N° d'ordre 208-2003 Année 2003

UNIVERSITÉ CLAUDE BERNARD - LYON I
Laboratoire de l'Informatique du Parallélisme
Équipe RESO : Protocoles et logiciels optimisés pour réseaux très haut débit

THÈSE

présentée par

Jean-Patrick Gelas

pour obtenir le titre de
Docteur de l'Université Claude Bernard de Lyon
Spécialité Informatique

Soutenue publiquement le 5 décembre 2003

Titre :

Vers la conception d'une architecture de réseaux actifs apte à supporter les débits des réseaux gigabits

---

Directeur de thèse : Laurent LEFÈVRE (Chargé de Recherche INRIA - LIP, ENS Lyon)

Composition du jury :

M. Ken CHEN	Rapporteur
M. Bernhard PLATTNER	Rapporteur
M. Michel DIAZ	Membre
M. Laurent LEFÈVRE	Membre
M. Bernard PEROCHE	Membre
Mme. Pascale VICAT-BLANC/PRIMET	Membre

A tous les soit disant mauvais élèves de $6^{\text{ème}}$*.*

Remerciements

Avant de remercier, comme il est de tradition justifiée, toutes les personnes ayant contribué directement ou indirectement à l'aboutissement de cette thèse, j'aimerais rappeler, afin d'encourager les soit disant mauvais élèves, mon parcours scolaire plutôt atypique.

Si l'on considère que les études débutent au collège, dés la $6^{\text{ème}}$ on décide d'entraver ma soif d'apprendre en me faisant redoubler sous prétexte que je maîtrise mal le français à l'écrit. Suite à cela, j'ai dû probablement devenir complètement hermétique à tous les cours de français que j'ai suivi par la suite, car je n'ai plus jamais progressé en grammaire ;-)

En fin de $3^{\text{ème}}$, conseillers d'orientation et psychologues scolaires me prédisent un avenir auquel je n'aspire guère (BEP, CAP). Heureusement, mes parents, plus confiant dans mes capacités, me poussent à aller au lycée. Je ne m'en sors pas trop mal car les cours sont de plus en plus intéressants et en fin de seconde, on me propose même d'aller en $1^{\text{ère}}$ E (scientifique), mais doutant moi même de mes aptitudes, je préfère une filière plus « cool ». Me voila en train de préparer un bac technologique F3 (Électrotechnique) que j'obtiendrai tout juste probablement du fait de mon manque de motivation. Je n'ai à l'évidence pas encore trouvé ma voie.

Parallèlement à ces études laborieuses, je « m'éclate » littéralement depuis maintenant plus de sept ans, d'abord sur un petit ordinateur pourvu d'à peine 4ko de mémoire RAM (Alice), puis sur le bien connu Amstrad CPC 464, et enfin sur un PC 80286 (MS-DOS 3.3). Contrairement à mes petits camarades qui comme moi ont la chance d'avoir un ordinateur à la maison, je ne joue pas. Je programme, je code, j'essaye de comprendre comment fonctionne ce fantastique automate pendant des heures, sans jamais me lasser. J'adore ça !

Mais revenons-en à mon parcours scolaire. Je réussis finalement à entrer en BTS informatique industrielle. Je me dis alors que je vais enfin rencontrer des gens qui partagent ma passion pour les ordinateurs ! Malheureusement, je constate rapidement que la plupart, sinon tous, ont choisi ce BTS car l'informatique est un domaine en plein essor. Ils sont loin d'être passionnés comme je le suis. Mon stage en entreprise me fait vite prendre conscience que je ne dois pas m'arrêter à Bac+2 pour m'éclater dans mon futur job dont je n'ai pas encore la moindre idée. Qu'à cela ne tienne, j'ai mûri, je suis motivé, gonflé à bloc ! J'ai enfin trouvé ma voie. Pour me remettre à niveau et acquérir une culture et une démarche plus scientifique je rentre en DEUG. J'ai un peu de mal à suivre mais les cours me passionnent et je travaille beaucoup. Je décroche finalement mon premier diplôme Universitaire dés la $1^{\text{ère}}$ session (juin). Nous ne sommes qu'une poignée à l'avoir du premier coup.

Ensuite, je me prépare à intégrer la Licence d'informatique, mais la France me rappelle à elle pour que je lui sacrifie 10 mois de ma jeunesse et une année scolaire, comme si je n'avais pas déjà pris suffisamment de retard. Je suis furieux. Heureusement, je passe mes journées à réparer des ordinateurs, à faire du sport et je découvre cette année-là les joies de l'enseignement.

J'intègre finalement la Licence d'informatique où je découvre de nombreux domaines liés à l'informatique dont j'ignorais quasiment l'existence (calcul parallèle, data mining,...). Certaines matières me paraissent plus abstraites, notamment la compilation et la théorie des langages. Cela provient sûrement du fait que les enseignants emploient le mot « grammaire » sur lequel je fais toujours un blocage. Je réussis sans encombre à entrer en Maîtrise où je fais la rencontre d'un tout jeune maître de conférence qui, lors de mon stage de second cycle, va me faire découvrir et aimer le monde de la recherche. En fin de Maîtrise, je sais exactement ce que je veux faire. Chercheur en informatique. La voie royale pour atteindre cet objectif est le DEA d'Informatique Fondamentale (DIF) dont les cours sont dispensés dans la prestigieuse École Normale Supérieure de Lyon. Le jeune maître de conf' qui croit en moi m'épaule et j'intègre finalement ce DEA. J'assiste au cours aux côtés des étudiants les plus doués de France. La plupart ont une année d'avance, j'ai redoublé ma $6^{\text{ème}}$. Tous ont eu le bac avec la mention Très Bien, j'ai eu un bac techno' au rattrapage. Ils sont sortis major de leur prépa, il m'a fallu quatre ans pour obtenir un DEUG (et un BTS). Je suis intimidé, je ne me sens pas à ma place. Pourtant je suis les cours sans problèmes et finalement, j'obtiens le DEA avec mention Bien et grâce à mon classement, je décroche même un financement du ministère (MENRT) pour 3 ans, au grand soulagement de mes parents.

La suite vous la connaissez tous. Trois années entières consacrées, à un domaine de recherche précis, à enseigner et à voyager dans le monde pour présenter mes travaux. Bref, trois années passionnantes et inoubliables.

Je tiens donc à remercier chaleureusement :

- tout d'abord les membres de mon jury Pascale Vicat-Blanc/Primet, Michel Diaz, Bernard Peroche, et plus particulièrement mes rapporteurs, Bernhard Plattner et Ken Chen pour leur investissement dans la relecture de mon manuscrit.
- Mes « co-bureau » Moufida Maïmour et Benjamin Gaidioz. Véritable compagnons de thèse qui m'ont « supporté » sans jamais se plaindre tout au long de ces trois années. Moufida pour nos longues conversations passionnées à propos de religions et de « relations » garçons-filles. Benjamin pour son extrême gentillesse, pour les milliers de réponses aux milliers de questions techniques et administratives que j'ai pu lui poser et pour son sens aigu de la morale (éthique) qui le tourmente bien souvent.
- Les membres de l'équipe, qu'ils soient permanents, « jeunes » thésards, ingénieurs experts ou chercheur invité. Marc (pour ses réponses toujours ultra-argumentées), Eric, Pascale, Cong Duc (pour nos fabuleuses aventures en VTT), Saâd, Faycal, Fabien, Mathieu, Olivier, Alessandro (pour son accueil inoubliable à Knoxville, TN) et Brice (bon courage !). Merci également à nos fantastiques secrétaires Isabelle, Anne-Pascale et Sylvie pour leur aide précieuse.
- Les stagiaires avec qui j'ai eu la chance de collaborer et qui ont énormément contribué par leur esprit critique et leur enthousiasme au projet Tamanoir : Blandine, Sadek, Pierpaolo, Jerôme Guilloux et Jerôme Mazuy,...
- Mes parents et beaux-parents qui ont toujours cru en moi et qui ne m'ont jamais empêché de poursuivre un cursus scolaire plus qu'interminable. Ma grand-mère maternelle (alias grand-maman) pour son soutien sans limites. Alexandre mon « petit » frère. Fabrice qui m'a beaucoup aidé lorsque j'étais en DEUG.
- Stéphanie, pour sa confiance en moi et son soutien ces sept derniers mois, qui incluent la période de rédaction universellement reconnue comme la plus difficile dans la carrière d'un jeune chercheur. Je la remercie également de me permettre de finaliser mon vieux projet d'une année postdoctorale à l'étranger.

– Magali, pour avoir partagé ma vie ces huit dernières années, supporté mon caractère impossible et adhéré à tous mes projets.
– Et enfin, je tiens à remercier du fond du cœur celui sans qui absolument rien de tout cela n'aurait été possible. Plus qu'un directeur de thèse, c'est aujourd'hui un véritable ami, Laurent Lefèvre. Je le remercie pour son enthousiasme et sa bonne humeur inébranlable, son esprit vif et son imagination toujours fertile de bonnes idées, indispensable dans ce métier. Le jeune maître de conf' dont je parlais ci-dessus, c'est lui !

Table des matières

Table des figures

Chapitre 1

Introduction

Les réseaux actifs reposent sur des équipements réseaux ouverts et programmables ayant la possibilité de déployer dynamiquement des programmes qui seront appliqués à la volée sur les paquets de données.

1.1 Les réseaux actifs

Les réseaux actifs constituent aujourd'hui un domaine de recherche fortement étudié, notamment aux USA, en Europe et au Japon, dont le but est d'exploiter le réseau de manière plus efficace et plus intelligente. L'expérience a montré qu'il n'existera probablement jamais de protocole (réseaux, transport) ou service unique dont les caractéristiques sont telles qu'il est idéal pour tous supports et toutes applications. `TCP` dans sa forme actuelle, n'est pas idéal dans un environnement sans fils, satellitaire ou sur un réseau très haut débit, et `IP` est mal adapté pour des applications exécutées dans des conditions de forte mobilité. Face à l'apparition incessante de nouvelles applications s'appuyant sur les réseaux informatiques, il est impossible de mettre au point un service global capable de satisfaire toutes les applications. On peut donc imaginer fournir autant de services qu'il y a de types de flux de données transportées.

Un réseau actif contrairement à un réseau traditionnel n'est pas un simple support passif de paquets. Il peut être vu comme un ensemble de nœuds (routeurs) *actifs* qui réalisent des opérations personnalisées sur les flux de données qui le traversent, et qui autorise les utilisateurs, les opérateurs, ou les fournisseurs de services à injecter leurs propres programmes dans les nœuds du réseau, permettant ainsi de modifier, stocker (cacher) ou rediriger le flux de données à travers le réseau.

Le module de base pour la programmation d'un réseau actif est appelé *Application Active* (AA) ou service actif. Une AA est un code destiné à traiter un flux de données ou modifier des états de l'équipement sur lequel elle s'exécute. Elle peut affecter le plan donnée (ex : adaptation d'un flux vidéo à la volée), le plan contrôle (ex : marquage de paquets) ou le plan gestion (ex : reconfiguration des routes) d'un équipement actif. Pour garantir la sûreté de l'utilisation des ressources et un support de l'hétérogénéité des différentes architectures matérielles où s'exécute l'AA, celle-ci est généralement rédigée dans un langage fortement typé et indépendant de la plate-forme d'exécution.

Il y a trois composants logiciels qui permettent de construire une architecture active complète pour le support de l'exécution d'une AA [1] :

- l'Environnement d'Exécution (EE),
- le système d'exploitation du nœud (*node OS*),
- et l'application utilisateur.

Une AA est généralement écrite pour être exécutée dans un EE donné. Un EE doit être vu comme un cadre standardisé qui offre aux AA la possibilité d'être chargées dynamiquement puis exécutées. L'EE doit fournir l'interface qui permet à l'AA d'interagir de façon contrôlée avec le système d'exploitation qui lui seul donne accès à toutes les ressources (mémoires, réseaux, traitements).

Une application utilisateur doit pouvoir émettre un flux de paquets actifs. Elle est généralement exécutée sur un poste client mais peut être également exécutée à partir d'autres programmes de contrôles dans le réseau.

La figure 1.1 illustre les différents composants qui constituent un équipement actif. Les AA sont chargées dynamiquement et exécutées dans un EE. Sur la figure, l'EE1 exécute 2 AA et l'EE2 une seule. Dans le cas où plusieurs EE sont disponibles dans un équipement, une interface de démultiplexage est nécessaire à la re-direction des paquets actifs vers l'EE requis. Si un seul EE est disponible cette interface devient facultative, et l'EE interagit directement avec le système d'exploitation. Ce dernier permet à l'EE d'accéder simplement aux ressources matérielles. Certaines mises en œuvre de nœuds actifs reposent soit sur un système d'exploitation spécialisé (*node OS*), soit sur un système d'exploitation standard.

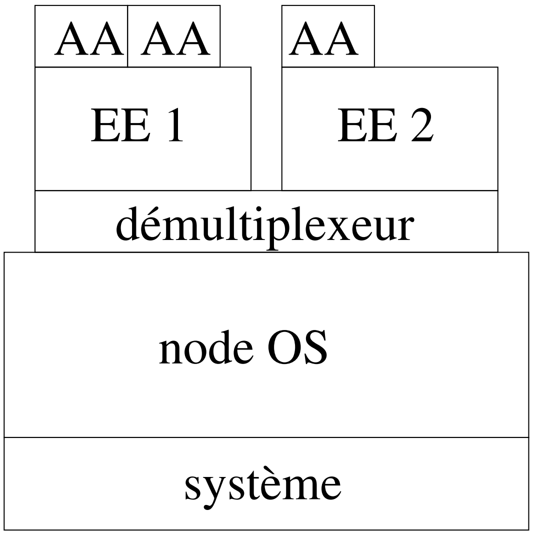


FIG. 1.1 – Composition d'un nœud actif.

1.1.1 Apports des réseaux actifs

Les réseaux actifs permettent un meilleur support de l'hétérogénéité qui se situe à trois niveaux (Figure 1.2) :

- L'hétérogénéité des contraintes de transport de données qui dépendent des exigences des applications en termes de débits, d'instants d'arrivées des données (flux temps-réel ou

asynchrone, gigue), et de qualité de transfert (erreurs induites lors du transport) ;
- L'hétérogénéité des technologies des réseaux de transport de données, qui bornent débits et latences, limitent la fiabilité et induisent parfois de l'asymétrie entre le lien montant et descendant (liaisons satellite, xDSL) ;
- L'hétérogénéité des postes clients (PDA, machine de bureau, serveurs,...) en termes de puissance de traitements et capacité à restituer l'information (résolution d'écran, équipements audio).

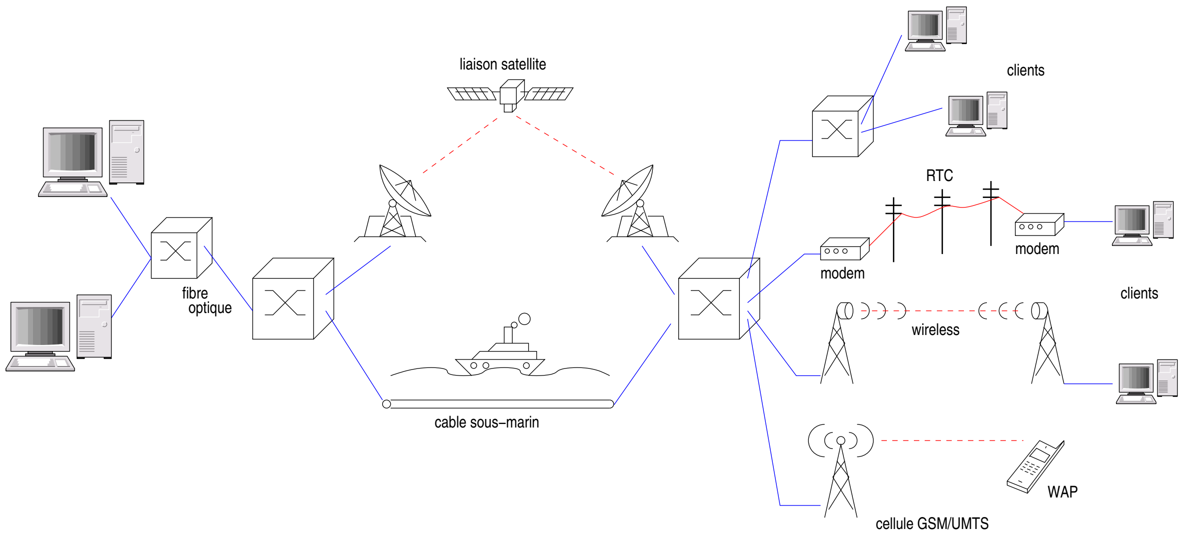


FIG. 1.2 – Réseau hétérogène en terme d'applications, de média de transport de données et de clients.

Afin d'exploiter au mieux les réseaux, il est nécessaire de disposer d'une infrastructure permettant le déploiement dynamique de nouveaux services dans les équipements du réseau. Ces services pourraient reconfigurer ces équipements, agir sur le plan contrôle ou sur le plan données du flux en fonction des données transportées, des médias de transport ou des caractéristiques des récepteurs.

1.1.2 Besoin en performances

Les réseaux actifs mettent en œuvre des services réseaux plus complexes que de la transmission simple de paquets de données, ce qui se traduit par une consommation plus importante de cycles CPU ou de mémoire pour l'équipement réseau.

Le développement des technologies de transmissions de données et de commutations ne cesse de s'améliorer. Le débit supporté par un lien ne cesse d'augmenter. Il existe aujourd'hui sur le marché des routeurs qui supporte plusieurs liens à 2.4 Gbps voire 10 Gbps [2].

Georges Gilder a énoncé une loi qui indique que la bande passante des réseaux augmente trois fois plus vite que la puissance des ordinateurs. Reste à savoir, si cette croissance est due à l'effet de cette loi, ou bien le résultat de la capacité de Georges Gilder à convaincre financiers, industriels et opérateurs à investir massivement dans le déploiement des réseaux. Les calculs d'Eric Schmidt (CEO de *Google*, ex-CEO de *Novell*) et les études de *Probe Research* montrent que la bande passante double en fait tous les ans depuis 1997[3].

Il est donc important de proposer des solutions efficaces pour mettre au point une architecture de routeurs actifs performante capable de fournir suffisament de ressources de calcul pour supporter des débits réseaux de plus en plus importants.

1.2 Contributions de la thèse

Dans un futur proche, la technologie « réseaux actifs » pourrait se retrouver dans différents types d'équipements réseaux (routeurs, passerelles, proxy...). Mais le concept des réseaux actifs, bien que séduisant, soulève de nombreuses problématiques, notamment de performances et de sécurité.

Mes travaux de thèse se sont principalement concentrés sur le problème de la performance ainsi que sur le déploiement dynamique de nouveaux services et du support de l'hétérogénéité matérielle et logicielle des équipements actifs déployés dans les réseaux dédiés longue distance haute performance et de l'Internet. Nous nous sommes focalisés sur la performance des nœuds actifs car elle contribuera à l'acceptation et au déploiement de cette technologie.

Mes travaux se placent dans le contexte des réseaux actifs logiciels. Cela signifie que nous proposons une architecture basée sur des logiciels et des matériels standards, non dédiés. Une solution au problème de la performance dans les réseaux actifs est de proposer des services efficaces et un Environnement d'Exécution (EE) adapté afin de minimiser le temps supplémentaire de transfert pour chaque paquet d'un flux de données sur le réseau. En profitant de notre expertise du calcul distribué et des réseaux haute performance nous proposons un ensemble de solutions architecturales originales dont certains sous-ensembles sont empruntés au domaine de la programmation système et des grappes de calcul haute performance. En combinant la haute performance et le déploiement de services portables, nous explorons de nouvelles approches capables d'ouvrir des pistes pour le développement des réseaux actifs. Nous proposons ainsi une nouvelle modélisation des réseaux actifs haute performance et sa mise en œuvre sous la forme d'un environnement et d'une suite logicielle appelé *Tamanoir*. C'est avec cette suite logicielle que nous menons de nombreuses expérimentations sur des réseaux très hauts débits afin de valider expérimentalement notre architecture. Les contributions de cette thèse se situent donc à un double niveau :

- Cette thèse apporte un ensemble de contributions originales à la conception d'une architecture de nœuds actifs apte à supporter les contraintes des réseaux actuels.
- Cette thèse propose la mise en œuvre d'un nœud actif logiciel performant validé expérimentalement.

Notre objectif est de concevoir un équipement actif logiciel bénéficiant à la fois de solutions de haut niveau pour pouvoir traiter facilement un flux applicatif, et des solutions de bas niveau, pour traiter efficacement un flux sur le plan contrôle.

Ce travail de recherche débute par un état de l'art exhaustif des travaux dédiés à l'aspect performance des réseaux actifs et a régulièrement été mis à jour au cours de ces trois années. Nous avons également observé dans quelles mesures, les opérateurs, les équipementiers et les industriels investissent dans ce domaine de recherche.

Notre première contribution consiste à mettre au point un nouveau modèle d'architecture de réseau actif et de nœud actif génériques qui fournissent toutes les fonctionnalités de base nécessaire à un déploiement rapide dans un contexte réaliste.

Notre seconde contribution consiste à enrichir ce modèle de base avec des caractéristiques novatrices. Nous proposons une classification des services en fonction des ressources qu'ils consomment pour les positionner le plus efficacement dans un équipement actif que nous avons divisé en plusieurs couches. Le paquet actif peut alors être traité dans un espace d'exécution adapté aux exigences du service. Nous proposons également plusieurs modes de déploiements des nœuds actifs et de services actifs.

Une contribution supplémentaire est de fournir des outils développés parallèlement à Tamanoir qui offre la possibilité de tester et gérer une infrastructure active déployée dans un réseau longue distance.

La dernière contribution est la proposition d'une validation expérimentale et fonctionnelle de la mise en œuvre de l'architecture active performante décrite dans ce document. Le retour d'informations des utilisateurs de Tamanoir dont nous présentons les projets dans ce document a souvent orienté la mise en œuvre des concepts proposés par notre architecture.

1.3 Plan du manuscrit

Le chapitre 2 rappelle les fonctionnalités des équipements réseau existant, propose une définition complète des réseaux actifs et fait un bilan exhaustif des travaux qui se sont concentrés sur le problème de la performance. Le chapitre 3 introduit notre proposition d'architecture de réseau actif et nœud actif à la fois générique et haute performance. Le chapitre 4 décrit une mise en œuvre de cette architecture sous la forme de l'environnement Tamanoir. Le chapitre 5 présente quelques services actifs Tamanoir ainsi que des outils de tests et de gestion d'une infrastructure active déployée dans un réseau longue distance. Le chapitre 6 valide expérimentalement la mise en œuvre de notre architecture. Le chapitre 7 clos ce document par des conclusions sur les contributions de cette thèse et diverses perspectives de recherches et développement pour la poursuite de ces travaux. Une annexe propose un complément d'information sur le langage Java employé dans un contexte haute performance.

Chapitre 2

État de l'art sur les réseaux actifs

2.1 Introduction

Ce chapitre propose dans une première section un bref rappel sur le mode de fonctionnement des principaux équipements rencontrés sur les réseaux de transport de données modernes comme l'Internet aujourd'hui. Nous rappelons ensuite qu'il existe une hiérarchie dans les équipements de routage. Nous définissons alors ce qu'est un réseau actif. Puis nous présentons quelques projets reconnus, ayant proposé de nouvelles architectures de réseau actif générique ; suivra une liste exhaustive de propositions d'environnements d'exécutions actifs hautes performances. Nous énumérons ensuite les quelques initiatives industrielles disponibles dans le domaine des réseaux actifs en général. Enfin nous discutons l'intérêt des réseaux actifs ainsi que le point de vue industriel dans ce domaine, avant de conclure ce chapitre.

2.2 Rappel sur les principaux équipements réseaux actuels

Cette section permet de rappeler brièvement l'usage et le principe de fonctionnement d'équipements réseaux, tels que les routeurs, les commutateurs, les passerelles (NAT), les pare-feu et les mandataires (*proxy*).

Le **commutateur** est typiquement utilisé dans un réseau local (LAN) haut débit. Il prend ses décisions de routage en examinant l'en-tête de niveau 2 (adresse *MAC*). Il utilise une table de retransmission mise à jour grâce à un processus d'auto apprentissage. Les fonctions de commutations sont mises en œuvre de façon matérielle afin de rediriger les paquets à la vitesse du lien sans les ralentir. Les commutateurs actuels utilisent généralement des architectures distribuées mises en œuvre sur des ASIC afin de réduire la latence de *store and forward* entre 5 et 50 μs. Pour la plupart des commutateurs Ethernet, c'est le temps de sérialisation (ou temps de transmission) qui est le principal responsable de cette latence. Le tableau 2.1 donne un ordre d'idée des latences moyennes constatées pour des réseaux Ethernet à 10Mbps, 100Mbps et 1Gbps pour un paquet de 1500 octets [4]. Si un paquet traverse plusieurs équipements avant d'atteindre sa destination, ce délai de transmission est ajouté pour chaque passage d'un équipement à l'autre le long du chemin.

Le **routeur** est généralement utilisé pour connecter différents réseaux locaux (LAN). Contrairement au commutateur de niveau 2 (*L2*), le routeur ignore l'en-tête *MAC*. Il prend sa décision de routage en fonction de l'en-tête IP c'est à dire de niveau réseau (*network layer,*

10 Mbps	1 ms
100 Mbps	100 μs
1 Gbps	10 μs

TAB. 2.1 – Latences induite par des commutateurs Ethernet.

L3). Son rôle est donc d'extraire l'en-tête de niveau 3 du paquet, trouver une adresse de destination qui correspond dans sa table de routage, identifier une nouvelle adresse *MAC* pour le paquet à partir d'un cache ARP, encapsuler le paquet IP dans une trame Ethernet (avec le nouvel en-tête *MAC*), et finalement transmettre le paquet sur le port associé à l'adresse de destination dans la table de routage. La table de routage est mise à jour (ou publiée) par une interface en ligne de commande ou par des messages provenant des routeurs voisins. Historiquement ces fonctions étaient mises en œuvre à l'aide de logiciels mais les progrès dans le domaine des composants VLSI (*Very Large Scale Integration*) ont permis de mettre en œuvre des fonctions de routage de niveau 3 en matériel, offrant des performances proches de celles du lien.

Une **passerelle NAT** (*Network Address Translation*) ou de translation d'adresse permet d'interconnecter un réseau privé à un autre réseau comme Internet. Le principal usage de cette passerelle est de pallier au manque critique d'adresses IPv4 disponibles. Des plages d'adresses ont été réservées pour un usage privé uniquement (RFC 1918)[5] (ex : 192.168.x.x ou 10.x.x.x). Ces adresses ne sont pas routables sur un réseau public comme Internet ; on utilise donc la translation d'adresse pour permettre aux machines d'un réseau privé d'accéder à Internet. La connexion est nécessairement initiée par l'une des machines du réseau privé car le réseau public ne peut accéder qu'à l'adresse publique utilisée par la passerelle. Par conséquent les machines du réseau privé sont protégées des attaques extérieures. Le principe de fonctionnement repose sur le remplacement à la volée des champs d'adresses dans les paquets qui sont destinés à un autre réseau. Les NAT évolués sont aujourd'hui capable non seulement de modifier les champs d'adresses mais aussi d'observer les couches supérieurs (transport) pour récupérer suffisamment d'informations afin de déterminer le contexte de l'application (ex : session FTP).

Un **pare-feu** est une passerelle qui protège un réseau local des attaques/intrusions provenant de l'Internet. Il permet de contrôler l'accès au réseau des applications installées sur une machine, de repérer les connexions suspectes d'une machine et également de les empêcher. On peut voir le pare-feu comme un système permettant de filtrer les paquets de données échangés avec le réseau. C'est un système logiciel et parfois matériel constituant un intermédiaire entre le réseau privé et le monde extérieur. Un pare-feu est paramétré avec un ensemble de règles prédéfinies permettant soit d'autoriser uniquement les communications ayant été explicitement autorisées, soit d'empêcher les échanges qui ont été explicitement interdits. Le principe de fonctionnement repose sur le filtrage de l'en-tête des paquets qui contient l'adresse IP de la machine source, l'adresse IP de la machine destination, le nom du protocole de transport employé, un numéro de port source et un numéro de port destination souvent associé à un service ou une application réseau. Le filtrage applicatif permet de filtrer les communications application par application. Il suppose donc une connaissance de l'application, et notamment la manière dont elle structure les données échangées. Un pare-feu effectuant un filtrage applicatif est appelé passerelle applicative car elle permet de relayer des informations entre deux réseaux en effectuant un filtrage fin au niveau du contenu des paquets échangés.

Un **mandataire** (proxy) est un mécanisme qui permet de réduire le temps de réponse d'un service réseau. Un proxy permet d'accéder à un service réseau de manière indirect. Les accès réseau se font par son intermédiaire. Les deux principales raisons d'utiliser un proxy sont : la réduction de bande passante et l'amélioration de la sécurité. L'usage de bande passante peut-être réduit par le mécanisme de cache du proxy. Dans le cas d'un usage Web, si plusieurs utilisateurs décident de visualiser une même page, celle-ci sera télé-chargée une unique fois à travers le réseau. Malheureusement toutes les pages ne sont pas cachables car certaines sont construites dynamiquement lors du télé-chargement. Mais il est en général possible de cacher les images et photos. La sécurité est améliorée car seul l'ordinateur exécutant le proxy peut-être autorisé à accéder directement au réseau extérieur. Cette machine doit donc être scrupuleusement sécurisée, mais il est plus facile de sécuriser une unique machine qu'un parc complet de machines.

Contrairement aux autres éléments d'un réseau (commutateur, routeur, passerelle et pare-feu décrits ci-dessus) le proxy n'attrape pas les paquets à la volée. Une véritable connexion est établie entre l'application d'un poste client et le proxy. D'un point de vu transport les paquets lui sont destinés. Les autres équipements sont là soit pour router, modifier des en-têtes ou bloquer des paquets. Pour réaliser ces opérations ils n'utilisent que les couches basses des protocoles de communications. Un routeur comme un commutateur doit examiner l'en-tête d'un paquet pour le rediriger vers le port ou connecteur de destination adéquat. Le proxy lui doit étudier le contenu applicatif du paquet pour prendre une décision.

2.3 Une hiérarchie dans les équipements de routage

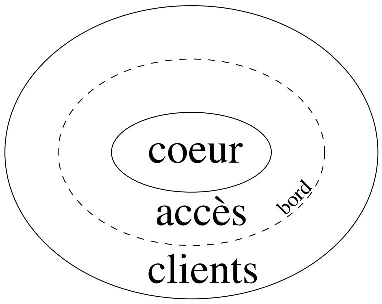


FIG. 2.1 – Hiérarchie de routeurs.

Le positionnement des équipements actifs dans un réseau existant n'est pas trivial. Un réseau longue distance comme l'Internet exploite différentes catégories de routeurs. Comme le montre la figure 2.1 on y trouve des routeurs d'accès, des routeurs de bords et finalement des routeurs de cœur. Le plus approprié pour le traitement est vraisemblablement le **routeur d'accès** car c'est lui qui rencontre les débits les moins importants et une grande diversité de supports de données. Malheureusement il a une connaissance limitée du réseau ou alors très similaire à celle que peut avoir un client terminal. A l'opposé un **routeur de cœur** a par sa situation une très bonne connaissance de l'état du réseau. Mais son objectif est de router de grandes quantités de données le plus rapidement possible. Les réseaux qu'il interconnecte fonctionne généralement à très haut débits, de l'ordre de plusieurs Gigabit par seconde. De plus, ces réseaux généralement tout optique obligeraient à convertir le flux optique en un flux électronique avant de pouvoir traiter les données et de les reconvertir sous forme optique avant de les re-émettre. Donc, c'est le **routeur de bord** qui semble offrir le meilleur compromis si on lui permet de traiter une partie des paquets (ceux qui sont actifs) sans ralentir le reste du trafic (paquets passif). Il bénéficie d'une position stratégique dans le réseau.

Bien que nous puissions déployer des services actifs sur tous les équipements réseau, il faut veiller à ne pas instancier un service actif de la même manière sur tous les équipements. Un service actif peut se voir distribuer de différente manière sur les différents nœuds. On peut imaginer une grand partie de la complexité localisée sur les équipement d'accès et de bord et le reste du traitement réalisé sur l'équipement de cœur, le tout agissant sur l'application complète.

2.4 Introduction aux réseaux actifs

Le concept de réseau actif est né en 1996 au MIT aux États-Unis, sur une proposition de David Tennennhouse et David Wetherall[6]. Il s'est rapidement étendu à l'Europe et constitue aujourd'hui un sujet de recherche universitaire et industriel très actif. En l'an 2000, on recense plus de 50 projets de recherche qui travaillent sur ce thème et regroupent plus de 80 laboratoires universitaires et industriels dans le monde. L'apport supposé en dynamicité des réseaux actifs est tel que de nombreux industriels (Alcatel, FT R&D, Thales,...) et même des organismes gouvernementaux (DARPA [1, 7]) jugent utile d'investir dans ce domaine de recherche. Bien que de nombreux progrès aient été réalisés ces quelques dernières années pour rendre la configuration des équipements des réseaux de données plus souple, ceux-ci restent relativement figés et spécifiques à un type d'application, comme la téléphonie par exemple. L'extensibilité de ces équipements reste limitée. Ils fournissent uniquement les services pour lesquels ils ont été conçus.

Or les opérateurs et fournisseurs de services ont un besoin crucial de dynamicité pour répondre dans un délai très court aux besoins des usagers. De nouvelles applications comme la téléphonie sur IP, la diffusion de radio ou flux vidéo issue de canaux TV, les services de cotations boursières ou encore d'achats aux enchères en ligne sur l'Internet existent déjà. Malheureusement, les services de base fournis par les protocoles existant du réseau sont mal adaptés à ces applications d'un genre nouveau et avec des exigences particulières. Pour y remédier on voit apparaître une multitude de protocoles de contrôle et de réservation de ressources (MPLS (*MultiProtocol Label Switching*), DiffServ, RSVP (*ReSerVation Protocol*),...) mais les constructeurs d'équipements (équipementiers) ne peuvent pas intégrer tous les nouveaux protocoles dans leurs matériels en un temps adapté aux besoins des usagers. Des solutions apparurent véritablement à partir de 1995 avec une focalisation sur les **réseaux programmables**.

> *Un réseau programmable est un réseau de transmission de données ouvert et extensible disposant d'une infrastructure dédiée à l'intégration et à la mise en œuvre rapide de nouveaux services sur l'ensemble de ses composants [6]*

Un réseau traditionnel (ou passif) est un réseau de transport de données qui possède un nombre restreint et fixé de services implantés dans les équipements et qui n'offre aucun moyen d'en ajouter. Par conséquent il est impossible de modifier dynamiquement le comportement global du réseau. Au contraire, un réseau programmable est un réseau de transport de données étendu par un environnement de programmation à l'échelle du réseau comportant un modèle de programmation des services, des mécanismes de déploiements et un Environnement d'Exécution (EE). Une initiative est apparue en 1997 au sein de l'IEEE sous la forme du projet P1520 (*Programmable Interface for Networks*) [8].

En 1996, sur une proposition de David Tennennhouse et David Wetherall est apparu le concept de **réseaux actifs**[9].

> *Un réseau actif est un réseau dans lequel les composants dans les différents plans (signalisation, supervision, données) sont programmables dynamiquement par des entités tierces (opérateurs, fournisseurs de services,. applications, usagers)[6]*

Cette approche est plutôt issue des travaux sur l'insertion de services applicatifs dans le réseau Internet, par opposition aux réseaux programmables qui sont issus de travaux sur la signalisation pure dans les réseaux de télécommunications de type ATM. Elle étend le concept de programmation en ouvrant les équipements de l'ensemble du réseau et en partant du principe que tout usager peut concevoir et déployer à l'aide d'interface de programmation (API) de nouveaux services. Ces nouveaux services peuvent alors être utilisés de manière dynamique dans le réseau.

Dans la famille des réseaux actifs, on trouve deux approches pour la mise à jour des équipements actifs : l'une se base sur l'approche **paquet actif**, l'autre sur une approche **nœud actif**. L'approche paquet actif permet de transporter le code des services dans le même flux que les données traitées par ce service. On parle aussi d'**approche intégré** ou de capsules [9, 10, 11, 12]. Classiquement le code du service est émis avant l'arrivée du flux de paquets de données. Mais l'émission du code est effectuée dans le flux usager. Et dans un cas extrême, chaque paquet de données contient le code du service à y appliquer. Généralement la vie du service est liée à celle du trafic affecté. Dans l'approche nœud actif les services sont également déployés dynamiquement sur les nœuds du réseau mais hors du flux de données utilisateurs. Cela se rapproche fortement du concept de réseau programmable. La différence avec ce dernier repose sur le fait que dans l'approche nœud actif, les composants de service sont déployés physiquement sur les nœuds du réseau, le plus souvent grâce à des techniques de code mobile. Dans les réseaux programmables, la logique de service et son implantation ne nécessitent la plupart du temps pas de déploiement sur les nœuds, celles-ci étant exécutées au sein de l'environnement de programmation du réseau. Cette programmation externe est aussi appelée **approche discrète**[13, 14]. Conceptuellement, l'approche paquet actif est beaucoup plus ouverte que l'approche nœud actif. Malheureusement dans la première approche les codes déployés devront pour des raisons évidentes de performances être de plus petite taille et donc leur usage sera restreint à un nombre de situations plus limitées.

La figure 2.2 illustre les deux types de structure de paquets qui sont employés dans l'une ou l'autre des approches. Dans la première approche, paquet actif, on parle de capsules qui inclus le code du service à appliquer sur les données. Dans la seconde approche, nœ actif, on parle de paquet actif qui ne transporte en plus des données qu'une référence au service à appliquer à ces données.

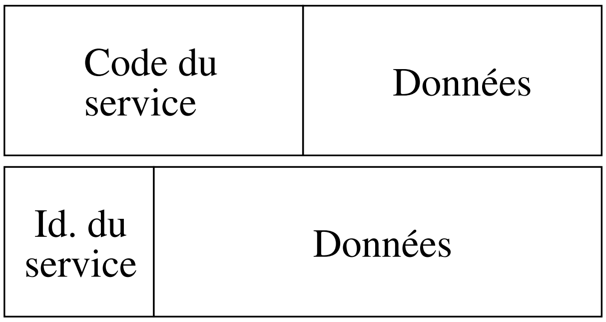


FIG. 2.2 – Capsule active et paquet actif.

2.5 Les différentes approches pour concevoir un réseau actif générique

Dans cette section nous dressons un état de l'art non exhaustif des projets dans le domaine des réseaux actifs en général. Il existe diverses synthèses qui proposent un bilan de l'avancement des recherches dans le domaine des réseaux actifs et qui dressent un état de l'art exhaustif, par exemple [6]. Nous nous focalisons ici sur les travaux ayant particulièrement influencé nos recherches.

2.5.1 ActiveIP : Le précurseur

Active IP est l'un des premiers projets ayant proposé un prototype de réseau actif [9]. Il a été développé en 1996 au MIT par David Wetherall. Le principe consiste à placer du code TCL (*Tool Command Language*) étendu par des primitives de manipulation de paquets IP dans l'option 63 de l'en-tête IP. Lorsque le paquet traverse un routeur actif, le code peut modifier le contenu du paquet, reconstruire un ou plusieurs paquets et les émettre. Ce projet exploite l'option IP Router Alert d'IPv4 afin que des équipements actifs puissent détecter les paquets à traiter. Si le paquet traverse un équipement ne fournissant pas de support actif, le paquet est transmis suivant un routage par défaut. Les primitives proposées par le langage de programmation sont de trois types :

- des fonctions d'accès à l'environnement du nœud pour la récupération de l'adresse IP, de la date, du MTU,...
- des fonctions de manipulations de paquets pour modifier les données d'un paquet, l'adresse source, l'adresse destination, la longueur, le protocole encapsulé et le TTL.
- des fonctions de contrôle pour l'envoi de paquets IP et UDP, l'abandon de paquets, la réponse à un paquet et l'évaluation d'un paquet.

De plus, toutes les primitives de contrôle de *safe-TCL* sont supportées. Ce projet a permis d'identifier deux limites : l'absence de partage de ressource dans un nœud et l'usage de paquets actifs purs (chaque paquet contient son code) ne permettent pas de cacher des services dans le nœud ce qui est dommageable pour un flux de données uniforme, dont le code transporté dans chaque paquet est identique. Pourtant, cela n'enlève rien à l'aspect novateur de son approche et qui grâce à la création de ce premier prototype propose une preuve que le concept de réseau actif fonctionne.

2.5.2 ANTS

L'architecture de réseau actif appelée ANTS (*Active Node Transfer System*) est le fruit de la suite des travaux sur le projet *ActiveIP*, menés au MIT et dont le principal investigateur reste David Wetherall. Il demeure le prototype d'EE actif expérimental le plus utilisé par la communauté [12]. Cette architecture offre la capacité à une application de déployer dynamiquement dans le réseau les services et les protocoles qu'elle utilise sur tous les routeurs actifs traversés par le flux de données. Le déploiement dynamique des protocoles est réalisé au sein du flux de données de l'application (approche intégré). Les principaux composants de l'architecture ANTS sont :

- le *protocole* qui définit de manière conceptuelle le traitement à effectuer sur un flux donné. Un protocole est défini par un identificateur (signature MD5 (*message digest*) du code qui l'implante), un ensemble de classes qui en définissent le comportement (groupe de code) ainsi que les unités de distribution de ce code appelé *capsules*.
- la *capsule* est l'unité de base de programmation du réseau. Elle est utilisée pour véhiculer les données de l'application entre les nœuds actifs ainsi que pour transporter le code d'un protocole à déployer sur les nœuds. La capsule contient une adresse source et destination ainsi qu'un champ TTL, un champ de version d'ANTS, un identificateur de méthodes déterminant de manière unique la primitive d'évaluation et le protocole associé, un champ comportant l'adresse du dernier nœud actif traversé et enfin, les données sur lesquelles le protocole référencé dans l'en-tête par la signature MD5 de sa méthode d'évaluation sera appliqué. La figure 2.3 illustre une capsule ANTS. Dans une capsule de transfert de code, les données sont le *bytecode* Java du protocole. Un protocole peut être transporté dans plusieurs capsules successives.
- le *nœud actif* est l'environnement d'exécution (EE). Il permet la réception, le traitement et l'envoi d'une capsule suivant le protocole prédéfini par l'application ayant émis la capsule.

Adresse source	Adresse dest.	TTL	Version	Type	Adresse préc.	Entetes de type	Données

FIG. 2.3 – Une capsule ANTS.

Le protocole définit l'unité d'isolation des traitements dans un nœud. Cela signifie que les traitements au sein d'un protocole partagent le même espace de nommage et le même espace mémoire et ne peuvent en aucun cas avoir accès à l'espace des autres protocoles ni générer des capsules dans d'autres protocoles. Ceci permet à des capsules d'un même protocole de s'échanger des données lors d'un passage dans un nœud mais isole les protocoles entre eux. Lorsque le flux de capsules traverse le premier nœud qui ne possède pas le protocole adapté, celui-ci est télé-chargé à partir de l'application émettrice du flux. Un groupe de codes est un ensemble de capsules contenant le code complet d'un protocole. Pour les nœuds suivants qui sont traversés par le flux et qui ne possèdent pas non plus le protocole adapté le groupe de code est télé-chargé du nœud précédent. Donc un nœud ayant à traiter une capsule avec un protocole encore non disponible dans son cache de protocoles pourra ainsi par récurrence (de proche en proche) assurer le déploiement d'un protocole. Le prototype ANTS a été entièrement écrit en Java et fonctionne au-dessus d'UDP. Il offre une approche simple et compréhensible du concept de réseau actif. Son principal avantage réside dans son modèle de déploiement de code uniquement sur le chemin du flux de données et son mécanisme de cache de code permettant aux capsules de ne pas avoir à transporter chacune son code. La disponibilité d'un cache permet également aux capsules de déposer des informations (d'états) pour les suivantes ce qui peut être utile pour de nombreux protocoles. En contrepartie ANTS est limité sur différents aspects. Le manque de performance de son implantation et son fonctionnement unique en mode tunnel UDP (liaison point à point UDP entre chaque nœud) qui pose des problèmes de fiabilité notamment pour le déploiement des protocoles. On peut également reprocher à l'implémentation son incapacité à supporter plusieurs applications en parallèle. Enfin les applications doivent être exécutées sur un nœud qui fait partie des nœuds actifs déployés. Une synthèse complète réalisée par les concepteurs d'ANTS est donnée dans [15].

2.5.3 Systèmes d'exploitations spécialisés pour les réseaux actifs

Cette section décrit les principaux projets de création et mise au point de systèmes d'exploitations optimisés pour un usage dans le domaine des réseaux actifs. Il est important que le système d'exploitation d'un nœud actif n'entrave pas son bon fonctionnement. Il ne doit pas réduire la vitesse de la remontée des paquets de la carte d'interface réseau vers l'environnement d'exécution. Sa disponibilité doit être très importante pour répondre aux exigences de performances requises et à la gestion fine des ressources. Ainsi pour accéder à la mémoire de l'équipement, aux interfaces de communications et aux ressources de traitements de manière efficace il existe des propositions de construire une interface située entre l'environnement d'exécution et le système. Cette interface est généralement appelée *NodeOS*. Elle définit un ensemble de fonctions d'accès et de gestions des ressources du nœud actif.

CANEs [16], (*Composable Active Network Elements*), est un projet dont l'objectif est de concevoir une architecture cohérente pour les réseaux actifs incluant une terminologie consistante, les fonctionnalités minimales indispensables, et la spécification d'une interface. CANEs est un environnement d'exécution exécuté au dessus de *Bowman*[17], qui met en œuvre seulement un sous ensemble de l'interface d'un *NodeOS*. Les programmes CANES sont développés dans un langage spécial appelé CUI (CANES User Interface language).

Scout [18] est un système d'exploitation spécialisé pour les communications qui offre une forte flexibilité de configuration et qui permet une gestion fine des ressources CPU, mémoire et bande passante allouées. Le système d'exploitation est entièrement basé sur des modules, un graphe de ces modules et des chemins. Le module est le composant logiciel de base qui propose un code de traitement sur des flux de données. Le graphe est une composition de modules exprimant les dépendances entre modules. Un chemin est la définition d'un circuit de transit de flux de données dans le système d'exploitation. La création ou la destruction d'un chemin peuvent être réalisées dynamiquement. C'est l'ensemble de ces composants qui permet la génération d'un système d'exploitation optimisé pour le traitement des flux. Scout est disponible pour les architectures *Intel Pentium* et *DEC*.

Joust [19] est une extension Java construite sur le système d'exploitation Scout. Cette extension adjoint à Scout une machine virtuelle Java optimisée fournie avec la bibliothèque de gestion de graphique Abstract Window Toolkit (AWT) et un compilateur Just In Time (JIT). Cette machine virtuelle Java contient également une API étendue intégrant les concepts de modules et de chemins propres à Scout. Afin d'obtenir des performances adaptées aux flux multimédia dans l'OS, la machine virtuelle comporte également de nombreuses optimisations sur la gestion des *threads* et des *exceptions* par rapport à celle de SUN. Le défaut de l'architecture Joust, bien que conçue pour les communications est qu'elle ne permet pas de gestion de ressources.

