler le trafic de communications intenses mais courtes. Cela est un problème pour l'émulation réseau, où tous les paquets doivent être retardés de la même manière. Il existe une solution en utilisant le paramètre `peakrate` décrit dans la page de manuel :

If it is not acceptable to burst out packets at maximum speed, a peakrate can be configured to limit the speed at which the bucket empties. This peakrate is implemented as a second TBF with a very small bucket, so that it does not burst.

Cependant, ce second seau à jetons rajoute de la complexité dans la configuration, notamment lorsqu'on la compare avec la configuration de Dummynet et NISTNet. Il est important de valider les paramètres afin de vérifier qu'ils émulent bien les conditions souhaitées avant de démarrer des expériences.

3.5 Différences fonctionnelles

3.5.1 Interfaces utilisateur

Si les performances et la précision des émulateurs est importante, leur souplesse et leur facilité d'utilisation est un autre aspect à prendre en compte.

Dumynet et NISTNet se configurent en utilisant des règles, comme un pare-feu, ce qui les rend facile à utiliser pour les utilisateurs familiers avec la configuration de pare-feus. Cependant, ils ne permettent pas de mettre en place des hiérarchies complexes de règles, ce qui pourrait être un problème dans le cas où l'utilisateur tente d'émuler une topologie réseau complexe.

TC utilise une approche différente : sa configuration se fait en utilisant un ensemble hiérarchique de *qdiscs* (*queueing discipline*) et de classes. Cela fournit plus de possibilités, mais est aussi plus difficile à appréhender, d'autant plus que la documentation n'est pas toujours parfaite.

3.5.2 Point d'interception des paquets avec TC

Un avantage important de Dumynet sur NISTNet et TC est qu'il peut capturer à la fois les paquets entrants et les paquets sortants. TC permet seulement de capturer les paquets sortants, ce qui est logique puisqu'il a été conçu comme un modèleur (*shaper*) de trafic, pas comme un émulateur. NISTNet permet seulement de capturer les paquets entrants.

Il est souvent nécessaire de réaliser l'émulation sur les paquets entrants, par exemple si l'utilisateur veut réaliser l'émulation sur le système où l'application à étudier est exécutée. Une solution existe avec le pseudo-périphérique réseau (logiciel) *ifb* (*Intermediate Functional Block*) : il est possible de rediriger tous les paquets entrants vers le périphérique *ifb*, et d'appliquer les paramètres d'émulation quand les paquets sortent du périphérique *ifb*. La figure 3.6 montre comment appliquer 50 ms de délai aux paquets entrants.

On peut toutefois se poser la question du surcoût engendré par cette solution. En utilisant une GtrcNET-1, une solution matérielle basée sur un FPGA permettant de réaliser de l'émulation réseau matérielle et des mesures réseau avec une grande précision, nous avons mesuré le temps pris par les paquets pour traverser une machine jouant le rôle de routeur. Dans le premier cas, aucune configuration particulière n'a été faite dans TC. Dans le deuxième cas, un périphérique *ifb* a été ajouté, et les paquets entrants y sont redirigés, mais aucun paramètre d'émulation (latence, débit) n'a été défini. La figure 3.7 montre que la différence entre les deux cas est mineure (environ 5.2 μ s), et probablement négligeable dans la plupart des cas.

Le surcoût processeur est malheureusement plus important. Sous une charge réseau importante (1 Gbps, petits paquets UDP générés avec *iperf*), notre sys-

```
# $ETH is our real network device
ETH=eth0

# initialize ifb
modprobe ifb
ifconfig ifb0 up

# add an ingress qdisc to process incoming packets
tc qdisc add dev $ETH ingress

# redirect everything to ifb0
tc filter add dev $ETH parent ffff:\
  protocol ip prio 10\
  u32 match ip src 0.0.0.0/0 flowid ffff:\
  action mirrored egress redirect dev ifb0

# set up netem on ifb0
tc qdisc add dev ifb0 root netem delay 50ms
```

FIG. 3.6: Utilisation d'un périphérique logiciel `ifb` pour appliquer les paramètres d'émulation aux paquets entrants avec TC.

tème de test avait une utilisation processeur d'environ 40% sans `ifb`. Avec `ifb`, la charge processeur est passée à 50%.

3.6 Conclusion

Les émulateurs réseaux sont un outil très utile dans le cadre de l'étude des applications distribuées. Ils permettent aux expérimentateurs de tester facilement leur application dans des conditions expérimentales variées. Toutefois, avant de s'appuyer sur ces émulateurs pour construire des infrastructures plus complexes, ou de chercher à obtenir des résultats avec ces outils, il était important de vérifier les caractéristiques de l'émulateur lui-même.

Dans ce chapitre, nous avons comparé trois émulateurs de liens réseaux : Dummynet, NISTNet, and Linux Traffic Control (TC), dans un grand nombre de configurations. Notre étude dégage 3 configurations permettant d'obtenir de bonnes performances, même si chaque configuration a ses propres inconvénients.

- FreeBSD 6 et Dummynet, avec la fréquence de l'horloge réglée à 10 kHz, permet une émulation précise en entrée et en sortie. Mais la fréquence d'horloge élevée provoque un ralentissement global de la machine.
- NISTNet permet une émulation précise en entrée, mais souffre du même

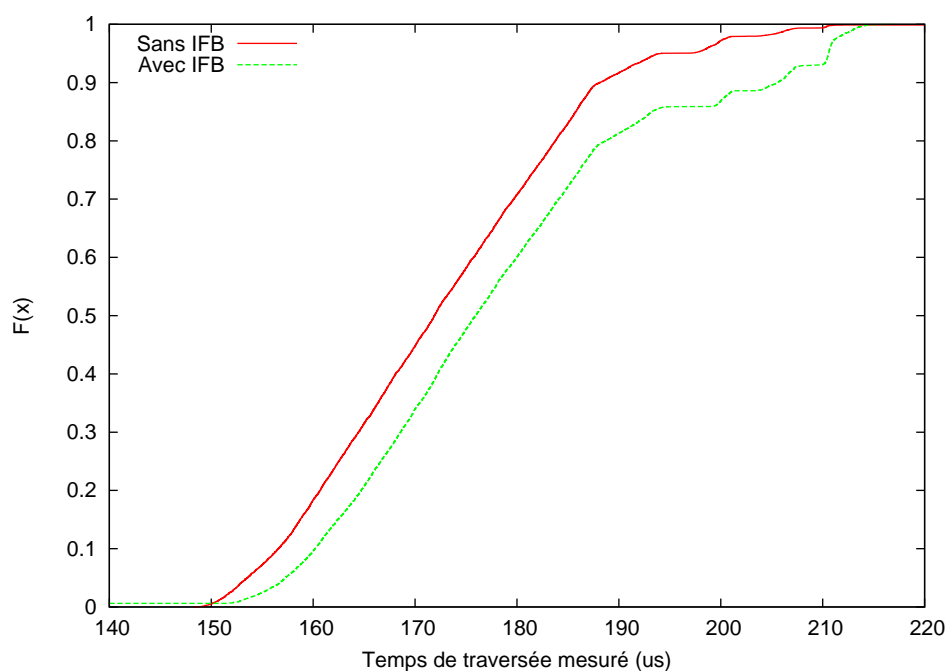


FIG. 3.7: Fonction de répartition du temps de traversée d'une machine agissant comme un routeur, avec et sans redirection des paquets entrants vers un périphérique i fb.

problème que FreeBSD à cause de la fréquence des interruptions.

- Linux TC, avec Netem, permet une émulation très précise avec les noyaux récents, grâce aux *High Resolution Timers*. Mais sa configuration est un peu plus difficile qu'avec FreeBSD et NISTNet.
- Seul Dumynet permet facilement de réaliser une émulation à l'entrée et à la sortie des paquets. NISTNet ne permet qu'un traitement en entrée, et Linux TC requiert l'ajout d'une interface réseau virtuelle pour permettre un traitement en entrée en plus du traitement en sortie.

Dans les chapitres suivants, nous allons illustrer l'utilisation de ces émulateurs dans plusieurs cadres. Au chapitre 4, nous réalisons quelques expériences simples, puis nous utilisons un émulateur pour évaluer une application réseau complexe 5. Enfin, au chapitre 6, nous nous appuyons sur un émulateur pour construire une plate-forme combinant émulation et virtualisation pour étudier les systèmes pair-à-pair.

CONDUITE D'EXPÉRIENCES AVEC UN ÉMULATEUR RÉ- SEAU

4

4.1 Introduction	45
4.2 Latence et shells distants	45
4.3 Performance d'outils de transfert de fichiers	47
4.4 Émulation de topologies réseaux	47
4.5 Conclusion	51

4.1 Introduction

Après avoir comparé les différentes solutions d'émulation de liens réseaux au chapitre précédent, nous nous intéressons maintenant à leur utilisation. Dans ce chapitre, nous illustrons d'abord l'utilisation d'émulateurs pour réaliser des expériences simples mettant en jeu des shells distants et des transferts de données, puis nous montrons comment utiliser ces émulateurs afin d'émuler des topologies réseaux.

4.2 Latence et shells distants

RSH et SSH sont deux applications permettant de se connecter à une machine, et d'y obtenir un *shell*. Tandis que RSH est simple et non sécurisé, SSH est développé dans l'optique d'en fournir une alternative sécurisée, et intègre donc de complexes mécanismes cryptographiques, ainsi que d'autres fonctionnalités comme l'établissement de tunnels. Mais toutes ces fonctionnalités ont un coût, et SSH est plus lent lors de l'établissement des connexions, en particulier quand la latence entre le client et le serveur est importante. Cela peut devenir un problème dans les environnements où la sécurité fournie par SSH n'est pas forcément utile (car elle est assurée par d'autres moyens).

Dans cette expérience, nous cherchons à montrer l'influence de la latence sur le temps nécessaire à l'établissement d'une connexion pour exécuter une commande simple (nous avons utilisé respectivement `rsh server "true"` et `ssh server "true"`).

Nous avons utilisé deux machines, utilisant Linux avec TC. Sur chaque ma-

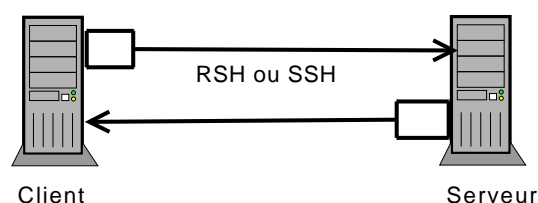


FIG. 4.1: Configuration expérimentale. La latence est ajoutée sur les paquets sortant du client et du serveur

chine, les paramètres d'émulation sont appliqués sur les paquets sortant de la machine, comme décrit dans la figure 4.1. Nous avons fait varier la latence sur le lien entre les deux machines de 0 à 150 ms (latence atteignable sur un lien France-Australie, par exemple). Nous avons répété les mesures 20 fois pour chaque paramètre de latence, et les résultats ont été très stables.

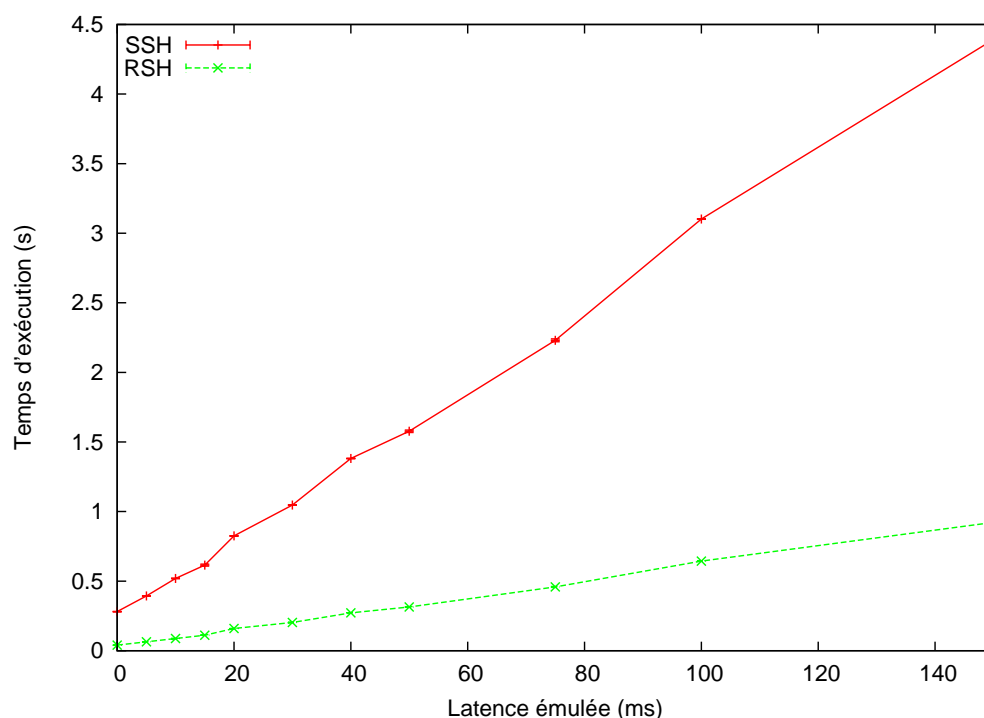


FIG. 4.2: Temps nécessaire pour exécuter une commande simple sur un serveur distant en utilisant RSH ou SSH.

La figure 4.2 montre qu'à la fois RSH et SSH deviennent plus lents quand la latence augmente. Cependant, la latence affecte SSH bien plus que RSH (4.7 fois plus). Cela s'explique par le fait que les négociations qui se produisent pendant l'établissement d'une connexion SSH nécessitent plusieurs allers-retours sur le réseau, alors que l'établissement d'une connexion RSH est bien plus simple.

4.3 Performance d'outils de transfert de fichiers

De nombreux outils, plus ou moins complexes et efficaces, permettent de transférer des fichiers d'une machine à une autre. Parmi eux, `scp` et `rsync` répondent à un besoin identique : le transfert de fichiers (ou de répertoires) entre deux machines, en utilisant le protocole SSH comme support. Ces deux outils sont d'ailleurs fréquemment utilisés sur les grilles.

Toutefois, les algorithmes utilisés par ces deux outils sont très différents, et cela influence leurs performances. Dans leur mode "récursif" (permettant de transférer une hiérarchie de répertoires, ainsi que les fichiers présents dans ces répertoires), `scp` négocie le transfert de chaque fichier séparément, tandis que `rsync` commence par négocier le transfert de tous les fichiers, puis utilise un pipeline pour le transfert.

En utilisant la même configuration expérimentale que précédemment (figure 4.1), nous avons mesuré le temps nécessaire pour transférer 120 fichiers extraits d'une partie du code source du noyau Linux. La taille totale des fichiers transférés était de 2.1 Mo. Nous avons fait varier à la fois la latence et la bande passante du lien réseau utilisé, afin d'évaluer l'importance de ces deux paramètres.

La figure 4.3 montre que les performances de Rsync sont bien meilleures que celles de SCP. En particulier :

- Avec une latence de 20 ms, le temps de transfert décroît linéairement lorsque la bande passante du lien augmente jusqu'à 1 Mbps avec SCP, et jusqu'à 10 Mbps avec Rsync. À ce stade là, le temps de transfert est probablement dominé par le temps d'établissement de la connexion (temps de transfert de 2.9 s).
- Avec une bande passante fixée à 10 Mbps, SCP est beaucoup plus affecté par une augmentation de la latence que Rsync (passage de 2.0 s à 10.8 s pour Rsync, de 6.1 s à 80.9 s pour SCP, lorsque la latence passe de 0 ms à 100 ms).

4.4 Émulation de topologies réseaux

Si les émulateurs présentés au chapitre précédent permettent de modifier les caractéristiques d'un lien réseau, il est souvent nécessaire d'émuler une topologie plus complexe, comprenant plusieurs machines, afin d'étudier des problèmes d'interactions de flux ou de congestion.

Cela peut être fait avec des émulateurs de topologie réseau comme Modelnet. Mais utiliser de tels émulateurs est difficile. Dans de nombreux cas, il est possible d'utiliser des émulateurs de liens pour modéliser une topologie plus complexe, sans perdre en précision. Dans cette partie, nous nous plaçons dans le cas où nous disposons des prérequis suivants :

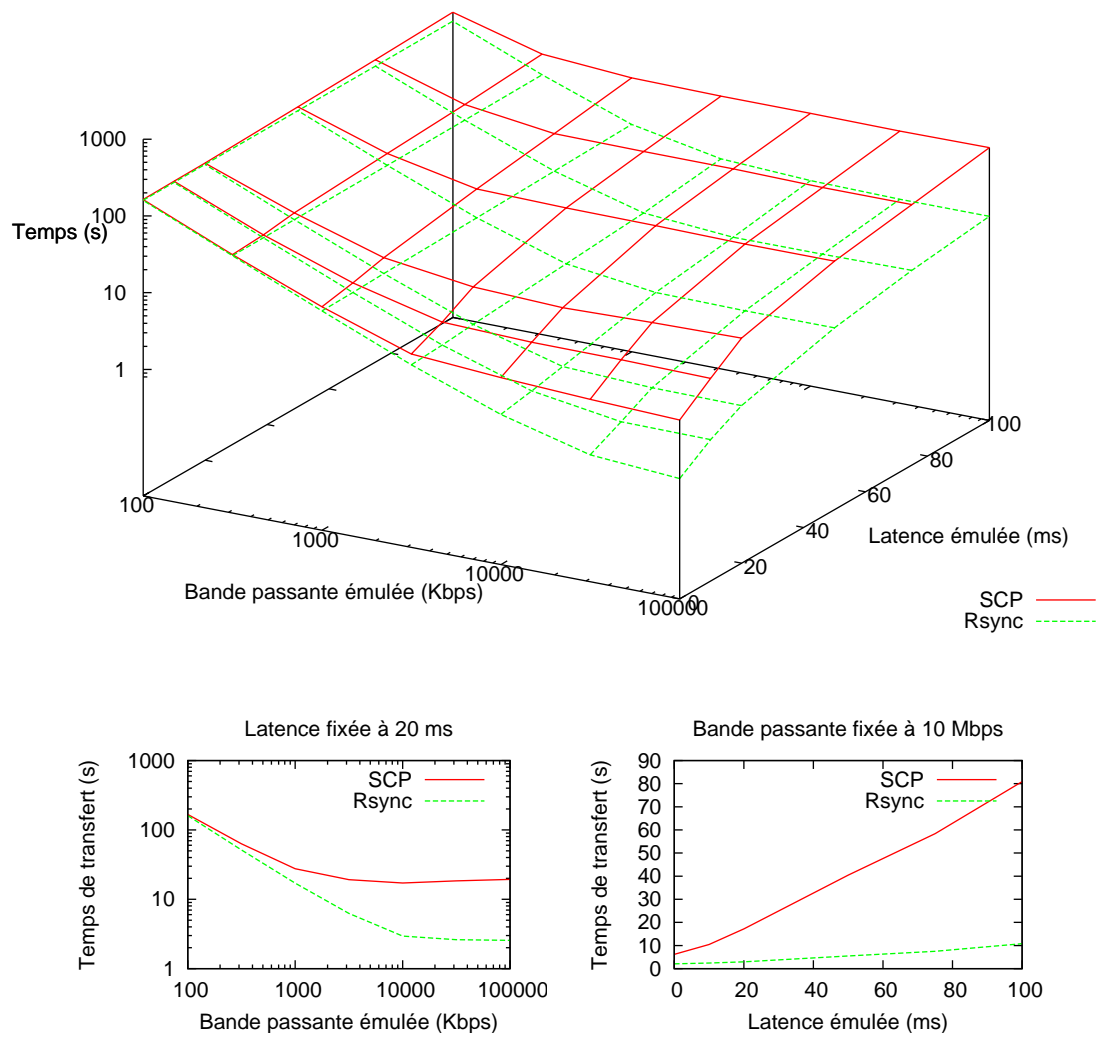


FIG. 4.3: Temps de transfert de 120 fichiers (taille totale : 2.1 Mo) en utilisant SCP et Rsync, avec différentes conditions de latence et de débit.

- un ensemble de machines situées sur le même réseau ethernet (niveau 2), par exemple les nœuds d'un cluster de Grid'5000 ;
- Ce réseau ethernet peut être considéré comme non-bloquant (le commutateur ethernet dispose d'un fond de panier de débit suffisant) ;
- l'émulateur utilisé permet le filtrage en entrée et en sortie (donc Dummy-net, ou TC/Netem avec IFB, mais pas NISTNet).

