Éléments de gestion des ressources de calcul dans les réseaux actifs hétérogènes

Virginie Galtier

To cite this version:

Virginie Galtier. Éléments de gestion des ressources de calcul dans les réseaux actifs hétérogènes. Autre [cs.OH]. Université Henri Poincaré - Nancy 1, 2002. Français. NNT : 2002NAN10017 . tel-01746866

HAL Id: tel-01746866
https://hal.univ-lorraine.fr/tel-01746866

Submitted on 29 Mar 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

AVERTISSEMENT

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document.

D'autre part, toute contrefaçon, plagiat, reproduction illicite encourt une poursuite pénale.

Contact : ddoc-theses-contact@univ-lorraine.fr

LIENS

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10
http://www.cfcopies.com/V2/leg/leg_droi.php
http://www.culture.gouv.fr/culture/infos-pratiques/droits/protection.htm

université HENRI POINCARÉ
NANCY 1

S.C.D. - U.H.P. NANCY 1
BIBLIOTHÈQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

Mademoiselle **GALTIER Virginie**

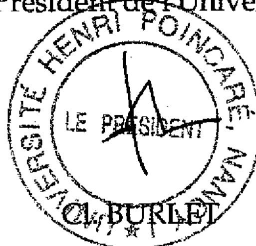
DOCTORAT de l'UNIVERSITE HENRI POINCARE, NANCY 1

en INFORMATIQUE

VU, APPROUVÉ ET PERMIS D'IMPRIMER

Nancy, le 27 MAI 2002 n° 675

Le Président de l'Université



Université Henri Poincaré, Nancy I
24-30 rue Lionnois - B.P. 3069 - 54013 NANCY CEDEX
Tél. : 03 83 85 48 00 - Fax : 03 83 85 48 48

S.C.D. - U.H.P. NANCY 1
BIBLIOTHÈQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

Département de formation doctorale en informatique **École doctorale IAEM Lorraine**
UFR STMIA

Éléments de gestion des ressources de calcul dans les réseaux actifs hétérogènes

THÈSE

présentée et soutenue publiquement le 23 avril 2002

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1
(spécialité informatique)

par

Virginie GALTIER

Composition du jury

Présidente :	Marie-Christine HATON
Rapporteurs :	Andrzej DUDA, Professeur, ENSIMAG, Grenoble Marie-Christine HATON, Professeur, Université Henri Poincaré, Nancy Jean-Jacques PANSIOT, Professeur, Université Louis Pasteur, Strasbourg
Examinateurs :	Laurent ANDREY, Maître de Conférence, Université de Nancy 2 Kim-Loan THAI, Maître de Conférence, Université Pierre et Marie Curie, Paris André SCHAFF, Professeur, Université Henri Poincaré, Nancy
Invité :	Kevin MILLS, *senior research scientist*, NIST, États-Unis

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Remerciements

- aux membres du jury, pour s'être intéressés à ce travail,
- à André SCHAFF, pour son soutien, ses encouragements et pour m'avoir conseillée et guidée pour le bon déroulement de ma thèse,
- à Laurent ANDREY et Olivier FESTOR, pour leurs critiques pertinentes du tapuscrit,
- à Michel COSNARD, directeur du LORIA de 1997 à 2000, pour son soutien,
- *last but not least*, à Kevin MILLS, manager de la division ANTD de NIST de 1999 à juin 2001 et collègue chercheur, pour m'avoir fait confiance dans la réalisation de ce projet, pour son soutien et son aide.

S.C.D. - U.H.P. NANCY 1
BIBLIOTHÈQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

Table des matières

Table des figures ix

Liste des tableaux xi

Introduction générale

1 Objectifs et organisation de la thèse 3
1.1 Partie I : contexte 3
1.2 Partie II : propositions 4
1.3 Partie III : évaluation et perspectives 5
2 Présentation de l'environnement scientifique 7

Partie I Contexte 9

Chapitre 1
Présentation générale des réseaux actifs

1.1 Introduction 11

1.1.1 Motivation : besoin accru d'adaptabilité 11
1.1.2 Bref historique 12
1.1.3 Définition - Principe 14
1.1.4 Plan de ce chapitre 15
1.2 Intérêts des réseaux actifs 17
1.3 Fonctionnement 20
1.3.1 L'architecture théorique 20
1.3.2 Les différentes approches 21
1.4 Comparaison avec des notions cousines 22
1.4.1 Réseaux programmables 22
1.4.2 Le Réseau Intelligent 23
1.4.3 Agents mobiles 23
1.5 Choix des plates-formes de test parmi l'existant 23
1.5.1 Systèmes d'exploitation de nœuds actifs 23
1.5.2 Environnements d'exécution 25
1.5.3 Applications actives 29
1.6 Débuts de standardisation 29
1.7 Problèmes 30
1.7.1 Changements dynamiques de configuration 30
1.7.2 Sécurité 30
1.7.3 Besoins accrus de retour d'information pour l'utilisateur . . . 31
1.7.4 Gestion des ressources plus cruciale 31
1.8 Résumé 31

Chapitre 2
Contrôle de l'utilisation des ressources CPU

2.1 Introduction 33
2.2 Nécessité de contrôler la consommation 33
2.3 Sources de variation des besoins en calcul 34
2.4 Solutions existantes pour limiter la consommation 36
2.4.1 Limites quantitatives fixes 36
2.4.2 Restrictions qualitatives 39
2.4.3 Acquisition des ressources « aux enchères » 40
2.5 Autres sources d'inspiration pour quantifier les besoins 40
2.6 Résumé 42

Problématique induite par le contexte

Partie II Contributions

Introduction

1 Objectifs et cahier des charges . . . 51
2 Idée générale de la solution proposée . . . 51
3 Plan de cette partie . . . 52
4 Description du réseau de test utilisé . . . 52

Chapitre 3
Outil d'observation

3.1 Objectifs du monitorage . . . 53
3.2 Inadaptation des outils existants . . . 55
3.2.1 GPROF . . . 55
3.2.2 Outils de génération de profils de programmes Java . . . 55
3.2.3 Instrumentation de la bibliothèque « glibc » . . . 56
3.2.4 Utilitaire « Strace » de Linux . . . 56
3.3 Implémentation du prototype de traçage . . . 57
3.3.1 Information temporelle fine et précise . . . 57
3.3.2 Grandes lignes de l'implémentation . . . 57
3.3.3 Collecte des résultats . . . 58
3.3.4 Remarques . . . 61
3.4 Évaluation du prototype . . . 61
3.4.1 Sensibilité aux mécanismes de *cache* . . . 61
3.4.2 Évaluation de l'*overhead* dû à l'instrumentation . . . 62
3.5 Instrumentation des plates-formes . . . 62
3.5.1 Nécessité d'une instrumentation . . . 62
3.5.2 Monitorage des applications *ANTS* . . . 63

3.5.3 Monitorage des applications *Magician* 63
3.6 Résumé 64

Chapitre 4
Modélisation des nœuds et modélisation simplifiée des applications

4.1 Modélisation simplifiée des applications actives (modèle M1) 65
4.2 Modélisation des nœuds actifs et prédictions 67
4.3 De la théorie à la pratique avec Linux 68
4.4 Calibration des nœuds 69
4.4.1 Performances du système d'exploitation actif 69
4.4.2 Performances des environnements d'exécution 69
4.4.3 Quand (re)calibrer un nœud ? 70
4.5 Résultats de prédiction 71

Chapitre 5
Améliorations de la modélisation des applications

5.1 Modèle d'application plus complet (modèle M2) 77
5.1.1 Éléments additionnels du modèle 77
5.1.2 Obtention des prédictions 78
5.1.3 Adaptation du modèle d'un nœud à un autre 78
5.1.4 Résultats 79
5.2 Modèle semi-stochastique (modèle M3) 81
5.2.1 Ajout d'informations 81
5.2.2 Obtention des prédictions par simulation 81
5.2.3 Résultats 82
5.2.4 Compromis précision-légèreté 84
5.3 Nœud de référence 84
5.4 Résumé 87

Partie III Applications et évolutions

Chapitre 6
Intégration expérimentale pour la gestion

6.1 Introduction 91
6.2 Modifications apportées à *Magician* 91
6.3 Expérience de contrôle d'exécution 92
6.4 Expérience de prédiction de charge 93
6.4.1 Limites des systèmes de gestion actuels 93
6.4.2 Présentation d'AVNMP 94
6.4.3 Amélioration d'AVNMP 97
6.5 Conclusion 97

Chapitre 7
Propositions pour repousser les limitations de la solution proposée

7.1 Inconvénients d'un modèle empirique 99
7.2 Travaux futurs 100
7.2.1 Modèle à apprentissage permanent 100
7.2.2 Construction du modèle par analyse statique 101
7.2.3 Complicité du développeur 101
7.2.4 Multi-modèle, pluri-prédictions à évaluation continue 102
7.2.5 Modèle compacté 102

Bilan

Résumé des contributions 107

Perspectives 109

Bilan personnel 111

Références

Bibliographie **115**

Communications produites **121**

Sigles utilisés **123**

Annexes

Annexe A
Applications actives utilisées

Annexe B
Éléments du fonctionnement de Linux

Annexe C
Modifications apportées au code du noyau Linux

C.1 linux/include/linux/sched.h 139
C.2 linux/arch/i386/kernel/entry.S 139
C.3 linux/arch/i386/kernel/sys_i386.c 142
C.4 linux/kernel/sched.c 144
C.5 linux/include/asm/unistd.h 146
C.6 linux/kernel/sys.c 147

Annexe D
Taxinomie des benchmarks

S.C.D. - U.H.P. NANCY 1
BIBLIOTHEQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

Annexe E
Prédictions des différents modèles

S.C.D. - U.H.P. NANCY 1
BIBLIOTHÈQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

Table des figures

1.1 Traitement d'un paquet par un nœud classique 15
1.2 Représentation d'un paquet classique 15
1.3 Traitement d'un paquet par un nœud actif 16
1.4 Représentation d'un paquet actif 16
1.5 Architecture d'un nœud actif 20
1.6 Format d'une capsule *ANTS* 27
1.7 Format d'un paquet *Magician* 28

2.1 Allocation de ressources non renseignée 37
2.2 Allocation de ressources optimale 38

3.1 Partage du processeur entre plusieurs processus 54
3.2 Trace d'exécution du processus P de la figure 3.1 55
3.3 Déroulement du traçage du processus de la figure 3.1 59

4.1 Point d'observation du comportement des applications actives 66
4.2 Exemple d'instanciation du modèle simplifié (M1) d'AA 67
4.3 Exemple d'instanciation du modèle d'un nœud actif X 68
4.4 Exemple de calcul du 85e pourcentile avec le modèle M1 68
4.5 Point d'observation des AA dans la pratique 69
4.6 Exemple d'écart entre distributions réelle et simulée du temps d'exécution 73

5.1 Exemple d'instanciation du modèle M2 d'une AA sur un nœud X 78
5.2 Distribution des temps d'exécution sous forme d'histogrammes 81
5.3 Transformation des histogrammes 82

6.1 Configuration utilisée lors des expériences 92
6.2 Fonctionnement d'AVNMP 96

A.1 Ping ANTS 130
A.2 Multicast ANTS : côté client 131
A.3 Multicast ANTS : méthode *evaluate* d'une capsule d'inscription 132
A.4 Multicast ANTS : côté serveur 132
A.5 Multicast ANTS : méthode *evaluate* d'une capsule de message 133
A.6 Ping Magician 134

B.1 Architecture de Linux 136

B.2 Graphe d'états d'un processus Linux 137

D.1 Taxinomie des *benchmarks* 150

Liste des tableaux

1 Caractéristiques des nœuds du réseau de test 52

4.1 Résultats de la calibration des nœuds (en cycles d'horloge) 71
4.2 Erreur (en %) des prédictions avec le modèle M1 73
4.3 Erreur (en %) des prédictions « naïves » 74

5.1 Erreur (en %) des prédictions avec le modèle M2 80
5.2 Erreur (en %) des prédictions avec le modèle M3 83
5.3 Compromis entre précision et légèreté 85
5.4 Influence de la double transformation sur les prédictions de M3 86
5.5 Comparaison résumée des pouvoirs de prédiction 88

6.1 Performances d'AVNMP avec un modèle non adaptatif 95
6.2 Performances d'AVNMP avec intégration du modèle M3 97

7.1 Problème d'un mélange de scénarios tracés différent du mélange réel . . . 100

E.1 Comparaison détaillée des pouvoirs de prédiction des 3 modèles 154

C.D. - U.
BIBLIOTHEQUE ... NCES
Rue du Jardin botanique
54600 VILLERS-LES-NANCY

Introduction générale

Ce document rend compte du travail de recherche réalisé en étroite collaboration avec un laboratoire du NIST[1] au cours de ma thèse dans le projet RÉSÉDAS[2] du LORIA[3].

La première partie de ce préambule introduit, en respectant l'organisation du manuscrit, le contexte technologique de la thèse, puis elle esquisse la problématique et présente sommairement la solution proposée avant de dévoiler succinctement une mise en application de la solution développée suivie d'une critique et de pistes pour des améliorations futures.

La seconde partie de cette introduction est consacrée à une présentation de l'environnement de travail (groupes de recherche).

1 Objectifs et organisation de la thèse

1.1 Partie I : contexte

Réseaux nouvelle génération

La demande de services réseaux ne cesse de s'accélérer car les utilisateurs (individus ou sociétés) expriment des besoins de plus en plus divers et pressants. Les fournisseurs doivent réagir le plus vite possible pour s'adapter, ou mieux, précéder la demande avec des offres compétitives par rapport à celles de la concurrence.

Malheureusement, les architectures réseaux actuelles ne permettent pas une telle flexibilité et dynamicité. Déployer un nouveau service est pour l'instant un processus long et coûteux, démontrant les limites des réseaux traditionnels sur ce point.

Pour dépasser les limites des réseaux actuels « vissés », statiques, une nouvelle génération de réseaux est en cours d'étude et de développement. Elle met en œuvre le concept de réseaux actifs. Ce concept exploite les progrès réalisés depuis le début des années 90 dans les domaines du code mobile ainsi que dans la construction de processeurs. On peut aujourd'hui envisager que de la puissance de calcul soit disponible à chaque nœud d'un réseau (commutateurs, routeurs, passerelles etc...). Ainsi, des instructions peuvent être envoyées et exécutées aux nœuds afin d'en modifier le comportement vis-à-vis des paquets de données qui y transitent. En particulier, cela permet à une application de définir et utiliser sa propre pile de protocoles de communication !

[1] *National Institute of Standards and Technology*, www.nist.gov
[2] www.loria.fr/equipes/resedas/index.html
[3] Laboratoire lorrain de recherche en informatique et ses applications, www.loria.fr

Le chapitre 1 est consacré à la présentation de ce nouveau concept. Nous détaillerons les multiples intérêts des réseaux actifs en en précisant le fonctionnement. Nous y ferons également un rapide tour d'horizon de l'existant pour situer les plates-formes choisies comme cadre de travail. Ce chapitre se termine en mettant l'accent sur les principaux problèmes qu'il reste à résoudre pour que les réseaux actifs convainquent les industriels et passent de l'état expérimental à une utilisation effective par des fournisseurs de services dans leurs architectures réseaux.

Gestion des ressources dans les réseaux actifs

L'introduction de programmabilité dans les réseaux ne va pas sans poser de problème. Les solutions classiques de gestion sont à revoir dans ce nouveau contexte où n'importe qui (ou presque) peut demander l'exécution de son code sur les nœuds qui l'intéressent. Un des défis majeurs freinant le déploiement des réseaux actifs est la résolution des problèmes de sécurité et de sûreté. Cela englobe la recherche de nouveaux mécanismes d'authentification par exemple mais aussi l'allocation et la gestion des ressources.

En effet, comme nous le verrons au chapitre 2, il serait très facile de dégrader très sérieusement les performances d'un réseau actif. Le déploiement d'un code malveillant ordonnant aux nœuds d'effectuer un calcul infini par exemple, peut monopoliser les ressources de calcul et affamer les autres utilisateurs. Nous passerons en revue les différentes solutions existantes pour éviter ce problème dans les réseaux actifs ou dans le domaine connexe du code mobile. La réponse qui est généralement utilisée jusqu'à présent consiste à fixer des bornes quantitatives à l'exécution du code. La difficulté est de définir la valeur de ces bornes.

A l'heure actuelle, aucune unité ne permet d'exprimer les besoins et limites en calcul d'un programme de façon homogène sur un réseau qui ne l'est pas. Par « réseau hétérogène », nous entendons un réseau dans lequel de nombreux facteurs (que nous inventorierons) interviennent et conduisent à des performances très diverses d'une machine à une autre.

Problématique

La recherche d'architectures et d'algorithmes innovants permettant de contribuer à la résolution du problème de l'expression des besoins en calcul d'un programme au niveau des nœuds actifs est l'objectif principal de cette thèse.

1.2 Partie II : propositions

La partie II propose de modéliser l'exécution d'une application active par une suite ordonnée et étiquetée d'éléments, où les éléments représentent les fonctions fournies par un nœud actif aux applications. Ce modèle est transmis aux nœuds avant le premier « vrai » paquet actif de l'application qu'il représente. D'autre part, chaque fois qu'un nœud est ajouté au réseau, ou lorsque sa configuration change, il est « calibré » pour déterminer combien de temps il utilise pour exécuter chacune des fonctions du système actif. La combinaison de ces deux informations (éléments utilisés par une application et

temps nécessaire pour exécuter les éléments sur un nœud particulier) permet à un nœud actif de prédire le temps d'exécution du programme actif modélisé.

Pour atteindre ces objectifs, il est nécessaire de collecter des informations très fines sur l'exécution des applications actives. Aucune fonctionnalité permettant d'obtenir ces informations n'étant prévue au niveau des plates-formes actives et les systèmes d'exploitation dédiés aux réseaux actifs n'étant pas assez avancés au moment où ce travail a débuté, j'ai dû commencer par développer un outil de supervision. Pour cela, j'ai apporté des modifications au noyau du système d'exploitation Linux pour observer et enregistrer combien de cycles d'horloge un processus donné passe à s'exécuter en mode utilisateur ainsi qu'en mode privilégié. Le chapitre 3 présente en détails et évalue cette première contribution.

Ensuite, le chapitre 4 précise quelle « série d'éléments » a été choisie pour modéliser les applications actives et présente une première version du modèle proposé pour représenter le comportement des applications. Nous détaillerons également le modèle représentant les performances des nœuds et le nécessaire processus de calibration des nœuds. Puis nous indiquerons comment les modèles d'applications et les modèles de nœuds se combinent pour fournir des prévisions du temps d'exécution d'une application active sur n'importe quel nœud. Ce chapitre se conclut par une évaluation de la précision des prédictions lors de phases de test.

Les premiers résultats n'étant pas à la hauteur des objectifs fixés, le modèle d'application est ensuite raffiné dans le chapitre 5. Des informations supplémentaires sur la distribution du temps passé dans et entre chaque élément caractéristique sont ajoutées. Les évaluations exposées au cours de ce chapitre permettront de juger empiriquement la qualité des nouveaux modèles.

1.3 Partie III : évaluation et perspectives

Intégration expérimentale pour la gestion

Le chapitre 6 présente notre solution en action au travers de deux expériences concrètes : l'une démontre l'intérêt de notre approche en sécurisation de réseau et l'autre son utilité en gestion prédictive.

Dans une première expérience, nous montrerons comment l'utilisation des modèles mis au point permet de fixer une limite « intelligente » à l'exécution de paquets actifs. La limite est adaptée aux performances des différents nœuds et tient également compte du comportement défini comme normal d'une application. Cela représente une vraie amélioration par rapport aux seuils fixes classiques. En cas d'attaque de refus de service, la limite « intelligente » permet de tuer le plus tôt possible les paquets malintentionnés, tout en ne perturbant pas trop l'exécution des paquets innocents contrairement à la solution « limite fixe » qui conduit soit à un gaspillage de ressources soit à une forte dégradation des applications innocentes.

Dans une seconde expérience, nous démontrons comment nos modèles peuvent s'intégrer dans un système de gestion prédictive du réseau (AVNMP) développé par General Electric. Ce système s'appuyait initialement uniquement sur le nombre de paquets reçus et la vitesse du processeur pour prévoir la charge d'un nœud. L'imprécision de ces

prédictions oblige le système à se resynchroniser souvent avec la charge effectivement mesurée. En utilisant nos modèles pour prévoir les ressources CPU consommées par un paquet actif, les prévisions du système sont meilleures et il peut prévoir l'état du réseau encore plus loin dans le futur, permettant ainsi une meilleure gestion.

Perspectives

Malgré un certain succès dans la mise en application de notre solution, nos modèles souffrent d'une limitation importante. Ils ne représentent correctement une application que si les situations expérimentées lors de la phase de construction du modèle sont représentatives des situations qui seront effectivement rencontrées lors du déploiement et de l'exécution réels de l'application. Le chapitre 7 détaille et illustre cette limitation.

Dans une seconde phase, ce chapitre propose des pistes de recherche ou de développement pour corriger ce défaut. Tout d'abord, on peut envisager qu'une fois créé, un modèle ne reste pas figé. Au fur et à mesure qu'il circule sur le réseau et qu'il rencontre des situations non prévues initialement, ces situations pourraient être intégrées au modèle et leur poids pourrait être renforcé si elles se présentent souvent dans la suite de leur progression dans le réseau. À l'inverse, si un cas initialement prévu n'est jamais rencontré en réalité, son poids pourrait être amoindri. Un modèle ayant cette capacité d'adaptation permet d'envisager un type de construction initial ne faisant pas appel au développeur d'application. Une analyse statique du code pourrait permettre de dégager les différentes situations d'exécution prévues pour un paquet actif. Ce modèle « brut » serait ensuite « renseigné » lors de son premier passage dans le réseau. Le modèle brut pourrait aussi ne contenir que les chemins acycliques à travers le code d'une application et l'expression déterminant le nombre d'itérations à réaliser ; à l'arrivée au nœud, où les conditions locales intervenant dans le nombre d'itérations sont connues, le modèle serait complété avant d'être utilisé. Nous envisagerons aussi une solution où le développeur d'application fournit, pour chaque alternative de son protocole, un modèle associé à un profil d'utilisation.

À l'avenir, on peut s'attendre à une multiplication du nombre et du type des applications actives. De nouveaux modèles rejoindront alors sûrement les nôtres et seront plus ou moins bien adaptés selon les situations. Nous proposons de faire une évaluation semi-continue de la qualité des différents modèles de façon à toujours utiliser celui le mieux adapté au cas présenté.

Puisque la validité de nos modèles est partiellement prouvée, nous suggérons de retravailler la représentation des distributions de temps incluses dans les modèles pour passer de la représentation actuelle sous forme d'histogrammes en une représentation beaucoup plus légère sous forme de combinaison de distributions statistiques classiques, ce qui permettrait de plus d'obtenir des prédictions de façon analytique et non plus en utilisant un simulateur comme c'est encore le cas pour l'instant.

Enfin, comme nous le verrons dans le bilan, notre travail met en lumière un certain nombre de manques qu'il conviendrait de combler, par exemple avec la solution que nous proposons, au niveau des spécifications des plates-formes actives.

2 Présentation de l'environnement scientifique

Au LORIA, le projet RÉSÉDAS développe ses activités de recherche sur les environnements logiciels pour la conception, le développement, le déploiement, l'exploitation et la supervision des applications, services et protocoles de communication dans des environnements hétérogènes. L'équipe a acquis des compétences dans le domaine de la supervision des réseaux et services, notamment via des recherches sur le Réseau de Gestion des Télécommunications (RGT). RÉSÉDAS participe également à l'effort de recherche sur les réseaux actifs, et plus particulièrement leur supervision, depuis 1998 au travers du projet ANAIS (*Active Network Architecture for Internet Service Providers*) et du projet national coopératif RNRT AMARRAGE[4] (Architecture Multimédia & Administration Réparties sur un Réseau Actif à Grande Échelle).

D'autre part, RÉSÉDAS entretient depuis des années des relations avec le NIST à Washington. Les objectifs du NIST sont de *développer les technologies, les méthodes de mesure et les standards pouvant aider les entreprises américaines à être compétitives sur les marchés américains et internationaux.* Dans le cadre de cette mission, un groupe particulier de NIST (ANTD) explore les challenges liés aux technologies émergentes comme les agents mobiles en 1996 (par exemple, développement de la plate-forme AGNI) ou les réseaux actifs depuis 1998. Le financement de ces projets, partiellement assuré par le département du commerce, est complété par des fonds accordés par la DARPA (*Defense Advanced Research Project Agency*). Ce patronage favorise les échanges avec les autres équipes de recherche du programme « Réseaux Actifs » lancé en 1997 pour concevoir une nouvelle plate-forme réseau, plus adaptée aux besoins en services de plus en plus complexes demandés par les applications de défense (sécurité, communication de groupes, routage efficace, re-configuration aisée et dynamique).

C'est dans le cadre de la collaboration forte entre RÉSÉDAS et le NIST que j'ai débuté mon travail de recherche en 1998 par un sujet de DEA mariant les sphères d'intérêt des deux structures : l'étude de l'apport des agents mobiles dans la décentralisation de la gestion de réseaux. En suite logique de ce premier travail, et vu le contexte que je viens de présenter, je me suis intéressée à la gestion des réseaux actifs, animée à la fois par la vocation « orientée mesures » de NIST et par l'héritage de l'importance de la modélisation reçu du LORIA.

[4]www.amarrage.net

Première partie

Contexte

S.C.D. - U.H.P. NANCY 1
BIBLIOTHÈQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

Chapitre 1

Présentation générale des réseaux actifs

Sommaire

1.1	**Introduction**	11
1.2	**Intérêts des réseaux actifs**	17
1.3	**Fonctionnement**	20
1.4	**Comparaison avec des notions cousines**	22
1.5	**Choix des plates-formes de test parmi l'existant**	23
1.6	**Débuts de standardisation**	29
1.7	**Problèmes**	30
1.8	**Résumé**	31

1.1 Introduction

1.1.1 Motivation : besoin accru d'adaptabilité

L'introduction assez récente dans plusieurs pays de la concurrence entre les opérateurs de téléphonie pousse les fournisseurs à se démarquer le plus possible en offrant au client le meilleur prix, la meilleure qualité, des services novateurs ou encore une réactivité accrue par rapport aux différentes offres et demandes qui ne cessent d'évoluer. D'autre part, l'explosion de l'Internet et des réseaux d'entreprise a conduit au développement d'une multitude de protocoles (GSMP[5], RSVP[6]...) et nouvelles classes d'applications ou services (*voice over IP*, vidéo-conférence, communications multipoints...). Ces applications et les protocoles associés peuvent avoir un impact partout dans le réseau : au cœur

[5] *General Switch Management Protocol*, protocole générique utilisé pour la gestion (configuration et contrôle) à distance de commutateurs. La version 2.0 est documentée dans le RFC 2297.

[6] *ReSerVation Protocol*, protocole utilisé par un nœud pour demander au réseau une certaine qualité de service pour un flot de données particulier et par un routeur pour réserver les ressources nécessaires pour assurer cette qualité de service le long du chemin suivi par les données. La première version est documentée par le RFC 2205.

du réseau comme à ses extrémités, et à différents niveaux (couche transport ou couche réseau...). Le rythme des évolutions logicielles inhérentes dépasse largement la capacité des équipementiers à les intégrer dans leurs nouveaux produits et celle des opérateurs à renouveler leur parc de matériel. En effet, les nœuds[7] internes d'un réseau actuel sont « fermés » : les fournisseurs de services n'ont que très peu de contrôle sur leurs mécanismes. Par conséquent, un long processus de standardisation suivi d'une installation « à la main » est nécessaire avant de pouvoir utiliser un nouveau service (cette procédure prend en moyenne 7 ans !).

Ainsi les « vieux » réseaux sont limités aux services pour lesquels ils ont été bâtis (le réseau téléphonique historique pour la téléphonie, X.25 pour les données etc.). Les réseaux large bande, bien que représentant une amélioration grâce à la possibilité de combiner sur un même réseau des services de transport de la voix et des données, n'offrent que peu de possibilités d'extensions face au rythme frénétique des nouvelles demandes de services car leurs composants sont généralement vissés (*hard-codés*) et ne permettent pas l'introduction de nouveaux services.

La mise en place du Réseau Intelligent [ZG97, Kun93] à la fin des années 80 a amorcé l'émergence de réseaux plus ouverts, prévoyant des mécanismes permettant le déploiement rapide de nouveaux services, pour les réseaux de téléphonie. Pour dynamiser de façon similaire les réseaux de transport de données à commutation de paquet, le concept de réseau actif a commencé à apparaître.

D'autre part, si les applications circulaient hier sur un réseau relativement homogène, elles doivent aujourd'hui pouvoir s'adapter à des capacités extrêmement variables selon que l'utilisateur est connecté à haut débit depuis un super-calculateur ou bien par une connexion moins robuste depuis son téléphone portable ou son PDA. Les applications doivent être conscientes des contraintes physiques sous-jacentes et s'y adapter. Les réseaux actifs permettent cette adaptabilité.

Cette tendance à la programmabilité des réseaux est également favorisée par le fait que suite à une diminution des coûts matériels, de plus en plus de puissance de calcul et de mémoire sont disponibles dans les éléments intermédiaires des réseaux. Les progrès réalisés dans les domaines du code mobile et de sa sécurité, de la compilation rapide et efficace et des systèmes d'exploitation extensibles dynamiquement ont, de façon concomitante, stimulé et été nourris par cette quête.

1.1.2 Bref historique

On considère souvent que le premier réseau actif fût le réseau radio expérimental de l'université de Linköpings (Suède) construit au début des années 80 (projet SOFNET [ZF83a, ZF83b]). Dans ce réseau, chaque nœud était un ordinateur et chaque paquet arrivant était considéré comme un programme (écrit en Forth) à exécuter. Mais la puissance de calcul était encore trop chère à l'époque pour que l'on puisse imaginer en équiper chaque nœud de façon généralisée. Puis l'arrivée du modèle en couches en 1984 a proposé une façon plus économique de faire un réseau et le concept de SOFNET est tombé dans

[7]Les nœuds d'un réseau sont les commutateurs (*switches*), routeurs, répéteurs multi-ports (*hubs*), ponts (*bridges*), passerelles (*gateways*) et éventuellement les machines des utilisateurs ou les serveurs.

l'oubli.

Une dizaine d'années plus tard, plusieurs projets cherchant à introduire plus de modularité au niveau des protocoles réseau ont commencé sans le savoir à réinventer cette idée.

L'idée de permettre au protocole de s'adapter à l'application (au lieu du contraire) est apparue au MIT avec le concept d'ALF (*Application Level Framing* [CT90, Chr96]). ALF part du constat que le modèle de protocoles en couches force les applications distribuées à n'utiliser que les services pré-définis du réseau qui ne sont pas forcément les plus adaptés au traitement des données particulières à l'application et peuvent ralentir inutilement le réseau ou l'application. ALF visait à rendre le contrôle de certains paramètres de transmission (en particulier le traitement des données perdues ou en désordre) à l'application puisqu'elle est la seule à comprendre la sémantique des données échangées et donc la seule à pouvoir fixer intelligemment les contraintes nécessaires pour leur transfert. ALF se distingue pourtant sur deux points du concept actuel de réseau actif que nous allons étudier :

- ce mécanisme ne concerne que les paquets de données et ne s'applique pas aux fonctions de régulation du trafic elles-mêmes,
- il n'a lieu qu'aux extrémités du réseau (là où l'application distribuée envoie ou reçoit ses données) et pas aux nœuds de transmission intermédiaires.

À peu près à la même époque, l'université de l'Arizona développe un nouveau noyau de système d'exploitation, appelé x-Kernel [HP91], fournissant une architecture permettant de construire et composer des protocoles réseaux de manière à faciliter l'implémentation de protocoles de communication efficaces. Horus [vRBM96] reprend ces idées et les étend aux communications de groupe (et non plus seulement point-à-point) et aux applications distribuées (qui peuvent notamment préciser si elles souhaitent une conservation de la causalité des interactions par exemple). Mais là encore, l'application ne peut préciser sa pile de protocoles préférée pour recevoir ou envoyer des données qu'aux extrémités du réseau.

Au milieu des années 90, plusieurs projets de réseaux réellement programmables de bout en bout (et non plus seulement aux extrémités) voient le jour, le plus souvent dans le cadre du programme financé par la DARPA[8]. Parmi les majeurs, on peut citer :

- À l'université de Pennsylvanie, le projet SwitchWare [Swi] développe un commutateur ATM logiciel dans lequel des programmes peuvent être téléchargés pour modifier le comportement des liens (entre autres),
- À Georgia Tech, les nœuds du réseau du projet Canes [BCZ97] sont équipés de fonctions et les paquets transportent, en plus des données, une information de contrôle qui sélectionne une des fonctions et un ensemble d'arguments à utiliser pour traiter le paquet. L'intérêt de cette approche est démontré en l'appliquant au traitement des congestions dans le cas d'un transfert de flux MPEG.
- À l'université de Columbia, NetScript [YdS96] permet de composer et de déployer sur le réseau une pile de protocoles « sur mesure ».

L'appellation actuelle de « réseau actif » a finalement été introduite dans la communauté réseau en 1996 par Wetherall et Tennenhouse avec leurs travaux sur l'option

[8]http ://www.darpa.mil/ito/research/anets/

active d'IP [TW96, WT96]. Leur prototype utilisait les extensions IP (v4) pour transporter du code (en TCL étendu par des fonctions de contrôle du paquet et de son environnement). À la réception, les routeurs reconnaissant cette option exécutaient le code (et les autres effectuaient par défaut un *forward* classique). Il pouvait en résulter une modification des champs de l'en-tête ou des données du paquet et la création de nouveaux paquets, utilisant éventuellement la connaissance de l'état du nœud (heure locale, adresse...). À la différence des projets précédents, cette approche est complètement *in-band* (c'est-à-dire que le code est transporté avec les données à traiter). Ce projet a ensuite donné naissance à un prototype en Java, appelé ANTS (*Active Node Transfer System* [WLG98, Van97, Wet97, WGT98, Wet99b]), qui était la plate-forme active la plus populaire durant mon travail de thèse même si à présent ASP (*Active Signaling Protocol*[9]) semble prendre le relais.

Mais d'autres projets qui ont mûri en parallèle partagent aussi la scène, comme Magician [KMH+98, Kul] ou PLAN ([HKM+98a], issu du projet SwitchWare). Il est encore impossible dans la conjoncture actuelle de prédire laquelle de ces architectures s'imposera vraiment comme réseau du futur (le cas échéant...).

Depuis 1997, le mouvement « réseaux actifs » n'a cessé de se développer comme en témoignent le nombre croissant de projets de recherche (notamment aux États-Unis grâce aux investissements de la DARPA, mais aussi un peu partout en Europe[10], là aussi avec des soutiens gouvernementaux), articles (numéros spéciaux dans les magazines de l'IEEE et de l'ACM) et conférences qui lui sont consacrés ou lui taillent tout du moins une place de plus en plus importante (IM, OPENARCH, IWAN, DSOM, NOMS pour n'en citer que quelques unes). Les industriels (Alcatel, Lucent, Hitachi, NEC, General Electric, Boeing ou encore Nortel Networks) commencent eux aussi à s'intéresser à ce concept.

1.1.3 Définition - Principe

Les réseaux traditionnels à commutation de paquets transportent des données d'un nœud initial vers la destination sans ouvrir « l'enveloppe » contenant les données (voir figures 1.1 et 1.2). La tâche des nœuds intermédiaires en est limitée au routage, avec parfois des contraintes de qualité de service (*QoS*).

Dans un réseau actif (voir figures 1.3 et 1.4), le nœud intermédiaire peut ouvrir l'enveloppe et y découvrir des instructions qui lui indiquent que faire du paquet ou des paquets suivants (étape 2 sur la figure 1.3). Ces instructions peuvent être de consulter l'état du nœud et de compresser toutes les données des paquets suivants si une congestion est détectée (étape 3a) (intervention sur la couche réseau), ou de modifier la table de routage du nœud (étape 3b) avant d'envoyer éventuellement le paquet modifié ou un nouveau paquet (étape 4) (couche transport).

Un réseau actif est un réseau qui permet à son fournisseur ou à ses utilisateurs de programmer dynamiquement la façon dont chaque flux qui le traverse doit être traité. Comme nous allons le détailler dans ce chapitre, plusieurs infrastructures correspondent

[9] http ://www.isi.edu/active-signal/ARP/index.html
[10] http ://www.loria.fr/~festor/RAF/RAF.html ou ~galtier/ANbookmark.html

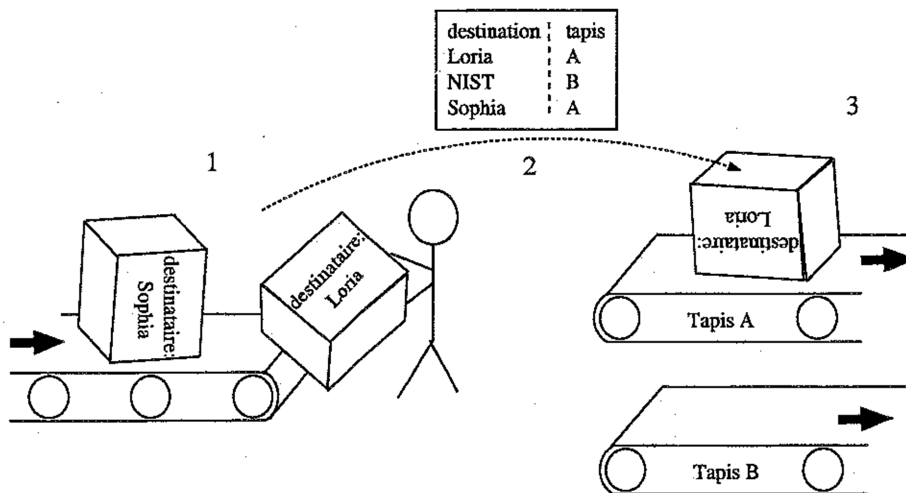


FIG. 1.1 – Traitement d'un paquet par un nœud classique

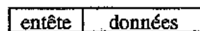


FIG. 1.2 – Représentation d'un paquet classique

à cette définition. Elles diffèrent dans la définition de qui peut modifier quoi dans le réseau et par les technologies employées mais toutes couplent à un réseau de transfert des données des abstractions permettant de le programmer.

Le réseau est doublement actif :

- les nœuds du réseau peuvent traiter les données des utilisateurs,
- les utilisateurs peuvent injecter du code dans le réseau pour un service « sur-mesure ».

Pour garder une vue critique sur ce nouveau concept, il peut être intéressant de noter qu'à l'opposé de cette tendance la nouvelle version d'IP (IPv6) essaye de limiter au maximum les quelques modifications qu'IPv4 inflige à l'enveloppe des données (*checksum* etc) car cela pénalise les routeurs.