Exokernel [20] est un système d'exploitation qui propose une gestion fine des ressources, sûre et ouverte aux applications. Le principe consiste à sortir du noyau toutes les fonctions de gestion de ressources et de les mettre à disposition dans des bibliothèques utilisateurs. Le noyau ne reste alors en charge que de la protection des ressources à un niveau très bas (ex : bloc disque). Ainsi les applications de niveau utilisateurs peuvent coder leurs bibliothèques système de gestion des ressources pour leurs propres besoins. Exokernel est fourni avec des bibliothèques par défaut et le noyau offre une vision très primaire des ressources matérielles, il n'y a pas d'abstraction. C'est aux applications, via les bibliothèques de systèmes spécialisés, de se construire leurs propres abstractions. Les deux principaux avantages de ce système sont sa flexibilité et les performances qu'il fournit à travers l'utilisation de bibliothèques systèmes adaptées à un traitement donné. Pour une architecture donnée l'accélération constatée est de 5 par rapport à un système d'exploitation Unix standard. En contre partie l'écriture de telles bibliothèques demande beaucoup de temps et de compétences. Exokernel existe pour les architectures *x86* et *MIPS*. AMP[21] par exemple est construit au dessus d'*Exokernel*. Son objectif est de fournir une plate-forme sécurisée sur laquelle des environnements d'exécution et des applications actives peuvent s'exécuter, sans compromettre l'efficacité d'exécution.

JANOS (*Java-oriented Active Network Operating System*)[22] est également un système d'exploitation pour les nœuds des réseaux actifs. Son rôle principal est de gérer et contrôler les applications actives écrites en Java pour lesquelles il n'y a pas de confiance établie.

2.6 Propositions d'environnements d'exécutions actifs hautes performances

Cette section propose de décrire de manière exhaustive les travaux réalisés dans le domaine des réseaux actifs haute performance.

2.6.1 PAN : Practical Active Network

Le projet PAN pour *Practical Active Networking* [23] est né dans les laboratoires du MIT. C'est un projet inspiré de ANTS mais exclusivement tourné vers la conception d'un nœud actif haute performance qui offre la possibilité de mener de réelles expérimentations. Le projet PAN étudie le chemin critique des données et essaie de supprimer les sources potentielles de coûts supplémentaires qui sont : les recopies mémoire, l'interprétation de code, le chargement de code et le passage de données de l'espace noyau à l'espace utilisateur. Écrit en C, PAN obtient de très bonnes performances brutes. Il en existe deux mises en œuvre : l'une s'exécute dans l'espace utilisateur, l'autre dans l'espace noyau, sous forme de modules. Les mesures ont été faite sur UDP pour des paquets dont la taille varie de 128 à 1500 octets. Les meilleures performances sont obtenues pour une exécution exclusivement dans l'espace noyau. Les résultats expérimentaux montrent que pour des petits paquets (128 octets) la recopie coûte peu. Pour des paquets de 1500 octets l'EE natif exécuté dans l'espace utilisateur coûte 5 fois plus que la version noyau. La version exécutée dans le noyau est capable de saturer un lien *Fast Ethernet* avec des paquets de 1500 octets et un surcoût de seulement 13 % pour traiter chaque paquet. Ces résultats sont obtenus grâce à des mécanismes limitant le nombre de copies en mémoire, un traitement des paquets lorsque cela est strictement nécessaire et enfin, par un usage intensif de code natif. L'utilisation de code natif pose un problème pour la portabilité d'un service qui doit pouvoir s'exécuter sur des équipements réseaux aux architectures hétérogènes et pose également des problèmes de sûreté et de sécurité.

2.6.2 ANN : Active Network Node

Le projet ANN[24] est aussi focalisé sur les réseaux actifs haute performance. Ce projet se destine à fournir un support de réseau Gigabit ATM. La conception matérielle de ce projet est réalisée à l'université de Washington et le serveur de code ainsi que l'environnement sont développés conjointement à l'ETHZ (Eidgenössische Technische Hochschule Zürich - Swiss Federal Institute of Technology Zurich) et à l'Université de Washington.

La mise en œuvre de ANN repose sur une approche matérielle connectée à un réseau ATM, un système d'exploitation performant adapté à l'architecture matérielle choisie (décrite ci-dessous), et un EE spécifique nécessaire à l'architecture et à la gestion du télé-chargement de code actif sur un nœud. Leur approche consiste à traiter des flux de paquets actifs et non pas des paquets isolés. Chaque paquet contient une référence à un module actif, appelé service, stocké sur un serveur de codes de confiance appelée DAN pour *Distributed code caching for Active Networks* [14]. Les modules sont liés dynamiquement et exécutés en code natif sur le routeur. Ce projet supporte ANTS [15] qui fournit l'environnement souple pour le prototypage de modules. ANN démontre qu'il est important de lier étroitement la capacité de traitement et le réseau ainsi que de distribuer le traitement sur les CPU disponibles. ANN utilise des modules appelés ANPE pour *Active Network Processing Elements* constitué d'un processeur (Pentium), de mémoire et d'un APIC (interface réseau). Un ensemble d'ANPE constitue un Active Network Node (ANN). Les ANN sont connectés aux commutateurs ATM.

2.6.3 SARA : Routeur et assistant

Les travaux sur SARA [25, 26] sont issus de l'*Universidad Carlos III* de Madrid pour le projet IST GCAP. L'objectif est de fournir un cadre supportant des fonctions de base pour les réseaux actifs, valide pour IPv4 et IPv6, et réaliste dans un contexte industriel. Afin de mettre au point un routeur actif utilisable dans un contexte industriel SARA utilise un vrai routeur industriel. Ce projet n'emploie pas les processeurs dédiés du routeur pour exécuter l'EE et les AA. Le routeur utilise un processeur spécialisé pour le routage et le *management*, des processeurs spécialisés pour les interfaces réseaux et finalement des processeurs classiques pour les AA afin de s'adapter aux contraintes des services. L'exécution d'une application génère beaucoup d'interruptions (*irq*). Afin de réduire le nombre d'interruptions sur le routeur, l'EE et les AA sont exécutées sur un autre système connecté au routeur industriel. On parle d'assistant de routeur. La figure 2.4 illustre un routeur industriel auquel on connecte un assistant équipé d'un processeur classique. Le traitement actif est donc réalisé hors du routeur. Il est délégué à un assistant ou "pool" d'assistants directement attachés au routeur par un réseau local rapide. Le coût supplémentaire du routeur se limite à identifier et détourner les paquets actifs vers un assistant. Pour cela le projet utilise l'option *IP Router Alert* disponible dans l'en-tête IPv4 et IPv6 qui marque le paquet comme étant à traiter. Le paquet actif doit transporter une référence au code et un certificat de sécurité. Le code peut être récupéré grâce à un protocole standard (ex : `https`) ou propriétaire. Les utilisateurs ne sont pas autorisés à injecter eux-mêmes leur code mais ont simplement le droit d'exécuter des codes certifiés.

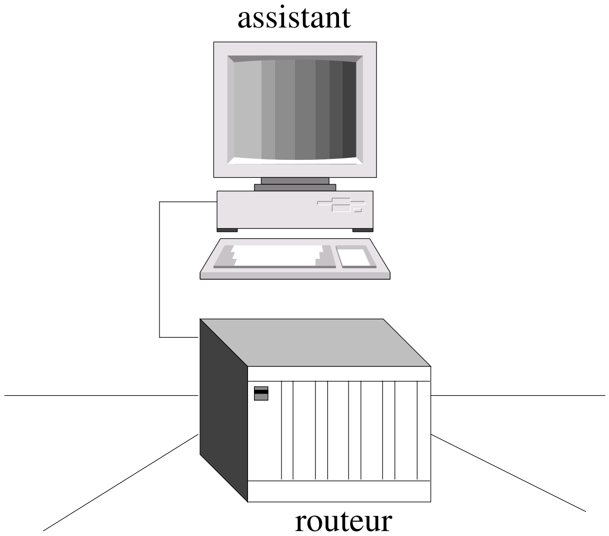


FIG. 2.4 – Sara : Routeur industriel avec son assistant qui exécute le code sur les paquets actifs.

Cette approche possède de nombreux avantages. Le fait de connecter des assistants maîtrisés à un routeur dont on ignore souvent le fonctionnement interne précis, conduit à un coût de développement plus faible. Les paquets classiques (sans l'option IP Router Alert activé) ne sont absolument pas pénalisés. Il n'y a pas de surcoût de routage non plus pour les paquets actifs car tous les paquets doivent être examinés pour être routés. Le faible couplage entre le routeur et ses assistants conduit à un contrôle de ressource moins critique. Enfin l'architecture semble bien supporter le passage à l'échelle. En contre partie cette architecture peut introduire une latence supérieure à un routeur actif « natif » à moins d'interconnecter le routeur à ses assistants avec un réseau très rapide à faible latence. Un autre problème pourrait provenir du faible couplage entre le routeur et ses assistants. Cela peut détériorer la qualité des informations temps réel (ex : charge des interfaces) provenant du routeur et susceptible d'être utile à une application active. D'un point de vue mise en œuvre, le projet SARA (*Simple Active Router Assistant*) a été écrit en Java et C. Si l'AA n'est pas présente elle est télé-chargée à partir d'un serveur de codes. Les AA sont des threads natifs capable de filtrer et envoyer des paquets actifs, consulter le routeur (ses états, sa *MIB* (*Management Information Base*)), et utiliser des sockets enrichies. La durée de vie d'une AA est déterminée par les paquets actifs. Le projet SARA a été confronté au problème de manque de contrôle des sockets Java. Il est par exemple impossible d'activer l'option IP Router Alert, utiliser IPv6 ou contrôler la couche 3 avec une socket Java classique. C'est pourquoi ils ont du adjoindre au projet des fragments de code C au code Java grâce à l'interface JNI. Enfin, les communications entre le routeur et les assistants sont encapsulées dans des datagrammes UDP. La plate-forme de test

est un routeur Ericsson Telebit AXI462 avec pour assistant un PIII/733MHz (SuSE 7.2/noyau 2.4.12) interconnecté par un réseau Fast Ethernet (100Mbps). L'AA employée, exécutée dans une JVM 1.3.0, se contente de filtrer et transmettre les paquets reçus. Pour des paquets de 512 octets le projet obtient 2.5 Mbps et environ 33Mbps pour des paquets de 5120 octets. En conclusion grâce à cette approche le projet SARA a réussi à suivre l'axiome donné par ses auteurs : « ne remplace pas les fonctionnalités d'un routeur, améliore les ».

2.6.4 CLARA : Nœud actif basé sur une grappe

Le projet CLARA a été mené par une équipe de la société NEC, partie du constat que la première cause de dégradation d'un service est la perte de bande passante. CLARA [27] est un prototype d'architecture de nœud de routage basé sur un modèle de grappe issue du projet *Journey* dont l'objectif est de déployer des routeurs auxquels on adjoint des capacités de traitements. Les unités de données sont appelées *media unit*. Elles pourront contenir par exemple un GoP (*Group of Picture*) d'un flux MPEG. Chaque média unit est considéré comme indépendante. Le modèle employé permet de traiter les média unit en fonction des conditions et disponibilités locales. Les décisions sont prises indépendamment des autres routeurs. Il n'y a pas de message de contrôle entre les routeurs pour traiter un flux particulier. Ce modèle ne garantit pas que chaque *media unit* arrivera à destination traité. Il reste dans l'esprit du routage *best effort* du réseau IP. Pour déterminer l'état traité ou non traité d'un média unit CLARA utilise l'option *IP Router Alert*. Traité : l'option IP RA est désactivée et le paquet est routé directement ; Non traité : l'option IP RA est activée et le paquet est pris en charge par CLARA pour savoir s'il va pouvoir être traité.

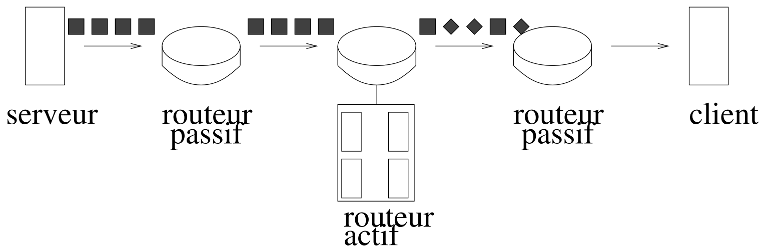


FIG. 2.5 – Clara : routeur actif déployé sur une grappe de machines.

La figure 2.5 représente un flux de données émis par un serveur (à gauche) vers un client (à droite). Les *media unit* sont représentés par des carrés lorsqu'elles n'ont pas été traitées et par des losanges lorsqu'elles ont été traitées. L'architecture d'un nœud actif CLARA est constituée d'un PC destiné au routage relié par un réseau local très haut débit à d'autres PC dédiés exclusivement aux traitements (au nombre de 4 sur la figure). Les média unit sont distribuées sur les PC de traitements suivant un simple algorithme de tourniquet.

2.6.5 APE : Édition Active de paquets

APE[28] (*Active Packet Edition*), proposé par une équipe du NTT Network Innovation Laboratory, est constitué d'une double architecture. Une partie logicielle pour le traitement des paquets actifs et une partie matérielle pour une édition efficace des paquets. La partie matérielle est basée sur des FPGA (Field Programmable Gate Array) et des CAM (*Content Addressable Memory*). L'éditeur de paquet appelé "Gorilla" permet de classifier les paquets et/ou de leur appliquer des modifications en fonction de règles pré-établies. La partie logicielle de traitement des paquets actifs appelée "Chimp" permet de configurer dynamiquement l'éditeur de paquets en fonctions des instructions contenu dans les paquets actifs. Ce projet utilise le format ANEP (décrit dans la section 3.4.5.1). La partie logicielle gère le traitement actif complexe qui ne peut être effectué par l'éditeur de paquets. L'objectif de l'architecture APE est de fournir des mécanismes de reconfiguration très rapides pour des fonctions ou des applications relevant du plan transport comme la classification de paquets, la translation d'adresse (NAT), le mode tunnel, les communications de groupes (multicast) et les proxy. Ces fonctions n'impliquent que des traitements simples sur l'en-tête. Néanmoins elles doivent être appliquées suffisamment rapidement pour supporter la vitesse du lien. La reconfiguration est réalisée par une série de paquets actifs qui reconfigure dynamiquement une table de règles dans la CAM qui fournit la souplesse requise pour les réseaux actifs. L'originalité de cette approche réside dans le fait d'avoir découpé en deux le traitement : une partie rapide, et une partie lente. Par exemple la partie configuration des règles ne nécessite pas d'être très rapide. Bien que le projet APE supporte les applications actives qui nécessitent des traitements plus compliqués que de la simple édition de paquets, il n'essaye pas de les accélérer. Le projet prévoit un mécanisme de sécurité en mettant en place avant l'émission d'un flux un chemin protégé établi grâce à un premier paquet actif muni d'une clef. Enfin, les mesures de performances ont été réalisées avec un service simple de transmissions sur un lien Gigabit Ethernet d'un routeur Cisco 7500. Les résultats montrent qu'avec son support matériel dédié, APE soutient la vitesse du lien, soit plus de 900 Mbps.

2.6.6 Protocol Boosters : le projet P4

L'objectif du projet P4 [29] (*Programmable Protocol Processing Pipeline*) est d'augmenter les performances des protocoles en supprimant dynamiquement les traitements inutiles, car les membres du projet P4 ont constatés que les protocoles sont mis au point en prenant toujours en compte le pire cas. *Protocol boosters* fonctionne en optimisant dynamiquement le protocole de communication. P4 emploie une approche matérielle à base de FPGA pour obtenir la flexibilité nécessaire. P4 a démontré le bon fonctionnement de son prototype sur un lien ATM à 155 Mbps en accélérant le protocole TCP/IP dans un environnement perturbé. La méthode employé consiste à ajouter ou supprimer dynamiquement le FEC (*Forwarding Error Correction*).

2.6.7 Services noyau

Plusieurs projets ont étudié la possibilité de déployer des services actifs dans l'espace noyau. La première architecture de nœud actif capable d'accueillir des services actifs directement dans le noyau du système d'exploitation (Solaris) est décrite dans [30]. C'est une approche purement nœud actif. La référence à la fonction à appliquer est passée en paramètre dans les options IP. Cette référence doit être unique pour une fonction donnée. Chaque fonction peut être paramétré sur le nœud ou les paramètres peuvent être fournis dans l'en-tête de chaque paquet. Les nœud permettent d'accéder à certain de leurs états et possèdent une table avec des pointeurs sur le code des fonctions disponibles. Le nœud appelle alors la fonction de traitement à appliquer lorsqu'un paquet arrive. L'objectif de ce projet était d'obtenir de bonne performance pour l'exécution des services. Malgré cela, il manque à cette architecture un mécanisme de déploiement dynamique pour rendre le nœud actif facilement extensible.

Le projet PromethOS [31] utilise quand à lui Netfilter qui permet d'étendre l'architecture Linux standard en ajoutant en cours d'exécution des extensions pour le support de nouveaux composants. Même si l'interface est contrainte à celle de Netfilter, les performances de ce projet sont comparables à l'environnement réseau standard de Linux, seul une table de hashage de *look up* a été ajouté afin de gérer le temps des modules PromethOS.

2.6.8 Magician : Gestion et allocation des ressources

Le prototype de réseau actif *Magician* [32] développé dans l'université du Kansas a repris la plupart des concepts de l'approche ANTS avec des paquets actifs appelés *SmartPackets*. Ce projet utilise les mécanismes de sérialisation de Java 1.1, fournit un mécanisme de démultiplexage des paquets actifs (ANEP) pour supporter plusieurs EE sur un même nœud et surtout, offre une gestion plus fine des ressources d'un nœud en limitant l'accès au CPU et à la mémoire des threads. Pour cela le projet Magician a mis en œuvre un mécanisme d'ordonnancements des threads utilisateurs basés sur des files d'attentes appliquant plusieurs type de priorités : forte, standard ou faible. Ils fournissent également une classification des threads en fonction de leur activité sur un nœud (supervision : prioritaire mais peu de CPU ; services déployés sur nœud : durée de vie longue mais activité sur des périodes courtes ; paquets actifs : durée de vie courte et activité CPU intense sur la durée de vie). L'allocation de mémoire est contrôlée par le nombre d'appels à l'instruction d'allocation `new` de Java.

2.6.9 AMnet : projet Flexinet

AMnet [33] est originellement un projet de multicast actif, cependant l'aspect multicast n'est aujourd'hui qu'une composante du projet Flexinet parmi d'autres. Le projet fournit une plate-forme programmable et des applications pour : la communication de groupe avec des récepteurs hétérogènes, le renforcement de la sécurité et le support de la mobilité. Ce projet utilise des *services modules* mis à disposition sur des *Service Module Repositories* (SMR) administré (i.e avec authentification, autorisation et gestion de compte). Le code des services modules est distribué sous différents formats pour le support des multiples architectures matérielles des équipements actifs. Le SMR tient à jour les exigences en ressources (matériel, mémoire, débits, droits d'accès,...) des modules. Ces modules peuvent être assemblés pour fournir un service plus élaboré. De plus le projet définie deux catégories de services : les services *applicatifs*, exploité par les applications d'un client. Et les services *infrastructure* employés par l'administrateur réseau. Dans [34] une évaluation des performances a été réalisée sur trois services différents : multicast de MP3, compression `http` et transcodage vidéo pour PDA. Le premier service (multicast) qui n'effectue aucune opérations sur la charge utile obtient un débit de 68 Mbps entre un émetteur et un récepteur à travers un nœud actif connectés en ligne sur un réseau 100 Mbps (Ethernet). Le second service de compression `http` optimise le transfert de gros fichiers textes et trouve parfaitement sa place dans un réseau à faible débit, type sans-fils (11 Mbps). L'amélioration de la vitesse de transfert est d'environ 25% et atteint 40% pour les très gros fichiers. Enfin le troisième service d'adaptation vidéo pour *Palm* ne supporte à ce jour le traitement que d'un seul flux donc un client unique.

2.6.10 SNAP : *Safe and Nimble Active Packets*

Le projet SNAP [35] est développé au laboratoire DSL (*Distributed Systems Laboratory*) de l'Université de Pennsylvania, USA. Il propose un nouveau type de paquets actifs basé sur un nouveau langage bas niveau. Leur objectif est de fournir un peu plus de flexibilité à IP sans pour autant compromettre sa fiabilité, sa sécurité et son efficacité. Les résultats présentés dans [35] montrent que le projet atteint des performances très proche des routeurs IP logiciel.

2.6.11 NP : Network Processor

La plupart des projets de réseaux actifs sont basés sur un environnement d'exécution logiciel (excepté ANN et APE dans cette section). Ce que l'on appelle routeur actif est le plus souvent dans les faits (dans la phase d'expérimentation en tous cas) une station de travail configuré en tant que routeur et équipé de plusieurs cartes d'interface réseau. Les travaux de [36] se concentrent sur la conception d'un routeur actif haute performance architecturé autour de plusieurs processeurs d'un type nouveau : les *Network Processors*. Depuis quelques temps déjà, des compagnies comme IBM, Intel, Motorola et bien d'autres ont mis sur le marché ces « moteurs » de traitements de paquets programmables pour les routeurs. Ces *Network Processors* permettent de réaliser des traitements à partir de la couche de liaison de données *(data link layer)* jusqu'à la couche application. La plupart sont équipés de plusieurs unités de traitements (jusqu'à 16), cadencé entre 133 et 450MHz, multi-threadé et livré avec de la mémoire rapide DRAM (*Dynamic RAM*). Ils sont principalement optimisés aujourd'hui pour des applications de traitement des en-têtes de paquets. Ils se présentent comme des circuits ASIC (*Application Specific Integrated Circuit*) intégrant processeur, mémoire et Entrées/Sorties. La plupart exploite des processeurs de type RISC comme les MIPS, ARM ou PowerPC dont les caractéristiques sont améliorées par l'ajout d'instructions spécialisées, d'un support multi-thread, et de mécanismes de changements de contextes sans sur coût. Une autre approche consiste à utiliser des processeurs SIMD (*single instruction, multiple data*) généralement utilisés pour des applications scientifiques mais adaptés ici au contexte réseau.

2.7 Initiatives industrielles

Toutes les initiatives brièvement présentées ci-dessous sont généralement le fruit d'un travail en collaboration avec des laboratoires universitaires dont l'objectif est de s'associer à un industriel pour réaliser un transfert technologique.

- **6WIND** : Ce constructeur d'équipements réseau propose à ces clients des plate-formes de périphérie et d'accès intégrant l'ensemble des caractéristiques pour le déploiement de services IPv4 et IPv6[37]. Cette société très impliquée dans la promotion d'IPv6 sur le plan national comme international a participé à la spécification d'IPv6. Elle participe activement au forum IPv6 et à la *French IPv6 Task Force*. Elle est également impliquée dans de nombreux projet français et Européen, notamment le projet ANDROID dont les objectifs sont : la gestion des réseaux actifs, et le déploiement dynamique et automatique de VPN IPv6[38].
- **France Telecom R&D** : Le service de recherche et développement de l'opérateur national France Telecom possède une équipe chargé de diffuser la technologie des réseaux actifs en interne. Leurs intérêts sont concentrés sur les réseaux actifs hautes performances sécurisés.
- **Alcatel R&I** : En 2001, le service Recherche et Innovation d'Alcatel a démarré un projet appelé *Active Networks Seed Project*. Les objectifs de ce projet étaient de réaliser une veille technologique pour la société Alcatel dans le domaine des réseaux actifs et mettre en place des collaborations avec des laboratoires de recherche universitaire. Ils

ont également rédigé un *draft* intitulé *COPS client-type for Active Networks*[39]. Le projet est aujourd'hui terminé mais Alcatel reste impliquée dans tout ce qui est relatif à l'architecture des réseaux IP, à la qualité de services (QoS), aux réseaux actifs et à la localisation et négociation de ressources.
- **Thales communications** : est un groupe français qui propose une gamme complète de produits, services et systèmes d'information et de communications. Ils mettent au point les systèmes de communications militaire de demain. Leurs clients sont des gouvernements, des forces armées et des entreprises. Cette société a exprimé son intérêt dans ce domaine de recherche par sa présence active aux réunions de l'action spécifique Aspronet et son implication dans le projet Amarrage avec François Haettel.
- **General Electric** (Management Prediction) : *General Electric* (GE) s'intéresse à la technologie des réseaux actifs[40]. L'objectif des travaux est de développer un réseau actif capable de prédire son propre comportement et d'utiliser cette caractéristique pour développer une gestion des réseaux actifs prédictive. GE appelle cette algorithme AVNMP pour *Active Virtual Network Management Prediction.* Aujourd'hui GE propose une boîte à outils appelé *Atropos* téléchargeable qui repose sur le concept de l'AVNMP et permet donc d'injecter dans les nœuds actifs des modèles de composant réseau prédictifs. Ces algorithmes défini par les utilisateurs permettent de prédire l'état du système et de propager cet état efficacement à travers le réseau. Ces algorithmes permettent une réaction en temps réel [41].
- **NEC, NTT, Hitachi, ...** : L'intérêt porté au réseaux actifs par les grand groupe Japonais comme NEC, NTT ou Hitachi est indiscutable. La preuve est faite par un taux de présence important de représentants et acteurs de ces grandes entreprises dans les comités de programme et comités techniques d'organisations de conférences et ateliers du domaine (ex : IWAN, AMS, ANTA,...). Le projet CLARA présenté ci-avant est développé chez NEC. Ils développent également un service de gestion pour un environnement télécom basé sur la technologie des réseaux actifs[42]. Le GMD qui est un centre de recherche national Allemand s'est associé à la société Hitachi pour mener des travaux de recherche sur les réseaux actifs. Leur objectif est de développer une nouvelle infrastructure réseau basée sur une nouvelle architecture de réseaux actifs. Quelques informations sont disponibles sur [43]. Enfin le projet APE (décrit ci-dessus) est issue de NTT.

Le volume plutôt réduit de ce premier bilan sur les initiatives industrielles est probablement plus dû à une difficulté d'obtenir de l'information qu'à un manque d'intérêt de leur part. Ce qu'il est ressorti de ce bilan est que la plupart des projets débute généralement par des initiatives internes émanant de quelques membres des services recherche (et développement) de ces compagnies.

Les réseaux actifs facilitent le déploiement de nouveaux services et protocoles ainsi que la reconfiguration rapide des équipements. Ces caractéristiques peuvent être particulièrement attrayantes pour des industriels. Force est de constaté qu'il n'existe encore à ce jour aucun équipement actif disponible sur le marché ou bien même d'équipements qui propose des fonctionnalités actives activables ou non en fonction des exigences du client. Les réseaux actifs sont issus de travaux de recherches académique récents, qui proposent trop d'ouverture et de flexibilité pour un déploiement opérationnel par des industriels dans un contexte de production. Ils ne peuvent se permettre de mettre en danger une infrastructure réseau ou d'en ralentir les performances. Mais, bien qu'aujourd'hui encore sceptiques face à ce nouveau concept de réseau, de nombreux industriels suivent néanmoins de près l'évolution des recherches dans ce domaine en effectuant une veille technologique active, c'est à dire en s'impliquant par la création d'équipes de R&D internes qui assurent la promotion interne des nouvelles technologies, qui collabore avec des universitaires par le prêt de matériels ou en proposant leurs expertises. Les équipements actuels proposent quelques services de surveillances relativement simple du réseau en comptant par exemple le nombre de paquets transmis, service indispensable pour un opérateur télécom. Mais les industriels attendent aujourd'hui beaucoup plus d'un équipement réseau. Ils souhaitent une infrastructure moins figée, capable de s'adapter dynamiquement dans des délais très bref à des nouveaux besoins exprimés par leurs clients. On notera qu'un industriel doit prendre en compte des aspects très pragmatiques comme la taille de l'empreinte mémoire de l'EE, la complexité de développement d'un nouveau service et sa véritable valeur ajoutée [44]. Les industriels pensent qu'il n'y aura pas qu'un seul EE adopté par tous les constructeurs. Il n'est donc de ce fait pas nécessaire de standardiser l'EE mais un modèle d'exécution qui décrit comment une AA sera exécutée dans tout EE. Devra également être décrit comment l'EE se comportera avec le flux de données et le moteur de routage. Il est donc important de bien définir l'interface que propose l'EE à l'AA. Les choix pourront être guidé par l'expérience acquise lors des précédentes initiatives de standardisation (ex : P1520, NPF, ActiveNodeOS). Il est indispensable que les nœuds actifs puissent coexister avec les réseaux passifs actuels. La transition du passif à l'actif ou l'intégration du passif doivent être sérieusement prisent en considération. Il ne doit pas y avoir d'impact sur le réseau existant. Enfin le besoin d'outils pour mesurer la valeur ajoutée des AA comparé à la somme des coûts de déploiement des nœuds actifs et de développement d'applications sont indispensables dans un contexte industriel.

2.8 Y a t-il vraiment besoin des réseaux actifs ?

Comme nous l'avons déjà vu dans la section (2.2) de ce chapitre il existe déjà des équipements capables de fournir des services dans le réseau. Mais ces équipements sont dédiés et fournissent un service généralement unique. De plus ces services sont très statiques. Ils interviennent sur le paquet d'une manière complètement prédéfini. Un administrateur doit impérativement intervenir localement pour mettre à jour la configuration d'un équipement, comme les règles de filtrages dans le cas d'un pare-feu. Si l'on considère qu'un équipement actif puisse effectivement fournir ce genre de service on se rend compte alors du formidable potentiel qu'offre les réseaux actifs pour reconfigurer ou mettre à jour dynamiquement des équipements déployés. On peut également imaginer qu'un équipement actif puisse assister un équipement classique pour justement fournir à ce dernier la souplesse et l'intelligence nécessaires. Un des arguments souvent invoqués par les détracteurs des réseaux actifs est que les réseaux de données sont déjà aujourd'hui souvent sur dimensionnés. On parle d'*over-provisionning*. Ils donnent donc l'avantage à un équipement performant, le plus simple et le plus efficace possible (d'où la célèbre expression *« keep it simple, keep it stupid »*). Il leur importe peu d'utiliser le réseau de manière simpliste et donc de gâcher de la bande passante inutilement. En réalité le réseau est effectivement de plus en plus sur dimensionné mais pas de bout en bout. Seul les grands axes sont sur dimensionnés. Dans la pratique, peu d'utilisateurs ou même organisations ont la chance d'être directement relié à ces grands axes très haut débit. La plupart et en particulier les utilisateurs domestiques de l'Internet possède un accès au réseau à un débit beaucoup plus modeste, même avec des technologies comme l'ADSL très répandu dans la plupart des grandes villes françaises aujourd'hui. Ce débit est encore plus faible dans un contexte mobile qui peut être en plus très perturbé. On parle du dernier kilomètre. Ce réseau dit capillaire représente une quantité gigantesque de courtes liaisons et donc représente des sommes d'argent phénoménales pour pouvoir être mis à niveau afin de supporter une liaison haut débit jusqu'aux clients. On a donc besoin des réseaux actifs pour soulager les liens réseaux de bouts en adaptant le protocole de transport au médium de transport (adaptation à la volée) et pour les serveurs, par exemple, fournir suffisamment d'intelligence dans le réseau pour agréger des accusés de réceptions (`ack`) et limiter ainsi la retransmission des données perdues ou corrompues. La quête de la haute performance est le début de nouvelles investigations dans le domaine des réseaux actifs. La haute performance est nécessaire pour passer du prototype de laboratoire à une mise en œuvre industrielle et voir un jour apparaître de véritables transferts technologiques dans le monde industriel.

2.9 Conclusion

On peut considérer que depuis le milieu des années 90 à aujourd'hui le domaine des réseaux actifs a bien été étudié. Ces études on permit de défricher le domaine et donc fait très nettement apparaître les verrous qui sont la sécurité et la performance. Bien qu'il existe de nombreuses propositions dans le monde, les prototypes opérationnels proposées jusque là sont peu nombreux, peu transposables lorsqu'il nécessite du matériel dédié, ou peu fonctionnel. D'un point de vue recherche et développement ces réseaux proposent un lieu formidable d'investigations pour la mise au point grandeur nature de nouveaux protocoles et services avant qu'ils ne soient déployés peut être un jour en « dur » dans des équipements plus standard pour plus de performances au dépend de la souplesse de modification.

Chapitre 3

Propositions pour une architecture de réseau actif haute performance

3.1 Introduction

Ce chapitre propose une description détaillée de notre architecture de réseau actif et des nœuds actifs qui y seront déployés. Le chapitre est divisé en quatre sections. Les deux premières sections introduisent respectivement une architecture de réseaux actif générique et une architecture de nœud actif générique. Ces sections décrivent les caractéristiques et les fonctionnalités qui s'avèrent indispensables à la mise en place d'un réseau actif. La troisième section introduit notre architecture de nœud actif haute performance. Enfin la quatrième section présente notre proposition d'architecture de réseau actif haute performance et son déploiement.

3.2 Une architecture de réseau actif générique

Comme nous l'avons dit en introduction de ce document, les réseaux actifs constituent un outil ouvert et dynamique pour le déploiement, la mise au point et la validation de nouveaux protocoles et services réseaux.

3.2.1 Architecture d'un réseau actif générique

Un réseau actif doit être constitué de trois éléments de base : des équipements actifs, des serveurs de services et des postes clients.

> *On appelle* ***équipement actif*** *ou* ***nœud actif*** *un équipement réseau qui permet le déploiement, le chargement dynamique et l'exécution de services actifs au sein d'un Environnement d'exécution (EE).*

Un équipement actif doit pouvoir être plongé dans un réseau de données classique comme l'Internet. Il pourra être déployé en lieu et place d'un routeur ou d'une passerelle ou venir en renfort de ces équipements déjà déployés.

> *Un* ***Environnement d'Exécution*** *(EE) fournit un cadre dans lequel le code d'une application active est chargé pour y être exécuté.*

> *On appelle **service actif** ou plus simplement **service** une application active (AA). Il est constitué d'un code exécuté dans l'Environnement d'Exécution d'un nœud actif.*

Les services actifs sont exécutés pour appliquer un traitement sur un flux actif ou comme outil de gestion et de paramétrage d'un équipement actif.

> *On appelle **flux actif** un flux de données constitué de paquets préfixés d'un en-tête particulier indiquant qu'ils doivent être traités. L'en-tête contient l'identificateur du service qui doit être appliqué aux données transportées.*

Dans notre architecture de réseau actif générique nous déployons également des *serveurs de services* ou *service repositories.*

> *On appelle **serveur de services** (service repository) un système connecté au réseau qui fournit un espace de stockage dans lequel sont placés et mis à disposition les codes des services actifs pour les équipements actifs.*

Un équipement actif adresse alors sa requête de télé-chargement de services au serveur de services.

Finalement, le dernier élément qui constitue cette architecture sont les **postes clients**. Ils exécutent les applications qui émettent les flux. Ils pourront être de nature hétérogène (ex : PDA, mobile, poste ou station de bureau, serveurs,...) et donc être connectés aux équipements actifs par des connections hétérogènes (ex : cuivre (Ethernet), sans-fils (WiFi, BlueTooth), fibre optique (GigaEthernet)).

Les équipements actifs doivent pouvoir communiquer les uns avec les autres. On prévoit des mécanismes de découverte et de maintien de voisinage comme pour les routeurs (BGP) afin que les services puissent optimiser le transport des flux actifs d'un nœud à l'autre en profitant de la connaissance acquise par l'équipement actif sur ses voisins (en termes de ressources mémoire et CPU disponibles ainsi que des caractéristiques des liaisons).

Il est nécessaire de distinguer les flux actifs des flux passifs à l'aide d'une solution simple et supportée par les équipements de routages, passerelles et pare-feux du réseau tels qu'ils sont aujourd'hui. Ce mécanisme de distinction permettra de filtrer les flux actifs et de les rediriger vers un équipement actif, local ou distant. Néanmoins un flux passif doit pouvoir se voir appliquer également un traitement actif car certaines applications s'y prêtent parfaitement (ex : web surfing). Pour cela on dirigera le flux passif sur le port d'un équipement actif donné, comme on dirige une requête web sur un proxy par exemple.

Les flux actifs devront être transportés à l'aide des protocoles de transports existants (`TCP`, `UDP`...), afin de faciliter le déploiement rapide de nœuds actifs dans l'infrastructure existante. Le transport de données actives par un protocole de transport non connecté comme `UDP` est la solution la plus communément adoptée par les projets de réseaux actifs existants. Dans le cas d'un protocole de transport connecté comme `TCP` deux solutions sont proposées dans le chapitre (4.2).

3.2.2 Connections en mode proxy ou de bout en bout

Notre architecture d'infrastructure active, dans un souci très pragmatique d'intégration au réseau existant, doit pouvoir proposer :

- soit un mécanisme d'attrapage à la volée des paquets actifs par les équipements actifs (approche bout en bout). Cela implique que le client qui émet un paquet actif sur le réseau, le marque comme devant être traité lors du passage dans un équipement actif. Cette marque ne doit pas entrer en conflit avec les standards existants ou introduire un trop lourd coût d'analyse supplémentaire pour les paquets (passifs). Attraper un paquet à la volée dans un équipement actif consiste à le détourner, le modifier, puis le réinjecter dans le réseau. Cette démarche peut se rapprocher d'une attaque type *man in the middle*. L'avantage de l'approche « attrapage à la volée » est que l'on respecte le principe de bout en bout. Les équipements actifs ne sont pas connus des machines clients et il n'est pas nécessaire de connaître la topologie des équipements actifs déployés. Par contre, il y a risque de destruction des paquets par les équipements classiques réseau lorsque le paquet se différencie avec une option mal supportée.
- soit la mise en place de tunnel IP entre chaque équipement actif, entre chaque client et premier nœud actif traversé et enfin entre chaque client et dernier nœud actif traversé (approche dite par morceaux ou en mode proxy). Pour les communications entre les clients et le premier ou dernier nœud actif traversé, nous proposons des nœuds actifs appelé *proxy*. Leur particularité réside simplement dans leur localisation. Ils sont les premiers à être traversé par les flux actifs. Dans cette architecture, dite par morceau, les équipements actifs doivent avoir connaissance des autres nœuds actifs déployés. Ils doivent donc maintenir une table de routage permettant d'y accéder. Par contre les clients n'ont besoin de connaître que le proxy actif qui leur sert de passerelle pour accéder à l'infrastructure active. Cette architecture brise le principe de bout en bout lors de l'usage du protocole `TCP`. L'avantage de l'approche en mode proxy est sa facilité de mise en œuvre. Elle est bien adaptée pour les services lourds appliqués à un flux de données circulant sur `TCP`.

Une alternative à ces connections par morceaux est l'attrapage des paquets à la volée.

3.2.3 Gestion et administration des nœuds actifs déployés

Les services destinés à être déployés dans l'infrastructure active doivent être au préalable vérifiés et validés. Ils doivent contenir des indications sur leur provenance : auteur du service, organisme... Le but est de fournir une traçabilité des services et pouvoir remonter à la source dans les plus bref délais en cas de problème. De plus, dans un souci de passage à l'échelle, un nœud actif doit être le plus autonome possible. Il doit s'autogérer. Il n'est plus concevable dans un contexte réaliste d'imaginer un administrateur devant gérer manuellement l'installation, la mise à jour ou le retrait d'un service devenu obsolète sur tous les équipement actif déployés. Seule la mise à jour de l'EE devra être réalisée manuellement. Mais cette opération devra avoir un caractère exceptionnel au même titre que l'on met à jour le *firmware* d'un équipement réseau classique. Des outils de surveillance qui permettent d'interroger l'état global d'un nœud actif ou l'état d'un service particulier sont indispensables. Ils seront nécessaires par exemple pour la mise au point de nouveaux services. Les outils de contrôle qui s'exécutent sur le nœud actif pourront être eux-mêmes mis en œuvre sous la forme de services. Cela signifie que ces outils seront également à même d'être déployés et mis à jour dynamiquement.

3.2.4 Déploiement dynamique de services

Notre architecture de réseau actif fournit un mécanisme de déploiement dynamique qui facilite l'implantation de nouveaux services (et protocoles). Le déploiement est réalisé sur le chemin du flux actif de données. Cela signifie qu'un service est déployé à la demande des équipements actifs traversés par ce flux actif. Trois scénarios sont alors possibles. Le premier, toujours déclenché, consiste à vérifier si le service requis par un flux venant d'arriver sur le nœud actif est disponible dans le cache local des services. Dans la pratique ce cache sera situé sur une mémoire de masse de l'équipement actif si celui-ci en est pourvu, sinon on pourra utiliser un *ramdisk*. Si le service est présent, il sera dynamiquement chargé en mémoire et appliqué sur le flux. Sinon, les deux autres scénarios peuvent intervenir en fonction de la position de l'équipement actif. Le premier nœud actif traversé, par un flux actif provenant d'une application et qui n'a donc pas encore rencontré d'équipements actif sur son chemin, n'est autorisé à télé-charger le service que depuis un serveur de services. Nous interdisons la distribution de services à partir d'un poste client où s'exécute l'application émettrice, pour des raisons de sécurité. Un service pour être déployé doit avoir été installé et enregistré sur un service repository officiel. Afin de déposer le code d'un nouveau service sur un service repository il est nécessaire d'avoir été au préalable enregistré comme étant autorisé à déposer un nouveau service sur celui-ci. Un nœud actif qui requiert un service devra avoir établi une relation de confiance avec le service repository tenant le service nécessaire à disposition. Le transport du service entre le service repository et le nœud actif pourra être réalisé de manière sécurisée (cryptée). Un nœud actif situé sur le chemin du flux aura pour alternative de pouvoir télé-charger le service requis à partir du service repository ou à partir du nœud actif précédent. Plusieurs critères permettant au nœud actif de faire un choix entre charger un service à partir du nœud précédent ou à partir du service repository sont possibles. Néanmoins le critère de choix peut être fait sur la charge CPU annoncée par le dernier actif traversé. Si celui-ci est extrêmement chargé on privilégiera un télé-chargement à partir du service repository. Telle que nous décrivons notre architecture, le service repository, indispensable au premier nœud actif traversé par un nouveau flux, ne doit pas entraver le déploiement d'un service en cas de panne. Un nœud actif doit pouvoir, en cas d'échec télé-charger le même service ailleurs. Cela implique que les serveurs de service sont dupliqués et distribués dans le réseau. Finalement notre architecture réduit les possibilités de la définition originelle des réseaux actifs qui énonce que quiconque doit pouvoir déployer son service actif dans un réseau actif. Nous pensons (comme de nombreux autres projets pragmatiques) que pour des raisons évidentes de sécurité, nous ne pouvons pas autoriser n'importe qui à déployer son propre service dans un réseau de production sans que ce service ait été au préalable validé par un organisme certifié.

Après cette vision globale d'un réseau actif générique, il convient de définir les fonctionnalités supportées par les équipements actifs.

3.3 Une architecture de nœud actif générique

Cette section propose de décrire une architecture de nœud actif générique basé sur une solution purement logicielle, c'est à dire qu'elle n'emploie pas de matériel dédié. L'Environnement d'Exécution est installé et exécuté dans l'espace utilisateur d'un équipement actif mono machine (Figure 3.2). Le paquet actif arrive par une carte d'interface réseau de l'équipement actif. Puis il est dirigé par le noyau vers le système local et atteint l'EE. Celui-ci se charge ensuite de diriger à l'aide d'un démultiplexeur qui analyse l'en-tête du paquet actif, la charge utile (les données) vers le service actif adéquat, représenté par un résistor sur la figure 3.2. Le paquet, une fois traité, peut alors poursuivre son chemin sur le réseau.

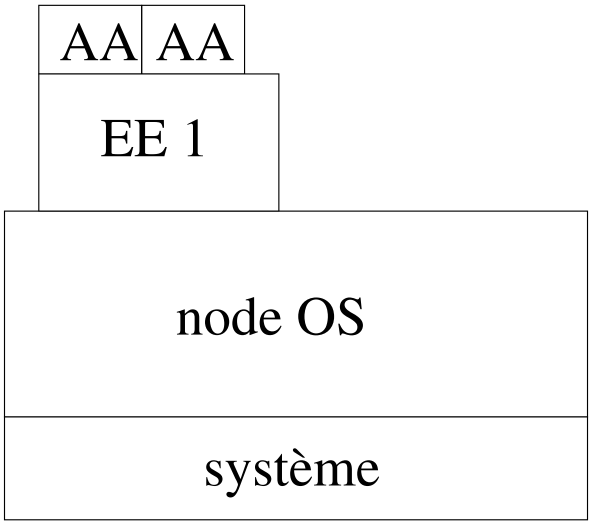


FIG. 3.1 – On supporte un unique EE, on supprime donc la couche de démultiplexage.

Contrairement à l'architecture de nœud actif générique présenté en introduction du document, nous exécutons un EE unique directement sur le système d'exploitation du nœud actif (cf. figure 3.1). Nous n'employons pas de démultiplexeur qui permet le support de plusieurs EE sur un même équipement actif ni d'interface destinée à standardiser les communications entre l'EE et le système (abstraction du système sous-jacent). L'EE est exécuté dans l'espace utilisateur de l'équipement actif. Le chronogramme 3.3 montre quelles parties du système sont traversées par le paquet actif[1]. Le paquet arrive par une carte d'interface réseau (NIC=*Network Interface Card*), est dirigé par le noyau vers le système local et atteint l'EE après avoir été recopié dans l'espace utilisateur ($K \rightarrow U$). L'EE se charge de diriger à l'aide d'un démultiplexeur, le paquet vers le service actif adéquat. Ensuite le paquet une fois traité est retransmis sur le réseau en retraversant l'espace noyau puis il subit une opération de routage pour déterminer la carte d'interface de sortie. L'équation $T(t_{AA})$ formalise très simplement le temps qu'un paquet actif met pour traverser la totalité d'un équipement actif. Cette fonction a une unique variable t_{AA} qui modélise le temps d'exécution de l'application active sur un paquet. On considère que le temps de traversée des autres couches est quasi constant. On considère également que le temps de recopie d'un espace vers l'autre est identique ($T_{U\rightarrow K} = T_{K\rightarrow U}$).

[1]On appelle chronogramme une frise du temps divisé en périodes dont les longueurs symbolisent le temps passé par un paquet actif dans une zone donnée du système.