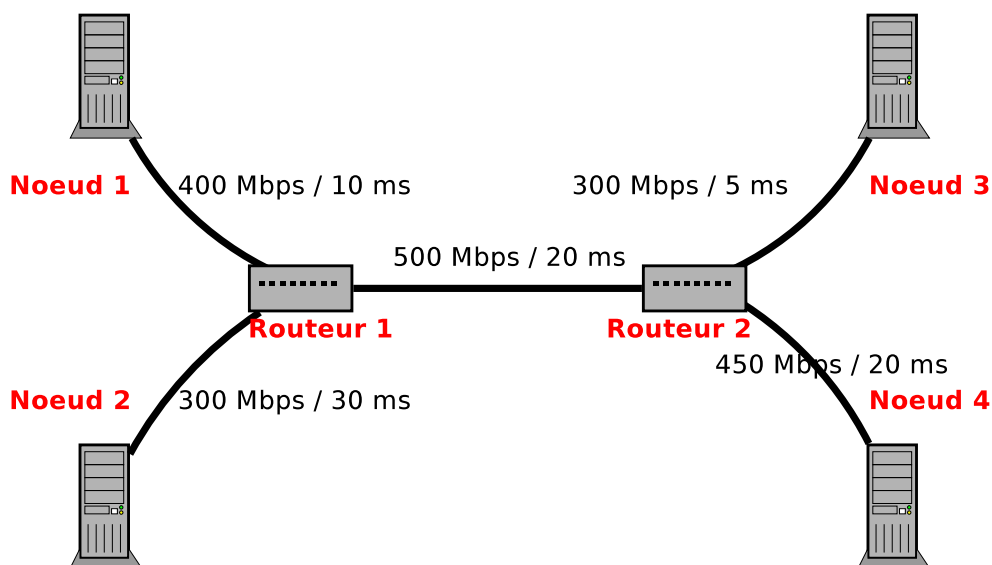


FIG. 4.4: Topologie *Dumbbell* à émuler

Nous cherchons à émuler la topologie en *Dumbbell* décrite dans la figure 4.4. Ce type de topologie est souvent utilisé pour étudier des problèmes d'interactions entre flux (congestion, famine), et est similaire aux topologies reliant deux clusters entre eux dans une grille, ou deux sites d'une entreprise reliés par un réseau privé virtuel (VPN).

Afin d'émuler cette topologie, nous allons procéder comme suit :

- Les deux routeurs sont remplacés par des machines classiques. Ces machines-routeurs doivent disposer de suffisamment d'interfaces réseaux pour émuler l'ensemble de leurs liens. Nous considérons ici que nous disposons de trois interfaces sur les deux machines-routeurs (c'est le cas sur les machines `netgdx` de GridExplorer, par exemple).
- Afin d'aiguiller (*router*) les trames ethernet vers les bonnes machines intermédiaires, les tables ARP des machines sont modifiées (il serait également possible de modifier l'adressage de toutes les machines prenant part à l'expérience afin de réaliser un *routing niveau 3* au lieu d'un *routing niveau 2*, mais cela est plus contraignant puisqu'il faut prendre des précautions par-

ticulières pour pouvoir continuer à accéder aux noeuds depuis la machine de contrôle d'expérience (route particulière pour cette machine, interface IP supplémentaire sur l'interface utilisée, ...).

- Sur chaque machine, l'émulateur réseau est configuré pour reproduire les caractéristiques souhaitées.

Dans le tableau ci-dessous, nous utilisons les notations suivantes :

$MAC_{R1,N1}$ adresse ethernet de l'interface du routeur 1 sur le lien R1-N1

MAC_{N1} adresse ethernet de la seule interface du nœud 1

IP_{N1} adresse IP de la seule interface du nœud 1

Avec ces notations, nous utilisons les tables ARP suivantes afin d'émuler la topologie de la figure 4.4 :

Machine configurée	Machine cible			
	IP_{N1}	IP_{N2}	IP_{N3}	IP_{N4}
Nœud 1		$MAC_{R1,N1}$	$MAC_{R1,N1}$	$MAC_{R1,N1}$
Nœud 2	$MAC_{R1,N2}$		$MAC_{R1,N2}$	$MAC_{R1,N2}$
Nœud 3	$MAC_{R2,N3}$	$MAC_{R2,N3}$		$MAC_{R2,N3}$
Nœud 4	$MAC_{R2,N4}$	$MAC_{R2,N4}$	$MAC_{R2,N4}$	
Routeur 1	MAC_{N1}	MAC_{N2}	$MAC_{R2,R1}$	$MAC_{R2,R1}$
Routeur 2	$MAC_{R1,R2}$	$MAC_{R1,R2}$	MAC_{N3}	MAC_{N4}

Cette technique nécessite l'ajout, dans les tables ARP de chaque nœud participant à l'expérience, d'une entrée par nœud participant. Toutefois, cela est peu coûteux : ces entrées ne sont pas différentes des entrées normalement présentes dans les tables ARP des machines, et bénéficient donc des mêmes optimisations pour la recherche dans ces tables (utilisation d'une table de hachage dans Linux, par exemple).

Il est important de noter que, par défaut, lorsqu'un nœud reçoit un paquet pour lequel il sait qu'un chemin plus court est disponible (sans passer par lui), il répond au paquet par un paquet ICMP *Redirect* afin d'informer la machine source de l'existence d'un chemin plus court. Avant de démarrer des expériences, il faut donc désactiver ce comportement (`sysctl -w net.ipv4.conf.all.accept_redirects=0; sysctl -w net.ipv4.conf.all.send_redirects=0`).

Si dans cet exemple, nous avons utilisé des machines-routeurs disposant de 3 interfaces réseaux, il est possible de réaliser le même montage avec des machines disposant de moins d'interfaces. Toutefois, il faut alors s'assurer que la somme des débits des liens connectés à un noeud ne dépasse pas les capacités physiques de la machine. Dans notre exemple, en particulier, on utilisera obligatoirement des noeuds disposant de 2 interfaces Gigabit. Émuler plusieurs liens virtuels sur le même lien physique pose également le problème de la concurrence d'accès et du contrôle de congestion : la machine ne pouvant recevoir deux trames ethernet simultanément sur la même interface, certaines trames seront forcément retardées, alors qu'elles pourraient être reçues en même temps si la machine disposait

de plus d'interfaces.

4.5 Conclusion

Au chapitre précédent, nous avons comparé différentes solutions d'émulation de liens réseaux, afin de dégager les configurations permettant de réaliser des expériences dans de bonnes conditions. Dans ce chapitre, nous avons donné quelques exemples d'utilisation de ces émulateurs. D'abord, nous avons montré un exemple de l'utilisation de ces émulateurs pour évaluer les performances d'outils couramment utilisés : d'abord RSH et SSH, puis SCP et Rsync. Puis nous avons montré comment ces émulateurs pouvaient être utilisés pour émuler des topologies réseaux.

Ces émulateurs sont donc des outils utiles dans le cadre de l'expérimentation sur des systèmes distribués, permettant d'obtenir rapidement des résultats utiles pour la compréhension et l'optimisation des protocoles et des solutions logicielles distribués. Sans l'utilisation d'émulateurs réseaux, il est très difficile d'obtenir ces mêmes résultats : une infrastructure spécifique est nécessaire, dont l'utilisation pose en général bien d'autres problèmes.

TUNNEL IP SUR DNS ROBUSTE 5

5.1	Introduction	53
5.2	Canaux cachés dans le DNS	54
5.2.1	Principe général	54
5.2.2	Implémentations existantes	55
5.2.3	Conclusion	56
5.3	TUNS	57
5.4	Évaluation des performances	59
5.4.1	Influence de la latence	60
5.4.2	Performances sous conditions réseaux dégradées	63
5.5	Adaptation des paramètres du tunnel	63
5.6	Perspectives	66
5.7	Conclusion	67

5.1 Introduction

De nombreux réseaux ne permettent qu'un accès restreint à Internet. C'est le cas de réseaux internes d'entreprises, de réseaux sans-fils, par exemple dans des hôtels, voire de l'ensemble de l'accès à Internet dans certains pays. De nombreuses personnes ont tenté d'utiliser un des protocoles non filtrés pour obtenir un accès plus complet à Internet, en établissant un canal de communication caché. Il existe par exemple des tunnels IP au-dessus d'HTTP, d'HTTPS ou ICMP.

Dans ce chapitre, nous nous intéressons aux canaux cachés utilisant le protocole DNS, utilisé pour la résolution de noms en adresses IP. Le protocole DNS est intéressant dans ce cadre, car il est omniprésent : il est très difficile de fournir un accès, même restreint, à Internet, sans fournir un service DNS (un cas relativement commun où cela est possible malgré tout est les réseaux ne fournissant qu'un accès au Web, à travers un serveur proxy ; dans ce cas, la résolution DNS peut se faire sur le proxy). De plus, sur les réseaux où une authentification ou un paiement via un portail captif¹ sont nécessaires, le serveur DNS ne peut pas retourner des résultats incorrects jusqu'à ce que l'utilisateur soit autorisé : à cause des effets de cache, cela empêcherait ensuite l'utilisateur d'accéder à certains ser-

¹Réseau où, une fois connecté, mais pas encore authentifié, les requêtes HTTP de l'utilisateur sont redirigées vers un site web spécifique lui proposant de s'authentifier ou de payer pour obtenir l'accès au réseau.

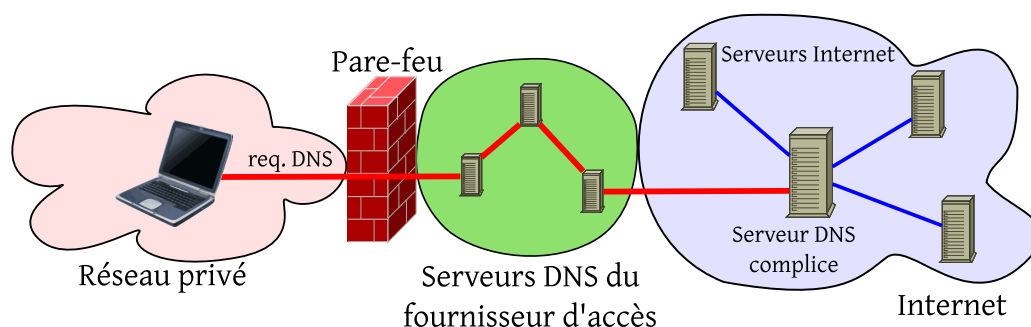


FIG. 5.1: Principe général des canaux cachés dans le DNS

veurs, puisque son navigateur, par exemple, conserverait dans son cache DNS un résultat erroné. En conséquence, les réseaux rencontrés dans les aéroports ou les hôtels, par exemple, donnent le plus souvent un accès complet au protocole DNS, même avant que l'utilisateur se soit authentifié.

Néanmoins, utiliser des requêtes DNS pour faire transiter des informations quelconques n'est pas trivial, car le protocole DNS n'a évidemment pas été conçu dans cette optique. Il restreint la taille des paquets et des enregistrements DNS, obligeant les implémentations de canaux cachés à faire plusieurs compromis entre respect du protocole et performance, ce qui les rend facilement détectables, et donc filtrables. Ces compromis sont difficiles à définir, car il est compliqué de réaliser des expériences en prenant en compte différentes configurations de réseaux Wifi, d'autant plus que du point de vue d'un utilisateur, les détails de l'infrastructures sont souvent invisibles.

Dans la suite de ce chapitre, nous décrivons les principaux généraux des tunnels IP au-dessus de DNS et les implémentations existantes, puis nous présentons TUNS, notre prototype de tunnel IP au-dessus de DNS, et le comparons avec les autres implémentations. Nous évaluons ensuite l'ensemble de ces solutions à l'aide d'un émulateur réseau, ce qui nous permet de réaliser facilement un ensemble d'expériences dans des conditions variées.

5.2 Canaux cachés dans le DNS

5.2.1 Principe général

La figure 5.1 présente le principe général des canaux cachés dans le DNS. Le client est connecté sur un réseau où les communications avec le monde extérieur sont filtrées par un pare-feu. Pour communiquer avec le monde extérieur, le client encapsule des données dans des requêtes DNS destinées au domaine DNS servi géré par un serveur complice situé sur Internet. Ces requêtes sont envoyées au serveur DNS du réseau local (une communication directe avec un serveur DNS

autre que celui du réseau locale est presque systématiquement filtrée), transitent à travers l'infrastructure DNS du fournisseur d'accès, puis atteignent le serveur DNS complice. Celui-ci décode les données, et les envoie à leur destination finale, comme si elles provenaient directement du serveur complice.

Les réponses sont traitées d'une manière similaire : comme les données ont été envoyées du serveur DNS complice vers le serveur Internet destinataire, ce dernier répond au serveur DNS complice, qui encapsule les données dans des réponses DNS qui seront renvoyées au client. Toutefois, comme les réponses DNS ne peuvent être envoyées qu'en réponse à une requête DNS, le client doit sonder régulièrement le serveur avec des requêtes DNS, pour vérifier s'il a des données à envoyer.

Comme les requêtes et les réponses DNS voyagent à travers l'infrastructure DNS du fournisseur d'accès, ils ne doivent pas être trop différentes de paquets DNS normaux. Si les paquets créés par le tunnel ne sont pas conformes à la RFC, ou trop faciles à détecter, ils seront filtrés. Par exemple, les données envoyées du client vers le serveur sont généralement encodées dans le nom sur lequel porte la requête, en utilisant un encodage Base32 (5 bits d'information par caractère) ou Base64 (6 bits par caractère). Mais le protocole DNS n'autorise que 63 caractères différents dans les noms (`[a-z][A-Z][0-9]-`), obligeant les implémentations utilisant Base64 à rajouter un caractère non conforme à cet ensemble. Un autre problème est que l'encodage Base64 est sensible à la casse, alors que les serveurs DNS sont autorisés à changer la casse des requêtes (*When data enters the domain system, its original case should be preserved whenever possible.* [72], section 2.3.3).

Les implémentations existantes font donc différents compromis, que nous décrivons dans le paragraphe suivant.

5.2.2 Implémentations existantes

On distingue deux types de canaux cachés dans DNS :

- les canaux cachés qui fournissent un tunnel IP au-dessus de DNS (qui permettent de transmettre des paquets IP à travers le canal de communication) ;
- les canaux cachés qui fournissent une interface de type *socket*, comme une connexion TCP, permettant par exemple d'établir une connexion SSH à travers le tunnel.

5.2.2.1 Tunnels IP dans DNS

Les tunnels IP dans DNS utilisent généralement une interface `tun` (au niveau 3) ou `tap` (au niveau 2), permettant à l'utilisateur de router les paquets vers cette interface. Leur utilisation est donc complètement transparente pour les applications.

NSTX [73] est la plus ancienne de ces implémentations. Pour encoder les informations dans les requêtes, il utilise un encodage Base64 non-conforme (utilisant "_" en plus des 63 caractères autorisés dans la RFC 1035). Les réponses sont encodées dans des enregistrements TXT.

Iodine [74] est un autre projet, plus récent. Il utilise un encodage Base32 ou un encodage Base64 non-conforme pour encoder les données (choisi avec une option de configuration). Les réponses sont envoyées dans des enregistrements NULL. Les enregistrements NULL sont décrits dans la RFC 1035, servent de conteneurs pour des extensions expérimentales de DNS, et peuvent contenir n'importe quel type d'informations. De plus, Iodine utilise EDNS0 [75] pour augmenter la taille des réponses au-delà de la limite de 512 octets imposée par la RFC 1035.

À la fois NSTX et Iodine découpent les paquets IP en plusieurs paquets DNS de taille plus petite, les envoient séparément, puis réassemblent les paquets IP à l'autre extrémité du tunnel.

5.2.2.2 Tunnels TCP dans DNS

La deuxième catégorie de tunnels ne fournit qu'une seule connexion TCP. L'utilisateur l'utilise généralement pour établir une connexion SSH, puis utilise les fonctionnalités de SSH (*port forwarding*, proxy SOCKS) pour établir d'autres connexions.

Le principal désavantage de ces solutions est qu'elles doivent fournir un canal de communication fiable au-dessus d'un protocole non-fiable, et donc gérer les pertes, retransmissions, réordonnements et duplications de paquets DNS.

OzymanDNS [76] est l'implémentation de ce type la plus connue. Il utilise un encodage Base32 pour les requêtes, et des enregistrements TXT pour les réponses, ainsi que l'extension EDNS0. Pendant nos tests, nous avons rencontré de nombreux *plantages* de OzymanDNS.

dns2tcp [77] est une autre implémentation, plus récente. Il utilise des enregistrements TXT, et un encodage Base64 non-conforme (utilisant le caractère "/" en plus des 63 caractères autorisés par la RFC).

5.2.3 Conclusion

Nous avons testé ces quatre implémentations sur une douzaine de réseaux réels différents (plusieurs connexions ADSL, réseaux d'universités ou de laboratoires, d'hôtels, d'aéroports, de gares, ...). Avec chaque implémentation, nos tests ont échoué dans une majorité de cas. Cela était prévisible :

- NSTX et dns2tcp peuvent être facilement bloqués en interdisant les requêtes portant sur des noms non-conformes ;

- Toutes les implémentations peuvent être bloquées en interdisant les requêtes utilisant des types d'enregistrements DNS (TXT, NULL) ou des extensions (EDNS0) rarement utilisés. Même si ce n'est pas possible sur certains réseaux (les enregistrements TXT sont par exemple utilisés par le système anti-spam SPF), cela ne pose pas de réel problème sur des points d'accès Wifi commerciaux.

5.3 TUNS

Après avoir étudié les implémentations existantes, nous avons écrit notre propre prototype, appelé TUNS. Nous nous sommes attachés à n'utiliser que les fonctionnalités standard et largement utilisées de DNS, pour rendre TUNS plus difficile à bloquer.

TUNS est un tunnel IP sur DNS, écrit en Ruby, et disponible sous la licence GNU GPL². Contrairement aux autres solutions, qui utilisent des enregistrements TXT ou NULL rarement utilisés pour des raisons légitimes, TUNS n'utilise que des enregistrements CNAME. Il encode les paquets IP en utilisant un encodage Base32 (figure 5.2). Contrairement à NSTX et Iodine, TUNS ne découpe pas les paquets IP en plusieurs petits paquets DNS. À la place, le MTU de l'interface du tunnel est réduit à une valeur beaucoup plus petite (140 octets par défaut), et le système d'exploitation se charge lui-même de découper les paquets IP en utilisant la fragmentation IP. Cela supprime le besoin d'implémenter une machine à états pour retransmettre les paquets DNS perdus, et augmente la fiabilité du tunnel en cas de perte de paquets, mais réduit la quantité de données utiles qui peuvent être transmises, puisque les en-têtes IP sont répétés dans chaque paquet DNS.

Quand des données à transmettre sont présentes côté client, elles sont encapsulées et transmises immédiatement. Pour recevoir des données du serveur, le client sonde régulièrement le serveur avec de courtes requêtes DNS. Si le serveur a des données à transmettre au client, il répond au client immédiatement, en encapsulant les données à transmettre dans la réponse. S'il n'a pas de données à transmettre, il attend pendant un délai de garde configurable avant d'envoyer une réponse vide. Si des données à transmettre arrivent pendant cette attente, elles sont transmises immédiatement. Ce mécanisme s'est montré très important pour réduire la latence ressentie dans les communications interactives (comme les sessions SSH, par exemple).

Un autre problème qui est pris en compte dans TUNS est le fait que les infrastructures DNS envoient parfois des requêtes en doublon (probablement, d'une manière égoïste, pour augmenter leurs chances d'obtenir une réponse malgré les pertes éventuelles de paquets DNS) : une seule requête d'un client peut être dupliquée par un serveur intermédiaire, et le serveur DNS final recevra cette requête

²TUNS peut être téléchargé sur <http://www-id.imag.fr/~nussbaum/tuns.php>

Le client envoie un paquet de données vers le serveur :

```
Domain Name System (query)
dIUAAAVAAABAAAQABJ5K4BKBVAHAKQNICBAAAOS5TD4ASKPSQIJ[...]
MRTGQ2TMNY0.domain.tld: type CNAME, class IN
```

Le client envoie un court paquet que le serveur utilisera pour envoyer une réponse :

```
Domain Name System (query)
r882.domain.tld: type CNAME, class IN
```

Le serveur acquitte les données qui ont été envoyées. Dans sa réponse, il indique la taille de la file d'attente de paquets de données côté serveur (ici 4 paquets), pour que le client puisse ajuster le nombre de requêtes qu'il va envoyer pour recevoir les données en attente côté serveur :

```
Domain Name System (response)
Queries
dIUAAAVAAABAAAQABJ5K4BKBVAHAKQNICBAAAOS5TD4ASKPSQI[...]
MRTGQ2TMNY0.domain.tld: type CNAME, class IN
Answers
dIU[....]Y0.domain.tld: type CNAME,
class IN, cname l4.domain.tld
```

Le serveur envoie une réponse contenant des données :

```
Domain Name System (response)
Queries
r882.domain.tld: type CNAME, class IN
Answers
r882.domain.tld: type CNAME, class IN, cname dIUAAAVCWIU
AAAQABHEIMBKBVALAKQNI BAAAC4SNTD4ASOQCQIJEK5VABAAEAS[...]
DC.MRTGNY0.domain.tld
```

Plus tard, le serveur répond qu'il n'a pas de données à envoyer au client :

```
Domain Name System (response)
Queries
r993.domain.tld: type CNAME, class IN
Answers
r993.domain.tld: type CNAME, class IN,
cname dzero.domain.tld
```

FIG. 5.2: Contenu des paquets DNS échangés entre un client et un serveur TUNS, capturés avec l'analyseur de trafic wireshark.