1.1.4 Plan de ce chapitre

De nombreux articles, dont [FCF00, Pso99, SCM+99b, CMK+99, TSS+97, Mun97, Nyg96, Jr98, SCM+99a, CBZS98], dressent un état de l'art plus ou moins à jour et plus ou moins détaillé de la recherche dans le domaine des réseaux actifs. L'objectif de ce chapitre est de familiariser le lecteur avec ce nouvel univers afin d'une part de motiver la recherche présentée dans la seconde partie de ce document et d'autre part de fournir les bases pour comprendre la solution proposée.

Dans un premier temps, nous détaillons l'intérêt des réseaux actifs avant d'en expliquer et illustrer les principes de fonctionnement. Une comparaison avec des notions voisines permettra de fixer le cadre. Puis nous présenterons les différentes architectures disponibles en insistant plus particulièrement sur celles que nous avons utilisées lors de

FIG. 1.3 – Traitement d'un paquet par un nœud actif

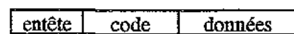


FIG. 1.4 – Représentation d'un paquet actif

nos expériences. Nous évoquerons ensuite les différentes actions de standardisation en cours. Enfin, nous énumérerons les problèmes qui doivent encore être résolus avant que les réseaux actifs puissent prospérer.

1.2 Intérêts des réseaux actifs

Comme nous l'avons esquissé en introduction, c'est principalement pour répondre au besoin accru d'adaptabilité des réseaux que le concept de réseau actif est apparu. Nous allons voir dans cette section qu'il a d'autres intérêts. On peut citer :

- **Déploiement rapide et sans rupture de nouveaux services et protocoles**
  En moyenne, 7 ans s'écoulent entre « l'invention » d'un service ou protocole et le moment où il peut enfin être utilisé après avoir été discuté, standardisé, implanté et déployé à large échelle après bien des problèmes techniques, des problèmes de compatibilité avec l'existant mais aussi des luttes « économico-politiques ». Les réseaux actifs permettent de déployer et d'utiliser un nouveau protocole sans longues procédures d'accord préalables et ce nouveau protocole peut en général coexister avec une autre version. Cela permet aux opérateurs de proposer aux usagers des services en phase avec leur demande. Parmi les exemples de déploiement de protocoles par les réseaux actifs, on peut citer le multicast fiable [LGT98].
- **Amélioration des performances d'une application**
  Grâce aux réseaux actifs, une application peut préciser le traitement qu'elle souhaite voir appliquer à ses paquets en fonction de l'état observé du réseau (congestion à tel nœud, à tel moment etc.). Cette « réactivité intelligente » améliore les performances des applications voire permet l'apparition d'applications jusqu'ici impossibles. [WLG98] présente l'exemple d'une application de cotation boursière en direct. Chaque nœud entre le serveur contenant les informations boursières à jour et l'utilisateur place dans son *cache* les valeurs des actions envoyées par le serveur en réponse à une requête, et note également la date à laquelle cette réponse a été envoyée. Ensuite, lorsque ce nœud reçoit une requête d'un client, il regarde d'abord s'il peut y répondre lui-même grâce aux valeurs dans son *cache* si elles sont assez récentes par rapport aux exigences formulées par l'utilisateur et accompagnant sa requête. Si c'est le cas, il répond lui-même au client, avec un meilleur temps de réponse que si la requête avait dû aller jusqu'au serveur (qui peut être surchargé de surcroît !), les performances de l'application sont donc améliorées. Si les informations contenues dans le *cache* ne satisfont pas la requête du client, elle est normalement acheminée plus en avant vers le serveur. Le débit de l'application est augmenté puisque plusieurs requêtes peuvent être traitées simultanément. Les *caches Web* classiques ne permettent pas cela car ils ne mémorisent pas les informations avec une granularité suffisamment fine et deux portefeuilles, même très proches, conduisent à « mettre en *cache* » deux pages Web différentes.
- **Amélioration des performances du réseau**
  L'exemple précédent montre que l'utilisation des réseaux actifs permet également d'améliorer les performances du réseau. Le serveur est plus disponible et les liens proches du serveur sont moins encombrés puisqu'une partie des flux est interceptée

et traitée par les nœuds intermédiaires.
Les travaux réalisés à Georgia Tech [BCZ97] fournissent un autre exemple : un traitement particulier est défini pour le transport des flux MPEG et permet de préciser à un nœud que s'il doit absolument éliminer des paquets pour résoudre une congestion, et qu'il élimine une trame I, alors il doit aussi éliminer les trames B et P correspondantes (car elles sont inutiles sans la trame I, par rapport à laquelle elles sont des différentiels). Cela contribue à résoudre plus intelligemment et efficacement la congestion.
De façon générale, en introduisant des traitements dans le cœur du réseau et non plus seulement à ses extrémités, les réseaux actifs permettent d'améliorer les performances du réseau en compressant ou regroupant des flux au plus proche de leur source, en filtrant plus rapidement des attaques, en adoptant dynamiquement un routage plus adapté, etc.

- **Placement optimisé des services dans le réseau**
  Les réseaux actifs permettent de placer des services (pas nécessairement actifs) à des endroits spécifiques ou stratégiques du réseau, emplacements qui bien sûr varient selon l'application et pour une application donnée, selon le trafic. Par exemple, ils offrent la possibilité de n'installer un protocole adapté à la visio-conférence que sur le chemin emprunté par les paquets de la visio-conférence (et eventuellement, seulement pour la durée de la conférence).
- **Gestion efficace du réseau**
  Les réseaux actifs sont de bons candidats pour la gestion de réseaux et sont bien adaptés au paradigme de gestion par délégation (le chapitre 6 présente un système qui illustre cette idée). Ils permettent d'améliorer la sécurité en filtrant les attaques au plus proche de la source, par exemple, des pare-feux actifs[11] évitent d'avoir un pare-feu par entrée (ce qui est coûteux, et parfois difficile à maintenir lorsque la topologie du réseau change fréquemment), permettent d'étendre le périmètre de sécurité autour du réseau privé (en positionnant des auxiliaires à l'extérieur) ou de renforcer la sécurité autour d'un nœud critique (en ajoutant une couronne d'auxiliaires autour de cet hôte), tout cela en équilibrant mieux la charge (le trafic est re-routé vers le pare-feu le moins occupé ou le plus proche, ou encore celui qui est programmé pour s'occuper d'un type de paquets particulier...). Les autres domaines de la gestion de réseaux peuvent également bénéficier des avantages des réseaux actifs pour déployer leurs services. Les éléments actifs permettent une plus grande automatisation des fonctions de gestion et pourraient même permettre l'apparition de réseaux auto-gérés. Les réseaux actifs offrent la possibilité de créer des outils de surveillance plus sélectifs, plus adaptés au passage à l'échelle, des outils de simulation, de test, très utiles pour développer des produits réseau. Une équipe de BBN Technologies se concentre sur l'application de la technologie active à la gestion et surveillance de réseau [SJTS+00].
- **Facilité d'expérimentation des réseaux**

[11] Les entrées du réseau privé ne sont plus systématiquement équipées de pare-feux dédiés mais simplement de nœuds actifs qui pourront soit traiter les paquets eux-mêmes soit les re-router vers un pare-feu placé n'importe où dans le réseau privé.

S.C.D. - U.H.P NANCY 1
BIBLIOTHEQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

En attendant leur utilisation réelle et à grande échelle, les réseaux actifs fournissent déjà une plate-forme intéressante pour l'expérimentation de tous les nouveaux protocoles réseaux, actifs ou non : il devient très facile, rapide et relativement bon marché de déployer, tester ou valider de nouveaux concepts. Les réseaux actifs constituent donc un outil particulièrement intéressant pour les chercheurs ou les développeurs. Ils constituent également une plate-forme idéale pour l'enseignement des principes des réseaux.

- **Adaptation autonome des services à l'état du réseau**
  Un routeur nomade actif pourrait modifier sa politique selon la nature du lien entre l'utilisateur et le réseau ; par exemple, les communications seraient cryptées lors des connexions depuis un réseau extérieur. Un routeur actif serait capable de détecter une congestion dans un traffic MPEG et de décider quels paquets supprimer pour y remédier. Un service installé pour la transmission de visio-conférence pourrait adapter sa transmission aux differentes contraintes de bande-passante.
- **Amélioration du routage**
  Les nœuds ont généralement la meilleure connaissance possible et disponible des conditions locales de connectivité. Un protocole actif tirerait avantage de cette connaissance pour ces décisions de routage. Par exemple, le projet REVERE d'UCLA[12] compte utiliser les réseaux actifs pour assurer que des messages d'alerte de sécurité arrivent à différents nœuds stratégiques par au moins deux routes différentes.
- **Gestion de la congestion en temps réel**
  En se basant sur le type du trafic et la nature et la sévérité d'une congestion, les réseaux actifs permettent de re-router le trafic autour des points de congestion en temps réel, c'est-à-dire beaucoup plus rapidement que ne le permettent les solutions actuelles.
- **Services adaptés à l'application**
  Les réseaux actuels fournissent un service homogène à toutes les applications. Les réseaux actifs fournissent une architecture qui permet une adaptation des services réseau application par application. Les applications fonctionnant sur un réseau actif peuvent contrôler la politique de suppression de leurs paquets, quels paquets doivent recevoir la priorité par rapport aux autres, et quels paquets recoivent des services spéciaux.
- **Etablissement de priorités entre applications**
  Les réseaux actuels traitent toutes les applications de la même manière. Pourtant, du point de vue du client certaines sont plus importantes que d'autres. L'utilisateur doit avoir la possibilité d'indiquer quelles sont ses priorités parmi les différentes tâches qu'il a lancées.
- **Sécurité itinérante**
  Dans les réseaux actuels, les mécanismes de sécurité sont appliqués aux extrémités des communications et doivent être suffisants pour contrer les attaques pouvant survenir dans l'environnement le plus risqué traversé par les paquets. Avec les réseaux actifs, les mécanismes de sécurité peuvent accompagner les paquets de façon que le mécanisme approprié soit choisi dynamiquement en fonction de l'environne-

[12] http ://lever.cs.ucla.edu/revere/

ment traversé : si le paquet traverse un réseau ou un nœud auquel on fait moins confiance, les mécanismes de protection élévée du paquet peuvent être activés puis désactivés lorsque l'on revient en territoire sûr.

- **Sécurité adaptée à l'application**
  Des solutions spécifiques de sécurité sont requises par certaines applications ou pour certains utilisateurs, les réseaux actifs autorisent l'adaptation de la sécurité à l'application.

1.3 Fonctionnement

1.3.1 L'architecture théorique

Un ensemble de projets financés par la DARPA se sont regroupés en ateliers de discussion pour tenter d'établir des documents définissant un cadre commun. La communauté internationale peut également participer à l'évolution de ces documents. En s'appuyant en partie sur [Cal99] qui identifie les différents composants d'un nœud actif et leurs relations, on peut représenter un nœud selon la figure 1.5.

Un nœud peut comporter plusieurs environnements d'exécution (ou « EE »). Un EE fournit un environnement d'exécution virtuel (similaire par exemple à la machine virtuelle de Java) dans lequel des applications actives (ou « AA ») peuvent s'exécuter. L'EE a parfois besoin d'accéder à des ressources physiques (carte réseau, disque...) ou logiques (table de routage par exemple) du nœud. Il le fait par l'intermédiaire d'appels aux fonctions du système du nœud actif (*nodeOS*). Les rôles du système d'exploitation actif sont principalement de gérer les ressources du nœud et de démultiplexer les paquets arrivant vers l'EE/AA approprié en se basant sur l'entête ANEP[13] du paquet.

Comme plusieurs EEs existent et que plusieurs systèmes d'exploitation actifs sont développés, une interface définie dans [Pet01] devrait permettre à n'importe quel EE de s'exécuter sur n'importe quel système d'exploitation présentant cette interface constituée de fonctions standard. Les ressources d'un nœud comprennent sa bande passante, sa capacité de calcul et sa mémoire. Lorsqu'un EE fait appel à un service du système d'exploitation actif, cette demande est accompagnée d'un identifiant de l'entité à facturer pour ce service (cette entité pouvant être l'EE lui-même ou un utilisateur ou une application pour le compte duquel l'EE s'exécute). Le système d'exploitation actif présente alors la requête à un module de sécurité qui vérifie, en utilisant la base de données de sécurité du nœud, que l'entité est autorisée à recevoir le service demandé. Les détails de ces authentifications et autorisations sont présentés dans un autre document de référence [Mur01]. De plus, le document d'architecture prévoit la présence sur chaque nœud actif d'un EE de supervision. Cet EE de supervision a pour rôle de maintenir la base de données de sécurité du nœud. Il doit aussi permettre de charger de nouveaux EE sur les nœuds, ou de mettre à jour ou configurer ceux existants. Enfin, il joue un rôle dans la gestion à distance des nœuds et peut aussi fournir aux EE locaux des informations de supervision.

[13] *Active Network Encapsulation Protocol* [ABG+97]

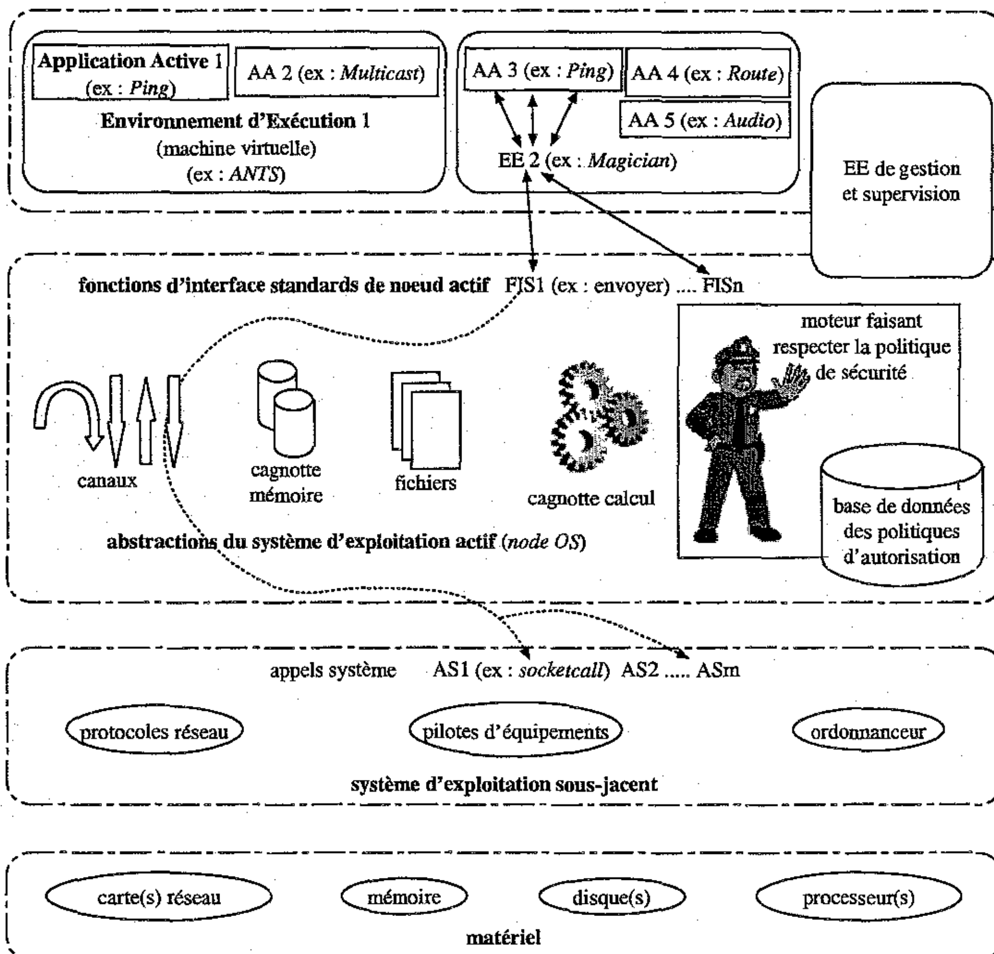


FIG. 1.5 – Architecture d'un nœud actif

Une des difficultés majeures rencontrée dans le travail présenté dans ce document est qu'aucune des plates-formes existantes n'implémentait complètement cette architecture théorique. Aucun système d'exploitation actif ne présentait l'interface décrite ; il était souvent impossible pour le système d'exploitation actif d'identifier finement d'où provenait une demande de service, l'EE de supervision - pourtant si important - reste encore mal défini, etc. Heureusement la situation semble changer ; de nombreux projets ont mûri depuis le début de ce travail et de plus en plus fournissent ces abstractions cruciales.

1.3.2 Les différentes approches

Le code de traitement des paquets peut être pré-installé à un nœud par le fournisseur de services (approche dite *discrète* car le transport des paquets de données est découplé du transport des paquets de code) ou transporté par des paquets actifs dans le même flux que les paquets actifs contenant des données (approche *intégrée*). Selon l'approche choisie, le paquet contient l'identifiant du traitement à lui appliquer ou le code du traitement lui-même. Des solutions intermédiaires existent où le programme actif est simplement une combinaison de primitives pré-définies présentes sur le nœud. Dans nos travaux, nous avons pour l'instant ignoré la phase quelquefois nécessaire de chargement du code car avec les plates-formes que nous avons utilisées, cette phase n'a lieu qu'une fois, avant ou avec le passage du premier paquet de données actif, ce que nous considérons comme négligeable par rapport aux très nombreux paquets de données qui vont transiter et être traités par la suite. Même si c'est possible, un service n'est pas installé sur un nœud pour traiter un seul paquet mais plutot pour traiter tout un flux.

Chaque approche a ses avantages et ses inconvénients, et ils dépendent du point de vue adopté. Dans l'approche discrète, l'opérateur a plus de contrôle sur ce qui se passe aux nœuds que dans l'approche intégrée où les utilisateurs ont plus de liberté et de dynamisme. L'approche discrète peut être plus performante pour transporter des codes volumineux, dans ce cas le code est généralement déployé pour plus longtemps que juste le passage d'un flux.

D'autres classifications sont parfois employées : séparer les architectures permettant de modifier les données de celles n'agissant qu'au niveau contrôle de flux, séparer les plates-formes permettant à un paquet de déposer des informations, un état à un nœud (*soft-state* si cette information disparaît après re-démarrage du nœud, *stateful* si cette information est conservée) de celles ne l'autorisant pas, etc.

1.4 Comparaison avec des notions cousines

Les réseaux programmables, le Réseau Intelligent et les agents mobiles proposent des concepts voisins des réseaux actifs. Cette brève section vise à établir la distinction.

1.4.1 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 (définition empruntée à [FCF00]).

Cette définition englobe tout d'abord les réseaux comportant des routeurs programmables. Plusieurs exemples prouvent que les équipements réseaux deviennent de plus en plus programmables :

- Les routeurs Windows NT peuvent être administrés à distance en utilisant le *Routing and Remote Access Service (RRAS)* (qui permet par exemple d'ajouter des protocoles de routage et des interfaces, de consulter la table de translation des adresses, d'ajouter ou supprimer des filtres de paquets, etc.).
- Les routeurs Cisco proposent une sorte d'*API* unifiée permettant leur configuration, leur maintenance et leur mise-à-jour : *Internetworking Operating System (IOS).*

Dans le domaine des télécommunications, l'approche dite « signalisation ouverte » entre également dans le cadre des réseaux programmables. Mais cette approche se limite comme son nom l'indique à fournir l'accès au plan de signalisation des équipements. Nous détaillons ci-dessous le cas du Réseau Intelligent.

Les réseaux actifs peuvent être vus comme un sous-ensemble des réseaux programmables qui va plus loin que les approches déjà évoquées : les plans de supervision et de données peuvent également être manipulés, différents acteurs (fournisseurs de services, utilisateurs...) peuvent agir sur le réseau et le déploiement des services peut-être couplé ou non au flux des données.

1.4.2 Le Réseau Intelligent

Le Réseau Intelligent (RI ou *IN*, [ZG97, Kun93]) a été introduit par les opérateurs de télécommunication pour fournir de nouveaux services dynamiquement. Le principe consiste en la possibilité d'exécuter diverses fonctions en différents points du traitement d'appel. Le Réseau Intelligent se distingue des réseaux actifs essentiellement sur les points suivants :

- dans le RI, seul le fournisseur introduit de nouveaux services alors que dans certains réseaux actifs, les utilisateurs aussi peuvent ajouter des fonctions de traitement aux nœuds,
- dans le RI, le contenu des paquets n'est jamais modifié, seule la signalisation est concernée,
- le RI est lié à des réseaux à commutation de circuit (type ATM, RTC) alors que les réseaux actifs sont plutôt orientés commutation de paquet,
- dans le RI, la description d'un nouveau service doit obligatoirement respecter le *Basic Call Process* pour assurer la cohérence avec le fonctionnement normal du réseau (par exemple, il n'est pas possible de demander des informations à un usager alors que le réseau est en train d'établir l'appel vers cet usager) ; ce modèle n'existe pas pour les réseaux à commutation de paquet.

1.4.3 Agents mobiles

Les agents mobiles ([Ber99] en dresse un état de l'art) sont des entités plus ou moins autonomes qui peuvent réaliser certaines tâches dans le réseau à la place d'un utilisateur

(comme par exemple rechercher des informations sur Internet pendant que l'utilisateur fait autre chose et lui présenter les résultats lorsqu'il se reconnecte).

Les plates-formes actives peuvent être considérées comme une catégorie de plates-formes à agents mobiles mais alors que ces dernières sont souvent orientées exécution, calcul ou application, les réseaux actifs sont plutôt orientés communication et protocole. [Cal99] stipule que les réseaux actifs doivent être capables de router des datagrammes IPv4 classiques à une vitesse comparable à celle des routeurs « passifs » classiques et que leur objectif n'est pas de servir de système de calcul distribué. Par conséquent, les plates-formes actives agissent généralement à des couches beaucoup plus basses que les agents mobiles.

1.5 Choix des plates-formes de test parmi l'existant

L'objectif de cette section est d'introduire les plates-formes que nous avons choisies comme support pour notre travail de recherche.

1.5.1 Systèmes d'exploitation de nœuds actifs

Les systèmes dédiés

Un certain nombre de projets ont développé des systèmes d'exploitation dédiés aux réseaux actifs :

- *Bowman* [MBZC00] : développé à Georgia Tech pour combler le manque de système d'exploitation actif en 1999, ce système est parmis ceux qui ont le plus influencé les spécifications présentées dans [Cal99] (abstractions pour les canaux, les « flots unitaires » et les enregistrements d'états dans les nœuds). Mais à ses débuts, ce système ne supportait que l'environnement d'exécution de Georgia Tech, *CANEs* (*Composable Active Network Elements*), par ailleurs supporté par aucun autre système d'exploitation. Ne voulant pas nous restreindre à *Odyssey* (nom de l'ensemble *Bowman* + *CANEs*), nous avons écarté *Bowman*. Aujourd'hui, il est dit que *Bowman* peut supporter d'autres EE que *CANEs* (mais sans préciser lesquels).
- *Moab* [THL01] : développé à l'Université de l'Utah en 1999 (à l'aide de leur boîte à outils *OSKit* permettant de construire facilement des systèmes d'exploitation), c'est une implémentation en C de l'*API* définie dans [Cal99] (les travaux préliminaires de *Moab* ont d'ailleurs influencé certaines orientations des spécifications). À terme, si les spécifications sont respectées, tous les EE devraient donc pouvoir tourner sur *Moab*. Mais ce n'était pas le cas en 1999 (et ca ne l'est toujours pas...). *Janos* a alors été développé en 2000 pour supporter au-dessus de *Moab* les EE basés sur Java. En pratique, *ANTSR* (une version de l'environnement d'exécution *ANTS* modifiée pour exploiter les services fournis par *Janos*) a longtemps été le seul EE supporté (et nous n'avons pas connaissance à ce jour que cette situation ait évolué).
- *Joust* [HPB+99] : développé à l'Université de l'Arizona en 1999 (au dessus de *Scout*, un système d'exploitation orienté communication dans lequel les applications peuvent créer des « chemins » pour que le traitement de leur flux soit optimisé), le but de cette plate-forme est de permettre l'exécution très rapide des

applications. Mais en contre-partie du gain de performance, ce système ne respecte pas les spécifications données dans [Cal99].
- *AMP* [Sch01] : Network Associates Inc. (NAI) Labs travaille sur un système d'exploitation actif basé sur l'Exokernel[14] du MIT [Eng98]. Malheureusement, très peu de documents ou de code sont disponibles pour *AMP* qui se dit conforme aux spécifications de [Cal99], performant et sûr. De même, il n'est pas possible d'obtenir le prototype développé qui supporterait les environnements d'exécution *ANTS*, *Secure ANTS* et *PLAN*.
- *SANE-O/S*[15] était un système d'exploitation actif en développement à l'Université de Pennsylvanie dont un des objectifs principaux était de réaliser un commutateur actif dans lequel ANEP tournerait à un niveau équivalent à IP. Ce projet s'est terminé en mars 2001 avec le départ des étudiants qui travaillaient dessus.
- *RCANE* (*Resource Controlled Active Network Environnement*) [Men99] est une architecture construite sur l'OS Nemesis[16] à l'Université de Cambridge. Elle permet à une application de réserver des ressources en bande passante, en mémoire et en calcul. Ces travaux très intéressants n'étaient pas terminés lorsque j'ai commencé les miens. À présent, *RCANE* supporte l'environnement d'exécution *PLAN* et une version de *ANTS* réécrite en OCaml (seul bytecode supporté par *RCANE*) mais il n'existe toujours pas de « guide » permettant à une application de savoir combien de ressources elle doit réserver. Nos travaux sont donc en fait complémentaires des mécanismes que veut offrir RCANE.

Choix de Linux

Comme on vient de le voir, en 1999, les projets de systèmes d'exploitation actifs n'étaient pas tous très avancés : pas toujours opérationnels, pas ou peu documentés, pas ou peu d'EE supportés, suivi incertain... En revanche, la majorité des EE disponibles à l'époque tournaient au-dessus d'un Unix, avec une machine virtuelle Java non modifiée ou un interpréteur TCL classique. Comme le projet exigeait de plus un contrôle très fin d'actions se déroulant parfois à très bas niveau, nous avons choisi d'utiliser Linux comme « système d'exploitation actif » pour la disponibilité de ses sources et d'outils divers.

Il faut cependant noter que notre plate-forme de test n'a servi qu'à évaluer des idées qui ne sont pas intimement liées à un système d'exploitation particulier et que nos tests pourraient sans doute être adaptés aux systèmes d'exploitation actifs dédiés qui ont beaucoup progressé depuis 1999.

[14]Exokernel est un système d'exploitation dans lequel toute la gestion des ressources est accessible directement aux applications au travers de bibliothèques, le système d'exploitation assurant uniquement qu'une application ne contrôle pas les ressources d'une autre. Cette grande finesse permet d'obtenir de très bonnes performances mais il devient très complexe d'écrire les applications.

[15]http ://www.cis.upenn.edu/~switchware/sane_os/

[16]http ://nemesis.sourceforge.net/

1.5.2 Environnements d'exécution

Examen des EE candidats

Ayant choisi Linux comme base, les EE envisagés parmi les plus prometteurs étaient :

- *SmartPackets* [SZJ+99, SJS+00], développé par BBN, propose un environnement de programmation actif orienté gestion de réseaux et suivant l'approche intégrée. Les fonctions de gestion injectées sont programmées en *Sprocket* (un langage proche de C++ mais sans pointeur et avec des fonctions supplémentaires pour l'interaction avec des agents SNMP) puis compilées en un assembleur particulier (*Spanner*) ; elles doivent tenir dans une trame Ethernet et ne peuvent pas laisser d'information en mémoire sur un nœud. Nous avons exclu cet EE de nos tests à cause de l'absence de documentation sur l'assembleur spécifique utilisé, ainsi que l'absence d'application développée pour cet environnement.
- *Netscript* [YdS96], développé à l'Université de Columbia, est un environnement dans lequel la définition d'un service est réalisée en interconnectant des « boîtes » pré-installées ou à télécharger aux nœuds (approche discrète) ; le résultat de l'interconnexion est ensuite compilé en Java. Un aspect intéressant de *Netscript* est la capacité de programmer le réseau à différents niveaux (Ethernet, UDP, TCP, IP). Mais il est impossible de faire la distinction entre le traitement d'un flot de paquets spécifique, pour lequel un service a été tout spécialement défini, et le traitement des autres paquets passant par le nœud qui, même s'ils ne passent pas dans toutes les boîtes que l'on a interconnectées, doivent quand même être examinés pour être classés « non-actifs ». Malgré l'aide du développeur principal[17], nous n'avons pas pu isoler les ressources consommées par une application spécifique des ressources utilisées par les autres flots de paquets. Pour cette raison, nous avons du écarter *Netscript* de notre plate-forme d'expérimentation.
- *PLAN* (*Packet Language for Active Networks*) [HKM+98b], développé à l'Université de Pennsylvanie dans le projet SwitchWare, adopte une approche hybride entre les approches intégrée (du code écrit dans le langage restrictif de *PLAN* peut être transporté par les paquets) et discrète (il est possible d'appeler des services installés aux nœuds, les *switchlets*). Par mesure de sécurité, le langage conçu pour l'écriture des paquets est basé sur le CAML et le λ-calcul typé (cela permet de vérifier automatiquement certaines propriétés comme la non-duplication exponentielle des paquets) et est assez restrictif tant au niveau des opérations autorisées que de la quantité de ressources utilisables. Pour son fonctionnement, la plate-forme envoie régulièrement des paquets non directement liés aux paquets des utilisateurs (pour faire la correspondance entre une adresse *PLAN* et une adresse IP par exemple), et elle consomme aussi un certain nombre de cycles d'horloge en plus de ceux imputables à l'exécution des paquets d'un protocole particulier. Pour les besoins de notre projet, nous avions besoin de pouvoir isoler la consommation de ressources directement imputable à l'exécution d'un paquet « utilisateur » de celle causée par le fonctionnement « interne » de la plate-forme. Malheureusement, nous n'avons pu réussi à réaliser concrètement cette séparation (que ce soit dans la version Java

[17] Merci à Sushil Da Silva !

ou dans la version OCAML) ; c'est la raison pour laquelle nous n'avons finalement pas utilisé *PLAN*.
- *ASP* (*Active Signaling Protocol*) [BCF+99, BCF+01] est un environnement d'exécution actif développé à l'Université de Southern California. Il rappelle sur beaucoup de points l'environnement plus ancien *ANTS* mais adopte l'approche discrète car il vise le déploiement de protocoles de signalisation (dont le code est généralement assez volumineux). Cet EE n'est disponible que depuis un an et n'apparaît donc pas dans notre plate-forme expérimentale.
- *ANTS* (*Active Node Transfer System*) est certainement la plate-forme la plus populaire et nous avons pu l'utiliser pour nos expériences (dans la version disponible en 1999 : 1.2). Nous détaillerons ses principes ci-après. Plusieurs groupes de recherche utilisent et étendent cette plate-forme comme par exemple NAI Labs qui a amélioré sa sécurité dans *SANTS* [MLP+01].
- *Magician* est une plate-forme moins utilisée mais un contact ayant été établi entre son développeur principal (Amit Kulkarni) et le projet de recherche sur les réseaux actifs du NIST, nous avons également pu utiliser cet EE. Nous détaillons ci-dessous son fonctionnement.

Présentation d'*ANTS*

ANTS (*Active Node Transfer System*) ([WLG98, Van97, Wet97, WGT98, Wet99b, Wet99a, WGT99]) est une des premières plates-formes actives, développée au MIT principalement par Wetherall et Tennenhouse en prolongement de leurs travaux sur l'option active d'IP.

Chaque nœud actif est une instance d'une classe *Node* s'exécutant dans une machine virtuelle Java. La description de la topologie du réseau doit lui être fournie au démarrage. Le nœud se connecte ensuite au réseau en instanciant un *Channel*. Dans nos expériences, nous avons utilisé la sous-classe *UDPChannel* permettant à nos nœuds d'écouter un port UDP spécifié au démarrage.

Lorsqu'un paquet arrive au port spécifié, le nœud commence par vérifier qu'il respecte le format d'une capsule[18] *ANTS*, illustré par la figure 1.6. Les champs *adresse source*, *adresse destination* et *TTL* sont semblables à ceux d'IP. Le champ *version* identifie la version d'*ANTS* utilisée dans l'encapsulation. Le champ *type* précise la fonction de traitement à appliquer aux données transportées par la capsule. Le champ *adresse précédente* indique l'adresse du dernier nœud traversé. Le champ *entête* contient des sous-champs dont la taille et le nombre varie selon le *type* de la capsule. Le reste de la capsule constitue les *données*.

Si la capsule reçue ne respecte pas le format *ANTS*, elle est tout simplement mise au rebut. Si le nœud reconnaît une capsule *ANTS*, il examine le *type* pour déterminer quel traitement doit être appliqué aux *données*. S'il a déjà dans son *cache* le code désigné par le *type*, il passe la capsule à un *ChannelThread* qu'il a instancié et qui va exécuter la méthode *evaluate* de la capsule. Sinon, il passe au *ChannelThread* une capsule *DLBootstrapCapsule* qui provoque la demande au nœud précédent (dont l'adresse est précisée dans l'entête

[18] Dans cette plate-forme, « capsule » = « paquet actif ».

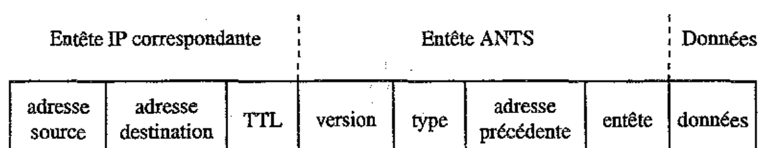


FIG. 1.6 – Format d'une capsule *ANTS*

ANTS du paquet) du code nécessaire. En réponse, le nœud précédent va renvoyer une capsule (ou plusieurs, selon la taille du code) contenant le *byte code* demandé (placé dans la partie *données* de la capsule). Le nœud que nous examinons va alors « charger » le code (grâce à une extension de la classe *ClassLoader* de chargement de code du JDK) et pourra traiter la capsule de données. Il garde le code dans son *cache* de façon à pouvoir traiter d'autres capsules du même type et pouvoir répondre à des requêtes de demande de code des nœuds en aval. Tous ces traitements s'effectuent dans la machine virtuelle Java dans laquelle le nœud a été démarré.

Pour les capsules dont le code de traitement ne redéfinit pas le routage, un routage par défaut est employé.

Présentation de *Magician*

Magician [KMH+98, BK01] est une plate-forme dont le développement a débuté en 1996 à l'Université du Kansas. Bien que moins connue qu'*ANTS*, cette plate-forme nous a initialement attirés à cause de l'attention qu'elle semblait porter à la gestion et à l'allocation des ressources (malheuresement il s'avère que les mécanismes envisagés ne sont pas effectivement implémentés...).

Chaque nœud est une instance de la classe *ActiveNode* s'exécutant dans une machine virtuelle Java. À son démarrage, un nœud actif va chercher la topologie du réseau qui doit avoir été préalablement déposée sur un serveur. *Magician* prévoit que plusieurs environnements d'exécution pourront exister sur le nœud. Il démarre en particulier le cœur de l'environnement *Magician* (*Node Manager*) qui à son tour lance :

- un module de gestion des ressources, qui devrait permettre de gérer les ressources du nœud *Magician* lorsque les mécanismes envisagés dans la documentation seront effectivement implémentés (nous apporterons une contribution sur ce point au chapitre 6),
- un module de journalisation des événements réseau, permettant de renseigner une base de données de gestion (*MIB*),
- un module de routage fournissant une interface pour installer de nouveaux protocoles de routage ou pour utiliser le routage par défaut,
- une interface de communication sur chacun des ports UDP sur lesquels le nœud *Magician* est configuré pour envoyer ou recevoir des paquets.

La figure 1.7 résume le format des paquets échangés par *Magician* (appelés *SmartPackets*).

Lorsqu'un paquet actif est reçu et passé au *Node Manager* (paquet dont l'en-tête

en-tête ANEP
adresse source
adresse destination
information d'authentification
classe Java
objet sérialisé
type

FIG. 1.7 – Format d'un paquet *Magician*

ANEP[19] indique qu'il s'agit d'un *SmartPacket*), l'interface de communication *Magician* concernée examine le champ *type*. Si c'est la première fois que le nœud reçoit un paquet de ce type, il va « charger » le *byte code* contenu dans la partie *classe Java* du *SmartPacket*. L'*objet sérialisé* est ensuite désérialisé. Il est prévu qu'il contienne des informations pour la gestion des ressources, comme par exemple son temps d'exécution maximal. Un *KUSmartPacket* est alors créé à partir de cet objet et est traité par la classe *SPExecEnv* conformément au code préalablement chargé. Un *thread* Java est créé par *SmartPacket*.

La sérialisation étant une opération coûteuse, quand un *SmartPacket* est reçu, un *hash* est calculé sur l'objet extrait et l'objet est placé dans un *cache*; ce *hash* est utilisé quand ce même *SmartPacket* est prêt à être transmis pour déterminer si l'ensemble de l'objet doit être re-sérialisé (si rien n'a changé dans l'objet, il est plus rapide de simplement envoyer l'objet placé dans le *cache*).

1.5.3 Applications actives

Les distributions de ces EE fournissent certaines applications, comme par exemple :

- *ping* (pour *ANTS* et pour *Magician*), qui envoie un message à un nœud distant donné et attend une réponse pour vérifier si le nœud est accessible ou pas,
- *route* (pour *Magician*), semblable à *ping* mais qui enregistre en plus la route suivie et/ou le temps d'aller-retour,
- *multicast* (pour *ANTS*), qui permet à un nœud serveur d'envoyer des messages à tous les nœuds s'abonnant à ce service (la durée de l'abonnement est à préciser, et une fois cette durée écoulée, un nœud doit renouveller sa demande s'il veut continuer à recevoir les messages du serveur).

L'annexe A résume le fonctionnement de ces applications.

Bien que relativement simples, ces applications font appel aux fonctions de base des EE qui seront vraisemblablement appelées par toutes les applications actives : envoi et réception de paquet, routage, consultation et modification de l'état du nœud...

Lors d'une expérience d'intégration de nos travaux à un contrôleur de nœud (voir chapitre 6), nous avons également utilisé une application de transport de flux audio

[19] *Active Network Encapsulation Protocol*, voir paragraphe 1.6

développée à l'Université du Kansas pour *Magician* : elle prend à la source un fichier audio, le découpe et envoie peu à peu l'information vers la destination grâce à des paquets actifs pouvant être compressés si nécessaire aux nœuds intermédiaires, la destination décompressant et ré-assemblant le tout avant de le jouer.

1.6 Débuts de standardisation

Même si le but des réseaux actifs est de venir à bout des processus de standardisation, certains accords ont été conclus entre différents acteurs de la recherche sur les réseaux actifs (notamment ceux financés par DARPA) afin de favoriser l'échange d'idées et faciliter les tests à grande échelle.