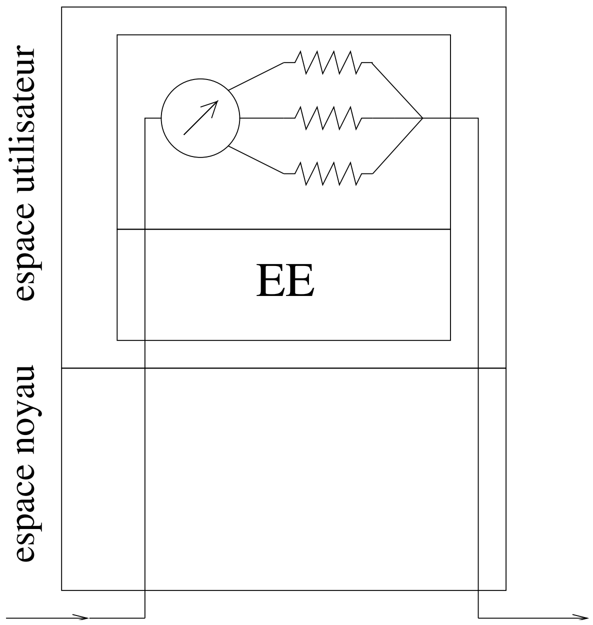
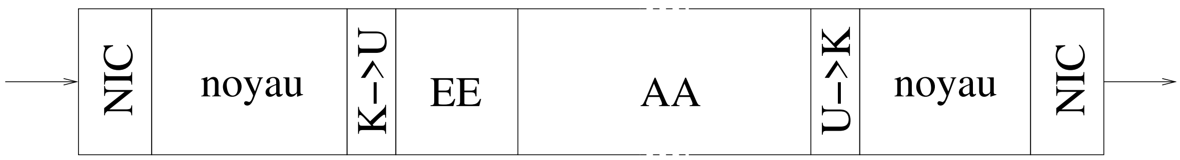


FIG. 3.2 – L'EE est exécuté dans l'espace utilisateur du système.



$$T(t_{AA}) = 2(T_{NIC} + T_K + T_{U\rightarrow K}) + T_{EE} + t_{AA}$$

FIG. 3.3 – Chronogramme d'un paquet actif traité dans l'espace utilisateur

3.3.1 Choix du langage de développement d'un nœud actif générique

Le choix du langage de développement pour le nœud doit être fait en fonction des critères suivants :

- Le langage employé pour l'écriture des services doit être simple avec un nombre d'instructions de base limitées rendant son apprentissage rapide pour l'écriture de nouveaux services. Nous voudrions que quiconque ayant quelques notions d'algorithmique puissent développer son propre service. La syntaxe du langage devra donc être facile d'accès, permettre de lire et comprendre aisément le fonctionnement d'un service existant dans un premier temps, puis offrir la possibilité d'écrire un service personnalisé adapté au transport des données de son application.
- Le caractère distribué de cette architecture réseau fait qu'il est indispensable d'employer un langage extrêmement portable pour l'écriture des services déployés de nœud en nœud. Une première approche naïve consiste à utiliser un langage interprété, dont l'interpréteur a été porté sur toutes les architectures matérielles qui exécute l'Environnement d'Exécution (EE). Mais comme nous le verrons plus loin cette solution est in-envisageable dans le contexte de performance où nous nous plaçons.
- Lors du démarrage de l'EE, celui-ci n'a pas à priori connaissance des services qu'il devra exécuter. Il les découvre lorsqu'un flux lui en fait la demande. Or l'EE ne peut être stoppé. Il est donc absolument nécessaire que l'EE puisse lier dynamiquement un nouveau service, c'est à dire le charger en mémoire et accéder à sa table des symboles, sans avoir à être arrêté puis redémarré. Nous considérons donc un service, d'un point de vue programmation système, comme une bibliothèque dynamique. Ces bibliothèques ont la particularité de pouvoir être chargées en mémoire, et donc d'en occuper l'espace que lorsqu'il y en a besoin. De plus, si plusieurs applications nécessitent une même bibliothèque, elle n'est chargée qu'une seule fois en mémoire. Seul les membres non statique de la bibliothèque sont dupliqués en mémoire. Pour information, les bibliothèques dynamiques sont repérables par leur extension `.dll` sous Windows et `.so` sous Unix.
- Les services actifs doivent pouvoir être exécutés dans un environnement robuste. Un service qui se bloque ou génère des erreurs (ex : division par 0, dépassement de capacités, lecture/écriture hors limites,...) ne doit en aucun cas mettre en danger le nœud actif et donc par effet de bord l'infrastructure active complète. Le langage doit donc mettre à disposition des mécanismes robustes de gestion des exceptions que les programmeurs de services seront tenus de gérer. Chaque service doit s'exécuter dans un espace qui lui a été alloué ; il ne doit pas pouvoir interagir avec d'autres services traitant un autre flux. Enfin, l'usage d'un langage strict sur le typage des données, qui ne permet pas de transtypage aléatoire est considéré comme un atout qui mène plus facilement à l'écriture de services fiables.

3.3.2 Multi services et multi *threads*

Il est indispensable comme pour n'importe quel type de serveurs de pouvoir répondre à de nombreuses requêtes en parallèle. Un serveur web par exemple émet des pages web en parallèle vers les clients d'où proviennent les requêtes lui ayant été adressé. Les pages (**`html, php`**) ne sont pas servies strictement les unes après les autres dans l'ordre d'arrivée des requêtes par exemple mais en parallèle. Un nœud actif doit pouvoir traiter de nombreux flux en parallèle. Dans notre cas tous les flux n'exigent pas forcément le même service à appliquer. Mais puisque les flux circulent en parallèle, les différents services doivent être exécutés en parallèle. C'est une approche dite multi-threadée. De plus, un nœud actif n'est absolument pas spécialisé ou réservé à appliquer un service type. L'EE doit être capable d'accueillir des services qui doivent pouvoir cohabiter et s'exécuter en parallèle.

3.3.3 Support de plusieurs protocoles de transports

Les deux principaux protocoles de transports de données sur le réseau supportés par notre architecture d'EE sont `TCP` et `UDP`. Il appartient au développeur de services d'employer l'un ou l'autre ou encore de supporter les deux en fonction des exigences de l'application. Étant donné les caractéristiques propre à chacun de ces protocoles de transport (connecté, non connecté) l'algorithme du service ne sera pas forcément codé de la même manière en fonction du protocole de transport choisi pour l'application.

3.3.4 Stockage dans le réseau

De nombreux services actifs ont besoin pour améliorer le transport, de cacher des données. Certains ne nécessitent de stocker que quelques variables d'états soit quelques octets, d'autres nécessite de cacher un flux de données complet (ex : séquence vidéo). Cela est d'autant plus critique si le débit du flux est important. Il est donc important de pouvoir fournir un outil de stockage générique pour tous les services exigeant de mémoriser temporairement de grandes quantités d'informations.

Les résultats de ces travaux préliminaires permettent une mise en œuvre relativement simple et complète d'un réseau actif. Cette architecture bénéficie déjà de fonctionnalités fournies aux nœuds actifs qui sont : les différentes possibilités de déploiement dynamique de services (de proche en proche ou à l'aide d'un serveur de services), une connexion en mode proxy ou de bout en bout, l'application de services en parallèle sur différents flux et une intégration facilitée par l'usage du support des principaux protocoles de transport existant. Dans la deuxième partie de ce chapitre nous allons maintenant focaliser notre étude sur l'aspect performance des réseaux actifs et proposer des solutions en vue d'améliorer notre architecture de nœud actif générique ainsi que le modèle de réseau actif.

3.4 Proposition pour une architecture de nœud actif haute performance

Notre objectif est de concevoir un nœud actif, capable de supporter un lien dont le débit soit adapté aux réseaux actuels (Gbps) (Fig. 3.4).

Afin de parvenir à ce résultat nous minimisons le coup de traversée de l'EE qui est la couche logicielle qui doit déterminer quel service doit être appliqué à un paquets actifs, diriger sa charge utile vers le service requis, appliquer le service, puis une fois traité, ré-emettre le paquet sur le réseau.

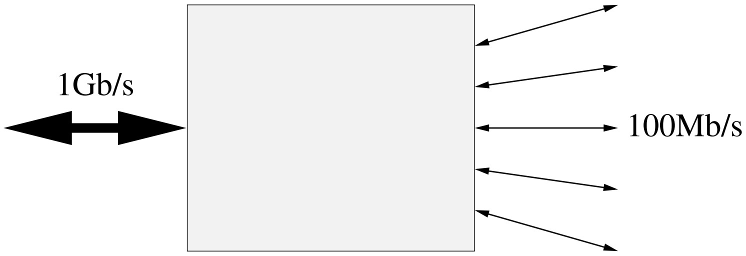


Fig. 3.4 – Notre objectif. Un équipement actif capable de supporter plusieurs flux actifs acheminés sur un lien Gigabit.

3.4.1 Une architecture de nœud actif à quatre niveaux

Afin de fournir un EE efficace nous proposons une architecture de nœuds actifs découpée en quatre niveaux. Cette proposition est issue du constat suivant. Afin de réduire le temps passé par un paquet dans un équipement actif il nous parait logique de réduire au maximum le nombre de couches qu'il doit traverser. Dans un système, ces couches sont matérialisées par les recopies de données d'une zone mémoire à l'autre, notamment lors d'un passage de l'espace noyau à l'espace utilisateur, par l'allocation dynamique de mémoire, par la génération d'interruptions et l'exécution d'appels systèmes. Cette architecture à quatre niveaux est illustrée figure 3.5.

La première couche (NIC) constitue la première zone du système susceptible d'embarquer suffisamment de logique pour traiter des paquets. Il existe aujourd'hui des cartes d'interface réseaux programmables équipées d'un (ou plusieurs) processeur(s), de mémoire et moteur DMA. Il existe également des composants dédiés programmables comme les *Network Processor*. Cette couche semble donc très bien adaptée dans un contexte haute performance puisque c'est celle qui est située au plus près du « fil ». Cela signifie qu'un paquet n'aura que très peu d'obstacles à traverser pour se voir traité. Néanmoins cette couche implique diverses contraintes. La puissance de traitement proposé par les cartes d'interfaces programmable est aujourd'hui encore inférieure à celle d'un processeur principal d'un système. L'espace mémoire disponible est également relativement limité et donc insuffisant pour un service qui nécessite de cacher des quantités importantes de données.

La seconde couche correspond à la zone mémoire où s'exécute le noyau d'un système d'exploitation. L'avantage de traiter un paquet dans cet espace est qu'on bénéficie de la puissance du processeur hôte, de sa mémoire et toujours de la proximité du lien. Mais l'exécution d'un service dans cette couche n'est pas sans contraintes. Le développement de services implique que l'on emploie un langage proche du processeur hôte (assembleur ou C compilé pour l'architecture) ce qui limitera la portabilité des services compilés. On peut imaginer un interpréteur embarqué dans le noyau mais bien sûr au détriment des performances. De plus un service mal programmé peut mettre tout le nœud actif en péril. Enfin le système d'exploitation doit supporter le chargement dynamique de nouveaux codes. Notre première proposition consiste donc à limiter la remontée du paquet du réseau vers l'espace noyau de l'équipement actif. La figure 3.6 et le chronogramme 3.7 illustre les parties du système traversées par le paquet actif. Ici le paquet est démultiplexé par un EE allégé appelé uEE (u=micro) et on constate que cette approche permet de nous affranchir des recopies coûteuses de l'espace noyau vers

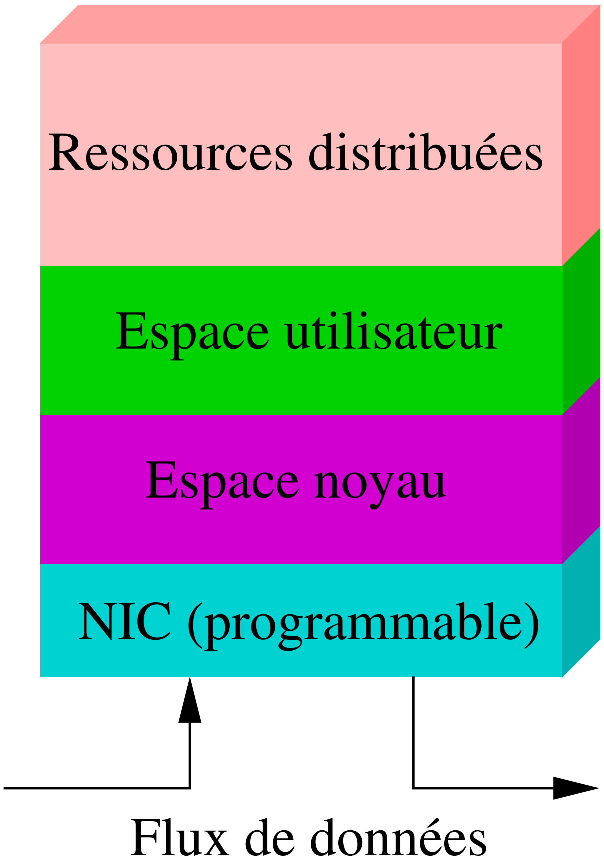


Fig. 3.5 – Architecture d'un nœud actif en couche.

l'espace utilisateur et vice et versa. En terme d'architecture nous sommes très proche du nœud actif générique décrit section 3.3. On retrouve un environnement d'exécution (simplifié), un mécanisme de démultiplexage et des services chargeables dynamiquement. L'équation $T(t_{AA})$ formalise de manière simple le temps de traversée d'un paquet actif dans un nœud actif dont l'EE est exécuté dans l'espace noyau. Par rapport à l'équation proposé pour un EE exécuté dans l'espace utilisateur on supprime le terme $2T_{U \to K}$ et on remplace T_{EE} par $T_{\mu EE}$. La variable de cette fonction reste bien sûr le coût de l'application active.

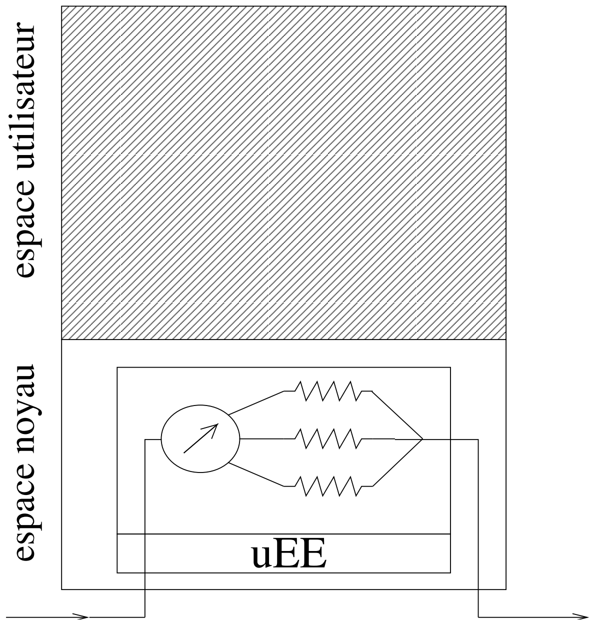


FIG. 3.6 – Le traitement est réalisé dans l'espace noyau.

$$T(t_{AA}) = 2(T_{NIC} + T_K) + T_{\mu EE} + t_{AA}$$

FIG. 3.7 – Chronogramme d'un paquet actif traité dans l'espace noyau.

La troisième couche correspond à la zone mémoire de l'espace utilisateur. Cette zone bénéficie non seulement du processeur et de la mémoire du système hôte mais également de tous les équipements dont le nœud actif est pourvu (mémoire de masse (disques durs), cartes dédiées (ex : compression, cryptage)). C'est la zone mémoire ou s'exécute toutes les applications des utilisateurs, dans le langage de leur choix (Java, Perl,...), c'est donc une zone protégée mais qui en contrepartie est situé loin du lien. L'architecture de nœud actif générique telle que décrite section 3.3 sera exécuté dans l'espace utilisateur.

La quatrième couche appelée ressources distribuées embarque diverses fonctionnalités sur lesquelles nous nous focaliserons dans la section suivante (3.4.2).

Quelle que soit la couche dans laquelle est exécuté un service, les deux modes d'attrapage de paquets, mode proxy ou mode bout en bout, devront pouvoir fonctionner. Notre proposition de découper un nœud actif permet de nous rendre compte que chaque couche possède ses propres caractéristiques, avantages et inconvénients. Plus on descend près du lien, plus les contraintes sont fortes (difficulté de mise en œuvre) et plus le gain en performance semble évident. On prévoira donc un mécanisme de communications inter couche afin qu'un même service puisse être distribué sur les différentes couches. Les couches hautes fourniront la souplesse nécessaire par exemple au déploiement et les couches basses la performance. Bref, il est nécessaire de classifier les services qui en fonction de leur caractéristiques (consommation CPU et/ou mémoire, complexité, fonctionnalités, langage d'écriture) seront exécutés dans la couche la mieux adaptée.

3.4.2 Extensibilité des performances : traitement distribué

Les flux de données auxquels doivent être appliqués des services consommateurs de ressources (CPU, mémoire) doivent être traités par une architecture parallèle. Nous proposons dans cette section plusieurs modes de traitements distribués. La figure 3.8 représente trois types d'architectures susceptibles de pouvoir accueillir un nœud actif distribué. Dans la figure

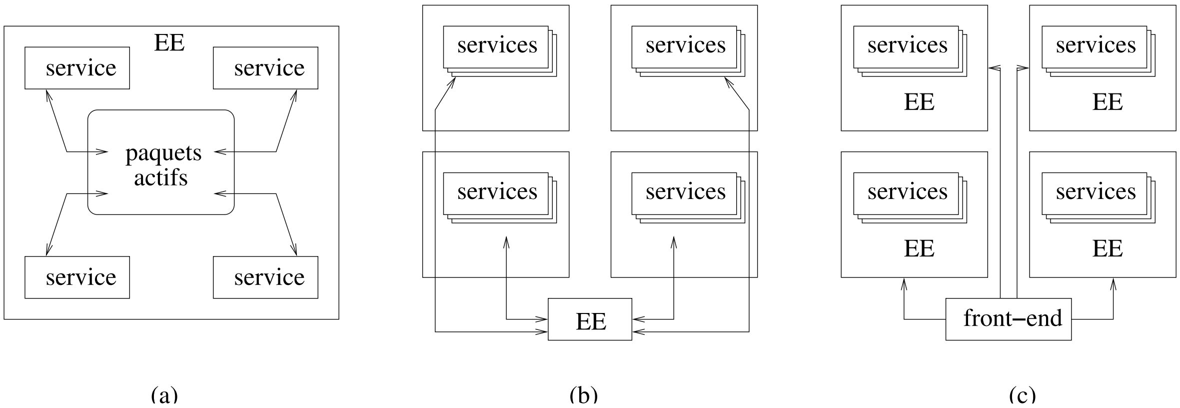


FIG. 3.8 – Différentes approches de conception d'un nœud actif distribué : (a) mémoire partagée, (b) passage de message et (c) EE répliqués.

3.8-a l'EE est exécuté soit sur une machine de type SMP (Symetrical Multi Processing) qui est un système à mémoire partagée, c'est à dire équipé d'une carte mère avec deux processeurs ou plus, soit sur une grappe de machine avec une bibliothèque de mémoire distribuée partagée [45]. Les paquets atteignant ce nœud actif sont placés dans des files d'attente situées dans la mémoire partagée avec une file pour chaque type de service disponible. Les services sont considérés comme des consommateurs de ces files d'attentes.

Dans la figure 3.8-b l'EE reçoit les paquets des flux actifs. Il analyse l'en-tête active de chaque paquet et les diriges vers le nœud qui possède le service requis. Une technique de passage de message comme MPI (*Message Passing Interface*) peut être utilisée. Ensuite, le nœud traite le paquet avec le service adapté avant sa retransmission. Chaque service doit être un processus en attente de message provenant de l'EE. Dans la figure 3.8-c l'EE et ses services sont répliqués sur chacun des nœuds de la grappe. Nous appellerons ces nœuds des *back-end*. On utilise un nœud frontal (*front-end*) complètement dédié à la distribution des flux sur les back-ends. Pour cela le frontal doit mettre en œuvre un algorithme de distribution de flux. On pourra utiliser un simple algorithme du tourniquet (*Round-Robin*(RR)) mais cela risque de conduire très rapidement, dans un contexte réel, à un déséquilibrage de la charge de calcul entre les différents back-ends. D'autres algorithmes d'équilibrage de charge plus sophistiqués sont disponibles. L'algorithme *Weighted-RR* consiste à coefficienter préalablement la puissance de calcul de chaque *back-ends*. Ainsi le back-end ayant reçu le coefficient le plus fort se verra attribuer un nombre de flux plus important que le back-end le plus faible. Cet algorithme permet ainsi la construction de grappes hétérogènes. Un autre algorithme appelé *Least Connection RR* existe. Celui-ci consiste à diriger un nouveau flux vers le back-end ayant le moins de connections en cours. Cela permet d'équilibrer le nombre de connections sur chaque back-end, car chaque connexion ne dure pas forcément le même temps. On peut ensuite imaginer un algorithme qui combine ces deux derniers afin de proposer une solution hybride. Nous étudions actuellement une stratégie de distribution différente de celles proposées ci-dessus. Cette nouvelle stratégie est basée sur le concept de boucle de rétroaction (*feedback*). Cela nécessite trois composants : un frontal qui mette en œuvre la distribution des flux ou des paquets à l'aide d'une table de charge des CPU des back-ends qu'il consultera au préalable, un démon exécuté sur le frontal qui met à jour la table des charges et, des sondes sur chaque back-end qui envoient leur charge CPU respective au démon du frontal. On peut ensuite raffiner l'approche en envoyant non seulement la charge CPU de chaque back-end mais également le taux d'occupation mémoire.

Le principal avantage d'utiliser un système à mémoire partagée, tel qu'une architecture SMP, ou une architecture distribuée est le parallélisme. Les flux sont traités par les services en parallèle. Le second avantage à utiliser une architecture distribuée comme une grappe de PC est la haute disponibilité du nœud actif, une excellente tolérance aux pannes et enfin la possibilité de réaliser des travaux de maintenance sans avoir à arrêter le nœud actif complet et les services qu'il fournit. Il reste malgré tout un élément qui risque de mettre tout le nœud actif en péril en cas de panne. Cet élément est le frontal, représenté en bas des figures 3.8-b et 3.8-c. Dans un cas, l'un contient l'EE, dans l'autre cas un directeur de flux actif. Dans un contexte de haute disponibilité il est indispensable de *monitorer* et de doubler cet élément critique du nœud actif.

Notre proposition distribuée utilise donc la solution où les EE sont répliqués (3.8-c). L'équipement frontal destiné à distribuer les paquets ou les flux sur les équipements actifs situés en arrière (*Back End* (BE)), sont représentés sur la figure 3.9. L'équilibrage de charge est donc réalisé par le frontal de préférence dans l'espace noyau. Le frontal pourra également être utilisé pour capturer les paquets actifs dans un mode de fonctionnement bout en bout. Les BE peuvent toujours exploiter l'une ou l'autre des deux approches présentées ci-dessus qui consistent à exécuter un EE dans l'espace utilisateur ou dans l'espace noyau. La figure 3.10 illustre le passage du paquet dans un équilibreur de charge (Bal.= *Balancer*) avant d'être traité par un BE. Dans l'architecture distribuée illustrée par la figure 3.9 le paquet devra, pour atteindre le BE, passer par un réseau local (LAN) et donc par une carte d'interface supplémentaire.

Finalement, le chronogramme 3.11 résume les différents cheminements possibles d'un paquet actif dans notre architecture. Dans le cas d'un traitement distribué les flux passent par un frontal qui distribue les flux ou les paquets aux BE qui proposent un EE dans l'espace noyau ou/et dans l'espace utilisateur.

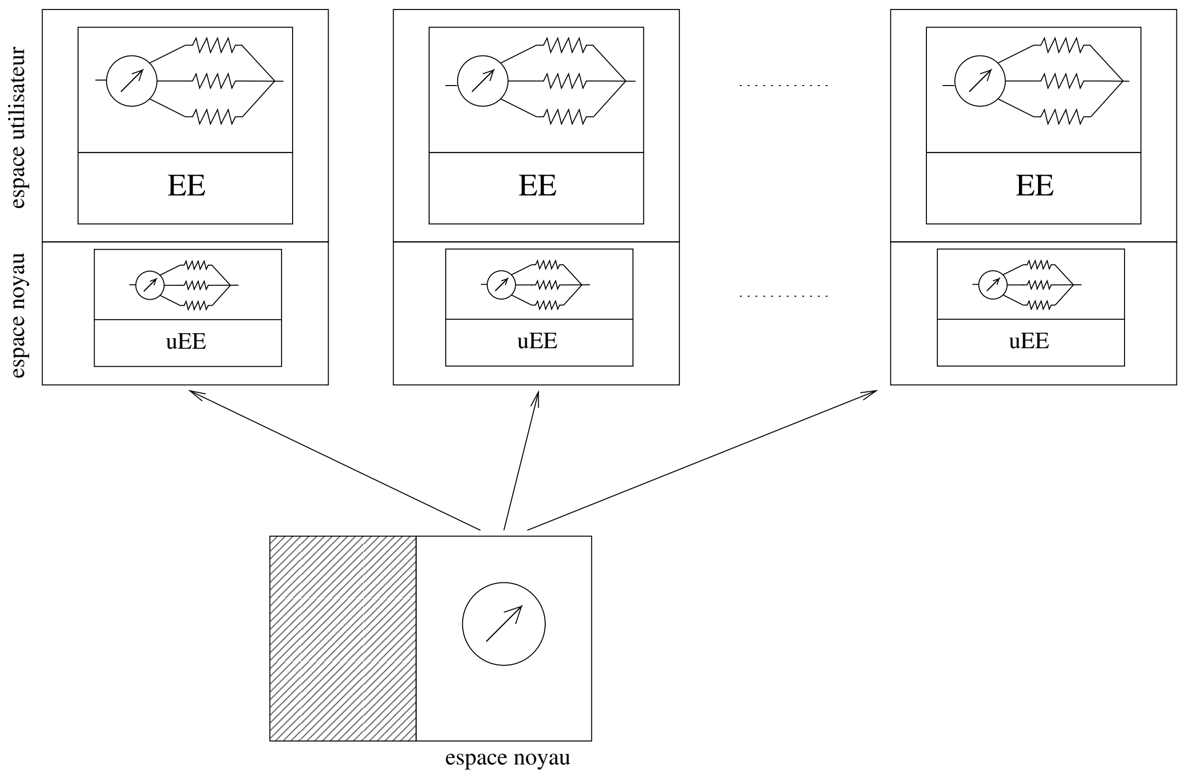


FIG. 3.9 – Une architecture distribuée.

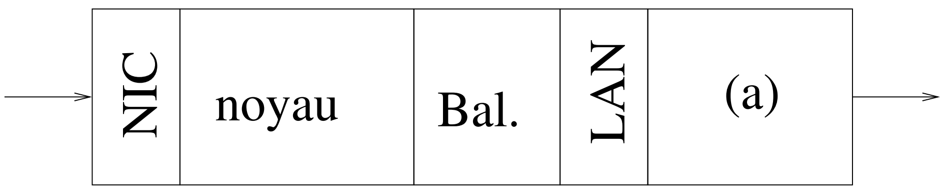
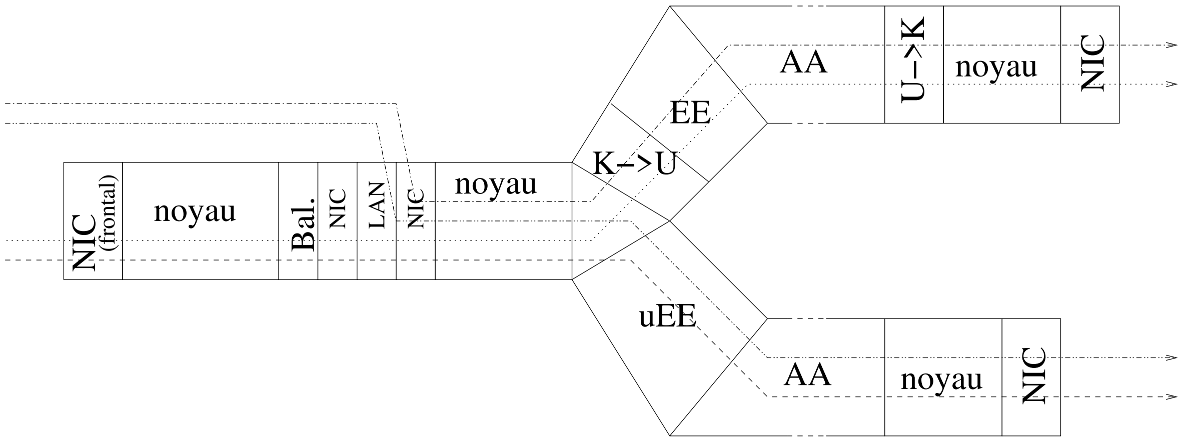


FIG. 3.10 – Le paquet actif passe par un équilibreur de charge (Bal.=Balancer) exécuté dans le noyau, puis il traverse un réseau local pour atteindre un des BE. (a) peut-être remplacé par les deux approches précédentes (EE ou uEE).

Les équations Θ_u et Θ_k formalisent, respectivement dans l'espace utilisateur et dans l'espace noyau, le temps de passage d'un flux de données de taille finie (à n paquets) dans un équipement actif pourvu de N backends.

On peut imaginer exécuter les services directement sur le frontal mais il serait probablement insuffisant pour supporter de nombreux flux. Une grappe de machine est constituée d'un groupe de nœuds mono machine auxquelles on accède par un frontal. Le paquet actif passe à travers une carte (NIC) puis s'arrête dans l'espace noyau pour être traité par le uEE ou le cas échéant remonte jusque dans l'espace utilisateur pour être traité par un EE de plus haut niveau. On remarque que mis à part le service actif (AA), tous les éléments traversés par le paquet ajoutent un temps de traversée constant. La somme de ces latences plus celle induite par le service (AA) donne la latence totale d'un paquet. La latence induite par le service (AA) soit t_{AA} n'est « maîtrisée » que par le développeur de celui-ci. Cela signifie que si il utilise un algorithme mal adapté ou coûteux en cycle CPU, il augmentera d'autant plus le temps de passage dans l'équipement actif.



$C = 2T_{NIC} + T_K + T_{Bal} + T_{LAN}$
$\Theta_u(t_{AA}, n, N) = n(C + \frac{2(T_{NIC}+T_K)+T_{EE}+2T_{U\rightarrow K}+t_{AA}}{N})$
$\Theta_k(t_{AA}, n, N) = n(C + \frac{2(T_{NIC}+T_K)+T_{\mu EE}+t_{AA}}{N})$
Θ : temps de traversée d'un TAN par un flux de taille finie
n : nombre de paquets, N : nombre de backends.

FIG. 3.11 – Chronogramme global.

3.4.3 Classifier les services

Les travaux de thèse de VIRGINIE GALTIER [46] ont montré qu'il est extrêmement complexe de réaliser des prévisions sur la quantité de ressources nécessaires au bon fonctionnement d'un service. L'usage des ressources est matérialisé par la consommation de cycle CPU et la quantité de mémoire nécessaire au bon fonctionnement du service. Néanmoins nous classifions les services en quatre grandes familles. Un paquet actif peut se voir appliquer des services :

- **hyper légers** (peu consommateur de CPU, peu consommateur de mémoire) : Ce sont des services sans état, qui appliquent un traitement extrêmement léger sur les paquets actifs tel qu'un service de marquage ou de filtrage. Ces services peuvent être exécutés

sur une carte d'interface programmable, au plus près du lien.
- **légers** : toujours peu consommateurs de cycles CPU, mais demandent un minimum de mémoire pour pouvoir y stocker quelques états. Ces services peuvent être exécutés dans l'espace noyau qui fournit un accès total à la mémoire du système.
- **moyens** : qui demandent un environnement suffisamment riche pour pouvoir effectuer des traitements relativement complexes. Ces services s'exécutent dans l'espace utilisateur donc dans un espace protégé, ne risquant pas de mettre tout le nœud actif en péril. Le service peut accéder à toutes les ressources matérielles mises à sa disposition sur le nœud (mémoire, disques, cartes spécialisées, réseau,...).
- **lourds** : très consommateurs de cycles CPU ou très consommateurs de mémoire (Mémoire volatile (RAM) ou mémoire disque). Ces services ont besoin d'être distribués et parallélisés. Cette parallélisation peut intervenir à deux niveaux de granularité différents : au niveau paquet ou au niveau flux. Dans le premier cas, les paquets sont distribués sur les unités de traitement qui leur appliquent le service. Dans le second cas un flux complet est associé à une unité de traitement ou de stockage. Les flux sont alors traités en parallèle. Il est ensuite important d'employer un algorithme bien adapté de distribution des paquets ou des flux, en fonction de la granularité choisie.

On peut également faire une distinction simplificatrice. La plupart des services agissant sur le *plan contrôle* semblent peu consommateurs en cycle CPU et espace mémoire ; ils pourront donc être mis en œuvre sous la forme de service léger ou hyper léger. En contre partie les services agissant sur le *plan données* nécessitent plus de moyens et seront mis en œuvre sous la forme d'un service lourd.

3.4.4 Langage de développement pour une exécution rapide et portable

Étant donné le contexte dans lequel nous nous situons, qui est celui de la performance, il est impensable d'utiliser un langage interprété. Bien que souvent portable, les langages interprétés ne peuvent se défaire du coût d'interprétation indispensable à son exécution. D'un autre coté, un langage machine bien que extrêmement rapide n'est pas portable et difficile à programmer. L'usage d'un langage intermédiaire comme le C apparaît comme une bonne solution pour la performance mais perd sa portabilité une fois compilé. Un service écrit en C ANSI, donc très portable, devrait être transporté sous forme de source et compilé à la volée dés qu'il arrive sur une nouvelle architecture. Une autre approche consiste à pré compiler le service pour toute les architectures possibles et susceptibles de l'exécuter. Toutes ces versions seraient alors disponibles sur le serveur de services. Un nœud actif ayant besoin d'un service devra donc en plus de préciser un identificateur de service, lui préciser sur quelle architecture il souhaite l'exécuter. Un nœud actif qui souhaite télé-charger un service à partir d'un autre nœud actif (le précédent) devra d'abord s'assurer si ils ont la même architecture matérielle, le même système d'exploitation et les même versions de bibliothèque. Il est donc nécessaire d'utiliser un compromis, comme le langage Java. Un langage portable, indispensable pour ce type de projet, et pouvant s'exécuter en code natif, indispensable pour l'aspect performance de ce projet.

3.4.5 Contributions aux protocoles d'encapsulation actifs pour la haute performance

Bien que certains types de données contenus dans un flux soient facilement identifiables grâce à leur syntaxe, par exemple un document `html`, d'autres le sont beaucoup moins et tous deviennent anonymes lorsqu'ils sont compressés. Il est donc nécessaire d'encapsuler chaque paquet d'un flux de données dans une structure indiquant au minimum le service qui doit lui être appliqué dans l'équipement actif. Cet identificateur permettra aussi de déterminer le type des données transportées.

La standardisation dans les réseaux actifs s'est dans un premier temps principalement concentrée sur la définition de protocoles d'encapsulation de données actives. Les deux protocoles d'encapsulation qui ont été étudiés sont : ANEP et SAPF

3.4.5.1 ANEP

Le protocole d'encapsulation ANEP (*Active Network Encapsulation Protocol*) [47] a été défini par les principaux acteurs de la communauté réseau actif aux USA en 1997. Ce format de paquet actif est aujourd'hui largement utilisé par la communauté réseaux actifs. Il permet aux paquets d'être :

- indépendant du protocole dans lequel le paquet actif est véhiculé (IP,TCP,UDP,trame Ethernet)
- dirigé vers un EE particulier sur un nœud actif qui exécuterai plusieurs EE différents (ex : ANTS, PLAN, SmartPackets,...) en parallèle.

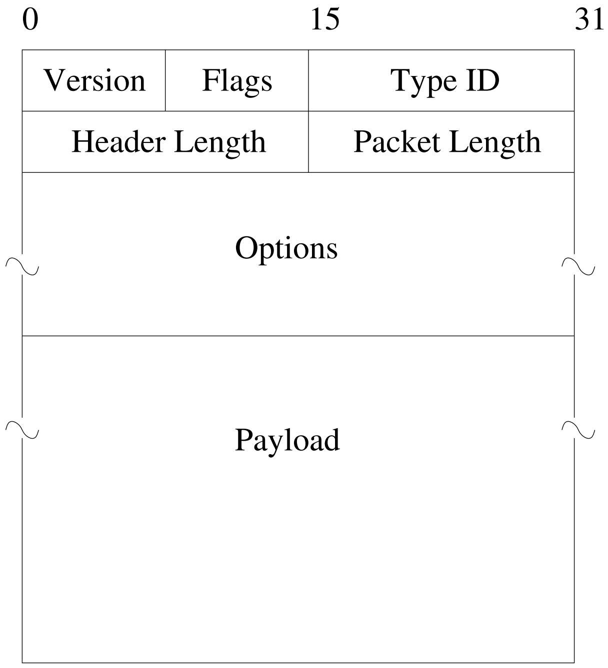


FIG. 3.12 – Structure d'un paquet ANEP.

La structure d'un paquet ANEP est illustrée sur la figure 3.12. Un paquet ANEP est composé d'un en-tête et d'une charge utile. Les champs de l'en-tête sont :

- un identificateur de version ;
- des drapeaux indiquant au routeur actif le comportement à adopter en cas de non connaissance d'un identificateur de type (0 : tenter de transmettre le paquet suivant un routage par défaut, 1 : jeter le paquet)
- un identificateur de type (`typeID`) permettant d'indiquer l'EE associé au paquet. Les valeurs de type sont gérées par l'ANANA (*Active Network Assigned Number Authority*) [48] et affecté à des environnements d'exécution.
- un nombre variable d'options ayant une représentation sur trois champs chacune (*Type/Length/Value*), appelé TLV. Quatre options sont prédéfinies :
  - une option d'adressage de la source ;
  - une option d'adressage de la destination ;
  - une option de *checksum* sur un paquet ANEP complet ;
  - une option de certification non négociée permettant à un émetteur de s'authentifier auprès des nœuds du réseau.

La taille maximale d'un paquet ANEP n'est limitée que par le protocole utilisé pour le véhiculer. On peut utiliser tout type de schéma d'adressage dans les options (*IPv6,IPv4,MAC*) étant donné que la longueur d'une option TLV peut varier. Des mises en œuvre de *codec* ANEP en langage C, OCaml[2] ou Java[3] sont disponibles sur l'Internet. De nombreux projets ont retenu ce format de paquet pour l'échange de données actives entre les nœuds actifs du réseau.

3.4.5.2 SAPF

Pour l'encapsulation de paquets actifs, un format alternatif à été défini et appelé SAPF (*Simple Active Packet Format*)[49].

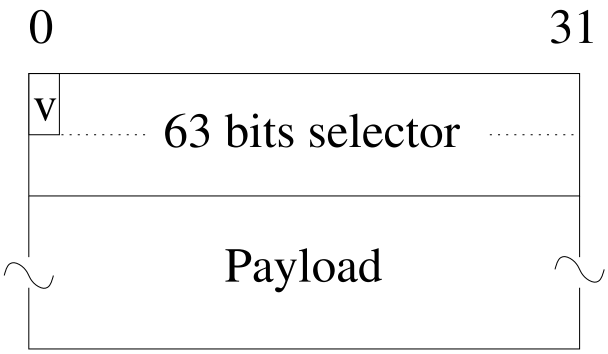


FIG. 3.13 – Une trame SAPF.

La structure d'une trame SAPF est illustrée sur la figure 3.13. Comparé au format ANEP son contenu est simplifié : un champ de version (actuellement 0), un en-tête de 63 bits représentant un label d'identification appelé aussi sélecteur, et un champs de données qui constitue la charge utile. Le sélecteur est utilisé pour faire du *Tag Switching* dans les routeurs. Trois méthodes d'allocations de sélecteurs sont offertes :

- allocation statique : des valeurs de sélecteurs sont associées à chaque type d'EE ;

[2]`http://www.cis.upenn.edu/~switchware/PLAN/`
[3]`http://www.cis.upenn.edu/~switchware/ANEP/`

- allocation dynamique par le récepteur : similaire à l'approche d'*IP switching*, l'allocation d'un sélecteur se fait par le destinataire sur demande de l'émetteur ;
- allocation dynamique par l'émetteur : l'émetteur choisit un sélecteur pour son flux actif sans coordination avec le récepteur.

L'adressage des machines sources et destinations est fait dans la charge utile. La taille maximale d'un paquet SAPF est déterminée par la taille maximale de la charge utile du protocole sous-jacent. Le démultiplexage de paquet SAPF se fait dans le noyau. Les paquets étant délivrés par la carte d'interface réseau, c'est le noyau qui décide comment traiter le paquet. Pour cela il consulte une table (*SAPF Forwarding Information Base*). Il associe un *handler* à chaque identificateur SAPF. Ce handler est ensuite responsable de traiter la charge utile ou de la transmettre à une autre carte d'interface. Les résultats obtenus montrent qu'un paquet SAPF est traité 30% plus rapidement qu'un paquet IP traditionnel utilisant ANEP[49]. Mais si la charge utile est un paquet au format ANEP, celui-ci arrive dans l'EE encore sous la forme d'un tableau d'octets. Il n'a toujours pas été filtré. Donc dans le cas d'un équipement actif qui n'héberge qu'un seul EE, par exemple ANTS, l'interprétation de SAPF devient un coût supplémentaire inutile. Ce format est utilisé dans le réseau d'expérimentation ANON [50].

3.4.5.3 Propositions pour un format de paquet actif plus léger : APF

Bien que le nombre de champs utiles ne soit pas forcément très grand, chaque en-tête de paquet reçu par un équipement actif doit être « filtré » par celui-ci. Dans le cas d'un format d'encapsulation complexe, contenant des champs facultatifs, avec des tailles variables ou contenant des informations exigeant des opérations de conversions, il est difficile d'optimiser le travail de récupération des différents champs contenus dans l'en-tête. Afin d'optimiser ce travail de filtrage sur l'en-tête, nous proposons une alternative au format ANEP appelée APF (*Active Packet Format*). Dans cette structure nous utilisons des champs dont la taille est fixée et dont la position détermine le contenant. Par exemple les n octets situés au m ème bits correspondent à une information précise comme une adresse, un identificateur de service, etc...Cette approche interdit toute modification ou ajout de la structure.

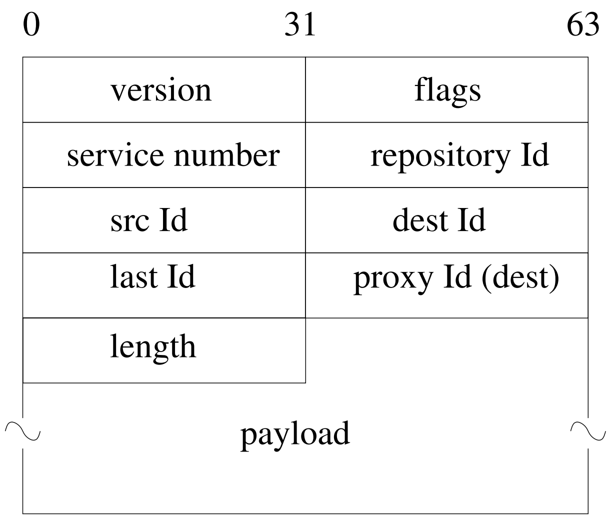


Fig. 3.14 – Structure d'un paquet APF.

Le format du paquet APF est illustré figure 3.14. Seule la charge utile peut voir sa taille varier. L'en-tête est lui figé avec chaque champs codé sur 4 octets. Nous avons fixé le numéro de version à 1 et n'avons défini actuellement aucun *flags*. L'identificateur de service est composé de deux entiers plutôt qu'une chaîne de caractères. Le premier entier est un numéro de service, le second représente l'IP physique du service repository sur lequel le service peut être téléchargé. Un service est donc identifié de manière unique par le couple IP repository + numéro de service. On parlera par exemple du service *4* du repository *10.0.0.5*. Les quatre champs suivants sont : un identificateur de source, un identificateur de destination du paquet, un identificateur du dernier nœud actif traversé, et un identificateur de proxyTAN du poste destinataire. Ces identificateurs sont les adresses IPv4 codées sur un entier de 4 octets. Le fait de ne manipuler que des entiers et non plus des chaînes de caractères nous permet d'accélérer sensiblement les opérations de démultiplexage et de création de paquets.

Le format ANEP est plus souple et autorise l'ajout de nouveaux champs à taille variable sans problème. Mais son format nécessite de nombreuses opérations pour récupérer toutes les informations qu'il contient (notamment les champs TLV (*Type, Length, Value*)). Pour chaque TLV, il est nécessaire de lire un premier entier (le type), brancher le programme sur la routine nécessaire à l'interprétation de ce type. Puis lire un second entier (la longueur) qui permet de savoir ou le champ valeur s'arrête et ou commence le TLV suivant, s'il y en a un autre qui suit. Enfin le champ valeur est un tableau d'octets qui devra alors probablement être converti ou interprété dans un type de données particulier (entier, chaîne de caractères,...). Ces opérations sont ensuite répétées sur tous les TLV suivants employés. Le format de l'en-tête SAPF est lui réduit au strict minimum. Un champs de version (sur 1 bit) et un identificateur (de 63 bits). Il a été conçu dans le but de rediriger efficacement sa charge utile vers l'EE requis d'un équipement actif qui en hébergerait plusieurs. Dans notre architecture nous considérons qu'un nœud actif exécute un unique EE. Nous n'avons donc pas besoin d'opérations de démultiplexage pour choisir le bon EE. Il reste alors à rediriger la charge utile du paquet vers le service requis. Cela pourra être réalisé efficacement grâce à un format d'en-tête simple qui ne nécessite que peu d'opérations pour être filtré.

3.5 Vers une architecture de réseau actif haute performance

Nous pensons que le gain en performance est étroitement lié à la localisation des composantes de notre architecture. On parlera :

- d'urbanisation ou de déploiement horizontal pour traiter du positionnement d'un équipement actif ou d'un service dans le réseau ;
- et de classification de services ou déploiement vertical pour trouver la couche la mieux adaptée pour l'exécution d'un service au cœur d'un nœud actif.

3.5.1 Urbanisation d'équipements actifs

Nous faisons la distinction entre les routeurs de bords et les routeurs de cœur du réseau. Il est à noter que cette distinction est classique dans l'approche *Next Generation Network*. Dans notre architecture, seuls les routeurs de bordure sont dotés de capacités de traitements (figure 3.15).

Les routeurs situés au cœur du réseau doivent garantir de très hauts débits (de l'ordre de plusieurs Gbits/s). Ils doivent rester le plus simple et le plus passif possible. En y plaçant de

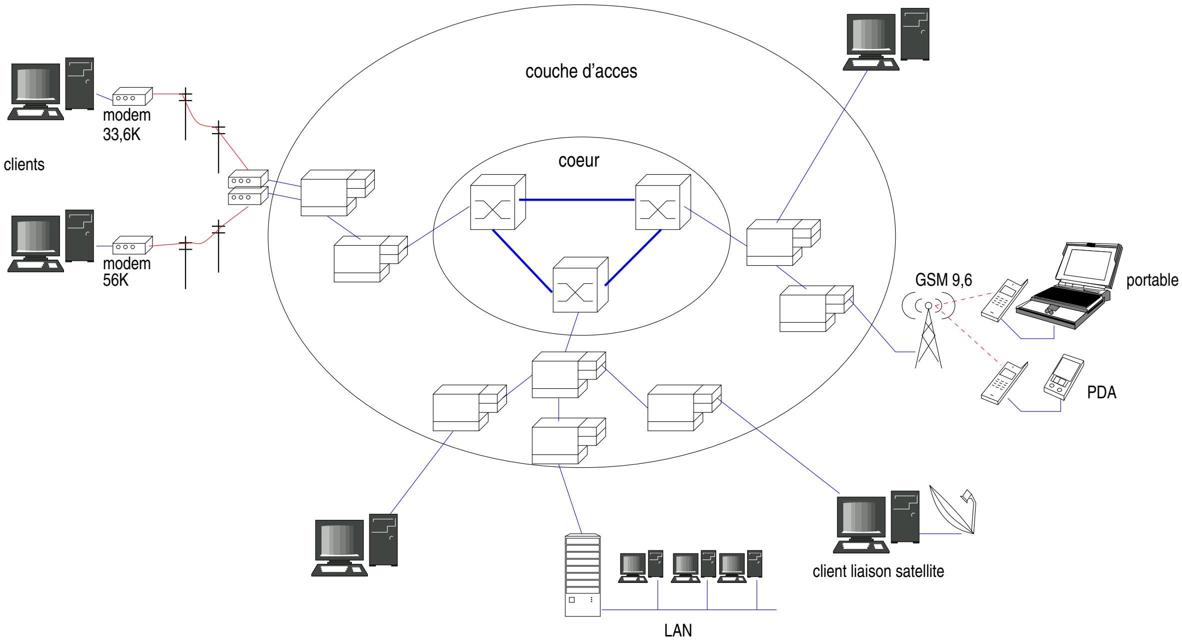


FIG. 3.15 – Équipements actifs haute performance déployés dans la couche d'accès.

l'*intelligence* et donc du traitement, on risque de réduire les performances de l'épine dorsale. Les nœuds actifs trouvent leur place dans un contexte réseau de performance plus modeste dans le réseau d'accès, où l'on rencontre des points de connections entre réseaux hétérogènes. On peut également imaginer une structuration hiérarchique des équipements actifs et ne pas limiter leur déploiement uniquement en entrée d'un domaine. Une disposition arborescente s'avérera probablement très utile pour des services de cache par exemple. Enfin on peut imaginer employer un nœud actif haute performance à l'intersection de deux réseaux d'opérateurs afin de faire du *peering*. L'intelligence embarquée dans l'équipement pourrait être utilisée pour faire respecter des accords de peering passés par les opérateurs concernés.