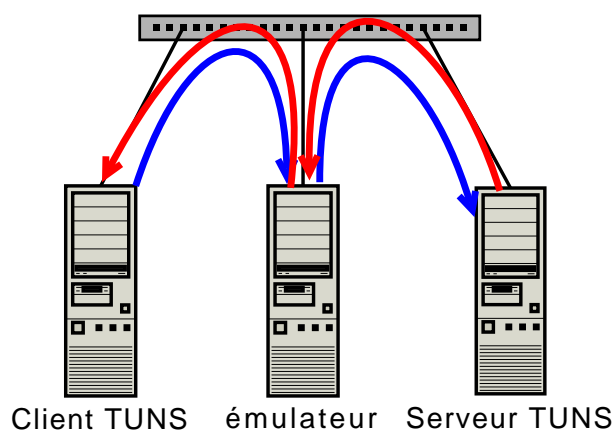


FIG. 5.3: Configuration expérimentale.

deux fois. Dans ce cas, le serveur DNS doit retourner la même réponse aux deux requêtes, ou un paquet IP sera perdu. Un cache a été rajouté dans TUNS pour prendre en compte ce problème.

Nous avons testé TUNS sur un grand nombre de réseaux (incluant ceux sur lesquels nous avons testé les quatre implémentations décrites précédemment). Jusqu'ici, nous n'avons pas rencontré de réseau où TUNS ne fonctionnait pas correctement.

5.4 Évaluation des performances

Nous avons évalué les performances de NSTX 1.1 beta6, Iodine 0.4.1, et TUNS 0.9.2. Afin de les comparer dans un grand ensemble de conditions réseaux, nous avons utilisé de l'émulation réseau (notre configuration expérimentale est décrite en figure 5.3). Les expériences ont été effectuées sur 3 nœuds de Grid'5000 utilisant Linux 2.6.22 sur l'architecture x86 (et non x86-64), pour pouvoir bénéficier de l'horloge à haute fréquence, qui n'existait pas encore sur x86-64 dans le noyau 2.6.22. L'outil d'émulation TC (Traffic Control) a été utilisé pour retarder les paquets quand ils sortent du nœud-émulateur, à la fois lorsqu'ils vont du client vers le serveur, et du serveur au client. Les mesures de latence ont été effectuées avec `ping`, avec un intervalle de mesure d'une seconde. Les mesures de débit ont été effectuées avec `iperf`. Pour toutes les expériences, la bande passante du réseau sous-jacent a été limitée à 1 Mbps, pour reproduire les conditions qu'on peut trouver sur des réseaux sans-fil à faible signal ou saturés (le réalisme de cette limitation de bande passante a été confirmé en mesurant la bande passante de quelques réseaux sans-fil).

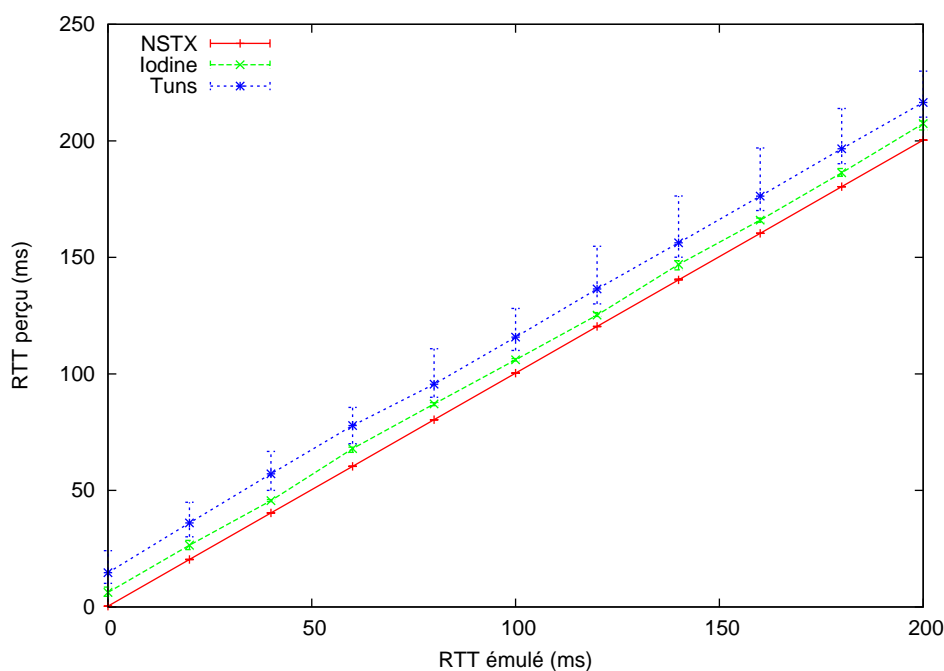


FIG. 5.4: Latence perçue, mesurée avec `ping` depuis le client. Les barres verticales indiquent les valeurs minimum et maximum sur 20 mesures.

5.4.1 Influence de la latence

Dans notre première série d'expériences, nous nous sommes intéressés à déterminer comment les différentes implémentations réagissent à des situations où la latence est importante. De telles conditions sont fréquentes avec de tels outils, par exemple quand le client se connecte à un serveur situé sur un autre continent. La figure 5.4 montre la latence perçue à travers le tunnel quand la latence du réseau sous-jacent entre le client et le serveur varie entre 0 ms et 100 ms (faisant donc varier le temps aller-retour jusqu'à 200 ms). Toutes les implémentations se comportent de manière similaire dans cette expérience, même si TUNS et Iodine montrent un léger surcoût, constant, comparé à NSTX.

La figure 5.5 décrit le débit montant (du client au serveur) disponible à travers le tunnel. Iodine donne les meilleurs résultats, NSTX est légèrement plus lent (en particulier quand la latence est faible), et TUNS est bien plus lent que les deux autres implémentations. Cela s'explique par plusieurs raisons :

- TUNS utilise un encodage Base32, tandis que NSTX et Iodine utilisent un encodage Base64, plus efficace mais non conforme à la RFC ;
- NSTX et Iodine découpent les paquets IP, tandis que TUNS utilise la fragmentation IP. Avec TUNS, les en-têtes IP sont répétés dans chaque paquet, laissant donc moins d'espace pour les données ;
- NSTX et Iodine sont écrits en C, tandis que TUNS est écrit en Ruby. L'utili-

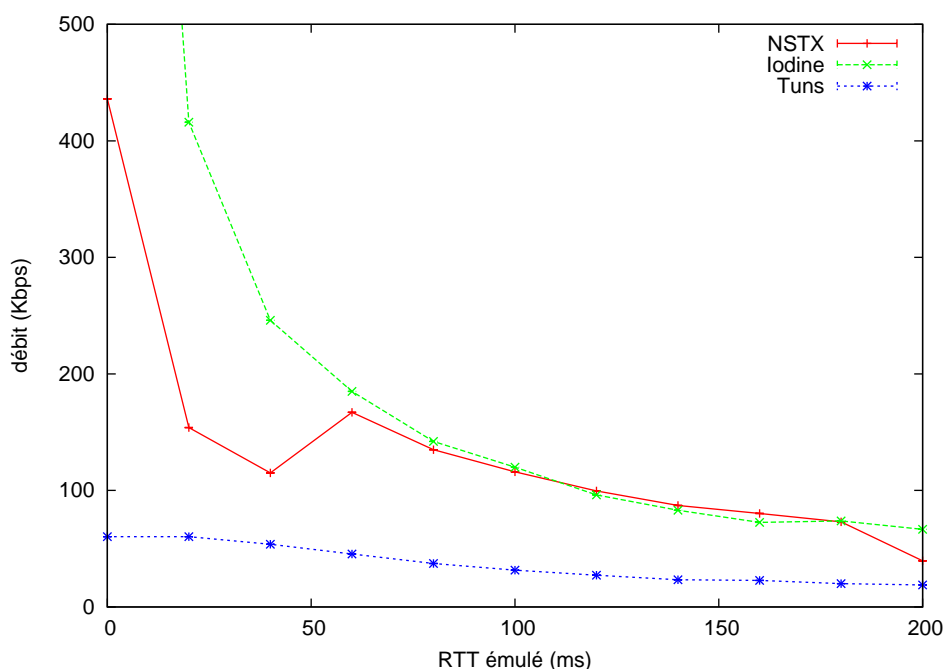


FIG. 5.5: Débit montant disponible (client vers serveur).

sation d'un langage interprété provoque forcément un surcoût lors du traitement des paquets. L'utilisation d'un *profiler* a montré que la bibliothèque DNS pour Ruby que nous avons utilisée était un goulot d'étranglement significatif. Avec la version de développement de Ruby (Ruby 1.9), les performances sont d'ailleurs légèrement meilleures.

La figure 5.6 décrit la latence du tunnel, lorsque les pings sont initiés côté serveur. Elle montre l'efficacité du mécanisme de sondage utilisé par le tunnel. Alors que NSTX et TUNS donnent des résultats similaires, la latence varie énormément avec Iodine. Des investigations plus poussées montrent que Iodine répond immédiatement lorsqu'il est sondé par le client, même s'il n'a rien à envoyer. Au lieu de cela, NSTX et TUNS attendent un certain temps s'ils n'ont rien à envoyer. Ainsi, si des données à transmettre arrivent pendant cette attente, elles peuvent être envoyées immédiatement. Cette optimisation a un désavantage : si le serveur met trop de temps à répondre, l'un des serveurs DNS intermédiaires (faisant partie de l'infrastructure DNS du fournisseur d'accès, par exemple) peut considérer qu'une erreur est apparue, et que le serveur n'a pas répondu dans le délai imparti. La réponse du serveur serait alors perdue. Dans TUNS, la durée pendant laquelle le serveur est autorisé à attendre peut être configurée depuis le client, à l'aide d'une requête spéciale, afin de pouvoir s'adapter à des infrastructures DNS différentes.

Comme vu dans la figure 5.7, le débit descendant est similaire au débit mon-

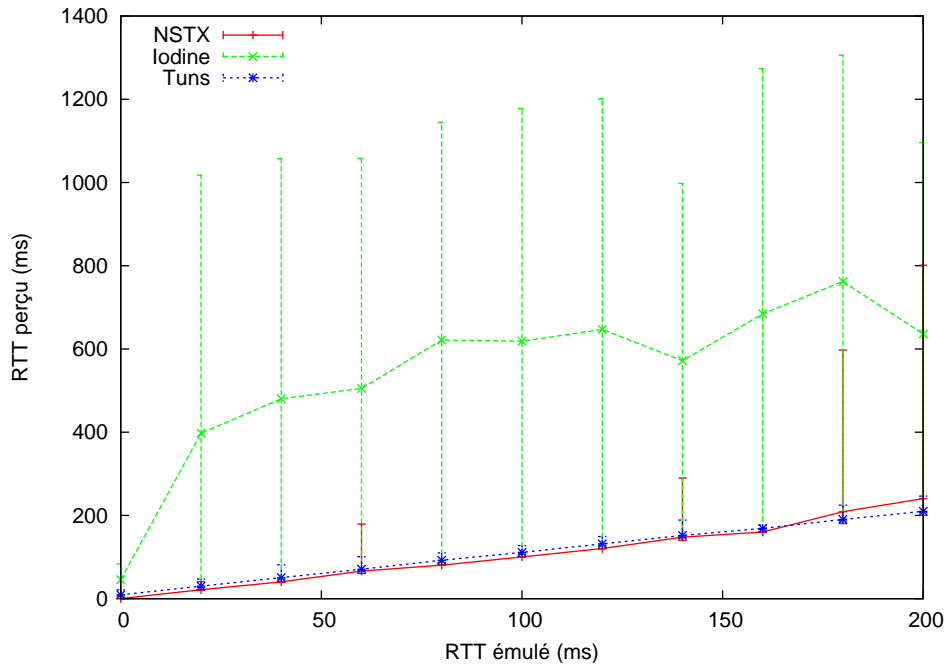


FIG. 5.6: Latence perçue, mesurée avec ping depuis le serveur.

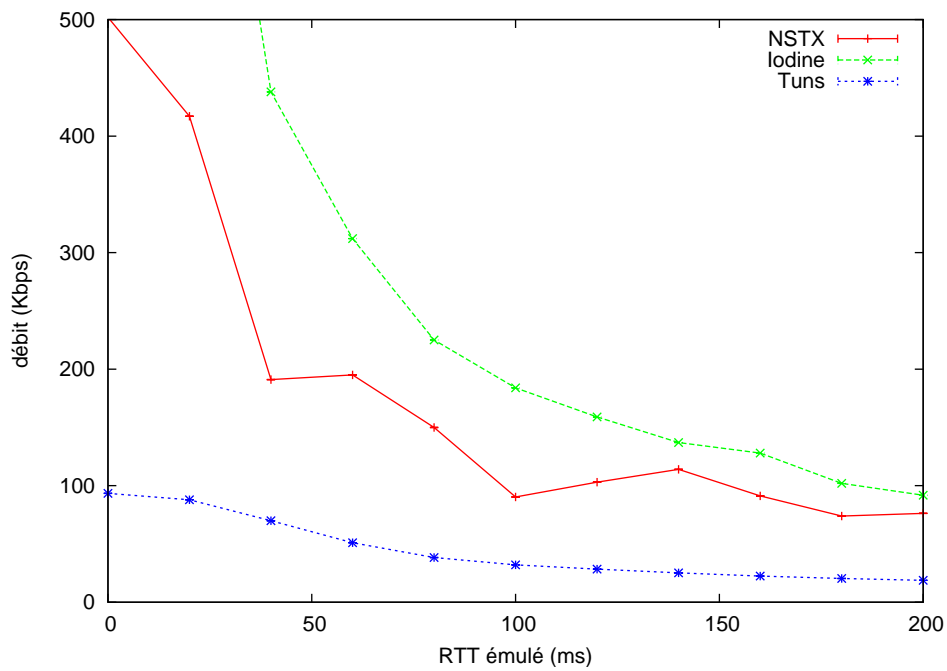


FIG. 5.7: Débit descendant disponible (serveur vers client).

tant (figure 5.5). Pendant toutes nos expériences, NSTX a fourni des résultats plus variables que TUNS et Iodine.

5.4.2 Performances sous conditions réseaux dégradées

Après cette première série d'expériences, nous nous sommes intéressés à la manière dont les différentes implémentations se comportent lorsque les conditions réseaux ne sont pas optimales. Nous avons émulé un réseau avec 5% de pertes de paquets (réparties uniformément), et une latence variant de 10 ms autour de la valeur centrale définie (en suivant une loi normale), provoquant des réordonnements de paquets.

Les résultats (figure 5.8 et 5.9) montrent que, alors que la latence du tunnel n'est quasiment pas affectée, le débit est clairement pénalisé par de tels paramètres. Alors que TUNS était l'implémentation la plus lente dans des conditions réseaux parfaites, il est maintenant plus performant que NSTX.

Cela est probablement causé par le fait que Iodine et NSTX découpent les paquets IP en plusieurs paquets DNS : quand un paquet DNS est perdu, Iodine et NSTX doivent soit prendre en charge la retransmission de ce paquet, soit ignorer plusieurs autres paquets DNS, morceaux du même paquet IP, et attendre que ce paquet IP soit retransmis.

5.5 Adaptation des paramètres du tunnel

Même si TUNS fait un compromis entre performances et respect du protocole, pour pouvoir fonctionner correctement sur plus de réseaux que les autres implémentations, il y a des cas où ce compromis n'est pas nécessaire. Il est intéressant de pouvoir ajuster les paramètres du tunnel, pour se rapprocher des limites de ce que le réseau auquel on est connecté tolère. Cependant, ce changement de configuration doit pouvoir être fait à distance, depuis le client : l'utilisateur ne doit pas risquer de *s'enfermer dehors* en essayant des paramètres différents.

Le client TUNS peut envoyer des requêtes DNS spéciales, permettant de changer la configuration du serveur. Actuellement, les paramètres suivants peuvent être changés :

- Le *Maximum Transmission Unit* de l'interface du tunnel : certaines infrastructures DNS autorisent l'envoi de paquets plus longs (plus de 512 octets). Cela permet d'augmenter la longueur des paquets IP passant à travers le tunnel, et donc d'augmenter l'efficacité du tunnel.
- Le délai pendant lequel le serveur peut conserver une requête avant d'y répondre : comme expliqué dans le paragraphe 5.4.1, cela affecte l'efficacité du mécanisme de sondage.

Il pourrait être intéressant de permettre la modification d'autres paramètres :

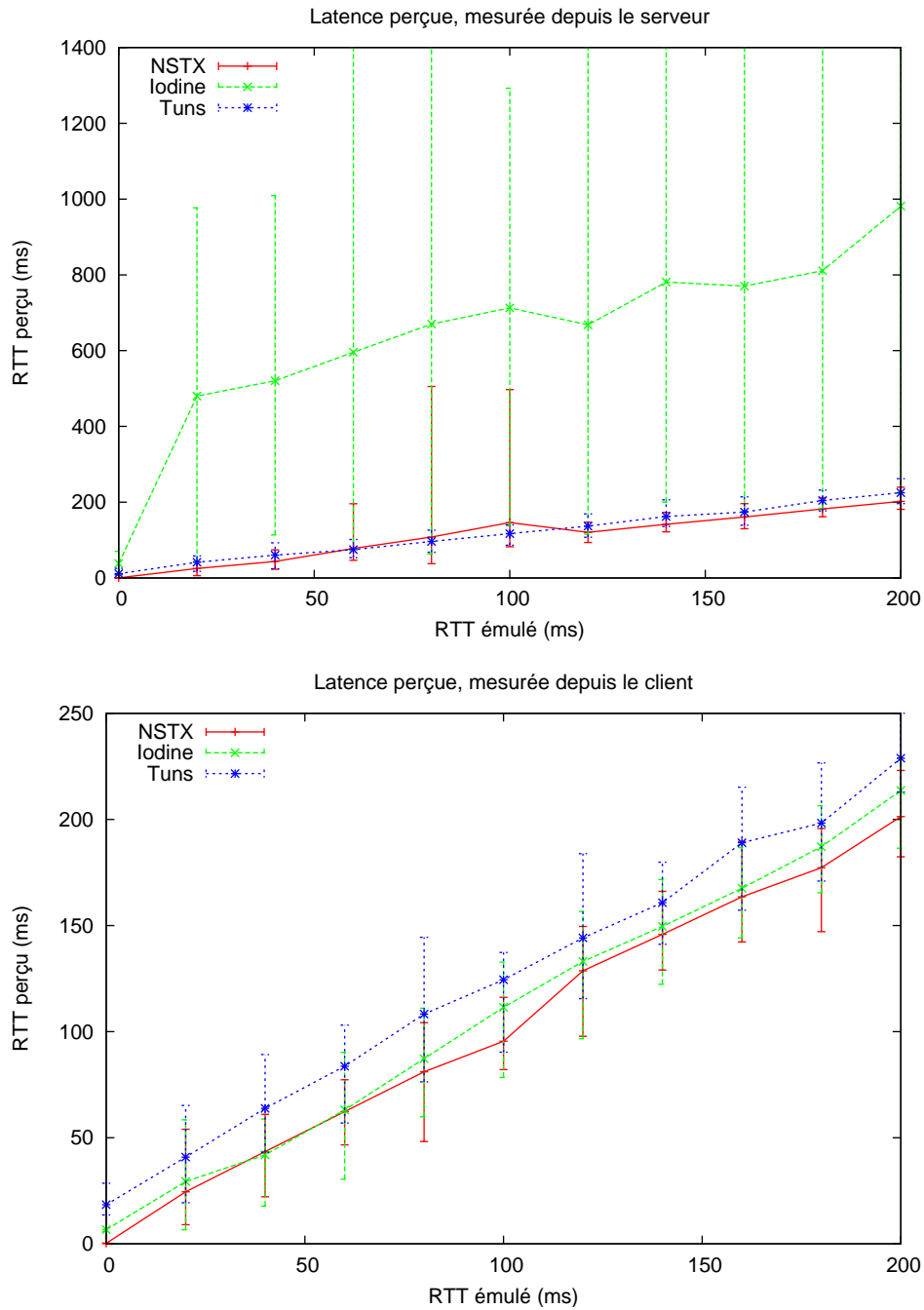


FIG. 5.8: Mesures de latence sur un réseau émulant pertes de paquets et réordonnancements.