Tout d'abord, les documents d'architecture déjà évoqués offrent une vision commune d'un nœud actif, ce qui constitue une référence très utile lors de discussions entre différents groupes de recherche.

Un réseau actif expérimental, appelé ABone (*Active network backbone*), a été mis en place pour permettre le déploiement et le test à assez grande échelle de composants actifs. Il regroupe en réseau virtuel (tunnel IPv4) une centaine de nœuds du monde entier (dont un au LORIA et 4 au NIST). [BBR00] décrit en détail son fonctionnement. L'ABone est constitué de nœuds « noyaux » stables mettant à disposition des développeurs un ensemble d'EE standard[20] (plus éventuellement d'autres). Les nœuds « feuilles » sont des nœuds « privés », ce qu'on y trouve et ce qu'on peut y faire n'est pas régulé et doit être négocié avec le propriétaire du nœud. La plupart des expériences décrites plus loin dans ce document ont été effectuées sur les 4 nœuds du ABone hébergés au NIST.

Un centre de coordination (ABOCC = *ABOne Coordination Center*) enregistre et gère les différents acteurs (développeurs d'EE et d'AA, nœuds actifs, serveur de code actif). Les nœuds sont administrés localement, mais les EE sont installés à distance par des particuliers ou par l'ABOCC en utilisant ANetD[21] (*Active NETwork Daemon*). L'ABOCC maintient (grâce à QCMD[22]) sur chaque nœud des listes indiquant qui a le droit d'installer ou modifier des EE et de quels serveurs il est considéré comme sûr de charger le code d'un EE.

ANetD a un rôle supplémentaire sur les nœuds actifs, il fournit une partie des fonctionnalités que l'on attend d'un système d'exploitation actif : il permet de démultiplexer un paquet actif entrant à l'EE approprié en se basant sur le type contenu dans l'entête ANEP. ANEP[23] (*Active Network Encapsulation Protocol*) définit un format commun pour les paquets actifs. L'ANANA (*Active Network Assigned Number Authority*) accorde un type aux organisations intéressées.

[20] Ces EE standard comprennent pour l'instant *ANTS*, *ASP* et *PLAN*.

[21] ANetD permet d'échanger des fichiers avec un nœud actif distant, de l'interroger pour connaître la liste des services qu'il propose, de lui demander de télécharger un nouveau service ou de tuer un service en place. http ://www.sdl.sri.com/projects/activate/anetd/

[22] *Query Certificate Manager Daemon*, infrastructure à clé publique permettant la maintenance sécurisée d'ensembles de données distribuées.

[23] http ://www.cis.upenn.edu/~switchware/ANEP/

1.7 Problèmes

Nous avons mis en évidence que les réseaux actifs comportent de nombreux avantages par rapport aux réseaux classiques. Mais le fonctionnement que nous avons décrit, et qui leur confère la flexibilité appréciée, est également source d'une complexité de gestion et de sécurisation accrue. Cette section cite quelques uns des principaux défis qu'il reste à relever.

1.7.1 Changements dynamiques de configuration

Les fonctions présentes à un nœud actif peuvent être modifiées ou supprimées dynamiquement, de nouvelles fonctions peuvent apparaître, et le modèle de gestion du nœud doit évoluer pour refléter ces modifications. Par exemple, si un nouveau protocole est déployé pour le routage de paquets MPEG, une variable comptant le nombre de paquets MPEG supprimés au nœud sera certainement ajoutée à la MIB (*Management Information Base*) de ce nœud si l'on suit la méthode traditionnelle de gestion. Mais cela obligera dans la plupart des cas à re-compiler l'agent SNMP (*Simple Network Management Protocol*, RFC 1157) associé (ce qui suppose commencer par l'arrêter et qui peut être gênant s'il est en train de collecter des données intéressantes).

1.7.2 Sécurité

Dans un réseau traditionnel, les fonctions présentes aux nœuds ont été installées par le fournisseur de services auquel l'administrateur des nœuds fait confiance. Dans certains réseaux actifs, les utilisateurs aussi pourront installer, modifier ou supprimer des fonctionnalités aux nœuds. Il est évident que dans ce contexte l'exploitant du réseau voudra contrôler qui a le droit de modifier quoi sur ses nœuds. Un nœud pourrait se baser sur l'identité de l'émetteur d'un paquet pour autoriser ou non son exécution. Des mécanismes d'authentification cryptographique supplémentaires doivent donc être mis en place, et ils ne doivent pas pénaliser les performances du réseau (vérifier un certificat d'authentification peut être long et consommer beaucoup de ressources, parfois plus qu'un paquet malveillant !).

Les mécanismes de sécurité doivent être forts mais suffisament légers (en temps, calcul, nombre de messages...) pour que le réseau actif reste simple et rapide à utiliser.

1.7.3 Besoins accrus de retour d'information pour l'utilisateur

Dans un réseau traditionnel, l'utilisateur n'a que très peu de contrôle sur ce qui se passe dans le réseau et n'exige donc pas beaucoup de retour d'information (*feed-back*). Dans les réseaux actifs en revanche, l'utilisateur veut qu'on lui fournisse des outils de mise au point et des informations sur l'exécution de ses programmes. Comment détecter et signaler les erreurs d'exécution des paquets ? De plus, comme l'utilisateur peut écrire ses propres fonctions de gestion ou utiliser des informations de gestion pour améliorer le fonctionnement de son application active, il a également besoin qu'on lui fournisse des informations relatives à la configuration du nœud par exemple.

1.7.4 Gestion des ressources plus cruciale

Dans un réseau actif, plusieurs protocoles peuvent être utilisés simultanément sur un même nœud. Une des tâches de gestion va être de s'assurer que le protocole défini par un utilisateur n'interfère pas avec un autre. Les ressources doivent être partagées équitablement. Mais le nœud doit tout d'abord se protéger lui-même contre la mauvaise utilisation de ses ressources ou les accès non autorisés. Même lorsqu'une liste d'accès existe pour décrire qui a le droit de faire quoi sur quelle ressource, il faut encore contrôler que l'utilisateur ne dépasse pas les bornes. Une des contributions principales de cette thèse concerne ce problème.

En conclusion, on peut dire que les réseaux actifs sont aux réseaux ce que l'architecture RISC a été à l'architecture des processeurs[24] : ils permettent de faire des choses plus complexes mais l'allocation de ressources et l'ordonnancement deviennent plus difficiles, tout comme la puissance des RISC se réalise aux dépends d'un langage d'assemblage plus difficile à programmer.

1.8 Résumé

Dans ce chapitre nous avons montré que le concept de réseau actif permet d'apporter une flexibilité enthousiasmante au niveau des services réseau en introduisant la possibilité d'installer et choisir un traitement « sur-mesure » des paquets aux nœuds d'un réseau. Nous avons expliqué les concepts de fonctionnement des réseaux actifs et présenté les plates-formes utilisées dans le travail décrit dans la suite de ce document. Nous avons également mis en évidence les problèmes qu'il sera nécessaire de résoudre avant leur utilisation industrielle, notamment des problèmes de gestion et sûreté, dont nous nous proposons de tenter de résoudre une partie dans cette thèse.

[24]L'architecture RISC a détrôné les systèmes CISC en proposant une technique permettant une parallélisation des tâches entraînant une amélioration des performances mais complexifiant le compilateur.

S.C.D. - U.H.P. NANCY 1
BIBLIOTHEQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

Chapitre 2

Contrôle de l'utilisation des ressources CPU

Sommaire

2.1 Introduction 33
2.2 Nécessité de contrôler la consommation 33
2.3 Sources de variation des besoins en calcul 34
2.4 Solutions existantes pour limiter la consommation . . . 36
2.5 Autres sources d'inspiration pour quantifier les besoins 40
2.6 Résumé 42

2.1 Introduction

Comme on vient de le voir, l'ajout « d'activité » dans les réseaux entraîne l'apparition de nouveaux problèmes de gestion. Dans ce chapitre, nous examinons plus spécialement l'impact qu'a l'introduction de la programmabilité sur la consommation des ressources de calcul du réseau.

Nous commencerons par identifier les menaces de surconsommation des ressources dans les réseaux actifs qui justifient le besoin de contrôle. Ensuite nous dégagerons les sources de variabilité des besoins en calcul. Puis nous étudierons comment le contrôle est actuellement réalisé dans des plates-formes actives ou dans des plates-formes à agents mobiles et pourquoi les solutions existantes ne sont pas entièrement satisfaisantes. Nous ferons également l'inventaire d'autres propositions existantes qui pourraient nous aider à quantifier les besoins en ressources de calcul d'un programme.

2.2 Nécessité de contrôler la consommation

Dans un réseau traditionnel, tous les paquets reçoivent un traitement identique comme le rappelle la figure 1.1 page 15, sans distinction selon l'application auxquels

ils appartiennent et sans différence d'un nœud à l'autre[25]. Le traitement des paquets sur un nœud donné requiert donc toujours la même quantité de calcul (à peu de choses près[26]). L'ordonnancement est donc simple à réaliser et il suffit de contrôler la bande passante d'une application pour limiter sa consommation en ressources de calcul à un nœud. Dans un réseau actif en revanche chaque paquet peut recevoir un traitement spécifique, et ce traitement peut être différent d'un nœud à un autre. Par conséquent la quantité de ressources de calcul nécessaire au traitement d'un paquet varie beaucoup d'une application à une autre et d'un nœud à un autre.

Sans contrôle de la consommation des ressources CPU utilisées, une application active peut présenter 3 types de menaces pour le réseau actif :

- Un paquet peut consommer une quantité grande, voire infinie, de ressources sur un nœud particulier; cette consommation locale excessive peut amener le nœud à refuser ses services à d'autres paquets[27].
- Un paquet (plus ceux qu'il peut éventuellement créer) peut consommer une quantité grande, voire infinie, de ressources sur un ensemble de nœuds (boucle de routage par exemple).
- Une application peut envoyer un nombre grand, voire infini, de paquets au travers du réseau ou d'une portion du réseau.

Ces menaces peuvent venir d'applications volontairement malveillantes ou simplement d'une erreur dans un programme.

Il est donc nécessaire de pouvoir exprimer les besoins en calcul d'une application afin qu'un nœud puisse décider d'accepter de l'exécuter ou non et puisse décider de stopper une exécution qui dépasse les limites du fonctionnement indiqué comme normal pour l'application.

Prévoir la consommation d'une application sur un nœud de réseau hétérogène présente d'autres avantages (précisés page 45) mais le problème de surconsommation malveillante des ressources de calcul reste pour nous le premier mal à traiter.

2.3 Sources de variation des besoins en calcul

L'analyse du modèle d'un nœud actif (présenté page 20) et d'évaluateurs de performances classiques (*benchmarks*) révèle les sources majeures de variabilité affectant les besoins en temps CPU d'une application active. Nous allons rapidement les passer en revue.

[25] À part dans les réseaux garantissant une certaine qualité de service (*QoS*), comme les réseaux *Diffserv* qui classifient le trafic et éliminent les paquets les moins prioritaires en cas de congestion. Ou bien dans les routeurs CISCO par exemple, où il est possible de diminuer la priorité de certaines applications comme *ftp*.

[26] Même si le coût de recherche dans une table de routage n'est pas constant, il n'est pas lié à l'application.

[27] Dans cette attaque dite "refus de service" (*denial of service*) l'attaquant n'a pas de bénéfice direct mais tente de dégrader les services reçus par les autres utilisateurs du nœud ou du réseau.

Le matériel

Au niveau matériel, de nombreux facteurs peuvent influencer le temps d'exécution d'une application. Les principaux sont la fréquence du processeur, l'architecture du processeur (CISC, RISC, CRISC[28]), la quantité de mémoire vive disponible, la vitesse des différents bus (mémoire, entrée/sortie, système), la technologie employée pour le stockage permanent (disque dur ou bande par exemple), les cartes réseaux[29]...

Le système d'exploitation (OS)

Le temps exécution est fonction des algorithmes choisis lors de l'écriture des pilotes d'équipements et de l'implantation des appels système du système d'exploitation de la machine. Les choix réalisés lors du recouvrement du système d'exploitation sous-jacent par des fonctions standard du nœud actif (*mapping*) peuvent également influencer les performances de l'application.

L'environnement d'exécution

Les différents EE proposent des fonctions différentes aux applications. Et même lorsque des fonctions identiques sont proposées, elles sont généralement implantées différemment. Ainsi un « envoi de paquet » avec *ANTS* aura sûrement des besoins différents d'un « envoi de paquet » avec *PLAN*. De plus, pour un EE donné, des variations peuvent également apparaître selon la façon dont la machine virtuelle sous-jacente (JVM par exemple) est réalisée sur les fonctions standard du nœud actif. Par exemple le compilateur et les bibliothèques C utilisés pour compiler la JVM supportant *ANTS* influencent les besoins en calcul d'une application *ANTS*.

Un environnement d'exécution pourrait choisir d'enclencher un mécanisme de sécurité supplémentaire selon la provenance des paquets actifs. Les traitements additionnels engendrés augmenteraient le nombre de cycles d'horloge nécessaires pour réaliser le traitement de l'application.

L'application

Selon le cheminement qu'elle suivra dans son code, une application aura des besoins en calcul variables. Le chemin suivi peut dépendre de beaucoup de paramètres comme par exemple l'heure, l'état du nœud d'exécution (par exemple la présence ou l'absence d'une certaine valeur dans un *cache*) ou d'autres nœuds (par exemple dans l'application

[28]Les systèmes CISC (Motorolla 68000, famille x86 d'Intel, VAX...) utilisent des jeux d'instructions de longueur variables demandant souvent plus d'un cycle d'horloge pour s'exécuter. Les systèmes RISC (PowerPC de Motorola, Alpha de Digital...) offrent au contraire des instructions de longueur courte et fixe pouvant la plupart du temps se dérouler en un cycle d'horloge et en même temps que d'autres opérations, améliorant ainsi les performances par rapport aux systèmes CISC (en contrepartie le compilateur est plus complexe). Les PentiumPro et PentiumII sont des hybrides entre les 2 architectures RISC et CISC.

[29]Ethernet a beau être un standard, le choix de la carte d'interface peut influencer les performances du système, selon les techniques choisies par le constructeur pour gérer l'utilisation du bus PCI par exemple [Ste01], ou le support effectif d'un mode *full-duplex*.

de multicast d'*ANTS* une capsule doit être créée et envoyée pour chaque nœud ayant souscrit au multicast), la longueur des données à traiter...

La liste (éventuellement infinie) des valeurs que peuvent prendre les quelques paramètres que nous venons de citer (et ils ne sont certainement pas les seuls influents le temps exécution !) est bien trop longue pour que l'on puisse espérer les capturer toutes et les combiner en une formule analytique donnant une estimation *a priori* du temps d'exécution.

2.4 Solutions existantes pour limiter la consommation

Soucieuse de protéger les réseaux actifs des applications consommant plus de ressources que raisonnable, chaque plate-forme s'est équipée d'un mécanisme limitant l'utilisation de la CPU. Ces mécanismes s'inspirent directement de ce qui existe dans les réseaux traditionnels ou pour les plates-formes à agents mobiles.

2.4.1 Limites quantitatives fixes

Limite fixée à chaque nœud

Pour éviter qu'à un nœud donné une application ne consomme trop de ressources, *ANTS* prévoit un mécanisme de « borne » ou seuil : chaque nœud fixe une limite pour la consommation de CPU pour chaque paquet[30]. Si un paquet ne parvient pas à terminer son exécution avant d'atteindre cette limite, il est détruit. Ce mécanisme est proche de celui des quotas que les systèmes d'exploitation peuvent imposer au niveau de la taille des fichiers et répertoires : une fois qu'un utilisateur a dépassé son quota, il ne peut plus rien écrire sur le disque.

Si cette technique évite le pire en assurant la sûreté du nœud, des scénarios peuvent être imaginés où cette solution ne permet pas une gestion optimale car elle traite toutes les applications de la même façon, sans distinction. Imaginons qu'un nœud propose 50 unités CPU et qu'il fixe la limite utilisable par chaque paquet à 10. (L'environnement d'exécution peut être lui-même soumis à un quota et on fait l'hypothèse qu'il admet au plus 10 paquets simultanément.)

Dans un premier scénario on peut imaginer qu'un seul paquet demande à être traité mais qu'il lui faille 12 unités pour terminer son exécution. Il sera détruit avant d'avoir fini son travail alors que les ressources nécessaires à son exécution étaient largement disponibles.

Dans un autre scénario, imaginons que 10 paquets nécessitant tous 6 unités pour s'exécuter demandent à être traités et que le nœud ordonnance leur exécution tour à tour à raison de 3 unités de temps à chaque fois. Arrivé au 2ème passage du 7ème paquet, les 6 premiers paquets auront terminé leur exécution mais les 50 unités seront épuisées et les 4 autres paquets seront détruits sans avoir terminé mais après avoir tout de même consommé des ressources (voir figure 2.1). Si les paquets avaient pu exprimer que

[30]En réalité, c'est le temps d'exécution apparent (*wall clock time*) qui est limité mais bien sûr cela borne automatiquement la consommation de CPU.

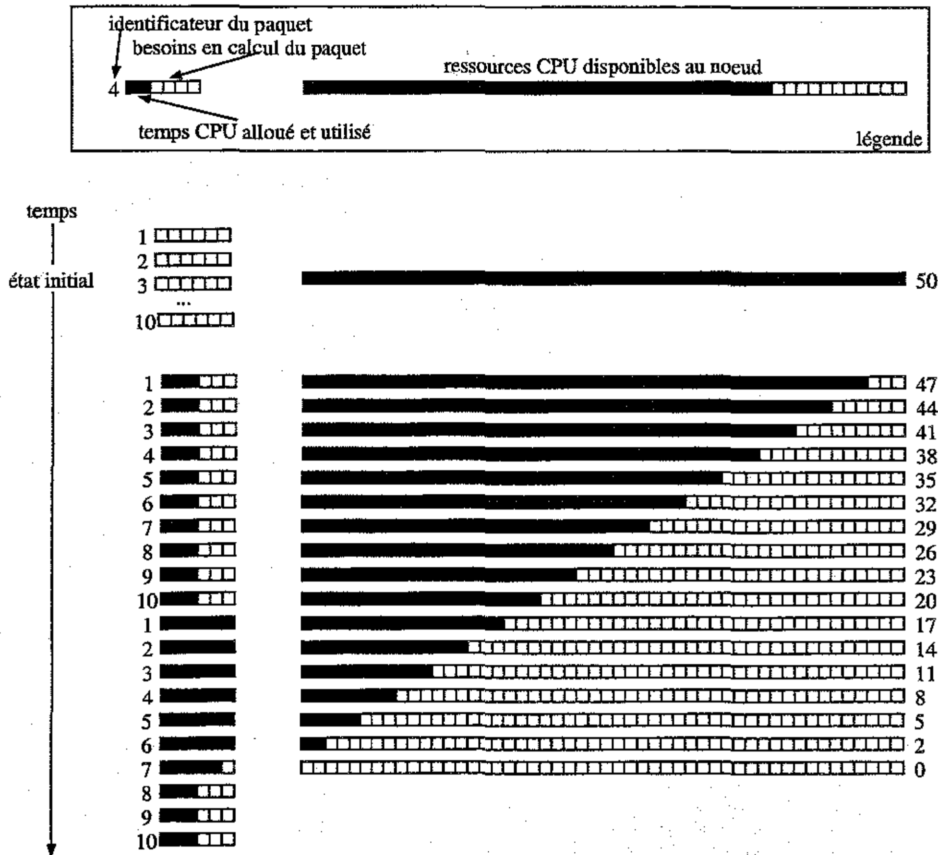


FIG. 2.1 – Allocation de ressources non renseignée

leur besoin était de 6 unités, le nœud aurait pu décider d'un ordonnancement différent et seuls 2 paquets n'auraient pas pu s'exécuter (figure 2.2).

D'autre part, utilisée seule, cette solution ne permet pas de régler le problème du gaspillage des ressources globales du réseau.

Limite fixée par paquet

Pour réduire la menace des applications tournant infiniment sur une boucle du réseau, certaines plates-formes (*ANTS* par exemple) utilisent un deuxième mécanisme : elles associent un crédit (semblable au *Time-To-Live* d'IP ou *Hop Count* d'IPv6) à chaque paquet actif. Ce crédit est diminué à chaque saut, de plus, chaque fois qu'un paquet crée des fils, la somme des crédits des fils et du crédit du père ne doit pas dépasser le crédit restant au père avant la création des fils. Lorsque le crédit d'un paquet est épuisé, il est détruit.

L'avantage de cette solution par rapport à la première (limite fixée par le nœud) est qu'elle limite le gaspillage des ressources globales du réseau par un paquet malintentionné ou erroné tournant sur une boucle de routage. Cependant, il faudra quand même un certain temps avant qu'un paquet faisant l'aller-retour entre deux nœuds soit stoppé.

Cette approche doit être couplée à la précédente sinon un paquet ne se déplaçant

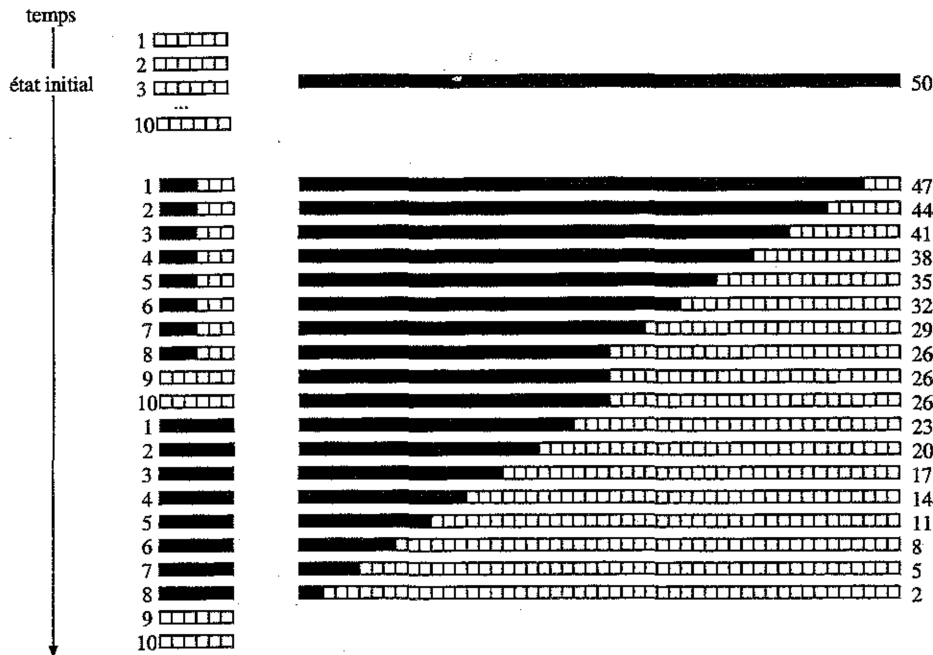


FIG. 2.2 – Allocation de ressources optimale

pas pourrait profiter de son crédit ne diminuant pas pour consommer énormément de ressources à un nœud donné.

Comme dans la méthode des bornes de nœud, le principal problème est de choisir judicieusement la valeur initiale du crédit. Pour IP, la valeur actuellement recommandée est de 64 sauts ([RP94]), *Windows 98-2000-NT* utilise 128. Cette valeur a augmenté et continuera peut-être d'évoluer au fur à mesure que le réseau se densifie. Mais même en admettant que cette valeur suffira également pour un paquet actif propageant une configuration entre deux machines en vidéoconférence par exemple, elle sera certainement très vite insuffisante si le paquet crée des fils (à qui il doit céder une partie de ses propres crédits). Mais il est en général difficile de prévoir combien de fils devront être créés. Cela dépend peut-être du nombre de branches dans le réseau (par exemple le père explore une branche et pour s'occuper de chaque branche restante il crée un fils) ou d'autres paramètres du réseau rarement connus à l'avance. [HT97] propose de donner la possibilité aux paquets qui n'ont pas terminé leur mission de demander des crédits supplémentaires lorsqu'ils sont presque à zéro. La décision de les leur accorder est faite (par l'application initiale pour le père ou par le père pour ses fils) sur examen d'un rapport d'avancement qu'ils joignent à leur demande. Même si cette solution est séduisante, la difficulté d'émettre ces rapports et surtout de les juger est certainement la cause de leur absence d'implantation. De plus, cette solution ne règle pas le problème des applications malveillantes.

Moralité : les solutions "taille-unique" ne sont pas entièrement satisfaisantes

Pour résumer, les deux approches les plus courantes pour contrôler les ressources CPU dans les réseaux actifs consistent à fixer une limite au temps CPU utilisable par un paquet actif. Dans une approche chaque nœud applique sa propre limite à chaque paquet alors que dans l'autre c'est le paquet qui transporte sa propre limite, limite pouvant être insuffisante pour s'exécuter sur certains nœuds et exagérément généreuse sur d'autres. Certes, ces solutions permettent en général d'assurer la sécurité du nœud en évitant les pires cas. Mais elles ne permettent pas, en revanche, d'effectuer une gestion optimale des ressources de calcul. Pire, elles peuvent conduire un nœud à stopper prématurément l'exécution d'un paquet correctement programmé, et mal servir une application pourtant « pacifique ». L'inefficacité de ces solutions à base de « bornes » provient principalement du fait que les limites sont fixées arbitrairement, en l'absence de connaissance des besoins de calcul des paquets.

2.4.2 Restrictions qualitatives

Une autre solution proposée pour protéger les plates-formes est de limiter qualitativement les exécutions : ne pas autoriser toutes les opérations, ou pas pour le monde. N'autoriser que l'exécution du code venant de clients de confiance (utilisation de mécanismes d'authentification et de privilèges pour l'accès à certains services) limite les possibilités d'attaque mais (1) cela n'exclut pas les problèmes de consommation excessive due à une erreur de programmation, (2) la seule connaissance de l'identité du client ne permet pas au nœud d'ordonnancer ses ressources au mieux et (3) cette solution restreint beaucoup des possibilités intéressantes des réseaux actifs. *PLAN* [HKM+98a] règle le premier problème en utilisant un langage de programmation restreint et des preuves formelles pour assurer que le code n'effectue pas d'opérations fondamentalement interdites. Le langage *SNAP* [MHN01] a été conçu pour que les ressources de calcul utilisées pour le traitement d'un paquet soient proportionnelles à la taille du paquet. Bien que cette approche fournisse un contrôle sur la quantité de cycles CPU utilisés, le langage est trop restrictif pour pouvoir représenter toutes les applications actives. Par exemple, seuls les branchements en avant sont autorisés; par conséquent, si une boucle est nécessaire elle doit être « déroulée » (linéarisée ou dépliée en « copiant-collant » les actions de la boucle autant de fois que nécessaire, à condition de connaître ce nombre à l'avance) ou le paquet doit se renvoyer lui-même sur le même nœud jusqu'à ce que la tâche soit accomplie, ce qui n'est pas très efficace. Dans la même catégorie de travail, on peut aussi citer *SafetyNet* [WJGO98] qui développe un langage typé (inspiré du π-calcul) pour programmer les réseaux actifs. Il est dit que les informations de typage permettent de calculer la borne maximale des ressources consommées par un paquet actif mais le modèle semble rester très abstrait et je n'ai pas pu identifier quelles sont les sources de variabilité qu'il permet de capturer parmi celles que nous avons citées.

2.4.3 Acquisition des ressources « aux enchères »

[YL00] présente un modèle pour la négociation des ressources d'un nœud entre un agent utilisateur compris dans le paquet actif et un agent gestionnaire d'une ressource du nœud, présent sur le nœud. L'agent gestionnaire propose sa ressource (qui peut être de la bande passante, de la mémoire ou des cycles CPU) aux agents utilisateurs à un prix qui varie en fonction de la demande (plus la demande est élevée, plus la ressource est chère). Un paquet actif possède un budget qui lui permet d'acheter les ressources des nœuds. En fonction du prix affiché pour la ressource qui l'intéresse, de son crédit et de l'importance de sa tâche, un agent utilisateur décide du traitement à appliquer au paquet qui le transporte : par exemple, si la ressource CPU est très demandée par les paquets concurrents, et donc coûteuse à utiliser, l'agent utilisateur peut décider d'appliquer un algorithme de compression sur ses données plus simple que celui qu'il avait initialement envisagé, qui était plus efficace mais aussi plus cher. Le problème de cette approche est là encore de fixer le budget initial alloué à un paquet. De plus, elle complexifie considérablement le fonctionnement des paquets actifs (il ne faudrait pas que le code dédié à l'acquisition des ressources soit trop coûteux !). Dès 1995 le projet *Messengers* [MMTH95] proposait un mécanisme similaire, à part que les crédits des paquets étaient re-initialisés à chaque nœud à une valeur choisie par le nœud. Ce mécanisme était donc équivalent au système de borne critiqué plus haut.

2.5 Autres sources d'inspiration pour quantifier les besoins

Nous n'avons pas connaissance d'autres projets cherchant à quantifier les besoins en ressources CPU d'une application active dans un réseau hétérogène mais nous avons examiné d'autres travaux pouvant nous aider à concevoir une solution efficace.

2.5.1 Applets java

Les applets Java posent des problèmes localement similaires : elles ne viennent pas toujours de sources de confiance mais s'exécutent pourtant en local, sur notre machine. La solution qui a été adoptée pour éviter le pire est de restreindre leurs droits d'accès au système (lancement d'autres programmes locaux (par *fork* et *exec*) interdits par exemple), limitant leur exécution dans un « bac-à-sable » aux politiques de sécurité configurables. Toutefois on a vu que certains intérêts des réseaux actifs découlaient de leur capacité à modifier l'état des nœuds. Les restrictions appliquées aux applets ne sont donc pas forcément les bienvenues dans ce contexte.

2.5.2 Cycles RISC

Les documents d'architecture n'abordent pas les problèmes d'allocation des ressources de bout en bout. En revanche, ils définissent qu'il est de la responsabilité du système d'exploitation actif d'allouer et d'ordonnancer les ressources du nœud, en particulier les

cycles CPU. [Cal99] souligne bien que le problème est de quantifier les besoins en calcul d'une application dans un contexte où ces besoins varient beaucoup d'un nœud à un autre.

Il a alors été suggéré d'utiliser les cycles RISC comme unité de mesure des besoins en calcul, mais sans répondre aux questions importantes qui se posent alors :

- étant donnée une application active, comment évaluer le nombre de cycles RISC qu'un paquet va demander ?
- comment convertir les cycles RISC de et vers les unités de mesure de calcul des autres machines ?

2.5.3 Caractérisations combinées du programme et de la machine

[SBSM89] présente des travaux effectués en 1989 visant à créer une caractérisation des performances d'une machine qui puisse être utilisée pour prévoir le temps d'exécution d'un programme donné : chaque machine est définie par un vecteur indiquant le temps CPU utilisé pour exécuter 102 opérations Fortran bien définies. Par ailleurs, un programme Fortran peut être analysé et ramené à cet ensemble d'opérations clés. En combinant les modèles de la machine et du programme, il est alors possible de prévoir le temps d'exécution du programme sur cette machine. Les bons résultats rapportés nous ont encouragé à modéliser les applications séparément des plates-formes. L'avantage du Fortran est que le nombre d'opérations est réduit et que pour la plupart d'entre elles, le temps exécution ne dépend pas de la valeur des arguments ; mais dans le cas des réseaux actifs, se limiter à Fortran est problématique ! Et surtout, seul un chemin dans le code du programme était examiné dans ce projet alors que nous voulons couvrir au mieux l'ensemble des comportements des applications.

2.5.4 Examen des chemins acycliques

Pour mesurer, expliquer ou améliorer les performances d'un programme, une technique régulièrement utilisée est la collecte d'informations de profil résumant combien de fois lors de l'exécution chaque instruction a été exécutée. Compacte et peu coûteuse à collecter, cette information peut servir à identifier des portions de code fréquemment exécutées. En revanche elle ne livre pas de détail sur le fonctionnement dynamique du programme, par exemple, elle ne capture pas les itérations. La solution alors utilisée est la production d'une trace de l'exécution, listant les instructions effectuées dans l'ordre d'exécution. Mais, plus le programme tourne longtemps, plus la trace est longue et lourde à manipuler. [BL00] propose une solution intermédiaire consistant à lister tous les chemins sans boucle qui apparaissent lors de l'exécution du programme avec leur nombre d'occurrences. Les auteurs montrent, entre autres, comment l'utilisation de ces chemins acycliques peut améliorer les performances des prédicteurs de branchement. Nous envisageons de nous inspirer de ces travaux pour améliorer le modèle que nous proposons plus loin, en particulier pour capturer efficacement le comportement des applications effectuant beaucoup de répétitions d'un bloc d'opérations. Mais cette approche nécessite

d'instrumenter le code de tout programme que l'on veut tracer et c'est la raison pour laquelle nous l'avons dans un premier temps écartée.

2.5.5 Informations grossières fournies par l'utilisateur

Dans le projet AppLeS (*application-level scheduling*) [BWF+96], l'utilisateur doit fournir des informations sur l'application qu'il veut exécuter sur le système distribué. Ces informations sont utilisées pour choisir l'ordonnancement supposé conduire aux performances maximales dans le cas proposé. Le développeur doit par exemple indiquer si l'application est plutôt "orientée communication" ou "orientée calcul" ou encore "équilibrée", le type de communication (point-à-point, multicast...), le nombre de MFlops par structure de données, etc. Cette méthode ne conduit à des prédictions intéressantes que si l'utilisateur est à la fois capable et désireux de fournir les caractéristiques requises de son programme. Mais des discussions avec des spécialistes de la modélisation de performances logicielles laissent à penser que c'est un cas rare !

2.5.6 Placement de tâches tenant compte de la charge

En parallélisme, on cherche parfois à déterminer la manière d'exécuter des tâches sur les différents nœuds ou processeurs du système minimisant le temps d'exécution de ces tâches. Les algorithmes de placement dynamique cherchent à atteindre ce but en affectant les nouvelles tâches aux machines (ou processeurs) les plus adaptés ou en migrant les tâches qui s'exécutent déjà. Cela implique d'une part de déterminer l'état courant d'un système (généralement, c'est la longueur de la file d'attente des processeurs actifs qui sert de descripteur de la charge) mais surtout d'anticiper la charge d'une machine après le placement ou la migration sur cette dernière. Pour cette estimation de la charge induite par le placement d'une tâche, l'outil de placement PLATO [San98] se base sur la puissance de la machine et sur une analyse de la trace d'exécutions précédentes. Mais dans le cas d'une application active, il est impensable d'effectuer, préalablement au déploiement réel, une exécution sur tous les nœuds du réseau uniquement pour observer la quantité de ressources CPU utilisées !

2.6 Résumé

Le concept de code mobile, et celui de réseau actif plus encore, sont des concepts relativement récents. Dans leur jeunesse, la plupart des projets se sont concentrés sur les nouvelles applications possibles ou sur un challenge précis de la mobilité. Si dans un environnement de recherche, il n'était pas problématique d'ignorer dans un premier temps les problèmes liés aux nouvelles attaques pouvant être initiées grâce au code mobile, cela n'est plus vrai dès que les plates-formes ont l'ambition de s'ouvrir « au monde réel ». Aussi des solutions de contrôle de l'utilisation des ressources (en particulier les ressources de calcul) ont commencé à apparaître. Mais comme il a été détaillé dans cette section, ces mesures ne permettent que d'éviter les pires cas mais ne conduisent pas à un

ordonnancement très efficace de la CPU et détériorent parfois dramatiquement les performances des applications distribuées. Comme analysé dans ce chapitre, la cause principale de ces problèmes est que les politiques de contrôle agissent un peu "en aveugle", sans tenir compte des besoins particuliers de chaque application ou des capacités particulières des différents nœuds. En effet la quantité de ressources de calcul dont une application a besoin sur un nœud dépend de différents facteurs que nous avons détaillés. Mais il faut également souligner qu'il n'existe à l'heure actuelle aucune technique permettant d'exprimer les besoins en ressources de calcul de la même façon que des besoins en mémoire ou en bande passante sont exprimés et utilisés dans les réseaux. En conclusion, dans les réseaux actifs il ne faut pas se contenter de gérer la bande-passante mais aussi le deuxième plan fondamental des réseaux actifs : la CPU et la mémoire.

S.C.D. - U.H.P. NANCY 1
BIBLIOTHÈQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

Problématique induite par le contexte

Dans le chapitre 1 nous avons montré les perspectives enthousiasmantes qu'offrent les réseaux actifs. Mais nous avons également précisé dans le chapitre 2 que leur gestion actuelle n'est pas optimale car les nœuds appliquent des politiques de sécurité à l'exécution des applications actives sans connaître les besoins en ressources de calcul de ces applications. En effet, ces besoins varient d'un nœud à un autre et d'une application à une autre, mais contrairement aux besoins en bande passante qui s'expriment universellement en bits par seconde ou aux besoins en mémoire unanimement exprimés en octets, il n'existe pas d'unité de mesure des besoins en ressources de calcul qui soit universellement reconnue et significative sur la grande variété de machines existantes.

L'objectif de ma recherche a été d'**imaginer une technique qui permettrait à une application de présenter à un nœud (appartenant à un réseau hétérogène[31]) une information permettant de prévoir le temps CPU dont elle aura besoin pour s'exécuter.**

Être capable d'exprimer les besoins en ressources de calcul d'une application d'une façon significative sur des nœuds hétérogènes aurait de multiples bénéfices :

Amélioration de la sécurité, de la sûreté et de l'efficacité des nœuds et du réseau

L'expression des besoins d'une application sur un nœud permettrait de mettre en place des politiques d'admission et de contrôle plus efficaces que la fixation d'une limite arbitraire, comme nous le démontrerons concrètement au chapitre 6. *RCANE* [AMK+01] permet à un utilisateur (ou une application) de réserver une portion de CPU avec une certaine périodicité : par exemple, si une application traite un flux vidéo traversant le réseau dont les paquets arrivent toutes les 20 ms et nécessitent 0.5 ms pour être traités, elle demandera à réserver une « tranche » de CPU de 0.5 ms toutes les 20 ms (de temps apparent). Mais cette approche suppose que l'application *sait* combien de temps CPU est nécessaire pour exécuter ses paquets sur l'ensemble des nœuds. Notre travail peut permettre d'obtenir les estimations nécessaires.

[31] Par « hétérogène » j'entends que les nœuds du réseau sont constitués de matériels différents, ou exploités par des systèmes divers, conduisant à des performances différentes.

Planification

Connaître les ressources en calcul nécessitées par une application active populaire peut être utile pour aider un fournisseur de réseau à dimensionner son offre. C'est une information supplémentaire à intégrer à ses modèles.