3.5.2 Urbanisation de services

Afin d'expliciter les différents déploiements possibles d'un service au sein d'une infrastructure active nous nous appuierons sur la figure 3.16. On appelle service composé un service constitué de plusieurs services, c'est donc un service de plus haut niveau. La figure 3.16-I présente un service composé, déployé uniquement sur le premier nœud actif traversé. Celui-ci est donc en charge d'appliquer les trois services à chacun des paquets. Afin d'optimiser l'application des services sur les paquets on peut utiliser une version de nœud actif distribué (section 3.4.2). Le flux passe ensuite à travers les autres équipements de manière transparente jusqu'à atteindre le récepteur. Le déploiement du service est transparent pour l'application. On parle de déploiement opportuniste de service car s'il n'y aucun équipement actif sur le chemin, le flux ne sera pas traité. La figure 3.16-II présente une distribution des services sur plusieurs nœuds actifs. Ce type de distribution permet de diviser la charge de calcul (ou de stockage) des nœuds. Cette topologie des services permet un traitement en mode *pipeliné* des paquets. L'inconvénient est qu'il est nécessaire de maîtriser l'infrastructure pour connaître le nombre d'équipements traversés et si ils sont opérationnels. La figure 3.16-III présente un service recyclé. Le même service est déployé sur tous les équipements actifs situés sur le passage du flux. Les paquets se voient donc appliquer le même service à chaque fois qu'ils passent dans un équipement actif. Le déploiement est également opportuniste.

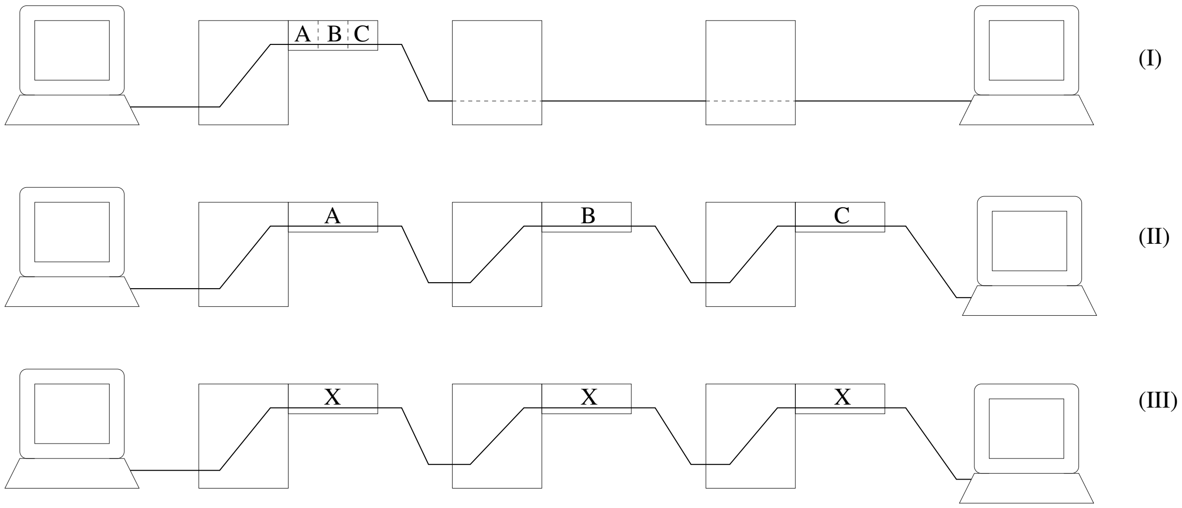


FIG. 3.16 – Différent déploiement possible de services dans un réseau de nœuds actifs.

On peut optimiser le placement d'un service sur le meilleur nœud actif, en entrée ou en sortie d'un domaine par exemple (mode proxy). On peut imaginer un nœud actif embarquant des cartes spécialisées (matériels dédiés) destinées à appliquer des services codés dans le matériel. On force alors le chemin des données vers ce nœud actif particulier pour l'application efficace d'un service.

3.6 Conclusion

Finalement nous avons proposé une architecture de réseau actif fonctionnel et performant en réduisant la latence des paquets qui traversent de tels équipements et en parallélisant le traitement appliqué aux paquets actifs. Nous proposons des solutions pour fournir des nœuds actifs plus riches et mieux adaptés aux besoins des applications et aux réalités des réseaux. Néanmoins certains points de cette architecture sont complexes et peuvent être même difficiles à mettre en œuvre. Nous avons ainsi étudié le positionnement intra-nœud d'un service (déploiement vertical), pour déterminer la couche d'exécution appropriée (espace noyau ou espace utilisateur) et discuté l'exécution d'un service dans une carte d'interface programmable. Notre architecture prévoit également un mécanisme de communication inter couche. En contre partie tout ce qui concerne la composition de services [51] et le placement efficace de services ou d'équipements actifs (urbanisation) dans un réseau (déploiement horizontal) est un problème difficile que nous avons peu discuté bien qu'il puisse être un facteur important dans notre quête de la performance. Dans le chapitre suivant nous décrivons avec précision la mise en œuvre et les solutions logicielles que nous avons employées pour mettre en œuvre notre architecture de nœud actif haute performance décrite dans ce chapitre.

Chapitre 4

Le projet Tamanoir

4.1 Introduction

Nos travaux consacrés au domaine des *réseaux actifs* hautes performances ont été développés sous la forme d'un environnement actif appelé *Tamanoir*[1], qui se propose de répondre aux problèmes de performance, d'hétérogénéité et de déploiement dynamique de services[52, 53, 54]. L'environnement *Tamanoir* fournit aux utilisateurs la possibilité de déployer et de maintenir dynamiquement des noeuds actifs, appelés TAN pour *Tamanoir Active Node*, distribués sur un réseau à grande échelle.

4.2 Déploiement des nœuds Tamanoir dans le réseau

Pour le déploiement des nœuds actifs Tamanoir dans le réseau nous proposons deux scénarios.

4.2.1 Connections par morceau : mode proxy

Un flux de données émis par une machine A vers une machine B située à une autre extrémité du réseau peut traverser un routeur non actif sans problème ; bien sur le flux ne se voit alors appliquer aucune optimisation ou adaptation sur ce type de nœud (Fig. 4.1). Le premier nœud actif traversé par le flux est situé en entrée de la couche d'accès du réseau, c'est à dire en entrée de campus par exemple. Ce premier nœud actif a donc un statut particulier puisqu'il interconnecte un réseau classique avec l'Internet. On l'appelle *proxyTAN*, et fait office de passerelle entre les clients et l'infrastructure active. Étant donné le caractère expérimental (simulation à grande échelle) de notre plate-forme, dans un premier temps, seuls les TAN ont connaissance de la localisation des autres TAN. Pour cela chaque TAN possède une table de routage qui lui permet d'atteindre les autres TAN déployés. On parle d'un *overlay* de nœuds actifs. La connexion est « morcelée » en multiples connections point à point entre les différents TAN. Cette technique nous permet de faire remonter un paquet actif du réseau vers la couche application de notre nœud actif, car du point de vue de la couche IP le paquet semble être arrivé à destination. Cette solution par morceau est bien adaptée à certains services comme l'*ActiveWeb*[55] qui permet de modifier ou réagir à un flux `http`, non actif.

[1] Le tamanoir aussi connu sous le nom de grand fourmilier se nourrit exclusivement de fourmis (30,000 par jour). Ce nom a été choisi en référence au projet ANTS (fourmis).

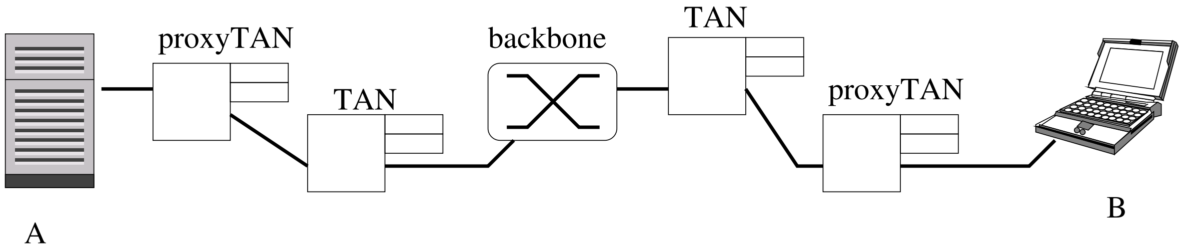


FIG. 4.1 – Topologie d'une architecture active par morceau. Les proxy Tan permettent d'accéder à l'infrastructure active déployée. Les routeurs classiques de l'épine dorsale n'interfèrent pas sur le flux.

4.2.2 Connections de bout en bout

Fournir une connexion de bout en bout signifie que les équipements actifs traversés par un flux soient capables :

- de reconnaître que le flux est actif,
- d'y appliquer un traitement,
- et de le réinjecter dans le réseau en direction du poste destinataire.

Afin de distinguer les paquets à traiter des paquets à transmettre directement, nous avons besoin d'un mécanisme de différenciation mis en œuvre sur les équipements réseaux (routeurs) existants. Une solution déjà mise en œuvre pour les protocoles de signalisation qui nécessitent une telle caractéristique est l'option *IP Router Alert* (RFC 2113[56])(figure 4.2). Cette option indique au routeur qu'il doit examiner le paquet de plus près pour déterminer si un traitement complémentaire est nécessaire.

Pour IPv6, Router Alert (RFC 2711)[57] (figure 4.3) est une option "point à point" (*Hop-by-Hop*) proposant la même sémantique. Cependant contrairement à IPv4 où seule la valeur 0 est fixée (les autres sont réservés), IPv6 a trois valeurs réservées dont l'une pour les *active network messages*. Les autres ont été assignés à RSVP et au message multicast de ICMPv6.

```
   Type    Length        Value
+--------+--------+--------+--------+
|10010100|00000100|  2 bytes value  |
+--------+--------+--------+--------+

Value:
    0        Router shall examine packet
    1-655535 Reseved
```

FIG. 4.2 – Format de l'option IPv4 Router Alert.

Le code source donné figure 4.4 présente l'activation de l'option Router Alert de l'en-tête IP d'une socket à l'aide de l'appel système `setsockopt` en C qui permet de manipuler les options associées à une socket.

```
+---+-----+--------+--------+--------+
|000|00101|00000010|  2 bytes value  |
+---+-----+--------+--------+--------+

Value:
    0       Datagram contains a Multicast Listener Discovery message
    1       Datagram contains RSVP message
    2       Datagram contains an Active Network message
    3-65535 Reserved
```

FIG. 4.3 – Format de l'option IPv6 Router Alert.

Le filtre est placé en entré de l'équipement actif Ce filtre analyse les options de l'en-tête IP. Lorsqu'un paquet actif est détecté le filtre modifie son champ IP destination et le remplace par l'adresse local (celle du TAN). Il recalcule ensuite le *checksum*. Puis, le paquet passe dans le module de routage et puisqu'il est destiné à l'adresse locale il est dirigé vers l'EE local en attente de paquet actif sur une socket. Le paquet est ensuite traité par le service puis transmis vers le destinataire. On a vu que l'IP destination est écrasée au profit de l'IP locale afin de faire remonter naturellement le paquet du noyau vers l'EE. Cette IP destination doit donc avoir été sauvegardée dans l'en-tête ANEP. L'IP source du paquet transmis est maintenant celle du TAN local. D'un point de vue IP le récepteur suivant, qu'il soit l'application réceptrice ou un autre TAN, croit que le paquet a été originellement émis depuis ce TAN. Pour certaines applications il est nécessaire de modifier à la volée lors de la sortie du paquet l'adresse IP source du paquet (et recalculer le *checksum*). Pour le protocole UDP toutes ces manipulations posent peu de difficultés. Remarquons tout de même qu'il est nécessaire de pouvoir émettre des paquets avec l'option IP Router Alert activé ce qui n'est actuellement pas possible en Java. Nous devons donc émettre les paquets ANEP à travers du code écrit en C (figure 4.4) et lié au programme Java grâce à l'interface JNI (*Java Native Interface*). Pour le protocole TCP, donc connecté de bout en bout, entre l'application émettrice et réceptrice, la mise en œuvre d'un tel dispositif de capture et modification de paquets sur le chemin des données devient plus difficile. Les services déployés dans ce cadre doivent avoir un impact limité sur les paquets (destruction, duplication interdites).

D'un point de vue expérimental sur un réseau longue distance, nos expériences ont montré que certains équipements réseau opérationnels écrasent les options IP. Donc un paquet étant marqué actif par l'application grâce à l'option IP Router Alert peut se voir amputer de ses options et donc impossible à détecter efficacement par un équipement actif effectuant une discrimination des paquets à l'aide de cette seule information. Pour anecdote, on remarque que le principe de fonctionnement décrit ci-dessus ressemble typiquement à une attaque réseau classique connue sous le nom de *man-in-the-middle attack* qui consiste à capturer les données d'un flux particulier en cours de transport, les dupliquer afin de pouvoir les analyser ou dans le pire cas, les modifier avant de les réinjecter vers le destinataire.

4.3 Déploiement de services dans Tamanoir

Deux stratégies de déploiement de services sont disponibles. La première consiste à utiliser un serveur de services Tamanoir que l'on appelle *service repository*. La seconde repose sur le

```
  /* Set the Router Alert field */
  //Get the previous value
  memset(optionsBuffer, IPOPT_END, IP_OPTIONS_SIZE);
  memset(pActSock->passiveConf.ipOptions, IPOPT_END, IP_OPTIONS_SIZE);

  len = IP_OPTIONS_SIZE;
  iret = getsockopt(pActSock->socket.socket, IPPROTO_IP, IP_OPTIONS,
(void *)optionsBuffer, &len);

  memcpy(pActSock->passiveConf.ipOptions, optionsBuffer, len);
  pActSock->passiveConf.ipOptionsLen = len;

  pBuf = optionsBuffer;
  while ( *pBuf != IPOPT_END )
    pBuf++;

  if ( pBuf - optionsBuffer > IP_OPTIONS_SIZE - 4 )
    {
      activePacket_M_errno(errno);
      return(NULL);
    }

  *pBuf = IPOPT_RA; //Option copy-class-number
  pBuf++;
  *pBuf = 4;        //Option length
  pBuf++;
  *pBuf = 0;        //Option data, cf ip_options.c
  pBuf++;
  *pBuf = 0;        //Option data, cf ip_options.c
  pBuf++;

  len = len + 4;

  while ( len & 3 )
    len++; //Align on 32bits boundary

  if ( len > IP_OPTIONS_SIZE )
    len = IP_OPTIONS_SIZE;

  iret = setsockopt(pActSock->socket.socket, IPPROTO_IP, IP_OPTIONS,
(void *)optionsBuffer, len);

  if ( iret != 0 )
    {
      activePacket_M_errno(errno);
      return(NULL);
    }
```

Fig. 4.4 – Activation de l'option IP Router Alert en C.

déploiement de services de proche en proche.

4.3.1 Déploiement de services par Service Repository

Le *Service Repository* est un serveur dédié à la distribution de services Tamanoir. Des applications de même classe (vidéo à la demande (VoD), téléphonie, broadcast radio,...) utiliseront probablement des services standards (transport de la voix, filtrage, multicast,...). Ces services pourront être proposés par les opérateurs, les ASP (*Application Service Provider*), les ISP (*Internet Service Provider*) ou les fournisseurs de services spécialisés. Ainsi, les routeurs actifs ne verront pas leur cache de services surchargé et pourront appliquer des services génériques sur différents flux. Dans notre mise en œuvre actuelle nous utilisons une adresse *URL* dans le nom du service ; les applications déploient ainsi des services connus que les nœuds actifs iront directement télé-charger sur le serveur de services (*service repository*). En guise de serveur de services nous utilisons actuellement un serveur Web *Apache*[58] configuré pour délivrer des services actifs Java (`.class` ou `.jar`). Afin de sécuriser le transport du service entre le service repository et le TAN requérant le service on utilisera le protocole `http` sécurisé, `https`. Le télé-chargement d'un service à partir du service repository est illustré sur la figure 4.5.

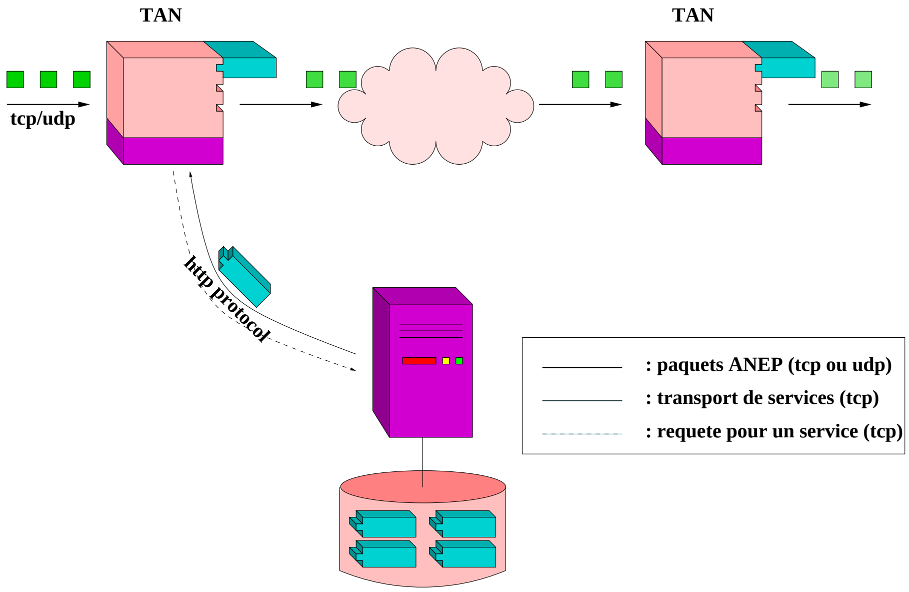


FIG. 4.5 – Déploiement de services à partir d'un service repository.

4.3.2 Déploiement de services de proche en proche

La seconde solution est d'autoriser un déploiement, à la volée des services au fur et à mesure de la traversée des routeurs actifs. Dans chaque paquet ANEP transporté, un champ est dynamiquement mis à jour avec le nom du dernier TAN traversé. C'est le champ `lastId`. Ainsi, si un TAN ne possède pas le service approprié, une demande de chargement est émise vers le dernier routeur actif TAN traversé qui lui renvoie le code du service (`.class` ou `.jar`). Les services sont transportés dans un canal de contrôle fiable (tcp) dédié. Après cette opération de déploiement, le service est chargé dans la mémoire du TAN, et est alors prêt à traiter les paquets qui exigent ce service. Le déploiement d'un service de proche en proche est illustré sur la figure 4.6.

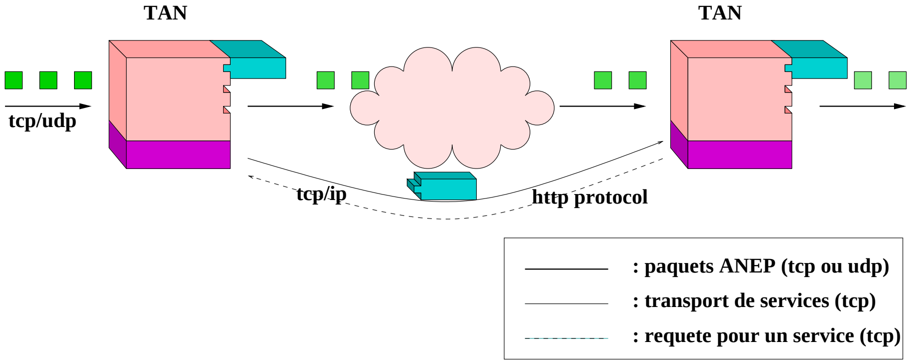


FIG. 4.6 – Déploiement de services de proche en proche

Pour des raisons de sécurité, nous n'autorisons pas le déploiement de services à partir de l'application émettrice. Cela signifie que le premier nœud actif traversé doit le cas échéant récupérer le service obligatoirement sur le service repository. Cette approche est contraire à celle de nombreux projets comme ANTS. Le projet ANTS exploite exclusivement la technique de déploiement de services de proche en proche. Le service est déployé au sein des capsules du flux de données de l'application de l'utilisateur (*in-band*) et donc en UDP (puisque ANTS n'utilise que UDP) ce qui pose parfois des problèmes lors du déploiement (perte de capsules). De plus le premier nœud actif doit faire sa demande de service (s'il ne l'a pas encore) vers l'application émettrice qui est supposée le posséder pour transporter correctement son flux.

4.4 Tamanoir : noeud actif hautes performances

4.4.1 Dans l'espace utilisateur

Un nœud actif Tamanoir simple ou *Tamanoir Active Node* (TAN) (figure 4.7) est constitué de trois éléments : un Environnement d'Exécution (EE), un gestionnaire local appelé *Active Node Manager* (ANM) et des services.

4.4.1.1 Architecture

L'**Environnement d'Exécution (EE)** peut recevoir des flux de données au format ANEP ou sous forme de données brutes transportées avec TCP ou UDP. L'EE agit comme un serveur persistant en attente de connexions, c'est à dire en écoute sur une *socket* dans un *thread*. Il y a donc un thread en attente pour chaque canal de communication possible : ANEP/UDP, ANEP/TCP, raw/UDP, raw/TCP. Les en-têtes des paquets au format ANEP sont lus par un démultiplexeur. Celui-ci extrait l'identificateur du service requis par la charge utile du paquet et en fonction de cet identificateur, utilisé comme clef de *hashage*, redirige le paquet ANEP vers le service adéquat. Si le service requis est indisponible le démultiplexeur émet une requête au gestionnaire local (ANM).

L'**ANM** se charge de déployer les services manquants. Tout d'abord, l'ANM interroge le système de fichier local dans un répertoire convenu pour vérifier si le service est déjà disponible dans le cache local. Sinon, en fonction de la position du TAN dans l'infrastructure active, l'ANM émet une requête au *Service Repository* ou au dernier nœud actif traversé. Le service requis est alors télé-chargé et stocké dans le cache local. Il est ensuite chargé en mémoire grâce à un mécanisme de chargement dynamique. Finalement le service est instancié et prêt à traiter les paquets ANEP qui exigent ce service. Le déploiement d'un service se fait sur un canal `tcp` (fiable) distinct.

Les **services** sont chargés et branchés à la demande dynamiquement dans l'EE. On appelle un service Tamanoir du code Java qui hérite d'une classe générique *Service* elle même dérivée de la classe standard *Thread*. La classe générique *Service* fournit quelques méthodes indispensables à tout service. Ces méthodes peuvent être surchargées par le développeur du service. Aujourd'hui un service Tamanoir offre une grande liberté de création de service puisqu'il autorise l'écriture de toute Application Active (AA). Un service peut ainsi :
- allouer de la mémoire ;
- créer des *threads* ;
- accéder aux ressources systèmes comme par exemple le système de fichiers ;
- créer de nouvelles connexions réseaux.

Sur un système Unix, les droits d'un service sont identiques à ceux accordés à l'application Tamanoir elle-même. Dans la pratique, il est donc préférable de créer, sur un système de type Unix, un utilisateur *tamanoir* qui exécute le nœud actif Tamanoir et dont les droits sont limités.

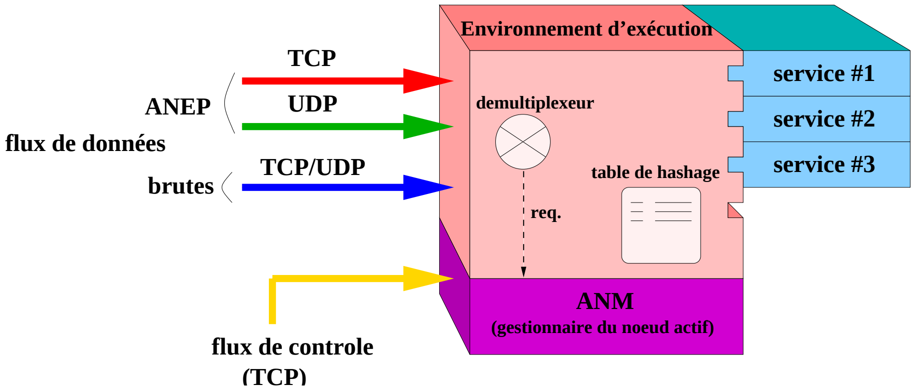


FIG. 4.7 – Un nœud actif Tamanoir.

L'architecture multi-threadée de Tamanoir permet la gestion simultanée de flux UDP et TCP. En fonction du mode employé les paquets actifs ne sont pas injectés de la même manière dans le TAN. En mode proxy les flux sont émis sur un port défini du TAN. En mode bout en bout un filtre sur le bit router alert exécuté dans le noyau capture les paquets actifs pour les injecter dans le TAN. Dans les deux cas, les flux peuvent être de type ANEP ou brut. Dans le second cas on associe un service à un numéro de port. Comme l'illustre la figure 4.8, le TAN est constitué de deux threads de démultiplexage. Ces deux threads agissent comme des serveurs en attente de messages sur les deux principaux protocoles de transport disponibles au-dessus d'IP : les protocoles TCP et UDP. Lorsqu'un paquet ANEP atteint le nœud Tamanoir, l'en-tête du paquet est analysé par le thread de démultiplexage afin de déterminer le service qui doit appliquer un traitement sur la charge utile du paquet.

Dans le cas d'un flux transporté par UDP le service est instancié une unique fois en mémoire. Il est ensuite chargé de traiter tous les datagrammes qui requièrent ce service sans pouvoir tenir compte (facilement) du fait qu'un datagramme appartient à un flux précis. On devra limiter les informations statistiques par exemple au nombre de datagrammes traités par ce service sans pouvoir caractériser le comportement d'un flux particulier. Les datagrammes de tous les flux doivent passer par le démultiplexeur.

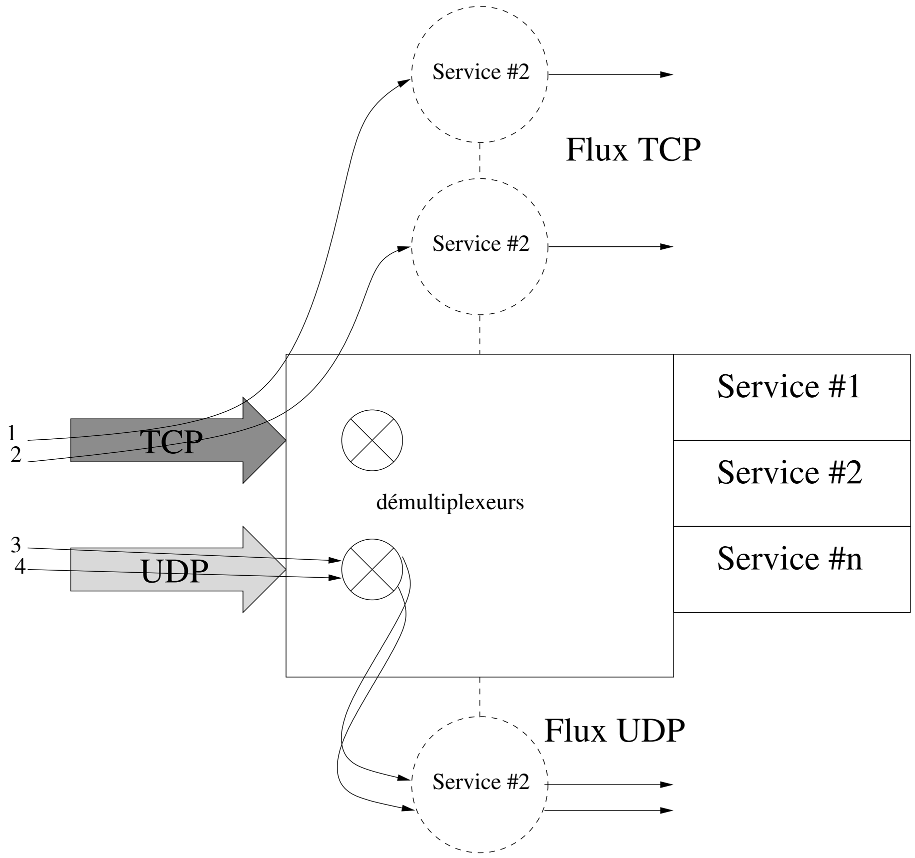


FIG. 4.8 – Un nœud actif Tamanoir avec les services disponibles localement et les instanciations de service threadé. On remarque que pour chaque flux `TCP` une instance du service est allouée. Pour `UDP` une seule instance du service est allouée pour tous les flux.

Lorsque le premier paquet d'un flux transporté par le protocole TCP atteint son thread de démultiplexage, le principe reste, dans un premier temps, le même. Le service threadé est chargé en mémoire. Mais, contrairement au cas d'un flux UDP, à chaque arrivée d'un nouveau flux TCP, l'instanciation d'un nouveau service threadé en mémoire est déclenchée (même si ce nouveau flux nécessite un service identique aux précédents). On profite du mode connecté du protocole TCP car les paquets ANEP suivants appartenant au même flux sont directement dirigés vers le service instancié sans avoir à passer par le démultiplexeur. Le grand avantage de cette approche réside au niveau du service lui-même. Il nous est maintenant possible d'associer des variables d'états à chaque flux. Ces variables nous permettent alors dans un second temps d'analyser avec une granularité extrêmement fine le comportement de chaque flux.

Ainsi, le démultiplexeur UDP est utilisé pour orienter chaque nouveau datagramme UDP vers l'unique instance du service alors que le démultiplexeur TCP est utilisé une unique fois, lors de l'établissement d'un nouveau flux. Les paquets sont alors ensuite à la charge de l'instance du service associé à ce flux.

4.4.1.2 Choix du langage de mise en œuvre

Afin de mettre en œuvre notre architecture, nous avons du faire le choix d'un langage de programmation qui fournisse le maximum de propriétés nécessaires à la réalisation d'un nœud actif fonctionnel et exploitable. Notre choix s'est porté sur le langage Java [59, 60, 61]. Les arguments énoncés ci-dessous ne sont qu'un bref résumé de la section A.1 disponible en Annexe et qui fournit une analyse en profondeur du langage Java pour la construction d'un TAN.

Java est un langage de programmation orienté objet qui a la caractéristique de pouvoir être exécuté sur de nombreuses architectures matérielles et logicielles. Ainsi, le code source est pré-compilé dans un langage intermédiaire, appelé *bytecode* qui est ensuite exécuté dans une machine virtuelle. Celle-ci fournit donc une indépendance de plate-forme indispensable pour un TAN et ses services. Le langage Java et sa machine virtuelle fournissent également un mécanisme de gestion automatique de la mémoire (appelé *Garbage Collector* (GC)), un mécanisme de gestion d'exceptions, un mécanisme de chargement dynamique de classes qui permet l'ajout dynamique de services en mémoire, et un environnement multi threadé intégré pour l'application simultané de services actifs sur différents flux. Les machines virtuelles récentes appliquent pour des raisons de performances un compilateur à la volée, dit *Just-in-Time*, sur le bytecode. Celui-ci produit ainsi du code natif directement exécutable par le processeur de l'architecture. Ainsi une routine appelée plusieurs fois n'est interprétée qu'une unique fois. Néanmoins, choisir Java est le résultat d'un compromis. Bien que ce soit un langage très attractif et qui réponde à de nombreux critères parfaitement adaptés à l'écriture d'un environnement d'exécution actif certaines caractéristiques peuvent être considérées comme limitatives dans le contexte d'un nœud actif haute performance. Par exemple, nous ne maîtrisons pas (ou peu) le déclenchement du *Garbage Collector* chargé de libérer les zones mémoires inutilisées. Les machines virtuelles fournissent des mécanismes de vérification supplémentaires coûteux, non désactivables qui testent en permanence la stabilité de la JVM. Enfin la machine virtuelle fournit une abstraction du système sous-jacent qui s'avère dans un contexte haute performance difficile à contourner. En conclusion, le langage Java et son environnement d'exécution que fournit sa machine virtuelle sont un excellent compromis pour la construction d'un EE actif et de ses services.

4.4.2 Dans l'espace noyau

4.4.2.1 Netfilter

Dans la version 2.0 du noyau Linux, est apparu un outil appelé *IpFwadm*. Il permet de faire du filtrage de paquets. La mise en place d'un pare feu (*firewall*) était devenu plus aisée. Cet outil a été ensuite réécrit pour la version 2.2 du noyau Linux, ce qui a donné *IpChains*. Il offrait plus de fonctionnalités et surtout était plus simple à utiliser. Mais certains problèmes déjà présents dans la version précédente persistaient. Notamment, une programmation du noyau trop complexe qui rendait le développement de ses propres modules dans le noyau Linux difficile ou, la création d'une règle de filtrage indépendamment de l'adresse de la carte réseau n'était pas possible. Enfin la version 2.4 du noyau Linux s'est particulièrement enrichie dans le domaine du filtrage des paquets, de la translation d'adresse (NAT ou *Network Address Translation*) et de la modification des paquets à la volée. Il devient possible de réaliser des passerelles et des pare feux performants. Afin de pouvoir réaliser de telles opérations (filtrage, modifications de paquets, NAT) un canevas Netfilter a été développé. Il s'inscrit bien sûr dans la suite logique des deux logiciels IpFwadm et IpChains. Ce canevas est situé dans le noyau Linux soit en tant que module (chargé au moment souhaité) ou bien lié statiquement au noyau monolithique de Linux. Pour un utilisateur avancé, le canevas permet de réaliser les trois opérations majeurs suivantes :

1. Filtrer des paquets, principalement pour assurer des fonctions de pare feu : par exemple on interdira le passage de paquets provenant de l'Internet de s'adresser au port 80 (http) de notre machine.
2. Réaliser des translations d'adresses (NAT) : cette fonction est particulièrement utile lorsque l'on veut faire communiquer un réseau privé, c'est à dire avec des adresses IP privées (192.168.x.x par exemple), avec l'Internet.
3. Marquer des paquets, pour leur appliquer un traitement spécial : cette fonction permet d'appliquer des priorités différentes aux paquets dans les files d'attentes d'un équipement réseau.

Afin de paramétrer ces trois opérations majeures, un outil, qui s'exécute dans l'espace utilisateur, est disponible. Il s'agit de *IpTables*.

Regardons maintenant comment fonctionne concrètement Netfilter. Nous nous limitons ici au protocole IPv4 car Netfilter n'a pas la même structure en fonction du protocole employé. Chaque protocole définit des *hooks* (accroches) qui sont des points précis dans le trajet du paquet dans la pile du protocole. IPv4 en définit 5 (voir Fig. 4.9).

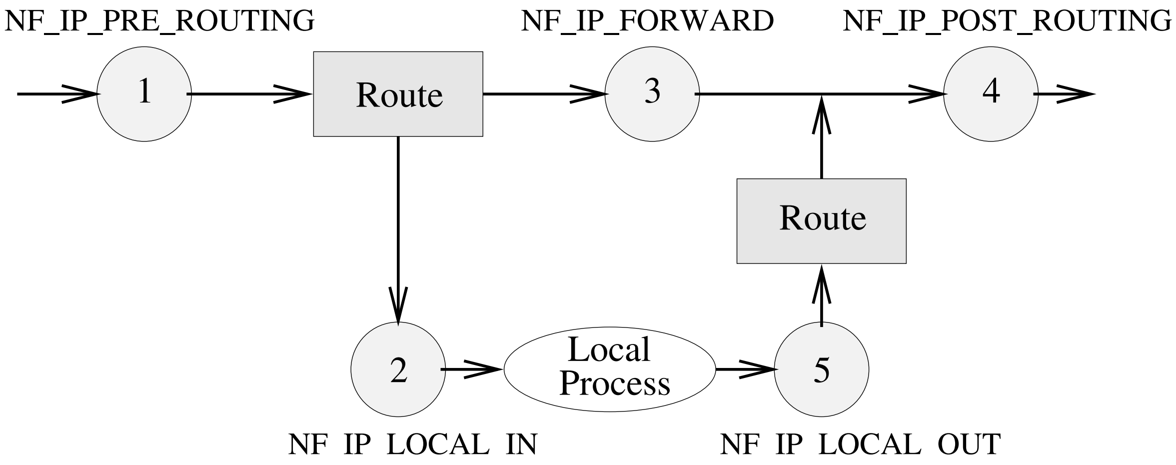


Fig. 4.9 – Hooks de Netfilter

A chacun de ces points, le protocole va appeler le canevas Netfilter et lui passer le paquet ainsi que le numéro du hook. Netfilter prend alors possession du paquet et effectue des manipulations sur ce dernier.

Ce qui rend particulièrement attractif Netfilter c'est la possibilité de pouvoir écouter les points d'accroche et d'ainsi scruter les hooks afin de récupérer les paquets qui les activent.

Observons en détail la traversée d'un paquet :

1. lorsqu'un paquet arrive dans le système en quittant la carte d'interface réseau, il active le hook NF_IP_PRE_ROUTING ;
2. les fonctions associées sont appliquées sur le paquet. Le paquet entre ensuite dans le code de routage. Trois actions sont possibles :
   (a) si le paquet n'est pas routable (adresse IP incohérente par exemple) : il est détruit ;
   (b) si le paquet est destiné à une autre interface : il poursuit son chemin et atteint le hook NF_IP_FORWARD ;
   (c) si le paquet est destiné à un processus local : il est transmis au hook NF_IP_LOCAL_IN.
3. si un paquet à été généré par un processus local, il est envoyé sur le hook NF_IP_LOCAL_OUT ;
4. le paquet passe par le dernier hook NF_IP_POST_ROUTING avant d'être pris en charge par la carte réseau de sortie.

Dans la pratique, il est rare que tous les hooks soient associés à des fonctions de traitement. L'activation d'un hook dépend des traitements effectués sur les paquets réseaux. Par exemple, lors d'un filtrage avec IpTables, Netfilter se sert uniquement de NF_IP_LOCAL_IN, NF_IP_LOCAL_OUT et NF_IP_FORWARD pour accrocher des fonctions de filtrage. Bien sur le service devra être écrit en C pour l'espace noyau Linux, ce qui peut représenter une source de problèmes pour sa portabilité.

4.4.2.2 Déploiement de services noyaux dans Tamanoir

Comme nous l'avons décrit dans le chapitre précédent, nous classons les services en fonction des ressources qu'ils consomment et de ce fait plaçons leur exécution dans l'espace le mieux adapté. Nous appelons services noyau des services actifs légers exécutés dans le noyau du système d'exploitation **Linux** (version 2.4.x minimum), sous forme de modules écrit en C chargés dynamiquement dans l'espace noyau grâce à Netfilter.

Nous mettons en oeuvre des mécanismes de communications entre un service Tamanoir écrit en Java et s'exécutant dans l'espace utilisateur, et un service écrit en C exécuté dans l'espace noyau. Ce service noyau reçoit des instructions du service Java pour éviter la remonté de paquets dans l'espace utilisateur. Cette fonctionnalité est utile pour alléger la charge de traitements de Tamanoir dans l'espace utilisateur. Avec une politique *best effort* les paquets non traités dans ce TAN pourront peut-être être traités dans le TAN suivant. La figure 4.10 représente de manière schématique les échanges de messages entre les services exécutés chacun dans leur espace respectif.

Les figures 4.11 et 4.12 illustrent ce mécanisme facultatif de remontées des paquets. Sur la première figure, on autorise un paquet sur deux à remonter dans l'espace utilisateur. La seconde plus réaliste indique que le service exécuté dans l'espace utilisateur traite 500 paquets puis demande au service noyau de ne pas faire remonter les 500 paquets suivants. Sur ces deux figures on voit également que la taille du paquet influence peu le temps de traversée du noyau. Par contre elle influence de manière significative le temps de traversée de l'espace utilisateur (entre 1 et 9 ms).

Le TAN exécuté dans l'espace utilisateur est utilisé pour déployer et charger dynamiquement de nouveaux services actifs dans l'espace noyau, par un appel à la commande `insmod`.

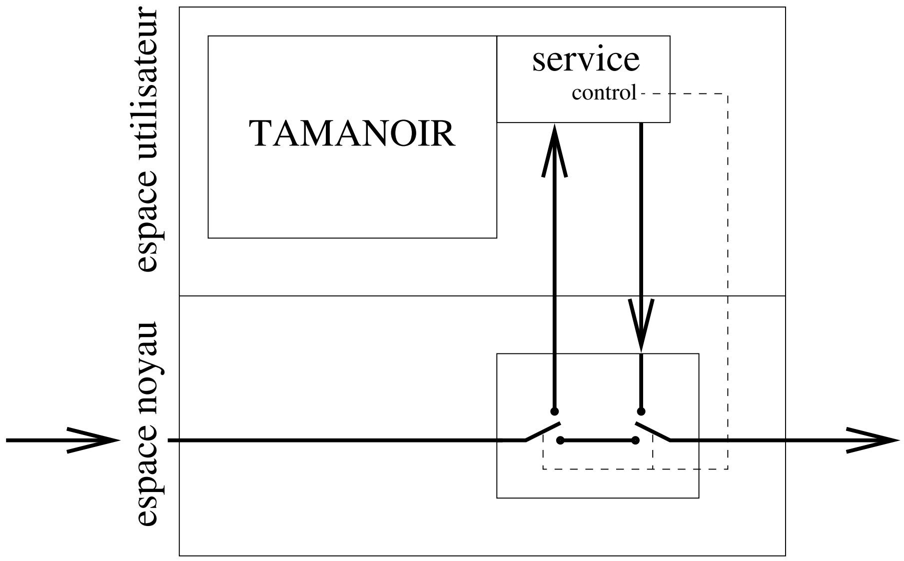


FIG. 4.10 – Mécanisme de communication entre le service exécuté dans l'espace utilisateur et son homologue exécuté dans l'espace noyau.

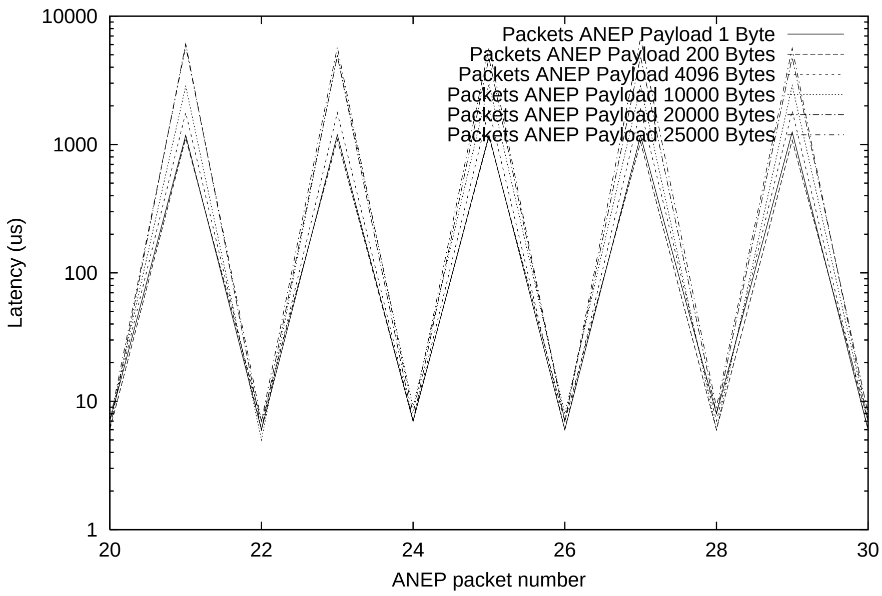


FIG. 4.11 – Les paquets pairs sont transmis par le noyau, les autres sont modifiés dans l'espace utilisateur.

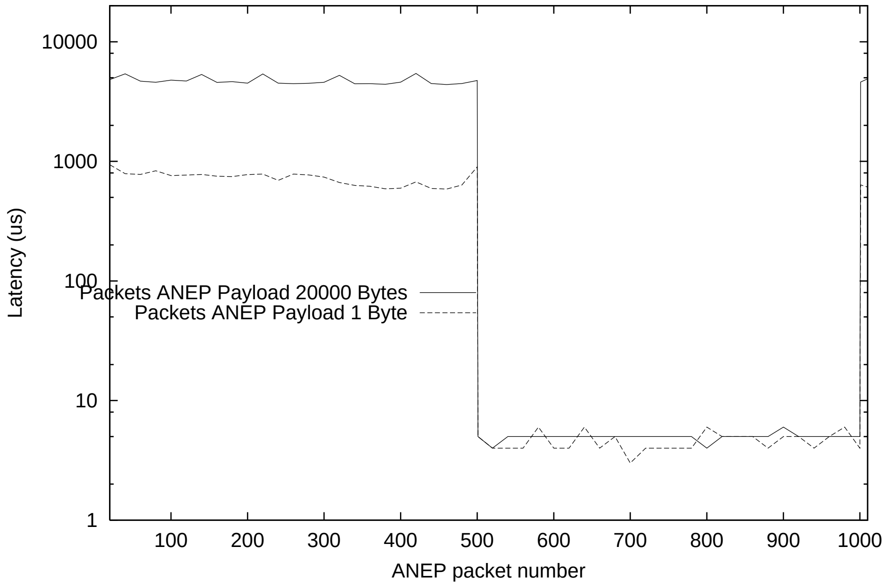


FIG. 4.12 – 500 paquets traversent la JVM, les 500 suivants sont transmis par le noyau directement sur l'interface de sortie.

4.4.3 Dans l'espace des ressources distribuées

Nos objectifs sont d'accélérer l'exécution de services sur les flux actifs en parallélisant le traitement, et d'augmenter la capacité de stockage d'un nœud actif. Pour cela nous avons créé un nœud actif Tamanoir qui s'appuie sur un système distribué de type grappe de PC (cluster). Pour ce faire nous proposons de détourner et d'améliorer la fonctionnalité originelle du projet LVS (Linux Virtual Server)[62], qui est de distribuer des requêtes sur une batterie de serveurs, pour distribuer les flux et donc la charge de traitement des paquets actifs.