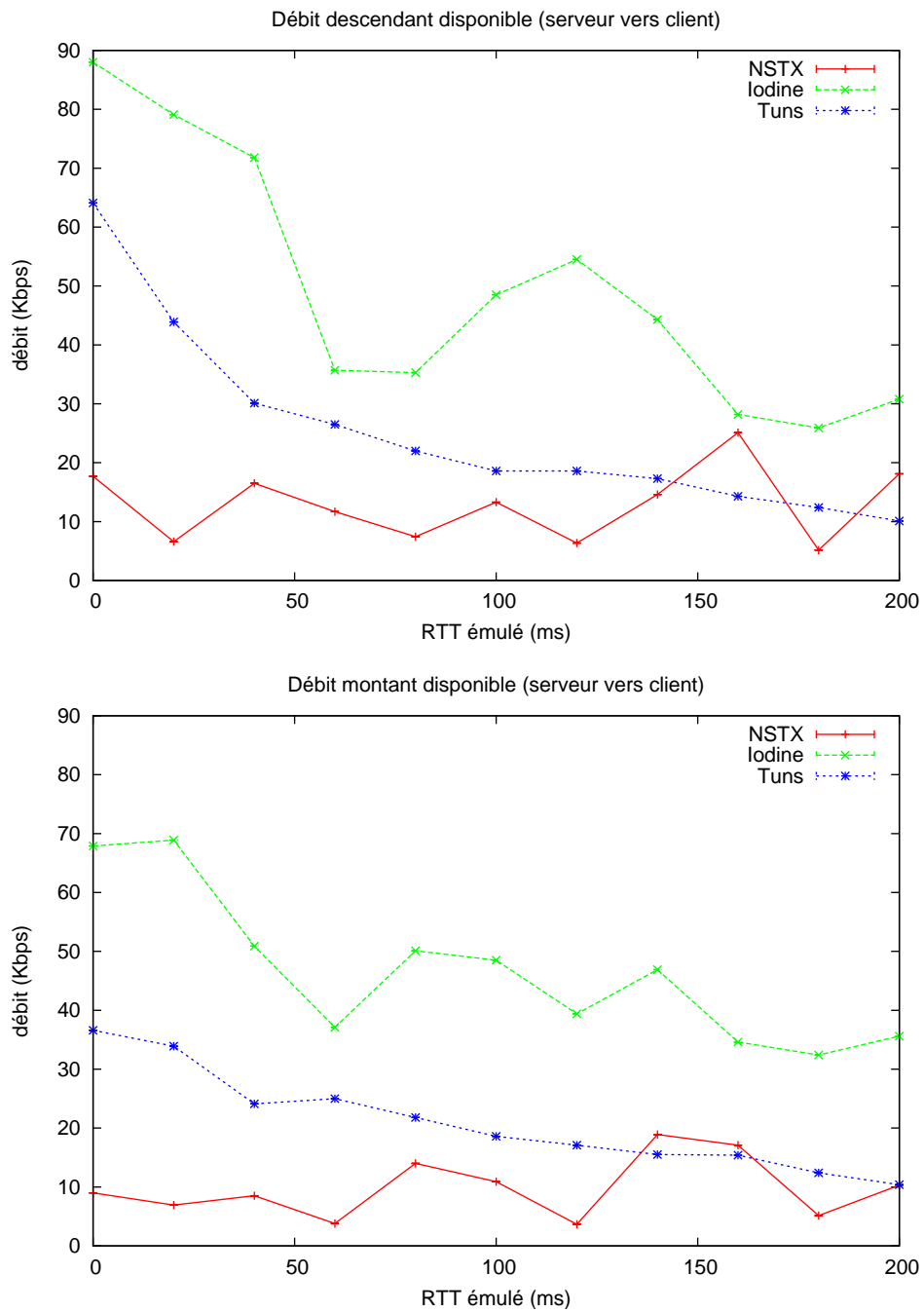


FIG. 5.9: Mesures de débit sur un réseau émulant pertes de paquets et réordonnements.

- L’encodage utilisé pour les requêtes et les réponses (permettre l’utilisation de Base64 au lieu de Base32 si le réseau le permet) ;
- Le type d’enregistrements utilisés (permettre l’utilisation d’enregistrements TXT ou NULL, permettant de stocker les informations d’une manière plus efficace) ;
- Utilisation de EDNS0.

5.6 Perspectives

En plus du changement, depuis le client, d’autres paramètres du tunnel, d’autres améliorations pourraient être apportées à TUNS.

D’abord, il serait intéressant que TUNS puisse s’adapter automatiquement aux caractéristiques d’un réseau : il pourrait déterminer quelles contre-mesures ont été mises en place sur le réseau, puis adapter ses paramètres (fréquence de sondage, encodage, etc) afin de maximiser ses performances.

D’autre part, afin d’augmenter la proportion de données *utiles* transmises dans chaque paquet DNS, des mécanismes comme la compression des en-têtes IP pourraient être utilisés. Mais afin que cela permette une réduction du nombre de paquets DNS, et pas seulement une réduction de la taille de chaque paquet DNS, ces mécanismes devraient être mis en place avant l’éventuelle fragmentation IP, dans le noyau. Une manière simple d’y parvenir serait d’utiliser le protocole PPP, au lieu de simplement encapsuler des paquets IP.

Une autre manière d’augmenter le débit est de changer l’encodage. Alors que la RFC originelle décrivant le protocole DNS n’autorise "_", ce caractère est utilisé par de nombreuses extensions de DNS (comme par exemple, les enregistrements DNS de type SRV). Lors de nos expériences, nous avons rencontré à la fois des réseaux où les requêtes contenant ce caractère étaient rejetées, et des réseaux où elles étaient traitées normalement. Utiliser Base64 avec ce caractère supplémentaire pourrait être une solution largement utilisable.

Une autre possibilité permettant d’utiliser un encodage Base64, serait de n’utiliser que 63 caractères, en utilisant un mécanisme d’échappement. Mais cela ferait varier la longueur des paquets. Lors des expériences où nous avons changé le MTU des paquets (qui augmente ou diminue la taille des paquets DNS), nous avons constaté que de nombreuses infrastructures DNS rejettent les paquets DNS de taille supérieure à la taille maximale autorisée par la RFC. Avec un mécanisme d’échappement, certains paquets DNS ne seraient alors pas transmis, provoquant le blocage de certaines communications. Une solution pourrait être de faire coexister plusieurs mécanismes d’échappement, et de choisir, pour chaque paquet, celui générant le paquet DNS le plus court.

5.7 Conclusion

Le nombre d'implémentations existantes le prouve aisément : l'idée d'encapsuler des données dans les paquets DNS n'est pas nouvelle. Toutefois, le nombre d'implémentations existantes prouve aussi qu'aucune d'entre elles n'apporte une solution définitive à ce problème, à cause des différents challenges proposés par l'implémentation d'une telle solution. Ce chapitre propose une exploration détaillée de ces challenges.

Plus spécifiquement, TUNS propose des réponses intéressantes à ces challenges. Il favorise des principes de conception simples, et reste dans les limites fixées par la spécification du protocole DNS. Cette approche s'est montrée bonne : TUNS est le seul tunnel à fonctionner sur tous les réseaux que nous avons testé, et fournit des performances raisonnables par rapport aux autres solutions, en particulier lorsque les conditions réseaux sont dégradées, ce qui est fréquent dans les conditions habituelles d'utilisation de ces tunnels.

Finalement, TUNS démontre qu'il est possible d'obtenir des performances raisonnables sans utiliser des fonctionnalités du protocole DNS très peu utilisées, ou non-standard. Du point de vue d'un administrateur réseau, il semble difficile de bloquer TUNS sans également bloquer du trafic légitime : la seule solution restante est de réduire la bande passante du canal caché jusqu'à le rendre quasiment inutile, en utilisant des solutions de modelage de trafic pour réduire le nombre de requêtes DNS pouvant transiter.

Enfin, l'intérêt d'utiliser un émulateur réseau est visible : réaliser ses expériences en utilisant des réseaux réels aurait pris un temps conséquent, à cause de la nécessité de trouver des réseaux correspondant aux différents paramètres que nous avons testé ici.

P2PLAB : UNE PLATE-FORME D'ÉMULATION POUR L'ÉTUDE DES SYSTÈMES PAIR-À-PAIR

6

6.1	Introduction	69
6.2	P2PLab : présentation générale	70
6.3	Virtualisation	71
6.3.1	Exécution concurrente d'un nombre important de processus par machine	73
6.3.2	Virtualisation de l'identité réseau des processus	77
6.4	Émulation de topologies réseaux	80
6.5	Conclusion	83

6.1 Introduction

Les systèmes pair-à-pair et les algorithmes utilisés par ces systèmes ont fait l'objet d'un intérêt considérable ces dernières années. Ils sont devenus plus performants, mais aussi plus complexes, ce qui a eu pour conséquence de les rendre plus difficiles à concevoir, à vérifier et à évaluer. Il devient nécessaire de pouvoir vérifier qu'une application se comportera correctement même lorsqu'elle sera exécutée sur des milliers de nœuds, ou de pouvoir comprendre des applications conçues pour fonctionner sur un tel nombre de nœuds.

Les applications distribuées sont traditionnellement étudiées en utilisant la modélisation (suivi d'une étude par une approche analytique "pure" ou par simulation) et l'exécution sur des systèmes réels. La modélisation et la simulation consistent à utiliser un modèle du fonctionnement de l'application dans un environnement synthétique (par opposition à un environnement réel). Cette méthode est largement utilisée, et permet d'obtenir des résultats de bonne qualité assez facilement. Toutefois, il est souvent nécessaire d'accepter un compromis entre le nombre de nœuds simulés et la précision du modèle : il est difficile de simuler efficacement un grand nombre de nœuds en utilisant un modèle complexe de l'application.

L'alternative consiste à exécuter l'application réelle à étudier dans un environnement d'expérimentation réel comme PlanetLab [29]. Mais ces plates-formes sont difficiles à contrôler et à modifier (afin de les adapter à des conditions d'expériences particulières), et les résultats sont souvent difficiles à reproduire, car

les conditions environnementales peuvent varier de manière importante entre les expériences. L'expérimentation sur ce type de plate-forme est nécessaire lors du développement d'un système pair-à-pair, mais ces expériences ne sont pas suffisantes, car il est très difficile de couvrir ainsi suffisamment de conditions d'expériences différentes pour pouvoir conclure dans le cas général. L'utilisation d'autres méthodes d'évaluation devient alors de plus en plus importante [36].

Entre ces deux approches, il est nécessaire d'explorer d'autres voies, permettant d'exécuter l'application réelle dans un environnement synthétique, sous des conditions reproductibles. Dans ce chapitre, nous décrivons P2PLab, une plate-forme pour l'étude des systèmes pair-à-pair combinant émulation et virtualisation.

Dans la suite du chapitre, nous distinguons émulation et virtualisation :

L'émulation consiste à exécuter l'application à étudier dans un environnement modifié afin de correspondre aux conditions expérimentales souhaitées. Il est nécessaire de déterminer quelles sont les ressources à émuler (et avec quelle précision) : il s'agit souvent d'un compromis entre réalisme et coût. Par exemple, lors de l'étude de systèmes pair-à-pair, l'émulation réseau est importante, alors que l'émulation précise des entrées/sorties sur le disque dur n'est probablement pas nécessaire. L'émulation est souvent coûteuse (temps processeur, mémoire), et son coût est souvent difficile à évaluer, car il dépend à la fois de la qualité de l'émulation et de ses paramètres : émuler un réseau à latence importante nécessitera une quantité de mémoire plus importante qu'émuler un réseau à faible latence, à cause de la nécessité de conserver les paquets dans une file d'attente (par exemple, émuler une latence de 1s sur un réseau avec un débit de 1 Gbps requiert une file d'attente occupant 125 Mo).

La virtualisation de ressources permet de partager une ressource réelle entre plusieurs instances d'une application. Elle est nécessaire afin de permettre l'étude d'un grand nombre de nœuds. Dans le cas des systèmes distribués, la virtualisation permet d'exécuter plusieurs instances de l'application ou du système d'exploitation sur chaque machine physique. Bien entendu, comme les ressources réelles de la machine physique sont partagées entre les différentes instances, l'équité est un problème important. Comme la précision de l'émulation, le niveau d'équité de la virtualisation est un compromis entre qualité et coût : un bon niveau d'équité s'obtient souvent au détriment des performances globales de la machine physique.

6.2 P2PLab : présentation générale

Nous cherchons à concevoir une plate-forme d'étude des systèmes pair-à-pair, combinant virtualisation et émulation, avec les caractéristiques suivantes :

- **Bon rapport de repliement** : un grand nombre de nœuds doivent pouvoir être étudiés sur un faible nombre de nœuds physiques. Ainsi, il sera possible d'utiliser une plate-forme de taille modeste pour réaliser des expériences à une échelle satisfaisante ;
- **Bon passage à l'échelle** : des expériences avec plusieurs milliers de nœuds doivent être possibles, pour pouvoir se placer dans des conditions suffisamment proches des conditions "cibles" d'utilisation des systèmes étudiés ;
- **Simplicité générale** : l'outil doit être suffisamment simple pour être facilement pris en main et compris, afin que les résultats d'expérience soient également simples à comprendre. Son déploiement doit être rapide, afin de permettre de réaliser des expériences en un temps raisonnable.

P2PLab apporte une réponse à ces objectifs, en faisant des choix différents de ceux des autres plate-formes largement utilisées, présentées au chapitre 2.

Tout d'abord, P2PLab virtualise au niveau des processus, pas au niveau du système d'exploitation comme les autres outils de virtualisation le font. En effet, une virtualisation complète du système n'est pas forcément nécessaire lorsque les objets étudiés sont des applications pair-à-pair, ces applications n'ayant en général que peu de dépendances sur le reste du système : au lieu de virtualiser un système complet, nous faisons le choix de demander à l'utilisateur de configurer l'application pour pouvoir en exécuter plusieurs instances sur le même système. Pour la plupart des applications que nous ciblons, cela se fait sans difficultés : il suffit par exemple d'exécuter les différentes instances dans des répertoires différents, ou dans le pire des cas, dans des comptes utilisateurs différents, et de les paramétrer pour utiliser des ports différents.

Ensuite, P2PLab utilise FreeBSD pour pouvoir utiliser DummyNet pour l'émulation réseau. Une approche décentralisée est utilisée pour émuler les topologies réseau, permettant un meilleur passage à l'échelle. La plupart des autres approches (Modelnet par exemple) dédient des nœuds à l'émulation, ce qui oblige à se poser la question du dimensionnement de la partie "émulation réseau" de la plate-forme, et rajoute une faible latence à cause des transmissions supplémentaires sur le réseau de la grappe.

Dans la suite de ce chapitre, nous détaillerons le système de virtualisation proposé par P2PLab, après avoir vérifié que les caractéristiques de FreeBSD correspondaient bien à celles attendues. Puis nous nous intéresserons à son modèle d'émulation réseau.

6.3 Virtualisation

Beaucoup de travaux ont été effectués récemment sur la virtualisation, avec différentes approches. Linux Vserver [78] est une modification du noyau Linux qui y ajoute des *contextes* et contrôle les interactions entre ces contextes, permettant à plusieurs environnements de partager le même noyau de manière invisible

pour les applications, avec un très faible surcoût. User Mode Linux [79] est une adaptation du noyau Linux vers un processus Linux. Enfin, Xen [57] et KVM [80] permettent l'exécution simultanée de plusieurs systèmes d'exploitation complets sur la même machine physique.

Ces travaux se distinguent par rapport aux différents ensembles logiciels ou matériels partagés ou non par les machines virtuelles : avec Linux Vserver, un seul noyau est exécuté, partagé par toutes les machines virtuelles. Avec les autres solutions, un noyau par machine virtuelle est exécuté. Dans tous les cas, le système de fichiers est dupliqué pour chaque machine virtuelle, même si certaines optimisations (*Copy On Write*) sont possibles pour éviter une multiplication de l'espace disque utilisé (*hard links* avec Linux Vserver - plusieurs fichiers pointant vers le même *inode*, ne multipliant donc pas l'espace disque utilisé, *snapshots* au niveau du *block device*, par exemple avec LVM sous Linux, avec toutes les solutions).

Cette multiplication de certains aspects du système pose un problème lors du passage à l'échelle : si on souhaite exécuter un nombre important de machines virtuelles par machine physique, il est important de minimiser le coût marginal d'une machine virtuelle. Par exemple, avec une solution à base de Xen (comme VDS [56] ou DieCast [59]), on va multiplier par le nombre de machines virtuelles :

- l'espace disque utilisé : un système complet doit être recopié pour chaque instance, mais des optimisations sont possibles pour éviter de dupliquer les fichiers identiques ;
- la mémoire vive : avec Xen, une quantité de mémoire vive fixe est affectée à chaque machine virtuelle. Or de nombreuses applications distribuées sont très gourmandes en mémoire, notamment lorsqu'elles nécessitent une machine virtuelle Java. De plus, réduire la quantité de mémoire affectée à une machine virtuelle peut influencer ses performances : le système d'exploitation de la machine virtuelle pourrait commencer à utiliser de la mémoire virtuelle (*swap*), ou, plus difficile à détecter, être affecté par le manque de mémoire qui l'empêcherait de l'utiliser comme cache pour améliorer les performances des entrées/sorties disque.

Dans P2PLab, nous faisons le choix de ne virtualiser que ce qui est strictement nécessaire lors d'expériences sur des systèmes pair-à-pair : l'identité réseau de chaque nœud virtuel. Chaque nœud virtuel s'exécute comme un processus UNIX classique, sur le même système d'exploitation que les autres nœuds virtuels de la même machine physique.

Cela pose plusieurs questions :

- Quelles sont les conséquences de l'exécution concurrente d'un nombre important de processus sur la même machine, en terme de surcoût et d'équité ?
- Comment réaliser la virtualisation de l'identité réseau, alors que nous n'avons pas de conteneur strict (comme un système d'exploitation différent) ?

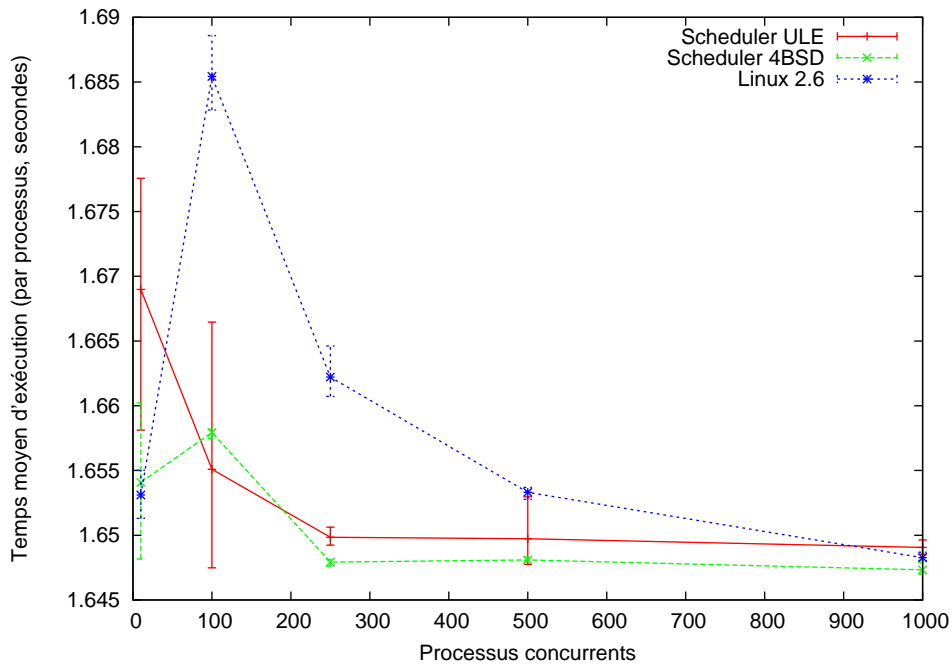


FIG. 6.1: Temps moyen d'exécution en fonction du nombre de processus concurrents. Les processus utilisent intensivement le processeur, mais n'utilisent que peu de mémoire

6.3.1 Exécution concurrente d'un nombre important de processus par machine

FreeBSD a été choisi pour P2PLab en raison de la disponibilité de Dummynet [40], l'émulateur réseau inclus dans FreeBSD. Mais il était nécessaire de commencer par vérifier que les caractéristiques de FreeBSD étaient suffisantes pour permettre l'exécution d'un grand nombre de processus sans fausser les résultats des expériences. FreeBSD 6 propose deux ordonnanceurs de processus différents :

- un ordonnanceur *historique* (appelé 4BSD) dérivé de celui de BSD 4.3 ;
- un ordonnanceur plus moderne, ULE [81]¹, développé pour permettre une meilleure interactivité, même sous une charge importante.

Comme le nombre de processus concurrents sera important, l'ordonnanceur jouera un rôle-clé, et il est important de comparer ces deux ordonnanceurs afin de sélectionner le plus adapté à l'exécution de nombreux processus concurrents.