Facturation

L'utilisation de l'Internet par exemple est déjà facturée, généralement au forfait (abonnement mensuel par exemple pour une connexion à un certain débit). Cette solution n'est pas très équitable car les utilisateurs au dessous du niveau d'utilisation ayant servi à établir le tarif payent en fait pour les utilisateurs très actifs. Une facturation à l'usage (proportionnelle à l'utilisation de la connexion et au niveau de qualité de service sélectionné) serait plus juste. Mais la facturation de l'Internet ne peut pas être basée sur la facturation téléphonique à cause de sa nature « sans connexion » (suivre une session unique, même courte, générerait des centaines d'enregistrements de comptabilité). De plus, le trafic envoyé par un serveur pour le compte d'un client devrait être imputé au client, pas au serveur, ce qui complique encore la tâche. Pour anecdote, deux pays ont tenté une facturation à l'utilisation de l'Internet : le Chili (échec total) et la Nouvelle Zélande (expérience positive avec NetTraMet). En France, Transpac le permet pour X25 depuis 1979. Mais même dans ces facturations plus fines, la seule ressource considérée est la bande passante ce qui se justifie complètement par le fait que dans les réseaux classiques tous les paquets consomment environ la même quantité de CPU puisqu'ils sont tous traités de façon identique (réception, consultation de la table de routage, renvoi). Mais ce n'est plus vrai dans le contexte des réseaux actifs (réception, traitement, éventuellement renvoi des paquets ou modification de l'état du nœud) où la consommation de ressources de traitement peut devenir très variable d'un paquet à l'autre. Il ne semble donc pas équitable de facturer l'utilisateur seulement à la bande passante utilisée mais aussi en fonction des traitements plus ou moins complexes qu'il a demandés. Dans ce contexte, il peut être utile de disposer d'un outil permettant d'évaluer *a priori* le prix d'un service.

Routage et qualité de service

Connaissant les besoins d'une application et les capacités des nœuds, des décisions de routage pourront se faire sur cette nouvelle base pour optimiser l'utilisation du réseau ou fournir aux applications ayant souscrit un haut niveau de qualité de service la possibilité d'emprunter la route la mieux adaptée à leurs besoins.

Critère de comparaison des applications

Si les utilisateurs doivent être facturés à l'utilisation qu'ils font du réseau, ils seront certainement intéressés par une technique leur permettant de comparer deux applications proposant le service qu'ils recherchent afin de choisir celle ayant le meilleur rapport qualité-prix pour eux.

Mécanisme d'adaptation du débit

Le contrôle de flux dans TCP est réalisé au moyen de fenêtres d'anticipation à taille variable. L'accusé de réception d'un message indique le nombre maximal d'octets que le récepteur est prêt à recevoir [Tan97]. L'équivalent de ce mécanisme pour les réseaux actifs ne devrait pas seulement prendre en compte la taille des buffers de réception mais aussi les capacités de calcul des nœuds par rapport aux besoins des applications !

Il faut également signaler que le travail que je vais présenter pourrait peut-être s'appliquer aussi à d'autres domaines du code mobile mais que je me concentre sur les réseaux actifs. Par exemple, un avantage du code mobile souvent mis en avant est la possibilité de placer le code à l'endroit le mieux adapté au traitement que l'on souhaite réaliser. La capacité d'exprimer les besoins en ressources de calcul de ce code permettrait de vérifier que l'endroit choisi pour son déploiement est un choix valide.

Enfin, il est intéressant de remarquer qu'avec cette problématique les nouvelles préoccupations de la communauté réseau (traditionnellement « orientée bande passante » mais bien forcée à présent de s'intéresser aussi aux ressources CPU) convergent avec celles des chercheurs en calcul parallèle, qui longtemps n'ont considéré que la puissance de calcul des nœuds, mais qui prennent aujourd'hui en compte la bande passante également [San98].

Deuxième partie

Contributions

Introduction

1 Objectifs et cahier des charges

Dans le contexte présenté dans la première partie de ce document, j'ai développé **une technique pour permettre aux applications actives d'exprimer leurs besoins en ressources CPU d'une façon qui puisse être interprétée utilement par les différents nœuds d'un réseau actif hétérogène.** Par « réseau hétérogène » j'entends un réseau constitué de machines aux configurations et performances variées (Cf. 2.3).

Idéalement, la technique utilisée, ou le modèle devrait :

- permettre de prédire non seulement le temps moyen d'exécution mais aussi les hauts pourcentiles (le P^e « pourcentile » donne le temps au-dessous duquel P % des exécutions se terminent),
- dans la mesure du possible, tenir compte d'un maximum des sources de variabilité dégagées dans le paragraphe 2.3,
- fournir des prédictions dont la marge d'erreur ne dépasse pas 10 % pour la moyenne et 20 % pour les hauts pourcentiles[32],
- fournir une estimation de l'erreur,
- être « léger » à transporter par le réseau,
- être rapide à fournir une prédiction sur les nœuds (ceci est particulièrement utile lors de la première réception d'un paquet d'un nouveau type dans un réseau actif adoptant l'approche dite intégrée),
- être facile à générer, c'est-à-dire demandant au programmeur le moins d'effort possible,
- être compatible avec le maximum d'environnements d'exécution et de systèmes d'exploitation de nœud actif.

Bien sûr, on désire pouvoir prédire le temps d'exécution d'une application sur un nœud qu'elle n'a jamais visité. Il est exclu d'exécuter l'application « partout » préalablement à son déploiement réel juste pour voir combien de temps elle mettrait à s'exécuter.

2 Idée générale de la solution proposée

S'inpirant des travaux de Saavedra et Smith cités au chapitre 2 ([SBSM89]), nous modélisons un programme actif par une suite ordonnée et étiquetée d'éléments, où les

[32]Nous avons en effet observé dans plusieurs *workshops* que ces valeurs permettent de concevoir des solutions de gestion satisfaisantes.

Nom de la machine	Type de processeur	Fréquence du processeur (MHz)	RAM (MBytes)
Green	Pentium Pro	199	64
Blue	Pentium II	451	131
Black	Pentium II	334	130
Yellow	Pentium 75	100	81
Red	Pentium II	267	131

TAB. 1 – Caractéristiques des nœuds du réseau de test

éléments représentent les fonctions fournies par un nœud actif. Cette série d'éléments est transmise aux nœuds avant le premier « vrai » paquet actif de l'application qu'elle représente. D'autre part, chaque fois qu'un nœud est ajouté au réseau, ou lorsque sa configuration change, il est « calibré » pour déterminer combien de temps il utilise pour exécuter chacune des fonctions du système actif. La combinaison de ces deux informations (éléments utilisés par un programme et temps nécessaire pour exécuter les éléments sur un nœud particulier) permet à un nœud actif de prédire le temps d'exécution du programme actif modélisé.

Pour tester cette idée, et en attendant l'émergence d'un système d'exploitation actif conforme aux spécifications [Pet01], nous examinons les appels qu'une application active fait au noyau Linux.

3 Plan de cette partie

La solution proposée pour atteindre les objectifs présentés nécessite de collecter des informations fines sur l'exécution des applications actives. Les systèmes d'exploitation existants ne permettant pas, dans leur état au moment où ce travail a débuté, d'obtenir ces informations, j'ai commencé par développer un outil d'observation pour Linux. Le chapitre 3 décrit ce travail. Le chapitre 4 présente ensuite le premier modèle proposé pour représenter le comportement des applications ainsi que le modèle représentant les performances des nœuds, puis explique comment ils peuvent être utilisés pour prévoir le temps d'exécution d'une application active sur n'importe quel nœud, et reporte une évaluation de la précision des prédictions lors de phases de test. Le modèle d'application est ensuite amélioré dans le chapitre 5.

4 Description du réseau de test utilisé

Le réseau que j'ai utilisé dans les expériences qui suivent est constitué de 5 PC avec les caractéristiques résumées dans le tableau 1 et reliés par un réseau Ethernet à 100 MBit/s. Ils utilisent une version légèrement modifiée du noyau Linux 2.2.7 (voir chapitre 3). Les environnements d'exécution *ANTS* et *Magician* sont installés sur tous les nœuds. Ces nœuds font partie du ABone.

Chapitre 3

Outil d'observation

Sommaire

3.1 Objectifs du monitorage 53
3.2 Inadaptation des outils existants 55
3.3 Implémentation du prototype de traçage 57
3.4 Évaluation du prototype 61
3.5 Instrumentation des plates-formes 62
3.6 Résumé 64

En l'absence de fonctions de monitorage adaptées dans les nœuds actifs, il est nécessaire de modifier légèrement le noyau de Linux pour pouvoir mesurer et contrôler finement les ressources CPU utilisées par une application active. Ce chapitre explique le fonctionnement de l'environnement d'observation créé. À terme, les spécifications des fonctions standards du nœud actif devraient imposer la présence de fonctions de mesures fines similaires.

3.1 Objectifs du monitorage

Les spécifications des premières versions des documents d'architecture laissaient supposer que l'exécution d'un paquet actif pouvait s'identifier à un processus du système d'exploitation (concrètement ce n'est malheureusement pas toujours aussi évident comme nous le verrons plus loin). Pour générer les modèles qui seront décrits dans les chapitres suivants, il faut **collecter des mesures fines sur le processus correspondant à l'exécution d'un paquet actif** sous la forme d'une trace de son exécution comportant :

- l'identité des appels système successifs effectués,
- pour chaque appel système, le temps[33] passé en mode privilégié à l'exécuter,

[33] Sauf mention spéciale, lorsque je parle de « temps » dans ce document il s'agit de « temps machine » encore appelé « temps CPU » ou « temps processeur » par opposition au « temps d'exécution apparent » ou « *wall-clock time* ». Le système d'exploitation utilisé dans ce travail (Linux) étant multi-tâches, 1 min peut s'écouler à la montre d'un observateur entre le lancement d'un programme et la fin de son exécution alors que le programme n'a réellement occupé le processeur que pendant 30 secondes, passant

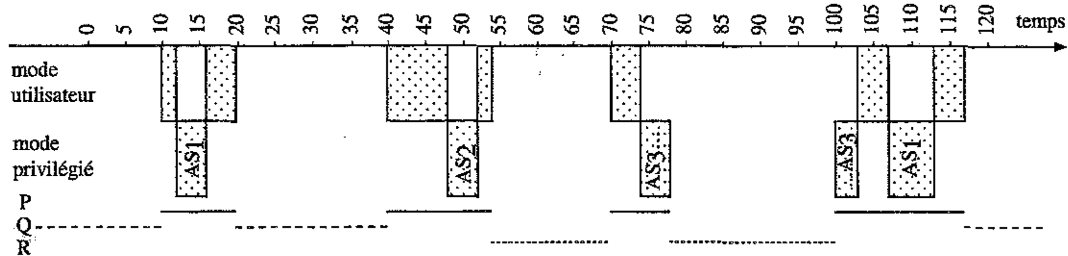


FIG. 3.1 – Partage du processeur entre plusieurs processus

- le temps passé dans l'environnement d'exécution, en mode utilisateur, entre deux appels système consécutifs.

Imaginez que l'on veuille tracer l'exécution d'un processus P sur une machine mono-processeur fonctionnant sous Linux. Supposons qu'il commence par invoquer l'appel système AS_1 puis AS_2 et AS_3 avant d'appeler encore AS_1 et de se terminer. Comme Linux est un système multi-tâches, P n'est pas le seul processus désirant s'exécuter et il doit partager le processeur avec des processus concurrents Q et R (dont nous ne détaillerons pas l'exécution). Imaginons que l'ordonnanceur oblige P à céder et reprendre le processeur selon le schéma de la figure 3.1.

Ce schéma se lit de la façon suivante : les chiffres du haut indiquent l'échelle de temps. P est choisi par l'ordonnanceur et commence à s'exécuter en mode utilisateur à la date 10. À la date 12, il commence l'exécution de l'appel système AS_1 en mode privilégié ; à la date 16 il termine l'exécution de AS_1 et poursuit son exécution en mode utilisateur jusqu'à la date 20. Mais à ce moment l'ordonnanceur décide qu'il est temps pour un autre processus prêt d'être exécuté et, après avoir sauvé l'état de P, il suspend son exécution pour accueillir le processus Q. À la date 40, P est choisi à nouveau et continue son exécution où il l'avait laissée et ainsi de suite jusqu'à ce qu'il ait terminé sa tâche. Le schéma fait également apparaître une situation légèrement différente qui peut se produire : à la date 78, au milieu de l'exécution d'un appel système, c'est le processus P lui-même qui prend l'initiative de se suspendre pour laisser d'autres processus s'exécuter pendant qu'il attend le résultat d'une entrée-sortie par exemple.

L'objectif du monitorage est de résumer l'exécution du processus P sous la forme d'une trace représentée par le tableau 3.2 (000 correspond à l'état initial et à l'état final). Les deux premières colonnes indiquent la succession des appels systèmes réalisés : de l'état initial on est passé à l'appel système AS_1 (001), puis de AS_1 à AS_2 (002), de AS_2 à AS_3, de AS_3 à un nouvel appel à AS_1 et de cet appel à AS_1 à l'état final. La troisième colonne de la trace indique le temps passé en mode utilisateur entre les appels. Par exemple, l'exécution du premier appel à AS_1 se termine à la date 16 et P continue son exécution en mode utilisateur jusqu'à la date 20 avant d'être interrompu. Il reprend son exécution, toujours en mode utilisateur, à la date 40 et à la date 48 il appelle AS_2. Le temps que P a passé à s'exécuter en mode utilisateur entre AS_1 et AS_2 est donc : 20-16 + 48-40 = 12, que l'on retrouve sur la deuxième ligne de la trace d'exécution.

30 autres secondes à attendre l'accès au processeur qu'il partage avec d'autres processus. L'annexe B résume ce fonctionnement.

transitions		unités de temps utilisées	
appel système de départ	appel système suivant	entre les 2 appels système	dans l'appel système de départ
000	001	2	0
001	002	12	5
002	003	6	4
003	001	5	8
001	000	4	7

FIG. 3.2 – Trace d'exécution du processus P de la figure 3.1

Enfin, la dernière colonne indique le temps passé dans les appels système. Par exemple, l'exécution de l'appel système AS_2 débute à la date 48 et se termine à la date 52. On retrouve sur la troisième ligne de la trace d'exécution le temps d'exécution de AS_2 : 52-48 = 4.

3.2 Inadaptation des outils existants

Lorsque j'ai commencé ce projet, j'ai été surprise de ne pas trouver d'outil permettant d'obtenir une trace telle que celle que je viens de décrire. Cette section présente les candidats étudiés.

3.2.1 GPROF

Gprof (*Gnu Profiling Tool*, [GPr]) est un outil Unix permettant d'afficher le graphe d'exécution d'une application. Malheureusement, les résultats qu'il fournit sont trop agrégés pour notre objectif : il est par exemple impossible de différencier deux occurrences d'un même appel système. De plus une trace n'est fournie qu'une fois le programme terminé ce qui pose un problème dans notre cas car en pratique l'exécution d'un paquet actif n'est pas contenue dans un programme individuel que l'on pourrait aisément arrêter et redémarrer à chaque paquet. On souhaite ne tracer qu'une partie de l'exécution de la plate-forme active, alors que cet outil est prévu pour observer un programme dans sa totalité.

3.2.2 Outils de génération de profils de programmes Java

Puisque beaucoup de plates-formes actives sont construites sur Java, on pourrait envisager l'utilisation du paramètre *-Xrunhprof* de l'interpréteur [Xru]. L'option *cpu=times* mesure le temps utilisé par chaque méthode et reporte cette information en pourcentage du temps total pris par l'application. Le nombre d'appels à chacune des méthodes est également indiqué. Mais cela ne répond pas complètement à nos besoins puisque (entre autres) les comptes-rendus d'utilisation de la CPU sont générés par occurrence d'appel aux méthodes Java et non pas par appel système. JVMPI et JVMDI (*JVM Profi-*

ler/Debugger Interface) présentent les mêmes inconvénients (de plus, le code à monitorer doit d'abord être instrumenté).

3.2.3 Instrumentation de la bibliothèque « glibc »

Puisque la JVM fait appel au noyau au travers de la bibliothèque C, une solution était d'ajouter des éléments de traçage à cette bibliothèque. En effet les sources de la bibliothèque C de GNU (glibc) sont disponibles librement et il est également possible d'obtenir les sources de la JVM et de la recompiler en utilisant la glibc. Mais de telles modifications demandent un haut niveau d'expertise et j'ai préféré m'orienter vers une solution qui d'une part me semblait plus rapide à mettre en place et surtout qui pourrait permettre de mesurer aussi les environnements d'exécution non basés sur Java.

3.2.4 Utilitaire « Strace » de Linux

Strace[34] [Str] est un programme Linux permettant d'observer, à l'exécution, les appels système et signaux émis ou reçus par n'importe quel programme. Pour cela, *Strace* commence par s'approprier la paternité du processus à examiner ; puis il contrôle l'exécution du processus à l'aide de l'appel système *ptrace* afin d'enregistrer les appels système invoqués et d'intercepter les signaux émis ou reçus qu'il mémorise avant de les retransmettre. Un aspect très intéressant de cet outil est que les programmes n'ont pas besoin d'une recompilation spéciale pour pouvoir être tracés. Le nom de chaque appel système, ses arguments et plusieurs autres informations peuvent être imprimés. Une option (-r) estampille l'entrée dans chaque appel système, enregistrant ainsi le temps écoulé entre les débuts de deux appels système successifs. Une autre option (-T) affiche le temps passé dans les appels système. Malheureusement, mis à part dans son mode résumé (option -c) qui ne fournit pas les détails dont j'avais besoin (ordre des appels système par exemple), *Strace* fournit des résultats exprimés en temps apparent (*« wall-clock time »*) et non pas en temps CPU. La mise en évidence de ceci est simple : en utilisant -T lors d'un test, on constate qu'une occurrence de *sleep(9)* semble durer 9,004735 secondes alors qu'en réalité le processus n'utilise que très peu le processeur et le cède à d'autres processus pendant les 9 secondes qu'il passe endormi (avec l'option -c on obtient un temps d'exécution de 2 micro-secondes pour le même *sleep*!). Par conséquent, nous avons dû rejeter ce candidat car, Linux étant un système multi-tâches, si beaucoup de tâches sont en concurrence le temps apparent nécessaire pour les exécuter sera plus élevé que dans le cas où le système est peu chargé ; or nous voulons obtenir des mesures les plus indépendantes possible de la charge. Cependant, comme nous le détaillerons plus loin, *Strace* (dans sa version « avec l'option -c ») a été utile pour évaluer l'*overhead* introduit par le monitorage mis en place.

[34] *Truss* est l'équivalent de *Strace* pour Solaris.

3.3 Implémentation du prototype de traçage

3.3.1 Information temporelle fine et précise

J'ai voulu modifier *Strace* afin qu'il ne fournisse non pas des résultats en temps apparent mais en temps CPU. Les processus Linux possèdent 2 informations potentiellement utiles : il est possible de savoir combien de temps (CPU) un processus a été ordonnancé en mode utilisateur et en mode privilégié depuis sa création (en consultant ses champs *utime* et *stime*). Les valeurs retournées sont exprimées en *jiffies*. Classiquement, il y a 100 *jiffies* par seconde. Malheureusement cette granularité n'était pas suffisante pour mesurer des événements aussi courts que certains appels système (il n'est pas rare qu'ils durent moins d'une demi milliseconde). Mais lorsque l'on augmente la fréquence d'interruption de l'horloge afin d'améliorer la précision d'un *jiffy*, le temps pris par les interruptions par rapport au temps laissé à l'exécution est trop grand et le système semble alors « gelé ». En effet, l'ordonnanceur utilise cette valeur comme base pour les tranches de temps allouées aux processus.

La solution vers laquelle je me suis alors tournée est d'utiliser les **cycles d'horloge**, une mesure très fine et précise. Sur les processeurs *Pentium*, une instruction spéciale permet de connaître le nombre de cycles d'horloge écoulés depuis le dernier re-démarrage de la machine (RDTSC, *Read Time Stamp Counter*, [Int]).

Mais utiliser et enregistrer cette information pour mesurer le temps d'exécution d'un processus donné nécessitait de modifier un peu le noyau Linux : des champs supplémentaires ont dû être ajoutés à la structure d'un processus et des modifications ont été effectuées pour que ces champs soient mis à jour à l'entrée et à la sortie du processeur et de l'ordonnanceur. Ce travail est détaillé dans cette section. L'annexe B précise le fonctionnement des processus sous Linux et l'annexe C contient les modifications apportées au code du noyau.

3.3.2 Grandes lignes de l'implémentation

Ajout de champs à la structure d'un processus

Pour enregistrer le nombre de cycles d'horloge passés par un processus en mode utilisateur et le nombre de cycles d'horloge passés en mode privilégié, on ajoute 2 champs à la structure d'un processus Linux : *ucc* pour le mode utilisateur et *kcc* pour le mode privilégié (ou « noyau » ou « *kernel* »). 2 champs auxiliaires utiles pour les algorithmes décrits plus loin ont également été ajoutés : e enregistre la valeur de l'horloge (résultat de RDTSC) à l'entrée dans un nouvel « état » et $kflag$ qui indique si le processus s'était « auto-suspendu » dans le noyau lors de sa dernière sortie du processeur ou s'il avait été interrompu en mode utilisateur.

Algorithme simplifié de mise à jour des temps horloge d'un processus

L'algorithme utilisé pour mettre à jour les champs *ucc* et *kcc* est le suivant :

- À l'entrée dans le processeur : le champ e reçoit la valeur courante du nombre de cycles d'horloge écoulés depuis le dernier re-démarrage obtenu grâce à l'instruction

rdtsc.
- À l'entrée en mode privilégié : le drapeau $kflag$ est mis à 1. Puis la valeur de *ucc* est mise à jour : la valeur $rdtsc - e$ donne le nombre de cycles d'horloge écoulés entre la dernière mise à jour de e (à l'entrée dans le processeur) et la « date » courante ; cette valeur est ajoutée à l'ancienne valeur de *ucc*. Enfin, e est reinitialisé à la valeur renvoyée par *rdtsc*.
- À la sortie du mode privilégié : le drapeau $kflag$ est remis à 0. Puis la valeur de *kcc* est mise à jour en y ajoutant la quantité $rdtsc - e$. Enfin, e reçoit la valeur renvoyée par *rdtsc*.
- À la sortie du processeur : si le drapeau $kflag$ vaut 0, c'est que le processus a été interrompu en mode utilisateur et on met *ucc* à jour en lui ajoutant $rdtsc - e$; sinon, le processus s'est « auto-suspendu » en mode noyau et on met *kcc* à jour.

La figure 3.3 montre la progression de l'algorithme de traçage sur le processus P exemple. À chaque mise à jour, *ucc* et *kcc* indiquent le nombre de cycles d'horloge passés en mode utilisateur et en mode noyau par le processus depuis le démarrage de son monitorage.

3.3.3 Collecte des résultats

Le processus est maintenant équipé d'informations très fines. Il ne reste plus qu'à les consulter à chaque appel système pour construire la trace d'exécution désirée.

S'inspirant de la technique utilisée par *Strace*, on pourrait utiliser l'appel système *ptrace* pour contrôler le processus à tracer. Une fois le processus à tracer attaché, le processus contrôleur exécute une boucle comprenant la requête *PTRACE_SYSCALL* pour que l'exécution du processus attaché se poursuive jusqu'au prochain appel système où les valeurs de *ucc* et *kcc* sont relevées avant d'autoriser le processus attaché à reprendre son exécution jusqu'au prochain appel système et ainsi de suite. Malheureusement je n'ai pas réussi à faire fonctionner ce prototype correctement[35].

J'ai donc implémenté mon propre mécanisme pour collecter les résultats de mesure. Ce mécanisme ne demande que quelques ajouts aux modifications déjà décrites pour imprimer un message avec les mesures aux points clés. À l'entrée en mode privilégié, un message de la forme :

```
trace{pid} I{numero_d_appel_systeme} U {32 bits de poids fort d'ucc}
    {32 bits de poids faible d'ucc}
```

est imprimé, et un autre l'est à la sortie du mode privilégié :

```
trace{pid} O{numero_d_appel_systeme} K {32 bits de poids fort de kcc}
    {32 bits de poids faible de kcc}
```

Le code d'instrumentation est une modification de la partie du noyau gérant le processus. Tout affichage ne peut se faire qu'avec des procédures dédiées comme *printk*. Le numéro de l'appel système est lu sur la pile. *ucc* et *kcc* sont codés sur 64 bits mais *printk* ne permet pas d'afficher des nombres codés sur plus de 32 bits d'où l'affichage en 2 parties.

[35] *ptrace* a été considérablement corrigé depuis, il serait intéressant de réessayer.

rdtsc	événement	ucc	kcc	kflag	e
0		0	0	0	0
10	P commence son exécution en mode utilisateur				10
12	invocation de l'appel système AS_1	0+(12-10)=2		1	12
16	fin de l'exécution de l'appel système AS_1		0+(16-12)=4	0	16
20	le processus P doit céder le proccesseur	2+(20-16)=6			
40	P reprend son exécution en mode utilisateur				40
48	invocation de l'appel système AS_2	6+(48-40)=14		1	48
52	fin de l'exécution de l'appel système AS_2		4+(52-48)=8	0	52
54	le processus P doit céder le proccesseur	14+(54-52)=16			
70	P reprend son exécution en mode utilisateur				70
74	invocation de l'appel système AS_3	16+(74-70)=20		1	74
78	l'exécution est suspendue pendant l'appel système AS_3		8+(78-74)=12		
100	P reprend son exécution en mode privilégié				100
103	fin de l'exécution de l'appel système AS_3		12+(103-100)=15	0	103
107	invocation de l'appel système AS_1	20+(107-103)=24		1	107
112	fin de l'exécution de l'appel système AS_1		15+(112-107)=20	0	112
116	le processus P termine son exécution et libère le processeur	24+(116-112)=28			

FIG. 3.3 – Déroulement du traçage du processus de la figure 3.1

La différence entre 2 valeurs successives de *kcc* ou *ucc* permet de déterminer le temps passé dans les appels système et entre les appels système. En revanche cette trace ne permet pas de connaître le nombre de cycles d'horloge écoulés entre la sortie du dernier appel système et la dernière sortie du processeur. Pour combler ce manque, à chaque sortie du processeur un message est également imprimé :

```
trace{pid} exit U {32 bits de poids fort d'ucc}
    {32 bits de poids faible d'ucc}
```

Comme le mécanisme de traçage n'est pas le seul à effectuer des *printk*, la trace doit être traitée avant analyse pour ne conserver que les lignes intéressantes. C'est la raison pour laquelle les mots-clés « trace » apparaissent dans les messages, ils facilitent l'opération de filtrage.

Dans un premier temps, j'avais modifié le fichier *syslog.conf* afin que tous les messages de priorité *kernel_notice* (niveau choisi lors des *printk*) soient imprimés dans un fichier */var/log/mytrace*. La fonction *printk* écrit les messages dans un buffer circulaire puis réveille les processus en attente de messages (c'est-à-dire les processus endormis dans l'appel système *syslog* ou ceux qui lisent */proc/kmesg*). Par ailleurs, si le démon *klogd* est actif, il relève régulièrement les messages du noyau et les distribue à *syslogd* qui consulte */etc/syslog.conf* pour savoir quoi faire des messages. Si le démon *klogd* n'est pas actif, les données restent dans le buffer circulaire jusqu'à ce qu'elles soient lues ou que le buffer déborde et qu'elles soient alors écrasées par de nouvelles données. Des tests ont montré que le rythme auquel j'écris dans ce buffer est trop soutenu pour qu'aucun débordement n'ait lieu avant que *klogd* ne le consulte. Par conséquent des informations de ma trace disparaissent. Pour résoudre ce problème, la première solution est d'augmenter la taille du buffer (*LOG_BUF_LEN*) ou la fréquence du démon *klogd*. J'ai trouvé plus simple d'inactiver le démon et récolter la trace avec un simple "*cat/proc/kmsg*".

La trace générée pour l'exemple (processus P de pid 132) ressemblera à :

```
trace132 I1 U2
trace132 O1 K4
trace132 exit U6
trace132 I2 U14
trace132 O2 K8
trace132 exit U16
VFS: Disk change detected on device ide1(22,64)
trace132 I3 U20
trace132 exit U20
trace132 I3 U20
trace132 O3 K15
trace132
```

Il ne reste plus qu'à utiliser un petit script pour transformer cette présentation en celle exposée dans la figure 3.2. Sur cet extrait de trace d'exécution, à la ligne « trace132 I1 U2 », on voit que lorsque le processus a invoqué l'appel système 1, son exécution avait déjà duré 2 cycles d'horloge en mode utilisateur. Et plus loin la ligne « trace132 I2 U14 » indique qu'au moment où P appelle l'appel système 2, il a passé 14 cycles d'horloge en mode utilisateur. Entre les appels système 1 et 2 il a donc passé 12 cycles d'horloge

S.C.D. - U.H.P. NANCY 1
BIBLIOTHEQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

en mode utilisateur. Et en regardant les valeurs successives suivant les « K », on peut déterminer de même le temps passé dans les appels système.

3.3.4 Remarques

Pour limiter la charge supplémentaire générée par le mécanisme de traçage, je ne trace que les processus m'intéressant. Pour cela j'ai ajouté un dernier champ à la structure de processus, $tflag$, indiquant si le processus doit être tracé ou pas. Initialement ce drapeau vaut 0. Les opérations décrites ci-dessus ne sont exécutées que si le drapeau est à 1. J'ai également ajouté un nouvel appel système (set_tflag) permettant de mettre à 1 le drapeau d'un processus.

Comme évoqué plus haut, *ucc* et *kcc* sont codés sur 64 bits. Sur un processeur à 850 MHz, le premier débordement aura lieu après 2^{64} cycles * (1 seconde / 850000000 cycles) = 688 ans. Cela ne constitue donc pas un problème pour nos expériences.

Les modifications du code présentées en annexe ne conviennent pas pour les machines multi-processeurs mais il devrait être possible d'adapter cette approche et modifier la partie « MP » (utilisée par les machines multi-processeurs) de l'ordonnanceur. Pour les machines ne proposant pas l'instruction RDTSC une autre solution doit être trouvée.

Dans le cadre d'un pré-projet collaboratif entre le NIST et NAI Labs, nous nous sommes interrogés sur la portabilité des modifications apportées au noyau Linux vers l'Exokernel du MIT [EKO95] (système d'exploitation expérimental utilisé par le projet sur les réseaux actifs de NAI Labs). Cela semble facilement faisable, à une exception près : il semble ne pas y avoir de notion de mode utilisateur et mode privilégié, tous les appels systèmes sont faits en mode utilisateur. Il n'y a donc pas de point clé unique à instrumenter comme dans Linux. Mais on pourrait ajouter du code au début et à la fin de chaque appel système. Une autre solution consisterait à ne pas essayer de porter ce que j'ai fait sous Linux mais plutôt d'utiliser les spécificités de l'Exokernel. L'autre problème est que *Kaffe* est le seul interpréteur Java disponible pour l'Exokernel et qu'il ne supporte pas les *threads* natifs alors que nous en avons besoin pour mesurer des applications *ANTS* ou *Magician* (Cf. paragraphe 3.5).

3.4 Évaluation du prototype

3.4.1 Sensibilité aux mécanismes de *cache*

Pour tester le prototype de monitorage, j'ai écrit un petit programme exécutant une vingtaine d'appels système dans une boucle répétée 1000 fois. J'ai utilisé le prototype pour mesurer le temps d'exécution de chaque occurrence de chaque appel système durant une exécution du programme de test. Puis pour chaque appel système j'ai calculé le temps moyen d'exécution. Et j'ai répété cela plusieurs fois en gardant les mêmes conditions. Le temps moyen d'exécution d'un appel système X obtenu après chaque exécution du programme de test aurait dû rester sensiblement constant. Et c'est effectivement le cas pour 9 exécutions sur 10. Mais 1 exécution du programme de test sur 10 fait apparaître, pour l'appel système X donné, un temps moyen d'exécution plus élevé que celui observé

lors des 9 autres exécutions du programme de test. Une analyse plus fine de la trace de cette exécution aux résultats particuliers révèle que sur les 1000 appels à cet appel système X, la plupart ont duré un temps compatible avec la moyenne observée lors des 9 exécutions aux résultats similaires mais très régulièrement (tous les 20 appels à *gettimeofday* par exemple) l'appel a mis plus de temps à s'exécuter.

La parfaite régularité du phénomène laissait imaginer un problème dû au mécanisme de *caches* : si l'ensemble du code et des données n'est pas présent dans les caches d'instructions et de données au moment de l'exécution (car ils sont trop longs ou ont été remplacés par le code et les données d'un autre processus), quelques cycles supplémentaires seront nécessaires par rapport à la situation où tout est déjà prêt. Ainsi le temps d'exécution d'un programme peut varier d'une exécution à l'autre, même si les paramètres restent exactement les mêmes. Pour vérifier cette hypothèse, j'ai réduit la longueur du programme et répété les expériences. Les irrégularités ont complètement disparu. Il semble que la taille maximale pour ne pas être sensible au *cache* est de 1920 octets (mais il faut noter que ce chiffre doit dépendre de l'architecture, de la taille du cache ou du nombre de processus prêts par exemple).

3.4.2 Évaluation de l'*overhead* dû à l'instrumentation

Pour évaluer les performances du prototype j'ai mesuré le même programme de test que ci-dessus avec *Strace* en utilisant l'option « -c ». Les temps fournis par le kernel instrumenté sont plus longs que ceux fournis par *Strace* de 30 micro-secondes. Cet overhead peut s'expliquer par les opérations supplémentaires que le noyau doit effectuer pour maintenir à jour les champs que j'ai ajoutés à sa structure.

3.5 Instrumentation des plates-formes

3.5.1 Nécessité d'une instrumentation

Comme indiqué dans l'introduction de ce chapitre, j'ai supposé que l'exécution d'un paquet actif correspondait à un processus du système d'exploitation. Malheureusement ce n'est pas aussi simple en pratique : *Magician* par exemple est un programme Java qui se comporte comme un démon, il est en perpétuelle attente de messages sur ses ports de réception, s'assure périodiquement que ses voisins sont toujours vivants en échangeant des messages de contrôles avec eux ; de même *ANTS* effectue régulièrement un nettoyage de la mémoire temporaire du nœud etc. Par conséquent, mesurer le processus Linux correspondant au programme Java d'*ANTS* ou de *Magician* enregistrerait beaucoup d'activités parasites non imputables directement au traitement d'un paquet.

Mais ces deux plates-formes sont programmées de telle façon que le traitement des paquets est isolé dans un *thread* Java : dans *ANTS* le *ChannelThread* exécute les paquets les uns à la suite des autres et dans *Magician*, un nouveau *KUSmartPacketV2* est créé pour l'exécution de chaque paquet. En demandant l'utilisation de *threads* natifs au démarrage des plates-formes, leur fonctionnement est éclaté en plusieurs processus Linux au lieu d'un seul et un de ces processus correspond au *thread* Java traitant les paquets

actifs. Il suffit donc de tracer ce processus spécifique pour n'enregistrer que l'activité directement liée à l'exécution des paquets.

D'autre part, comme l'unité de base des modèles d'applications actives que nous verrons dans les chapitres suivants est le paquet et non pas le flux de paquets, il faut pouvoir identifier le début et la fin de chaque paquet dans la trace d'exécution. Pour cela, j'ai légèrement modifié les plates-formes afin qu'elles impriment un message de repère dans la trace au début et à la fin de l'exécution de chaque paquet[36].

3.5.2 Monitorage des applications *ANTS*

Pour tracer une application *ANTS*, son développeur doit démarrer le nœud *ANTS* sur lequel il veut collecter la trace en utilisant l'option « -native ». Avec le JDK 1.2.2 sur un noyau Linux 2.2.7 modifié conformément à ce qui est présenté dans l'annexe C, la plate-forme s'exécutera sous forme de 7 processus Linux. Le dernier créé de ces processus exécute les paquets actifs (il correspond au *ChannelThread*).

Pour le monitorer, il suffit de mettre à 1 la valeur du champ spécial *tflag* grâce au nouvel appel système *set_tflag*. Un petit programme automatisant cette tâche est disponible. À partir du moment où *tflag* est mis à 1, l'exécution des paquets actifs est monitorée et une trace est écrite.

Mais l'exécution d'un paquet actif PA_n est enregistrée dans la trace immédiatement après l'exécution du paquet précédent PA_{n-1}. Pour pouvoir distinguer les limites des exécutions des différents paquets, j'ajoute un petit message dans la trace au début de chaque paquet. De plus, pour pouvoir déterminer facilement le temps écoulé entre le début de l'exécution d'un paquet et le premier appel système réalisé, j'indique aussi, suite au message de « début », la valeur courante de *ucc*. Idem à la fin des paquets[37].

Pour insérer ces messages dans la trace d'exécution, j'ai introduit un nouvel appel système, *trace_tag*. Cet appel système est invoqué, via une méthode native Java, au début et à la fin de la méthode *run* de la classe *ChannelThread*.

3.5.3 Monitorage des applications *Magician*

Contrairement à *ANTS* où tous les paquets actifs sont exécutés par le même processus Linux, sous *Magician* un nouveau processus est créé pour chaque paquet actif (puis détruit à la fin de l'exécution du paquet). J'ai donc bien évidemment intégré la mise à 1 de *tflag* à la plate-forme : *set_tflag* est appelé au début de la méthode *run* de *KUSmartPacketV2* pour activer le traçage du paquet. Le développeur n'a qu'à démarrer la plate-forme en utilisant l'option « -native » et la génération de trace est automatique.

[36] Il faut noter que certains cycles d'horloge qui devraient être directement facturés sur le compte du traitement du paquet échappent à mon monitorage : la réception et désérialisation du code ne sont pas mesurées, seule l'exécution du code l'est. Cependant ce temps non comptabilisé est susceptible d'être identique pour tous les paquets (ou être fonction de leur taille) et nous intéresse donc moins.

[37] Une fois la phase de mise au point terminée, j'ai éliminé le message « exit » ajouté à chaque sortie de l'ordonnanceur comme expliqué précédemment et j'ai adopté cette solution, légèrement plus « économique ».

Puis, comme pour *ANTS*, un message est inscrit dans la trace pour indiquer le début et la fin de l'exécution du paquet et la valeur de *ucc*. *Magician* envoie régulièrement à ses voisins des paquets actifs qui ne sont pas directement reliés aux applications. Pour éviter que l'exécution de ces paquets apparaisse dans la trace, une discrimination est faite sur le nom du type du paquet avant de mettre *tflag* à 1.

3.6 Résumé

Dans ce chapitre j'ai présenté et évalué le prototype que j'ai développé pour pouvoir monitorer l'exécution des applications actives avec plus de finesse que les outils existants ne le permettaient. Les mesures collectées à l'aide de ce prototype sont indispensables pour construire les modèles qui vont être présentés dans les chapitres suivants.