4.4.3.1 Linux Virtual Server

Un serveur virtuel Linux (*Linux Virtual Server* : LVS) [62] est constitué d'un groupe de serveurs, appelés *realservers* et d'un directeur (ou *director*). Cet ensemble de machines forme le serveur virtuel qui apparaît comme une seule machine pour les clients. On appelle directeur (ou *load balancer* ou frontal) le nœud qui exécute le code `ipvs` qui se présente sous la forme d'un module Netfilter. Les clients se connectent au directeur. Le directeur transmet les paquets au realservers. Le directeur n'est autre qu'un routeur avec des règles spécifiques qui font fonctionner le LVS. Les *realservers* ou *backends* (BE) permettent le déploiement des services. Ils gèrent les requêtes des clients. Un LVS offre des services de plus grande capacité en terme de débits et de disponibilités par rapport à un serveur unique. Chaque client croit être connecté directement au *realserver* et vice-versa. Ni les clients, ni les *realservers* n'ont le moyen de détecter qu'un *director* est intervenu dans la connexion.
Un LVS n'est à la base pas une grappe de machines destinée à calculer des petites parties d'un grand problème de façon coopérative. Les *realservers* ne coopèrent pas, ils n'ont pas connaissance de la présence de leurs voisins et sont connectés à un client. LVS propose de

transmettre les paquets de trois manières différentes :

- **LVS-NAT**, basé sur la translation d'adresse (NAT).
  - **avantages** : tout système d'exploitation supportant `TCP/IP` peut exécuter un realserver ; les realservers peuvent utiliser des adresses `IP` privées, seul le directeur nécessite une adresse `IP` publique.
  - **inconvénients** : supporte mal le passage à l'échelle (mauvaise extensibilité), une vingtaine de realservers seulement et le directeur se retrouve saturé car il doit modifier les en-têtes des paquets entrants et sortants. Donc le directeur devient trop vite un goulot d'étranglement.
- **LVS-TUN** (TUNnelling) où le paquet est IPIP encapsulé et transmis au *realserver*.
  - **avantages** : le directeur n'a qu'à distribuer les requêtes sur les différents realservers suivant un algorithme de distribution donné. Les réponses sont émises directement vers les clients par les realservers. Plus de 100 realservers peuvent être gérés par le directeur avant qu'il ne devienne un goulot d'étranglement.
  - **inconvénients** : le système d'exploitation des realservers doit fournir un support pour l'IP tunneling (IP encapsulation).
- **LVS-DR** (Direct Routing) où les adresses MAC des paquets sont changées et le paquet transmis au *realserver*.
  - **avantages** : tout comme LVS-TUN, le directeur gère la demi connexion entre le client et le serveur et les paquets de réponses suivent ensuite une route du réseau différente. Là aussi le passage à l'échelle est bien supporté.
  - **inconvénients** : exige que l'interface du directeur soit dans le même segment physique que l'interface du realserver.

4.4.3.2 Tamanoir cluster

Comme il à été dit en introduction l'usage premier de LVS est de distribuer la charge d'un serveur Web, DNS ou Proxy Squid en distribuant les connexions sur un ensemble de realservers. Dans le cas d'un serveur Web par exemple, chaque realserver est un miroir du serveur web. Le director se charge de distribuer les requêtes `GET` provenant des navigateurs des clients sur l'un des realservers qui honorera alors la requête.

Dans le projet Tamanoir nous détournons l'usage de LVS pour distribuer non plus des requêtes Web par exemple mais des flux TCP de paquets actifs. Le director est employé à rediriger chaque nouveau flux vers un realserver qui exécute un EE Tamanoir indépendant. La figure 4.13 illustre l'architecture d'un Tamanoir cluster où l'EE Tamanoir exécuté dans l'espace utilisateur est dupliqué sur chaque realserver. Parmi les trois modes de traitements distribué que nous proposions dans le chapitre précédent, section 3.4.2, c'est donc le troisième illustré figure 3.8-c que nous avons choisi de mettre en œuvre à l'aide du projet LVS.

Lorsqu'un flux actif est dirigé par le frontal vers l'un des BE, le même scénario de déploiement de service que pour un EE Tamanoir classique exécuté dans l'espace utilisateur s'applique. Cela signifie qu'un BE pourra appliquer un service que son voisin ne possède peut-être pas encore. Nous n'avons pas encore prévu à ce jour de mécanismes de déploiement de services intra-nœud. La figure 4.14 présente la sortie de la commande `ipvsadm -L` qui permet d'afficher la configuration et l'état du frontal (director). On voit ici qu'il a été configuré pour distribuer des connections sur trois *back ends* (BE) (realservers) (`192.168.0.2, 3` et `4`) de même poids, cela signifie que la grappe est constituée de trois systèmes aux caractéristiques identiques. On s'aperçoit également que le frontal (`192.168.1.6`) applique un algorithme du

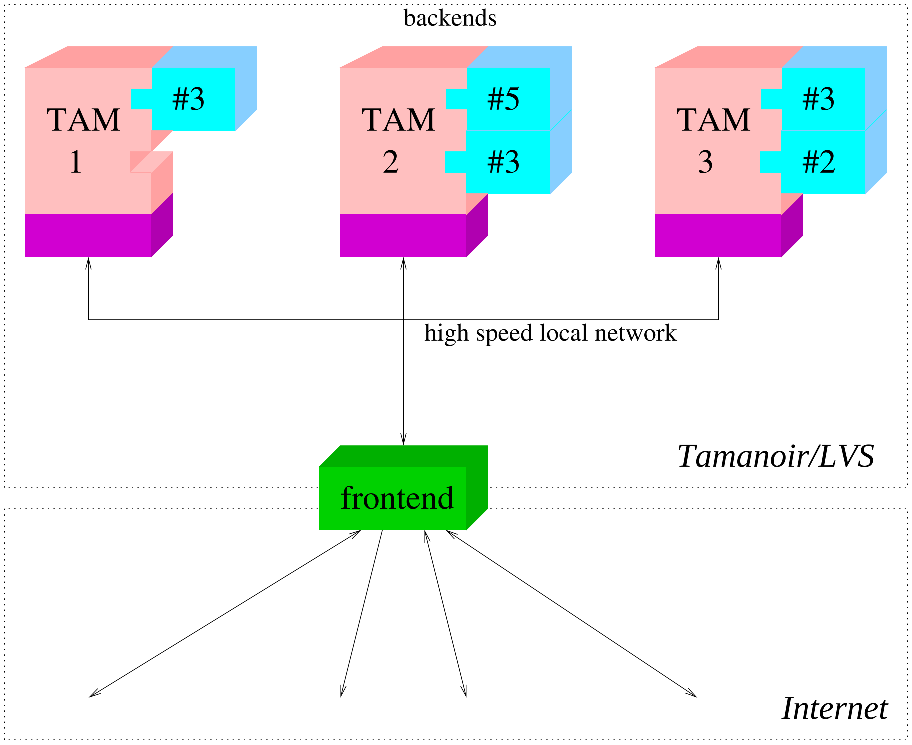


FIG. 4.13 – Un nœud actif Tamanoir cluster. L'EE Tamanoir de l'espace utilisateur est dupliqué sur plusieurs nœuds (realservers ou backends) auxquels on accède à travers un frontal (director) qui distribue les connections.

tourniquet (`rr=round robin`). C'est l'algorithme de distribution proposé par défaut.

```
\$ ipvsadm -L

IP Virtual Server version 1.0.7 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  192.168.1.6:tamanoir rr
  -> 192.168.0.4:tamanoir         Route   1      0          0
  -> 192.168.0.3:tamanoir         Route   1      0          0
  -> 192.168.0.2:tamanoir         Route   1      0          0
```

Fig. 4.14 – Liste le statut des BE et du frontale

4.4.3.3 Distribution des connections

Le director exécute un module de filtrage sur les paquets TCP et un port destination qu'on lui a spécifié. Le numéro de port correspond généralement à un service (port 80 pour un serveur Web, port 23 pour le service telnet,...). Lorsqu'une nouvelle connexion est établie, l'algorithme de distribution du module lui associe un realserver à qui il transmet ensuite les données. Le module tient donc à jour une table des couples identificateurs de flux, realserver. L'algorithme de distribution est par défaut un simple algorithme du tourniquet (*Round- Robin* (RR)). Afin de conduire moins rapidement à un déséquilibre de charge des realservers, l'algorithme peut être affiné en associant à chaque realserver un coefficient de capacité de traitement. Une sorte de pondération. On utilise alors un algorithme du tourniquet pondéré (*Weighted Round-Robin*(WRR)). On peut également utiliser un algorithme dont la pondération est basée sur le nombre de connexions par back-ends. On parle de *Least Connection Scheduling (LC)*. Une combinaison de WRR et LC (WLC) peut également être envisagée.

4.5 Conclusion

Ce chapitre présente le projet Tamanoir qui résulte de la mise en œuvre de l'architecture de réseau et de nœud actif haute performance décrite dans le chapitre précédent. Nous avons d'abord proposé deux modes de déploiement : mode proxy et mode bout en bout. Le premier s'avère le plus facile à déployer puisque basé sur une redirection explicite des paquets actifs sur une machine et un port particulier. Néanmoins cela impose la gestion d'un *overlay* de nœuds actifs. Un nœud actif doit donc acquérir la connaissance de ses voisins de manière autonome, opération redondante avec les équipements de routage classique. La seconde approche simplifie le travail du nœud actif ; néanmoins celui-ci doit embarquer un mécanisme de détection et de capture des paquets actifs. De plus l'application émettrice du flux actif doit marquer les paquets (dans l'en-tête IP) comme étant à traiter. La mise en œuvre de cette opération peut limiter la portabilité des applications écrites en Java par exemple, car on doit passer par un bout de code C (JNI) pour effectuer cette opération de marquage. Ensuite nous avons présenté la mise en œuvre de deux modes de déploiements de services actifs. Le mode basé sur Service Repository exploite un serveur Web qui sert pour l'occasion des fichiers `.class` ou `.jar`. Le mode de proche en proche utilise un canal fiable secondaire (*out of band*) pour communiquer avec le nœud précédent. Nous motivons ensuite brièvement le choix du langage Java pour la mise en œuvre de l'EE Tamanoir et des services qui s'y exécutent dans l'espace utilisateur.

Un introduction et une argumentation plus complète sur l'usage de Java dans un nœud actif est disponible en annexe. Nous présentons nos travaux sur le déploiement de services légers dans le noyau. Nous l'utiliserons ensuite pour filtrer et capturer les paquets actifs (mode bout en bout). Nous l'emploierons également dans notre architecture distribuée pour distribuer les flux actifs sur plusieurs nœuds Tamanoir et ainsi paralléliser le traitement.

Chapitre 5

Services actifs et environnement de développement

5.1 Introduction

Ce chapitre présente une bibliothèque de services actifs mis à disposition avec l'environnement Tamanoir et qui représentent les différents niveaux de déploiement. Le premier (PingS) introduit un service léger typique des services employés dans le plan contrôle. Le second est un service lourd (GzipS) que l'on peut rencontrer dans le plan données. Nous introduisons également un service original de stockage dans les réseaux actifs exploitant une solution externe de réseaux logistiques (IBP). L'objectif est d'illustrer la possibilité d'interfacer un service Tamanoir avec des applications externes. Nous présentons enfin un environnement de développement et la mise en œuvre de nœuds actifs.

5.2 Un exemple de service léger : PingS

La classe *PingS* (*Ping Service*) est donnée en exemple figure 5.1. Un service Tamanoir exécuté dans l'espace utilisateur, donc écrit en Java, est dérivé d'une classe *Service* générique héritant elle même de toutes les méthodes et caractéristiques de la classe *Thread*. D'un point de vue pratique, le *PingS* a pour fonction de re-émettre un paquet arrivé à destination d'un TAN, ou de le transmettre vers sa destination. Son fonctionnement est donc similaire au ping classique ICMP (*Internet Control Message Protocol*). La méthode `process()` n'est pas utilisée (surchargée) dans ce service car *PingS* n'effectue aucun traitement sur la charge utile des paquets. On réserve généralement la méthode `process()` au traitement de la charge utile du paquet ANEP. Ce service très simple compare le champ destination du paquet ANEP avec l'identifiant du TAN récepteur. Si il y a concordance, le service inverse les champs source et destination, reconstruit un paquet ANEP et le renvoi en direction de l'émetteur. Sinon le paquet est simplement transmis vers le TAN suivant. Ce service est en fait associé à une application cliente appelé *Pingtan* permettant de réaliser des mesures de temps d'aller/retour (*round-trip time*) et de vérifier l'état des nœuds actifs.

```
class PingS extends Service {
    PingS() { }

    public void run()
    {
        try {
            while(!FINISHED) {
                if(net.localhost.equals(net.destId)) {
                    // -- PING --
                    net.makeANEPkt( net.serviceNameinBytes,
                                    net.destId.getBytes(),
                                    net.srcId.getBytes(),
                                    net.destId.getBytes(),
                                    net.proxyTANId.getBytes(),
                                    net.getANEPkt().payload );
                    net.sendANEPkt();
                } else {
                    // -- FORWARD --
                    try {
                        //send ANEP pkt to the next node
                        net.sendANEPkt_next();


                        if(net.proto == TANet.TCP) {
                           net.recvANEPkt_next();
                           net.sendANEPkt();
                        }
                    } catch (BadANEPversionException e) {
                        System.err.println( ... );
                    } catch (InterruptedException e) {
                        System.err.println( ... );
                    }
                }
                recv(); // wait for the next ANEP packet
            }
        } catch (IOException e) {
            System.err.println( "PingS : run() : " + e );
        }
    }

    public void process()
    {
    }
} // class PingS
```

FIG. 5.1 – la classe *PingS* (Ping Service) hérite de la classe générique Service

5.3 Un service de calcul lourd : GzipS

La classe *GzipS* (*Gzip Service*) dont le code source est donné en exemple figure 5.2, est également dérivée d'une classe *Service* générique héritant elle même de toutes les méthodes et caractéristiques de la classe *Thread*. Ce service permet de compresser à la volée la charge utile d'un paquet ANEP à l'aide de la classe `GZIPOutputStream` fournie en standard par Java. Lorsque le service GzipS reçoit le paquet que le démultiplexeur du TAN s'est chargé de lui transmettre, il passe comme paramètre à la méthode de compression (appelée `GZIPdata()`) la charge utile du paquet. La méthode de compression renvoie alors un tableau d'octets (`bufout`) qui contient la charge utile du paquet compressé. Puis un nouveau paquet ANEP est construit avec comme charge utile `bufout` et finalement transmis au nœud suivant. Le service se met ensuite en attente d'un autre paquet à compresser (`recv()`).

```
class GzipS extends Service {
    byte[] bufout;

    GzipS() {}

    public void run()
    {
        try {
            while(!FINISHED) {
                process();
                net.makeANEPkt( net.serviceNameinBytes,
                                net.srcId.getBytes(),
                                net.destId.getBytes(),
                                net.lastId.getBytes(),
                                net.proxyTANId.getBytes(),
                                bufout
                                );
                net.sendANEPkt_next();
                recv();    // wait for the next ANEP packet
            }
        } catch (IOException e) {
            System.err.println( "GzipS : run() " + e );
        }
        net.close();
        if(DBG) System.out.println("GzipS : run() thread done.");
    } // run()

    public void process()
    {
      bufout = GZIPdata( net.getANEPkt().payload );
    }
    // GZIPdata : return a byte array of compressed data in the GZIP
    // format.
    public static byte [] GZIPdata( byte [] bufin )
        throws IOException
    {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        GZIPOutputStream gzos = new GZIPOutputStream(baos);

        gzos.write(bufin,0,bufin.length);
        gzos.close();
        return (baos.toByteArray());
    }
} // class GzipS
```

FIG. 5.2 – la classe *GzipS* (Gzip Service)

5.4 Service actif avec fonctions de haut niveau

Le projet Tamanoir exploite une approche originale pour stocker des données à la volée dans le réseau en s'associant à IBP(*Internet Backplane Protocol*)[63, 64, 65]. Le projet IBP offre la possibilité de gérer des ressources de stockage de données (dépôts) dans le réseau en les exposant à la manière d'Internet, c'est à dire en partageant au mieux (*best effort*) ces ressources. Nous commençons par décrire le projet IBP. Nous présentons ensuite la mise en œuvre d'un service actif Tamanoir simple de stockage de données à la volée dans un dépôt IBP. Finalement, nous décrivons des nouveaux services réseaux devenus aisément envisageables grâce à la combinaison des projets Tamanoir et IBP. On trouvera des résultats de mesures du surcoût de stockage à la volée dans la section 6.6.1.

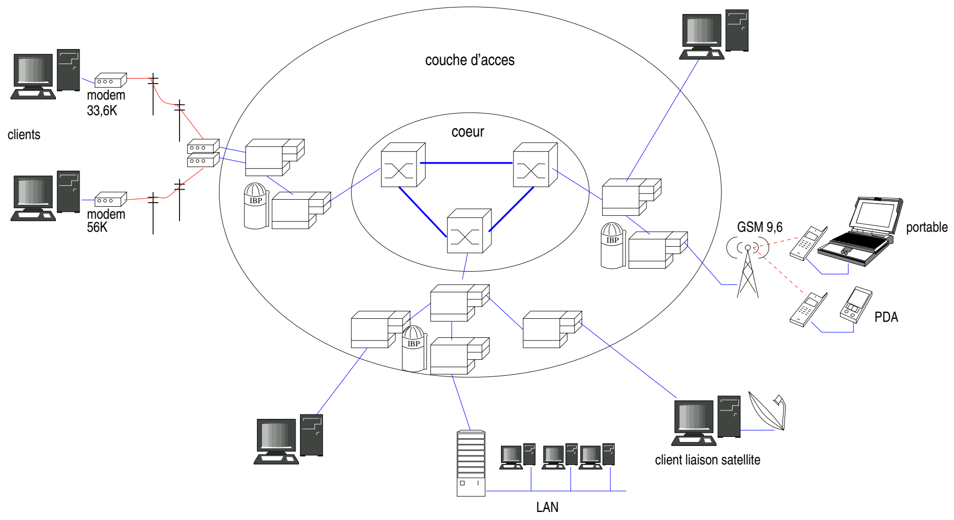


FIG. 5.3 – Les dépôts IBP peuvent être distribués à coté des équipements actifs Tamanoir

5.4.1 Description du projet IBP

Cette section décrit l'infrastructure de réseau logistique dont IBP (*Internet Backplane Protocol*, Figure 5.4) constitue la brique de base.

IBP[66] est un logiciel intermédiaire (*middleware*) construit pour partager des ressources distribuées de stockage. On appelle ces ressources de stockage des dépôts. Ce qui rend IBP unique est sa manière d'exposer les ressources : une application peut allouer une quantité d'espace de stockage pour une période de temps prédictible sur un dépôt donné. Cet aspect clef du modèle de stockage d'IBP qui est la capacité d'allouer un espace sur une ressource réseau partagée peut être perçu comme une allocation mémoire (`malloc`) à la C sur une ressource Internet avec certaines différences comme par exemple une limitation dans le temps de l'espace alloué afin de prévenir les attaques de type *Denial-of-use*. D'un point de vue réseau, la conception d'IBP respecte la conception du protocole Internet IP. Tout comme IP est un service qui abstrait la couche liaison (*link layer*), IBP est un service qui abstrait les blocs de données, gérés comme des tableaux d'octets. Le protocole IP seul ne garantit pas de nombreuses caractéristiques utiles, et nécessite d'être associé à un protocole de transport.

De manière similaire, de nombreuses applications peuvent exiger des allocations fiables et donc trouvent la sémantique d'IBP trop faible. Pour cette raison, l'*exNode*, ou *external Node*, qui suit le même concept que l'`i-node` Unix, agrège des ensembles d'allocations de stockage sur l'Internet (tout comme on agrège des blocs sur un unique volume disque). Comme sa définition est très générique, tout type d'application peut être adapté. La nature exposée des allocations IBP rend les tableaux d'octets parfaitement adaptés pour de telles agrégations. Le logiciel IBP, développé en C, pour la partie serveur, et C ou Java pour la bibliothèque cliente a été testé avec succès sur différents systèmes d'exploitations et architectures matérielles. La bibliothèque *exNode*, développée en C et Java, permet de sérialiser son contenu en utilisant XML, afin d'ouvrir l'usage d'*exNode* à des outils basés sur XML et interopérables avec des systèmes et des protocoles XML. On trouvera des explications sur le rôle des briques *L-Bone*, *Data mover* et le *Logistical set of tools* de la figure 5.4 dans [66]. Nous n'employons pas ces fonctionnalités dans les services actifs de Tamanoir.

5.4.2 Service de cache systématique pour Tamanoir

Un nœud actif Tamanoir (TAN) est un processus autonome exécuté sur un nœud en attente de traiter de nouveaux flux actifs. Un dépôt IBP est également un processus autonome qui peut être exécuté sur un nœud accessible par le réseau. Comme le montre la figure 5.4, Tamanoir et ses services accèdent directement à la couche IBP.

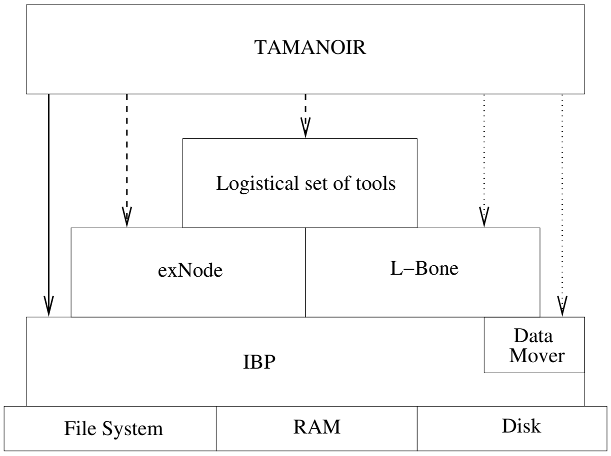


FIG. 5.4 – Architecture de Tamanoir/IBP.

Un service Tamanoir communique avec un dépôt à travers une socket et un protocole de transport fiable (`tcp`). Remarquons qu'il n'est pas nécessaire pour un dépôt IBP d'être exécuté sur le même nœud parce que les communications entre dépôts et service actif sont réalisées par l'intermédiaire de sockets classiques. Le client original Java a été réutilisé pour mettre en œuvre le code de notre service actif puisque ce dernier est considéré du point de vue du dépôt IBP comme un client. Nous devons donc instancier les classes IBP cliente dans chaque service qui nécessite de communiquer avec un dépôt IBP. Ces classes fournissent constructeurs et méthodes pour créer des *Capabilities*, qui sont des références vers des allocations IBP, sur n'importe quel dépôt IBP sur lequel le service actif voudrait effectuer des opérations d'écriture, de lecture ou de gestion de données distante.

Le code source dans la figure 5.5 présente un service pour un nœud actif Tamanoir. Ce service est conçu pour cacher systématiquement les données d'un flux dans un dépôt IBP et de les transmettre immédiatement. Lorsque le service est instancié, le constructeur `Ibp_testS()` est appelé. Son rôle est d'allouer une zone de stockage dans le dépôt IBP choisi et de récupérer la capabilité sur cette zone. Ensuite, la boucle principale de la méthode `run()` se charge de copier systématiquement la charge utile du paquet ANEP dans le dépôt IBP et de transmettre le paquet ANEP entier vers le prochain nœud actif ou l'application réceptrice. Dans ce service nous n'effectuons pas de traitements sur les données. Nous nous contentons de stocker les charges utiles des paquets les unes à la suite des autres dans une zone de stockage précédemment allouée dans le dépôt IBP.

```
class Ibp_testS extends Service
{
    static final String IBP_DEPOT_NAME = "ibp01.ens-lyon.fr";
    static final int    IBP_DEPOT_PORT = 6716;
    static final int    CAP_SIZE = 1000 ; // KBytes
    IBPDepot     dpt ;
    CapAttribute attr ;
    IBPCaps      caps ;
    WriteCap wc ;
    ReadCap  rc ;

    Ibp_testS()
    {
        try{
            dpt = new IBPDepot( IBP_DEPOT_NAME ,IBP_DEPOT_PORT );
            attr = new CapAttribute(0,IBPProtocol.IBP_STABLE,
                                    IBPProtocol.IBP_BYTEARRAY);
            caps = new IBPCaps(dpt,CAP_SIZE*1024,attr,0,0);
            wc = caps.getWriteCap();
            rc = caps.getReadCap();
        } catch (UnknownHostException e) {
        } catch (IBPException e) {...}
        } catch (IOException e) {...}
    }
    public void run()
    {
        long ret1 = 0;
        while(!FINISHED) {
            try {
                // 1/ cache payload in the local ibp depot
                ret1 = wc.write( getANEPkt().payload,
                                 getANEPkt().payload.length );
                System.out.println("Ibp_testS: write in " + IBP_DEPOT_NAME +
                                   " depot : " + ret1 + " bytes");
                // 2/ forward cached data immediately
                forward();
            }catch (IOException e) {
                System.err.println("Ibp_testS 1: " + e);
            }catch (IBPException e) {
                System.err.println("Ibp_testS 2: " + e);
            }
            // 3/ wait for the next ANEP packet
            recv();
        } // while(!FINISHED)
    } // run()
    public void process() {}
} // class Ibp_testS
```

Fig. 5.5 – Service Tamanoir de test permettant l'interaction avec un dépôt IBP.

Le service de cache systématique pour Tamanoir constitue la brique de base pour la construction de nombreux services actifs bénéficiant de stockage temporaire dans le réseau :

- Cache de données lors du déploiement d'un service : Le déploiement d'un service dans un TAN (à partir du Service Repository ou de TAN en TAN) est déclenché lors de l'arrivée du premier paquet actif. Pendant le déploiement il convient de stocker temporairement les paquets actifs suivants afin d'éviter leur perte dans l'équipement actif. Dans le cas où le protocole UDP est utilisé les pertes sont irrémédiables. Dans le cas où l'on emploie le protocole TCP, l'émetteur doit réduire sa vitesse ou re-émettre les paquets. Afin de corriger ce comportement, nous exécutons un dépôt IBP sur le nœud local (où le TAN est exécuté) afin de cacher les données (cf. figure 5.6) Ce service appelé *IBPService* utilise un appel à *IBP_store* pour rediriger le début du flux vers le dépôt IBP. Le service IBP vérifie également la présence du service requis chaque fois qu'un nouveau paquet arrive, et si c'est le cas, un appel à *IBP_load* est réalisé pour rediriger toutes les données cachées dans le dépôt IBP vers le service capable de les traiter, les router et les transmettre efficacement. Le service *IBPService* est pré déployé lors de l'installation du noeud actif.
- Communication de groupe fiable : Il y a trois avantages à faire du stockage dans le réseau pour une application de communication de groupe (multicast) fiable. Premièrement, nous pouvons considérer un TAN et son dépôt IBP comme un miroir pour la distribution de données afin de les télécharger vers le client le plus proche d'un point de vue géographique ou réseau. Ensuite, les clients peuvent consommer les données à leur propre vitesse et capacité de traitements sans déranger le serveur d'où les données proviennent. Finalement, un TAN peut retransmettre les données perdues sans surcharger inutilement le réseau situé entre le serveur et le TAN. Les considérations ci-dessus nous ont conduit à mettre au point un service appelé *XcastIbpService* et destiné au applications multicast. La figure 5.6 montre une vue d'une topologie multicast simple avec un serveur de données (ex : *push server, ASP server*) et des clients hétérogènes en terme de vitesse de connections et de capacité de traitement. Les clients ne demande pas les même données au même moment et ne les consomment pas à la même vitesse, nous utilisons un dépôt IBP à côté du TAN pour cacher ces données. Pour cette application, un TAN peut être vu comme un nœud multicast avec des capacités de caches. Le serveur n'envoie qu'une seule fois les données vers le plus proche nœud actif des clients. Ainsi les clients peuvent consommer les données à leur propre vitesse.
- IBP-mail avec Tamanoir : IBP-mail [67] permet à un utilisateur d'émettre un fichier attaché (de toute taille) à un mail sans surcharger le serveur de mail, en utilisant le stockage du réseau disponible à travers un ensemble de dépôts IBP. Le fichier est stocké dans le dépôt IBP pour une durée spécifiée. Le récepteur peut télé-charger le fichier à sa guise en utilisant un outil développé dans ce but. Pour supporter cette application nous exploitons efficacement la capacité de traitement d'un TAN pour gérer le mouvement de données d'un dépôt proche de l'émetteur vers un dépôt IBP proche du récepteur afin de tirer avantage de la proximité (ex : haut débit, faible latence). Ce type d'application implique l'usage de capacités de traitements dans le réseau pour diriger un fichier attaché vers le dépôt IBP le plus proche du récepteur. La notion de proximité réseau reste un problème ouvert [68] et peut être mesurée avec différentes métriques : temps d'aller-retour (round-trip time), latence, débit, espace, nombre de saut (hops).

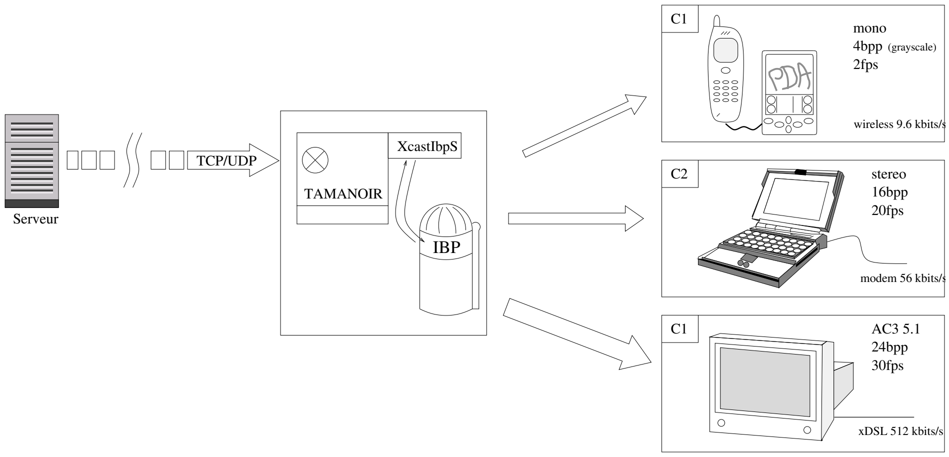


FIG. 5.6 – Communication de groupe (mulitcast) fiable avec Tamanoir et IBP

5.5 Générateur distribué de flux actif

Afin de tester une infrastructure active déployée dans un réseau longue distance nous avons défini et mis en œuvre *Echidna* : un outil de génération de flux de données actives distribué. Cet outil, développé en collaboration avec Saad El Hadri, est basé sur des mécanismes d'invocation de méthodes à distance RMI (*Remote Method Invocation*) Java. Il permet de lancer des tests suivant des scénarios préétablis dans des conditions réalistes. Un scénario de test est décrit dans un fichier de configuration. Ce fichier est exploité par un gestionnaire qui génère l'invocation des méthodes sur les différents clients prêts à s'exécuter. Un client peut devenir soit un émetteur de flux, soit un récepteur de flux. Le fichier scénario décrit des paires de communications entre postes clients. En fonction des paramètres, un client peut émettre soit un flux de données aléatoire, soit un flux constitué de paquets identiques, soit le contenu d'un fichier local (disponible sur le poste client). Le flux est par défaut encapsulé dans des paquets ANEP ou sous la forme de flux TCP ou UDP. La figure 5.7 illustre un scénario d'expérimentation. Un gestionnaire (en haut) lance ses requêtes sur des postes clients. Ceux-ci génèrent ou réceptionnent alors des flux. Afin d'exposer des informations sur l'état des nœuds on y déploiera des services actifs de monitoring (la remontée d'informations des nœuds n'est pas représentée).

L'intérêt d'un tel outil est qu'il permet de réaliser des expérimentations reproductibles sur un réseau longue distance et de vérifier la stabilité des résultats en reproduisant le test à différentes heures de la journée par exemple.

5.6 Environnement de gestion de nœuds Tamanoir déployés dans une grille de calcul

Les travaux que nous avons menés dans le cadre du projet RNTL e-Toile concernent le support efficace d'intergiciels de Grille à l'aide de réseaux actifs [69, 70, 71, 72]. Pour cette raison nous avons mis au point un outil permettant de gérer des nœuds Tamanoir et des services actifs déployés dans une grille de calcul longue distance. Pour cela nous avons adapté et élaboré les modules d'une interface Web appelée *MapCenter*[73], fournie par le projet

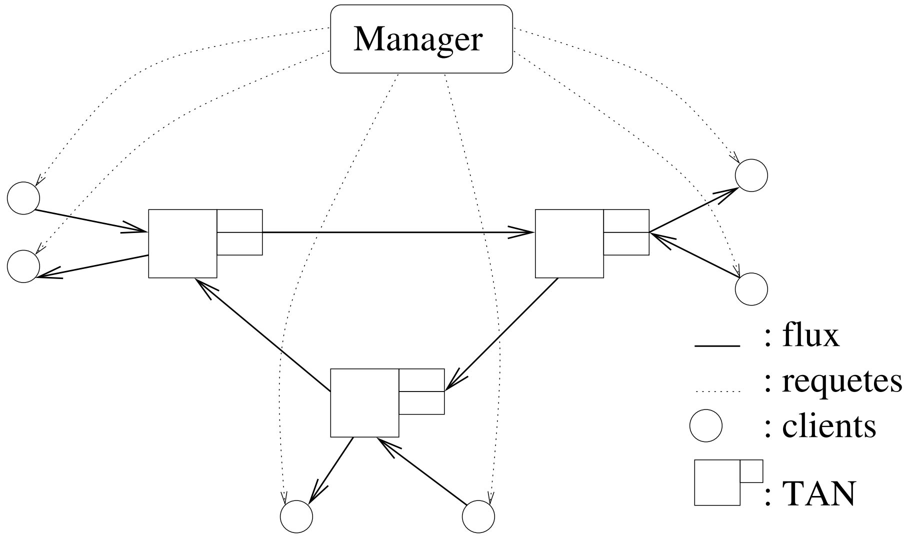


FIG. 5.7 – Générateur de flux distribué pour la validation d'une infrastructure active Tamanoir à 3 nœuds.

européen DataGrid et conçue pour interroger l'état d'une plate-forme de Grid computing. Cette adaptation nous permet aujourd'hui d'interroger l'état de l'infrastructure active, d'un nœud Tamanoir et d'une instance de service exécutée sur un TAN au travers d'un navigateur web. La figure 5.8 présente les équipements actifs Tamanoir que nous avons déployés en France sur plusieurs sites de recherche autour du réseau VTHD. Ces site ont permis de valider le déploiement de Tamanoir et de mener une série de tests sur une plate-forme longue distance maîtrisée à haut débit (cf. section 6.5.5).

La figure 5.9 suivante propose une vue arborescente des nœuds Tamanoir déployés. On notera la présence du Service Repository à l'ENS de Lyon. Il suffit de cliquer sur l'un des TAN pour connaître son état et la liste des services déployés.

La figure 5.10 est obtenue après avoir sélectionné l'une des instances d'un service, en l'occurrence celle d'un service de compression à la volée. Les informations exposées par ce service concernent la quantité de données entrées pour être compressées et la quantité de données compressées sorties.

5.7 Conclusion

Nous avons présenté dans ce chapitre quelques exemples illustratifs de la bibliothèque de services actifs fournis dans le cadre du projet Tamanoir. Ces services sont chargés dynamiquement dans un EE exécuté dans l'espace utilisateur. A l'aide de ces exemples, nous avons montré que l'écriture d'un nouveau service ne nécessite pas de compétence de développement particulière. Nous avons également présenté un service original de stockage dans le réseau. Celui-ci nous a permis d'illustrer la grande facilité avec laquelle on peut concevoir une interface à une application externe (ici IBP) sous la forme d'un service actif. Il permet de démontrer la flexibilité de Tamanoir et la facilité avec laquelle on peut ajouter de nouvelles fonctionnalités. Nous avons également introduit dans ce chapitre deux outils qui sont les briques de base

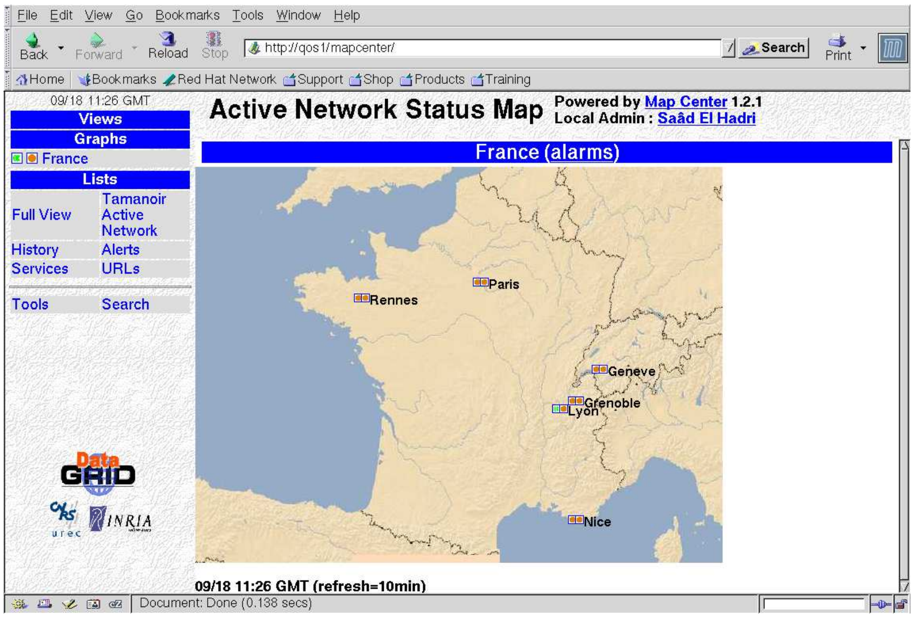


FIG. 5.8 – Interface Web offrant une vue globale du déploiement national de nœuds actifs Tamanoir autour de VTHD.

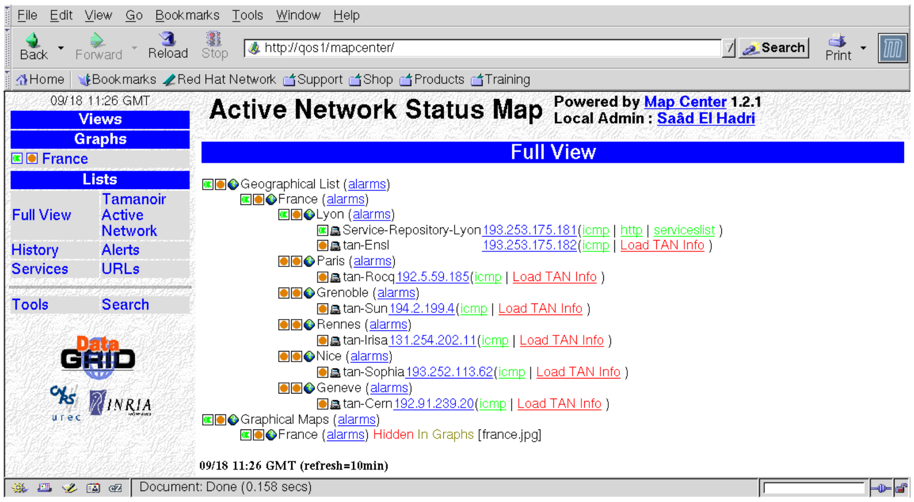


FIG. 5.9 – Liste des nœuds Tamanoir déployés.

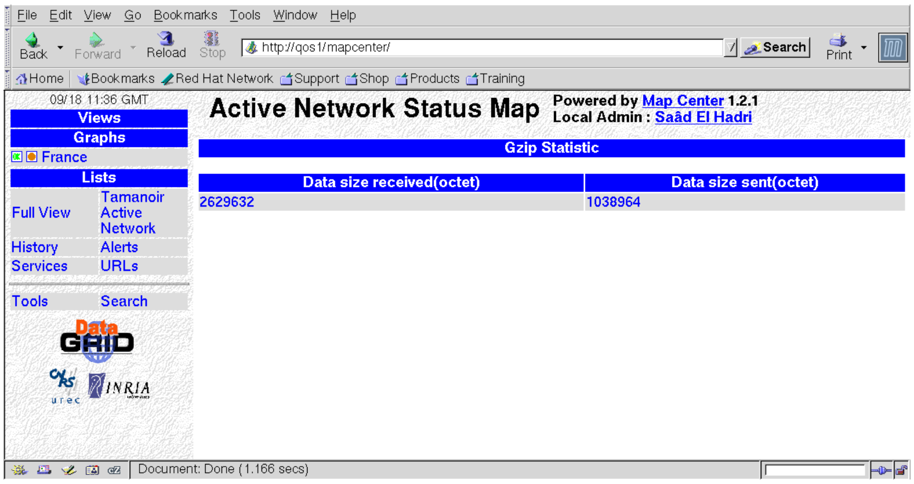


FIG. 5.10 – Informations renvoyées par une instance du service de compression (GzipS)

d'un environnement de développements de services actifs. Le premier permet aux utilisateurs de tester artificiellement leur services déployés dans infrastructure active grandeur nature. Le second permet aux utilisateurs de contrôler, à travers une interface Web intuitive, l'état du réseau actif ainsi que le bon fonctionnement des services déployés en consultant les informations exposées par le service.

Chapitre 6

Expérimentations

6.1 Introduction

Dans ce chapitre nous commençons par décrire les différentes plate-formes de tests utilisées tout au long des expérimentations de Tamanoir. Nous réalisons une comparaison qualitative avec l'EE ANTS. Ensuite nous présentons les performances en latence et débit brut obtenues avec un nœud Tamanoir mono machine. Nous validons ensuite l'apport d'une architecture distribuée sur deux types de services. Enfin nous terminons en présentant des mesures réalisées avec une plate-forme Tamanoir longue distance.

6.2 Description des plate-formes matérielles et logicielles de tests de Tamanoir

De nombreuses expérimentations ont été menées avec différents prototypes de Tamanoir au cours de cette thèse. La suite logicielle Tamanoir ainsi que le matériel ont évolué.

6.2.1 Plate-formes matérielles

Différentes plate-formes matérielles ont été utilisées :

- **Plate-forme 1** : Cette première plate-forme de test, financée par l'ANVAR, est constituée de trois PC bi-processeur Pentium III cadencés à 1GHz qui sont utilisés comme nœuds actifs Tamanoir. Les applications clientes sont déployées sur des PC AMD Athlon 1GHz. Cette plate-forme est interconnectée par un réseau FastEthernet (100Mbps, driver `tulip.o` et EtherExpress Pro (`eepro.o`)). Les bus PCI 32bits sont cadencés à 33MHz. Les nœuds actifs sont pourvus de plusieurs cartes d'interfaces réseau afin qu'ils soient utilisés comme routeur (nous n'utilisons pas de commutateurs).
- **Plate-forme 2** : Cette seconde plate-forme de test, financée dans le cadre du projet RNRT VTHD, est constituée de sept Compaq DL360 (G2) Proliant, bi-PIII à 1.4GHz avec un bus PCI à 66MHz. Ils sont interconnectés par un réseau Gigabit Ethernet cuivre (connecteur RJ45) sur un commutateur Gigabit *Foundry*.
- **Plate-forme 3** : Cette troisième plateforme, financée dans le cadre du projet RNTL e-Toile, est une grappe (*cluster*) constituée de 16 SUN LX50, bi-PIII à 1.4GHz avec également un bus PCI à 66MHz. Les nœuds de calculs sont interconnectés par deux réseaux : FastEthernet et Myrinet. Ce dernier est un réseau de type SAN (*System Area*

Network) très haut débit, très faible latence (Gigabit). Chaque machine est reliée par fibre optique à un commutateur Myrinet.
- **Plate-forme 4** : Nous avons disposé d'une connexion directe à la plate-forme VTHD (Vraiment Très Haut Débit). C'est un réseau expérimental haut débit dont les liens internes sont constitués par des longueurs d'onde sur fibre optique exploitées à un débit de 2,5 Gbps ; les nœuds de routage sont réalisés à base de routeurs de type Giga routeurs (i.e. quelques dizaines de Gbps de capacité de commutation). Ce réseau relie de nombreux sites de recherche français comme l'INRIA Rocquencourt et Sophia, l'ENS de Lyon, les SUN labs/Europe à Grenoble, l'ENST, Eurecom ainsi que les différents centres de recherche de France Telecom (FT R&D). Les machines utilisées dans cette plate-forme dépendent des sites interconnectés pendant les expérimentations.

Afin de mener une évaluation la plus complète possible nous avons effectué des mesures à l'aide de différentes technologies réseaux mises à notre disposition :
- un réseau local classique FastEthernet à 100 Mbps,
- un réseau local Giga Ethernet à 1000 Mbps (1 Gbits),
- un réseau SAN Myrinet pouvant atteindre un débit de plus de 1.7 Gbps pour un flux transporté sur TCP/IP,
- le réseau expérimental VTHD, longue distance, grande latence.

6.2.2 Plate-formes logicielles

Les premières expériences ont été réalisées sur différentes machines virtuelles Java. Celle de SUN[74], d'IBM[75], la version libre Kaffe[76] et celle proposé par l'organisation Blackdown[77]. Nous avons également mené des expérimentations sur une version Tamanoir écrite en Java mais compilée pour être exécutée en code natif à l'aide du frontal à GCC appelé GCJ (GNU Compiler for Java)[78]. Les mesures les plus récentes ont été réalisées à l'aide de la JVM J2RE 1.4.2 fournie par SUN et incluant la technologie Hotspot [79]. Certaines mesures ont été réalisées avec un prototype de Tamanoir (version 0.1) ne bénéficiant pas de toutes les optimisations disponibles dans la version actuelle (Tamanoir version 1.0). Toutes les mesures présentées dans ce chapitre ont été réalisées sur des plate-formes animées par le système d'exploitation GNU/Linux (distribution *Debian*). Les versions de noyau ont évolué jusqu'à la version 2.4.19 aujourd'hui. Cette version offre la possibilité de déployer des services dans le noyau avec *Netfilter*. Des expérimentations anecdotiques ont eu lieu sur plate- forme Windows mais n'ont apporté aucun gain. Elles ont uniquement validé la portabilité de l'EE Tamanoir. Nous avons focalisé nos expérimentations sur deux types de services (léger et lourd) caractéristiques des fonctionnalités dynamiques déployables sur les noeuds Tamanoir :

- Le **service léger** (*lightweight service*) consiste à compter les paquets pour chacun des flux. Il ne consomme que très peu de mémoire puisqu'il ne possède qu'une variable d'état, le compteur.
- Le **service lourd** (*heavy service*) se propose de compresser la charge utile des paquets actifs. C'est un service très exigeant en ressource CPU. Sa consommation mémoire se limite à allouer une nouvelle zone mémoire de la taille de la charge utile pour chaque nouveau paquet. Une fois celle-ci compressée la zone mémoire allouée est libérée.