Dans une première expérience, nous avons démarré un grand nombre de processus utilisant intensivement le processeur en même temps et examiné le temps nécessaire à la complétion de l'ensemble des tâches. Cette évaluation, ainsi que toutes les suivantes, a été réalisée sur la plate-forme GridExplorer, qui fait partie

¹Il s'agit d'un jeu de mots, l'option pour l'activer étant appelée SCHED_ULE.

du projet Grid'5000 [27]. Les machines utilisées sont des Bi-Opteron 2 Ghz avec 2 Go de mémoire vive et un réseau Gigabit Ethernet. Le programme utilisé réalisait un calcul utilisant peu de mémoire (calcul de la fonction de Ackermann) et prenant de l'ordre de 3,3 secondes.

Lors de cette expérience, nous avons rencontré un problème intéressant. Nous nous sommes aperçus que les systèmes utilisés (FreeBSD et Linux) ne fournissaient pas les mêmes performances, même sans exécuter plusieurs instances de manière concurrente. Nous avons finalement déterminé que les versions des compilateurs utilisés sur les deux systèmes n'étaient pas les mêmes, et que les deux compilateurs optimisaient différemment le code C que nous avons écrit. Pour contourner ce problème, nous avons généré le code assembleur sur un des systèmes, puis nous l'avons assemblé sur les deux systèmes. Cela montre l'importance d'effectuer des tests de "qualification" avant les expériences.

Comme les machines utilisées disposent de deux processeurs, les calculs se répartissent naturellement sur les deux processeurs, et le débit moyen de chaque système est donc de 1 tâche toutes les 1,65 secondes.

Sur la figure 6.1, on n'observe pas de surcoût lié au nombre de processus concurrents. Au contraire, le temps moyen d'exécution semble même baisser très légèrement lorsque le nombre de processus augmente (surtout sous Linux 2.6), probablement à cause d'effets de cache ou d'économies d'échelle (certaines parties du coût n'étant pas dépendantes du nombre de processus, leur part diminue lorsque le nombre de processus augmente).

Nous avons ensuite réalisé la même expérience, mais avec des processus utilisant la mémoire de manière intensive (réalisant des calculs simples sur des matrices de grande taille). Exécutés seuls, ils ont besoin d'environ 3 secondes de temps processeur avant de terminer. La figure 6.2 permet d'abord de constater que les résultats obtenus avec Linux et FreeBSD sont très différents. Sous Linux 2.6, l'ordonnanceur et/ou la gestion mémoire permettent d'éviter au temps d'exécution d'augmenter lorsque l'ensemble des instances ne peuvent plus être contenues dans la mémoire vive. Au contraire, sous FreeBSD, dès que la mémoire virtuelle (*swap*) est utilisée, le temps d'exécution augmente de manière importante.

Dans les expériences futures, il sera important de se placer dans des conditions où l'utilisation de *swap* n'est pas nécessaire. Nous avons donc fait le choix de ne pas activer le *swap* sur les machines utilisées. Ainsi, un manque de mémoire se traduira par une erreur franche, facilement détectable, plutôt que par une perte progressive de performances.

L'équité entre les processus est également importante : certaines instances ne doivent pas bénéficier plus souvent ou plus longtemps du processeur. L'expérience suivante nous permet d'avoir une première estimation du niveau d'équité

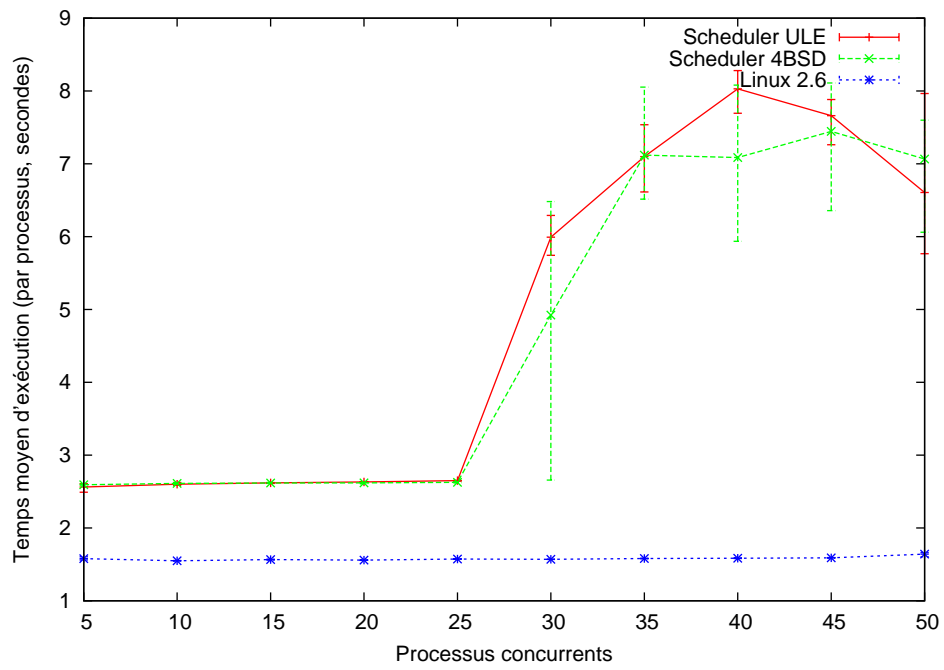


FIG. 6.2: Temps moyen d'exécution en fonction du nombre de processus concurrents. Les processus utilisent intensivement le processeur et la mémoire

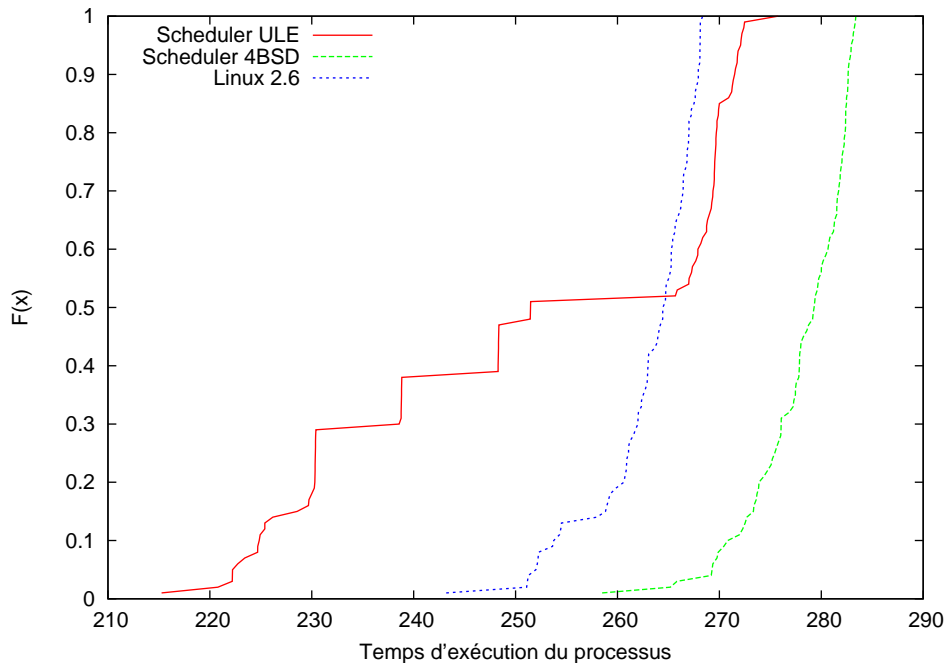


FIG. 6.3: Fonction de répartition des temps d'exécution des processus avec Linux 2.6 et les deux ordonnanceurs de FreeBSD. $F(x)$ indique la proportion des processus ayant terminé avant t

fourni par Linux et FreeBSD. Nous démarrons en « même temps »², sur la même machine, 100 instances d'un même programme utilisant intensivement le processeur, et mesurons les temps de terminaison de chaque instance. Le programme exécuté seul prend environ 5 secondes à terminer.

La figure 6.3 montre qu'avec l'ordonnanceur 4BSD et celui de Linux, la plupart des processus terminent *presque* en même temps. L'ordonnanceur ULE laisse par contre apparaître de plus grandes variations. Il faut noter que ces résultats sont différents de ceux qu'une précédente étude réalisée avec FreeBSD 5 [82] avait donnés : avec l'ordonnanceur ULE, certains processus étaient excessivement privilégiés et s'exécutaient seuls sur l'un des processeurs. Ce problème semble avoir été corrigé dans FreeBSD 6.

Même si cette approche fournit des résultats satisfaisants, tant sur le plan du passage à l'échelle que de l'équité, elle est limitée : elle ne permet pas, par exemple, de réaliser des expériences où des processeurs virtuels de vitesses différentes sont affectés aux processus. Cette approche n'est donc pas adaptée à une étude poussée des systèmes de type *Desktop Computing*. L'utilisation de solutions de virtualisation plus lourdes permettrait un contrôle plus précis du partage du processeur, et permettrait également de contrôler la mémoire.

Dans la suite des expériences, nous avons choisi d'utiliser l'ordonnanceur 4BSD, malgré cette limitation : il n'existe pas de solution sans cette limitation à l'heure actuelle sous FreeBSD.

6.3.2 Virtualisation de l'identité réseau des processus

Comme indiqué précédemment, P2PLab virtualise au niveau de l'identité réseau des processus : les instances exécutées sur la même machine physique partagent toutes les ressources (système de fichiers, mémoire, etc.) comme des processus normaux, mais chaque processus virtualisé a sa propre adresse IP sur le réseau. L'adresse IP de chaque machine physique est conservée à des fins d'administration, tandis que les adresses IPs des nœuds virtuels sont configurées comme des *alias d'interface*, comme montré sur la figure 6.4 (la plupart des systèmes Unix, dont Linux et FreeBSD, permettent d'affecter plusieurs adresses IP à une même interface réseau à travers un système d'alias). Nous avons vérifié que les alias d'interface ne provoquaient pas de surcoût par rapport à l'affectation normale d'une adresse IP à une interface.

Pour associer une application à une adresse IP particulière, nous avons choisi d'intercepter les appels systèmes liés au réseau (la figure 6.5 présente ces différents appels systèmes et leur ordre d'appel). Ainsi, avant chaque appel à `connect()`, un appel à `bind()` est fait pour que l'adresse d'origine de la connexion soit celle souhaitée. De même, les appels à `bind()` sont modifiés pour restreindre le

²Un processus, lancé avec une priorité élevée, lance les instances qui sont exécutées à une priorité plus faible. Les résultats ne montrent pas une influence significative de l'ordre de lancement.

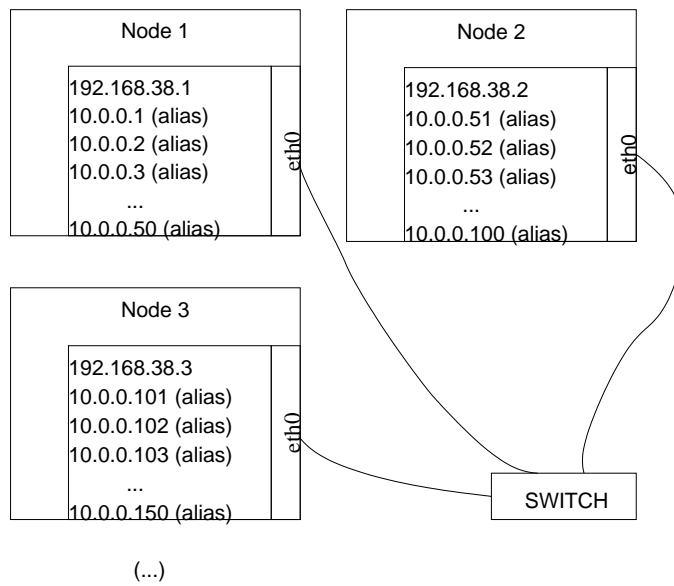


FIG. 6.4: Utilisation d’alias d’interface sur chaque machine physique, pour chaque IP de nœud virtuel

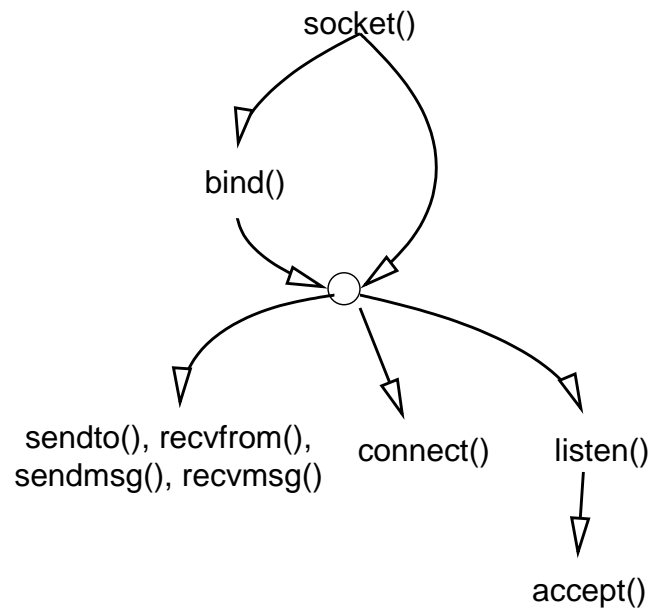


FIG. 6.5: Enchaînement des appels systèmes réseaux dans le cadre de connexions TCP

`bind()` à l'adresse souhaitée. Plusieurs solutions étaient possibles pour réaliser techniquement ces restrictions :

- modifier l'application étudiée, par exemple en faisant explicitement un `bind()` avant chaque appel à `connect()`. Cette méthode a le désavantage de nécessiter une compréhension du code de l'application ;
- lier l'application avec une bibliothèque réalisant la surcharge des appels système, soit à la compilation, soit en utilisant la variable d'environnement `LD_PRELOAD`.
- modifier le noyau pour modifier le traitement des appels systèmes. Sous Linux, c'est la méthode choisie par VServer [78]. Mais sa mise en œuvre est compliquée ;
- utiliser `ptrace()` pour intercepter les appels systèmes réseaux. Sous Linux, c'est la méthode choisie par User Mode Linux [79]. Cette méthode a un surcoût relativement important à cause des changements de contexte supplémentaires nécessaires ;
- modifier la bibliothèque C. Cette approche permet de se placer « au-dessous » des bibliothèques éventuelles, mais ne fonctionne pas avec les exécutables compilés statiquement.

La solution de lier l'application avec une bibliothèque réalisant la surcharge des appels système est très populaire : elle est par exemple utilisée par Modelnet et Trickle. Toutefois, cette technique a plusieurs inconvénients :

- Pour des raisons de sécurité, elle ne permet pas d'intercepter les appels de fonction faits par des applications `setuid` ou `setgid`, à cause du risque d'élévation de privilège ;
- Elle ne permet pas toujours d'intercepter les appels faits via une bibliothèque intermédiaire ;
- Elle ne permet pas d'intercepter les appels aux fonctions d'une bibliothèque qui peuvent également être appelées via une autre fonction de cette bibliothèque. Par exemple, si l'objectif est d'intercepter les appels à `write()`, les appels à `write()` faits via un appel à `printf()` ne seront pas interceptés ;
- Elle ne permet pas d'intercepter les appels faits par des binaires compilés statiquement.

Pour ces raisons, nous avons choisi, dans P2PLab, de modifier la bibliothèque C de FreeBSD. Cette technique ne souffre que du dernier des désavantages listés ci-dessus (impossibilité d'intercepter les appels d'un binaire compilé statiquement). Nous avons modifié les fonctions `bind()`, `connect()` et `listen()` pour toujours se restreindre à l'adresse IP spécifiée dans la variable d'environnement `BINDIP`.

Pour mesurer le surcoût induit par cette technique, nous avons réalisé un programme de test très simple : un client TCP qui se connecte sur un serveur tournant sur la même machine. Dès que la connexion a été acceptée par le serveur, le serveur coupe la connexion. Puis le cycle recommence : le client se reconnecte.

En exécutant ce test un nombre important de fois, nous avons pu mesurer avec un intervalle de confiance satisfaisant que la durée d'un cycle était de $10,22 \mu s$ sans la modification de la `libc`, et de $10,79 \mu s$ avec la modification de la `libc`. Il faut noter que ce surcoût n'est présent que lors de l'établissement de la connexion. Notre test montre que, même dans un cas limite (application passant tout son temps à établir des connexions), ce surcoût reste très limité comparé au temps nécessaire pour établir une connexion.

Notre méthode de surcharge ne fonctionne que pour les connexions TCP. Une approche identique permettrait dans le futur de traiter également les communications UDP, mais il faudrait alors surcharger les appels `send()` et `receive()`, ce qui provoquerait un surcoût plus élevé : les appels de bibliothèque seraient alors surchargés à chaque envoi de datagramme, et non plus uniquement lors de l'établissement de la connexion.

6.4 Émulation de topologies réseaux

Les émulateurs de topologies réseau existants comme Modelnet [49] visent une émulation réaliste du cœur du réseau (routage et interactions entre les systèmes autonomes (AS) ou entre les principaux routeurs). Mais la plupart des applications pair-à-pair sont exécutées sur des nœuds situés *en bordure d'internet* [83], par exemple sur des ordinateurs personnels d'abonnés à l'ADSL. Même si le trafic dans le cœur du réseau peut influencer certains aspects du comportement des systèmes pair-à-pair (la congestion dans le cœur du réseau peut influencer la latence, par exemple), le principal goulot d'étranglement reste le lien entre le système participant et son fournisseur d'accès à internet. Dans le cadre d'une plate-forme d'expérimentation comme P2PLab, on peut donc utiliser un modèle réseau simplifié, et faire abstraction des problèmes de congestion dans le cœur du réseau, au moins dans un premier temps. Dans tous les cas, des évaluations sur un outil comme Modelnet ou P2PLab ne peuvent pas être considérées comme suffisantes, et doivent être complétées par des évaluations sur des systèmes réels comme PlanetLab [29] ou DSLLab [38], qui permettent de prendre en compte les problèmes de congestion dans le cœur du réseau, ou par des simulations.

Dans P2PLab, nous modélisons donc le réseau internet en mettant l'accent sur le point de vue du nœud participant, en excluant ce qui est moins important de ce point de vue là.

L'émulation réseau est réalisée d'une manière décentralisée : chaque nœud physique se charge de l'émulation réseau pour les nœuds virtuels qu'il héberge, à l'aide de Dummynet [40] : il y a ainsi deux règles dans le pare-feu pour chaque nœud virtuel hébergé, l'une pour les paquets entrants à destination de ce nœud virtuel, l'autre pour les paquets sortant en provenance de ce nœud virtuel.

Mais ce modèle permet uniquement d'avoir des paramètres de bande passante et de latence pour chaque nœud virtuel. Il ne permet pas d'exprimer la

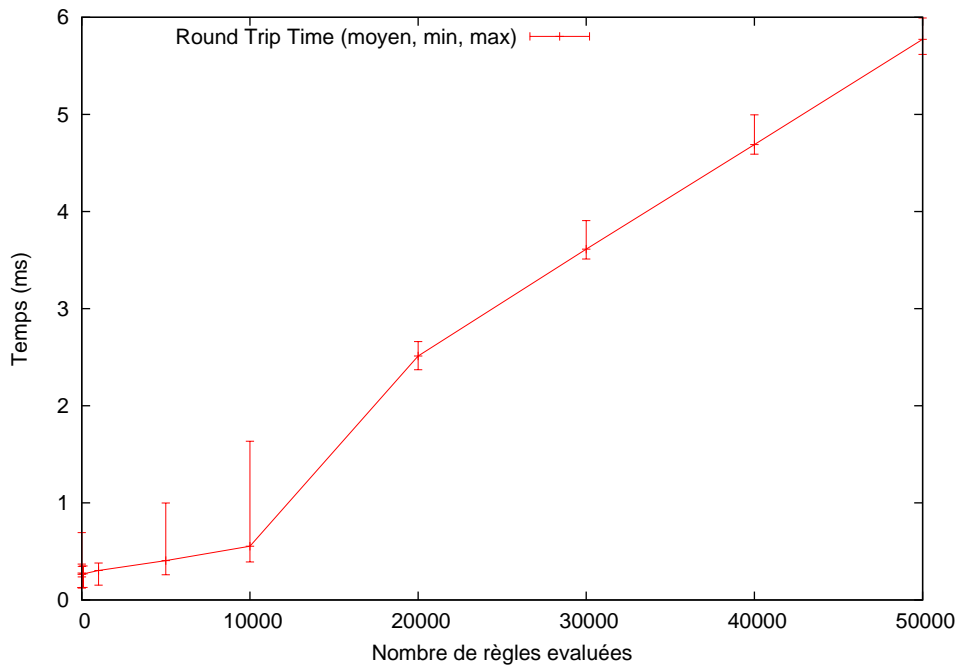


FIG. 6.6: Evolution du *round-trip time* en fonction du nombre de règles de pare-feu à évaluer

proximité entre deux nœuds. Nous rajoutons donc une notion de groupe de nœud, permettant d'évaluer des applications utilisant la localité. Dans un système réel, ces groupes de nœuds pourraient correspondre à un ensemble de nœuds provenant du même fournisseur d'accès, du même pays, ou du même continent.