Cette partie instrumentation m'a pris beaucoup de temps. D'une part, je n'étais pas familière avec le fonctionnement précis du noyau Linux. Et d'autre part, les 4 plates-formes que j'ai essayé de mesurer (*ANTS*, *Magician*, *PLAN* et *Netscript*) n'étaient que très peu documentées, ou seulement au niveau utilisateur sans donner de détails sur les choix d'implémentation.

Ce travail a mis en lumière le gouffre qui existe actuellement entre les spécifications idéales des documents d'architecture (où l'on peut théoriquement identifier avec précision l'entité responsable d'une consommation de ressource CPU, mémoire ou réseau) et les plates-formes souvent concrètement implémentées en Java qui sont beaucoup moins contrôlables !

Chapitre 4

Modélisation des nœuds et modélisation simplifiée des applications

Sommaire

4.1	**Modélisation simplifiée des applications actives (modèle M1)**	**65**
4.2	**Modélisation des nœuds actifs et prédictions**	**67**
4.3	**De la théorie à la pratique avec Linux**	**68**
4.4	**Calibration des nœuds**	**69**
4.5	**Résultats de prédiction**	**71**

4.1 Modélisation simplifiée des applications actives (modèle M1)

Comme présenté dans le chapitre 1, différents environnements d'exécution existent, développés avec divers langages (Java ou OCaml par exemple) et supportés par différents systèmes d'exploitation actifs. Mais les animateurs du programme DARPA fédérant la plupart des projets de recherche américains sur les réseaux actifs encouragent la création d'une interface standard au travers de laquelle les systèmes d'exploitation actifs proposeraient leurs services aux environnements d'exécution [Pet01]. Ainsi, à l'avenir, toutes les requêtes faites par un environnement d'exécution au système d'exploitation actif se feront par l'intermédiaire de fonctions d'interface standardisées (parfois abrégé en « FIS » dans la suite de ce document). Considérant que cette interface sera le seul composant standardisé de l'architecture, j'ai choisi d'observer le comportement des applications à ce niveau comme l'illustre la figure 4.1. Rappelons que les applications actives s'exécutent en mode utilisateur dans un environnement d'exécution mais sollicitent de temps en temps des services du système d'exploitation actif du nœud grâce aux FIS (par exemple, accès à des données enregistrées sur le disque dur ou envoi et réception de paquets sur le réseau). Un observateur situé au niveau de cette interface voit donc le comportement de l'application active comme une série de transitions entre des FIS :

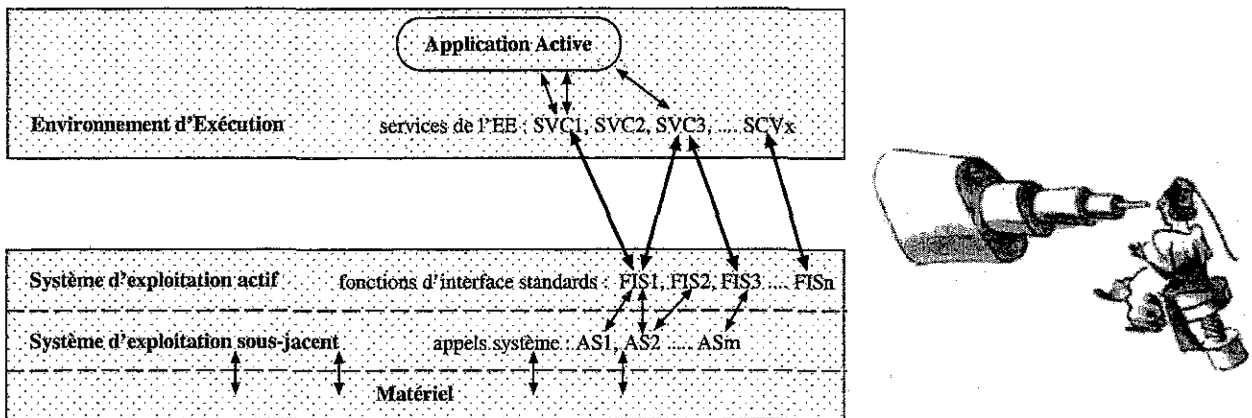


FIG. 4.1 – Point d'observation du comportement des applications actives

à partir d'un état initial, l'application s'exécute en mode utilisateur dans l'EE pendant un certain temps, puis prend un certain temps pour exécuter une fonction d'interface standard en mode privilégié avant de revenir à l'EE, repartir exécuter une autre fonction d'interface standard et ainsi de suite jusqu'à ce que le paquet actif ait été traité. En première approximation, les éléments caractérisant une application active seront donc les transitions qu'elle effectue entre l'EE, considéré comme une boîte opaque, et le système d'exploitation actif également assimilé à une boîte noire. Par exemple, une application active pourrait être caractérisée par la séquence suivante : « état initial - FIS_3 - FIS_16 - état final ».

Mais même si les applications actives sont censées rester des programmes relativement simples, il existe plusieurs chemins possibles dans leur code (selon par exemple que l'on passe dans la branche « alors » ou dans la branche « sinon » d'une conditionnelle). Lors d'une exécution ultérieure (traitement d'un autre paquet actif), la même application active appellera peut-être la fonction FIS_5 au lieu de FIS_16. Afin de tenir compte de cette variabilité, l'application active est exécutée plusieurs fois (de nombreux paquets sont traités) par son développeur de façon à refléter tous ses comportements possibles et une trace d'exécution est générée. Cette trace est ensuite analysée pour dégager la liste de séquences de FIS représentant l'application ; pour chaque séquence, on note la probabilité d'apparition conformément aux comportements observés lors des multiples exécutions de l'application. Pour l'instant, le développeur de l'application a la responsabilité de mesurer son application pour l'ensemble des situations qu'il prévoit pour assurer que le modèle couvre bien le comportement de l'AA[38]. Par exemple, pour décrire le comportement de l'application *Ping*, la machine du développeur devra tour à tour émettre des Pings, en transmettre et en recevoir. Le modèle est construit à partir de la trace des exécutions (le chapitre 3 a expliqué comment la trace est construite concrètement sous Linux). Par exemple, si sur 100 paquets actifs traités 10 exhibent le comportement « état initial -

[38]Comme j'ai instrumenté les plates-formes pour la génération automatique des traces, cette étape ne demande pas un grand niveau d'expertise au développeur, contrairement à ce qui est attendu dans l'*Application Level Scheduling* (voir section 2.5.5). Cependant, cela constitue malgré tout une limitation du modèle à laquelle nous proposerons des solutions plus tard.

EE = ANTS

Informations sur les scénarios :

probabilité	séquence d'appels aux FIS
0.1	état initial - FIS_3 - FIS_16 - état final
0.23	état initial - FIS_3 - FIS_5 - état final
0.67	état initial - FIS_5 - FIS_3 - FIS_5 - état final

FIG. 4.2 – Exemple d'instanciation du modèle simplifié (M1) d'AA

FIS_3 - FIS_16 - état final », 23 exhibent un deuxième comportement (« état initial - FIS_3 - FIS_5 - état final ») et les 67 autres exhibent un troisième comportement (« état initial - FIS_5 - FIS_3 - FIS_5 - état final »), l'application sera caractérisée par une liste comprenant 3 « scénarios » comme illustré par la figure 4.2. Le modèle indique également pour quel EE l'application active a été développée.

4.2 Modélisation des nœuds actifs et prédictions

Un nœud qui nécessite 3 unités de temps (ut) pour exécuter la fonction FIS_3, 7 ut pour FIS_7, 5 ut pour FIS_5 et 16 ut pour FIS_16 est caractérisé par le vecteur <FIS_3=3ut, FIS_5=5ut, FIS_7=7ut, FIS_16=16ut>. Un autre nœud, équipé de matériel différent ou configuré différemment exhibera des temps d'exécution différents. Ce vecteur modèle de nœud sert à capturer les variabilités inhérentes aux nœuds (voir section 2.3).

Quand le nœud reçoit le modèle de l'application exemple, il peut prédire qu'il aura besoin de $3 + 16 = 19$ ut pour exécuter le premier scénario, 8 pour exécuter le second et 13 ut pour le troisième. Comme il connaît aussi la probabilité de ces scénarios, le nœud peut de plus prédire que 23 % des exécutions devraient se terminer en 8 ut, 23+67=90 % avant 13 ut et toutes en 19 ut maximum.

Mais ces prédictions ne seraient qu'une grossière approximation car elles ne tiennent pas compte du temps passé dans l'EE entre deux appels aux fonctions d'interface standard ou entre l'état initial et le premier appel à une FIS ou entre le dernier appel à une FIS et l'état final. Pour remédier à cela, des informations supplémentaires sont introduites dans le modèle du nœud : pour chaque environnement d'exécution qu'un nœud supporte, le modèle du nœud indique le temps moyen que les applications passent dans l'EE entre deux appels à des FIS ou avant le premier appel ou après le dernier, par exemple 20 ut pour ANTS et 34 ut pour Magician. Bien sûr, là encore ces valeurs varient avec chaque nœud, influencées par le matériel ou la configuration utilisée.

La figure 4.3 résume les composants du modèle du nœud exemple.

En utilisant ces informations, les nouvelles prédictions sont : 23 % des exécutions devraient durer 20+3+20+5+20 = 68 ut, 23+10=33 % devraient être terminées en 20+3+20+16+20 = 79 ut et toutes devraient finir en 20+5+20+3+20+5+20 =93 ut maximum.

Vecteur NP : performance du Nœud en mode Privilégié (temps moyen nécessaire pour exécuter les FIS) : - FIS_3 = 3 ut - FIS_5 = 5 ut - FIS_7 = 7 ut - FIS_16 = 16 ut
Vecteur NU : performance du Nœud en mode Utilisateur dans les EE (temps moyen passé dans les EE entre les appels aux FIS) : - ANTS = 20 ut - Magician = 34 ut

FIG. 4.3 – Exemple d'instanciation du modèle d'un nœud actif X

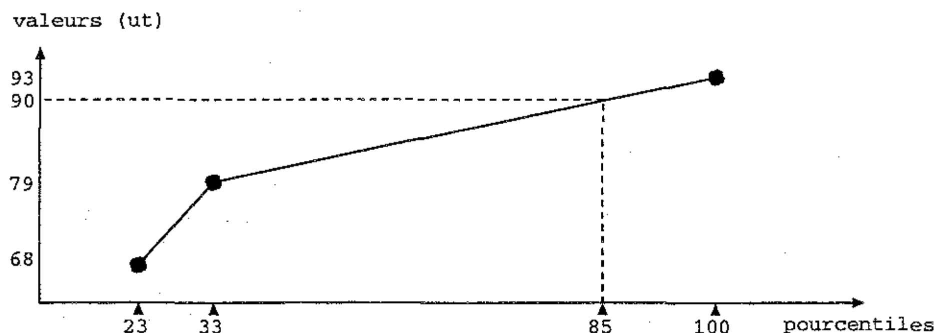


FIG. 4.4 – Exemple de calcul du 85ᵉ pourcentile avec le modèle M1

La prédiction du temps moyen d'exécution est calculée en faisant la somme des produits du temps d'exécution prévu pour un scénario par sa probabilité. Par exemple : 0.23*68 + 0.67*93 + 0.1*79 = 85.85 ut.

On a déjà prévu que le 33ᵉ pourcentile (temps au bout duquel 33 % des exécutions se terminent) vaudrait 79 ut, et que le 100ᵉ vaudrait 93 ut. Pour calculer le 85ᵉ pourcentile, on suppose que la distribution est uniforme entre le 33ᵉ et 100ᵉ pourcentiles[39]. En linéarisant, on obtient alors 89.9 ut comme valeur pour le 85ᵉ pourcentile (voir figure 4.4).

4.3 De la théorie à la pratique avec Linux

Au début de ce travail de recherche (début 1999), aucun système d'exploitation actif ne fournissait encore les fonctions d'interface standard. Pour évaluer malgré tout la solution exposée précédemment, j'ai décidé de considérer Linux comme un système d'exploitation actif et de confondre les FIS avec les appels système fournis par Linux comme indiqué sur la figure 4.5. À partir de maintenant, nous ne parlerons donc plus de fonc-

[39]Cette hypothèse est fausse mais nous n'en avons pas trouvé qui donne de meilleurs résultats.

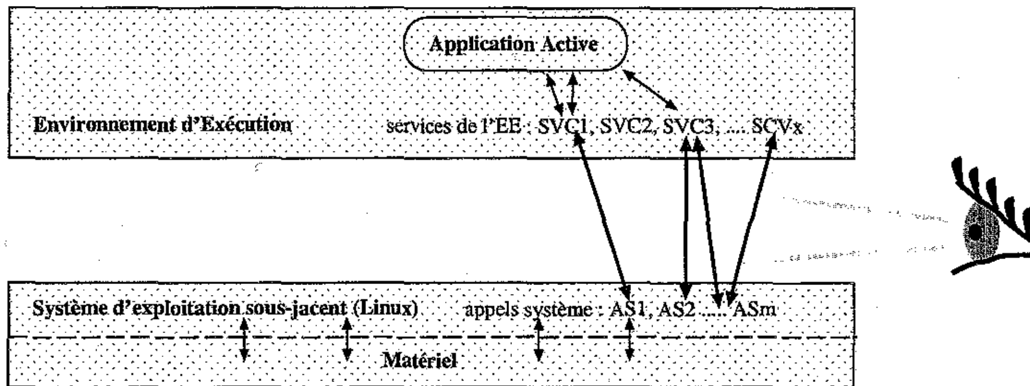


FIG. 4.5 – Point d'observation des AA dans la pratique

tions d'interface standard mais d'appels système (abrégé en AS dans les schémas). Le chapitre 3 a exposé comment le temps passé dans et entre les appels système peut être mesuré de façon fine. Dans le reste du document, je n'utiliserai plus « ut », le temps processeur sera exprimé en cycles d'horloge (« ch » dans les schémas).

4.4 Calibration des nœuds

Nous avons introduit une description d'un nœud actif au paragraphe 4.2 sans indiquer comment obtenir une instanciation du modèle. Nous allons maintenant préciser les processus dits « de calibration » nécessaire pour capturer les performances d'un nœud à la fois en mode privilégié et en mode utilisateur.

4.4.1 Performances du système d'exploitation actif

Le premier vecteur du modèle d'un nœud, celui qui indique les performances du nœud en mode privilégié (vecteur NP sur la figure 4.3), est très simple à obtenir. Il suffit de monitorer le nœud pendant qu'il exécute (plusieurs fois, afin d'obtenir des temps d'exécution moyens) un petit programme (*benchmark*) ne réalisant aucune tâche utile mais invoquant tous les appels système pouvant être utilisés par les applications actives. Bien sûr, des valeurs de paramètres sont fixées arbitrairement pour ces appels. Nous allons voir que c'est problématique lors de l'utilisation du modèle M1 ; mais cet inconvénient sera très estompé avec les versions suivantes du modèle d'application (modèles M2 et M3, que nous verrons dans le chapitre suivant).

4.4.2 Performances des environnements d'exécution

Le second vecteur du modèle d'un nœud (vecteur NU), celui qui indique le temps moyen passé dans chacun des EE entre deux appels système, est obtenu en monitorant le nœud pendant qu'il exécute une charge de travail significative pour chaque EE. Le plus délicat est de définir cette « charge de travail significative ». Si l'on s'inspire de ce qui

est fait dans les *benchmarks* classiques (voir annexe D), les deux principales possibilités pour le choix de la charge de calibration sont :

- utiliser une charge réelle formée d'un mélange d'applications actives existantes,
- fabriquer un programme sans but concret mais contenant un échantillon représentatif d'appels aux fonctions utilisées par les applications actives.

Malheureusement les réseaux actifs sont un concept trop récent pour que l'on puisse dégager des classes d'applications parmi les applications existantes (pour la plupart encore seulement expérimentales). Alors pour l'heure, pour *ANTS* par exemple, nous utilisons une version modifiée de l'application de Multicast fournie avec la distribution de la plate-forme ; cette application fait appel aux principales fonctions qui seront vraisemblablement utilisées par la majorité des applications actives : réception et envoi de paquets, enregistrement et lecture d'informations dans la « mémoire » du nœud, consultation et modification de tables... Lorsque les applications seront plus nombreuses et plus variées, il faudra mettre à jour la charge de calibration pour refléter les nouvelles fonctionnalités ou les nouveaux motifs.

4.4.3 Quand (re)calibrer un nœud ?

Le processus de calibration (à la fois du système d'exploitation actif et des EE) est susceptible d'entraîner une charge non négligeable sur les nœuds. Il faut donc trouver une façon appropriée de réaliser cette calibration pour ne pas perturber le système. Plusieurs approches peuvent être envisagées.

La calibration pourrait par exemple être réalisée *off-line*, avant que le nœud ne soit ajouté au réseau. Cette solution a l'avantage de ne pas utiliser de ressources pendant le fonctionnement normal du nœud. Mais d'une part toutes les fonctions ne peuvent pas être calibrées quand la machine est isolée (essentiellement les fonctions réseaux) et d'autre part cette calibration pourrait perdre de sa justesse si la configuration du système était modifiée ultérieurement.

Comme la plupart des nœuds doivent encore être redémarrés après des changements de configuration importants, une alternative est de réaliser la calibration automatiquement au moment du re-démarrage sur le réseau, ce qui résout les problèmes précédents. Un premier inconvénient est un allongement, que l'on peut deviner considérable, du temps de re-démarrage. De plus, on s'oriente actuellement vers des systèmes où des modifications de configuration peuvent se faire dynamiquement (surtout avec les réseaux actifs !), sans avoir à redémarrer la machine. Une calibration au re-démarrage ne capture donc pas tous les changements de performance possibles.

Une troisième solution consiste à réaliser une première calibration *off-line*, ou au démarrage sur le réseau et de réaliser ensuite des ajustements pendant que le nœud fonctionne. On pourrait même envisager de modifier automatiquement la fréquence des ajustements en fonction de la variance observée entre deux calibrations successives : plus la variance entre deux calibrations successives diminue (respectivement augmente), plus les calibrations s'espacent (respectivement se rapprochent). Bien sûr cette méthode a l'inconvénient d'utiliser des ressources pendant le fonctionnement du nœud. Mais on pourrait diviser la calibration en plusieurs programmes plus petits et n'exécuter que ceux qui nécessitent un ajustement.

EE ou appels système	Green	Black	Yellow	Blue	Red
ANTS	15 189	16 074	16 537	28 889	15 946
Magician	98 665	63 672	109 476	84 534	61 905
2 (fork)	52 692	64 462	96 252	82 360	59 356
3 (read)	9 796	10 676	10 158	17 771	10 264
4 (write)	13 247	15 194	14 754	29 138	14 128
5 (open)	16 448	15 014	13 790	20 113	12 514
6 (close)	8 633	10 465	10 387	16 849	9 623
9 (link)	16 474	17 176	17 818	23 816	16 385
10 (unlink)	15 043	16 068	14 852	22 546	15 540
37 (kill)	9 019	10 627	10 092	17 475	10 120
45 (brk)	9 253	11 672	10 403	18 649	10 899
78 (gettimeofday)	8 837	10 458	9 773	17 282	9 879
102 (socketcall)	16 692	16 261	18 030	23 236	16 358
106 (stat)	12 079	13 943	12 652	19 855	12 559
107 (lstat)	10 096	12 067	12 124	18 990	11 542
108 (fstat)	8 637	10 402	10 038	17 226	9 972
114 (wait4)	9 939	11 249	10 856	18 771	10 545
119 (sigreturn)	9 390	10 497	11 154	17 252	10 160
140 (_llseek)	8 731	10 171	9 674	17 128	9 504
174 (rt_sigaction)	9 253	10 721	10 100	17 404	10 326
175 (rt_sigprocmask)	8 899	10 144	9 878	24 099	9 728
179 (rt_sigsuspend)	11 892	12 603	12 556	20 632	12 128

TAB. 4.1 – Résultats de la calibration des nœuds (en cycles d'horloge)

La solution que j'ai adoptée dans le prototype est de lancer manuellement la calibration avant les expériences. Le tableau 4.1 indique, pour chacun des nœuds du réseau de test, les temps moyens d'exécution (en cycles d'horloge) pour différents appels système ainsi que le temps moyen passé dans les EE entre les appels système lors de la phase de calibration. Ces résultats mettent en évidence la variabilité qui existe d'une machine à une autre.

4.5 Résultats de prédiction

Pour tester cette première modélisation des nœuds et des applications actives, j'ai monitoré l'exécution de 4 applications actives (*Ping* et *Multicast* pour *ANTS* et *Ping* et *Route* pour *Magician*) sur les 5 nœuds du réseau de test.

À partir des traces d'exécution, j'ai déterminé, pour chaque nœud et chaque application, le temps moyen d'exécution ainsi que le temps au bout duquel 80 % des exécutions se terminaient (80e pourcentile). J'ai aussi relevé les 85e, 90e, 95e et 99e pourcentiles. Par exemple, les valeurs observées pour le *Ping* d'*ANTS* sur la machine *Blue* sont (en cycles d'horloge) :

- moyenne : 1 499 479
- 80e pourcentile : 2 403 600
- 85e pourcentile : 2 900 283
- 90e pourcentile : 3 059 091
- 95e pourcentile : 3 116 139
- 99e pourcentile : 3 963 778

À partir des traces d'exécution, j'ai également construit, toujours pour chaque nœud, le modèle de chacune des applications. En utilisant le modèle d'une application construite sur un nœud X et le modèle d'un autre nœud Y, j'ai prédit le temps moyen d'exécution de l'application sur le nœud Y. J'ai fait de même pour les hauts pourcentiles. Par exemple, en utilisant la trace d'exécution du *Ping* d'*ANTS* relevée sur le nœud *Yellow*, j'ai construit un modèle pour l'application semblable à celui présenté par la figure 4.2. D'autre part, la phase de calibration initiale des nœuds fournit le temps moyen d'exécution des appels système et d'une application *ANTS* de calibration sur la machine *Blue* (Cf tableau 4.1). À partir de ces deux informations, et selon ce qui a été expliqué au paragraphe 4.2, on prédit les temps d'exécution suivants pour le *Ping* d'*ANTS* sur *Blue* :

- moyenne : 1 159 126
- 80e pourcentile : 1 487 446
- 85e pourcentile : 2 013 556
- 90e pourcentile : 2 619 532
- 95e pourcentile : 3 319 532
- 99e pourcentile : 4 945 168

J'ai ensuite comparé les valeurs prédites aux valeurs réelles. Le tableau 4.2 présente un extrait des résultats. Pour chaque prédiction, l'erreur est indiquée en pourcentage par rapport à la valeur réelle. Les chiffres en gras indiquent les erreurs n'entrant pas dans la fourchette de précision recherchée (±10% pour la moyenne et ±20% pour les hauts pourcentiles). Par exemple, nous avons prédit un temps moyen d'exécution du *Ping* d'*ANTS* sur *Blue* de 1 159 126 cycles d'horloge alors qu'en réalité on mesure 1 499 479. Nous commettons donc une erreur d'estimation de (1 159 126-1 499 479)*100/1 499 479 = -23 % comme l'indique le tableau 4.2 à la ligne 2 (*ANTS*, *Ping*, à partir des obvervations sur *Yellow*, prédictions pour le nœud *Blue*), colonne 4 (erreur en % sur la moyenne). Comme l'indique la figure 4.6, la distribution prédite du temps d'exécution a souvent un aspect moins lisse que la distribution réelle (moins il y a de scénarios dans une application et plus les « marches » dans la distribution simulée sont « raides »[40]). L'erreur de prédiction peut être parfois positive et parfois négative.

À titre de comparaison, le tableau 4.3 indique les erreurs des prédictions obtenues en transposant simplement sur le nœud Y le temps moyen d'exécution observé sur le nœud X. Par exemple, on mesure que sur le nœud *Yellow* le *Ping* d'*ANTS* met en moyenne 853 547 cycles d'horloge à s'exécuter. Si on suppose qu'il mettra le même temps CPU sur *Blue*, on commettra une erreur de (853 547-1 499 479)*100/1 499 479 = -43 %, comme on peut le retrouver à la seconde ligne et cinquième colonne du tableau 4.3.

[40]Pour s'en convaincre, il suffit de reprendre l'exemple de la figure 4.2 et de remplacer le scenario à la probabilité de 67% par 2 scenari de 30 et 37%. Alors, au lieu d'avoir les 3 seuils (23e, 33e et 100e pourcentiles) calculés au paragraphe 4.2, on aurait un seuil de plus, ce qui affine la distribution du temps d'exécution prévu.

S.C.D. - U.H.P. NANCY 1
BIBLIOTHEQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

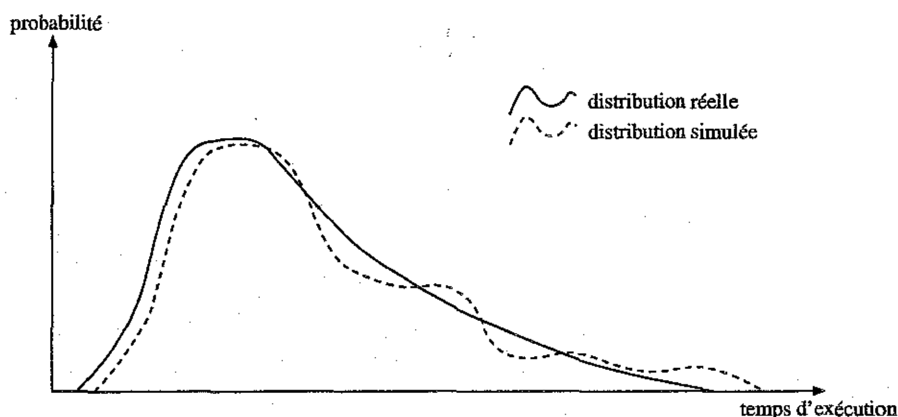


FIG. 4.6 – Exemple d'écart entre distributions réelle et simulée du temps d'exécution

Environnement d'Exécution	Application Active	Nœud X à partir duquel le modèle est construit	Nœud Y pour lequel on établit les prédictions	Erreur (en %) de prédiction de la moyenne	Erreur (en %) de prédiction du 80e pourcentile	Erreur (en %) de prédiction du 85e pourcentile	Erreur (en %) de prédiction du 90e pourcentile	Erreur (en %) de prédiction du 95e pourcentile	Erreur (en %) de prédiction du 99e pourcentile	Moyenne des valeurs absolues des erreurs de prédiction des hauts pourcentiles (en %)
Ants	Ping	Yellow	Blue	**-23**	**-38**	**-31**	-14	7	**25**	**23**
		Yellow	Green	**-23**	-16	-20	**22**	**28**	**37**	**25**
		Yellow	Black	**-20**	**-27**	-20	**-21**	**-23**	-19	**22**
		Yellow	Red	**-26**	**-24**	**-30**	**-25**	**-23**	**-34**	**27**
		Yellow	Yellow	**-14**	**-29**	**-27**	**32**	**48**	**37**	**35**
Magician	Route	Green	Blue	**-20**	**-27**	**-21**	**-32**	-20	**-25**	**25**
		Green	Green	-10	**-22**	**-22**	**-21**	-19	**22**	**21**
		Green	Black	**-28**	**-32**	**-31**	**-40**	**-43**	**-34**	**36**
		Green	Red	**-21**	**-28**	**-27**	**-36**	**-40**	**-33**	**33**
		Green	Yellow	**-30**	**-43**	**-38**	**-40**	**-36**	**-25**	**37**

TAB. 4.2 – Erreur (en %) des prédictions avec le modèle M1

Environnement d'Exécution	Application Active	Nœud X servant de base	Nœud Y pour lequel on établit les prédictions	Erreur (en %) de prédiction de la moyenne	Erreur (en %) de prédiction du 80e pourcentile	Erreur (en %) de prédiction du 85e pourcentile	Erreur (en %) de prédiction du 90e pourcentile	Erreur (en %) de prédiction du 95e pourcentile	Erreur (en %) de prédiction du 99e pourcentile	Moyenne des valeurs absolues des erreurs de prédiction des hauts pourcentiles (en %)
Ants	Ping	Yellow	Blue	**-43**	**131**	**193**	-3	**947**	**-68**	**268**
		Yellow	Green	**37**	**-78**	**260**	**-28**	**-74**	**-91**	**106**
		Yellow	Black	**-74**	**428**	**376**	**-92**	**2484**	**-25**	**681**
		Yellow	Red	**60**	**-63**	**909**	**-49**	**82**	**-100**	**240**
Magician	Route	Green	Blue	**-60**	**-92**	**47**	**-90**	-6	**262**	**99**
		Green	Black	**-68**	**-60**	**-58**	**-49**	**-44**	**42**	**51**
		Green	Red	**-31**	**-57**	**-83**	**-91**	**82**	**96**	**82**
		Green	Yellow	**-91**	**12**	**-11**	**72**	**90**	**222**	**81**

TAB. 4.3 – Erreur (en %) des prédictions « naïves »

Le tableau E.1 de l'annexe E synthétise les résultats obtenus avec ces deux méthodes de prédictions (« naïve » ou utilisant le modèle M1) pour l'ensemble des nœuds du réseau de test et les 4 applications actives. Il indique qu'en moyenne on commet un erreur de 177 % dans la prédiction du temps moyen d'exécution d'une application si on utilise la méthode directe. Cette erreur est en moyenne réduite à 14 % si on utilise le modèle M1. De même, l'erreur moyenne de prédiction des hauts pourcentiles passe de 361 % à 24 %. Les résultats obtenus avec le modèle M1 sont donc bien meilleurs. Cependant, ils ne satisfont pas encore les critères fixés de ne pas s'éloigner de la réalité de plus de 10 % pour la moyenne et 20% pour les hauts pourcentiles.

Les modèles doivent être plus précis

Une des raisons pour laquelle les prédictions obtenues avec le modèles M1 sont encore décevantes est qu'aucune distinction n'est faite entre les différentes transitions : une application active donnée peut passer beaucoup plus (ou moins) de temps dans l'environnement d'exécution entre 2 appels système que ce que le modèle prévoit. Il ne faut pas oublier que les temps d'exécution moyens dans l'EE indiqués dans les modèles des nœuds ont été obtenus par observation d'une charge de calibration mêlant plusieurs AA. De même, les paramètres utilisés par une application active lors des invocations des appels système peuvent être différents de ceux qui ont été fixés lors de l'écriture du programme de calibration du premier vecteur du modèle des nœuds, conduisant à un temps d'exécution différent. Pour résumer, le problème vient donc du fait que les seules informations temporelles utilisées pour calculer les prédictions sont celles du modèle des nœuds, issues de *benchmarks* « arbitraires ».

Pour corriger ce problème, des informations supplémentaires doivent être introduites dans le modèle d'une application active. Le chapitre suivant présente les nouvelles versions du modèle, obtenues en introduisant des informations temporelles spécifiques à l'application.

Chapitre 5

Améliorations de la modélisation des applications

Sommaire

5.1 Modèle d'application plus complet (modèle M2) 77
5.2 Modèle semi-stochastique (modèle M3) 81
5.3 Nœud de référence 84
5.4 Résumé 87

Pour corriger le manque de précision du modèle M1 présenté au chapitre précédent, nous introduisons des informations temporelles dans le modèle des applications actives. Ce chapitre présente les nouvelles versions de modèle obtenues, leur fonctionnement et leurs résultats. Il introduit aussi un élément permettant un meilleur passage à l'échelle des techniques proposées.

5.1 Modèle d'application plus complet (modèle M2)

5.1.1 Éléments additionnels du modèle

Les informations supplémentaires introduites dans le modèle d'une application active sont les suivantes : nous indiquons le temps moyen que l'application a passé

- dans chaque appel système et
- dans chaque transition

lorsque son exécution a été tracée.

Le nouveau modèle d'application (appelé M2) est illustré par un exemple avec la figure 5.1. Construit à partir des informations récoltées sur un nœud X dans une trace d'exécution, il contient :

- le nom de l'EE supportant l'application,
- la liste des scénarios (un scénario = une séquence d'appels système ayant une certaine probabilité), placée dans un vecteur AS (« Application-Scénarios »),
- le temps moyen passé dans les appels système, constituant le vecteur AP_X (« Application-mode Privilégié »,

EE = ANTS

Informations sur les scénarios (vecteur AS) :

probabilité	séquence d'appels aux FIS
0.1	état initial - AS_3 - AS_16 - état final
0.23	état initial - AS_3 - AS_5 - état final
0.67	état initial - AS_5 - AS_3 - AS_5 - état final

Temps moyen passé dans les appels système (vecteur AP_X) :
- AS_3 : 13 ch
- AS_5 : 25 ch
- AS_16 : 20 ch

Temps moyen passé entre les appels système (vecteur AU_X) :
- état initial - AS_3 : 30 ch
- état initial - AS_5 : 50 ch
- AS_3 - AS_16 : 60 ch
- AS_3 - AS_5 : 20 ch
- AS_5 - AS_3 : 55 ch
- AS_16 - état final : 45 ch
- AS_5 - état final : 40 ch

FIG. 5.1 – Exemple d'instanciation du modèle M2 d'une AA sur un nœud X

- le temps moyen passé dans les transitions entre les appels système utilisés, constituant le vecteur AU_X (« Application-mode Utilisateur »).

5.1.2 Obtention des prédictions

Pour prédire le temps d'exécution d'un paquet de l'application active de l'exemple de la figure 5.1, on n'a seulement besoin du modèle de l'application (alors que le modèle M1 mêlait informations du modèle du nœud et informations du modèle d'application). On prévoit que 10 % des paquets vont suivre le premier scénario et mettre en moyenne 30 ch (temps d'exécution de la transition « état initial - AS_3 ») + 13 ch (pour AS_3) + 60 ch (AS_3 à AS_16) + 20 ch + 45 ch soit 168 ch. 23 % des paquets vont mettre 128 ch et les 67 % restants vont mettre 228 ch.

Pour prédire ensuite le temps moyen d'exécution et les hauts pourcentiles, on procède comme pour le modèle M1 (voir page 67).

5.1.3 Adaptation du modèle d'un nœud à un autre

Mais bien sûr, comme je l'ai expliqué en exhibant les sources de variabilité page 34, les indications de temps contenues dans le modèle M2 ne sont valables que pour le nœud sur lequel l'exécution de l'application active a été tracée (nœud X dans l'exemple). D'autre part, il n'est pas envisageable de monitorer, avant son déploiement réel, l'exécution d'une

AA sur tous les nœuds où elle pourrait s'exécuter (car les nœuds peuvent être très nombreux et surtout de nouveaux nœuds peuvent être introduits à tout moment, ou les existants peuvent être reconfigurés). Une fois le modèle d'une application construit à partir des observations réalisées sur un nœud, il faut donc trouver un moyen de « traduire » les valeurs temporelles stockées dans le modèle et significatives sur le nœud de traçage uniquement en valeurs significatives sur un nœud aux performances différentes sans avoir à y exécuter l'application (on ne veut faire qu'une seule fois la génération des traces).

Nous avons d'abord fait l'hypothèse que pour un scénario donné, une application active appelle les fonctions d'interface standards avec le même type de paramètres sur un nœud X et sur un nœud Y. De même, nous avons supposé qu'elle réalise le même type de traitement dans l'EE.

Par conséquent, si une application active met en moyenne 13 cycles d'horloge pour exécuter l'appel système AS_3 sur une machine X dont les *benchmarks* indiquent qu'elle met en moyenne 3 cycles d'horloge pour cet appel système, on peut supposer que sur une autre machine Y, pour laquelle les *benchmarks* indiquent un temps moyen d'exécution de AS_3 de 6 cycles d'horloge, l'application mettra 6*13/3 = 26 cycles d'horloge pour exécuter le même appel que sur la machine X. En utilisant ainsi les informations comprises dans le premier vecteur du modèle du nœud X[41] (vecteur NP_X) et celles comprises dans le premier vecteur du modèle du nœud Y (vecteur NP_Y), on peut « proportionnaliser » les valeurs du modèle de l'application mesurée sur le nœud X indiquant les temps moyens passés dans les appels système (vecteur AP_X) et obtenir des temps « significatifs » pour le nœud Y (vecteur AP_Y). De même, on traduit les temps moyens passés entre les appels système sur le nœud X (vecteur AU_X) en utilisant le deuxième vecteur du modèle de chacun des nœuds (vecteurs NU_X et NU_Y). On obtient alors un modèle de l'application indiquant des valeurs adaptées au nœud Y.

5.1.4 Résultats

Nous avons évalué ce nouveau modèle de la même façon que pour le premier modèle (voir la section 4.5). Le tableau 5.1 présente un extrait des résultats détaillés et le tableau E.1 page 154 synthétise l'ensemble des résultats.

Pour les applications ANTS, les prédictions sont meilleures que celles obtenues avec le premier modèle et satisfont les exigences de précision souhaitée. En moyenne, on commet une erreur de 5 % dans la prédiction du temps moyen d'exécution des applications et une erreur moyenne de 8 % dans la prédiction des hauts pourcentiles du temps d'exécution. En revanche les prédictions pour les applications Magician ne sont toujours pas satisfaisantes en ce qui concerne les hauts pourcentiles : l'erreur moyenne dans leurs prédictions est de 23 %.

Cette différence peut venir du fait que les applications ANTS produisent généralement plus de scénarios que les applications Magician (plates-formes programmées différemment) ; l'intervalle sur lequel on linéarise pour rechercher un pourcentile entre 2 pourcentiles déjà prévus est donc plus petit dans le cas d'ANTS que dans le cas de

[41] Rappel : le modèle des nœuds est défini page 67 et illustré par la figure 4.3.