6.3 Comparaisons qualitatives de Tamanoir et ANTS

ANTS est un EE pour réseau actif expérimental destiné à faciliter le développement et la mise au point de nouveaux services actifs. Il est l'un des premiers EE ouverts et disponibles utilisés par différents projets. Dans une plate-forme d'expérimentation ANTS tous les nœuds, qu'ils soient consommateurs ou producteurs de données ou qu'ils soient nœuds actifs, devront faire partie de l'infrastructure active. Une table de routage statique est construite avant le lancement de tous les nœuds. C'est un *overlay* du réseau qui décrit la topologie de tous les nœuds impliqué dans l'expérimentation. Cela signifie que dans la pratique il est impossible de rajouter un client (par exemple) au cours de l'exécution. Il est nécessaire de stopper tous les nœuds, mettre à jour leur table et enfin les relancer. Tamanoir propose deux autres approches :

- La première, dite par *morceau* ou en mode proxy : seuls les nœuds actifs possèdent une table de routage permettant d'accéder aux autres nœuds Tamanoir. Les clients ne sont pas décrits dans cette table. Cette table peut être dynamiquement mise à jour (lors de l'ajout d'un nouveau TAN). Afin que les applications clientes puissent accéder à l'infrastructure active elles doivent adresser leur flux à un point d'entrée de cette infrastructure. On appelle ce point d'entrée un *proxyTAN*. Il se singularise uniquement par sa position dans l'infrastructure.
- La seconde approche, dite de *bout en bout*, est moins facile à déployer dans la pratique. Elle utilise les mécanismes de routage du réseau classique. Les paquets sont marqués (option IP Router Alert) pour pouvoir être capturé et modifié par un service Tamanoir. Un producteur de flux actif adresse donc dans cette approche son flux directement au récepteur. Si le flux traverse un équipement actif il se verra alors traité.

ANTS ne supporte que le protocole `UDP`. Aujourd'hui à la vue des applications (de *Grid computing*[69, 70] par exemple) susceptibles d'exploiter les réseaux actifs, il semble nécessaire de fournir le support d'un protocole fiable. Tamanoir supporte les protocoles `UDP` et `TCP`. Bien que le déploiement de nœuds ANTS ne soit pas aussi complexe que cela puisse paraître, la mise en place d'une plate-forme Tamanoir locale ou longue distance est extrêmement aisée. Pour la version dite « par morceau » il n'y a qu'une table de routage décrivant comment sont interconnectés les TAN à mettre à jour. Si l'expérimentation ne nécessite que deux TAN (un sur chaque site distant) la configuration de la table de routage devient facultative. Les applications de tests fournies sont bien documentées. Tamanoir ne nécessite pas d'environnement graphique. Le support de différents flux se fait dynamiquement dans Tamanoir. Il n'y a pas à ce jour de scripts nécessaires pour exécuter et paramétrer un nœud actif. Pour des raisons de sécurité Tamanoir interdit le télé-chargement d'un service à partir d'un client. Le proxyTan doit s'adresser à un serveur de services, les autres TAN ont l'alternative de télé-charger le service à partir d'un *service repository* ou, à partir du dernier TAN traversé, comme dans ANTS, on parle de déploiement de proche en proche. Dans Tamanoir, même si le service provient du dernier TAN traversé, celui-ci est télé-chargé dans un autre canal fiable (*out-of-band*). ANTS a une approche dite encapsulée. Le service est télé-chargé dans le même canal que le flux de données (*in-band*). Lors du démarrage de cette thèse nous avions mené sur ANTS quelques mesures de performances en débits. Les résultats obtenus en débit crête étaient d'à peine 6 Mbps alors qu'une version préliminaire de Tamanoir nous permettait déjà d'atteindre un débit moyen de 34 Mbps. Ce manque de performance dans les environnements disponibles comme ANTS nous ont donc conduit à développer et améliorer notre propre environnement d'exécution actif Tamanoir.

6.4 Mesures de latence

Pour mesurer l'efficacité de notre EE nous déployons un service qui applique un traitement extrêmement léger, en terme de consommation CPU et mémoire, sur la charge utile du paquet actif. On appelle **latence d'un nœud actif** le temps total mis par un paquet actif pour le traverser.

Afin de réaliser des mesures précises, nous plaçons le chronomètre au plus près de l'interface réseau. Le chronomètre est enclenché à l'arrivée d'un paquet ANEP et stoppé lorsque ce même paquet quitte le routeur actif. La précision des résultats est de l'ordre de la microseconde. Pour cela, nous avons employé l'outil *Netfilter* fourni par le noyau Linux. Cet outil nous a permis d'installer un filtre qui démarre ou stoppe des chronomètres lors du passage de paquets ANEP. Ce filtre est en fait un module Netfilter qui s'exécute au niveau noyau.

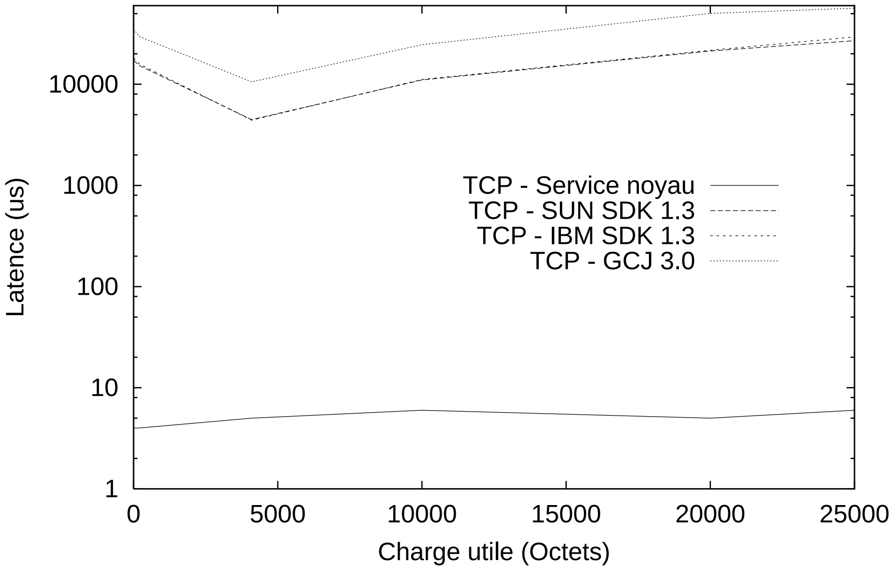


FIG. 6.1 – Temps de traversée d'un nœud actif Tamanoir par un paquet ANEP sur `TCP`.

Les mesures de latence présentées dans les figures 6.1 et 6.2 ont été réalisées sur la plate-forme 1 avec deux JVM (IBM et SUN 1.3) ayant le compilateur *Just-In-Time* activé ainsi que sur une version compilée d'un TAN avec GCJ. En abscisse, nous faisons varier la taille de la charge utile jusqu'à 25ko ; en ordonnée, les temps de traversée (latence) sont données en microsecondes. Le service appliqué est extrêmement léger puisqu'il se contente d'estampiller et de compter les paquets ANEP. La courbe *TCP-Service noyau* représente la latence induite par un routage noyau classique. Ici un service actif noyau est appliqué. Les paquets sont simplement comptabilisés et transmis sur l'interface de sortie le plus rapidement possible. Les valeurs obtenues oscillent autour de 5 à 7 μs et semblent indépendantes de la taille du paquet. Quelle que soit la JVM employée les résultats obtenus restent similaires. Seule la version compilée par GCJ se démarque nettement en offrant des résultats plus faibles. Ces faibles résultats obtenus avec GCJ pour une exécution en mode natif (compilé) s'expliquent par le manque de maturité du compilateur qui produit un code natif non optimal.

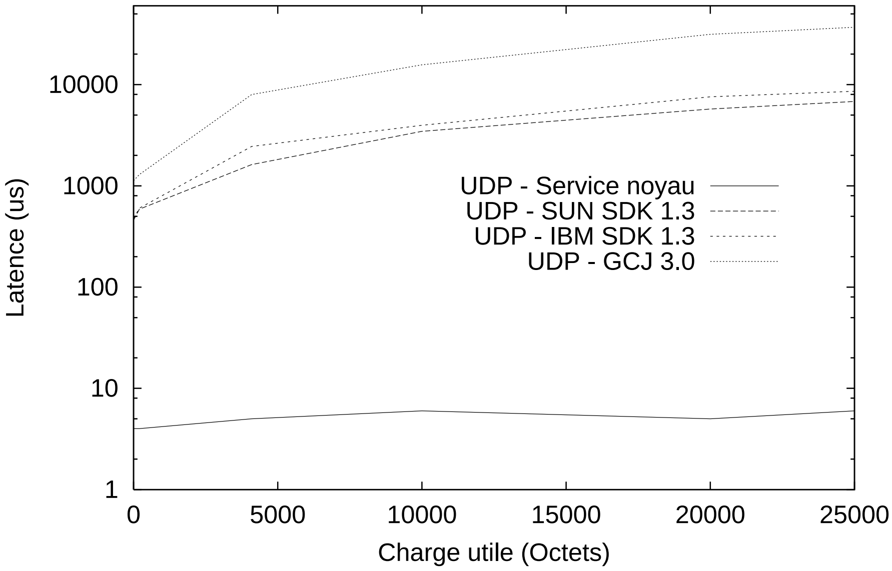


Fig. 6.2 – Temps de traversée d'un nœud actif Tamanoir par un paquet ANEP sur `UDP`.

Pour un flux `UDP` la latence obtenue avec les deux JVM reste inférieure à 10 ms et atteint de bons résultats pour des paquets de petite taille : 7ms pour des paquets de 128 octets. Pour un flux `TCP` on constate des performances plus faibles pour des paquets de petite taille. Cela s'explique par le fait que l'implémentation de la couche `TCP` du noyau Linux tente d'agréger les paquets de petite taille avant de les émettre. Dans ce cas, la première partie de la courbe reflète les temps d'expiration avant émission. Ensuite la courbe augmente de manière régulière.

Ainsi, bien que la latence de traversée d'un paquet `TCP` soit sensiblement supérieure à celle d'un paquet `UDP` on constate que cela reste dans le même ordre de grandeur. L'évolution des courbes reste également similaire en terme de linéarité et de pente. Par contre on constate un facteur d'environ 1000 entre un routage noyau simple et un routage dans l'espace utilisateur avec l'application d'un service même très simple. Ces expérimentations ont été menées avec la version 0.1 de Tamanoir qui n'inclut pas toutes les optimisations. Néanmoins elles illustrent bien le fait que la remontée et le traitement d'un paquet dans l'espace utilisateur ont un impact sur la latence.

6.5 Mesures de débits

Nous avons mené des nombreuses campagnes d'expérimentations afin de mesurer le débit d'un noeud actif Tamanoir dans différentes conditions (matérielles et logicielles). Les résultats sont donnés en Million (Méga) de bits par seconde (Mbps). Ils représentent la vitesse moyenne de traitement des données dans un équipement Tamanoir déployant un service actif. Jusqu'à la section 6.5.3 nous utilisons pour les mesures un nœud actif Tamanoir mono machine dont l'EE est exécuté dans l'espace utilisateur. Pour ces mesures nous utilisons plusieurs machines clientes pour émettre ou recevoir des flux actifs. Nous limitons l'émetteur à un maximum de deux ou trois flux car au delà le débit brut par flux s'écroule rapidement. Cela s'explique par les contentions qui interviennent lorsque des processus émettent en parallèle des flux à travers un bus système et une carte d'interface réseau unique. Toutes les mesures présentées dans cette section ont été réalisées avec la version 1.0 de Tamanoir.

6.5.1 Mesures de débits sur réseau FastEthernet

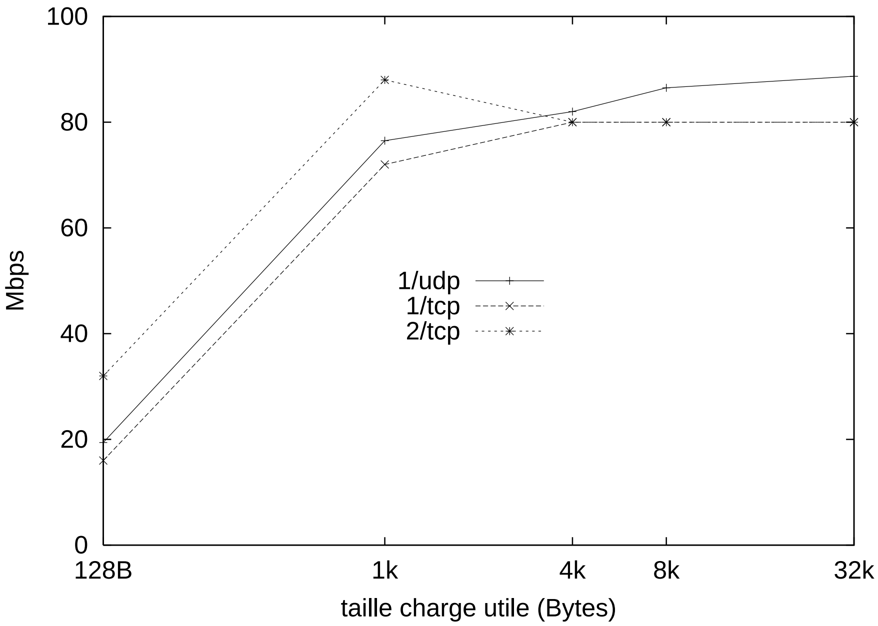


Fig. 6.3 – Débits obtenus sur réseau FastEthernet avec un service léger.

Les mesures de débits présentées sur la figure 6.3 ont été obtenues sur le réseau FastEthernet de la plate-forme 3 (grappe SUN). Nous avons déployé un service léger, en faisant varier le nombre de flux. Sur TCP, dès que la charge utile des paquets atteint 1 kB on obtient un débit de plus de 70 Mbps pour un flux unique, près de 45 Mbps par flux lorsque deux flux traversent Tamanoir. Pour UDP le fonctionnement est plutôt bon. Il est à noter que la mise en œuvre actuelle de Tamanoir ne supporte pas l'architecture SMP pour des services appliqués à des flux UDP. On retiendra donc de ces mesures que nous atteignons rapidement le débit maximum proposé par un réseau FastEthernet avec un service léger et un tout petit nombre de flux sur TCP.

Les débits présentés figure 6.4 ont également été obtenus avec le réseau FastEthernet de la plate-forme 3. Nous appliquons un service lourd à un ou deux flux de données transportés sur TCP. On constate que ce service est plus efficace sur des paquets dont la charge utile est d'au moins 8kB. Pour deux flux, nous tirons clairement bénéfice de l'architecture SMP (bi-processeur) du nœud Tamanoir. Cela laisse présager de l'intérêt de distribuer la charge de calcul pour des réseaux plus rapides. Pour deux flux nous atteignons déjà quasiment la bande passante maximum offerte par la technologie FastEthernet.

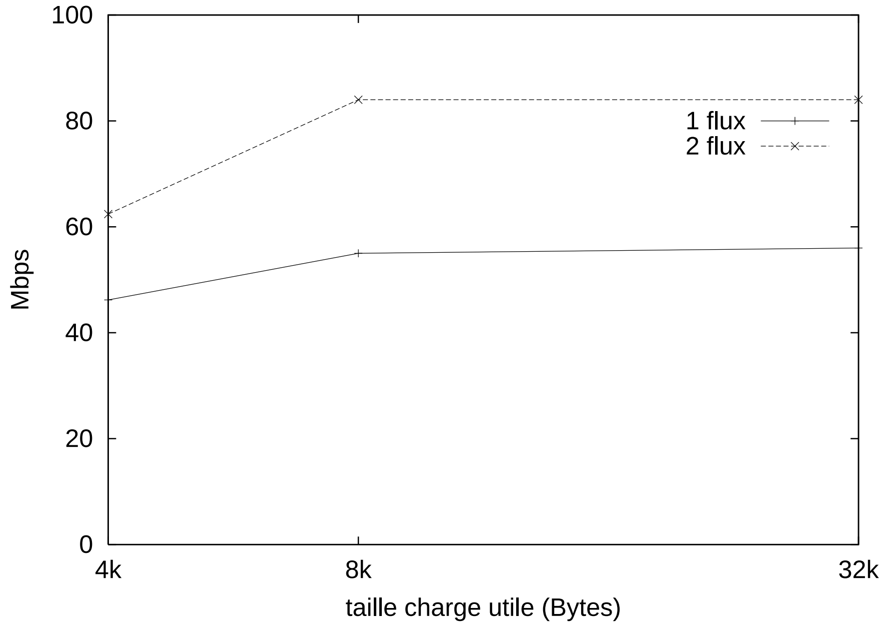


FIG. 6.4 – Débits obtenus sur réseau FastEthernet avec un service lourd.

6.5.2 Mesures de débits sur réseau GigaEthernet

Les mesures de débits présentées dans les figures 6.5, 6.6 et 6.7 sont obtenues sur des machines similaires à celles utilisées précédemment mais cette fois-ci interconnectées par un réseau local GigaEthernet (1000Mbps) sur un commutateur haut de gamme (plate-forme 2).

La figure 6.5 présente les débits obtenus sur TCP avec un service léger pour un ou deux flux. Pour exploiter au mieux la bande passante il semble préférable d'utiliser des paquets dont la charge utile est de 8 à 32 kB. Cela est probablement du à Java qui semble avouer quelque faiblesse à gérer des paquets de cette taille à un tel débit. Finalement on constate que nous ne saturons pas le lien même avec deux flux.

La figure 6.6 montre les débits obtenus sur UDP avec un service léger pour un ou deux flux. Les résultats obtenus sont inférieurs à ceux atteints en TCP. Comme pour les mesures en FastEthernet les paquets de très grande taille sont mal supportés.

Les mesures suivantes dont les résultats sont présentés sur la figure 6.7 ont été obtenus avec un service lourd appliqué à un ou deux flux sur TCP. Ici les paquets de très grandes tailles sont bien adaptés. A partir de 8kB pour deux flux on double presque le débit agrégé. On tire donc clairement avantage dans cette expérience de l'architecture bi-processeur (SMP) du nœud Tamanoir. On constate que pour un flux nous atteignons près de 90 Mbps qui est la limite d'un réseau FastEthernet, or sur la figure 6.4 le débit maximum d'un unique flux n'était que de 55 Mbps bien que le service appliqué soit identique et consomme la même quantité de ressources. Cette différence permet donc d'illustrer le temps supplémentaire nécessaire pour accéder à une interface. Cela est du à la qualité des interfaces réseaux et leur pilote associé.

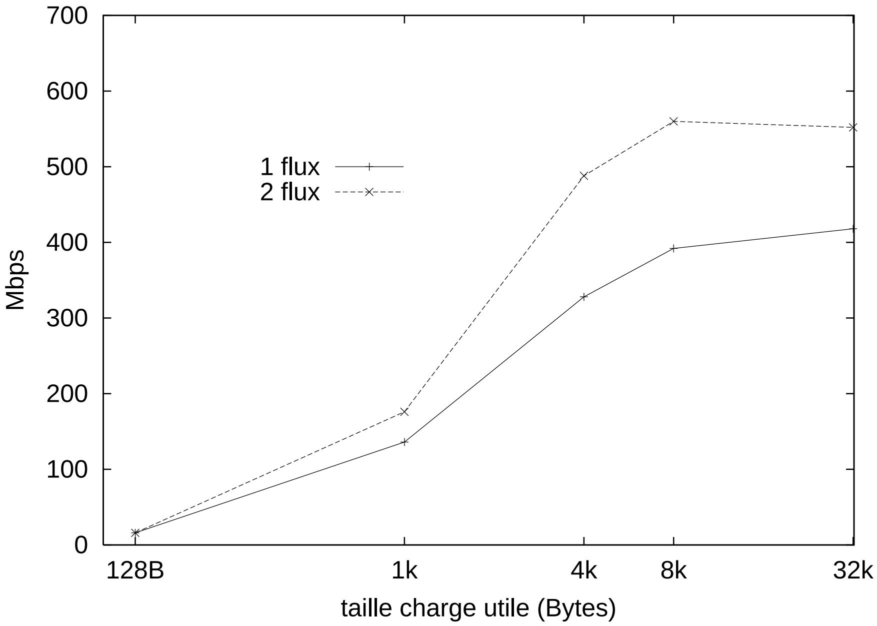


Fig. 6.5 – Débits obtenus sur réseau Giga Ethernet en TCP avec un service léger.

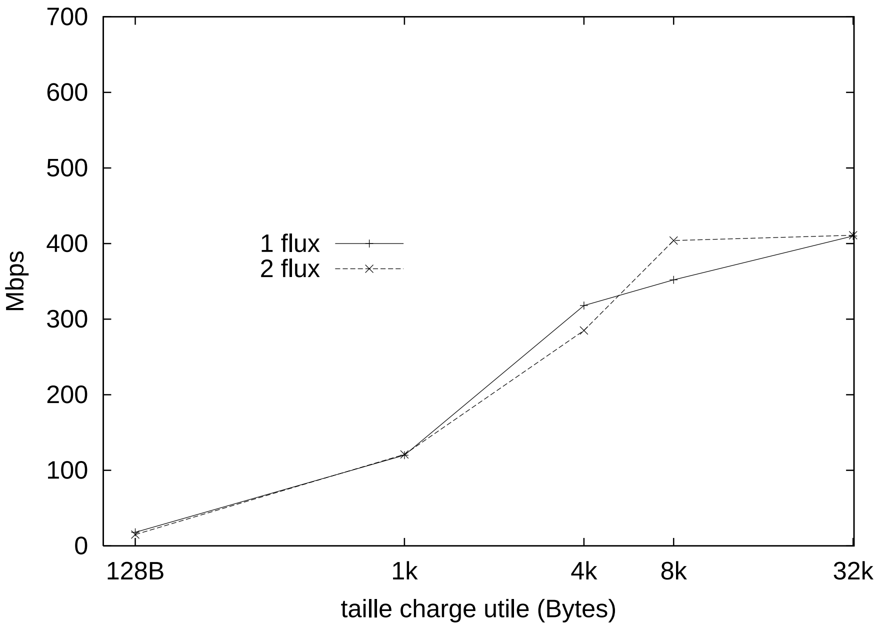


Fig. 6.6 – Débits obtenus sur réseau Giga Ethernet en UDP avec un service léger.

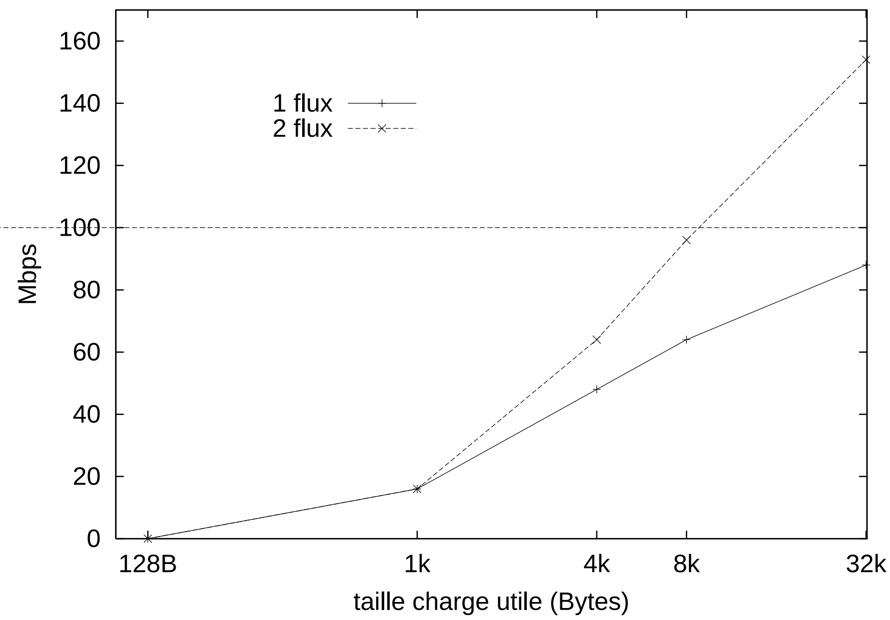


FIG. 6.7 – Débits obtenus sur réseau Giga Ethernet en TCP avec un service lourd.

6.5.3 Mesures de débits sur réseau Myrinet

Nous avons effectué une série de mesures des débits obtenus sur un réseau Myrinet très haut débit faible latence avec un nœud actif Tamanoir mono machine qui applique un service léger à un flux unique sur TCP ou UDP. Pour cela nous utilisons la plate-forme 3 (grappe SUN/Myrinet). Nous dédions une machine comme noeud actif Tamanoir. Les autres sont utilisés pour émettre et/ou recevoir des flux actifs. Nous faisons varier la taille de la charge utile des paquets ANEP entre 128 octets et 32 kB (en abscisse, échelle logarithmique). Le service appliqué à chaque flux est un service léger. Les résultats de ces mesures sont présentés figure 6.8. On constate que le débit obtenu avec TCP est légèrement supérieure au débit obtenu avec UDP. Les débits obtenus pour de gros paquets (>32kB) s'écroulent littéralement probablement pour les même raisons décrites dans la section précédente (6.5.2). Ces courbes peuvent être comparées à celles obtenues sur un réseau GigaEthernet, illustrées figures 6.5 et 6.6 afin de mettre en évidence l'impact de la technologie réseau employé. Myrinet atteint 600Mbps alors que la technologie GigaEthernet supporte seulement autour de 500Mbps. Soit une différence d'environ 100Mbps pour des paquets entre 8kB et 32kB.

6.5.4 Mesure de débits sur Tamanoir cluster

Dans cette section en plus d'évaluer Tamanoir sur un réseau local très haute performance, nous évaluons le bénéfice d'utiliser une architecture distribuée dans un nœud actif Tamanoir, on parle de *Tamanoir cluster*. La figure 6.9 illustre de manière extrêmement simplifiée la topologie logique que nous avons mise en œuvre pour les mesures de débits. Un ensemble de clients émet un flux de paquets ANEP vers un client opposé de manière simultanée afin de charger le plus possible le nœud Tamanoir constitué de plusieurs back-ends (BE) et d'un frontal chargé de distribuer les flux.

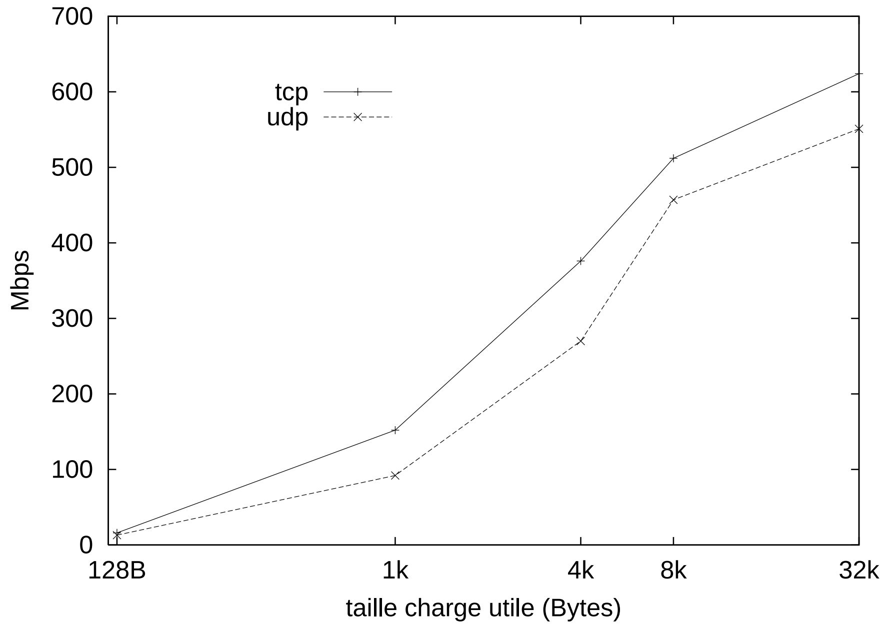


FIG. 6.8 – Débit obtenu sur réseau Myrinet pour un flux traité par un service léger sur UDP ou TCP.

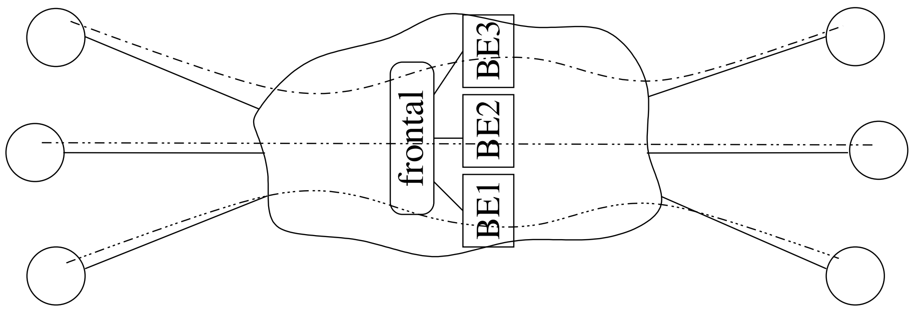


FIG. 6.9 – Vue logique simplifiée pour les premières expérimentations avec Tamanoir cluster.

Nous ne présentons ici que les résultats obtenus avec l'approche DR (*Direct Routing*). Bien que l'approche TUN (*Tunneling*) proposée par LVS offre sensiblement les même performances celle-ci est plus difficile à mettre en place. Enfin les mesures réalisées avec l'approche NAT ont été décevantes en reportant le goulot d'étranglement sur la machine directeur.

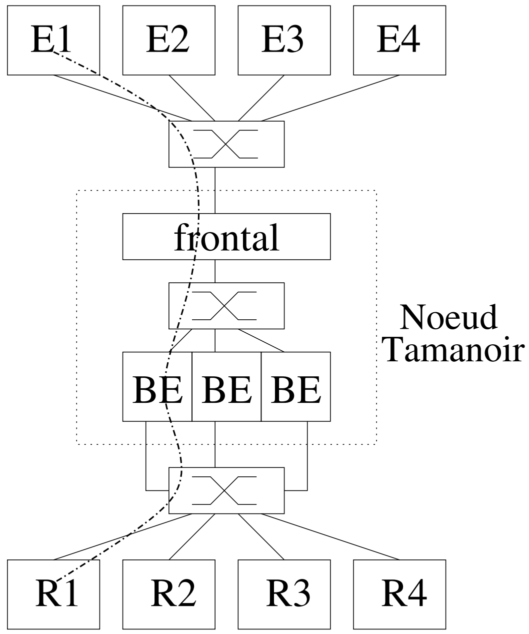


FIG. 6.10 – Vue logique de la plate-forme d'expérimentation Tamanoir cluster sur Myrinet.

La figure 6.10 suivante décrit une vue logique de la plate-forme d'expérimentation locale. Elle met en évidence le parcours des paquets ANEP émis par un émetteur E vers un récepteur R. On constate que les paquets traversent trois commutateurs. C'est le frontal qui est en charge de sélectionner un BE.

Enfin, la figure 6.11 est une vue physique de la plate-forme d'expérimentation n° 3. Tous les nœuds (frontal, *Back Ends* ou clients), sont connectés à un commutateur unique Myrinet. Le flux de paquets passe donc trois fois par ce commutateur. Le frontal LVS se charge de dérouter les paquets d'un même flux sur l'un des BE. Chaque BE exécute une instance de Tamanoir.

Les figures 6.12 et 6.13 présentent les résultats d'un service léger appliqué par un nœud Tamanoir cluster à trois Back Ends (BE) sur un à douze flux TCP simultanés. Les meilleurs résultats sont obtenus pour 3 flux avec des paquets de 8 à 32 kB. Sur la figure 6.12 on constate que pour des paquets de 32kB le débit pour quatre flux devient tout à coup inférieur à trois flux. Nous expliquons cela par le fait qu'à un tel débit il se produit des phénomènes difficilement maîtrisables notamment à travers une JVM. Il est probable qu'il y ait plus de contentions pour quatre flux plutôt que pour trois et que des paquets d'environ 32kB placent l'environnement dans une situation particulièrement défavorable. Dans la figure 6.13, pour des paquets de 8 à 32kB plus le nombre de flux est important plus le débit agrégé réduit. Contrairement à ce que l'on pourrait supposer cette expérience ne met pas en évidence la limite du nœud Tamanoir mais celle des clients émetteurs. Bien que les émetteurs soient des machines performantes, bi-processeurs, le nombre de flux théorique optimal qu'elle devraient être à même d'émettre

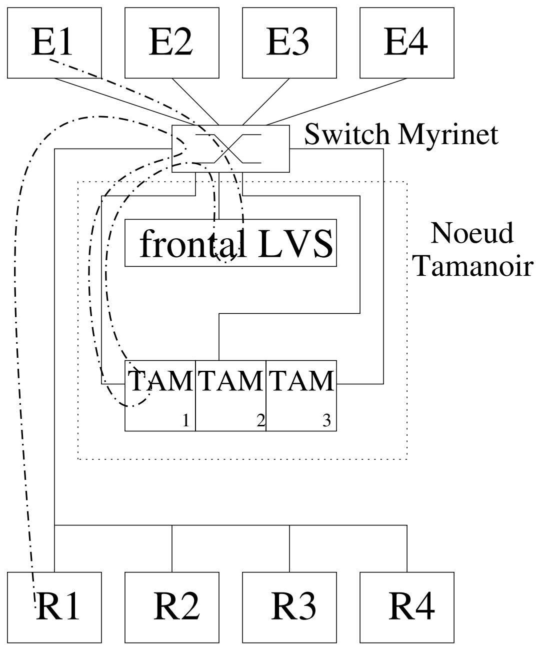


FIG. 6.11 – Vue physique de la plate-forme d'expérimentation Tamanoir cluster sur Myrinet.

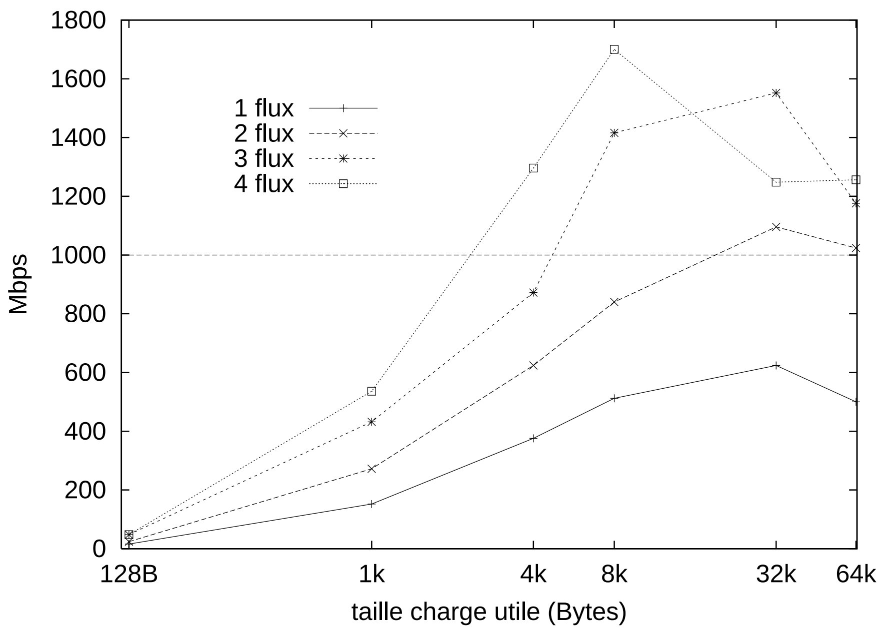


FIG. 6.12 – Débits sur réseau Myrinet avec un Tamanoir cluster à 3 BE appliquant un service léger.

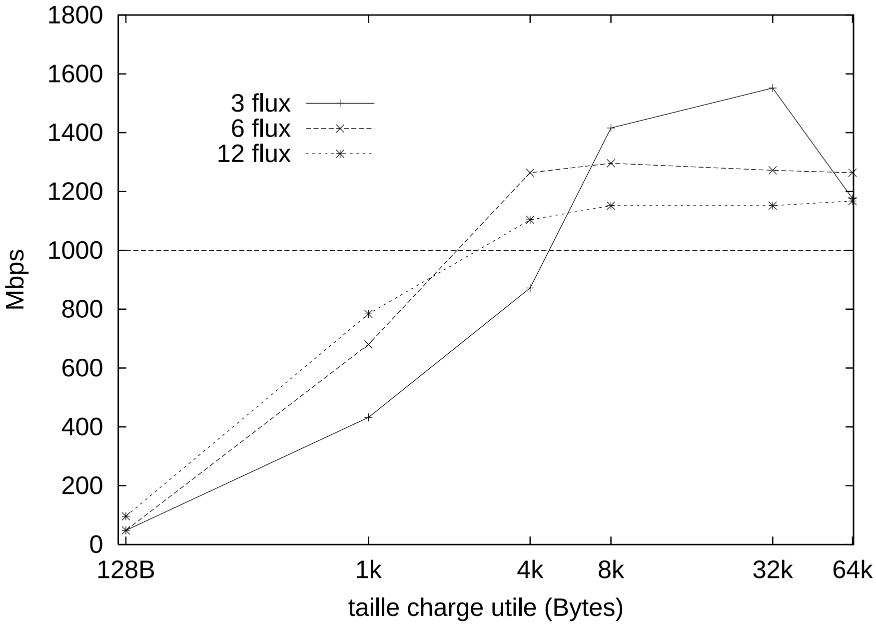


FIG. 6.13 – Débits sur réseau Myrinet avec un Tamanoir cluster à 3 BE appliquant un service léger.

est de deux, soit un flux par processeur. Nous avons généralement employé trois machines pour émettre et trois machines pour recevoir. Donc le nombre de flux optimal est de six, soit deux flux par émetteur. Mais les émetteurs sont équipés d'une carte d'interface réseau unique. C'est donc dans celle-ci que des contentions peuvent apparaître et ainsi grever les performances globales. Il est donc nécessaire d'avoir autant de postes émetteurs que de flux. Le nombre de postes récepteurs est moins critique, car la réception de données est beaucoup moins coûteuse en cycle CPU. Il est donc possible de lancer plusieurs clients récepteurs sur une même machine en lui associant plusieurs IP virtuelles.

La figure 6.14 présente les résultats d'un service lourd appliqué par un nœud Tamanoir à 3 Back Ends (BE) sur un à douze flux TCP simultanés. Les débits agrégés bénéficient de la taille des paquets. Pour 6 ou 12 flux, les débits affichés par les courbes sont sensiblement identiques. Cela montre que pour 6 flux nous avons atteint la capacité de traitement maximum pour cette configuration de nœud actif (à 3 BE) et ce service. Soit un débit maxi de 270Mbps. On tire partie de l'architecture bi-processeur car de 3 à 6 flux nous doublons littéralement le débit.

6.5.5 Mesures de débits sur réseau Gigabit longue distance (VTHD)

La figure 6.15 illustre les trois sites mis à contribution pour effectuer les mesures qui suivent. Ces sites sont :

- SUN labs Europe localisé à Grenoble et relié à VTHD par un routeur gigabit CISCO 2912G. Ce site met à notre disposition neuf racks *cobalt* (SUN LX50) interconnectés par un réseau FastEthernet sur un commutateur Cisco avec un lien GigaEthernet ;
- INRIA de Rocquencourt qui met à notre disposition trente PC interconnectés par un

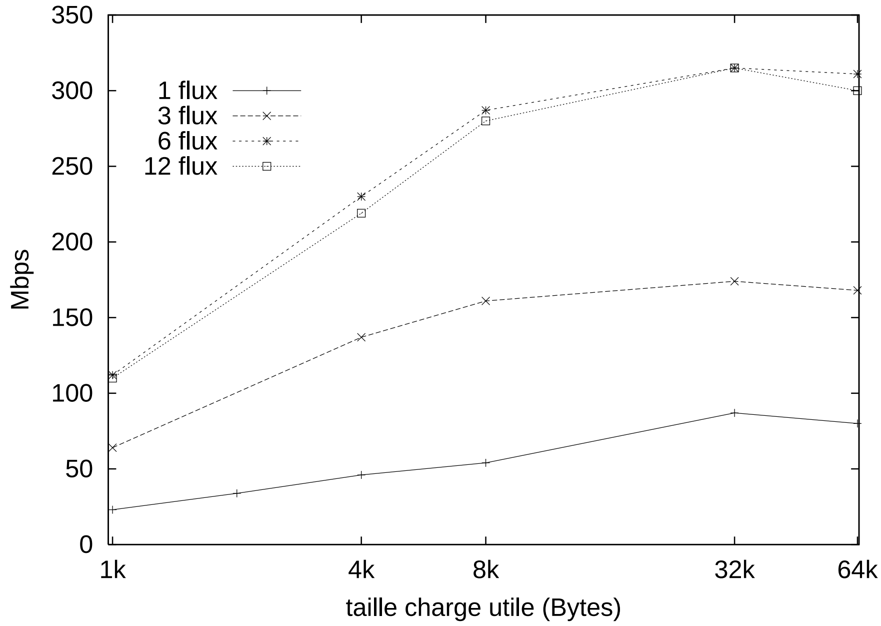


FIG. 6.14 – Débits sur réseau Myrinet avec un Tamanoir/3BE appliquant un service lourd.

réseau FastEthernet sur un commutateur avec un lien GigaEthernet ;
- ENS Lyon reliée également directement à VTHD par un routeur gigabit CISCO 6500. A ce routeur sont connectés en GigaEthernet sept racks Compaq Proliant (plate-forme 2) ainsi qu'un cluster de seize racks SUN LX50 interconnecté en FastEthernet par l'intermédiaire d'un commutateur Netgear.

Les adresses IP attribuées ne sont visibles que par VTHD. Un poste connecté à l'Internet ne peut donc accéder directement aux ressources de VTHD sans « rebondir » sur une passerelle (*gateway*). La passerelle que nous utilisons à l'ENS n'est pas représentée sur la figure.

La figure 6.16 présente les résultats des premières mesures sur le réseau longue distance haut débit VTHD. Nous fixons le nombre de flux à 12 et faisons varier le nombre de Back Ends (BE). Le service appliqué est un service léger. Afin de supporter au mieux les petits paquets il semble préférable d'employer le plus de BE possible. Par contre pour des paquets de 8kB et plus il semble inutile de mettre à disposition un nœud Tamanoir avec plus de deux BE puisque nous sommes déjà à environ 900Mbps. Enfin pour les très gros paquets (64kB) le Tamanoir à six BE supporte mieux la charge. Un relevé de la consommation CPU des BE lors de l'expérience à 12 flux distribués sur 6 BE, pour des paquets de 8kB auxquels on applique un service léger montre qu'ils sont chargés à seulement 25%. Lors de la même mesure avec seulement 2 BE, ces derniers sont chargés à plus de 90%, cela signifie qu'ils ont atteint leur limite et qu'il n'ont plus de ressources suffisantes pour traiter des flux supplémentaires avec le même service.

La figure 6.17 permet de comparer les débits obtenus par un service léger et un service lourd. Les machines clientes réparties autour de VTHD génèrent six flux qui sont traités par un Tamanoir cluster à 6 BE. Le débit agrégé du service lourd est d'un peu moins de 200Mbps. Le service léger oscille autour de 500Mbps pour 6 flux alors que dans la figure 6.16 précédente on oscillait autour de 800 Mbps pour 12 flux. Cela prouve que l'on profite clairement de

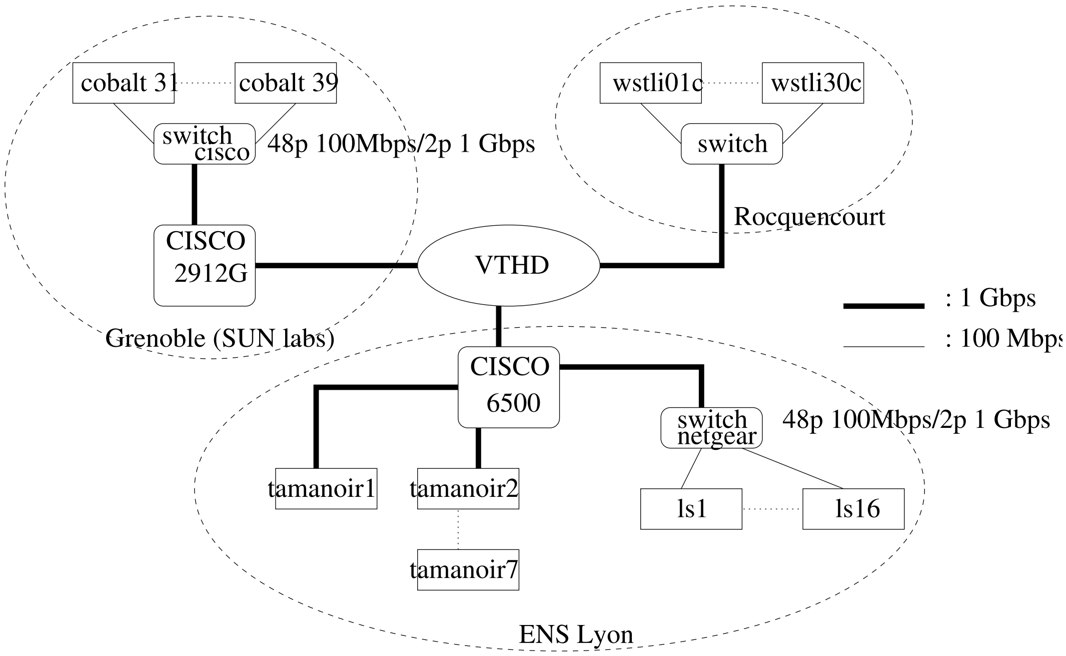


FIG. 6.15 – Topologie physique de l'environnement d'expérimentations sur VTHD.

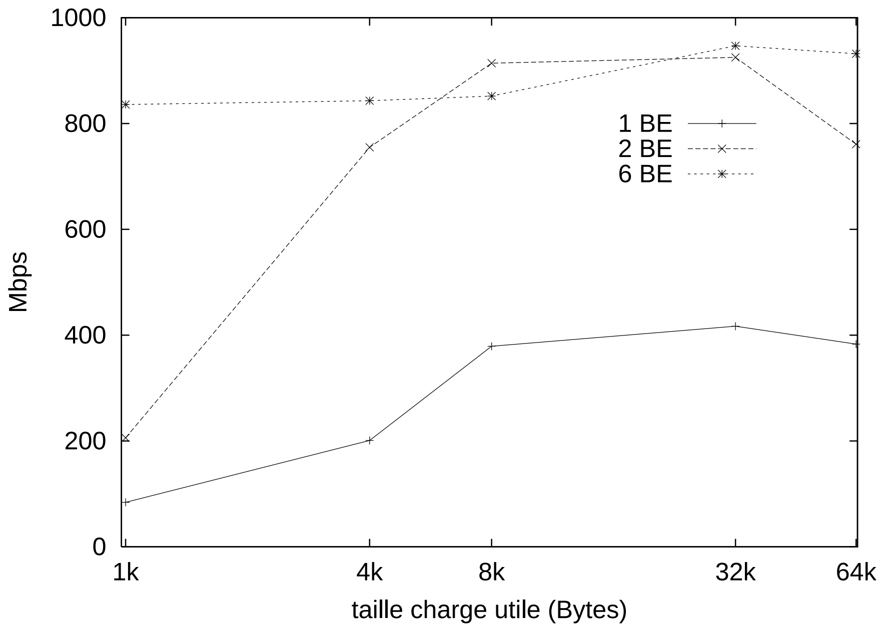


FIG. 6.16 – Mesures sur VTHD. 12 flux traités par un service léger dans un Tamanoir simple (1BE), puis un Tamanoir cluster à 2 ou 6 BE.

l'architecture SMP. La courbe supplémentaire, *1 BE service léger* illustre l'intérêt d'employer un Tamanoir cluster autour d'une épine dorsale hautes performances surtout pour les petits paquets.