Le modèle d'émulation de P2PLab permet donc de contrôler :

- la bande passante, la latence et le taux de perte de paquets sur les liens réseaux entre les nœuds et leur fournisseur d'accès à internet ;
- la latence entre des groupes de nœuds, permettant d'étudier des problèmes mettant en jeu la localité des nœuds, par exemple.

Le nombre de règles nécessaires est le principal paramètre limitant le passage à l'échelle de P2PLab. Pour montrer l'importance de ce facteur, nous mesurons le *round-trip time* (temps aller/retour) à l'aide de `ping` entre deux nœuds. Lors de la sortie du premier nœud, le pare-feu doit, pour chaque paquet, évaluer un nombre important de règles. La figure 6.6 permet d'observer que la latence commence par croître assez lentement, elle croît ensuite linéairement, avec un surcoût non négligeable quand le nombre de règles est important.

Ce surcoût provient du fait que les règles sont évaluées linéairement : avec DummyNet, il n'est pas possible d'évaluer les règles d'une manière hiérarchique, ou à l'aide d'une table de hachage. Une solution possible aurait pu être de mo-

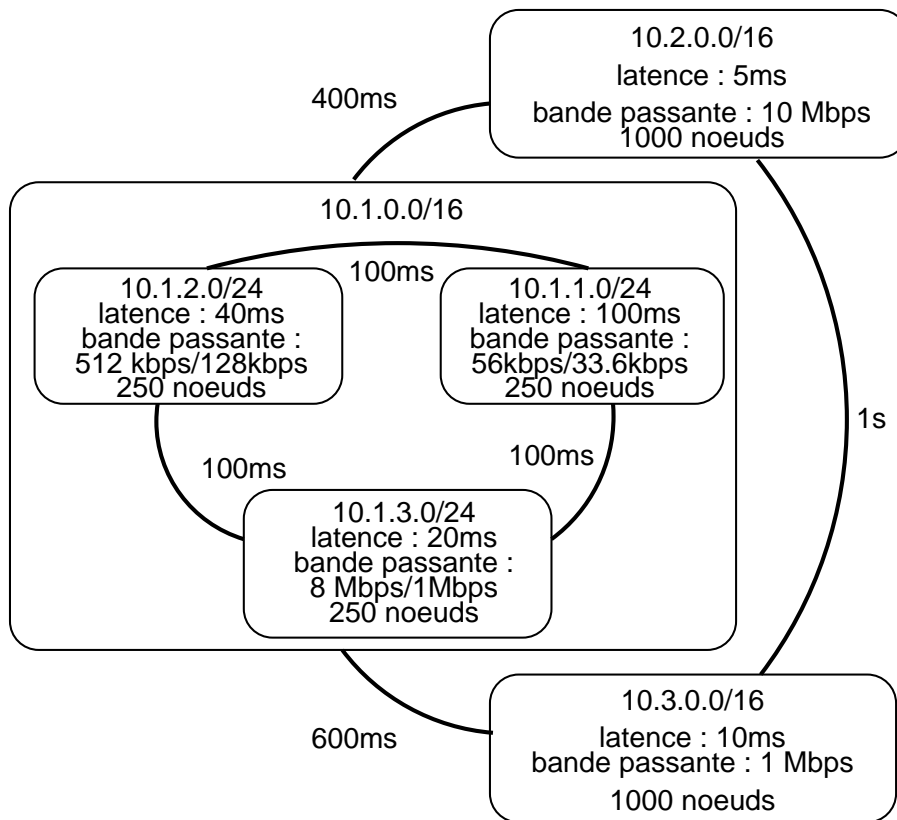


FIG. 6.7: Exemple de topologie émulée par P2PLab

difier le pare-feu de FreeBSD pour y rajouter ce type d'évaluations, mais dans le cadre de P2PLab, nous avons préféré utiliser la notion de groupes de nœuds pour limiter le nombre de règles (une seule règle par groupe permettant alors de traiter la latence pour l'ensemble des nœuds composant le groupe).

La figure 6.7 montre un exemple de topologie que nous avons émulée avec P2PLab. Le nœud physique hébergeant le nœud virtuel 10.1.3.207 aura par exemple :

- deux règles par nœud virtuel hébergé (paquets entrants et sortants, respectivement) ;
- une règle pour traiter l'ajout de 100 ms de latence pour les paquets provenant du groupe 10.1.3.0/24 à destination du groupe 10.1.1.0/24 (la règle réciproque étant présente sur les nœuds hébergeant le groupe 10.1.1.0/24) ;
- une règle pour traiter l'ajout de 100 ms de latence pour les paquets provenant du groupe 10.1.3.0/24 à destination du groupe 10.1.2.0/24 ;
- une règle pour traiter l'ajout de 400 ms de latence pour les paquets provenant du groupe 10.1.0.0/16 à destination du groupe 10.2.0.0/16 ;
- une règle pour traiter l'ajout de 600 ms de latence pour les paquets provenant du groupe 10.1.0.0/16 à destination du groupe 10.3.0.0/16.

Nous avons mesuré la latence entre les nœuds 10.1.3.207 et 10.2.2.117 (il n'y

avait pas d'autre trafic entre les nœuds). La latence mesurée de 853 ms se décompose en :

- 20 ms de délai lorsque le paquet est parti de 10.1.3.207 (délai pour le groupe 10.1.3.0/24) ;
- 400 ms de délai entre le groupe 10.1.0.0/16 et le groupe 10.2.0.0/16 ;
- 5 ms lorsque le paquet est arrivé sur 10.2.2.117 (délai pour le groupe 10.2.0.0/16) ;
- 425 ms lorsque le paquet est revenu de 10.2.2.117 à 10.1.3.207, comme ci-avant ;
- 3 ms de surcoût, dûs à la latence lors du transit sur le réseau de la grappe, et au parcours des règles du pare-feu.

6.5 Conclusion

Dans ce chapitre, nous avons présenté P2PLab. Cette plateforme d'émulation se distingue des autres plateformes par plusieurs aspects :

- Elle prône une solution de virtualisation légère, au niveau processus, permettant de passer facilement à l'échelle en limitant le surcoût de chaque instance ;
- Elle propose une solution d'émulation de topologies réseau décentralisée, là aussi pour permettre un bon passage à l'échelle.

Dans le chapitre suivant, nous nous attacherons à vérifier que P2PLab répond bien aux propriétés souhaitées, et présenterons son utilisation afin de tester deux implantations d'un protocole pair-à-pair largement utilisé.

P2PLAB : VALIDATION ET EXPÉRIMENTATIONS 7

7.1 Valider une plate-forme d'émulation	85
7.2 Validation de P2PLab à l'aide d'expériences sur BitTorrent	86
7.2.1 Rapport de virtualisation	87
7.2.2 Passage à l'échelle	89
7.3 Comparaison de différentes implantations de BitTorrent	91
7.4 Conclusion	92

7.1 Valider une plate-forme d'émulation

Au chapitre précédent, nous avons présenté P2PLab, notre proposition de plate-forme d'émulation pour l'étude des systèmes pair-à-pair. Dans ce chapitre, nous nous attachons à vérifier son bon comportement. L'émulation réseau de P2PLab utilise Dummynet, dont nous avons étudié le fonctionnement au chapitre 3. Il nous reste donc ici à vérifier que les autres aspects de P2PLab, notamment la virtualisation, n'induisent pas de résultats erronés ou biaisés.

Vérifier qu'un outil combinant émulation et virtualisation se comporte correctement est difficile. Ces deux approches impliquent obligatoirement des compromis entre réalisme et coût, et chercher un réalisme absolu rendrait l'outil moins utilisable, et donc moins utile : faire un outil au réalisme quasi-parfait est toujours possible, mais dans la plupart des cas, il sera moins intéressant pour l'expérimentateur qu'un outil réalisant un compromis plus intéressant.

Un autre aspect important est le comportement de l'outil lorsqu'il est poussé dans ses limites, c'est-à-dire quand la somme des ressources demandées par les noeuds virtuels d'une machine physique excède les ressources de cette machine physique. Il n'est pas souhaitable que l'outil arrête forcément l'expérience dans ce cas là, car il est possible que ce cas ne dure qu'une période courte pendant l'expérience, par exemple au démarrage de l'expérience. Mais l'outil doit fournir des indications à l'expérimentateurs (via des sondes systèmes et réseaux) pour qu'il puisse prendre une décision sur la validité des résultats obtenus. Si ces indications ne sont pas suffisantes, une autre solution est de répéter la même expérience en faisant varier les paramètres de virtualisation, afin de vérifier que les conditions limites n'ont pas été atteintes : si on constate une différence de résultat significative entre un rapport de 50 noeuds virtuels par noeud physique, et 40 noeuds virtuels par noeud physique, cela signifie qu'il y a un surcoût lié au

rapport de virtualisation.

Plusieurs approches sont possibles pour valider une plate-forme d'émulation. Au chapitre 3, nous avons utilisé des *micro-benchmarks* pour valider et comparer les différentes solutions d'émulation réseau. Nous en avons aussi utilisé au chapitre précédent, pour valider partiellement notre solution de virtualisation, notamment sur l'équité entre les processus. Mais dans la validation globale de notre solution, un *micro-benchmark* est moins intéressant : un micro-benchmark va souligner que le compromis choisi provoque une baisse de qualité des résultats par rapport à une approche sans émulation, mais ne permettra pas d'évaluer l'importance de cette régression.

Une autre approche serait de comparer notre solution avec d'autres solutions concurrentes. Cela est toutefois difficile pour l'instant, car les solutions diffèrent sur de nombreux aspects : objectifs, type de topologie émulée, plateforme d'exécution utilisée, disponibilité du code, etc. Dans les résultats, il serait difficile de distinguer ce qui est causé par des différences entre les émulateurs, et ce qui est causé par des différences dans la manière dont les expériences ont été conduites.

L'approche que nous avons choisie ici est de valider P2PLab en réalisant une même expérience avec des paramètres de virtualisation différents : on vérifie que le système étudié se comporte de la même manière quand on exécute une seule application par noeud, ou quand on utilise la virtualisation pour exécuter plusieurs applications sur le même noeud.

7.2 Validation de P2PLab à l'aide d'expériences sur BitTorrent

BitTorrent [84] est un système pair-à-pair de distribution de fichiers très populaire. Il fournit de très bonnes performances grâce à un mécanisme de réciprocité complexe, permettant de s'assurer que les nœuds récupérant un fichier coopèrent en envoyant les morceaux déjà récupérés vers d'autres nœuds. Lors d'un transfert avec BitTorrent, les clients contactent un *tracker* pour récupérer une liste d'autres nœuds participant au téléchargement de ce fichier, puis se connectent directement aux autres nœuds. Les nœuds disposant du fichier complet sont appelés *seeders*.

BitTorrent a déjà été étudié par l'analyse de traces d'une utilisation à grande échelle [85, 86], ainsi qu'à l'aide de modélisation [87] et de simulation [17]. Cependant, ces travaux ont rarement été comparés à des évaluations à grande échelle sur des systèmes réels, ou à des études utilisant l'émulation. BitTorrent est avant tout un travail d'ingénierie, et non un prototype de recherche : plusieurs parties de son code sont très complexes, et le nombre élevé de constantes et de paramètres utilisés dans les algorithmes le rendent très difficile à modéliser avec précision.

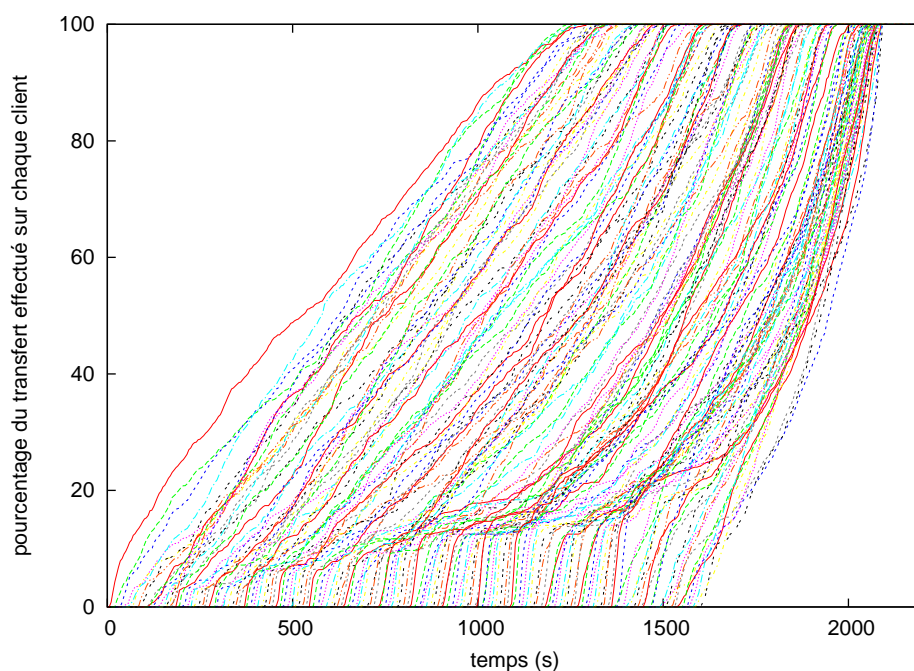


FIG. 7.1: Evolution du téléchargement des 160 clients

Pour cette expérience et toutes celles qui suivent, nous avons utilisé la plateforme GridExplorer, qui fait partie du projet Grid'5000 [27]. Les machines utilisées sont des Bi-Opteron 2 Ghz avec 2 Go de mémoire vive et un réseau Gigabit Ethernet. Les clients BitTorrent utilisés sont BitTorrent 4.4.0 (écrit en Python par l'auteur originel de BitTorrent) et une version modifiée de CTorrent 3.1.4¹ (écrite en C++).

7.2.1 Rapport de virtualisation

Dans une première expérience, nous comparons le téléchargement d'un fichier de 16 Mo entre 160 clients. La taille du fichier n'est pas importante dans le cas de BitTorrent, puisque le fichier est toujours découpé en blocs de 256 Ko. Le fichier est fourni par 4 *seeders*. Tous les nœuds (à la fois les téléchargeurs et les *seeders*) disposent des mêmes conditions réseaux :

- bande passante descendante de 2 mbps ;
- bande passante ascendante de 128 kbps ;
- latence (en entrée et en sortie des nœuds) de 30 ms.

Ces conditions sont similaires à celles rencontrées avec des connexions ADSL. Tous les clients ont des caractéristiques identiques : nous avons préféré utiliser ces conditions non réalistes, plutôt que de chercher à utiliser des conditions

¹Disponible sur <http://www.rahul.net/dholmes/ctorrent/>.

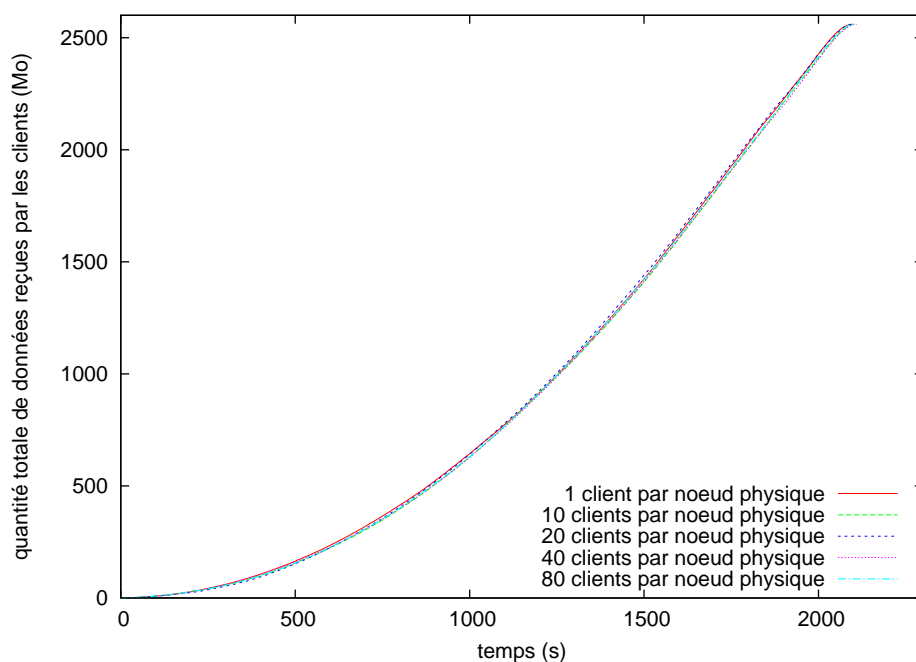


FIG. 7.2: Rapport de virtualisation de P2PLab : quantité totale de données récupérées par les 160 clients avec les différents déploiements

d'expériences supposées réalistes, sans pouvoir le justifier. Quand les clients terminent leur téléchargement, ils restent connectés et transmettent des données aux autres téléchargeurs.

Les 160 clients sont déployés successivement sur 160 nœuds physiques (soit l'ensemble des nœuds disponibles de GridExplorer lorsque cette expérience a été faite), puis 16 nœuds physiques (10 nœuds virtuels par nœud physique), puis 8, puis 4, et 2 nœuds physiques (avec 80 nœuds virtuels par nœud physique).

Les clients sont démarrés toutes les 10 secondes. La figure 7.1 montre l'évolution du téléchargement sur chacun des 160 clients quand ils sont déployés sur 160 nœuds physiques. On peut constater qu'avec ces conditions expérimentales, toutes les phases d'un téléchargement avec BitTorrent sont représentées :

- d'abord, il y a une période de quelques secondes pendant laquelle seuls les *seeders* sont capables de transmettre des données ;
- puis les téléchargeurs commencent à s'échanger des données ;
- enfin, après environ 1 300 secondes, les premiers téléchargeurs deviennent *seeders* à leur tour, et aident les autres téléchargeurs à finir leur téléchargement.

Ces paramètres sont donc suffisamment réalistes pour vérifier les capacités de virtualisation de P2PLab.

La figure 7.2 montre la progression du transfert avec les différentes répartitions (de 1 à 80 noeuds virtuels par noeud physique). Elle montre que l'expérience s'est réalisée sans surcoût quel que soit le rapport de virtualisation utilisé. Même avec 80 noeuds virtuels par noeud physique, les résultats sont presque identiques (ce qui est amplifié par le fait que nous mesurons la quantité totale de données reçues, ce qui masque les différences éventuelles entre les clients). Les sources potentielles de surcoût ont été recherchées, et il a été déterminé que le premier facteur limitant était la vitesse du réseau : avec des paramètres légèrement différents pour le réseau émulé, le réseau Gigabit de la plate-forme était saturé par les téléchargements.

7.2.2 Passage à l'échelle

Dans cette expérience, nous cherchons à montrer qu'il est possible de réaliser des expériences avec un nombre important de noeuds. Nous transférons un fichier de 16 Mo en utilisant 13 040 noeuds répartis sur 163 machines physiques (80 noeuds virtuels par machine physique). Parmi ces 13 040 noeuds, il y un *tracker*, et 8 noeuds (*seeders*) qui disposent du fichier complet dès le début de l'expérience afin de le transférer vers les autres noeuds. Le *tracker* et les *seeders* sont démarrés au début de l'expérience, tandis que les clients sont démarrés toutes les 0,25 secondes.

Puisque P2PLab permet d'exécuter une application non modifiée, nous choisissons aussi d'utiliser deux implémentations très différentes de BitTorrent : les clients (*seeders* compris) à numéro pair utilisent CTorrent, tandis que les clients à numéro impair utilisent BitTorrent 4.4.0. Lorsqu'un client termine le transfert, il se transforme en *seeder* et reste présent afin d'aider les autres clients à terminer leur transfert.

La figure 7.3 montre l'avancement du transfert sur les clients numérotés 101, 200, 301, 400, 501 ... On constate que, malgré des dates de démarrage décalées, la plupart des clients terminent leur transfert à des dates proches, ce qui est confirmé par la figure 7.4. Cela est dû au mécanisme de réciprocité de BitTorrent : les clients préfèrent aider les clients ayant reçu peu de données que des clients proches de la fin de leur transfert, car ces derniers, une fois le transfert terminé, pourraient quitter le système et ne plus contribuer aux transferts des autres clients.

P2PLab a permis de réaliser cette expérience assez facilement. Toutefois, le nombre important de noeuds virtuels et la fréquence élevée de leurs lancements a causé des problèmes avec les serveurs SSH des noeuds (le serveur SSH n'accepte qu'un nombre faible de connexions en attente, et rejette les connexions en surplus, ce qui pose problème lorsque le rapport de virtualisation est important et que la fréquence de connexion est importante). L'utilisation d'outils spécifiques plus adaptés au contrôle de ce type d'expériences (lanceurs parallèles, outils de contrôle d'expérience comme Expo [88]) permettrait d'éviter ces problèmes.