Environnement d'Exécution	Application Active	Nœud X à partir duquel le modèle est construit	Nœud Y pour lequel on établit les prédictions	Erreur (en %) de prédiction de la moyenne	Erreur (en %) de prédiction du 80e pourcentile	Erreur (en %) de prédiction du 85e pourcentile	Erreur (en %) de prédiction du 90e pourcentile	Erreur (en %) de prédiction du 95e pourcentile	Erreur (en %) de prédiction du 99e pourcentile	Moyenne des valeurs absolues des erreurs de prédiction des hauts pourcentiles (en %)
Ants	Ping	Yellow	Blue	-3	11	9	13	11	19	13
		Yellow	Green	2	10	4	6	5	11	7
		Yellow	Black	**18**	-10	-6	8	13	15	10
		Yellow	Red	4	14	14	10	10	15	12
		Yellow	Yellow	0	1	-5	-4	-6	-3	4
Magician	Route	Green	Blue	-11	**23**	20	6	-14	-19	16
		Green	Green	0	**34**	**32**	**32**	17	14	**26**
		Green	Black	-6	15	5	-7	-17	**-28**	14
		Green	Red	3	**21**	11	0	-13	**-27**	15
		Green	Yellow	-5	**-26**	**-22**	-18	-16	**-25**	**22**

TAB. 5.1 – Erreur (en %) des prédictions avec le modèle M2

S.C.D. - U.H.P. NANCY 1
BIBLIOTHEQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

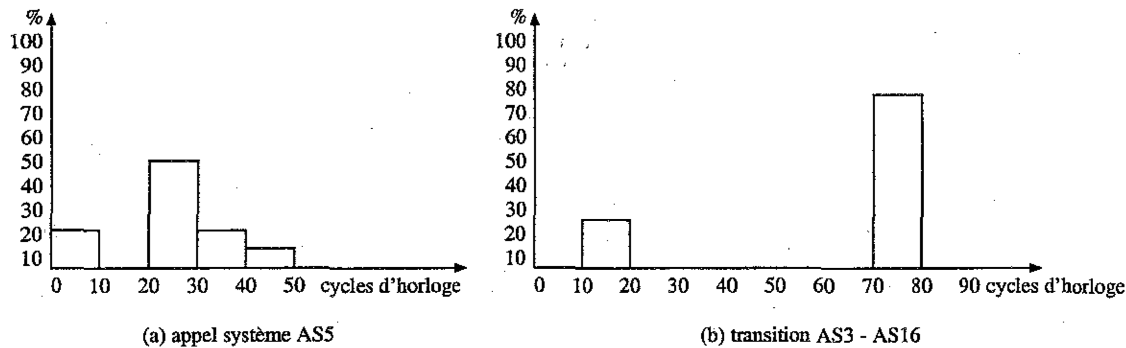


FIG. 5.2 – Distribution des temps d'exécution sous forme d'histogrammes

Magician et l'erreur de prédiction est donc réduite.

Pour obtenir des prédictions des hauts pourcentiles correctes, il faudrait prédire la distribution complète. Donner seulement le temps moyen d'exécution des éléments (appels système et transitions) ne suffit donc pas et il faut plutôt donner leur distribution (la convolution des distributions des éléments fournira la distribution du temps d'exécution des paquets).

5.2 Modèle semi-stochastique (modèle M3)

5.2.1 Ajout d'informations

Pour encore améliorer la précision des prédictions, nous avons essayé d'intégrer encore plus d'informations dans le modèle des applications actives : pour chaque appel système et chaque transition entre appels système, nous indiquons la distribution du temps d'exécution au lieu de simplement la moyenne comme précédemment. Dans un premier temps, nous espérions pouvoir représenter les distributions observées à l'aide de distributions classiques. Mais l'irrégularité trop grande des distributions observées ne l'a pas permis et nous avons donc choisi de les représenter par des histogrammes.

Par exemple, au lieu d'indiquer « AS_5 : 25 ch », le modèle d'une application présentera un histogramme comme celui de la figure 5.2 indiquant que dans 20 % des cas, l'appel système AS_5 s'exécute en moins de 10 cycles d'horloge, dans 50 % des cas entre 20 et 30 ch etc.

Lors de la transmission du modèle de l'application d'un nœud à un autre, les modèles des nœuds sont utilisés comme précédemment (5.1.3) pour dilater ou contracter l'axe des cycles d'horloge des histogrammes. La figure 5.3 illustre le résultat de la transformation des histogrammes de la figure 5.2 en utilisant les modèles de deux nœuds exemple.

5.2.2 Obtention des prédictions par simulation

L'impossibilité de représenter les distributions de temps d'exécution des éléments des scénarios par des distributions statistiques classiques rend toute solution analytique très lourde à mettre en œuvre et coûteuse à exécuter. En supposant que les histogrammes

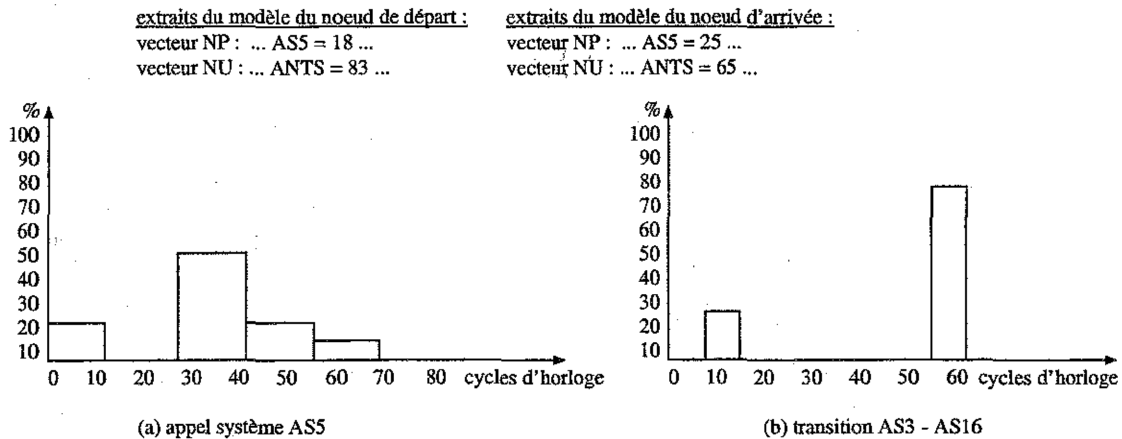


FIG. 5.3 – Transformation des histogrammes

comportent tous 5 barres (que ce soit des histogrammes représentant la distribution du temps d'exécution des appels système ou représentant la distribution du temps passé entre les appels système), et que l'application ne présente que 2 scénarios de 12 appels système chacun, il y a déjà $2 * 5^{12+13}$ possibilités à classer. Or on a généralement plus de 2 scénarios et certains peuvent comporter plus d'une cinquantaine d'appels système.

Pour prédire le temps d'exécution des paquets actifs à partir du modèle M3 j'utilise un simulateur. Chaque passage dans le simulateur calcule le temps d'exécution d'un paquet : tout d'abord un scénario est choisi dans la liste des scénarios par un test de Monte-Carlo ; puis pour chaque élément du scénario (appel système ou transition entre deux appels système), un nouveau test de Monte-Carlo est utilisé pour choisir une barre dans l'histogramme décrivant la distribution du temps d'exécution de l'élément. La somme des temps d'exécution tirés pour chaque élément fournit un temps d'exécution simulé pour le traitement d'un paquet. Après plusieurs simulations, on peut calculer le temps d'exécution simulé moyen, ainsi que les hauts pourcentiles.

Au lieu de choisir un scénario dans la liste des scénarios à l'aide d'un test de Monte-Carlo, une autre solution envisagée est de simuler les scénarios les uns après les autres en tenant compte de leur probabilité : par exemple, simuler le premier scénario 10 fois puis le second 23 fois et le dernier 67 fois. Mais j'ai choisi la méthode du test de Monte-Carlo car elle permet d'avoir simulé les différents scénarios en respectant plus ou moins leur probabilité quel que soit le nombre de tours effectués par le simulateur. Avec la méthode « directe », si on décide d'arrêter le simulateur au bout de 20 tours parce qu'on ne peut plus attendre sa réponse plus longtemps, le résultat sera erroné puisque le scénario 3 ne sera pas du tout représenté.

5.2.3 Résultats

Le tableau 5.2 présente un extrait des résultats obtenus en utilisant des histogrammes à 5 barres et en faisant 10000 tours de simulateur.

Pour ANTS, les résultats sont du même ordre avec ceux obtenus avec le modèle

Environnement d'Exécution	Application Active	Nœud X à partir duquel le modèle est construit	Nœud Y pour lequel on établit les prédictions	Erreur (en %) de prédiction de la moyenne	Erreur (en %) de prédiction du 80e pourcentile	Erreur (en %) de prédiction du 85e pourcentile	Erreur (en %) de prédiction du 90e pourcentile	Erreur (en %) de prédiction du 95e pourcentile	Erreur (en %) de prédiction du 99e pourcentile	Moyenne des valeurs absolues des erreurs de prédiction des hauts pourcentiles (en %)
Ants	Ping	Yellow	Blue	5	14	9	9	14	19	13
		Yellow	Green	0	15	15	15	15	12	14
		Yellow	Black	7	11	14	15	13	14	13
		Yellow	Red	3	16	10	13	13	14	13
		Yellow	Yellow	1	-3	-6	-7	-2	10	6
Magician	Route	Green	Blue	-4	-7	15	-1	-5	5	7
		Green	Green	1	3	6	7	1	5	4
		Green	Black	-6	6	7	8	3	-3	5
		Green	Red	-7	6	6	-1	0	10	5
		Green	Yellow	-6	-7	4	8	5	2	5

TAB. 5.2 – Erreur (en %) des prédictions avec le modèle M3

M2 (tableau 5.1) : le temps moyen d'exécution des applications est prédit avec une erreur moyenne de 9 % et les hauts pourcentiles avec une erreur de 10 %. En revanche, les prédictions pour les applications Magician sont grandement améliorées et entrent à présent dans la fourchette de précision désirée : l'erreur moyenne de prédiction du temps moyen d'exécution est de 6 % et celle des hauts pourcentiles est de 13 %. Les prédictions de la moyenne sont légèrement moins bonnes qu'avec le modèle M2 mais en augmentant le nombre de barres par histogramme ou le nombre de tours de simulateur, on peut améliorer les prédictions (et elles peuvent alors être même meilleures qu'avec le modèle M2).

5.2.4 Compromis précision-légèreté

Comme il était impossible de représenter les distributions du temps d'exécution des éléments des scénarios par des distributions statistiques classiques, nous les avons représenté par des histogrammes, comme expliqué précédemment. Mais cela rend le modèle assez volumineux. La seule optimisation adoptée pour l'instant consiste à ne représenter que les barres qui ne sont pas vides (par exemple, l'histogramme de la figure 5.2 est représenté par : 0-10 : 20, 20-30 : 50, 30-40 : 20, 40-50 : 10. L'intervalle 10-20 n'est pas indiqué.). Une façon d'alléger le modèle pour le transport entre nœuds est de réduire le nombre de barres décrivant les histogrammes (paramètre ajustable à la construction des histogrammes à partir de la trace d'exécution). Par exemple, le modèle du *Ping* de *Magician* sur *Blue* avec 200 barres par histogramme est 2 fois plus grand que lorsqu'on réduit à 20 barres par histogramme (rappel : les barres « vides » ne sont pas représentées ce qui explique pourquoi le modèle avec 20 barres n'est pas 10 fois plus petit : le modèle à 200 barres comportait beaucoup de « trous »). Malheureusement la finesse des prédictions diminue également. D'autre part le modèle fournit de meilleurs résultats si on laisse le simulateur effectuer un nombre assez grand de tours, en augmentant la couverture de la simulation, cela améliore aussi la précision (mais cela allonge évidemment le temps d'obtention des prédictions). À titre d'exemple, la table 5.3 compare les erreurs obtenues avec différentes combinaisons « nombre de barres par histogramme / nombre de tours du simulateur » (pour *Ping* d'*ANTS* sur *Green*). Le calcul des prédictions a été fait sur un Pentium Pro à 547 MHz.

Dans un premier temps, nous avons choisi d'ignorer ces problèmes. En effet, on pourra négliger le coût de transport du modèle s'il n'est transporté qu'une fois et ensuite beaucoup utilisé. Idem pour le coût de la prédiction. On peut aussi imaginer un ensemble de nœuds dédiés dont la tâche serait de faire les calculs et simulations pour les autres (les prédictions peuvent être établies sur n'importe quelle machine du moment qu'elle dispose du modèle du nœud client et du modèle de l'application active).

5.3 Nœud de référence

Un inconvénient de la solution proposée (modèle M2 ou modèle M3) est que les nœuds sur lesquels les modèles des applications sont construits doivent diffuser leur modèle de nœud à tous les autres nœuds afin que ces derniers puissent réaliser les adaptations

nombre de barres	5	5	5	50
nombre de répétitions	100	10000	100000	10000
taille du modèle (en octets)	136805			144778
temps de calcul des prédictions (en secondes CPU)	0.83	4.54	427.61	5.20
erreurs (en %) de prédiction...				
de la moyenne	17.54	1.48	1.02	1.08
du 80ᵉpourcentile	-29.97	7.96	5.87	6.58
du 85ᵉpourcentile	-21.56	8.78	8.12	7.34
du 90ᵉpourcentile	32.02	7.29	-6.45	-7.06
du 95ᵉpourcentile	-37.10	-9.18	-2.45	-4.36
du 99ᵉpourcentile	-28.10	1.79	3.65	4.70
des hauts pourcentiles (en moyenne)	29.75	7.00	5.31	6.01

TAB. 5.3 – Compromis entre précision et légèreté

nécessaires dans les indications temporelles des modèles des applications actives.

Pour résoudre ce problème, un nœud est choisi comme « nœud de référence » et son modèle est distribué à tous les autres nœuds. Avant de transférer le modèle d'une application entre deux nœuds X et Y, le modèle est soumis à une transformation "X-vers-référence" qui consiste à transformer les valeurs de temps contenues dans le modèle construit sur X en valeurs significatives pour le nœud de référence grâce à la règle de trois expliquée précédemment. Le nouveau modèle obtenu est alors envoyé à Y qui avant de l'utiliser pour fournir les prédictions applique une transformation inverse "référence-vers-Y". La combinaison des deux transformations permet de traduire des valeurs significatives sur X en valeurs significatives sur Y (sans que X n'aient eu à envoyer son modèle de nœud à Y).

Pour évaluer l'impact de cette double traduction, nous avons choisi *Blue* comme nœud de référence. La table 5.4 donne un extrait des résultats, pour le *Ping* d'*ANTS*. On peut constater d'une part que la précision des prédictions n'est pas affectée et d'autre part que le temps d'obtention des prédictions est à peine allongé (pour le modèle avec 5 barres par histogramme, le temps moyen de calcul des prédictions avec une seule transformation à partir d'un modèle construit sur *Yellow* est de 4.8 secondes).

Bien sûr, plus le nombre de barres par histogramme est grand et plus la double transformation du modèle prend du temps. Mais on pourrait, après la calibration d'un nœud, transformer son modèle, le « normaliser » par rapport au modèle reçu du nœud de référence une bonne fois pour toutes et les temps de calcul des prédictions resteraient alors identiques à ceux déjà présentés. Mieux, le nœud de référence n'a même pas besoin d'exister réellement. Il suffit que l'ensemble des nœuds actifs s'accordent sur une valeur par rapport à laquelle normaliser ! Les deux transformations successives se compensant, cette valeur n'a pas d'importance.

Nœud X à partir duquel le modèle est construit	Nœud Y pour lequel on établit les prédictions	Erreur (en %) de prédiction de la moyenne	Erreur (en %) de prédiction du 80e pourcentile	Erreur (en %) de prédiction du 85e pourcentile	Erreur (en %) de prédiction du 90e pourcentile	Erreur (en %) de prédiction du 95e pourcentile	Erreur (en %) de prédiction du 99e pourcentile	Moyenne des valeurs absolues des erreurs de prédiction des hauts pourcentiles (en %)	Temps de calcul des prédictions (en secondes CPU)
Yellow	Green	-2	-14	-14	-18	-12	14	14	4.91
Yellow	Black	-5	12	-14	18	14	16	15	4.92
Yellow	Red	4	-11	-11	13	10	16	12	4.93

TAB. 5.4 – Influence de la double transformation sur les prédictions de M3

5.4 Résumé

Dans ce chapitre et le précédent, nous avons présenté 3 modèles de plus en plus raffinés pour représenter une application active, ainsi qu'une façon de modéliser les nœuds actifs. Nous avons expliqué comment les modèles M1, M2 et M3 pouvaient être utilisés conjointement aux modèles de nœuds pour prédire le temps d'exécution d'une application active sur n'importe quel nœud préalablement calibré. Nous résumons ci-dessous le processus en rappelant également les différents acteurs :

Phase 1 : calibration du nœud pour construire le modèle du nœud
Chaque propriétaire de nœuds doit monitorer ses machines pendant l'exécution d'une charge de calibration pour déterminer combien de temps le nœud passe en moyenne dans chacun des différents appels système et, entre 2 appels système, dans chacun des environnements d'exécution.

Phase 2 : normalisation du modèle du nœud
Chaque nœud doit récupérer le modèle du nœud de référence ou la valeur commune de référence. Il peut être de la responsabilité du propriétaire du nœud d'aller chercher ce modèle ou cette valeur sur un serveur ou bien le gestionnaire du réseau ou un organisme tel que l'ANANA pourrait « pousser » le modèle ou la valeur de référence sur les différents nœuds. Le modèle de chaque nœud peut alors être « normalisé » (par rapport aux performances du nœud de référence ou par rapport à la valeur arbitraire commune).

Phase 3 (qui peut éventuellement se dérouler avant la phase 1) : construction du modèle de l'application active
Le développeur doit monitorer l'exécution de son application active dans l'ensemble des situations que cette application pourra rencontrer lors de son passage dans le réseau. La trace d'exécution permet de déterminer les différents scénarios possibles (c'est-à-dire les différentes suites d'appels système invoqués pour traiter un paquet actif), ainsi que leur probabilité. De la trace d'exécution on extrait également les distributions du temps passé dans chacun des différents appels système et dans l'environnement d'exécution entre 2 appels système donnés. Le développeur peut choisir la finesse avec laquelle les distributions seront représentées (compromis entre légèreté du modèle à transporter et précision des prédictions).

Phase 4 : normalisation du modèle de l'application active
Les indications de temps présentes dans le modèle d'une application active sont traduites en valeurs significatives sur le nœud de référence grâce à une règle de trois utilisant le modèle normalisé du nœud sur lequel a été récoltée la trace d'exécution ayant permis la construction du modèle de l'application active.

Phase 5 : envoi du modèle
Avant de traiter un paquet actif, un nœud doit se procurer le modèle de l'application active auquel appartient le paquet actif en question. Dans nos travaux, nous n'avons pas intégré cet envoi du modèle aux plates-formes mais on pourrait très bien imaginer un mécanisme s'inspirant d'*ANTS* où un nœud réclamerait le modèle manquant au nœud qui vient de lui envoyer le paquet. Ce module devrait s'intégrer à l'EE de gestion et supervision d'un nœud actif et ne pas être lié à une plate-forme particulière.

Phase 6 : adaptation du modèle de l'application active
Avant de pouvoir utiliser le modèle d'une application active qu'il vient de récupérer, un

		direct		M1		M2		M3	
EE	AA	moy.	h. p.	moy.	h. p.	moy.	h. p.	moy.	h. p.
Ants	Ping	**175**	**907**	**15**	**22**	5	9	9	8
Ants	Multicast	**149**	**148**	**13**	**24**	6	7	9	12
Magician	Ping	**154**	**163**	**14**	**25**	4	**24**	6	13
Magician	Route	**231**	**225**	**12**	**25**	8	**21**	6	12

TAB. 5.5 – Comparaison résumée des pouvoirs de prédiction

nœud doit transformer les indications de temps que le modèle contient en valeurs qui seront significatives pour cette machine particulière. Pour cela, il utilise son modèle de nœud et la règle de trois « inverse » de celle utilisée en phase 4.

Phase 7 : calcul des prédictions

À partir du modèle de l'application active, le nœud calcule des temps d'exécution simulés. Après plusieurs simulations, il peut estimer le temps moyen d'exécution prévu ainsi que les hauts pourcentiles.

Les prédictions réalisées grâce à la modélisation sont bien meilleures que celles obtenues en utilisant simplement le temps d'exécution observé sur un nœud pour prévoir le temps d'exécution sur une autre machine. Le tableau E.1 en annexe page 151 indique pour 4 applications actives et 5 nœuds les erreurs de prédiction obtenues selon la technique utilisée (en utilisant directement le temps d'exécution observé sur un nœud X (colonnes « direct »), ou bien en utilisant les modèles M1, M2 ou M3 construits à partir de traces d'exécution de l'application sur le nœud X pour prédire le temps d'exécution sur Y). Pour chaque méthode employée, la première colonne (« moy. ») indique la valeur absolue de l'erreur en % par rapport au temps moyen d'exécution réellement observé et la deuxième colonne (« h. p. ») indique la moyenne des valeurs absolues des erreurs de prédiction (en %) des 80, 85, 90, 95 et 99èmes pourcentiles. Le tableau 5.5 synthétise ces résultats par application.

Pour les applications *ANTS*, le modèle M2 est celui fournissant les meilleures prédictions (erreur la plus faible) ; M3 est également satisfaisant. Pour les applications *Magician*, le modèle M3 fourni des prédictions entrant dans la fourchette de précision souhaitée : ± 10 % pour la moyenne et ± 20 % pour les hauts pourcentiles.

M3 étant suffisant dans les cas envisagés (qui représentent une palette de situations assez large si on considère les limitations des solutions proposées au chapitre 2), nous n'avons pas immédiatement cherché à l'améliorer ou chercher à lui trouver une alternative et nous l'avons utilisé concrètement dans deux expériences de gestion de réseau. La dernière partie de ce document présente les applications et les perspectives de notre solution.

Troisième partie

Applications et évolutions

Chapitre 6

Intégration expérimentale pour la gestion

Sommaire

6.1	**Introduction**	**91**
6.2	**Modifications apportées à *Magician***	**91**
6.3	**Expérience de contrôle d'exécution**	**92**
6.4	**Expérience de prédiction de charge**	**93**
6.5	**Conclusion**	**97**

6.1 Introduction

Pour illustrer l'intérêt de notre solution, nous avons utilisé le modèle M3 présenté précédemment dans deux expériences concrètes de gestion : pour améliorer la détection de paquets malveillants (section 6.3) et pour améliorer la prédiction de la charge globale d'un réseau (section 6.4). Au préalable nous avons dû modifier légèrement *Magician* afin que cet EE (en attendant que le nœud soit doté d'une véritable entité de contrôle) puisse monitorer et contrôler l'exécution des paquets (section 6.2) : l'EE peut savoir depuis combien de temps un paquet s'exécute et peut décider de le détruire.

L'application utilisée dans les expériences est une application de transfert audio développée à l'université du Kansas qui découpe un fichier audio présent à la source en paquets (2 278 dans notre cas) et les transporte jusqu'au nœud destination où les paquets sont re-assemblés pour pouvoir être joués. La topologie utilisée est représentée par la figure 6.1.

6.2 Modifications apportées à *Magician*

Pour les expériences décrites dans la suite, nous avons dû modifier *Magician* afin que les paquets dépassant un temps d'exécution pré-défini soient détruits. Nous avons introduit une nouvelle classe *Predictor* qui fournit la limite à ne pas dépasser. Pour l'instant,

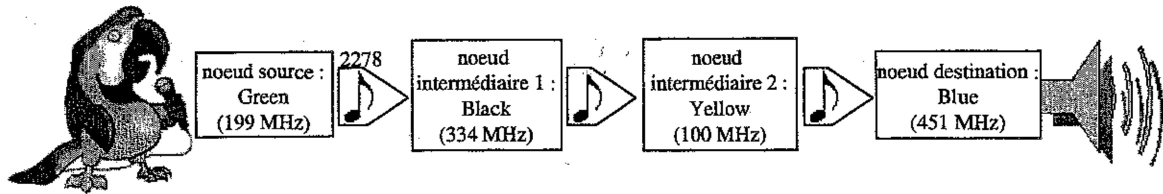


FIG. 6.1 – Configuration utilisée lors des expériences

cette classe se contente de lire la valeur de la prédiction du 99ᵉ pourcentile préalablement calculée à partir du modèle M3 et stockée dans un fichier ; mais à terme, elle devrait recevoir le modèle d'une application active et calculer elle-même la prédiction ou l'obtenir de l'EE de gestion et de supervision du nœud. Nous avons modifié la classe *SPExecEnv* de façon à interrompre régulièrement (ou tout du moins aussi régulièrement qu'il était possible de le faire en Java sans réécrire complètement la plate-forme) l'exécution d'un paquet actif et vérifier que son temps d'exécution reste sous la limite fixée. Pour connaître le temps d'exécution courant du paquet, nous avons ajouté une méthode *getExecTime* dans *KUSmartPacketV2* qui invoque le nouvel appel système *getPidCc* dont le code est présenté dans l'annexe C. Si le temps d'exécution est supérieur à la valeur prévue, le paquet est tué et la variable de la MIB indiquant le nombre de paquets tués est incrémentée. Si le paquet termine son exécution dans les temps, c'est la variable indiquant le nombre de paquets correctement terminés qui est incrémentée. Dans les 2 cas, le temps moyen d'exécution des paquets est mis à jour.

6.3 Expérience de contrôle d'exécution

Pour les besoins de cette démonstration, tous les 5 paquets « normaux », la source envoie également un paquet qui prétend être un paquet appartenant à l'application audio mais qui contient en fait un programme malintentionné qui boucle et consomme du temps CPU pour rien (455 mauvais paquets). Un bon gestionnaire du réseau doit évidemment stopper ces mauvais paquets au plus vite en perturbant le moins possible les bons paquets. Nous avons modifié *Magician* de façon à ce qu'il détruise les paquets s'exécutant au delà d'une certaine limite. Comme je l'ai expliqué au chapitre 2, la difficulté est de fixer la valeur de cette limite.

Dans un premier temps, nous mesurons le temps d'exécution des bons paquets sur le nœud source. Ces mesures indiquent que le 99ᵉ pourcentile du temps d'exécution des paquets se situe à 1 649 710 cycles d'horloge. C'est cette valeur qui sera utilisée comme seuil ou borne sur les autres nœuds dans une première expérience (placée dans le fichier consulté par la classe *Predictor*).

Dans une seconde expérience, le modèle M3 de l'application préalablement construit est « envoyé » aux nœuds. À partir de ce modèle, chacun des nœuds intermédiaires et le nœud destination prévoient leur valeur du 99ᵉ pourcentile du temps d'exécution. Suite à cette génération de prédiction, le premier nœud intermédiaire fixe la limite d'exécution à 470 789 cycles d'horloge et le nœud suivant à 2 399 157 cycles d'horloge.

Dans les deux expériences, les mauvais paquets sont bien arrêtés comme prévu dès

qu'ils atteignent la borne limite sur le premier nœud intermédiaire. Mais comme la valeur de cette limite est plus basse dans la deuxième expérience, on gaspille moins de temps CPU à traiter les mauvais paquets : (1 649 710-470 789)*455 = 536 409 055 cycles d'horloge (= 1606 ms sur le premier nœud intermédiaire) de temps CPU économisé en tout. Le temps moyen d'exécution par paquet (bons et mauvais paquets confondus) dans la deuxième expérience est donc inférieur de 1606/(2278+455) = 0.59 ms au temps moyen de la première expérience. Les mesures effectuées confirment ce résultat théorique puisqu'on mesure une économie moyenne de presque 0.63 ms.

De plus, sur le second nœud intermédiaire, aux performances bien inférieures à celles du nœud source ou du premier nœud intermédiaire, les « bons » paquets n'ont généralement pas le temps de s'exécuter dans les 16.5 ms limite de la première expérience (sur *Yellow*, 1 649 710 cycles d'horloge = 16.5 ms). 2186 bons paquets (soit 96 % !) sont ainsi tués, rendant complètement impossible la reconnaissance des sons restitués à la destination. En revanche, en utilisant le modèle M3 pour adapter la limite au nœud, seulement 19 paquets (soit moins de 1 %) sont tués.

6.4 Expérience de prédiction de charge

Nous avons également combiné nos travaux à ceux réalisés par General Electric sur l'AVNMP (*Active Virtual Network Management Prediction*, [BK01]), outil de prévision de la charge d'un réseau. Dans un premier temps, je vais rapidement citer les limitations des systèmes actuels de gestion puis j'expliquerai comment AVNMP aspire à les dépasser. Puis nous proposerons une amélioration de ce système grâce à une intégration des modèles exposés dans la partie II.

6.4.1 Limites des systèmes de gestion actuels

Dans les systèmes actuels de gestion de réseaux, une station centrale interroge régulièrement ou ponctuellement des agents situés sur les différents éléments du réseau, puis analyse et corrèle les données récoltées avant de prendre éventuellement des mesures correctrices. Dans cette catégorie, on retrouve les outils spécifiques aux équipementiers pour les réseaux homogènes comme par exemple *OpenView* pour *HP* ou *Transcend* pour *3com*, le réseau de gestion des télécommunications (plus connu comme *TMN*), et les approches MIB/SNMP de l'IETF (RFC 1901 et 1213) ou MIT/CMIP de l'OSI (norme ISO 9596).

Ce principe de fonctionnement s'adapte mal à l'évolution qui a eu lieu dans les réseaux.

Tout d'abord, lorsque SNMP a été mis en place en 1989, peu de ressources étaient disponibles sur les équipements à gérer, conduisant à l'écriture d'agents très simples et laissant toute l'intelligence de gestion à la charge du gestionnaire central. Mais cette approche exploite les ressources qui ont finalement peu évolué depuis et restent « critiques » (bande passante en particulier) et ne tire pas avantage de la mémoire et de la capacité de calcul de plus en plus disponibles aux nœuds : les agents ne résument pas

l'information à transmettre et la station centrale fait parfois des interrogations intensives ce qui gaspille de la bande passante.

D'autre part, cette solution passe mal à l'échelle : quand le nombre d'équipements à gérer croît, on constate souvent des encombrements autour de la station de gestion lorsque les informations de gestion sont transportées par le réseau géré (*in-band management*, cas fréquent). Pire : la station de gestion intensifie parfois ses requêtes en cas de congestion pour essayer de cerner le problème, conduisant dans le cas présent à un encombrement encore plus important.

De plus, si la station centrale de gestion tombe en panne, ou si elle n'est plus accessible (à cause de problème réseau par exemple !), la gestion n'est plus assurée, ce qui peut avoir de graves conséquences.

Enfin, les services offerts par les agents sont figés, définis *a priori* et accessibles seulement au travers d'interfaces fixées. Par conséquent, toutes les évolutions demandent beaucoup de temps de re-analyse, conception et implantation.

Toutes ces limites sont causées par la centralisation de la solution. Un autre reproche que l'on fait à cette approche est qu'elle est réactive : les gestionnaires ne sont souvent capables de détecter et répondre à une attaque ou une panne que longtemps après qu'elle aie commencé à paralyser le système.

6.4.2 Présentation d'AVNMP

Les réseaux actifs constituent des candidats idéaux pour faire évoluer les solutions de gestion existantes et résoudre les problèmes évoqués car ils permettent non seulement de décentraliser les traitements mais aussi des les modifier en cours de gestion si nécessaire. C'est en partie pour faire la démonstration de ces bénéfices que *General Electric* a développé son système AVNMP (*Active Virtual Network Management Prediction*, [Bus98]). Un autre objectif principal était de permettre une gestion pro-active — et non plus seulement réactive — en analysant les performances courantes du réseau et en prévoyant les performances futures sur la base d'évènements futurs probables et des réactions du réseau face à ces événements. Comme nous allons le détailler, leur travail utilise bien sûr les réseaux actifs (mais il vise la gestion des réseaux aussi bien classiques qu'actifs). Le premier domaine auquel AVNMP s'est intéressé, et qui est celui que nous avons amélioré, est la prédiction de la charge du réseau.

Les éléments et le fonctionnement d'AVNMP sont résumés sur la figure 6.2. Les éléments rectangulaires de la partie basse de la figure représentent un réseau actif classique, dans lequel des paquets actifs circulent entre les nœuds et sont exécutés comme applications actives dans un environnement d'exécution présent aux nœuds. Cet environnement d'exécution est *Magician*, et il est configuré de façon à placer dans une base de données de gestion située sur chacun des nœuds un certain nombre d'informations, comme par exemple le nombre de paquets actifs reçus et envoyés.

Les éléments ovales de la partie supérieure de la figure 6.2 représentent les composants d'AVNMP. Ces composants sont en fait implantés comme une application active particulière de *Magician*. Sur le nœud source de l'application active, un *Driving Process* observe l'envoi des paquets actifs réalisés par l'AA jusqu'à la date t. Puis, en extrapolant ces observations (*curve fitting*), il émet une prédiction sur le nombre de paquets qui

Nœud	Black	Yellow	Blue
nombre de *rollbacks*	92	42	12
avance dans le futur (secondes)	-101	-20	54

TAB. 6.1 – Performances d'AVNMP avec un modèle non adaptatif

seront envoyés au nœud suivant à une certaine date future $t + a$. Il envoie alors cette information de prévision (nombre de paquets prévus et date future) au *Logical Process* situé sur le nœud suivant. Celui-ci utilise cette information, plus éventuellement des prédictions reçues d'autres nœuds, pour mettre à jour dans la base de données de gestion de son nœud le nombre de paquets dont il prévoit la réception pour la date $t + a$. Puis, le *Logical Process* utilise son extrapolateur pour prédire combien de paquets quitteront le nœud à une date future $t + b$ où $b > a$ et il envoie cette information au nœud suivant. Puis il attend une nouvelle prédiction en provenance du nœud précédent pour essayer de prédire la charge encore plus loin dans le futur.

Lorsque la date $t + a$ est effectivement atteinte, le *Logical Process* du premier nœud intermédiaire compare le nombre de paquets réellement reçus (information placée dans la MIB par l'EE) avec le nombre de paquets dont il avait prévu la réception pour cette date (information placée dans la MIB par lui-même à la date $t + \delta_t$). Si la différence entre les deux valeurs est inférieure au seuil de tolérance fixé, les prédictions continuent plus en avant dans le futur. En revanche, si l'écart constaté est trop grand, le *Logical Process* annule toutes les prédictions faites depuis le message reçu du *Driving Process* qui avait conduit à la prédiction erronée de $t+a$. Ce mécanisme est appelé *rollback* dans la suite. Un mécanisme supplémentaire permet de prévenir les nœuds en aval pour qu'ils ne tiennent pas compte des prédictions reçues.

AVNMP permet ainsi de prévoir le nombre de paquets actifs qui seront reçus et envoyés et donne donc une estimation de la bande-passante qui sera nécessaire pour le bon fonctionnement de l'application. Pour prévoir également la charge au niveau des nœuds, les *Logical Process* supposent que tous les paquets vont mettre le même temps à s'exécuter : le temps d'exécution d'un paquet actif sur le nœud source. Nous avons déjà vu que cette technique ne fournit pas de prédictions satisfaisantes car elle ne tient pas compte de la multitude des sources de variabilité influençant le temps d'exécution d'un paquet actif.

Pour le vérifier, nous avons modifié AVNMP de façon à ce qu'il ne fasse des *rollbacks* que lorsqu'il constate une différence entre le temps moyen d'exécution prévu et le temps moyen d'exécution réel (moyenne pour 20 paquets actifs). Le seuil de tolérance a été fixé à plus ou moins 10 % de la valeur réelle. Le tableau 6.1 indique les résultats obtenus lors du déploiement et de l'exécution de l'application active de transfert audio (sans paquet malintentionné) dans le réseau illustré par la figure 6.1. On constate tout d'abord que le nombre de *rollbacks* est important, en particulier sur les 2 premiers nœuds intermédiaires. Obligé de revenir sans cesse en arrière, AVNMP n'a pas réussi à avancer dans le futur et n'a même pas réussi à « tenir la cadence » du présent comme l'indiquent les valeurs négatives de « l'avance dans le futur » relevées sur les nœuds *Black* et *Yellow*.

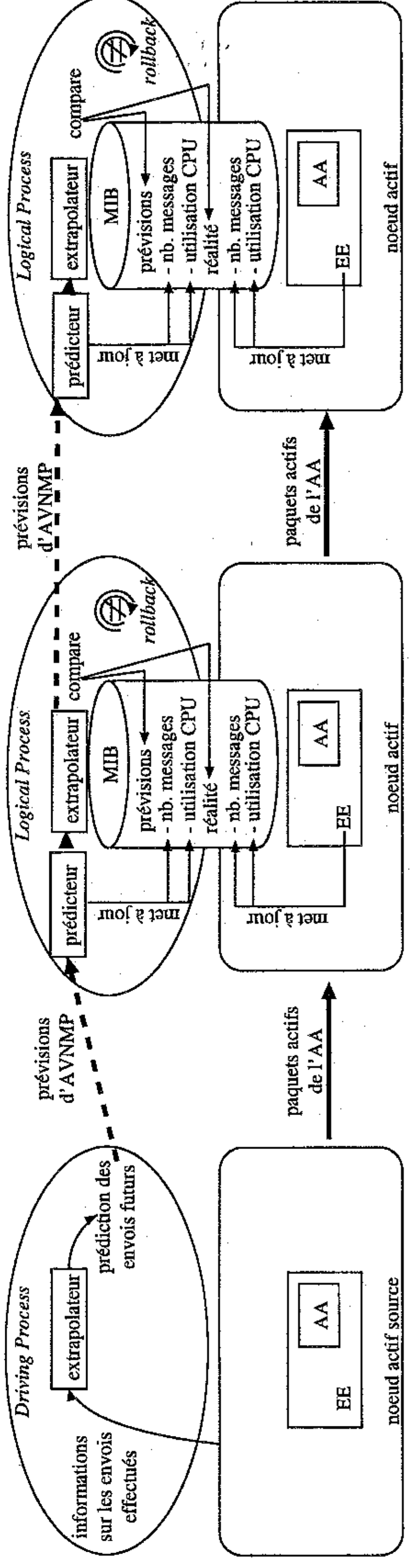


FIG. 6.2 – Fonctionnement d'AVNMP

S.C.D. - U.H.P. NANCY 1
BIBLIOTHEQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

Nœud	Black	Yellow	Blue
nombre de *rollbacks*	28	0	0
avance dans le futur (secondes)	432	102	313

TAB. 6.2 – Performances d'AVNMP avec intégration du modèle M3

6.4.3 Amélioration d'AVNMP

Pour améliorer la prédiction de la charge au niveau des nœuds nous avons remplacé dans les *Logical Process* le module qui utilise directement le temps d'exécution constaté sur le nœud source par un module utilisant le modèle M3 de l'AA pour fournir la prédiction de charge. Le tableau 6.2 donne les nouveaux résultats. Les prédictions étant meilleures, moins de *rollbacks* ont été nécessaires et AVNMP a donc pu s'envoler un peu plus vers le futur.

Les prévisions d'AVNMP auraient également pu être améliorées par observation du temps de traitement des paquets à chaque nœud et adaptation en conséquence. Cependant, intégrer le modèle/prédicteur présente plusieurs avantages sur cette méthode : (1) des prédictions peuvent être faites dès le premier paquet actif reçu, sans attendre de voir une tendance se dégager et (2) le travail de prédiction est moins lourd, il est réalisé à la réception du premier paquet et il est inutile de faire de constantes adaptations.

6.5 Conclusion

Ces deux expériences illustrent comment nos modèles peuvent servir de « briques de base » pour l'établissement de politiques plus sophistiquées. D'autres applications sont bien sûr envisageables : les prédictions pourraient être utilisées comme un critère supplémentaire dans des algorithmes de routage pour respecter une certaine qualité de service, des mécanismes d'admission pourraient être développés aux nœuds, des simulations du coût d'utilisation d'une application par rapport à une autre pourraient être faites...