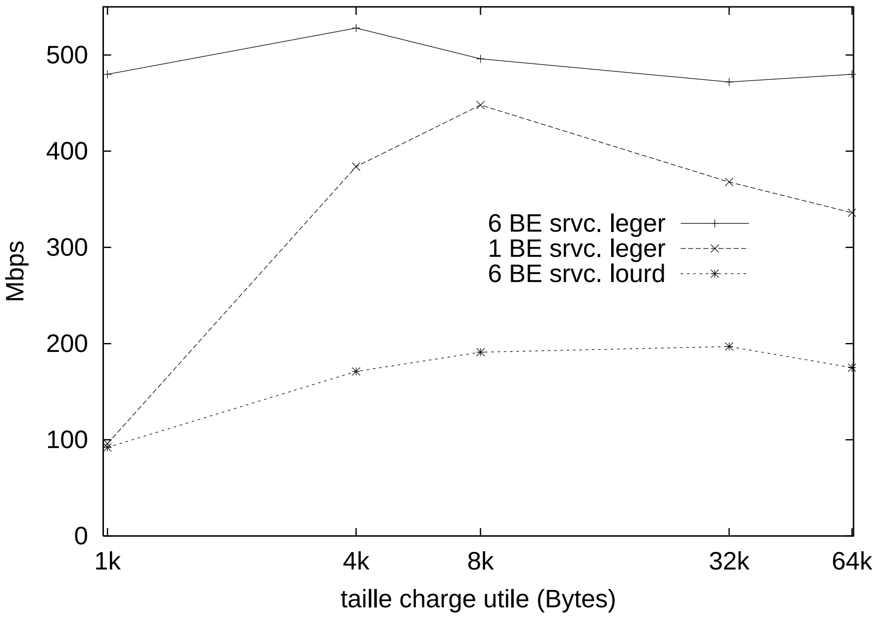


FIG. 6.17 – Mesures sur VTHD d'un service lourd contre un service léger exécuté sur 6 flux par un Tamanoir Cluster à 6 BE ou un Tamanoir simple.

6.6 Expérimentations annexes

6.6.1 Performance d'un service de stockage à la volée avec IBP

Dans cette section nous mesurons le coût supplémentaire introduit par l'opération de cache dans un dépôt IBP proche du nœud actif Tamanoir. Pour ces résultats nous exécutons un dépôt IBP sur le même nœud que le TAN, avec le service *XcastIbpService* (décrit section 5.4.2) Pour réaliser ces mesures nous utilisons la plate-forme 1. Un PC exécute un TAN et un dépôt IBP. Les autres ont soient le rôle d'émettre ou de recevoir des données. Les mesures sont réalisées grâce à Netfilter qui nous permet de placer un chronomètre au plus prés de l'interface réseau. Dans la figure 6.18 nous présentons le temps de traversée par un paquet ANEP d'un noeud Tamanoir avec ou sans stockage dans un dépôt IBP.

Comme on peut le constater l'opération d'écriture dans le dépôt IBP n'introduit pas un coût supplémentaire trop important. Quelle que soit la taille des paquets le coût reste constant. Pour les petits paquets, les performances sont faibles dû à la politique de transmission de la pile TCP du noyau Linux qui agrège les petits paquets avant qu'ils soient transmis. Donc les petits paquets sont libérés après un temporisateur.

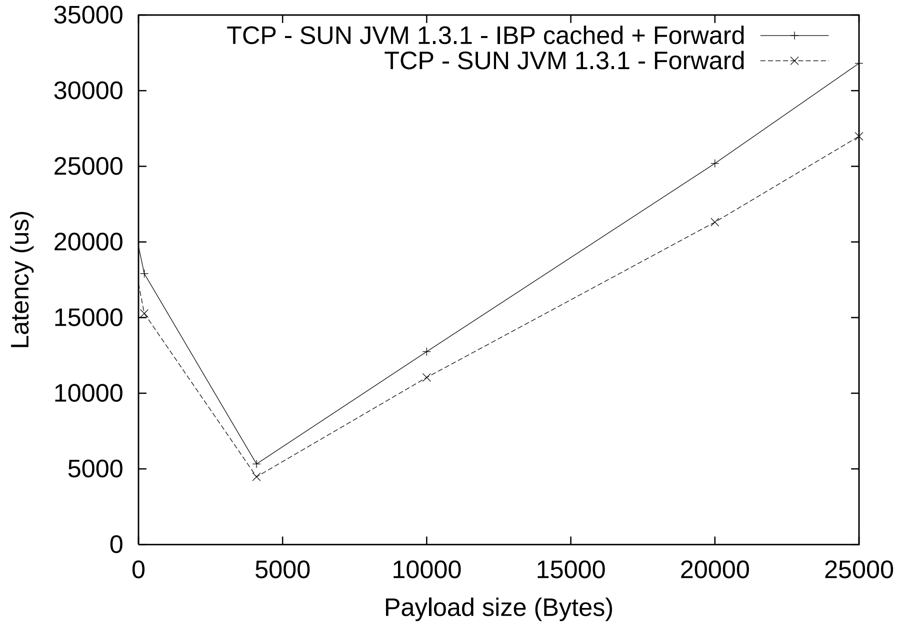


FIG. 6.18 – Coût supplémentaire du stockage du flux de données dans un dépôt IBP.

6.6.2 Impact du format de paquet sur les performances

Le démultiplexage rapide d'un en-tête de paquet actif est un critère important dans un contexte de réseau actif haute performance. Le format de paquet ANEP fournit une grande souplesse pour ajouter de nouvelles options. Malheureusement, la récupération des informations contenues dans les champs options de l'en-tête demande de nombreuses opérations. Par exemple, si l'une des options est un entier, on doit lire son type et sa longueur puis convertir le tableau d'octets qui suit en un entier. Pour les adresses IP stockées dans les options de l'en-tête ANEP, nous utilisons actuellement des chaînes de caractères. Cela oblige de lire et convertir le tableau d'octet en une chaîne de caractères et de convertir enfin cette chaîne en une IP physique (`InetAddress`).

Les expérimentations que nous avons menées nous ont permis de constater que les options que nous utilisons et/ou avons défini pour Tamanoir ont convergé vers un petit nombre. Les applications qui nécessitent une option supplémentaire peuvent très bien la stocker dans la charge utile du paquet actif. Afin d'améliorer les performances et de minimiser le nombre d'opérations effectuées sur chacun des paquets arrivant dans un paquet actif et donc susceptible d'être entièrement « décortiqué » nous avons évalué notre format APF (*Active Packet Format*) basé sur le format ANEP simplifié donc les champs sont fixés en taille et position.

Pour les mesures nous utilisons une ancienne version de Tamanoir modifiée pour accepter le format de paquets actifs APF. L'expérience a été réalisée avec un service léger au-dessus de `TCP` sur la plate-forme 2 (GigaEthernet). La figure 6.19 présente les mesures de débits pour un ou deux flux de données dont on fait varier la taille de la charge utile des paquets de 128B à 64kB. On remarque un gain notable pour les paquets de petites tailles. De 128B à 1kB que ce soit pour un ou deux flux on double le débit en utilisant un format de paquets simplifié. Pour les paquets de plus grosse taille les performances sont bien moins bonnes car cette version de

Tamanoir modifiée n'a pas encore bénéficié de toute les optimisations aptes à mieux supporter les paquets de grande taille. On peut donc raisonnablement supposer qu'à partir de 4kB les courbes devraient se superposer.

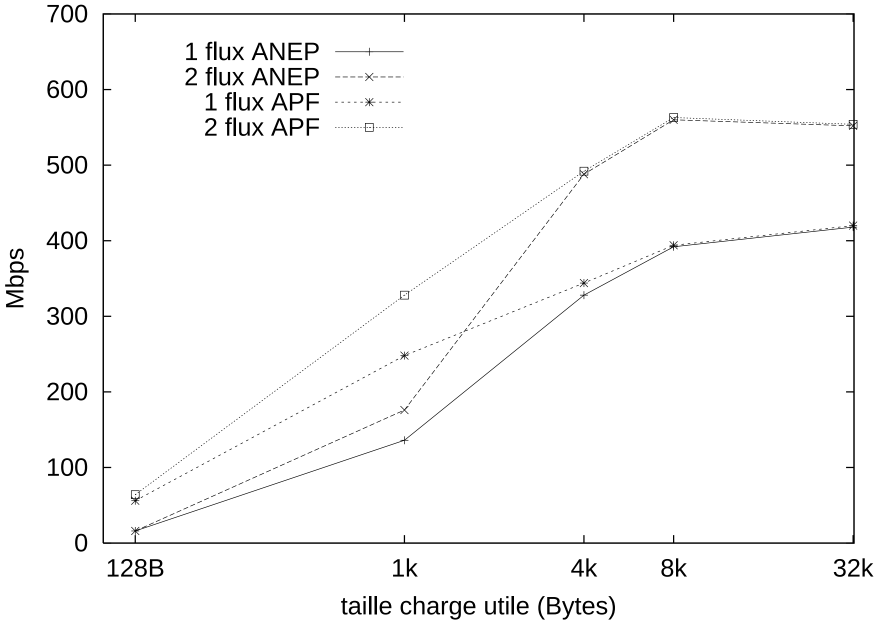


Fig. 6.19 – Impact du format de paquets (ANEP, APF).

Malgré tout, le format ANEP garde l'avantage d'être plus souple et extensible. Utilisé un format de paquets actifs « propriétaire » peut néanmoins poser des problèmes de compatibilité avec des équipements actifs qui exécutent un autre EE ne supportant que le format ANEP. Toutefois, c'est une piste à ne pas négliger surtout pour des services qui manipulent de manière intensive l'en- tête des paquets actifs. De plus, étant donné le caractère figé de l'en- tête, il devient difficile d'ajouter un nouveau champs d'option. Il faudra donc adapter l'entête lors d'un passage à IPv6 puisque nous n'utilisons que 4 octets pour transporter les adresses IPv4.

6.7 Déploiement de Tamanoir dans différents projets

La suite logicielle Tamanoir est actuellement utilisée comme plate-forme de déploiement et support dynamique dans différents projets internes, développés au sein de notre équipe (RESO) ou externes et pris en charge par des équipes extérieures. Ces déploiement de Tamanoir dans des contextes hétérogènes nous permettent d'avoir des retour d'informations des utilisateurs. Tous profitent de la mise en œuvre facile et rapide d'une plate-forme Tamanoir et de son aspect ouvert pour pouvoir tester et évaluer de nouveaux services réseaux ou protocoles de transports.

6.7.1 Projets internes

– **VTHD++** : Le projet VTHD (Vraiment Très Haut Débit) repose sur un réseau expérimental très haut débit (2.5 Gbps). L'équipe RESO est impliquée dans le sous projet 4 (réseaux haute performance et services). Notre objectif dans ce sous projet est de valider l' Environnement d'Exécution haute performance Tamanoir pour les réseaux actifs sur un réseau expérimental Gigabit longue distance. Dans le cadre de ce sous projet nous avons déployé des nœuds actifs en périphérie de VTHD ; le but de nos travaux concerne la validation du routeur actif logiciel Tamanoir apte à supporter les flots de trafic gigabit provenant de VTHD. Ces routeurs actifs sont ainsi utilisés dans le cadre d'applications multimédia ou de grille déployées sur des épines dorsales haute performance. Des expérimentations de réseaux actifs pour la qualité de service (QoSINUS) et le multicast fiable (DyRAM) ont été ainsi mené sur ces nœuds actifs. La plupart des mesures présentes dans ce document ont été réalisées avec Saâd El Hadri, ingénieur expert INRIA sur le projet VTHD.

- **e-Toile** : Le projet e-Toile (RNTL) est de développer et déployer une infrastructure de grille haute performance. La grille e-Toile est construite au dessus du réseau VTHD. L'équipe RESO, à travers ses recherches sur les réseaux actifs et service actif de multicast fiable et de qualité de service, fournit un support réseau pour les middlewares et les applications de grille.
  - **QoSINUS** (QoS INvocation and Usage System) : La plupart des routeurs actuels proposent plusieurs classes de services différenciés (DiffServ). Les paquets sont marqués avec l'une de ces classes et reçoivent le traitement de transmission associé à cette classe dans les routeurs. Le projet QoSINUS [80, 81] permet à une application de spécifier par une requête de haut niveau son besoin de QoS en terme de délais, pertes et débits. La classe de service est alors choisie dynamiquement en fonction des conditions de transmission dans le réseau. Le système de QoS programmable proposé permet de lier la requête de QoS d'une application haut niveau avec les routeurs de l'épine dorsale qui appliqueront le traitement de transmission requis. Pour fournir cette fonctionnalité un service actif est exécuté sur le routeur localisé en entrée de l'épine dorsale. Ce service a la responsabilité de monitorer le comportement de chaque classe DiffServ dans le cœur du réseau en terme de délais, pertes et débits. En fonction de la requête de haut niveau reçue, il détermine la classe la mieux adaptée. C'est également ce service actif qui est en charge de marquer de la classe appropriée les paquets du flux. Le tout permettra d'assurer la QoS requise par l'application cliente.
  - **DyRam** : est un protocole de multicast fiable avec une stratégie de recouvrement basés sur une structure en arbre qui s'appuie sur les paquets avec l'assistance des routeurs [82, 83]. Le protocole DyRam est une amélioration des propositions existantes mais il a été conçu pour fournir des fonctionnalités actives à faible coût avec une élection dynamique du re-émetteur basé sur les paquets. Le protocole repose sur les NACKs (accusés de réception négatif) avec les recouvrements locaux basé sur les récepteurs. Ces derniers sont responsables de la détection des pertes et de la retransmission des paquets lorsque cela est possible. Afin de réaliser un contrôle de flux et un contrôle de congestion [84] ainsi qu'une gestion mémoire en plus d'une élection du re-émetteur plus efficace, les ACKs sont agrégés dans les NACKs. Dans l'absence de NACKs, les ACKs sont également agrégés périodiquement dans des messages spéciaux appelés “Congestion Report” (CR). Ces CR contiennent l'information nécessaire pour le contrôle de flux et de congestion. Les équipements Tamanoir qui jouent un rôle prépondérant dans DyRAM fournissent, en plus des services qui évitent les congestions et réaliser le contrôle de flux, les services actifs suivants :
    - la suppression des NACKs dupliqués afin de limiter le problème d'implosion des

NACKs.
- l'agrégation des CR ce qui évite leur implosion aux nœuds intermédiaires et à la source.
- la diffusion des paquets de réparations seulement vers les récepteurs ayant subi des pertes.
- l'élection dynamique d'un re-émetteur qui consiste à choisir un lien comme re-émetteur pour réalisé un recouvrement local à partir d'un flux descendant d'un récepteur au lieu de cacher les paquets de données dans les routeurs. L'élection dynamique fournit une bonne robustesse contre les pannes d'un hôte ou d'un lien.
- une détection précoce des pertes de paquets perdus et l'émission des NACKs correspondants. Cela permet de fournir un délai de recouvrement très court.

Tamanoir a été utilisé pour une validation expérimentale de ce protocole de mulicast fiable.

6.7.2 Projets externes

- **Laboratoire LIRIS - INSA/Lyon** : *Intégration de la fonction de proxy cache aux routeurs actifs* : Sid Ali Guebli et Jean-Marc Pierson [85]. Comme les services Web sont devenus de plus en plus populaires, les utilisateurs souffrent des problèmes de surcharge des serveurs et de congestion du réseau. Plusieurs recherches ont été faites pour améliorer la performance du Web. Le cache est reconnu comme étant une technique efficace pour diminuer la congestion des serveurs, et de réduire le trafic réseau, et de cette façon, minimiser le temps de latence des utilisateurs. Exploiter la coopération entre les caches permet d'accroître les performances du système de cache, plutôt que d'avoir plusieurs proxies qui accèdent aux mêmes ressources de manière indépendante. La clé de l'efficacité d'un schéma de coopération réside dans le protocole de communication de cache. Ils ont donc développé un protocole de coopération pour des systèmes de caches, en intégrant la fonctionnalité de proxy cache aux routeurs actifs, et en profitant de la souplesse des réseaux actifs. Le but principal de ce système de cache est de permettre une localisation rapide d'un document dans une communauté de caches coopératifs, en évitant que les caches parent soient obligés de garder une copie de ces documents, comme c'est le cas pour les caches hiérarchiques. Dans cette approche, et contrairement à l'approche hiérarchique, le cache n'a pas besoin d'interroger ses voisins pour déterminer si un objet existe chez eux ou non. On évite les problèmes de gestion liés à la maintenance des informations des caches voisins pour chaque cache fils. Cette tâche est attribuée aux caches parents : chaque cache à une vision globale du contenu de ses fils en gardant la trace de chacun d'eux. Cela permet à tout cache fils de connaître le contenu de ses caches voisins en envoyant une seule requête à son cache parent. Ce système supprime l'obligation d'avoir une grande capacité de stockage au niveau des caches parents, et d'utiliser la hiérarchie pour avoir une localisation rapide des objets. Tamanoir a été utilisé dans ce projet pour valider le modèle d'intégration de la fonction de proxy cache aux routeurs actifs qu'ils ont mis au point.

- **Laboratoire LAAS - Toulouse** : *Support des réseaux actifs Tamanoir pour le déploiement du protocole de transport FPTP* : Ernesto Exposito, Patrick Sénac and Michel Diaz[86]. Le protocole de transport FPTP (Fully Programmable Transport Protocol) représente un exemple d'implémentation de services de communications pour flux multimédia dans le cadre de l'architecture XQoS. Ces services sont réalisés en déployant de nouveaux mécanismes de transport et en configurant les mécanismes de transport et réseau existants. Des solutions dans le cadre de réseaux actifs sont envisagées afin de résoudre la problématique liée au déploiement des services FPTP (Tamanoir, Sara). Des expériences à l'échelle du réseau de recherche européen ont permis de valider expérimentalement ce protocole (GCAP [87]).

– **Université de Bretagne Sud (Vannes)** :
Concerto : Composants parallèles adaptables : Yves Mahéo, Luc Courtrai, Frédéric Guidec[88]. Distribution de flux en java/équilibrage de charge. La plate-forme logicielle Concerto, développée au sein du groupe Casa du laboratoire Valoria (Université de Bretagne-Sud), permet de déployer des composants Java distribués sur une grappe de machines tout en fournissant à ces composants les moyens de s'adapter. Concerto définit un modèle simple de composant distribué, c'est-à-dire de composant mettant en jeu plusieurs activités parallèles réparties sur une architecture distribuée. Le programmeur du composant peut faire appel à une boîte à outil lui permettant de construire un composant capable de s'adapter à son environnement d'exécution modélisé par un ensemble de ressources. Les ressources considérées sont d'une part les ressources offertes par le système (mémoire disponible, CPU, interfaces réseau,...) et d'autre part des ressources dites « conceptuelles », liées à l'application elle-même (sockets et threads utilisés dans l'application,...). L'infrastructure est extensible dans la mesure où elle est conçue de manière à pouvoir incorporer facilement de nouveaux types de ressources. Concerto a été utilisé pour encapsuler dans un composant adaptatif le code d'une version écrite entièrement en Java du code d'un nœud actif Tamanoir. Ce code réalise les fonctions d'un routeur actif, capable d'appliquer un service adapté (compression, cryptage, filtrage,...) sur chacun des flux qui le traversent. L'architecture d'un nœud actif Tamanoir est distribuée afin qu'il puisse être implanté sur une architecture parallèle comme une grappe et d'en exploiter la puissance de calcul et toutes les ressources qui peuvent éventuellement être réparties de manière irrégulière sur cette grappe (*e.g.* puissance de processeurs ou quantité de mémoire différentes, cartes de chiffrement ou de compressions installées sur certains nœuds de la grappe,...). Cette architecture repose sur la présence d'un frontal distribuant les flux entrant vers des entités appliquant les services. Le composant Concerto obtenu est constitué d'un ensemble de threads regroupés en deux types de *fragments* (unités de placement) :
  – un fragment (frontend) composé des différents threads du frontal Tamanoir, chargé de rediriger les flux entrant dans le routeur vers les backends. Un seul fragment de ce type est présent dans le composant.
  – un fragment (backend) composés des threads mettant en œuvre des services actifs. Un fragment de ce type est présent sur chacun des autres nœuds de la grappe.

  La plate-forme Concerto a permis de :
  – spécifier et effectuer le déploiement des différents fragments du composant (frontend et backends) sur une grappe de machines ;
  – construire rapidement des stratégies d'affectation des flux vers les backends. Cette affectation peut être aisément effectuée par le frontend à partir de la collecte à distance d'informations sur l'état des ressources des backends. On peut ainsi prendre en compte plusieurs critères comme la présence d'un dispositif matériel spécifique sur un des nœuds, l'espace disque disponible ou la charge CPU de chacun des nœuds.

La possibilité de définir ses propres ressources au sein de Concerto, en réifiant des notions propres à l'application, permet d'envisager de doter le composant Tamanoir de nouvelles ressources conceptuelles (services, flux,...) dont l'état deviendrait facilement accessible au sein de l'ensemble du composant. Le programmeur du composant dispose d'un cadre de conception le guidant dans la création de nouvelles ressources conceptuelles. Un composant Concerto possède une interface dite interface « ressource ». Via cette interface, les clients de ce composant (hors de la grappe) peuvent observer l'état des ressources du composant. On peut donc envisager de construire rapidement en Java un moniteur de l'activité du routeur Tamanoir, moniteur qui aurait accès à la fois à l'état des ressources système des machines de la grappe et à celui des ressources conceptuelles définies par l'application.

6.8 Conclusions sur les expérimentations

Les mesures de performances sont des opérations longues mais indispensables. Elle doivent être menées de manière régulière afin de mesurer l'impact des modifications de la partie logicielle de l'Environnement d'Exécution ou de l'ajout de nouveaux matériels. Nous nous sommes parfois heurté à des systèmes mal configurés ou qui n'utilisaient pas les derniers pilotes disponibles pour tirer le meilleur partie de l'architecture. Le passage d'une architecture matérielle plus récente à une autre, doublé parfois, d'un changement de technologie réseau (FastEthernet, GigaEthernet, Myrinet) complexifie d'autant plus l'analyse des mesures.

La plupart des mesures présentes dans ce chapitre sont réalisées avec des services légers. Cela signifie que ce sont des services qui consomment peu de cycles CPU et peu de mémoire pour chaque paquet traité. L'objectif ici n'est pas de fournir un service actif performant, optimal dans un contexte de traitement à la volée par exemple, mais de fournir un EE optimal, suffisamment générique pour des services actifs à développer. Si l'on considère la latence introduite par un service léger comme extrêmement faible, les résultats produits ci-dessus mettent en évidence les performances brutes de l'EE à récupérer un paquet, l'analyser, le transmettre au service adéquat, puis le retransmettre sur le réseau.

	FastEthernet		GigaEthernet		Myrinet	
	Mbps	pps	Mbps	pps	Mbps	pps
1 flux	80	8759	418	16600	624	18554
x flux	96	9765	560	21484	792	30273
x flux/LVS	-	-	-	-	1700	95703

TAB. 6.1 – Débits maximum atteints avec un service léger sur TCP.

Le tableau 6.1 est un récapitulatif des débits maximum atteints avec un service léger sur TCP et sur trois technologies réseaux différentes (FastEthernet, GigaEthernet et Myrinet). Les résultats en Mbps ont été obtenus avec des paquets de 8 ou 32kB. Les résultats en paquets par seconde (pps) ont été obtenus avec des paquets de plus petite taille (1kB). Bien que les résultats en Méga bits par seconde soient encourageants, les résultats en paquet par seconde restent inférieurs à ceux généralement annoncés par les constructeurs de routeurs.

Finalement, les expériences menées dans ce chapitre ont montré que nous sommes aujourd'hui capable de supporter les débits des réseaux locaux (LAN) classiques à 100 Mbps. Nous supportons également à l'aide d'une solution issue du calcul parallèle (grappe de PC) les réseaux au Gigabit. Enfin nous avons également montré qu'il est facile de déployer des nœuds Tamanoir sur un réseau longue distance et de mener des expériences autour d'une épine dorsale Gigabit maîtrisé (VTHD). En conclusion Tamanoir fournit une plate-forme d'exécution idéale pour déployer, tester et valider facilement de nouveaux services.

Chapitre 7

Conclusions et perspectives

Ce chapitre clos ce document. Une première section résume les principales contributions et nouvelles connaissances produites dans cette thèse. La section suivante ouvre de nouvelles perspectives de recherches et développements.

7.1 Travaux présentés

Dans ce document nous nous proposions de résoudre deux problématiques, l'une sur un aspect architectural et l'autre sur un aspect mise en œuvre d'un nœud actif logiciel haute performance. Nous avons également proposé une validation expérimentale et fonctionnelle de la mise en œuvre de notre architecture active.

7.1.1 Une architecture active haute performance

Nous avons proposé dans cette thèse une architecture originale de nœud actif logiciel multi-couches, multi-protocoles et multi-services.

Notre architecture est basée sur un modèle en couches. Chaque couche correspond à un espace d'exécution des services. La distinction d'une couche par rapport à une autre se fait en fonction des caractéristiques et fonctionnalités qu'elle propose. La couche de l'espace utilisateur autorise un accès aisé à toutes les ressources du système (mémoires et CPU générique). La couche de l'espace noyau et celle d'une carte d'interface réseau programmable sont plus proches du lien. Enfin, la couche des ressources distribuées permet de profiter de l'accélération conférée par le traitement en parallèle des paquets. Remarquons que les services actifs qui font du cache dans le réseau pourront également profiter de la grande quantité de mémoire de masse disponible.

Notre architecture est mutli protocoles cela signifie qu'elle supporte les principaux protocoles de transports disponibles.

Elle est également multi-services, cela signifie qu'elle supporte le chargement et l'application simultanés de différents services sur différents flux.

Nous avons ensuite proposé une classification des services en fonction des ressources qu'ils sont susceptibles de consommer. Un service est considéré comme lourd s'il est très consommateur en cycle processeur ou mémoire. Un service est léger s'il n'a besoin de garder que quelques états sur le flux et s'il applique un traitement peu consommateur en cycle processeur.

Grâce à cette classification nous positionnons l'exécution des services dans les couches appropriées (déploiement vertical). Plus le service est léger (ex : plan contrôle) plus on privilégiera une couche basse, près du lien.

Nous proposons le support de différents formats de données. Le format ANEP standard, un format de données brutes sans structure d'encapsulation, et un format de paquet simplifié appelé APF (*Active Packet Format*). Ce dernier restreint la flexibilité de l'en-tête du paquet mais facilite sa lecture par l'usage de champs d'informations fixés (en taille et position).

Enfin nous proposons deux modes de communication. Un mode proxy et un mode bout en bout. Le premier repose sur des équipements en attente sur un numéro de port donné. Il permet de lier deux clients par l'intermédiaire de connections indépendantes entre chaque équipement actif. On peut lui reprocher de briser le modèle de bout en bout mais il a pour avantage, de faciliter le déploiement de nœuds actifs, et en adressant explicitement leur paquet à un équipement actif, les applications s'assurent que le flux passe par un équipement actif.

Le second mode de communication repose sur des équipements autonomes non déclarés auprès des applications. Ce mode propose d'attraper les paquets à la volée quand ils passent dans un équipement actif. Cela implique de repérer les paquets d'un signe distinctif pour qu'ils soient capturés et traités.

7.1.2 Tamanoir : une mise en œuvre de nœud actif haute performance

La mise en œuvre de notre architecture décrite dans ce document (et résumé ci-dessus) nous a conduit à l'écriture du premier Environnement d'Exécution (EE) logiciel actif haute performance, appelé *Tamanoir*. Il constitue une mise en œuvre pragmatique de notre architecture de nœuds actifs logiciels haute performance.

Pour l'espace utilisateur nous avons écrit un EE complet et une bibliothèque de services en Java. Nous avons utilisé Java pour répondre au problème de l'hétérogénéité importante dans ce type d'application distribuée. Un service peut donc être chargé dynamiquement et exécuté sur n'importe quel équipement quel que soit son architecture matérielle et logicielle pourvu qu'une machine virtuelle Java soit disponible. Bien que l'usage du langage Java puisse faciliter le déploiement il constitue une contrainte supplémentaire pour une mise en œuvre de notre architecture dans un contexte haute performance, car Java propose une abstraction importante du système sous-jacent.

Notre EE Tamanoir est le premier nœud actif disponible qui supporte le protocole de transport TCP. Cette caractéristique est maintenant nécessaire pour certaines applications qui exige de communiquer leurs flux sur des liaisons fiables. On tire avantage de l'aspect connecté de TCP en associant une instance de service par flux. Cela permet de maintenir des variables d'état sur chaque flux. C'est donc un EE multi thread. Notre EE supporte également UDP.

On profite du mécanisme de chargement dynamique de modules et des caractéristiques offertes par *Netfilter* disponible dans le système d'exploitation Linux pour l'écriture de services légers exécutés dans l'espace noyau. Ces services sont écrits en C ce qui limite la portabilité au profit des performances. Les services noyau ont l'avantage d'être exécutés au plus près du lien et d'accéder directement à la zone mémoire où les paquets arrivent, sans passer par une interface et sans changement de contextes. Tous les projets existant concentrent l'exécution de leurs services dans une couche unique de l'équipement, soit dans l'espace utilisateur, soit dans l'espace noyau. Par exemple, le projet PAN[89] propose les deux mises en œuvres, mais sous la forme de deux prototypes indépendants. Notre architecture propose de déployer un même service sur plusieurs couches de l'équipement actif afin de placer les différentes fonctionnalités du service dans les couches les mieux appropriées. On utilise la portion de service exécutée dans l'espace utilisateur pour assister le déploiement de la portion de service exécutée dans l'espace noyau. Un mécanisme de communication inter couche est également prévu afin que les deux portions d'un service communiquent. La portion d'un service exécuté dans une couche haute peut émettre des consignes vers ses fonctions simples et rapides exécutées dans l'espace noyau, et vice et versa.

En profitant de notre expertise du monde du calcul distribué, nous avons proposé un modèle qui exploite des solutions issues des systèmes distribués destinés à paralléliser l'exécution d'applications et fournir un nœud actif extensible en performance. Nous proposons deux architectures distribuées : l'une basée sur l'architecture SMP, l'autre basée sur l'usage d'une grappe de PC (Tamanoir cluster). Ces architectures distribuées peuvent être associées en employant des systèmes SMP dans la grappe de PC. Les services lourds sont exécutés dans l'espace utilisateur d'un Tamanoir simple nœud, ou dans l'espace utilisateur de l'un des nœuds d'un Tamanoir cluster. Les services légers sont exécutés dans l'espace noyau, ou dans l'espace utilisateur. Pour la mise en œuvre de la couche des ressources distribuées nous nous basons sur le projet Linux Virtual Server (LVS). Nous proposons de répliquer un nœud Tamanoir simple sur plusieurs nœuds d'une grappe de PC équipée d'une machine frontal destinée à distribuer les flux sur les nœuds de la grappe.

Nous proposons ensuite deux mécanismes de déploiement de services dynamiques qui libèrent un administrateur réseau des fastidieuses tâches de mise à jour des équipements. Le premier utilise un *service repository* qui facilite l'injection de services approuvés dans le réseau, le second emploie un mécanisme de déploiement de proche en proche.

Le premier prototype de l'EE Tamanoir est entièrement basé sur du matériel et des logiciels standards. Il est disponible, logiciel libre, déposé auprès de l'APP (Agence de Protection des Programmes) et exploitable immédiatement dans des applications réalistes.

7.1.3 Expérimentations

Les projets cités dans l'état de l'art se sont pour la plupart limités à des services « jouets » déployés sur des plate-formes d'expérimentations locales dédiées. On regrette l'absence d'une phase de valorisation de leurs travaux par des expérimentations de grande envergure sur un réseau longue distance. Une étape fondamentale de ces travaux de thèse a été de proposer une validation autant expérimentale que fonctionnelle de notre architecture.

Pour la validation expérimentale nous avons mené des campagnes de mesures sur quatre technologies de réseaux différentes. Une technologie de réseaux locaux classiques (100Mbps, Fast Ethernet), une technologie de réseaux locaux haute performance (1000Mbps, Giga Ethernet), une technologie de type SAN (*System Area Network*) très haute performance, faible latence (Myrinet), et une technologie réseau haut débit (1Gbps) longue distance autour d'une épine dorsale (VTHD).

Grâce au projet Tamanoir nous proposons aujourd'hui un équipement actif apte à supporter un lien Gigabit et donc déployable autour d'une épine dorsale au Gigabit. Bien que nous nous soyons focalisé sur la performance nous ne nous sommes pas limité au support de services légers (uniquement destinés au plan contrôle par exemple). Nous fournissons un environnement apte à supporter des services de haut niveau, capable d'agir sur le plan données des paquets. Les résultats obtenus pour un service lourd (compression à la volée) sont prometteurs et montrent l'intérêt d'agréger encore plus de ressources de calcul dans un équipement actif.

La validation fonctionnelle a été réalisée par l'utilisation de la plate forme Tamanoir dans des projets menés par des membres de notre équipe mais également dans des projets conduits par des équipes extérieures. La mise en œuvre de prototypes de protocoles ou services avec la plate-forme Tamanoir leur a permis d'établir de rapides validations expérimentales. Le retour d'information de ces utilisateurs a été très positif, a souvent contribué à l'amélioration de la suite logicielle Tamanoir sur le plan ergonomique et a confirmé l'intérêt de disposer d'une plate-forme d'expérimentations de réseaux actifs française ouverte et facilement déployable.

Nous avons également montré qu'il est possible de faire du routage sur un réseau Gigabit avec un PC standard sous Linux, tout en appliquant des services actifs sur un lien Gigabit sans l'assistance de matériels dédiés.

Les réseaux grande distance maîtrisés et l'Internet seront marqués dans les prochaines années par une formidable demande de la part des applications et des utilisateurs en nouveaux services réseaux (plan contrôle ou plan données). Les organismes de standardisation, comme l'IETF, imposent des processus de standardisations longs et complexes qui ralentissent le déploiement de nouveaux services dans les réseaux. La technologie des réseaux actifs permet de court-circuiter ces organismes en proposant une solution intermédiaire pour la mise au point de nouveaux protocoles. Grâce aux travaux apportés par cette thèse, les réseaux actifs autorisent maintenant des expérimentations réalistes de services (support du débit Gbits, déploiement autour d'épines dorsales,...). Une fois les services mis au point et validés à large échelle, un constructeur peut les embarquer dans des équipements matériels.

7.2 Perspectives

Les conclusions apportées par cette thèse laissent entrevoir de nombreuses perspectives intéressantes.

7.2.1 Supporter les réseaux du futur

Dans ce projet nous avons atteint le Gigabit. Nous nous posons maintenant la question de savoir comment nous pourrions supporter plus de bande passante (plusieurs Gigabits). Cette question mérite d'être posée dés à présent car ce sont des débits de réseaux techniquement déjà disponibles. En augmentant simplement le nombre de nœuds de la grappe de calcul attachée (Tamanoir cluster) nous pourrions probablement supporter quelques Gigabits supplémentaires par seconde. Mais pour atteindre des débits vraiment supérieurs une assistance matérielle semble s'imposer. Nous pensons à l'adjonction de matériels dédiés localisés au plus près du lien pour l'application efficace de services légers. Ces matériels peuvent comprendre des cartes d'interfaces réseaux programmables ou des solutions à base de *Network Processors* (NP). Il pourrait également être intéressant d'adapter une version de notre EE Tamanoir pour qu'il exploite de manière moins triviale un système multi processeur (SMP).

7.2.2 Faciliter l'intégration dans les réseaux

Une option qui n'a pas été explorée, plus liée aux problèmes de déploiement des nœuds actifs que de performance, est de mixer les modes proxy et bout-en-bout sur le chemin des données. Le mode proxy plus simple à déployer emploie des connexions explicites entre chacun des équipements. Le mode bout-en-bout, plus transparent, est mieux adapté à un déploiement massif (scalable) d'équipements actifs ; néanmoins il ne garantit pas que les paquets seront traités. L'objectif est de ne pas modifier les applications clientes et de garantir le passage du flux dans au moins un équipement actif en employant le mode proxy entre le client et le premier nœud actif. Ce nœud actif pourrait alors en plus d'appliquer un service, encapsuler les données dans un format de paquet actif (ANEP) ou les marquer d'un signe distinctif (bit IP router alert) pour qu'elles soient reconnues en tant que données à traiter par les équipements actifs dans lesquels elles sont susceptibles de passer ensuite.

7.2.3 Poursuivre la validation

Nous avons mené quelques tests préliminaires de fonctionnalités sur l'apect multi-noeuds actifs traversés par des flux actifs, mais les mesures présentées dans ce document se sont principalement focalisées sur l'évaluation d'un nœud actif unique, mono machine ou distribué. Il est nécessaire de poursuivre une évaluation de l'impact d'un déploiement massif de nœuds Tamanoir sur le chemin des données. Nous pourrions ainsi évaluer l'impact des connections par morceau (en mode proxy) sur les performances.

Il manque également une évaluation d'un nœud actif Tamanoir soumis à un très grand nombre de flux pour observer son comportement en fonction de la technologie réseau utilisée et du positionnement du service dans les couches de l'architecture. Cette expérimentation si elle n'est pas simulée, demande une quantité de ressources matérielles très importante et est donc difficile à mettre en œuvre. Ces travaux pourraient être menés à l'aide d'un vaste émulateur réseau comme celui mis en place actuellement en France dans le cadre du projet *Grid5000* (ACI GRID)[90].

7.2.4 Sécuriser le déploiement

La sécurité constitue un véritable frein au transfert technologique des réseaux actifs dans le monde industriel. C'est un problème complexe dont la solution ne doit pas réduire les performances de notre architecture au bénéfice de la sécurité. On parlera de sécurité sous contrainte de performances. Une réflexion préliminaire a montré que l'on pouvait sans trop de difficulté déployer de manière sécurisé des services entre les nœuds actifs et le service repository.

La sécurisation d'une infrastructure active complète pour un niveau de sécurité raisonnable, ne peut se limiter à la mise en place de mécanismes sécurisés de déploiement de services. Un équipement actif doit pouvoir s'assurer de l'authenticité d'un flux et de sa provenance. Il doit également pouvoir s'assurer de l'identité d'un destinataire. Il est donc impératif de mener des travaux sur la sécurisation pour ce type d'infrastructure hautement distribuée. On pourrait s'appuyer sur des modèles de sécurité mis au point pour des infrastructures un peu analogues comme les réseaux P2P (*Peer-to-Peer*), c'est à dire sans informations centralisées[91].

Mais pour assurer ce transfert technologique nous devons probablement restreindre l'ouverture et la flexibilité proposées par la définition des réseaux actifs. L'étude et la proposition d'une gamme de services dédiés ou « vendables » et la conception d'une interface en lien avec des standards établis pourraient constituer les leviers pour accélérer l'introduction de cette technologie dans le monde industriel, et simplifier la démarche de sécurisation.

7.2.5 Restreindre les fonctionnalités

Une autre solution très pragmatique est peut-être de limiter l'ouverture en contraignant par exemple les équipement actifs à ne supporter que des services de très bas niveau (ex :

plan contrôle). En restreignant les fonctionnalités d'un service on le maîtrisera mieux. Il sera ainsi plus aisé de le positionner dans l'équipement actif afin qu'il soit exécuté dans la couche la mieux adaptée.

7.2.6 Supporter de nouveaux domaines applicatifs

Au début des réseaux programmables les services proposés ne nécessitaient pas une grande capacité de calcul : quel que soit le débit du flux puisque c'était principalement des services de gestion des équipements qu'il était prévu de déployer. Ces services peuvent aujourd'hui être encore supportés par les réseaux actifs mais restent sur des plate-formes d'expérimentations maîtrisées et difficilement généralisables.

Les domaines d'applications des réseaux actifs proposés dans la littérature sont par exemple des services capable de supporter la gestion des réseaux ou leur reconfiguration dynamique, la communication de groupe (multicast) ou des applications multimédia.

Grâce à nos travaux sur la performance, les réseaux actifs peuvent maintenant efficacement supporter toutes ces applications et plus spécialement les applications multimédia. Mais déjà de nouveaux domaines d'applications se profilent à l'horizon et envahissent notre quotidien. La technologie active telle que nous la proposons doit pouvoir supporter des solutions dynamiques, pour des réseaux sans fils ultra mobile de type *adhoc* avec des terminaux à ressources limitées, des applications de traitement pervasif, des applications paire-à-paire (P2P), des applications de calcul distribué, et des grilles de calcul.

La promotion de la technologie des réseaux actifs passera ainsi par la proposition de nouveaux services. Il est donc impératif de tisser dés à présent des liens avec les concepteurs et développeurs des nouvelles applications pour mettre au point les services dont ils ont besoin. A cette occasion, nous continuerons à promouvoir l'environnement d'exécution Tamanoir afin qu'il soit déployé dans des contextes applicatifs académiques ou industriels hétérogènes.

Annexe A

Le langage Java

A.1 Java : un langage pour le développement d'un nœud actif haute performance

Cette section propose une introduction au langage Java ainsi qu'une justification complète sur le choix de ce langage pour le développement d'un nœud actif Tamanoir haute performance.

A.1.1 Introduction au langage Java

Le langage Java [59, 60, 61] créé dans les laboratoires de la société SUN Microsystem en 1994[1]. Le projet, intitulé *Green Project* était conduit par NAUGHTON, GOSLING et SHERIDAN. Lors de sa conception les deux objectifs principaux étaient de :

- fournir un environnement de programmation multi plate- forme, d'où la fameuse expression *« write once, run anywhere »* ;
- fournir un support de programme interactif intégré dans des pages web et exécutable dans tout type de navigateur (*applet*).

Finalement, Java s'est imposé comme un langage de programmation d'application aujourd'hui concurrent des langages C et C++. Java est un langage de programmation Orienté Objet (OO) dont la syntaxe est similaire au C, C++ ou Objective C mais n'est basé sur aucun de ces langages. Les concepteurs du langage Java ont ajouté des caractéristiques que le C++ n'avait pas (Garbage Collector (GC), multi-thread) et supprimé des caractéristiques du C++ meilleurs en théorie qu'en pratique (héritage multiple, surcharge d'opérateurs). Java n'a pas été conçu seulement pour être portable au niveau source mais aussi au niveau binaire. Pour que cela soit possible le source Java est compilé à un niveau intermédiaire appelé *bytecode* qui est « interprété »à la volée par une machine virtuelle Java (JVM). Voici les principales caractéristiques que nous détaillons ensuite qui rendent ce langage de programmation particulièrement attrayant pour le développement de notre EE et des services dans lequel ils seront exécutés au niveau espace utilisateur. Le langage Java et son environnement d'exécution fournissent :

- une indépendance de plate-forme ;
- un mécanisme de gestion automatique de la mémoire (appelé *Garbage Collector* (GC)) ;
- un langage Orienté Objet (OO), beaucoup plus concis et simple que le C++ ;
- une syntaxe à la C très confortable ;

[1]Mais c'est BILL JOY qui eu le premier l'idée de concevoir un tel langage en 1970.

- la possibilité d'interfacer du code Java avec du code C grâce aux *Java Native Interface*(JNI) ;
- un mécanisme de gestion des exceptions ;
- un mécanisme de chargement dynamique de classes ;
- un environnement multi-threadé intégré ;
- un mécanisme de programmation concurrente ;
- des mécanismes de sécurité ;
- et enfin, SUN fournit une documentation très complète.

Des implémentations de Java sont disponibles pour quasiment tous les systèmes d'exploitation UNIX et Microsoft ainsi que Apple. Tous les programmes 100% Java, ou dit *pure-Java* sont exécutables sur toutes les plateformes sur lesquelles ont été porté la machine virtuelle Java (JVM). Bien que la portabilité soit excellente, elle est parfois limité par des implémentations de bibliothèques incomplètes, notamment entre les premières et secondes versions[2]. Il existe une version open source de Java appelé *Kaffe*[92]. Elle est fournit aujourd'hui avec la plupart des classes du kit de développement Java 1.1 (JDK 1.1) et une partie du JDK 1.2. Il existe également un frontal Java à GCC appelé GNU Compiler for Java (GCJ)[78]. Il permet de compiler du code Java soit en bytecode java soit en code natif. Il permet également de compiler du bytecode en code natif. Il est disponible avec des bibliothèques *open source* qui mettent en œuvre une grande partie du JDK 1.2, sauf en particulier les classe graphiques AWT et Swing.

A.1.1.1 Java et le développement

Java permettra de développer plus facilement de nouveaux services. Il a été conçu pour écrire facilement un code sans *bug*. Il a été montré que dans une application écrite en C on trouve environ un *bug* toute les 55 lignes en moyenne dont la moitié sont relatifs à des problèmes d'allocations mémoires [93]. Java offre un meilleur environnement de développement logiciel d'un point de vue génie logiciel qu'un langage comme le C car c'est un langage : facile à apprendre, le nombre d'instructions est limitées ; facile à lire et écrire ; il est OO donc facilement réutilisable ; il n'a pas de constructeur indéfini ou dépendant de l'architecture ; fait une vérification stricte des types de données ; n'utilise plus des pointeurs mais des références afin qu'un programme ne puisse accédé à une adresse mémoire arbitraire ; propose une détection de dépassement de limite (écriture hors des limites d'un tableau de données) et ; offre un Garbage Collector (GC). Tous ces points permettent un prototypage rapide, conduit à un code moins erroné et à un temps de développement plus rapide.

A.1.1.2 Java et la portabilité

La portabilité de Java est un avantage dominant pour le projet Tamanoir, car il offre la possibilité d'exécuter notre projet sur une très grande variétés de systèmes d'exploitations et architectures matérielles. Malgré les efforts de standardisation d'un langage comme le C, le portage d'une application écrite dans ce langage demande du temps ainsi qu'une expérience considérable.

[2]A partir de la version 1.2 on parle aussi de Java 2

A.1.1.3 Java et les réseaux

Java est un langage très orienté réseau. Il fournit un très bon support pour l'invocation de méthodes à distance (RMI), l'accès à des fichiers distants et l'accès à des bases de données. L'API réseau fournit par Java s'adapte bien au besoin de notre projet. Le langage C semble bien adapté pour ce genre de projets d'infrastructure qui ont besoin de hautes performances, mais le manque de standardisation de l'API réseau aurait rendu la portabilité du projet significativement plus complexe. Le même problème aurait été rencontré avec le C++. Des langages comme Tcl, Perl ou Python auraient probablement réduit le problème de portabilité mais au dépend des performances.

A.1.2 Java et la performance

Un programme non compilé, donc interprété, est logiquement plus lent à s'exécuter puisqu'il y a le coût supplémentaire inévitable des opérations d'interprétation qui consistent à transformer un langage universel, ici le *bytecode*, dans un langage machine spécifique à l'architecture matérielle sur laquelle l'application s'exécute. Ce coût supplémentaire est dû à la couche de la machine virtuelle qui fournit en échange une abstraction du matériel « sous-jacent ». De plus l'aspect dynamique de Java difficile à contrôler est également un problème. Ces raisons font qu'une application écrite en Java peut s'exécuter plus lentement que la même application écrite dans un langage de plus bas niveau, comme le C qui, lui aussi s'exécute généralement plus lentement qu'un programme écrit en assembleur. Ce ralentissement de performances était notable avec les premiers kits de développements (*Java Development Kit (JDK)* ou *Software Development Kit (SDK)*) et d'exécutions (*Runtime Environment (RE)*). Le bytecode produit par le compilateur *javac* n'était pas optimal et les machines virtuelles Java n'étaient pas optimisées c'est à dire qu'elle ne fournissait aucun mécanisme de compilation à la volée, appelée *Just-in- Time* (JIT) qui ralentit la première exécution d'une méthode, mais accélère considérablement les exécutions suivantes de cette même méthode. Aux avantages de Java énuméré ci-dessus il faut donc soustraire :
- le temps d'interprétation indissociable d'au moins la première exécution ;
- le Garbage Collector (GC) que l'on ne maîtrise pas et qui décide de son propre chef de s'exécuter parfois au moment le plus inopportun ;
- les contrôles de Threads (ou *thread monitor*) ;
- et les vérifications supplémentaires lors de l'exécution (*extra runtime checks*) comme le test de dépassements de capacité d'un tableau.