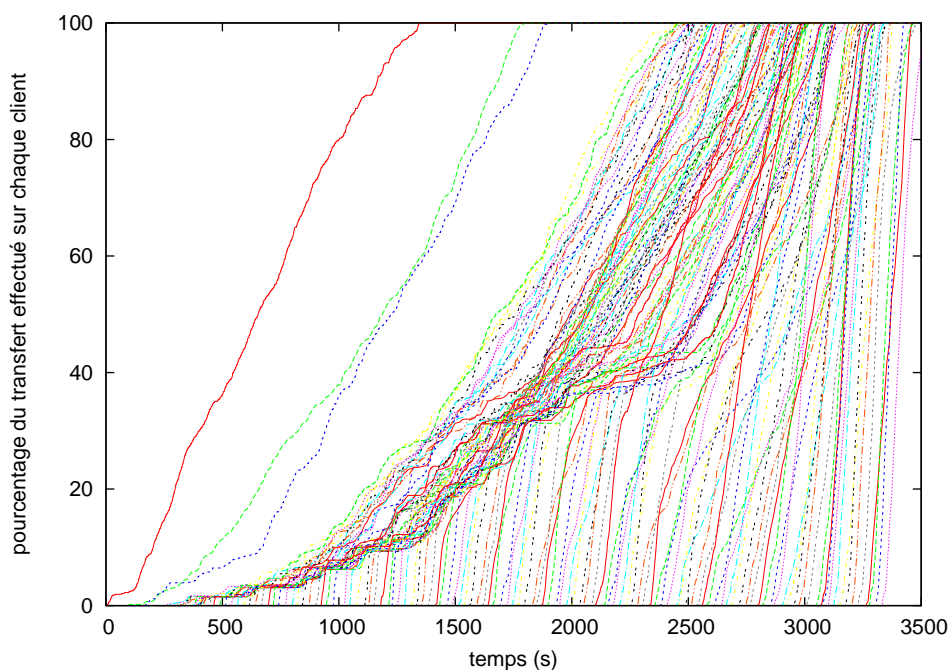


FIG. 7.3: Evolution du transfert d'un fichier de 16 Mo entre 13 032 clients sur quelques nœuds sélectionnés (nœuds 101, 200, 301, 400 ...)

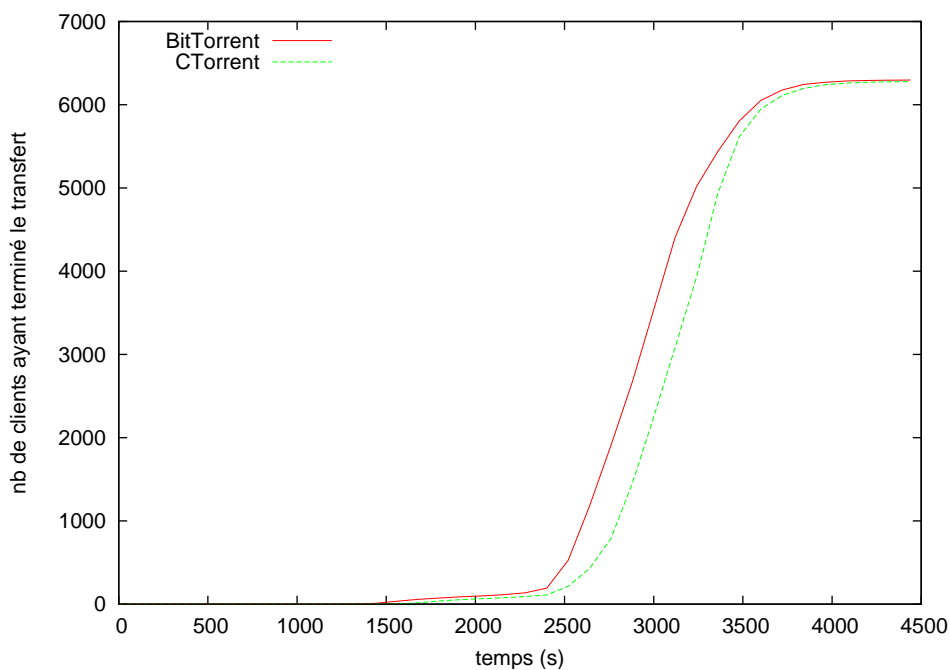


FIG. 7.4: Fonction de répartition du temps de complétion du transfert sur les différents nœuds

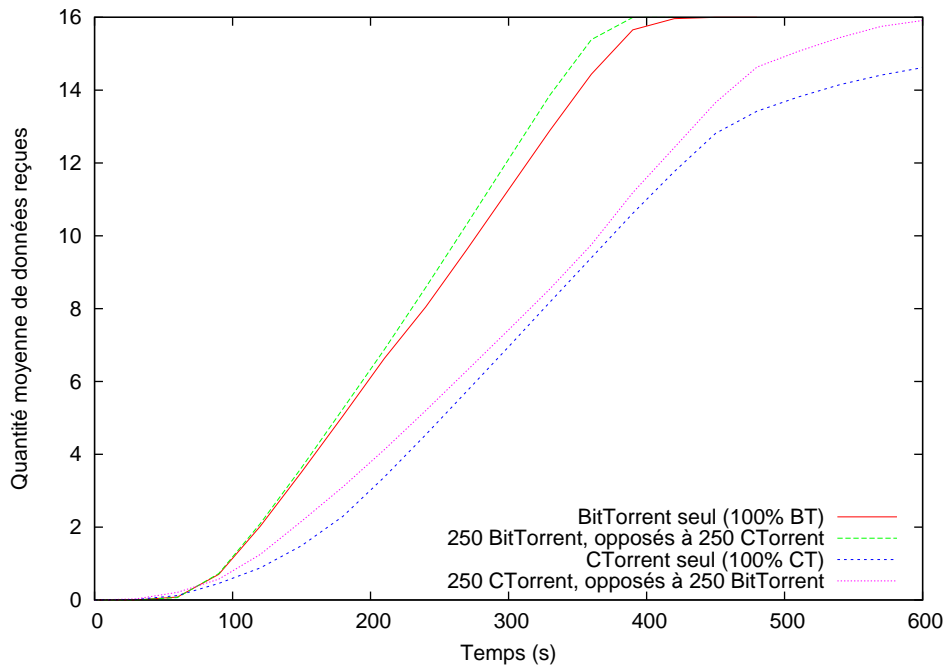


FIG. 7.5: Comparaison de BitTorrent et CTorrent, exécutés soit seuls, soit avec des clients de l'autre type comme "adversaires"

7.3 Comparaison de différentes implantations de BitTorrent

Puisqu'il permet d'exécuter directement des applications réelles, P2PLab est adapté à la comparaison d'implantations différentes du même protocole, en garantissant que les conditions environnementales (réseau) seront identiques.

Nous utilisons donc pour comparer BitTorrent, le client de référence développé par l'auteur du protocole [84], et CTorrent, qui a été utilisé dans des travaux visant à montrer certaines limites du protocole [89]. Nous réalisons le transfert d'un fichier de 16 Mo entre 500 clients et 8 *seeders* (4 utilisent BitTorrent, 4 CTorrent) en nous plaçant dans les mêmes conditions réseau que précédemment, et lançons tous les clients en même temps, au début de l'expérience.

Pour vérifier qu'un client BitTorrent est performant, il est important de vérifier qu'il est performant à la fois lorsqu'il ne communique qu'avec des clients identiques, et lorsqu'il communique avec des clients différents. Nous comparons donc les performances de BitTorrent et CTorrent lorsqu'ils sont lancés seuls (500 clients du même type) ou en mélangeant des clients BitTorrent et CTorrent (250 clients de chaque type). Les résultats sont présentés en figures 7.4 et 7.5, et montrent que BitTorrent 4.4.0 semble plus performant dans les deux cas avec ces paramètres d'expérience.

Si l'objectif était réellement de comparer de manière détaillée ces deux implantations de BitTorrent (et pas seulement de montrer la faisabilité de ce type d'expériences avec P2PLab), il faudrait compléter cette expérience par d'autres expériences avec des paramètres différents : il est par exemple possible que CTorrent soit plus performant que BitTorrent sous certaines conditions réseau, et notre outil pourrait permettre facilement de comparer ces deux implantations sous un large spectre de conditions réseau.

7.4 Conclusion

Dans ce chapitre, nous avons montré que P2PLab permettait bien de réaliser des expériences sur des systèmes pair-à-pair réels et largement utilisés comme BitTorrent. De plus, P2PLab répond bien aux caractéristiques souhaitées :

Validation : nous avons cherché à valider l'ensemble des briques de base utilisées, notamment pour l'émulation et la virtualisation, avant de vérifier leur intégration par des expériences.

Rapport de virtualisation : nous avons exécuté jusqu'à 80 machines virtuelles par machine physique sans constater de biais dans les résultats.

Passage à l'échelle : nous avons pu réaliser une expérience avec plus de 13000 noeuds, et la seule limite apparue au cours de cette expérience est la taille du cluster.

CONCLUSION ET PERSPECTIVES 8

8.1	Contexte d'étude et difficultés	93
8.2	Contributions	94
8.3	Perspectives	95

8.1 Contexte d'étude et difficultés

Les outils d'évaluation, de test, de validation et de débogage jouent un rôle important depuis les débuts de l'informatique [90]. Les outils spécifiques aux systèmes distribués s'inscrivent logiquement dans cette tradition, et prendront probablement une place de plus en plus importante dans les années à venir. En effet, si certains contestent l'apport d'un débogueur par rapport à un débogage à l'aide de `printf`, l'utilisation de solutions spécifiques semble inévitable pour les applications distribuées, même si certaines de ces solutions ne seront évidemment pas aussi complexes que celles décrites au chapitre 2 : on constate déjà que de certains développeurs créent des infrastructures de test ad-hoc (par exemple, le développeur originel de BitTorrent a mentionné utiliser une douzaine de machines hébergées chez des amis pour évaluer de nouvelles versions).

Mais le domaine des outils d'évaluation pour les systèmes distribués est encore en plein mouvement. De nombreux simulateurs ont été écrits, mais un grand nombre d'entre eux ont été abandonnés depuis. Si pour certains sous-domaines, on assiste à une convergence vers un simulateur unique, c'est loin d'être partout le cas. Les plates-formes expérimentales souffrent d'un problème similaire : de nombreuses plates-formes sont créées, mais tendent à évoluer en vase clos, sans chercher à construire des passerelles vers les autres plates-formes. Certains travaux de rapprochement de Emulab et Planet-Lab [25] existent, mais sont l'exception plutôt que la règle. Chaque plate-forme a des caractéristiques différentes, et fournit une interface et des services différents, rendant toute interopérabilité difficile.

Les mêmes difficultés sont présentes pour les émulateurs. Ceux-ci sont souvent conçus pour une plate-forme spécifique, et ne sont pas destinés à être réellement diffusés (seuls Modelnet et Emulab ont fait l'objet de déploiements en dehors de leurs laboratoires d'origine) : les auteurs semblent chercher d'abord à valoriser leur outil par des publications, plutôt que par la diffusion de logiciels. Cela rend leur comparaison difficile, voire impossible si on souhaite aller plus

loin qu'une simple comparaison de leurs fonctionnalités annoncées. De plus, une convergence sur certains aspects, comme sur quelques modèles de topologie ou quelques suites de tests serait souhaitable, en prenant garde de ne pas provoquer une ossification de ce domaine : standardiser un domaine de recherche en introduisant des suites de tests peut limiter les possibilités de changements fondamentaux dans ce domaine.

Il faut aussi noter la difficulté intrinsèque du travail dans ce domaine au carrefour du système et du réseau. On y travaille sur des infrastructures logicielles très complexes, où des changements à des niveaux éloignés peuvent grandement changer les résultats obtenus. Nous avons aussi travaillé sur des systèmes proches mais qui diffèrent parfois de manière subtile (FreeBSD et Linux, puis Dummynet, NISTNET et Linux TC). Tous ces systèmes étaient également en constante évolution, et des résultats obtenus à une date donnée ne seront pas forcément encore vrais six mois plus tard.

De plus, il a fallu porter une grande attention à la « qualification » de la plate-forme et des logiciels utilisés : il est facile de laisser un paramètre sans rapport perturber les résultats d'une expérience : une version de compilateur différente sur deux systèmes qui optimisera différemment un programme de test, ou une infrastructure réseau qui sera affectée par d'autres expériences exécutées simultanément par d'autres utilisateurs. Enfin, le travail sur une plate-forme comme Grid'5000 n'est pas non plus toujours facile, et nécessite également une expertise spécifique : on se retrouve malheureusement trop souvent bloqué par ce que la plate-forme permet à l'utilisateur de faire.

8.2 Contributions

Dans ce contexte, nous avons utilisé une approche incrémentale : nous avons commencé par valider les *briques de base* utilisées, avant de construire une plate-forme complète. Nos principales contributions sont les suivantes :

Étude comparative des émulateurs de liens réseaux

Dans un premier temps, nous avons conduit une étude comparative de Dummynet, NISTNet et Linux TC/Netem. Ces émulateurs sont utilisés à l'intérieur d'autres plates-formes d'émulation, mais n'avaient pas été validés et comparés. Nous avons montré que ces émulateurs souffraient d'un problème lié à la source de temps utilisée, problème qui se manifeste par une latence émulée variant fortement et l'envoi de paquets en masse (*burst*). Ces problèmes peuvent fortement perturber les résultats des expérimentations sur les plates-formes utilisant ces émulateurs, comme Emulab, Wrekavoc, V-DS, ou eWan. Nous proposons un ensemble de configurations ne présentant pas ce problème, et évoquons également d'autres problèmes à prendre en compte dans l'utilisation de ces émulateurs,

comme le point d'interception des paquets.

Nous avons ensuite donné quelques exemples de l'utilisation de ces émulateurs, sur l'étude d'applications simples et pour l'émulation de topologies réseaux sans autre infrastructure particulière.

Ce travail n'a pas encore fait l'objet de publications.

Développement d'un tunnel IP sur DNS, et comparaison avec les autres implantations

Nous avons ensuite cherché à utiliser un émulateur pour comparer des applications au protocole réseau plus complexe. Nous avons développé un tunnel IP sur DNS, appelé TUNS, et comparé ce dernier avec les autres implantations existantes de tunnels IP sur DNS dans un grand nombre de conditions expérimentales, à l'aide de Linux TC/Netem. Malgré des choix très conservateurs dans la définition du protocole afin de fonctionner sur n'importe quel réseau, TUNS délivre des performances acceptables.

TUNS est diffusé publiquement [91] et a fait l'objet d'une publication dans une conférence francophone [92].

P2PLab : une plate-forme d'émulation pour l'étude des systèmes pair-à-pair

Nous avons ensuite proposé une plate-forme d'émulation pour l'étude des systèmes pair-à-pair, P2PLab. P2PLab utilise de la virtualisation légère, au niveau processus, au lieu d'un système d'émulation plus lourd. L'émulation réseau est déportée sur les noeuds et réalisée avec Dummynet. Nous avons réalisé des expériences en utilisant deux implantations différentes du protocole de diffusion de fichiers BitTorrent, montrant que la solution de virtualisation de P2PLab, sous les conditions que nous avons choisies, ne provoquait pas de biais jusqu'à un rapport de 80 noeuds virtuels par noeud physique. De plus, nous avons utilisé jusqu'à 13000 noeuds virtuels dans nos expériences.

P2PLab est diffusé publiquement [93], mais n'est probablement pas encore prêt à être largement utilisé. Il a fait l'objet de publications dans un Workshop international [82] et une conférence francophone [94], toutes deux suivies de publications dans des revues [95, 96].

8.3 Perspectives

Notre travail ouvre de nombreuses perspectives dans plusieurs domaines.

Vers l'émulateur idéal

Si P2PLab apporte des spécificités intéressantes, il est loin d'être la solution d'émulation idéal, répondant à tous les besoins. Chaque émulateur réalise un nombre important de compromis, et il serait intéressant de converger vers un framework générique pour l'émulation de systèmes distribués, où l'utilisateur pourrait, pour chaque paramètre, choisir parmi différentes solutions. Cet émulateur idéal permettrait de faire à la fois de la virtualisation légère (comme notre approche) ou plus lourde (*FreeBSD jails*, *Linux VServer*, *Xen* comme dans V-DS [56], ...), d'utiliser différents émulateurs de topologies réseaux, d'utiliser ou non de la dilatation temporelle, etc.

D'autres aspects ont pour l'instant été laissés de côté. Nous nous sommes pour l'instant placés dans des conditions idéales. Des travaux récents ont montré l'importance du trafic de fond [97], qu'il faudrait pouvoir émuler pendant nos expériences. Il faudrait aussi pouvoir injecter des fautes, notamment pour émuler le départ et l'arrivée de noeuds, qui joue un rôle important dans les performances des systèmes distribués.

D'autre part, afin de pouvoir aborder des expériences à plus de 100000 noeuds, il faudrait être capable de tirer partie d'infrastructures distribuées : exécuter une expérience sur plusieurs clusters répartis géographiquement pose des problèmes spécifiques (prise en compte de la latence et des goulots d'étranglement réseaux).

Il serait aussi intéressant de profiter de technologies du HPC : tirer profit de manière plus explicite des multi-coeurs dans la virtualisation, et utiliser les réseaux rapides (Myrinet, Infiniband) pour l'émulation du réseau.

Vers un émulateur utilisable et utilisé

Il est aussi important de progresser vers une « industrialisation » de l'outil, afin qu'il soit facile à utiliser par la communauté, et donc qu'il puisse s'enrichir grâce aux retours des utilisateurs. Cela pourrait passer par une intégration plus poussée dans Grid'5000, qui ne dispose pour l'instant pas de solution de ce type : c'est à chaque utilisateur de construire sa solution ad-hoc, au contraire de ce qui est proposé dans Emulab, par exemple.

Mais ce travail, positionné entre ingénierie et recherche, est difficile : construire un outil réellement utilisable et utilisé nécessite de prendre en compte des problématiques de diffusion, de qualité logicielle, d'ergonomie de l'interface utilisateur.

BIBLIOGRAPHIE

- [1] Carl Kesselman et Ian Foster. *The Grid : Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998.
- [2] Wu chun Feng et Kirk Cameron. The green500 list : Encouraging sustainable supercomputing. *Computer*, 40(12) :50–55, 2007.
- [3] Brian Hayes. Cloud computing. *Commun. ACM*, 51(7) :9–11, 2008.
- [4] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), April 1965.
- [5] David Oppenheimer, Archana Ganapathi et David A. Patterson. Why do internet services fail, and what can be done about it? In *USITS'03 : Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.
- [6] Jeffrey C. Mogul. Emergent (mis)behavior vs. complex software systems. In *EuroSys '06 : Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 293–304, New York, NY, USA, 2006. ACM.
- [7] Andreas Haeberlen, Alan Mislove et Peter Druschel. Glacier : highly durable, decentralized storage despite massive correlated failures. In *NSDI'05 : Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 143–158, Berkeley, CA, USA, 2005. USENIX Association.
- [8] Henri Casanova, Arnaud Legrand et Martin Quinson. SimGrid : A generic framework for large-scale distributed experiments. In *UKSIM '08 : Proceedings of the Tenth International Conference on Computer Modeling and Simulation (uksim 2008)*, pages 126–131, Washington, DC, USA, 2008. IEEE Computer Society.
- [9] Anthony Sulistio, Uros Cibej, Srikumar Venugopal, Borut Robic et Rajkumar Buyya. A toolkit for modelling and simulating data grids : an extension to gridsim. *Concurr. Comput. : Pract. Exper.*, 20(13) :1591–1609, 2008.
- [10] William H. Bell, David G. Cameron, Luigi Capozza, A. Paul Millar et Kurt Stockinger. OptorSim - a grid simulator for studying dynamic data replication strategies. *International Journal of High Performance Computing Applications*, 17 :2003, 2003.
- [11] S. Naicken, B. Livingston, A. Basu, S. Rodhetbhai, I. Wakeman et D. Chalmers. The state of peer-to-peer simulators and simulations. *SIGCOMM Comput. Commun. Rev.*, 37(2) :95–98, 2007.
- [12] p2psim, a simulator for peer-to-peer protocols. <http://pdos.csail.mit.edu/p2psim/>.