Chapitre 7

Propositions pour repousser les limitations de la solution proposée

Sommaire

7.1 **Inconvénients d'un modèle empirique** 99
7.2 **Travaux futurs** 100

La solution présentée dans les chapitres 4 et 5 souffre d'une limitation importante. Dans ce chapitre, je détaille le problème et propose des solutions pour améliorer encore les modèles de façon à repousser cette limitation.

7.1 Inconvénients d'un modèle empirique

Contrairement à ce qui se passe dans le projet Apple (mentionné page 42), l'obtention des éléments permettant la construction du modèle ne requiert aucune compétence ou connaissance particulière de la part du développeur : il lui suffit d'utiliser les versions modifiées du noyau et des environnements d'exécution présentés au chapitre 3 pour qu'une trace soit automatiquement générée lorsqu'il effectue ses derniers tests. Cependant il est souhaitable de s'affranchir plus encore de la participation du développeur d'application active pour la construction du modèle.

De plus, la construction des modèles à partir d'une phase « tracée » représente une sérieuse limite de la solution proposée. En effet, si la réalité diffère des situations prévues dans la phase préliminaire de traçage, les prédictions ne seront pas satisfaisantes. Par exemple, dans l'application *Multicast* d'*ANTS* un nœud peut recevoir 2 types de paquets : des paquets d'inscription de la part des nœuds qui souhaitent informer le serveur de leur désir de faire partie de la session de multicast et des paquets de données envoyés par le serveur de multicast. Si pendant la phase de construction du modèle seuls quelques nœuds étaient intéressés par le multicast et que le serveur était très bavard, la probabilité du scénario associé au traitement d'un paquet de données sera beaucoup plus grande que celle du scénario associé au traitement d'un paquet d'inscription. Mais la réalité pourra être très différente, avec une multitude de nœuds désirant s'inscrire par exemple. Les prédictions rendues seront donc erronées. Dans PLATO (outils de placement de tâches

Rôle	Expéditeur	Intermédiaire	Récepteur	Multi-rôle
Taille du modèle (octets)	8831	5638	5659	16282
(Erreurs en %)				
Moyenne	-0,67	-2,09	-0,66	-2,37
80e pourcentile	-2,28	-6,48	-6,37	11,94
85e pourcentile	-4,23	-8,42	-6,10	11,05
90e pourcentile	-4,52	3,68	2,38	10,78
95e pourcentile	-1,32	0,05	-0,19	13,76
99e pourcentile	-11,85	-10,89	-13,44	18,45
Hauts pourcentiles	4,84	5,91	5,70	13,20

TAB. 7.1 – Problème d'un mélange de scénarios tracés différent du mélange réel

pour la programmation par échange de messages) on constate le même problème lié au rapport entre un « jeu de données de pré-exécution » et le jeu de données réel et on laisse à l'utilisateur la charge de décider si les données sur les exécutions récoltées par le biais du traçage sont valides ou non [San98].

Pour illustration, la table 7.1 compare les erreurs de prédictions sur *Yellow* du modèle M3 du *Ping* de *Magician* construit à partir d'une trace sur le nœud *Green* n'assumant qu'un rôle unique (expéditeur, intermédiaire ou récepteur du ping, chaque traitement prenant un temps assez différent) ou construit à partir d'une trace sur le nœud assumant tour à tour les 3 rôles (colonne « Multi-rôle »). On constate que les erreurs sont plus importantes lorsque l'on mélange les différents rôles.

De plus, certains comportements de l'application (comme le nombre de passages dans une boucle ou le choix fait à un "si-alors-sinon") peuvent dépendre de conditions locales au nœud impossibles à prévoir lors de la construction du modèle. Par exemple, dans l'application de *Multicast* d'*ANTS*, le traitement d'un paquet de données à un nœud intermédiaire amène à exécuter une boucle B autant de fois qu'il y a d'inscrits dans la liste de multicast du nœud. Si 3 nœuds seulement étaient intéressés dans la session lors de la génération de la trace, les scénarios envisagés seront B, BB et BBB mais si 5 nœuds s'inscrivent en réalité, les scénarios BBBB et BBBBB apparaissent.

Ces exemples indiquent les limites d'un modèle basé uniquement sur une trace d'exécution « arbitraire ».

7.2 Travaux futurs

7.2.1 Modèle à apprentissage permanent

Pour résoudre le problème que je viens de présenter, on peut imaginer que le modèle d'une application déposé à un nœud continue à évoluer : la probabilité des chemins que l'application prend le plus sera augmentée alors que celles des chemins beaucoup empruntés lors de la trace initiale mais jamais utilisés sur ce nœud précis diminuera. De même, les distributions de temps (ou moyennes) présentes dans le modèle pourront être ajustées. Pour que ce processus ne soit trop lourd, on peut imaginer faire la mise à jour

du modèle en ne mesurant qu'1 paquet sur 100 par exemple. Cette idée est valable dans la mesure où un grand nombre de paquets de l'application concernée passent sur le nœud. On peut également imaginer qu'un nœud qui « découvre » un chemin non présent dans le modèle en fasse part à ses voisins (en utilisant du code actif par exemple). Un nœud fédérateur pourrait aussi régulièrement regrouper les différentes versions du modèle en circulation.

7.2.2 Construction du modèle par analyse statique

Une des raisons pour laquelle l'approche « boîtes noires » présentée a été choisie était qu'au début de ce travail beaucoup de plates-formes émergeaient et comme il était difficile de prévoir lesquelles allaient éventuellement perdurer, et combien de systèmes allaient se partager la scène, nous voulions nous affranchir au maximum de leurs spécificités et nous concentrer sur le seul élément promettant de se standardiser, à savoir l'interface entre les environnements d'exécution et le système d'exploitation actif.

Cependant, si on se restreint à une plate-forme donnée, il devrait être possible de construire une première mouture du modèle d'une application active par analyse statique de son code. Cela est plus particulièrement envisageable si on se contente de déterminer les chemins acycliques dans le code d'une application. (Mais les travaux de Larus et Ball ([BL00]) suggèrent que cela nécessite d'instrumenter le code de l'application.)

Ce modèle « brut » pourrait ensuite être envoyé sur le réseau où dans un premier temps il ne serait pas utilisé mais simplement « renseigné » par les nœuds qui indiqueraient la probabilité des différents chemins tels qu'ils les observeraient (approche similaire aux « modèles à apprentissage permanents précédents »). Puis, lorsque le modèle aurait atteint une certaine stabilité, il serait re-étiqueté de « brut » à « utilisable » et les nœuds pourraient l'utiliser pour leur gestion.

Alternativement, pour les applications au comportement itératif dont le nombre d'itérations est déterminé par une condition locale au nœud (comme dans l'exemple précédent du *Multicast* d'*ANTS*), il faudrait définir une façon standard de décrire ces conditions locales et à l'arrivée du modèle d'application (ne comportant que la liste des chemins acycliques et l'expression déterminant le nombre d'itérations en fonction des conditions du nœuds), un nœud, qui connaît ses conditions locales, compléterait le modèle avant de l'utiliser.

7.2.3 Complicité du développeur

Une autre solution serait de modifier partiellement le cahier des charges en augmentant la contribution du programmeur d'application.

Imaginons qu'un développeur conçoive une application susceptible d'utiliser beaucoup de ressources mais qu'il prévoit aussi une solution de repli qui sera exécutée si les ressources sont insuffisantes par exemple. Le comportement qui va être choisi à un nœud est totalement imprévisible, on ne peut pas dire à priori dans 20 % des cas les nœuds n'auront pas assez de ressources et le deuxième scénario sera exécuté etc. Donc notre solution risque de ne pas donner de très bons résultats car le développeur ne peut pas fournir une trace d'exécution qu'il espère honnêtement représenter la réalité.

Une solution est alors de demander au développeur de générer un modèle différent pour chacun des profils d'utilisation. Ça ne devrait pas être trop compliqué pour lui puisqu'il a explicitement programmé les alternatives. Chaque modèle serait associé à un profil d'utilisation comme par exemple un ratio charge du nœud sur mémoire disponible. À l'arrivée sur un nœud, la version à utiliser serait sélectionnée en fonction des conditions locales.

Bien sûr, cette solution demande d'implanter quelques mécanismes supplémentaires mais les réseaux actifs sont justement très bien adaptés à ce genre de choses !

7.2.4 Multi-modèle, pluri-prédictions à évaluation continue

Il est probable que la suite des travaux mène non pas à un modèle unique ou à une technique unique pour en dériver des prédictions mais à plusieurs modèles et plusieurs façons de les utiliser. Cependant, un modèle/prédicteur sera certainement plus efficace dans certaines situations ou pour une certaine classe d'applications que pour une autre. Par analogie avec ce qui est fait par le « service météo du réseau » (*Network Weather Service*, [WSH98]), on peut imaginer l'évaluation continue suivante : lorsqu'un paquet arrive, on prédit son temps exécution avec tous les modèles/prédicteurs possibles mais on ne fournit à l'ordonnanceur que le résultat du meilleur modèle/prédicteur. Lorsque la vraie exécution est terminée, on compare le temps exécution réel avec les différentes prédictions et on attribue une note à chacun des modèles/prédicteurs en fonction de l'erreur de la prédiction fournie. Le prédicteur avec la meilleure note sera utilisé pour le prochain paquet. Bien sûr, on peut ne réévaluer les modèles que tous les 100 paquets par exemple, ou choisir le meilleur basé sur une moyenne sur les 10 derniers échantillonnages au lieu du dernier uniquement...

7.2.5 Modèle compacté

Nous avons évoqué au chapitre 5 que les distributions de temps observées étaient trop irrégulières pour que l'on puisse les identifier aisément à des distributions statistiques classiques et cela nous a conduit à utiliser une représentation des distributions de temps sous forme d'histogrammes. Cependant, d'une part cela rend complexe la génération de prédictions de façon analytique et nous avons dû utiliser un simulateur. Et d'autre part, malgré une petite optimisation consistant à ne pas coder les barres d'histogramme « vides », cela produit des modèles de taille non négligeable (on cherche à minimiser le surcoût engendré par la gestion).

À présent que nous avons démontré que notre approche peut conduire à des prédictions intéressantes, il serait profitable de se pencher à nouveau sur le problème de la représentation la plus efficace possible des distributions de temps. Par exemple, la distribution du temps d'exécution de certaines transitions entre 2 appels système invoqués par une application *Magician* semble pouvoir être identifiée à la convolution d'une distribution du χ^2 et d'une gaussienne.

De nombreux outils d'analyse statistique existent (comme par exemple Dataplot[42]),

[42]http ://www.itl.nist.gov/div898/software/dataplot/

ils devraient faciliter cette tâche.

Bilan

Résumé des contributions

Pour dynamiser les réseaux de transport de données à commutation de paquets, les réseaux actifs offrent aux applications la possibilité de déployer leurs propres piles de protocoles au cœur du réseau. Sans moyen pour un nœud de prévoir les besoins à l'exécution des paquets actifs, cette flexibilité rime avec inefficacité de gestion et faible sûreté de fonctionnement du réseau. Dans ce contexte, l'objectif de cette thèse était la recherche de mécanismes permettant de contribuer à l'expression des besoins en calcul d'un programme au niveau des nœuds actifs.

En l'absence de dispositif adapté sur les plates-formes expérimentales disponibles, j'ai tout d'abord développé un outil d'observation permettant d'enregistrer avec finesse (l'unité de temps est le cycle d'horloge) le temps qu'une application active passe dans les différentes parties du système sur lequel elle s'exécute. Cet outil a tout d'abord été utile pour nous donner une idée des échanges ayant lieu entre l'environnement d'exécution et le système d'exploitation. Ensuite, il m'a permis de collecter les données qui m'ont servi de référence lors de l'évaluation des différents modèles. Enfin, il a été utilisé dans nos expériences de contrôle et de prédiction pour fournir en temps réel la consommation effective des paquets.

Une autre contribution de cette thèse consiste en la proposition de modèles pour représenter d'une part les performances relatives des différents nœuds d'un réseau actif hétérogène et d'autre part les besoins en temps CPU d'un paquet actif. Ces modèles se combinent pour permettre d'estimer le temps d'exécution d'un paquet actif sur n'importe quel nœud.

Les tests de validation ayant empiriquement montré que le temps moyen d'exécution pouvait être prédit avec moins de 10 % d'erreur et que les hauts pourcentiles (approchant les temps maximums d'exécution) étaient prédits avec moins de 20 % d'erreur, j'ai ensuite utilisé les modèles dans deux expériences concrètes qui ont montré qu'ils permettaient une meilleure gestion des ressources.

S.C.D. - U.H.P. NANCY 1
BIBLIOTHÈQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

Perspectives

Malgré les contributions de cette thèse, le problème de la gestion efficace des ressources de calcul aux nœuds actifs persistera au moins tant que les différents acteurs du domaine ne seront pas accordés sur une façon d'intégrer les mécanismes adéquats dans l'architecture commune.

En effet, comme notre travail l'a fait ressortir, il est tout d'abord nécessaire que les plates-formes aient une architecture qui permette de distinguer clairement les cycles d'horloge utilisés pour le traitement direct d'un paquet actif des cycles d'horloge à imputer au fonctionnement global de l'environnement d'exécution. D'autre part, l'outil d'observation que nous avons dû développer devrait être une partie intégrante de l'environnement d'exécution de supervision envisagé dans les spécifications servant de guide à de nombreux projets de recherche [Cal99]. Enfin, lorsque ces exigences seront satisfaites, il faudra développer un protocole (éventuellement actif!) pour permettre le déploiement automatique des modèles d'applications, non lié à une plate-forme particulière.

Des travaux de persuasion et concertation sont donc encore nécessaires avant que l'on puisse réellement imaginer alléger les processus de standardisation des réseaux (alors que c'est l'ambition des réseaux actifs, c'est assez ironique...).

Bilan personnel

Pour conclure ce manuscrit, j'aimerais ajouter quelques mots sur l'apport que constitue cette thèse sur un plan plus personnel. En effet, durant ces 40 mois j'ai bien sûr acquis de nombreuses compétences au niveau des réseaux actifs et autres concepts ou techniques informatiques évoqués (ou non) dans ce document, mais la thèse m'a également apporté d'autres expériences qui seront je pense utiles quelle que soit la voie que je suive dans le futur.

S'il était motivant de me voir confier dès la sortie de l'ESIAL[43] la responsabilité d'un projet à moyen terme (3 ans), me laissant entièrement le choix des orientations à donner au travail, de sa gestion et sa présentation, il faut bien avouer qu'il y a eu des périodes où cette liberté a été un peu lourde à porter. Cette expérience m'a donc permis de tester mon autonomie et de repousser ses limites.

Je pense qu'il aurait été beaucoup plus difficile en allant dans l'industrie avec mon diplôme d'ingénieur en 1998 de travailler sur ce sujet de pointe, les équipementiers ou fournisseurs de services ne prenant souvent le risque d'explorer de nouvelles solutions que lorsqu'elles ont déjà été un peu « débroussaillées » ailleurs. Je crois que j'ai pris goût à prendre part à l'innovation.

Sur le plan de la méthodologie, la thèse m'a également formée à la veille technologique, j'estime que je sais à présent mieux chercher, et mieux apprendre qu'il y a 3 ans. Être capable de partager ensuite cette connaissance est tout aussi important et diverses activités liées à la thèse (rédaction d'articles, exposés lors de conférences, et enseignement) m'ont permis d'améliorer mon aptitude à structurer, illustrer, expliquer et présenter mes informations.

J'ai eu la chance de profiter de nombreuses opportunités de voyager et séjourner à l'étranger dans le cadre de mon travail. Cela m'a permis d'observer (et parfois adopter) de nouvelles habitudes de vie et de travail, d'améliorer mon anglais, de développer un petit réseau de connaissances avec des chercheurs de différents laboratoires et enfin de découvrir de magnifiques paysages et de belles histoires.

[43] *École Supérieure d'Informatique et Applications de Lorraine*, dont j'ai reçu le diplôme en 1998.

Références

S.C.D. - U.H.P NANCY 1
BIBLIOTHEQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

Bibliographie

[ABG+97] D. Scott Alexander, Bob Braden, Carl A. Gunter, Alden W. Jackson, Angelos D. Keromytis, Gary J. Minden, and David Wetherall. Active network encapsulation protocol (anep). Request for Comments : DRAFT, Category : Experimental, July 1997.

[AMK+01] D. Scott Alexander, Paul B. Menage, Angelos D. Keromytis, William A. Arbaugh, Kostas G Anagnostakis, and Jonathan M. Smith. The price of safety in an active network. *Computers and Networks, Special Issue on Programmable Switches and Routers*, March 2001. http ://www.chiark.greenend.org.uk/~paulm/papers/jcn.ps.

[BBD+98] Michael Beck, Harald Bohme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, and Dirk Verworner. *Linux kernel internals.* Addison-Wesley, 1998.

[BBR00] Steven Berson, Bob Braden, and Livio Ricciulli. Introduction to the abone. http ://www.isi.edu/abone/DOCUMENTS/ABoneIntro.ps, June 2000.

[BCF+99] Bob Braden, Alberto Cerpa, Ted Faber, Bob Lindell, Graham Phillips, and Jeff Kann. The asp ee : An active execution environment for network control protocols. Technical report, ISI, December 1999. http ://fmg-www.cs.ucla.edu/classes/239_3.winter00/papers/ASP_EE.ps.

[BCF+01] B. Braden, A. Cerpa, T. Faber, B. Lindell, G. Phillips, J. Kann, and V. Shenoy. *Introduction to the ASP Execution Environment (Release 1.5).* USC ISI, November 2001. http ://www.isi.edu/active-signal/ARP/DOCUMENTS/ASP_EE.ps.

[BCZ97] Samrat Bhattacharjee, Kenneth L. Calvert, and Ellen W. Zegura. An architecture for active networking. In *Proceedings of High Performance Networking (HPN) '97)*, April 1997. http ://www.cc.gatech.edu/projects/canes/papers/hpn97.ps.gz.

[Ber99] Guy Bernard. Technologie du code mobile : état de l'art et perspectives. In Hermes, editor, *CFIP'99*, pages 367–381, Nancy, France, April 1999.

[BK01] Stephen F. Bush and Amit B. Kulkarni. *Active Networks and Active Network Management.* Kluwer Academic/Plenum Publishers, March 2001. ISBN 0-306-46560-4.

[BL00] Thomas Ball and James R. Larus. Using paths to measure, explain, and enhance program behavior. *IEEE Computer*, pages 57–65, July 2000. http ://www.research.microsoft.com/~tball or ~larus.

[Bus98] Stephen F. Bush. Active virtual network management protocol. DARPA PI Meeting, October 1998.

[BWF+96] Fran Berman, Rich Wolski, Silvia Figueira, Jennifer Schopf, and Gary Shao. Application-level scheduling on distributed heterogeneous networks. In *Supercomputing '96*, 1996. http ://www-cse.ucsd.edu/groups/hpcl/apples/sup96/main.ps.

[Cal99] Ken Calvert. Architectural framework for active networks, version 1.0. http ://www.cc.gatech.edu/projects/canes/papers/arch-1-0.ps.gz, July 1999.

[CBZS98] Kenneth Calvert, Samrat Bhattacharjee, Ellen Zegura, and J. Sterbenz. Directions in active networks. *IEEE Communications Magazine*, 1998.

[CDM97] Rémy Card, Eric Dumas, and Franck Mével. *Programmation Linux 2.0, API système et fonctionnement du noyau.* Eyrolles, 1997.

[Chr96] Isabelle Chrisment. *Etude et développement d'applications distributées dans l'architecture ALF.* PhD thesis, Université de Nice-Sophia Antipolis, June 1996. http ://www.loria.fr/~ichris/thesis/these.ps.gz.

[CMK+99] Andrew T. Campbell, Herman G. De Meer, Michael E. Kounavis, Kazuho Miki, John B. Vicente, and Daniel Villela. A survey of programmable networks. *ACM SIGCOMM Computer Communication Review*, 29(2) :7–24, April 1999. http ://comet.columbia.edu/~campbell/papers/pnccr.pdf.

[CT90] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM symposium on Communications architectures & protocols*, pages 200–208, September 1990. http ://www.acm.org/pubs/citations/proceedings/comm/99508/p200-clark/.

[EKO95] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr. The operating system kernel as a secure programmable machine. *Operating Systems Review*, 29(1) :78–82, January 1995. http ://www.pdos.lcs.mit.edu/papers/osr-exo.ps.

[Eng98] Dawson R. Engler. *The Exokernel Operating System Architecture.* PhD thesis, Massachusetts Institute of Technology, October 1998. http ://amsterdam.lcs.mit.edu/exo/these/engler/thesis.ps.gz.

[FCF00] Olivier Festor, Isabelle Chrisment, and Eric Fleury. Les réseaux programmables 1.0. rapport de recherche, INRIA, March 2000. http ://www.inria.fr/rrrt/rr-3913.html.

[GHL+99] Virginie Galtier, Craig Hunt, Stefan Leigh, Kevin L. Mills, Doug Montgomery, Mudumbai Ranganathan, Andrew Rukhin, and Debra Tang. How much cpu time? Technical report TR-ANTDANETS -111999, NIST, September 1999. http ://w3.antd.nist.gov/activenets/Documentation/Documents/darpa_meeting_Sep1999.ps.

[GPr] Gprof. http ://www.gnu.org/manual/gprof-2.9.1/gprof.html.

[HKM+98a] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. Plan : a packet language for active networks. In *International Conference on Functional Programming (ICFP) '98*, 1998. http ://www.cis.upenn.edu/~switchware/papers/icfp98-plan.ps.

[HKM+98b] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. Plan : A packet language for active networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93. ACM, 1998. http ://www.cis.upenn.edu/~switchware/papers/plan.ps.

[HP91] Norman C. Hutchinson and Larry L. Peterson. The x-kernel : An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, January 1991. ftp ://ftp.cs.arizona.edu/xkernel/Papers/architecture.ps.

[HPB+99] John Hartman, Larry Peterson, Andy Bavier, Peter Bigot, Patrick Bridges, Brady Montz, Rob Piltz, Todd Proebsting, and Oliver Spatscheck. Joust : A platform for liquid software. *IEEE Computer*, 1999. http ://www.cs.arizona.edu/scout/Papers/joust.ps.

[HT97] Oliver J. Huber and Laurent Toutain. Mobile agents in active networks. In *ECOOP '97*, June 1997. http ://www.rennes.enst-bretagne.fr/~ader/oliver/Papers/ECOOP-97.ps.gz.

[Int] Intel. Intel pentium processor documentation : Using the rdtsc instruction for performance monitoring. http ://cedar.intel.com/software/idap/media/pdf/rdtscpm1.pdf.

[Jr98] Sixto Ortiz Jr. Active networks : The programmable pipeline. *IEEE Computer Magazine*, 31, August 1998.

[KMH+98] A. B. Kulkarni, G. J. Minden, R. Hill, Y. Wijata, A. Gopinath, S. Sheth, F. Wahhab, H. Pindi, and A. Nagarajan. Implementation of a prototype active network. In *OPENARCH '98*, 1998. http ://www.ittc.ukans.edu/~kulkarn/docs/OpenArch98.ps.gz.

[Kul] Amit Kulkarni. Magician - an active networking toolkit. http ://www.ittc.ukans.edu/~kulkarn/projects/Magician.html.

[Kun93] Roberto Kung. Le réseau intelligent : ce qui est fait actuellement. In *Forums France Télécom Recherche*, number 1. October 1993. http ://www.rd.francetelecom.com/fr/conseil/mento/m1chap3.pdf.

[LGT98] Li-Wei Lehman, Stephen J. Garland, and David L. Tennenhouse. Active reliable multicast. In *IEEE Infocom '98*, March 1998.

[MBZC00] S. Merugu, S. Bhattacharjee, E. Zegura, and K. Calvert. Bowman : A node os for active networks. In *IEEE Infocom 2000*, March 2000. http ://www.cc.gatech.edu/projects/canes/papers/bowman.pdf.

[Men99] Paul Menage. Rcane : A resource controlled framework for active network services. In *Proceedings of the First International*

Working Conference on Active Networks (IWAN) '99), July 1999. http ://www.chiark.greenend.org.uk/~paulm//papers/iwan99.ps.

[MHN01] Jonathan Moore, Michael Hicks, and Scott Nettles. Practical programmable packets. In *IEEE InfoCom 2001*, April 2001. http ://infocom.ucsd.edu/papers/627.ps.

[MLP+01] Sandra Murphy, Edward Lewis, Ralph Puga, Robert Watson, and Richard Yee. Strong security for active networks. In *IEEE Open Architectures and Network Programming Proceedings (OpenArch 2001)*, pages 63–70, April 2001. http ://www.comet.columbia.edu/activities/openarch2001/papers2001/OA_06.PDF.

[MMTH95] G. Di Marzo, M. Muhugusa, C. F. Tschudin, and J. Harms. The messenger paradigm and its impact on distributed systems. In *Intelligent Computer Communications, ICC'95*, Romania, 1995. ftp ://cui.unige.ch/pub/tios/papers/mpid.ps.Z.

[Mun97] Sohail Munir. Active network - a survey. http ://www.cis.ohio-state.edu/~jain/cis788-97/active_nets/index.htm, 1997.

[Mur01] Sandra Murphy. Security architecture for active nets. http ://www.dcs.uky.edu/~calvert/sec-latest.ps, May 2001.

[Nyg96] Erik L. Nygren. Active network technologies. http ://www.pdos.lcs.mit.edu/~nygren/pan/, October 1996.

[Pet01] Larry Peterson. Nodeos interface specification. http ://www.cs.princeton.edu/nsg/papers/nodeos.ps, January 2001.

[Pso99] Konstantinos Psounis. Active networks : Applications, security, safety, and architectures. *IEEE Communications Surveys & Tutorials*, 1999. http ://www.comsoc.org/pubs/surveys/1q99issue/psounis.html and http ://www.stanford.edu/~kpsounis/actnet.html.

[RP94] J. Reynolds and J. Postel. Rfc 1700 assigned numbers. ftp ://ftp.enst.fr/pub/rfc/all/1700.gz, page 63, October 1994.

[San98] Carlos Gamboa Dos Santos. *Apport de l'approche gestion de réseaux pour le placement de tâches dans le modèle de programmation par échange de messages.* PhD thesis, Université Henri Poincaré, October 1998.

[SBSM89] Rafael H. Saavedra-Barrera, Alan Jay Smith, and Eugene Miya. Machine characterization based on an abstract high-level language machine. *IEEE Transactions on Computers*, 38(12) :1659–1679, December 1989.

[Sch01] Stephen Schwab. Amp - enabling active networks via secure exokernel implementations, January 2001. http ://download.nai.com/products/media/pgp/pdf/NAI-Labs-AMP-1-5-01.pdf.

[SCM+99a] Jonathan M. Smith, Kenneth L. Calvert, Sandra L. Murphy, Hilarie K. Orman, and Larry L. Peterson. Activating networks. *IEEE Computer*, April 1999. http ://www.cs.princeton.edu/nsg/papers/an.ps.

[SCM+99b] Jonathan M. Smith, Kenneth L. Calvert, Sandra L. Murphy, Hilarie K. Orman, and Larry L. Peterson. Activating networks : A progress report. *IEEE Computer*, 32(4) :32–41, April 1999.

[SJS+00] Beverly Schwartz, Alden W. Jackson, W. Timothy Strayer, Wendy Zhou, Dennis Rockwell, and Craig Partridge. Smart packets : applying active networks to network management. *ACM Transactions on Computer Systems*, 18(1) :67–88, February 2000. http ://delivery.acm.org.

[SJTS+00] B. Schwartz, A. Jackson, W. Zhou T. Strayer, D. Rockwell, and C. Patridge. Smart packets : applying active networks to network management. *ACM Transactions on Computer Systems*, February 2000.

[Ste01] David Stellmack. Does the choice of network interface cards impact system performance, August 2001. http ://www6.tomshardware.com/network/01q3/010820/nic-02.html.

[Str] strace homepage. http ://www.liacs.nl/~wichert/strace/.

[Swi] The switchware project. http ://www.cis.upenn.edu/~switchware/.

[SZJ+99] Beverly Schwartz, Wendy Zhou, Alden W. Jackson, W. Timothy Strayer, Dennis Rockwell, and Craig Partridge. Smart packets for active networks. In *OPENARCH*, March 1999. ftp ://ftp.bbn.com/pub/AIR/smart.ps.

[Tan97] Andrew Tanenbaum. *Réseaux (3ème édition)*. Prentice Hall, 1997.

[THL01] Patrick Tullmann, Mike Hibler, and Jay Lepreau. Janos : A java-oriented os for active networks. *IEEE Journal on Selected Areas in Communications*, 19(3), March 2001. http ://www.cs.utah.edu/flux/papers/janos-jsac01.ps.gz.

[TSS+97] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1) :80–86, January 1997. http ://www.tns.lcs.mit.edu/publications/ieeecomms97.html.

[TW96] David Tennenhouse and David Wetherhall. Toward an active network architecture. *Computer Communication Review*, 26(2), April 1996. http ://www.tns.lcs.mit.edu/publications/ccr96.html.

[Van97] Van C. Van. A defense against address spoofing using active networks. Master's thesis, MIT, May 1997. http ://www.sds.lcs.mit.edu/publications/postscript/van97.ps.

[vRBM96] Robbert van Renesse, Kenneth P. Birman, and Silvamo Maffeis. Horus : A flexible group communications system. *Communications of the ACM*, April 1996. ftp ://ftp.cs.cornell.edu/pub/horus/doc/cacm.ps.gz.

[Wet97] David Wetherall. Developing network protocols with the ants toolkit. in the ANTS distribution (ants/docs/papers/programming.ps), August 1997.

[Wet99a] David Wetherall. Active network vision and reality : lessons from a capsule-based system. In *17th ACM Symposium on Operating System Principles (SOSP) '99*, December 1999. http ://www.cs.washington.edu/homes/djw/papers/anet-sosp99.pdf.

[Wet99b] David J. Wetherall. *Service Introduction in an Active Network.* PhD thesis, MIT, February 1999. http ://www.cs.washington.edu/research/networking/ants/ants-thesis.ps.gz.

[WGT98] David J. Wetherall, John V. Guttag, and David L. Tennenhouse. Ants : a toolkit for building and dynamically deploying network protocols. In *OpenArch '98*, April 1998. ftp ://ftp.tns.lcs.mit.edu/pub/papers/openarch98.ps.gz.

[WGT99] David Wetherall, John Guttag, and David Tennenhouse. Ants : Network services without the red tape. *IEEE Computer*, 32(4) :42–48, April 1999.

[WJGO98] Ian Wakeman, Alan Jeffrey, Rory Graves, and Tim Owen. Designing a programming language for active networks. Submitted to Hipparch special issue of Network and ISDN Systems, June 1998.

[WLG98] David Wetherall, Ulana Legedza, and John Guttag. Introducting new internet services : Why and how. *IEEE Network. Special issue on active and programmable network*, pages 12–19, May/June 1998. http ://www.sds.lcs.mit.edu/publications/network98.html.

[WSH98] Rich Wolski, Neil Spring, and Jim Hayes. The network weather service : A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 1998. http ://www.cs.ucsd.edu/users/rich/papers/nws-arch.ps.gz.

[WT96] David J. Wetherall and David L. Tennenhouse. The active_ip option. In *7th ACM SIGOPS European Workshop*, September 1996. http ://www.tns.lcs.mit.edu/publications/sigops96ws.html.

[Xru] Xrunhprof. http ://www.java.sun.com/products/jdk/1.2/docs/tooldocs/win32/java.html.

[YdS96] Yechiam Yemini and Sushil da Silva. Towards programmable networks. In *IFIP/IEEE International Workshop on Distributed Systems : Operations and Management (DSOM) '96*, October 1996. http ://www.cs.columbia.edu/~dasilva/pubs/dsom96.ps.

[YL00] Lidia Yamamoto and Guy Leduc. An agent-inspired active network resource trading model applied to congestion control. In *MATA 2000*, pages 151–169, September 2000. http ://www.run.montefiore.ulg.ac.be/~yamamoto/RUN_mata00.ps.gz.

[ZF83a] J. Zander and R. Forchheimer. Softnet : An approach to high level packet communication. In *AMRAD*, 1983.

[ZF83b] Jens Zander and Robert Forchheimer. Softnet, an approach to high level packet communication. In *Second ARRL Amateur Radio Computer Networking Conference*, 1983. Department of Electrical Engineering, Linkoping University.

[ZG97] Simon Znaty and Marie-Pierre Gervais. *Les réseaux intelligents : Ingénierie des services de télécommunication.* Hermès, September 1997.

Communications produites

Conférences internationales avec comité de lecture

- Mudumbai Ranganathan, Laurent Andrey, **Virginie Galtier** et Virginie Schaal. AGNI : encore des agents mobiles !. Dans les *Actes du 7e Colloque Francophone sur l'Ingénierie des Protocoles (CFIP'99)*, pages 383-398, Nancy, France, avril 1999.
- Mudumbai Ranganathan, Virginie Schaal, **Virginie Galtier** et Douglas Montgomery. Mobile Streams : A Middleware for Reconfigurable Distributed Scripting. Dans les *Proceedings of the First International Symposium on Agent Systems and Applications, Third International Symposium on Mobile Agents (ASAMA '99)*, pages 162-175, Palm Springs, Californie, États-Unis, octobre 1999.
- Yannick Carlinet, **Virginie Galtier**, Kevin Mills, Stefan Leigh et Andrew Rukhin. Calibrating an active network node. In *Workshop on Active Middleware Services convened in conjunction with HPDC-9 (Ninth IEEE International Symposium on High Performance Distributed Computing)*, Pittsburgh, Pennsylvanie, États-Unis, août 2000.
- **Virginie Galtier**, Kevin L. Mills, Yannick Carlinet, Stefan Leigh et Andrew Rukhin. Expressing meaningful processing requirements among heterogeneous nodes in an active network. In *Second International Workshop on Software and Performance (WOSP 2000)*, Ottawa, Canada, septembre 2000.
- **Virginie Galtier**, Kevin L. Mills, Yannick Carlinet, Stephen Bush et Amit Kulkarni. Predicting and Controlling Resource Usage in a Heterogeneous Active Network. In *Workshop on Active Middleware Services convened in conjunction with HPDC-10*, San Francisco, Californie, États-Unis, août 2001.
- **Virginie Galtier**, Kevin Mills et Yannick Carlinet. Predicting resource demand in heterogeneous active networks. In *MILCOM 2001*, Vienna, Virginie, États-Unis, octobre 2001.

Revue nationale avec comité de lecture

- **Virginie Galtier**. Un élément de gestion des réseaux actifs. *Technique et science informatiques, numéro spécial Agents et Code Mobile*, 2002 (à paraître).

Rapport de DEA

- **Virginie Galtier**. Décentraliser la gestion de réseaux grâce aux agents mobiles. UHP-Nancy I / ESIAL, septembre 1998.

Autres communications

- **Virginie Galtier**, Craig Hunt, Stefan Leigh, Kevin L. Mills, Doug Montgomery, Mudumbai Ranganathan, Andrew Rukhin et Debra Tang. How Much CPU Time? Expressing Meaningful Processing Requirements among Heterogeneous Nodes in an Active Network. Rapport technique NIST TR-ANTDANETS-111999, présenté lors d'une conférence de travail DARPA, Albuquerque, Nouveau Mexique, États-Unis, septembre 1999.
- Craig Hunt, Kevin Mills et **Virginie Galtier**. A model expressing meaningful processing requirements among heterogeneous nodes in an active network. Poster présenté à une délégation du National Research Council, Gaithersburg, Maryland, États-Unis, janvier 2000.
- Stephen Bush, Amit Kulkarni, **Virginie Galtier**, Kevin L. Mills, Yannick Carlinet et Livio Ricciulli. Predicting and Controlling Resource Usage in an Active Network. Présentation et démonstration lors d'une conférence DARPA, Atlanta, Georgie, États-Unis, décembre 2000.
- Kevin Mills et **Virginie Galtier**. ITL-GE Researchers Demonstrate Mobile-Code Control. In *gazette du NIST Information Technology Laboratory*, décembre 2000.
- **Virginie Galtier**, Yannick Carlinet, Kevin Mills, Stephen Bush et Amit Kulkarni. Predicting and Controlling Resource Usage in an Active Network. Poster présenté à une délégation du National Research Council, Gaithersburg, Maryland, États-Unis, février 2001.
- **Virginie Galtier**, Yannick Carlinet, Kevin Mills, Stefan Leigh et Andrew Rukhin. Measurement ans Modeling for Resource Prediction and Control in Heterogeneous Active Networks. Présentation lors d'une conférence DARPA, Jackson Hole, Wyoming, États-Unis, juin 2001.
- **Virginie Galtier**, Kevin Mills et Yannick Carlinet. Modeling CPU Demand in Heterogeneous Active Networks. Article de synthèse pour DANCE (DARPA *Active Network Conference and Exposition*), Californie, États-Unis, mai 2002.

Sigles utilisés

AA : Application Active, application précisant comment les paquets qu'elle utilise doivent être traités aux nœuds.

ABOCC : *ABOne Coordination Center*, enregistre et gère les différents acteurs du ABone : développeurs d'EE et d'AA, nœuds actifs, serveurs de code.

ABone : *active network backbone*, réseau actif expérimental mis en place pour permettre le déploiement et le test à assez grande échelle de composants actifs.

AGNI : *AGent at NIst*, plate-forme à agents mobiles développée au NIST dans ANTD.

ALF : *Application Level Framing*, architecture permettant d'inclure la sémantique des données de l'application dans la conception du protocole de l'application.

AMARRAGE : Architecture Multimedia & Administration Réparties sur un Réseau Actif à Grand Échelle, projet RNRT auquel RESEDAS participe et qui porte sur la conception et le déploiement au niveau national d'une plate-forme de réseau actif dans un objectif d'évaluation de services innovants.

ANAIS : *Active Network Architecture for Internet Service Providers*, architecture de supervision pour des réseaux actifs développée au LORIA permettant à un fournisseur de déployer ses services efficacement et de façon sécurisée.

ANANA : *Active Network Assigned Number Authority*, organisation accordant un « type » (pour les paquets ANEP) aux projets de recherche le demandant.

ANEP : *Active Network Encapsulation Protocol*, définit le format des paquets actifs.

ANETD : *Active NETwork Daemon*, permet d'échanger des fichiers avec un nœud actif distant, de l'interroger pour connaître la liste des services qu'il propose, de lui demander de télécharger un nouveau service ou de tuer un service en place.

ANTD : *Advanced Network Technologies Division*, département du NIST se consacrant aux réseaux.

ANTS : *Active Node Transfer System*, une plate-forme pour le développement, le déploiement et l'exécution d'applications actives.

API : *Application Program Interface*, interface (conventions d'appel) par laquelle un programme applicatif accède au système d'exploitation ou à d'autres services; par extension, une représentation abstraite d'un niveau inférieur présenté à un niveau supérieur.

ASP : *Active Signaling Protocol*, un environnement d'exécution actif développé à l'Université de Southern California.