Malgré tout, rappelons que quel que soit le choix du langage de développement d'une application et le choix d'architecture matérielle sur laquelle l'application sera destinée à être exécutée, elle devra faire face à différentes ressources limitantes :

1. La vitesse et disponibilité du CPU ;
2. la taille de la mémoire du système et ;
3. la vitesse d'Entrée/Sortie (E/S) des disques et connections réseaux.

Quelle que soit la fréquence d'horloge auxquelles un CPU est connecté, l'usage d'un algorithme inefficace ou dit explosif produira un goulot d'étranglement. Le création de beaucoup de petits objets à la vie courte sont également coûteux en CPU dû aux temps de création et de destructions par le GC de ces objets. La création d'un grand nombre d'objets ou de grande taille peut conduire également à une consommation mémoire excessive. Le système d'exploitation est alors obligé de déplacer des zones mémoire (appelées pages) sur le disque local. On parle de *swapping*. Cela dégrade dramatiquement les performances des applications. Enfin, un processus en attente de données en provenance de disques ou de cartes d'interfaces réseaux se bloque jusqu'à l'obtention des données qu'il attend ou écoulement d'un délai de garde (*timeout*). Afin qu'une application entière ne se bloque pas, il est impératif de la structurer en plusieurs threads afin de recouvrir ces temps d'attentes et donc maximiser l'usage du CPU. En résumé, quel que soit le temps que l'on passera à affiner le code (*code tuning*) d'une application, les performances de celle-ci seront toujours limités à des contraintes matérielles. Par contre, l'expérience montre que souvent la première version d'une application, quel que soit le langage de développement, s'exécute plus lentement que prévu. Mais après la mise en place d'optimisations pas toujours triviales à trouver ou à mettre en place, les temps d'exécution sont bien meilleurs.

Lorsque l'on fait de la performance et de l'optimisation de code il est impératif de se munir d'outils qui permettent de mesurer le temps afin d'évaluer de manière expérimentale le gain (ou la perte) apporté par les dernières modifications faites au code. La manière la plus évidente pour mesurer le temps en Java est d'utiliser la méthode statique (et native) `currentTimeMillis()` de la classe `System`. Cette méthode renvoie un entier (de type `long`) qui représente une estampille (*timestamp*). A cette estampille on soustrait l'entier renvoyer par un second appel à `currentTimeMillis()` pour savoir quel temps s'est écoulé entre les deux appels. Il faut savoir que cette méthode de mesure n'est efficace que pour mesurer des laps de temps suffisamment long. Le temps d'exécution d'un appel à `currentTimeMillis()` est d'environ $\frac{1}{2}$ milliseconde. quelconque mesure incluant les deux appels devra être dans un intervalle supérieur à 100 ms pour être sur que le coût des appels à `currentTimeMillis()` est inférieur à 1% de la mesure totale. Les autres types de mesures doivent être spécifique au système. On pourra par exemple mesurer : Le temps CPU alloué à chaque processus, le nombre de processus en attente afin d'évaluer les contentions, et les débits ou latence en écriture et lecture sur les E/S (disques ou réseaux).

Il existe aujourd'hui trois types de technologies pour l'exécution d'une application. La première, très classique, consiste à compiler les sources de l'application avant de l'exécuter (*ahead of time compiler*). La seconde, dite juste-à-temps, consiste à compiler le source à la volée (*Just-in-Time ou JIT*). Enfin la troisième, dites adaptative, plus récente, ressemble beaucoup à la seconde sauf qu'elle effectue une analyse statistique pour savoir quel sont les parties du code intéressantes à compiler (on parle d'*adaptative compiler*). C'est une technologie provenant de SUN Microsystem mieux connu sous le nom *HotSpot*. Elle optimise donc l'exécution grâce à la compilation en code natif des zones du code les plus sollicitées.

A.1.2.1 Java et la compilation *Just-in-Time*

A chaque sortie d'une nouvelle version de Java des progrès sont réalisés à tous les niveaux du système d'exécutions : GC, code, manipulation d'objets et threads par la machine virtuelle, et optimisation du compilateur. La version de JVM minimal est la version 1.1.6. Elle inclus le minimum d'optimisations pour fournir une base d'expérimentation intéressante et utilisable. Elle fournit notamment un compilateur Just-in-Time désactivable au besoin. Les premières Machines Virtuelle (VM) étaient basiquement des interpréteurs de bytecode. Elles l'exécutaient après une phase de décodage qui représente un coût de temps pur n'ajoutant aucune autre fonctionnalité supplémentaire à l'application. Un compilateur JIT élimine la plupart de ce coût de temps en décodant juste une seule fois le bytecode. La première fois qu'une méthode est chargée, les instructions décodés sont convertit en code machine natif pour le CPU sur lequel l'application s'exécute. Ensuite, lors des prochaines invocations à cette méthode, aucun coût de temps supplémentaire dû au décodage ne sera nécessaire. Cependant le compilateur doit être rapide pour ne pas réduire la vitesse d'exécution. Il ne peut donc appliquer d'excellentes optimisations, comme un compilateur *ahead of time* multi passe pourrait le faire par exemple. Le code machine résultant n'est donc pas aussi rapide qu'il devrait l'être. Il est à savoir qu'une machine virtuelle JIT implique également une empreinte mémoire plus importante. Décrivons maintenant le temps de démarrage d'une application Java. Ce temps dépend :

- Premièrement du temps mis par le système d'exploitation pour démarrer le processus exécutable qui dépend de la taille du processus plus le nombre de bibliothèques partagées à charger. Une façon de réduire le temps de démarrage est d'avoir les bibliothèques partagées déjà copiées (*map*) en mémoire système. On constate de manière évidente ce gain lorsque l'on redémarre une application juste après l'avoir arrêté. Il est également possible d'utiliser, dans le cas d'application ou le temps de démarrage est vraiment critique, un système de fichier en mémoire vive (Ramdisk, tmpfs, cachefs).
- Deuxièmement, le démarrage de l'application dépend du type de JVM utilisé. Les machines virtuelles qui ne font que interpréter (ou avec l'option `-nojit`) démarre généralement plus rapidement que les VM qui doivent consommer un temps supplémentaire a compilé durant la phase de chargement initial des classes.
- Troisièmement, c'est l'architecture de l'application et la configuration des fichiers de classes qui détermine la dernière composante du temps de démarrage. L'application peut avoir à charger beaucoup de fichiers de classes et réaliser de nombreuses initialisations avant que l'application soit disponible pour l'utilisateur.

A.1.2.2 Java et les méthodes natives

Un programme Java peut par l'intermédiaire de l'interface JNI (*Java Native Interface*) appeler des fonctions écrites en C. Si c'est dans l'objectif de gagner en performance on devra être très prudent. L'interface elle même introduit un coût de temps supplémentaire (*overhead*) ce qui signifie que souvent, les performances obtenues avec du code 100% Java seront très proche d'un code Java qui fait appel à du code natif. Cependant, parfois le système sous jacent fournit un appel système indisponible à partir de Java et ne peut être mise en œuvre pour fonctionner aussi rapidement en Java. Dans ce cas JNI est utile pour affiner les performances de son code. Pour faire appel à des fonctions natives, il suffit d'écrire les routines en C et de les « brancher »dans le code Java à l'aide du mot clef `native`. Lorsque l'on fait appel à la routine native dans une boucle on prendra soin de la descendre dans la routine native elle même. L'usage de JNI implique de perdre la certification 100% pure java et ajoute un peu de complexité au développement et à la procédure de déploiement de l'application. Finalement, si c'est juste pour des raisons d'optimisations, il est conseillé d'essayer d'optimiser le code Java avant d'envisager JNI.

A.1.2.3 Java compilé

Lorsque l'on connaît l'environnement d'exécution (système d'exploitation et architecture matériel) cible d'une application on peut compiler son code Java directement en code machine exécutable. Aujourd'hui des compilateurs existent pour différentes plateformes (GCJ[78], TowerJ[94], Jove[95],...). Ces compilateurs fonctionnent souvent directement sur le bytecode sans le code source, donc n'importe quelle classe dont nous n'avons pas les sources peuvent être incluses. La difficulté est ensuite de pouvoir rester multi plateforme. Pour cela on pourra utiliser un fichier *batch*, qui devra lui aussi supporter de multiples plateformes (écrit en Java ? :-). Malheureusement, l'expérience montre que les performances obtenues sont décevantes avec des exécutables compilés en code natif comparé aux performances obtenues avec les dernières JVM ($<$ à 2 ans) donc livré avec la technologie JIT. Il s'avère que l'exécutable compilé doit encore gérer le GC, les threads, les exceptions,... au sein même de l'exécutable. De plus les performances dépendront énormément de la qualité du compilateur à produire un code natif efficace pour une architecture donnée et de gérer les caractéristiques propre à Java. Dans [96] de nombreuses expérimentations ont été menées sur différentes architectures, compilateurs et JVM. Les mesures ont été effectuées sur des applications scientifiques haute performance courantes (FFT, tri, factorisation,...) qui ont été écrites en Java, C et Fortran. Ils montrent que les JVM adaptative et JIT récentes pouvaient être extrêmement performantes sur des architectures courantes (type Pentium) et rivaliser avec le C et le Fortran. De manière plus anecdotique, il existe aussi des traducteurs de source Java en C. Nous n'en n'avons pas essayé mais en général les traducteurs automatiques d'un code source en un autre code source ne génère pas du code très bon.

La plupart des compilateurs comme GCC offre de nombreuses options de compilations pour l'optimisation. Certaines options sont destinées à améliorer la vitesse d'exécution, d'autre pour minimiser la taille. Il y a souvent un compromis entre ces deux aspects. Le compilateur `javac` n'offre qu'une seule option standard au moment de la compilation (*compile time option*). L'option `-O` (O pour *Optimize*). Cette option applique en fait deux types d'optimisations :

- Elle élimine les tables optionnelles dans les fichiers classes tels que les numéros de lignes du source Java et les tables de variables locales. Cela améliore la vitesse de chargement des fichiers de classes étant donné qu'ils sont plus petits. Cette réduction de taille est également profitable lorsque ces classes sont transférées sur un réseau.
- Elle force l'*inlining* des méthodes ayant les modificateurs suivants : `private`, `static`, et `final`. En fait, seule les méthodes très simples (à une ligne) sans effet de bord comme les accesseurs, ou qui retournent une constante sont dupliqués. Le fait que tout le code ne soit pas *inlinable* permet de ne pas réduire les performances.

Il est conseillé de fournir les fichiers de classes d'une application dans un fichier `.jar` unique. Un fichier `.jar` est une archive compressée, ou non, équivalent à un fichier `.zip` ou `.tgz` (archive compressée), ou `.tar` (archive non compressée). Charger les fichiers de classes les uns à la suite des autres à travers le réseau ou bien même, les chargés individuellement à partir du système de fichiers local est coûteux. Les mettre dans un fichier unique réduit le coût d'E/S dû aux nombreux appels systèmes d'ouverture et fermeture répétés des fichiers, et améliore le temps d'accès. Lorsque beaucoup de fichiers sont fermés au même moment cela provoque un vidage des tampons disques (*disk cache flush*) ce qui a pour effet de bloquer sensiblement le système temporairement. En utilisant un seul fichier on évite ce problème de performance potentiel. La meilleur façon de délivrer de manière performante des classes locales, c'est à dire présente sur le disque de l'hôte de l'application, est d'utiliser un fichier unique `.zip` ou `.jar` non compressé[3]. Pour améliorer encore les performances de chargement, de manière subtil, on peut stocker et ordonner les classes (*class packing*) dans l'ordre dans lequel elles sont chargées par l'application. Afin de déterminer l'ordre de chargement on peut utiliser l'option `-verbose`

[3]C'est de cette façon que les classes du JDK sont livrées

de la commande `java` lors d'une exécution préliminaire de l'application. Remarquez que le moindre petit changement dans l'application peut facilement altérer l'ordre de chargement.

A.1.3 Java et l'extension dynamique pour l'ajout de nouveaux services

L'extension dynamique est la caractéristique qui rend la mise en œuvre d'un projet comme Tamanoir possible. L'architecture de Java permet d'écrire des programmes qui s'étendent dynamiquement lors de l'exécution. Un programme Java peut s'étendre dynamiquement en choisissant des classes à charger et à utiliser lors de l'exécution. Cela signifie que lors de la phase de la compilation nous n'avons pas besoin de connaître nécessairement toute les classes que le programme utilisera lors de l'exécution. Ces classes peuvent même ne pas avoir encore été écrites lorsque l'on compile l'application. Le langage C, par exemple, fournit aussi une interface de programmation pour charger en mémoire et lier dynamiquement une bibliothèque. Ce sont les fonctions `dl` (dlopen(), dlsym(),...) [97]. Grâce à ce mécanisme on peut imaginer des bouts de code que l'application charge dynamiquement comme des *« plug-ins »* ou des modules personnalisés qui permettraient à l'utilisateur d'intégrer ses propres algorithmes, filtres ou fonctionnalités. Les applets sont par exemple une extension dynamique d'une application Java exécutée dans un navigateur Web. Elles permettent aux programmeurs d'Applets de personnaliser le navigateur en le transformant par exemple en calculatrice, plateforme de jeu,...Les *servlets* permettent sur le même principe de fournir des extensions dynamiques cette fois-ci à un serveur Web. En fait, tout les fichiers de classes ne sont pas chargés en une seule fois mais à la demande c'est à dire lorsqu'ils sont requis par l'application. Cette opération est réalisée grâce au `ClassLoader`. Le `ClassLoader` fait partie de la JVM et charge les classes en mémoire. Ce chargeur de classes a de plus été écrit en Java, cela signifie que l'on peut étendre cette classe pour créer son propre `ClassLoader`. Celui fournit par défaut ne permet de charger des classes qu'à partir du système de fichier local, ce qui suffit lorsque toutes les classes de l'application sont déjà présente sur le système local. Une innovation du langage Java est qu'il facilite la récupération des classes situées ailleurs que sur le disque local grâce à une connexion réseau. Par exemple, les navigateurs web utilisent un `ClassLoader` personnalisé pour charger des contenus exécutables à partir de site web. Mais on peut imaginer un chargeur de classes personnalisé qui décrypte de manière transparente des codes avec un mot de passe fournit par l'utilisateur, vérifie une signature numérique avant d'exécuter du code dans lequel nous n'avons pas encore établie de relation de confiance,... Java propose deux méthodes pour localiser et charger dynamiquement un objet `java.lang.Class` qui correspond à un nom de classe donné. Ces méthodes sont `Class.forName()` et `ClassLoader.loadClass()`. Leurs comportements diffèrent suivant le `java.lang.ClassLoader` qu'ils utilisent pour le chargement de classes et si l'objet `Class` résultant est ou n'est pas initialisé. La méthode la plus couramment utilisée est `Class.forName(String)` avec un seul paramètre qui est le nom de la classe à charger (sans l'extension `.class`). C'est une méthode `static` de `java.lang.Class` qui propose une façon simple et direct pour faire de l'extension dynamique. Elle utilise le même chargeur de classes qui à chargé le code exécutant la méthode `forName()`. L'autre méthode `ClassLoader.loadClass()`, plus complexe à utiliser mais aussi plus puissante est une méthode instanciée et oblige le programmeur de l'application à sélectionner un chargeur de classes particulier. Ce sera probablement celui qu'il aura écrit mais cela peut être aussi plus simplement le chargeur qui a chargé le code appelant. Depuis la version 1.2 (Java 2) on peut utiliser la méthode `forName(String, Boolean, ClassLoader)` à trois paramètres dont le troisième paramètre permet de préciser le chargeur de classe à utiliser. La classe `forName(String)` classique (à un seul paramètre) initialise la classe chargé. C'est à dire que cela exécute les initialiseurs `static` de classes ainsi que le bytecode des expressions d'initialisations de tous les champs `static` et de manière récursive pour toutes les super classes d'une classe. Contrairement à la méthode `ClassLoader.loadClass()` qui repousse l'initialisation jusqu'à ce que la classe soit utilisée pour la première fois.

```
// fichier MonAppli.java
public class MonAppli
{
    public static void main(String [] args) throws
        ClassNotFoundException, InstantiationException,
        IllegalAccessException
    {
        Class cl = Class.forName(args[0]);
        LangDefault ld = (LangDefault) cl.newInstance();
        ld.salutation();
    }
}

// fichier LangDefault
public class LangDefault
{
    public void salutation()
    {
        System.out.println("Hello world !");
    }
}

// fichier LangFrench
public class LangFrench extends LangDefault
{
    public void salutation()
    {
        System.out.println("Bonjour monde !");
    }
}
```

FIG. A.1 – Chargement dynamique d'une classe avec `forName()`

La figure A.1 donne un exemple de chargement dynamique de classe avec la méthode `forName()`. Imaginons que nous ayons appelé la classe `MonAppli` avec pour argument `LangFrench`. Cette chaîne sera passée en argument à la méthode `forName()` qui tentera alors de charger ce type. Lorsque la classe `MonAppli` invoque la méthode `salutation()` avec la référence `ld` de type `LangDefault`, la JVM réalise une liaison dynamique et exécute l'implémentation de `salutation()` de `LangFrench`. L'application affichera alors : "Bonjour monde!". Cela reste vrai même si `MonAppli` ne connaît pas `LangFrench` au moment de la compilation. Au moment de la compilation `MonAppli` ne connaît que `LangDefault`. C'est la manière la plus courante pour interagir avec une classe chargée dynamiquement : invoquer des méthodes défini dans une super classe de la classe chargée dynamiquement sur une instance de cette classe.

A.1.4 Conclusions sur Java pour la création d'un environnement actif haute performance

Les caractéristiques du langage Java telles que : la simplicité de sa syntaxe, son excellente portabilité, son mécanisme de chargement dynamique de classe et sa bibliothèque très riche font de lui un excellent candidat pour la mise en œuvre d'un nœud actif générique.

Les progrès réalisés ces trois dernières années dans le domaine de la conception de machines virtuelles Java efficaces autorisent aujourd'hui l'écriture d'applications performantes. Et si parfois le langage Java fournit une abstraction trop importante du système, l'interface JNI autorise la construction d'applications bas niveau. Bref, Java constitue un sérieux candidat pour la mise en œuvre d'un nœud actif dans un contexte haute performance.

Publications personnelles

Chapitre de livre :

– Laurent Lefèvre and Jean-Patrick Gelas. Chapter 14, "High Performance Execution Environments" of the "Programmable Networks and their Management" book - ISBN 1-58053-745-6 ; à paraître fin 2003, Artech House Books, *État de l'art des travaux de recherche dans le domaine des réseaux actifs haute performance et proposition d'un modèle de nœud actif haute performance validé par des résultats d'expérimentations sur différentes plateformes de test.*

Revue :

– Jean-Patrick Gelas, Saâd El Hadri and Laurent Lefèvre. Towards the Design of an High Performance Active Node to be published in Vol. 13, No. 2, pp. 149-167 issue of the Parallel Processing Letters journal (PPL), June 2003 *Reprend toutes les idées de conceptions permettant la création d'un nœud actif haute performance.*

Publications internationales :

2003

– Alex Galis,Jean-Patrick Gelas, Laurent Lefèvre, Kun Yang. Active Network Approach to Grid Management & Services in Workshop on Innovative Solutions for Grid Computing - ICCS 2003 Conference, LNCS 2658, ISBN 3-540-40195-4, pages 1103-1113, Melbourne, Australia, June 02-04, 2003 *Présentation de l'usage possible des réseaux actifs pour aidé à la gestion d'une grille de calculs.*
– Jean-Patrick Gelas, Saâd El Hadri, Laurent Lefèvre. Tamanoir : a software active node supporting gigabit networks in ANTA 2003 : The second International Workshop on Active Networks Technologies and Applications, pages 159-168, Osaka, Japan, 28-30th May 2003 *Présentation des résultats au Gigabit et les solutions archictecturale adopté pour obtenir de tel résultats.*
– Laurent Lefèvre and Jean-Patrick Gelas. Active Web : active networking support for web transport in ANTA 2003 : The second International Workshop on Active Networks Technologies and Applications, pages 147-156, Osaka, Japan, 28-30th May 2003 *Présentation des apports de la technologie active pour le transport efficace de pages web.*

2002

– L. Brunie, L. Favory, JP. Gelas, L. Lefèvre, A. Mostefaoui, and F. Nait-Abdesselam. Sirsale : Integrated video databases management tools. In T. Zhang J.R. Smith, S. Panchanathan, editor, ITCOM2002 : Multimedia Networks and Management Systems, volume 4862, Boston, August 2002. SPIE. ISBN : 0-8194-4641-6. *Proposition d'un en-*

vironnement d'exécution actif uniquement dans le cas d'une application de distribution de vidéo.

- A. Bassi, JP. Gelas, L. Lefèvre, Tamanoir-IBP : Adding Storage to Active Networks, AMS 2002, pages 27-34, Edinburgh, Scotland, July 2002. IEEE computer society. ISBN : 0-7695-1721-8 *Proposition de plusieurs scénarios où un service Tamanoir utilise un dépôt IBP pour réaliser du stockage dans le réseau*
- A. Bassi, JP. Gelas, L. Lefèvre, Logistical Storage in Active Networking : a promising framework for network services, IC 2002, Las Vegas, NV, USA, 24-27 June 2002 *Utilisation de dépôt de données IBP pour faire du stockage intelligent de données dans le réseau grâce à des services actifs Tamanoir.*
- JP. Gelas, J. Guilloux, L. Lefèvre, Using OS filtering capabilities for the improvement of software active routers, PDPTA 2002, Las Vegas, NV, USA, 24-27 June 2002 *Proposition d'exécuter des services actifs dans le noyau grâce à la boite à outil Netfilter fournit dans Linux 2.4.x*
- L. Lefèvre, JP. Gelas, Towards the design of an Active Grid, Computational Science - ICCS 2002, Lecture Notes in Computer Science, vol. 2230, pp.578-587, Amsterdam, The Netherlands, April 2002, ISBN 3-540-43593-X *Seconde publications sur l'apport des réseaux actifs pour le support d'applications de grilles.*

2001

- L. Lefèvre, C. Pham, P. Primet, B. Tourancheau, B. Gaidioz, JP. Gelas, and M. Maïmour. Active networking support for the grid, IFIP-TC6 Third International Working Conference on Active Networks, IWAN 2001, pages 16-33 , Ian W. Marshall, Scott Nettles, Noaki Wakamiya, vol.2207, Lecture Notes in Computer Science, ISBN : 3-540-42678-7, October 2001 *Publication fondatrice sur l'apport de la technologie réseau actif pour la grille de calculs.*
- JP. Gelas, L. Lefèvre, Mixing High Performance and Portability for the design of Active Network Framework with Java, 3rd International Workshop on Java for Parallel and Distributed Computing, International Parallel and Distributed Processing Symposium (IPDPS 2001), San Fransisco, USA, April 2001 *Cette publication met en avant l'aspect performance de Tamanoir écrit en Java. Elle fournit également une comparaison avec ANTS.*

2000

- JP. Gelas and Laurent Lefèvre. Tamanoir : A high performance active network framework. In C. S. Raghavendra S. Hariri, C. A. Lee, editor, Active Middleware Services, Ninth IEEE International Symposium on High Performance Distributed Computing, pages 105-114, Pittsburgh, Pennsylvania, USA, August 2000. Kluwer Academic Publishers. ISBN 0-7923-7973-X. *Première publication sur le projet Tamanoir.*

Publication nationale :

- JP. Gelas, L. Lefèvre, Performance et dynamicité dans les réseaux : l'approche Tamanoir, JDIR 2002, Toulouse, France, 4-6 Mars 2002 *Reprend toutes les grandes idées de conception ayant conduit à la réalisation du prototype de réseau actif Tamanoir.*

Rapports de recherche :

- JP. Gelas, First Tamanoir-IBP Interaction , Internship report, LoCI lab, University of

Tennessee, Knoxville, TN, USA, July 2001 *Rapport de stage de 3 semaines. Etude des interactions possibles entre un nœuds actif Tamanoir et un dépôt IBP. Description du prototype.*

- JP. Gelas, L. Lefèvre, LIP RR2003-18/ Performance et dynamicité dans les réseaux : l'approche Tamanoir. Mars 2003. *Rapport en français qui résume toutes les idées mis en œuvre dans le projet Tamanoir.*

Logiciel :

- Environnement d'exécution actif haute performance Tamanoir 1.0, déposé à l'APP (Agence de Protection des Programmes) le 01/02/2002 (environ 6000 lignes)

Glossaire

AMS Active Middleware Services
ANANA Active Network Assigned Number Authority
ANEP Active Network Encapsulation Protocol
ANON Active Network Overlay Network
ANTA Active Network Technologies and Applications
ANTS Active Node Transfer System
APE Active Packet Edition
APF Active Packet Format
API Application Program(ming) Interface
APP Agence de Protection des Programmes
ARP Address Resolution Protocol
ASIC Application-Specific Integrated Circuits
ASP Application Service Provider
ATM Asynchronous Transfer Mode
BE Back End
B Bytes
CAM Content Addressable Memory
CANES Composable Active Network Elements
CNI Cygnus Native Interface
CPU Central Processing Unit
DNS Domain Name System
DOSMOS Distributed Objects Shared MemOry System
DRAM Dynamic RAM
DR Direct Routing
DyRAM Dynamic Replier Active reliable Multicast
EE Execution Environment
FEC Forward Error Correction
FE Front End
FPGA Field Programmable Gate Array

FPTP Fully Programmable Transport Protocol

GCJ GNU Compiler for Java

Gbps Giga bits per second

IBP Internet Backplane Protocol

ICMP Internet Control Message Protocol

INRIA Institut National de Recherche en Informatique et en Automatique

IP Internet Protocol

ISP Internet Services Provider

IWAN International Working Conference on Active Networks

Internet International Network

J2RE Java 2 Runtime Environment

JANOS Java-oriented Active Network Operating System

JDK Java Development Kit

JIT Just in Time

JNI Java Native Interface

JVM Java Virtual Machine

KB Kilo Bytes

LAAS Laboratory for Analysis and Architecture of Systems

LAN Local Area Network

LIP Laboratoire de l'Informatique du Parallélisme

LVS Linux Virtual Server

MAC Media Access Control

MB Mega Bytes

MIB Management Information Base

MIT Massachusetts Institute of Technology

MPLS MultiProtocol Label Switching

MTU Maximum Transmission Unit

Mbps Mega bits per second

NACK Negative Acknowlegement

NAT Network Address Translation

NF Netfilter

NP Network Processor

OSI Open System Interconnexion

PAN Practical Active Network

QoS Quality of Service

RESO Réseaux haut débit, protocoles et services

RFC Request For Comments

RJ Registred Jack
RSVP ReSerVation Protocol
RTP Real Time Protocol
SAN System Area Network
SAPF Simple Active Packet Format
SDK Software Development Kit
SIMD Single Instruction Multiple Data
SMP Symetrical Multi Processing
SMR Service Module Repositories
TAN Tamanoir Active Node
TCP Transport Control Protocol
TLV Type, Length, Value
TUN Tunneling
UCBL Université Claude Bernard - Lyon I
UCLA University of California - Los Angeles
UDP User Datagram Protocol
VLSI Very Large Scale Integration
VPN Virtual Private Network
VTHD Vraiment Très Haut Débit
VoD Video on Demand
WAN Wide Area Network

Bibliographie

[1] K. Calvert. Architectural framework for active networks. Technical report, gatech, jul 1999. www.cc.gatech.edu/projects/canes/papers/arch-1-0.ps.gz.

[2] Force10. http ://www.force10networks.com. Developer of scalable high performance Ethernet systems.

[3] Gary Rivlin. The madness of king george. Wired, jul 2002. http ://www.wired.com/wired/archive/10.07/gilder_pr.html.

[4] Steve Bell. Testing switch performance : A primer. http ://www.nwfusion.com/reviews/1019bg.html, oct 1998.

[5] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. RFC 1918 : Address allocation for private internets, February 1996.

[6] Olivier Festor, Isabelle Chrisment, and Eric Fleury. Les réseaux programmables 1.0. Technical Report 3913, INRIA, mar 2000. Rapport de Recherche.

[7] DARPA homepage. Active network. http ://www.darpa.mil/ato/programs/activenetworks/actnet.htm.

[8] J. Biswas, J.F Huard, A. Lazar, K. Lim, S. Mahjoub, L.F Pau, M. Suzuki, S. Torstensson, W. Weiugo, and S. Weinstein. Application programming interfaces for networks. *IEEE P1520 WG Draft White Paper*, jan 1999.

[9] David Wetherall and David Tennenhouse. The active ip option. Technical report, MIT, sept 1996. ACM SIGOPS European Workshop.

[10] Beverly Schwartz, Alden W. Jackson, W. Timothy Strayer, Wenyi Zhou, Dennis Rockwell, and Craig Partridge. Smart packets for active networks. In *IEEE OPENARCH 99*, mar 1998.

[11] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. Plan : A packet language for active networks. In *ICFP'98*, 1998.

[12] David Wetherall, John Guttag, and David Tennenhouse. Ants : A toolkit for building and dynamically deploying network protocols. In *IEEE OPENARCH '98*, April 1998.

[13] S.Bhattacharjee, K. Calvert, and E. Zegura. An architecture for active networking. In *INFOCOMM 97*, apr 1997.

[14] Dan Decasper and Berhard Plattner. Dan : Distributed code caching for active networks. In *INFOCOMM 98*, apr 1998.

[15] David Wetherall. Active network vision and reality : lessons from a capsule-based system. *Operating Systems Review*, 34(5) :64–79, December 1999.

[16] S. Merugu, S. Bhattacharjee, Y. Chae, M. Sanders, K. Calvert, and E. Zegura. Bowman and canes : Implementation of an active network. In *37th Annual Allerton Conference, Monticello, IL*, September 1999.

[17] S. Merugu, S. Bhattacharjee, E. Zegura, and K. Calvert. Bowman : A node os for active networks. In *IEEE INFOCOM '2000*, mar 2000.

[18] Mosberger D. *Scout : A path based operating system*. PhD thesis, University of Arizona, 1997. http ://www.cs.arizona.edu/scout.

[19] Hartman J., Bigot P., Bridges P., Montz B., Piltz R., Spatscheck O., Proebsting T., and Peterson L. Joust : A platform for liquid software. *IEEE computer*, 1999.

[20] Engler D. *The Exokernel Operating System Architecture*. PhD thesis, MIT, oct 1998.

[21] Hrishikesh Dandekar, Andrew Purtell, and Stephen Schwab. Amp : Experiences with building an exokernel-based platform for active networking. In *DANCE'02*, page 77. IEEE, may 2002. San Francisco,CA.

[22] Patrick Tullmann, Mike Hibler, and Jay Lepreau. Janos : A java-oriented os for active networks. *Communications*, march 2001. IEEE.

[23] Erik L.Nygren, Stephen J.Garland, and M.Frans Kaashoek. PAN : A High-Performance Active Network Node Supporting Multiple Mobile Code Systems. In *IEEE OPENARCH '99*, March 1999.

[24] D. Decasper, G. Parulkar, S. Choi, J. DeHart, T. Wolf, and B. Plattner. A scalable, high performance active network node. In *IEEE Network*, volume 13, January 1999.

[25] D. Larrabeiti, M. Calderón, A. Azcorra, and M. Urueña. A practical approach to network based processing. In *AMS*, Edinburgh, Scotland, jul 2002. http ://matrix.it.uc3m.es/ sara.

[26] David Larrabeiti, Maria Calderón, Arturo Azcorra, and Manuel Urueña. Sara : a simple active router-assistant architecture. Technical report, Univ Carlos III, Madrid, nov 2001.

[27] Max Ott, Girish Welling, and Saurabh Mathur. Clara : A cluster based active router architecture. In *In Proceedings of the Hot Interconnects VIII*, August 2000. Stanford University, CA.

[28] Noriyuki Takahashi, Toshiaki Miyazaki, and Takahiro Murooka. Ape : Fast and secure active networking architecture for active packet editing. In *IEEE OPENARCH 2002*. NTT Network Innovation Laboratories, 2002.

[29] I. Hedzic and J.M. Smith. P4 : A platform for fpga implementation of protocol boosters. In LNCS, editor, *FPL'97*, volume 1304, pages 438–447, sep 1997. http ://www.cis.upenn.edu/ boosters/.

[30] Bhattacharjee S., Calvert K., and Zegura E. An architecture for active networking. In *HPN'97*, White Plains, NY, apr 1997.

[31] R. Keller, L. Ruf, A. Guindehi, and B. Plattner. Promethos : A dynamically extensible router architecture supporting explicit routing. In *Fourth annual Internatioanl Working Conference on Active Networking*, Zürich, dec 2002. IWAN'02.

[32] Wijata Y. Ressource management in active networks. Technical report, U.Kansas, may 1997.

[33] Thomas Fuhrmann, Till Harbaum, Marcus Scholler, and Martina Zitterbart. Amnet 2.0 : An improved architecture for programmable networks. In *IWAN'02*, Zurich, Switzerland, dec 2002.

[34] Thomas Fuhrmann, Till Harbaum, Panos Kassianidis, Marcus Scholler, and Martina Zitterbart. Results on the practical feasibility of programmable network services. In *ANTA 2003*, page 141, Osaka, Japan, may 2003.

[35] Jonathan T. Moore, Michael Hicks, and Scott M. Nettles. Practical programmable packets. In *INFOCOM'01*. IEEE, apr 2001. the 20th Annual Joint Conference of the IEEE Computer and Communications Societies.

[36] Tilman Wolf and Jonathan S. Turner. Design issues for high performance active routers. *IEEE Journal on Selected Areas of Communication*, 19(3) :404–409, March 2001.

[37] 6wind. http ://www.6wind.com.

[38] Projet android. http ://www.cs.ucl.ac.uk/research/android.

[39] Yacine El Mghazli and Olivier Marce. Cops client-type for active networks. Internet Draft, Experimental, dec 2000. http ://www.watersprings.org/pub/id/draft-yacine-cops-an-00.txt.

[40] Stephen F. Bush. Active network initiative in general electrc. http ://www.crd.ge.com/ bushsf/AVNMP.html.

[41] Stephen F. Bush and Amit Kulkarni. *Active Networks and Active Network Management : A Proactive Management Framework*. Plenum Publishers, kluwer academic edition, 2001. 196 pp. Hardbound, ISBN 0-306-46560-4.

[42] M. Brunner, B. Plattner, and R. Stadler. Service creation and management in active telecom environments. *Communications of the ACM*, mar 2001. http ://www.brubers.org/marcus/sman.html.

[43] Active network initiative in hitachi with fokus gmd. http ://www.fokus.gmd.de/news/hitachi.

[44] Olivier Marcé. Towards an industrial active ip network. In *IWAN'01*, 2001.

[45] Lionel Brunie, Laurent Lefèvre, and Olivier Reymann. DOSMOS+ : A scalable Distributed Shared Memory environment including monitoring facilities. In *Parallel Programming Environments for High Performance Computing*, pages 165–168, Alpe d'Huez, France, April 1996.

[46] Virginie Galtier, Kevin L. Mills, Yannick Carlinet, Stefan Leigh, and Andrew Rukhin. Expressing meaningful processing requirements among heterogeneous nodes in an active network. In *Workshop on Software and Performance*, pages 20–28, 2000.

[47] Scott D. Alexander, Bob Braden, Carl A. Gunter, Alden W. Jackson, Angelos D. Keromytis, Gary J. Minden, and David Wetherall. Active network encapsulation protocol (anep). RFC Draft, Category : Experimental, July 1997.

[48] Active Network Assigned Number Authority. Anana. http ://www.isi.edu/ braden/anana/.

[49] Tilman Wolf and Dan Decasper. Tags for high performance active networks. In *OpenArch2000, Tel Aviv, March 2000*, 2000.

[50] Bob Braden, Michael Hicks, and Christian F. Tschudin. Active network overlay network (anon), dec 1997. RFC.

[51] Marie-Pierre Gervais. Composition of active services : A methodological approach. In *Fourth Annual International Working Conference on Active Networks*, Zürich, dec 2002. IWAN'02.

[52] Jean-Patrick Gelas and Laurent Lefèvre. Tamanoir : A high performance active network framework. In C. S. Raghavendra S. Hariri, C. A. Lee, editor, *Active Middleware Services,*

Ninth IEEE International Symposium on High Performance Distributed Computing, pages 105–114, Pittsburgh, Pennsylvania, USA, August 2000. Kluwer Academic Publishers. ISBN 0-7923-7973-X.

[53] Jean-Patrick Gelas and Laurent Lefèvre. Mixing high performance and portability for the design of active network framework with java. In *3rd International Workshop on Java for Parallel and Distributed Computing, International Parallel and Distributed Processing Symposium (IPDPS 2001)*, San Fransisco, CA, USA, April 2001.

[54] Jean-Patrick Gelas and Laurent Lefèvre. Performance et dynamicité dans les réseaux : l'approche tamanoir. In *JDIR 2002*, pages 81–90, Toulouse, FRANCE, March 2002. in french.

[55] Laurent Lefèvre and Jean-Patrick Gelas. Active web : active networking support for web transport. In *The second International Workshop on Active Networks Technologies and Applications*, pages 147–156, Osaka, Japan, May 2003. ANTA 2003.

[56] D. Katz. RFC 2113 : IP router alert option, February 1997. Status : PROPOSED STANDARD.

[57] C. Partridge and A. Jackson. RFC 2711 : Ipv6 router alert option, October 1999. Status : PROPOSED STANDARD.

[58] http server project. Apache. http ://httpd.apache.org/.

[59] J. Gosling and B. Joy. The java language specification, 1996.

[60] A. Lindholm and T. Yellin. The java virtual machine specification, 1996.

[61] K. Arnold and J. Gosling. The java programming language 2nd ed, 1998.

[62] Wensong Zhang. Linux Virtual Server for Scalable Network Services. In *Ottawa Linux Symposium*, 2000. http ://www.linuxvirtualserver.org.

[63] Jean-Patrick Gelas. First tamanoir-ibp interaction. Technical report, Internship report, University of Tennessee, Knoxville, TN, USA, July 2001.

[64] Alessandro Bassi, Micah Beck, Jean-Patrick Gelas, and Laurent Lefèvre. Logistical storage in active networking : a promising framework for network services. In Hamid R.Arabnia and Youngsong Mun, editors, *IC 2002*, volume 2, Las Vegas, Nevada, USA, June 2002. ISBN : 1-892512-36-x.

[65] Alessandro Bassi, Jean-Patrick Gelas, and Laurent Lefèvre. Tamanoir-ibp : Adding storage to active networks. In *Active Middleware Services*, pages 27–34, Edinburgh, Scotland, July 2002. IEEE computer society. ISBN : 0-7695-1721-8.

[66] J. Plank, A. Bassi, M. Beck, T. Moore, M. Swany, and R. Wolski. Managing data storage in the network. *IEEE Internet Computing*, 5(5), september 2001.

[67] Wael Elwasif, James Plank, Micah Beck, and Rich Wolski. Ibp-mail : Controlled delivery of large mail files. In *NetStore'99 : Network Storage Symposium*, Seattle, WA, October 1999.

[68] M. Swany. Network proximity resolution with sonar and sonardns. Technical Report CS-99-429, University of Tennessee, January 1999.

[69] L. Lefèvre, C. Pham, P. Primet, B. Tourancheau, B. Gaidioz, J.P. Gelas, and M. Maimour. Active networking support for the grid. In Noaki Wakamiya Ian W. Marshall, Scott Nettles, editor, *IFIP-TC6 Third International Working Conference on Active Networks, IWAN 2001*, volume 2207 of *Lecture Notes in Computer Science*, pages 16–33, October 2001. ISBN : 3-540-42678-7.

[70] Laurent Lefèvre and Jean-Patrick Gelas. Towards the design of an active grid. In Lecture Notes in Computer Science, editor, *Computational Science - ICCS 2002*, volume 2230, pages 578–587, Amsterdam, The Netherlands, April 2002. ISBN 3-540-43593-X.

[71] Alex Galis, Jean-Patrick Gelas, Laurent Lefèvre, and Kun Yang. Active network approach to grid management. In *ICCS*, pages 1103–1113, Melbourne, Australia, June 2003. ICCS. LNCS 2658, ISBN 3-540-40195-4.

[72] F. Bouahfs, B. Gaidioz, J.P. Gelas, L.Lefèvre, M. Maimour, C. Pham, P. Primet, and B. Tourancheau. Designing and experimenting an active grid architecture. *Future Generation Computer Systems (FGCS), Special issue on : Advanced Grid Technologies*, 2003.

[73] Franck Bonnassieux, Robert Harakaly, and Pascale Primet. Mapcenter : an open grid status visualization tool. In *ISCA 15th International Conference on parallel and distributed computing systems*, Louisville, Kentucky, USA, sep 2002. http ://mapcenter.in2p3.fr.

[74] Java Sun Microsystem. langage java. http ://java.sun.com.

[75] Java IBM. langage java. http ://www-106.ibm.com/developerworks/java/.

[76] Kaffe. machine virtuelle java sous licence gnu. http ://www.kaffe.org/.

[77] Java technology on linux. Blackdown JVM. http ://www.blackdown.org/.

[78] GCJ. The gnu compiler for the java programming language. http ://sourceware.cygnus.com/java/.

[79] Hotspot technology. http ://java.sun.com/j2se/1.4/.

[80] Pascale Vicat-Blanc Primet, Fabien Chanussot, Christophe Blanchet, Nicolas Lacorne, and Philippe d'Anfray. E-toile : High performance grid middleware. In *IEEE International Cluster Conference. Grid Demo session*, December 2003.

[81] P. Primet and F. Chanussot. Spécification du service actif qosinus. Technical Report RR-0287, INRIA, 2003.

[82] Moufida Maimour and Cong-Duc Pham. Dynamic replier active reliable multicast (dyram). In *Proceedings of the 7th IEEE Symposium on Computers and Communications (ISCC 2002)*, pages 275–282, Taormina, Sicily, July 2002.

[83] M. Maimour and C. Pham. Dealing with heterogeneity in a fully reliable multicast protocol. In *Proceedings of IEEE International Conference On Networks (ICON 2003)*, Sydney, Autralia, September 2003.

[84] Moufida Maimour and Cong-Duc Pham. Amca, an active-based multicast congestion avoidance algorithm. In *the 8th IEEE Symposium on Computers and Communications (ISCC 2003)*, Kemer-Antalya, Turkey, July 2003.

[85] Sid Ali Guebli. Intégration de la fonction de proxy cache aux routeurs actifs. Technical report, LIRIS/INSA Lyon, jul 2003.

[86] Ernesto Exposito, Patrick Sénac, and Michel Diaz. Fptp : the xqos aware and fully programmable transport protocol. In *IEEE ICON*, Sydney, Australia, sep 2003.

[87] Gcap : Global communication architecture and protocols for new qos services over ipv6 networks. http ://www.laas.fr/ berthou/Public/gcap.html.

[88] L. Courtrai, F. Guidec, Nicolas Le Sommer, and Y. Mahéo. Resource management for parallel adaptive components. In *IPDPS*, Nice, France, apr 2003. in Workshop on Java for Parallel and Distributed Computing.

[89] Erik L.Nygren, Stephen J.Garland, and M.Frans Kaashoek. Pan : A high-performance active network node supporting multiple mobile code systems. In *IEEE OPENARCH '99*, March 1999. http ://www.pdos.lcs.mit.edu/ nygren/pan/.

[90] Globalisation des ressources informatiques et des données, programme national grid'5000, 2003. http ://www.recherche.gouv.fr/appel/2003/grid5000.htm.

[91] Julien Laganier. Securite Distribuée pour Internet. M.Sc. Thesis DEA-2002-07, LIP, July 2002.

[92] Tim Wilkinson. Kaffe : an open source implementation of a java virtual machine. http ://www.kaffe.org/, 2000.

[93] Elliotte Rusty Harold. *Java FAQ*, oct 1997. http ://www.ibiblio.org/javafaq/javafaq.html.

[94] TowerJ. Java compiler. http ://www.towerj.com/.

[95] Jove. Java compiler. http ://www.instantiations.com/jove/product/thejovesystem030314.htm.

[96] J. M. Bull, L. A. Smith, C. Ball, L. Pottage, and R. Freeman. Performance evaluation of java against c and fortran for scientific applications. *Java Grande/ISCOPE 2001*, aug 2001.

[97] Hongjiu Lu. Elf : From the programmer's perspective, may 1995. citeseer.nj.nec.com/lu95elf.html.

L'objectif des réseaux actifs est d'exploiter les réseaux de transport de données de manière plus « intelligente » par l'injection dynamique de services personnalisés dans les équipements réseaux. Le concept de réseaux actifs bien que séduisant soulève de nombreuses problématiques en termes de performances et de sécurité. Ces travaux de thèse se sont principalement concentrés sur le problème de la performance dans les équipements actifs. Nous avons apporté un ensemble d'idées originales pour la conception d'un routeur actif logiciel hautes performances apte à supporter les débits des réseaux gigabits. Nous avons défini une nouvelle architecture de réseau actif générique déployable dans une infrastructure Internet. Deux mécanismes de déploiement dynamique de services ont été intégrés. Nous avons proposé une architecture de nœud actif hautes performances découpée en couches logiques. Cette architecture supporte une classification des services actifs, en termes de consommation de cycles processeur et ressources mémoire, afin de les exécuter dans la couche la plus appropriée. Cette architecture a été validée expérimentalement avec la mise en oeuvre de la suite logicielle Tamanoir. Nous proposons un Environnement d'Exécution ouvert et performant, une bibliothèque de services expérimentaux et des outils de développement assurant la visualisation d'équipements distants et la génération de trafic actif. Des expérimentations sur des plateformes locales très haut débit et des liaisons longue distance Gigabits (RNRT VTHD++) ont démontré l'efficacité de l'environnement Tamanoir. Nous avons par ailleurs déployé Tamanoir dans une grille de calcul nationale (RNTL E-Toile) afin de supporter efficacement les middlewares et applications de grille.

Towards the design of an active network architecture supporting throughputs of gigabit networks

Active networks exploit networks cleverly by dynamic injection of customized services in networks equipments. This new network concept raise performance and security problems. Our work focus on high performance aspects, dynamic deployment of new services and active equipment heterogeneity support. We define a new model of software Execution Environment (EE) and experimentally validate it within the Tamanoir software suite (registered to the APP). We propose solutions coming from distributed system and improve the data path inside active equipments by providing different logical layers. The architecture of our EE is used in RNRT VTHD++ and RNTL E-Toile (Gigabit links) research projects and also by teams to deploy protocols and services deployed on a large scale.

Informatique

Mots-clés : réseaux actifs, programmable, haute performance, gigabits, distribué, service, dynamique, Tamanoir

Laboratoire LIP (UMR N° 5668 CNRS - ENS Lyon - UCB Lyon - INRIA)
Équipe RESO (Protocoles et logiciels optimisés pour réseaux très haut débit)
École Normale Supérieure de Lyon
46, allée d'Italie
69364 LYON Cedex 07 - FRANCE