- [13] Pedro García López, Carles Pairet, Rubén Mondéjar, Jordi Pujol Ahulló, Helio Tejedor et Robert Rallo. Planetsim : A new overlay network simulation framework. In *SEM*, volume 3437 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2004.
- [14] Kazuyuki Shudo, Yoshio Tanaka et Satoshi Sekiguchi. Overlay weaver : An overlay construction toolkit. *Comput. Commun.*, 31(2) :402–412, 2008.
- [15] PeerSim : A peer-to-peer simulator. <http://peersim.sourceforge.net/>.
- [16] Adolfo Rodriguez, Charles Killian, Sooraj Bhat, Dejan Kostić et Amin Vahdat. Macedon : methodology for automatically creating, evaluating, and designing overlay networks. In *NSDI'04 : Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.
- [17] Ashwin R. Bharambe et Cormac Herley. Analyzing and improving BitTorrent performance. Technical Report MSR-TR-2005-03, Microsoft Research, 2005.
- [18] FreePastry. <http://freepastry.org/FreePastry/>.
- [19] ns-2 : Network simulator. http://nsnam.isi.edu/nsnam/index.php/Main_Page.
- [20] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu et Haobo Yu. Advances in network simulation. *Computer*, 33(5) :59–67, 2000.
- [21] Network emulation with the NS simulator. <http://www.isi.edu/nsnam/ns/ns-emulation.html>.
- [22] The ns-3 network simulator. <http://www.nsnam.org/>.
- [23] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb et Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Décembre 2002. USENIX Association.
- [24] Robert Ricci, Chris Alfeld et Jay Lepreau. A solver for the network testbed mapping problem. *SIGCOMM Comput. Commun. Rev.*, 33(2) :65–81, 2003.
- [25] Robert Ricci, Jonathon Duerig, Pramod Sanaga, Daniel Gebhardt, Mike Hibler, Kevin Atkinson, Junxing Zhang, Sneha Kasera et Jay Lepreau. The Flexlab approach to realistic evaluation of networked systems. In *Proc. of the Fourth Symposium on Networked Systems Design and Implementation (NSDI 2007)*, Cambridge, MA, Avril 2007.
- [26] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb et Jay Lepreau. Large-scale virtualization in the emulab network testbed. In *USENIX 2008 Annual Technical Conference*, Berkeley, CA, USA, 2008. USENIX Association.

- [27] Franck Cappello, Eddy Caron, Michel Dayde, Frederic Desprez, Emmanuel Jeannot, Yvon Jegou, Stephane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet et Olivier Richard. Grid'5000 : a large scale, reconfigurable, controlable and monitorable Grid platform. In *Grid'2005 Workshop*, Seattle, USA, November 13-14 2005. IEEE/ACM.
- [28] Kadeploy. <http://kadeploy.imag.fr/>.
- [29] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak et Mic Bowman. PlanetLab : An Overlay Testbed for Broad-Coverage Services. *ACM SIGCOMM Computer Communication Review*, 33(3) :3–12, July 2003.
- [30] Elliot Jaffe, Danny Bickson et Scott Kirkpatrick. Everlab : a production platform for research in network experimentation and computation. In *LISA'07 : Proceedings of the 21st conference on Large Installation System Administration Conference*, pages 1–11, Berkeley, CA, USA, 2007. USENIX Association.
- [31] OneLab - future internet test beds. <http://www.onelab.eu/>.
- [32] Larry Peterson, Andy Bavier, Marc E. Fiuczynski et Steve Muir. Experiences building planetlab. In *OSDI '06 : Proceedings of the 7th symposium on Operating systems design and implementation*, pages 351–366, Berkeley, CA, USA, 2006. USENIX Association.
- [33] Neil Spring, Larry Peterson, Andy Bavier et Vivek Pai. Using planetlab for network research : myths, realities, and best practices. *SIGOPS Oper. Syst. Rev.*, 40(1) :17–24, 2006.
- [34] Suman Banerjee, Timothy G. Griffin et Marcelo Pias. The Interdomain Connectivity of PlanetLab Nodes. In *Proceedings of the Passive and Active Measurement Workshop (PAM2004)*, Antibes Juan-les-Pins, France, April 2004.
- [35] Himabindu Pucha, Y. Charlie Hu et Z. Morley Mao. On the impact of research network based testbeds on wide-area experiments. In *IMC '06 : Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 133–146, New York, NY, USA, 2006. ACM.
- [36] Andreas Haeberlen, Alan Mislove, Ansley Post et Peter Druschel. Fallacies in evaluating decentralized systems. In *Proceedings of the 5th International Workshop on Peer-to-Peer Systems (IPTPS'06)*, February 2006.
- [37] Marcel Dischinger, Andreas Haeberlen, Ivan Beschastnikh, Krishna P. Gummadi et Stefan Saroiu. Satellitelab : adding heterogeneity to planetary-scale network testbeds. *SIGCOMM Comput. Commun. Rev.*, 38(4) :315–326, 2008.
- [38] DSLLab. <http://www.dsllab.org/>, 2005.
- [39] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounie, P. Neyron et O. Richard. A batch scheduler with high level components. In *CCGRID '05 : Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2*, pages 776–783, Washington, DC, USA, 2005. IEEE Computer Society.

- [40] Luigi Rizzo. Dummynet : a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1) :31–41, 1997.
- [41] Luigi Rizzo. Dummynet and forward error correction. In *Proceedings of the FreeNIX Track : USENIX 1998 annual technical conference*. Usenix, 1998.
- [42] Mark Carson et Darrin Santay. NIST Net : a Linux-based network emulation tool. *SIGCOMM Comput. Commun. Rev.*, 33(3) :111–126, 2003.
- [43] Stephen Hemminger. Network emulation with NetEm. In *linux.conf.au 2005*, 2005.
- [44] Y. Kodama, T. Kudoh, R. Takano, H. Sato, O. Tatebe et S. Sekiguchi. GNET-1 : gigabit ethernet network testbed. In *CLUSTER '04 : Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 185–192, Washington, DC, USA, 2004. IEEE Computer Society.
- [45] Anué systems. <http://www.anuesystems.com>.
- [46] H. J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura et A. Chien. The MicroGrid : a scientific tool for modeling computational grids. In *Supercomputing '00 : Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 53. IEEE Computer Society, 2000.
- [47] Xin Liu et Andrew Chien. Realistic large scale online network simulation. In *ACM Conference on High Performance Computing and Networking (SC2004)*. ACM Press, 2004.
- [48] Andy Bavier, Nick Feamster, Mark Huang, Larry Peterson et Jennifer Rexford. In VINI veritas : Realistic and controlled network experimentation. In *Proceedings of ACM SIGCOMM 2006*, Pisa, Italy, September 2006.
- [49] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostik, Jeff Chase et David Becker. Scalability and accuracy in a large-scale network emulator. In *OSDI '02 : Proceedings of the 5th symposium on Operating systems design and implementation*, pages 271–284, New York, NY, USA, 2002. ACM Press.
- [50] Ken Yocum, Ethan Eade, Julius Degeys, David Becker, Jeffrey S. Chase et Amin Vahdat. Toward scaling network emulation using topology partitioning. In *MASCOTS*, pages 242–245. IEEE Computer Society, 2003.
- [51] Ken Yocum et Jeffrey S. Chase. Payload caching : High-speed data forwarding for network intermediaries. In *Proceedings of the General Track : 2002 USENIX Annual Technical Conference*, pages 305–317, Berkeley, CA, USA, 2001. USENIX Association.
- [52] Jay Chen, Diwaker Gupta, Kashi V. Vishwanath, Alex C. Snoeren et Amin Vahdat. Routing in an internet-scale network emulator. In *MASCOTS '04 : Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 275–283, Washington, DC, USA, 2004. IEEE Computer Society.

- [53] Priya Mahadevan, Adolfo Rodriguez, David Becker et Amin Vahdat. Mobinet : a scalable emulation infrastructure for ad hoc and wireless networks. In *WiTMeMo '05 : Papers presented at the 2005 workshop on Wireless traffic measurements and modeling*, pages 7–12, Berkeley, CA, USA, 2005. USENIX Association.
- [54] Louis-Claude Canon et Emmanuel Jeannot. Wrekavoc a Tool for Hemulating Heterogeneity. In *15th IEEE Heterogeneous Computing Workshop (HCW'06)*, Island of Rhodes, Greece, Avril 2006.
- [55] Benjamin Quétier, Mathieu Jan et Franck Cappello. One step further in large-scale evaluations : the v-ds environment. Research Report RR-6365, INRIA, December 2007.
- [56] Benjamin Quétier. *EMUGRID : études des mécanismes de virtualisation pour l'émulation conforme de grilles à grande échelle*. Thèse de Doctorat, Université Paris XI - Orsay, 2008.
- [57] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt et Andrew Warfield. Xen and the art of virtualization. In *SOSP '03 : Proceedings of the nineteenth ACM symposium on Operating systems principles*, New York, NY, USA, 2003. ACM Press.
- [58] Franck Cappello Benjamin Quetier, Vincent Neri. Scalability comparison of four host virtualization tools. *Journal of Grid Computing*, 2006.
- [59] Diwaker Gupta, Kashi V. Vishwanath et Amin Vahdat. Diecast : testing distributed systems with an accurate scale model. In *NSDI'08 : Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 407–422, Berkeley, CA, USA, 2008. USENIX Association.
- [60] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat et Geoffrey M. Voelker. To infinity and beyond : time-warped network emulation. In *NSDI'06 : Proceedings of the 3rd conference on Networked Systems Design & Implementation*, pages 7–7, Berkeley, CA, USA, 2006. USENIX Association.
- [61] The DiskSim simulation environment. <http://www.pdl.cmu.edu/DiskSim/>.
- [62] Pei Zheng et Lionel M. Ni. Empower : A cluster architecture supporting network emulation. *IEEE Trans. Parallel Distrib. Syst.*, 15(7) :617–629, 2004.
- [63] M. Zec et M. Mikuc. Operating system support for integrated network emulation in IMUNES. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure / ASPLOS-XI*, 2004.
- [64] George Apostolopoulos et Constantinos Hassapis. V-em : A cluster of virtual machines for robust, detailed, and high-performance network emulation. In *MASCOTS '06 : Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation*, pages 117–126, Washington, DC, USA, 2006. IEEE Computer Society.

- [65] P. Vicat-Blanc Primet, R. Takano, Y. Kodama, T. Kudoh, O. Gluck et C. Otal. Large scale gigabit emulated testbed for grid transport evaluation. In *PFLD-net 2006*, 2006.
- [66] Ryousei Takano, Tomohiro Kudoh, Yuetsu Kodama, Motohiko Matsuda, Hiroshi Tezuka et Yutaka Ishikawa. Design and evaluation of precise software pacing mechanisms for fast long-distance networks. In *PFLDNet 2005*, 2005.
- [67] Jong Suk Ahn, Peter B. Danzig, Zhen Liu et Limin Yan. Evaluation of TCP vegas : emulation and experiment. *SIGCOMM Comput. Commun. Rev.*, 25(4) :185–195, 1995.
- [68] David B. Ingham et Graham D. Parrington. Delayline : A wide-area network emulation tool. *Computing Systems*, 7(3) :313–332, 1994.
- [69] Mark Allman, Adam Caldwell et Shawn Ostermann. ONE : The ohio network emulator. Technical Report TR-19972, Ohio University, August 1997.
- [70] Hxibt : WAN emulator for Solaris. <http://www.opensolaris.org/os/community/networking/readme.hxibt.txt>.
- [71] Thomas Gleixner et Douglas Niehaus. Hrtimers and beyond : Transforming the linux time subsystems. In *Proceedings of the Ottawa Linux Symposium*, volume vol. 1, pages p.333–346, 2006.
- [72] RFC 1035 : Domain names - implementation and specification. <http://www.ietf.org/rfc/rfc1035.txt>.
- [73] Nstx. <http://thomer.com/howtos/nstx.html>.
- [74] Iodine. <http://code.kryo.se/iodine/>.
- [75] RFC 2671 : Extension mechanisms for DNS (EDNS0). <http://www.ietf.org/rfc/rfc2671.txt>.
- [76] Ozymandns. <http://www.doxpara.com/>.
- [77] Dns2tcp. <http://hsc.fr/ressources/outils/dns2tcp/>.
- [78] The Linux VServer Project. <http://www.linux-vserver.org/Linux-VServer-Paper>, 2003.
- [79] Jeff Dike. A user-mode port of the Linux kernel. In *Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta*, page 63. Usenix, 2000.
- [80] KVM - kernel based virtual machine. <http://kvm.qumranet.com/>.
- [81] Jeff Roberson. ULE : A modern scheduler for FreeBSD. In *Proceedings of BSDCon*, 2003.
- [82] Lucas Nussbaum et Olivier Richard. Lightweight emulation to study peer-to-peer systems. In *Third International Workshop on Hot Topics in Peer-to-Peer Systems (Hot-P2P 06)*, Rhodes Island, Greece, 4 2006.
- [83] Ian T. Foster et Adriana Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, 2003.

- [84] Bram Cohen. Incentives build robustness in BitTorrent. <http://www.bittorrent.com>, 2003.
- [85] J.A. Pouwelse, P. Garbacki, D.H.J. Epema et H.J. Sips. The Bittorrent P2P file-sharing system : Measurements and analysis. In *4th International Workshop on Peer-to-Peer Systems (IPTPS)*, feb 2005.
- [86] Mikel Izal, Guillaume Urvoy-Keller, Ernst W Biersack, Pascal A Felber, Anwar Al Hamra et Luis Garces-Erice. Dissecting BitTorrent : five months in a torrent's lifetime. In *PAM'2004, 5th annual Passive & Active Measurement Workshop, April 19-20, 2004, Antibes Juan-les-Pins, France / Also Published in Lecture Notes in Computer Science (LNCS), Volume 3015, Barakat, Chadi ; Pratt, Ian (Eds.) 2004, Apr 2004*.
- [87] Dongyu Qiu et R. Srikant. Modeling and performance analysis of BitTorrent-like peer-to-peer networks. In *SIGCOMM '04 : Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 367–378, New York, NY, USA, 2004. ACM Press.
- [88] Brice Videau, Corinne Touati et Olivier Richard. Toward an experiment engine for lightweight grids. In *MetroGrid*, Octobre 2007.
- [89] Nikitas Liogkas, Robert Nelson, Eddie Kohler et Lixia Zhang. Exploiting bittorrent for fun (but not profit). In *Proceedings of The 5th International Workshop on Peer-to-Peer Systems (IPTPS '06)*, Santa Barbara, CA, February 2006.
- [90] Software bug, etymology – wikipedia. http://en.wikipedia.org/wiki/Software_bug#Etymology.
- [91] Tuns. <http://www-id.imag.fr/~nussbaum/tuns.php>.
- [92] Lucas Nussbaum et Olivier Richard. Prototype de canal caché dans le DNS. In *Colloque Francophone sur l'Ingénierie des Protocoles (CFIP)*, Les Arcs, France, 03 2008.
- [93] P2plab. <http://www-id.imag.fr/~nussbaum/p2plab.php>.
- [94] Lucas Nussbaum. Une plate-forme d'Émulation légère pour Étudier les systèmes pair-à-pair. In *Rencontres francophones du Parallélisme (RenPar'17)*, Perpignan, France, 10 2006.
- [95] Lucas Nussbaum et Olivier Richard. Lightweight emulation to study peer-to-peer systems. *Concurrency and Computation : Practice and Experience*, 2007.
- [96] Lucas Nussbaum et Olivier Richard. Une plate-forme d'Émulation légère pour Étudier les systèmes pair-à-pair. *Technique et Science Informatiques - numéro spécial RenPar'17*, 26, 2008.
- [97] Kashi Venkatesh Vishwanath et Amin Vahdat. Evaluating distributed systems : does background traffic matter ? In *ATC'08 : USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 227–240, Berkeley, CA, USA, 2008. USENIX Association.

TABLE DES FIGURES

2.1	Emulab : Interface graphique de définition de topologie.	18
2.2	Disponibilité des 891 noeuds de PlanetLab au 30 septembre 2008. . .	21
2.3	Processus de distillation de Modelnet.	25
3.1	Configuration expérimentale	33
3.2	Émulation de latence : évolution au cours du temps de la latence appliquée au paquets.	35
3.3	Émulation de latence : fonction de répartition des latences émuloées, pour des paquets envoyés avec un intervalle aléatoire.	36
3.4	Bande passante mesurée lorsque la bande passante paramétrée dans l'émulateur varie entre 100 kbps et 1 Gbps (échelle logarithmique), puis entre 500 Mbps et 1 Gbps	38
3.5	Différence entre la bande passante paramétrée dans l'émulateur et celle mesurée	39
3.6	Utilisation d'un périphérique logiciel <code>ifb</code> pour appliquer les paramètres d'émulation aux paquets entrants avec TC.	42
3.7	Fonction de répartition du temps de traversée d'une machine agissant comme un routeur, avec et sans redirection des paquets entrants vers un périphérique <code>ifb</code>	43
4.1	Configuration expérimentale. La latence est ajoutée sur les paquets sortant du client et du serveur	46
4.2	Temps nécessaire pour exécuter une commande simple sur un serveur distant en utilisant RSH ou SSH.	46
4.3	Temps de transfert de 120 fichiers (taille totale : 2.1 Mo) en utilisant SCP et Rsync, avec différentes conditions de latence et de débit. . .	48
4.4	Topologie <i>Dumbbell</i> à émuler	49
5.1	Principe général des canaux cachés dans le DNS	54
5.2	Contenu des paquets DNS échangés entre un client et un serveur TUNS, capturés avec l'analyseur de trafic <code>wireshark</code>	58
5.3	Configuration expérimentale.	59

5.4	Latence perçue, mesurée avec <code>ping</code> depuis le client. Les barres verticales indiquent les valeurs minimum et maximum sur 20 mesures.	60
5.5	Débit montant disponible (client vers serveur).	61
5.6	Latence perçue, mesurée avec <code>ping</code> depuis le serveur.	62
5.7	Débit descendant disponible (serveur vers client).	62
5.8	Mesures de latence sur un réseau émulant pertes de paquets et réordonnements.	64
5.9	Mesures de débit sur un réseau émulant pertes de paquets et réordonnements.	65
6.1	Temps moyen d'exécution en fonction du nombre de processus concurrents. Les processus utilisent intensivement le processeur, mais n'utilisent que peu de mémoire	73
6.2	Temps moyen d'exécution en fonction du nombre de processus concurrents. Les processus utilisent intensivement le processeur et la mémoire	75
6.3	Fonction de répartition des temps d'exécution des processus avec Linux 2.6 et les deux ordonnanceurs de FreeBSD. $F(x)$ indique la proportion des processus ayant terminé avant t	76
6.4	Utilisation d'alias d'interface sur chaque machine physique, pour chaque IP de nœud virtuel	78
6.5	Enchaînement des appels systèmes réseaux dans le cadre de connexions TCP	78
6.6	Evolution du <i>round-trip time</i> en fonction du nombre de règles de pare-feu à évaluer	81
6.7	Exemple de topologie émulée par P2PLab	82
7.1	Evolution du téléchargement des 160 clients	87
7.2	Rapport de virtualisation de P2PLab : quantité totale de données récupérées par les 160 clients avec les différents déploiements	88
7.3	Evolution du transfert d'un fichier de 16 Mo entre 13 032 clients sur quelques nœuds sélectionnés (nœuds 101, 200, 301, 400 ...)	90
7.4	Fonction de répartition du temps de complétion du transfert sur les différents nœuds	90
7.5	Comparaison de BitTorrent et CTorrent, exécutés soit seuls, soit avec des clients de l'autre type comme "adversaires"	91

LISTE DES TABLEAUX

3.1	Fonctionnalités de Dummysnet, NISTNet, and TC/Netem	32
3.2	Temps moyen d'exécution d'un processus utilisant le CPU sur FreeBSD, en utilisant différentes fréquences d'horloge	37

Cette thèse s'inscrit dans le domaine de l'expérimentation sur les systèmes distribués, et en particulier de leur test ou de leur validation. À côté des méthodes d'évaluation classiques (modélisation, simulation, plates-formes d'expérimentation comme PlanetLab ou Grid'5000) les méthodes basées sur l'émulation et la virtualisation proposent une alternative prometteuse. Elles permettent d'exécuter l'application réelle à étudier, en lui présentant un environnement synthétique, correspondant aux conditions d'expérience souhaitées : il est ainsi possible, à moindre coût, de réaliser des expériences dans des conditions expérimentales différentes, éventuellement impossibles à reproduire dans un environnement réel. Mais l'utilisation de tels outils d'émulation ne peut se faire sans répondre à des questions sur leur réalisme et leur passage à l'échelle.

Dans ce travail, nous utilisons une démarche incrémentale pour construire une plate-forme d'émulation destinée à l'étude des systèmes pair-à-pair à grande échelle. Nous commençons par comparer les différentes solutions d'émulation logicielle de liens réseaux, puis illustrons leur utilisation, notamment en étudiant une application réseau complexe : TUNS, un tunnel IP sur DNS. Nous construisons ensuite notre plate-forme d'émulation, P2PLab, en utilisant l'un des émulateurs réseaux précédemment étudiés, ainsi qu'un modèle de topologies réseaux adapté à l'étude des systèmes pair-à-pair. Nous y proposons une solution légère de virtualisation, permettant un bon rapport de repliement (grand nombre de noeuds émulsés sur chaque machine physique). Après avoir validé cette plate-forme, nous l'utilisons pour étudier le protocole de diffusion de fichiers pair-à-pair BitTorrent à l'aide d'expériences mettant en jeu près de 15000 noeuds participants.