ATM : *Asynchronous Transfer Mode*, réseau à commutation de cellules orienté connexion.

AVNMP : *Active Virtual Network Management Prediction*, système de gestion prédic-

tive des réseaux développé par General Electric.

CANEs : *Composable Active Network Elements*, un environnement d'exécution développé à Georgia Tech.
ch : abbréviation utilisée dans ce document pour *cycle d'horloge*, une unité de mesure du temps d'exécution.
CISC : *Complex Instruction Set Computer*, une architecture de processeur.
CPU : *Central Processing Unit*, partie d'un ordinateur contrôlant toutes les autres. La CPU est composée d'un ou plusieurs processeurs, d'une unité arithmétique et logique (*ALU*) et de différents types de mémoire et *buffers*. Par abus de langage, on utilise parfois « CPU » pour parler de « processeur ».

DARPA : *Defense Advanced Research Project Agency*, département de la recherche pour la défense des États-Unis.

EE : Environnement d'Exécution, machine virtuelle dans laquelle s'exécutent les applications actives.

FIS : utilisé dans ce document en abbréviation de *fonction d'interface standard*. L'ensemble des FIS constitue l'API qu'un nœud actif présente aux environnements d'exécution.

GSMP : *General Switch Management Protocol*, protocole générique utilisé pour la gestion (configuration et contrôle) à distance de commutateurs. La version 2.0 est documentée dans le RFC 2297.

IN : *Intelligent Network*, voir *Réseau Intelligent*.

JVM : *Java Virtual Machine*, interpréteur de programmes Java compilés en *byte-code*.

MIB : *Management Information Base*, base de données d'informations (variables et tables) à gérer présente aux nœuds et accessible via le protocole SNMP.

NIST : *National Institute of Standards and Technology*, laboratoire de recherche du gouvernement américain.

OS : *Operating System*, système d'exploitation permettant aux applications d'accéder aux ressources matérielles et logiques d'une machine.

PDA : *Personal Digital Assistant*, un ordinateur tenant dans la main et pouvant être connecté à un ordinateur classique (desktop) pour y déposer ou y récupérer des informations.
PLAN : *Packet Language for Active Networks*, environnement d'exécution restrictif pour les paquets actifs, développé à l'Université de Pennsylvanie.

QCMD : *Query Certificate Manager Daemon*, infrastructure à clé publique permettant la maintenance sécurisée d'ensembles de données distribuées comme par exemple les listes de contrôle d'accès (*ACL*) utilisées par l'ABOCC pour gérer l'ABone.

QoS : *Quality of Service*, qualité de service, niveau de garantie associé à un transport de flux sur le réseau.

RCANE : *Resource Controlled Active Network Environment*, architecture active développée à l'Université de Cambridge.

RGT : Réseau de Gestion des Télécommunications, structure générique pour le développement d'applications de gestion des réseaux de télécommunications.

RI : Réseau Intelligent (*IN* en anglais), architecture facilitant l'introduction de nouveaux services dans les réseaux de télécommunications (architecture normalisée par l'ITU-T dans les recommandations des séries Q120x et Q121x).

RISC : *Reduced Instruction Set Computer*, une architecture de processeur.

RRAS : *Routing and Remote Access Service*, un service permettant d'administrer à distance des routeurs Windows NT.

RSVP : *resource ReSerVation Protocol*, protocole utilisé par un nœud pour demander au réseau une certaine qualité de service pour un flot de données particulier et par un routeur pour réserver les ressources nécessaires pour assurer cette qualité de service le long du chemin suivi par les données. La première version est documentée par le RFC 2205.

RTC : Réseau Téléphonique Commuté, le réseau téléphonique traditionnel.

SANTS : *Secure ANTS*, une version d'ANTS plus sécurisée, développée à NAI Labs.

SNAP : *Safe and Nimble Active Packets*, un langage pour la programmation des paquets actifs.

SNMP : *Simple Network Management Protocol*, protocole très répandu pour gérer des équipements réseau (RFC 1157).

TMN : *Telecommunications Management Network*, voir *RGT*.

TTL : *Time-To-Live*, « crédit de vie » accordé aux paquets IP par exemple. Ce nombre figure dans l'en-tête du paquet et est diminué à chaque saut (passage d'une machine à une autre dans le réseau). Lorsque ce crédit est épuisé, le paquet est détruit. Ce champ s'appelle *Hop Count* dans la version 6 d'IP.

UDP : *User Datagram Protocol*, service sans garantie utilisant IP pour le transport de messages entre machines.

ut : abbréviation utilisée dans ce document pour désigner une *unité de temps* non précisée.

Annexes

A

Applications actives utilisées

Les diapositives suivantes résument le fonctionnement des applications actives *Ping* et *Multicast* pour ANTS et *Ping* pour Magician.

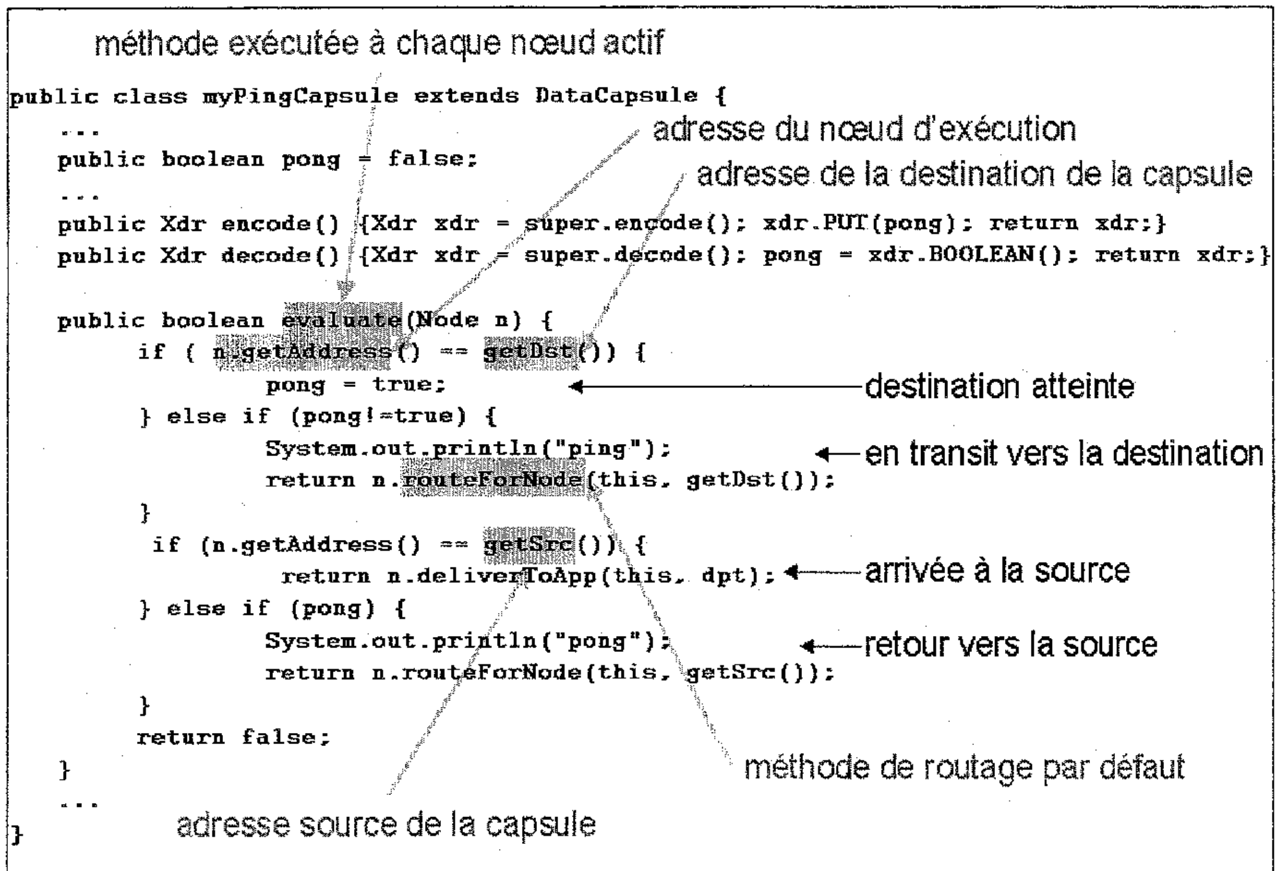


FIG. A.1 – Ping ANTS

```
public class myMulticastApplication extends Application {

   int target;  <-- serveur
   int group;   <-- groupe multicast
   int sdelay;  <-- délai entre les inscriptions

   synchronized public void receive(Capsule cap) {
         ...
         myMulticastMessageCapsule mMMcap = (myMulticastMessageCapsule)(cap);
         System.out.println("message "+ mMMcap.indice+" received");
   }

   public void start() throws Exception { ...
         new Thread(new myMulticastSubscribeDaemon(this)).start();
   }...
}

class myMulticastSubscribeDaemon implements Runnable { ...
   public void run() {
         while (true) { ...
                  Thread.sleep(myMulticastApp.sdelay);
                  System.out.println("send subscribe");
                  myMulticastApp.send(new myMulticastSubscribeCapsule(
                           myMulticastApp.target, myMulticastApp.group));
          }
   }...
}
```

affiche les messages reçus du serveur multicast

envoie une inscription au serveur *target* pour le groupe de multicast *group* toutes les *sdelay* secondes

FIG. A.2 – Multicast ANTS : côté client

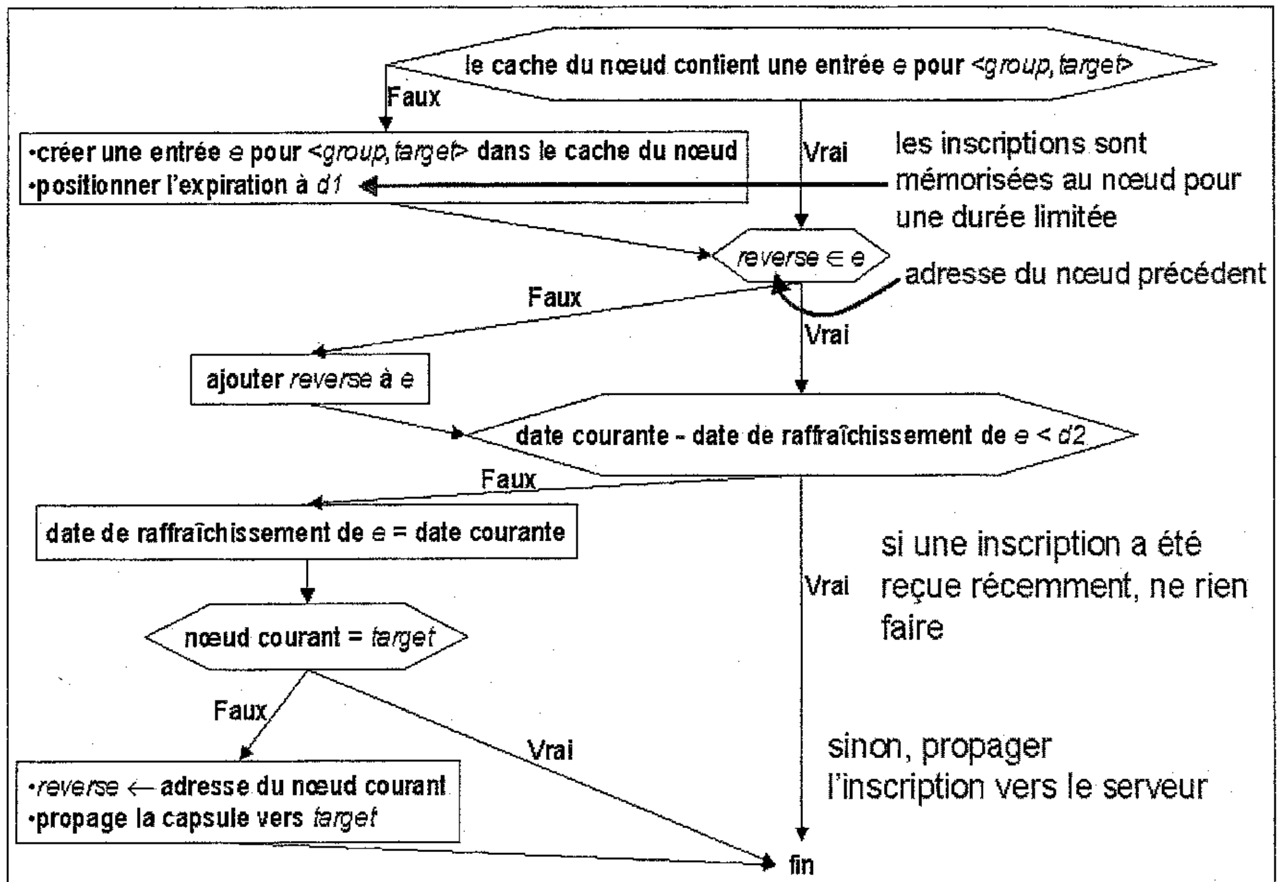


FIG. A.3 – Multicast ANTS : méthode *evaluate* d'une capsule d'inscription

```
public class myMulticastServer extends Application implements Runnable {
   int target, group;
   ...
   public void run() {
         ...
         for (int i=0; i < iter; i++) {
                  send(new myMulticastMessageCapsule(group,target,...,i,...));
                  System.out.println("message "+i+" sent"); ...
                  Thread.sleep(interv); ...
         }
   }
...}
```

FIG. A.4 – Multicast ANTS : côté serveur

S.C.D. - U.H.P. NANCY 1
BIBLIOTHEQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

```
public class myMulticastMessageCapsule extends Capsule {
    ...
    int group;
    int target;
    ByteArray data;
    int indice;
    ...
    public boolean evaluate(Node n) {
        W_JAI m = (W_JAI)n.getCache().get(group, target);
        if (m != null) {
                if (m.ail != null) {
                        for (int i = 0; i < m.ail.length; i++) {
                                if (m.ail[i] == n.getAddress()) {
                                        n.deliverToApp(this, dpt);
                                } else {
                                        n.routeForNode(this, m.ail[i]);
                                }
                        }
                }
        }
        return true;
    }
    ...
}
```


FIG. A.5 – Multicast ANTS : méthode *evaluate* d'une capsule de message

```
public class SmartPing extends magician.Node.KUSmartPacketV2 {

   private boolean hasPinged;
   private boolean reachedDestination;

   public SmartPing (){
        hasPinged = false;
        reachedDestination = false;
   }

   public void exec() {
        String NodeName = GetNodeName();
        if (!hasPinged) {
                System.out.println("Ping");
        }
        if (NodeName.equals(Destination_Address)) {
                hasPinged = true;
                Destination_Address = Source_Address;
                Source_Address = NodeName;
        }
        if (hasPinged) {
                System.out.println("Pong");
        }
   }
}
```

FIG. A.6 – Ping Magician

B

Éléments du fonctionnement de Linux

Cette partie résume certains aspects de Linux dont la connaissance est nécessaire à la compréhension du chapitre 3. Pour une description plus détaillée, consulter [CDM97] et [BBD+98].

Linux est né il y a une dizaine d'années lorsque pour étudier l'architecture des ordinateurs Linus Torvalds étend Minix (un mini-Unix développé entre autres par Andrew Tanenbaum) et diffuse les sources de son travail sur Internet. Au départ assez rudimentaire, Linux est à présent compétitif avec d'autres systèmes Unix commerciaux grâce aux contributions et évolutions apportées par une communauté mondiale de développeurs bénévoles, le code source reste librement et gratuitement accessible sur Internet et de nombreux groupes de discussion et listes de courrier électronique lui sont consacrés.

Linux respecte les spécifications POSIX et possède en plus certaines extensions d'autres versions d'Unix. Cela facilite le portage du code d'applications développées pour d'autres Unix.

[CDM97] récapitule les principales caractéristiques de Linux :
- muti-tâches,
- multi-processeurs,
- multi-plates-formes,
- multi-utilisateurs,
- support des communications inter-processus,
- gestion des signaux,
- gestion des terminaux (suivant la norme POSIX),
- support d'un grand nombre de périphériques populaires,
- *buffer cache* (zone mémoire servant de tampon pour les entrées/sorties des différents processus),
- gestion mémoire « à la demande » (une page n'est chargée en mémoire que lorsque c'est nécessaire),
- bibliothèques partagées et dynamiques,
- système de fichiers permettant de gérer différents formats de partitions,
- support de la couche TCP/IP et d'autres protocoles réseaux.

Comme pour tout système d'exploitation moderne, le rôle de Linux est de « masquer » la machine physique sur laquelle il s'exécute et de fournir aux applications des abstractions de haut niveau et une interface évoluée. Il possède une interface avec le

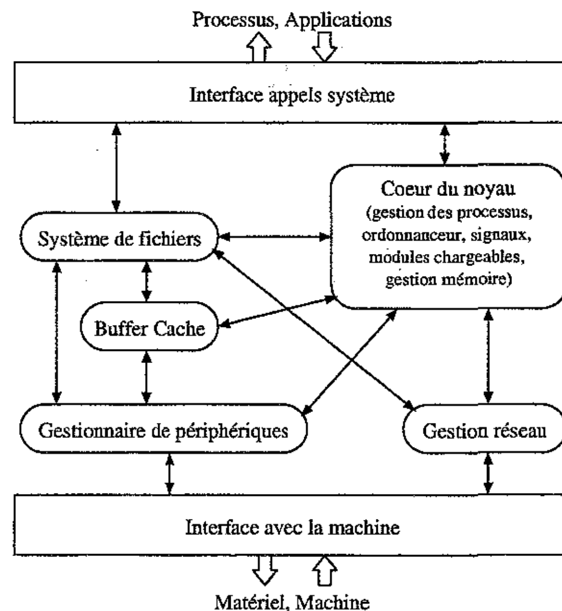


FIG. B.1 – Architecture de Linux

matériel, qui le notifie de différents évènements (par exemple frappe d'une touche au clavier, ou « sonnerie » d'un *timer*) par des « interruptions » et avec lequel il communique par requêtes de bas niveau (code généralement écrit en assembleur) pour les tâches matérielles (par exemple lire physiquement des octets sur un disque étant donné leur emplacement décrit par une adresse physique telle un numéro de disque, de cylindre, de tête de lecture et de secteur). Différentes fonctionnalités sont bâties au-dessus de cette interface : système de fichiers, gestionnaires de périphériques, gestion réseau...

Pour permettre le partage des ressources matérielles entre plusieurs applications et faciliter l'écriture d'applications moins dépendantes du matériel, les applications ne peuvent pas accéder le matériel directement elles-mêmes. Elles doivent passer par les fonctionnalités offertes par Linux en effectuant des « appels système » au travers de l'interface haut-niveau de Linux.

La figure B.1, empruntée à [CDM97], résume cette structuration.

Dans son mode d'exécution normal (ou « mode utilisateur »), un processus n'a accès qu'à certaines instructions et aux ressources qui lui ont été allouées, et il peut-être interrompu n'importe quand. Pour accéder aux ressources de la machine (autres zones de mémoire, contrôleur de périphériques...), il doit effectuer un appel système. Les appels système sont censés être fiables et peuvent donc s'exécuter dans un mode privilégié (dit « mode noyau »).

Linux est un système multi-tâches. Cela signifie que plusieurs programmes ou processus peuvent s'exécuter en pseudo-parallèle : le système donne le processeur à tour de rôle à chacun des processus pendant une courte période de temps, donnant ainsi l'illusion à l'utilisateur que tous les processus s'exécutent en même temps (alors que s'il n'y a qu'un processeur un seul programme est actif à un instant donné). Chaque processus a un type et des priorités utilisés par l'ordonnanceur au moment de choisir le processus

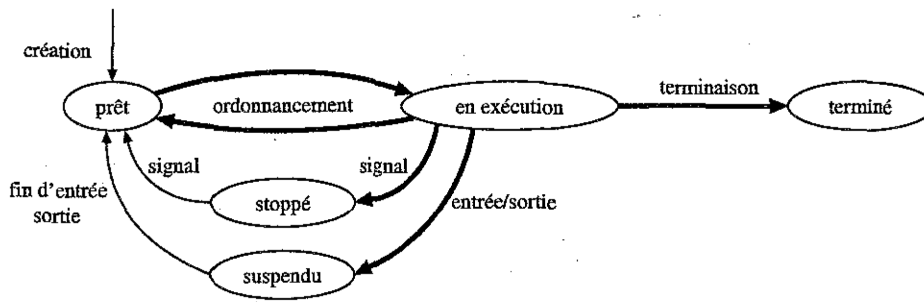


FIG. B.2 – Graphe d'états d'un processus Linux

qui va acquérir le processeur. Les changements d'état d'un processus sont résumés sur la figure B.2.

Dans le chapitre 3, nous instrumentons les transitions représentées sur le schéma par les flèches en gras.

C

Modifications apportées au code du noyau Linux

Cette annexe indique en détails les modifications que j'ai apportées au code du noyau Linux pour introduire la possibilité d'un monitorage fin des appels systèmes réalisés par un processus, comme expliqué au chapitre 3.

... indique l'omission de code existant, + indique les lignes ajoutées et ~ les lignes modifiées.

C.1 linux/include/linux/sched.h

Modification de la structure des processus :

```
...
struct task_struct {
          ...
+         unsigned long long int e; /* user or kernel entry */
+         unsigned long long int ucc; /* nb clock cycles in user mode */
+         unsigned long long int kcc; /* nb clock cycles in kernel mode */
+         int kflag; /* kflag put to 1 on entry in the kernel, to 0 on exit */
+         int scn; /* system call number */
+         int tflag; /* 1 to trace the process */
};
...
```

C.2 linux/arch/i386/kernel/entry.S

À la section 3.3.2, nous avons identifié 4 points-clé nécessitant une instrumentation : obtention du processeur, invocation d'un appel système, retour d'un appel système et cession du processeur. Nous ajoutons ici le code appelant nos fonctions de mise à jour des informations de monitorage, en prenant bien soin de restituer la pile à l'identique.

```
...
```

```
ENTRY(lcall7)
        pushfl
        pushl %eax
+        pushl %es;
+        pushl %ds;
+        pushl %eax;
+        pushl %ebp;
+        pushl %edi;
+        pushl %esi;
+        pushl %edx;
+        pushl %ecx;
+        pushl %ebx;
+        pushl %eax;
+        call SYMBOL_NAME (entry_to_kernel);
+        popl %eax;
+        popl %ebx;
+        popl %ecx;
+        popl %edx;
+        popl %esi;
+        popl %edi;
+        popl %ebp;
+        popl %eax;
+        popl %ds;
+        popl %es;
        SAVE_ALL
        movl EIP(%esp),%eax
        movl CS(%esp),%edx
        movl EFLAGS(%esp),%ecx
        movl %eax,EFLAGS(%esp)
        movl %edx,EIP(%esp)
        movl %ecx,CS(%esp)
        movl %esp,%ebx
        pushl %ebx
        andl $-8192,%ebx
        movl exec_domain(%ebx),%edx
        movl 4(%edx),%edx
        call *%edx
        popl %eax
+        pushl %es;
+        pushl %ds;
+        pushl %eax;
+        pushl %ebp;
+        pushl %edi;
+        pushl %esi;
+        pushl %edx;
```

```
+        pushl %ecx;
+        pushl %ebx;
+        call SYMBOL_NAME (exit_from_kernel);
+        popl %ebx;
+        popl %ecx;
+        popl %edx;
+        popl %esi;
+        popl %edi;
+        popl %ebp;
+        popl %eax;
+        popl %ds;
+        popl %es;
        jmp ret_from_sys_call
...
ENTRY(system_call)
        pushl %eax
+        pushl %es;
+        pushl %ds;
+        pushl %eax;
+        pushl %ebp;
+        pushl %edi;
+        pushl %esi;
+        pushl %edx;
+        pushl %ecx;
+        pushl %ebx;
+        pushl %eax;
+        call SYMBOL_NAME (entry_to_kernel);
+        popl %eax;
+        popl %ebx;
+        popl %ecx;
+        popl %edx;
+        popl %esi;
+        popl %edi;
+        popl %ebp;
+        popl %eax;
+        popl %ds;
+        popl %es;
        SAVE_ALL
        GET_CURRENT(%ebx)
        cmpl $(NR_syscalls),%eax
        jae badsys
        testb $0x20,flags(%ebx)
        jne tracesys
        call *SYMBOL_NAME(sys_call_table)(,%eax,4)
        movl %eax,EAX(%esp)
```

```
+       pushl %es;
+       pushl %ds;
+       pushl %eax;
+       pushl %ebp;
+       pushl %edi;
+       pushl %esi;
+       pushl %edx;
+       pushl %ecx;
+       pushl %ebx;
+       call SYMBOL_NAME (exit_from_kernel);
+       popl %ebx;
+       popl %ecx;
+       popl %edx;
+       popl %esi;
+       popl %edi;
+       popl %ebp;
+       popl %eax;
+       popl %ds;
+       popl %es;
...
ENTRY(sys_call_table)
...
+       .long SYMBOL_NAME(sys_set_tflag) /* 191 */
+       .long SYMBOL_NAME(sys_trace_tag) /* 192 */
+       .long SYMBOL_NAME(sys_get_pid_cc) /* 193 */}
~       .rept NR_syscalls-{\bf 193}
                .long SYMBOL_NAME(sys_ni_syscall)
       .endr
```

C.3 linux/arch/i386/kernel/sys_i386.c

C'est ici que nous définissons et implémentons les fonctions de mise à jour des informations de monitorage utilisées à l'invocation et au retour d'appels système.

```
...
+  __inline__ unsigned long long int rdtscdeux()
+  {
+    unsigned long long int x;
+    __asm__ volatile (".byte 0x0f, 0x31" : "=A" (x));
+    return x;
+  }
+
+
+
+  void entry_to_kernel(int eax )
```

```
+  {
+    unsigned long long int aux;
+    lock_kernel();
+    if (current->tflag == 1) {
+      aux = rdtscdeux() - current->e;
+      if ((eax == 192) || (eax == 193)) {
+        current->ucc = current->ucc + aux;
+        current->scn = eax;
+      } else {
+        current->scn = eax;
+        current->kflag = 1;
+        current->ucc = current->ucc + aux;
+        current->e = rdtscdeux();
+        current->tflag = 0;
+        printk(KERN_NOTICE "trace%u I%u U %.8x %.8x \n",  current->pid,
                               current->scn,
                               (long)(((current->ucc)>>32)&0xffffffff),
                               (long)((current->ucc)&0xffffffff));
+        current->tflag = 1;
+      }
+    }
+    unlock_kernel();
+  }
+
+  void exit_from_kernel()
+  {
+    unsigned long long int aux;
+    lock_kernel();
+    if (current->tflag == 1) {
+      aux = rdtscdeux() - current->e;
+      if ((current->scn == 192) || (current->scn == 193)) {
+        current->e = rdtscdeux();
+      } else {
+        current->kflag = 0;
+        current->kcc = current->kcc + aux;
+        current->e = rdtscdeux();
+        current->tflag = 0;
+        printk(KERN_NOTICE "trace%u O%u K %.8x %.8x \n",  current->pid,
                               current->scn,
                               (long)(((current->kcc)>>32)&0xffffffff),
                               (long)((current->kcc)&0xffffffff));
+        current->tflag = 1;
+      }
+    }
+    unlock_kernel();
```

```
+  }
```

C.4 linux/kernel/sched.c

Mise à jour des informations de monitorage à l'obtention et à la cession du processeur.

```
...
+  __inline__ unsigned long long int rdtscun()
+  {
+    unsigned long long int x;
+    __asm__ volatile (".byte 0x0f, 0x31" : "=A" (x));
+    return x;
+  }
...

asmlinkage void schedule(void)
{
        struct schedule_data * sched_data;
        struct task_struct * prev, * next;
        int this_cpu;
+       unsigned long long int aux;

        run_task_queue(&tq_scheduler);

        prev = current;
        this_cpu = prev->processor;
        sched_data = & aligned_data[this_cpu].schedule_data;

        if (in_interrupt())
                goto scheduling_in_interrupt;
        release_kernel_lock(prev, this_cpu);

        if (bh_active & bh_mask)
                do_bottom_half();

        spin_lock(&scheduler_lock);
        spin_lock_irq(&runqueue_lock);

        prev->need_resched = 0;

        if (!prev->counter && prev->policy == SCHED_RR) {
                prev->counter = prev->priority;
                move_last_runqueue(prev);
        }
```

```
	switch (prev->state) {
		case TASK_INTERRUPTIBLE:
			if (signal_pending(prev)) {
				prev->state = TASK_RUNNING;
				break;
			}
		default:
			del_from_runqueue(prev);
		case TASK_RUNNING:
	}

	sched_data->prevstate = prev->state;

/* this is the scheduler proper: */
	{
		struct task_struct * p = init_task.next_run;
		int c = -1000;

		next = idle_task;
		if (prev->state == TASK_RUNNING) {
			c = goodness(prev, prev, this_cpu);
			next = prev;
		}

		spin_unlock_irq(&runqueue_lock);

		while (p != &init_task) {
			if (can_schedule(p)) {
				int weight = goodness(p, prev, this_cpu);
				if (weight > c)
					c = weight, next = p;
			}
			p = p->next_run;
		}

		if (!c) {
			struct task_struct *p;
			read_lock(&tasklist_lock);
			for_each_task(p)
				p->counter = (p->counter >>1) + p->priority;
			read_unlock(&tasklist_lock);
		}
	}

#ifdef __SMP__
```

```
...
#endif /* __SMP__ */
         if (prev != next) {
#ifdef __SMP__
...
#endif
                kstat.context_swtch++;
                get_mmu_context(next);

+               if (prev->tflag == 1) {
+                 lock_kernel();
+                 aux =  rdtscun() - (prev->e);
+                 if (prev->kflag == 0) {
+                   prev->ucc=(prev->ucc) + aux;
+                 } else {
+                   prev->kcc=(prev->kcc) + aux;
+                 }
+                 prev->tflag = 0;
+                 printk(KERN_NOTICE "trace%u exitU%u\n", prev->pid, prev->ucc);
+                 prev->tflag = 1;
+                 unlock_kernel();
+               }

                switch_to(prev,next);

+               if (prev->tflag == 1) {
+                       prev->e=rdtscun();
+               }

                __schedule_tail();
        }

        reacquire_kernel_lock(current);
        return;

scheduling_in_interrupt:
        printk("Scheduling in interrupt\n");
        *(int *)0 = 0;
}
...
```

C.5 linux/include/asm/unistd.h

Déclaration des appels sytèmes ajoutés.

```
...
+ #define __NR_set_tflag          191 /* ActiveNet */
+ #define __NR_trace_tag          192 /* ActiveNet */
+ #define __NR_get_pid_cc         193 /* ActiveNet */
...
```

C.6 linux/kernel/sys.c

Implémentation des appels systèmes ajoutés.

```
...
+ __inline__ unsigned long long int rdtsctrois()
+ {
+   unsigned long long int x;
+   __asm__ volatile (".byte 0x0f, 0x31" : "=A" (x));
+   return x;
+ }
+
+
+ asmlinkage int sys_set_tflag(pid_t thePid)
+ {
+   struct task_struct *p;
+   lock_kernel();
+   if (p = find_task_by_pid(thePid)) {
+     p->ucc = 0;
+     p->kcc = 0;
+     p->tflag = 1;
+     p->kflag = 0;
+     p->e = rdtsctrois();
+     unlock_kernel();
+     return 0;
+   } else {
+     printk(KERN_NOTICE "PROBLEM: process %u not found\n",thePid);
+     unlock_kernel();
+     return -1;
+   }
+ }
+
+
+
+ asmlinkage int sys_trace_tag(char* message)
+ {
+   lock_kernel();
+   if (current->tflag == 1) {
+     current->tflag = 0;
```

```
+      printk(KERN_NOTICE "ANtag%u %s\n",  current->pid, message);
+      printk(KERN_NOTICE "ANtag%u K %.8x %.8x \n", current->pid,
                           (long)(((current->kcc)>>32)&0xffffffff),
                           (long)((current->kcc)&0xffffffff));
+      printk(KERN_NOTICE "ANtag%u U %.8x %.8x \n", current->pid,
                           (long)(((current->ucc)>>32)&0xffffffff),
                           (long)((current->ucc)&0xffffffff));
+      current->tflag = 1;
+    }
+    unlock_kernel();
+    return 0;
+  }
+
+
+
+  asmlinkage int sys_get_pid_cc(pid_t thePid, unsigned long long int *res)
+  {
+    unsigned long long int result, auxUcc, auxKcc;
+    struct task_struct *p;
+    lock_kernel();
+    memset((char *) &result, 0, sizeof(result));
+
+    result = 0;
+    if (p = find_task_by_pid(thePid)) {
+      if (p->tflag == 1) {
+        if (p->kflag == 0) {
+          auxUcc = p->ucc + rdtsctrois() - p->e;
+          auxKcc = p->kcc;
+        } else {
+          auxUcc = p->ucc;
+          auxKcc = p->kcc + rdtsctrois() - p->e;
+        }
+        result = auxUcc + auxKcc;
+      }
+    }
+    unlock_kernel();
+    return copy_to_user(res, &result, sizeof(result)) ? -EFAULT : 0;
+  }
```

D

Taxinomie des benchmarks

La figure D.1, inspirée de [GHL+99], présente une classification des *benchmarks* classiques examinés pour la phase de calibration des nœuds exposée au chapitre 4.

Le programme de calibration adopté pour la partie « mode privilégié » (conduisant au vecteur NP de la figure 4.3) se classe dans la branche Synthétique/Dynamique/Single-Thread tandis que celui utilisé pour la partie « mode utilisateur » (conduisant au vecteur NU) est Réel/Dynamique.

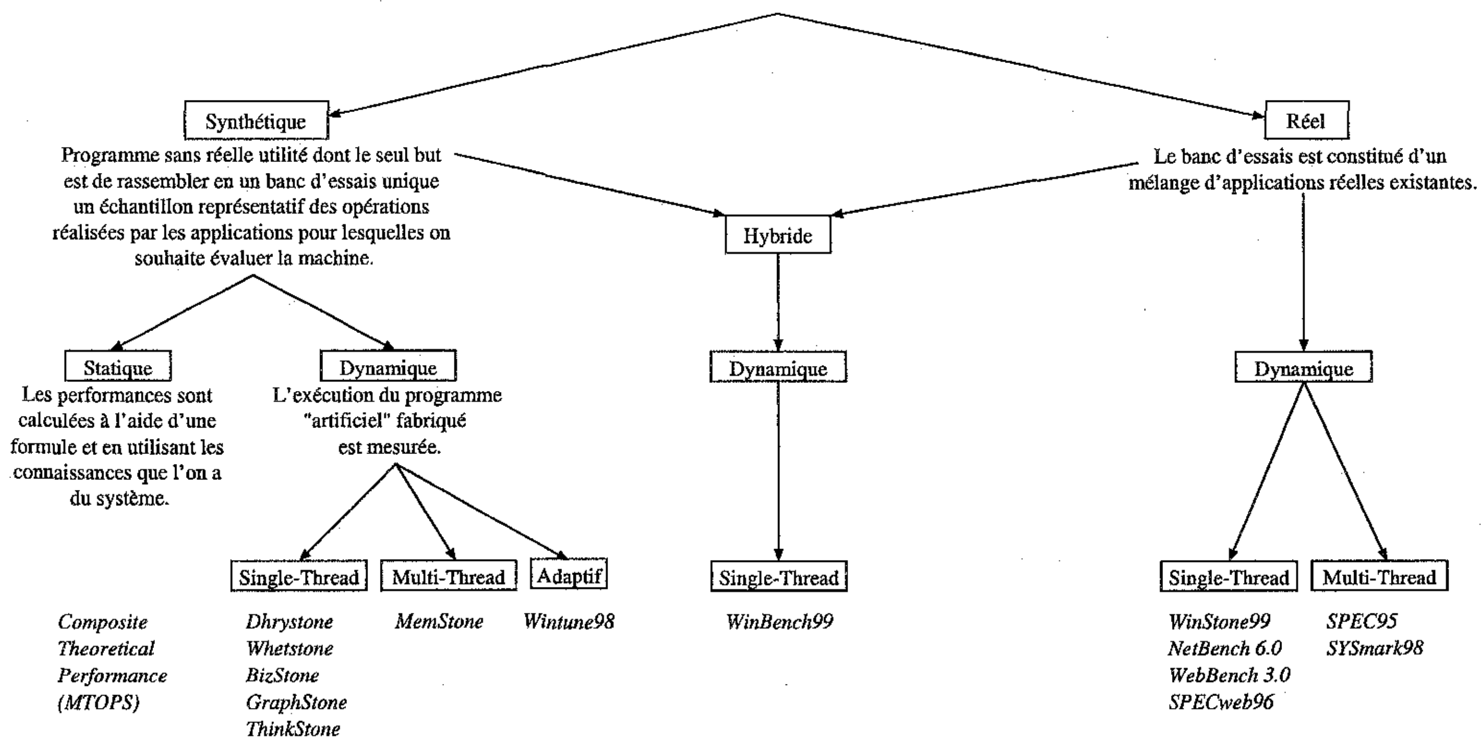


FIG. D.1 – Taxinomie des *benchmarks*

S.C.D. - U.H.P. NANCY 1
BIBLIOTHEQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

E

Prédictions des différents modèles

Le tableau E.1 présente les erreurs des prédictions obtenues avec les différentes techniques : en utilisant directement comme prédiction pour un nœud Y le temps d'exécution observé sur un nœud X (colonnes « direct »), ou bien en utilisant les modèles M1, M2 ou M3 construits à partir de traces d'exécution de l'application sur le nœud X. Pour chaque méthode employée, la première colonne (« moy. ») indique la valeur absolue de l'erreur en % par rapport au temps moyen d'exécution réellement observé et la deuxième colonne (« h. p. ») indique la moyenne des valeurs absolues des erreurs de prédiction (en %) des 80e, 85e, 90e, 95e et 99e pourcentiles.

				direct		M1		M2		M3	
EE	AA	nœud X	nœud Y	moy.	h. p.	moy.	h. p.	moy.	h. p.	moy.	h. p.
Ants	Ping	Blue	Blue			10	21	0	10	0	7
			Green	141	143	24	22	8	10	15	9
			Black	145	116	21	21	7	10	7	7
			Red	181	139	26	24	0	11	16	8
			Yellow	137	142	15	28	3	13	11	11
		Green	Blue	59	1138	0	24	8	7	28	7
			Green			15	23	0	8	1	7
			Black	119	2646	12	23	12	6	18	4
			Red	16	275	18	22	1	8	8	7
			Yellow	2	440	5	17	4	10	16	9
		Black	Blue	321	312	8	23	10	7	10	8
			Green	634	262	22	20	9	8	8	7
			Black			19	21	0	8	1	5
			Red	722	193	24	19	9	8	11	5
			Yellow	625	302	13	21	3	8	4	8
		Red	Blue	64	1710	3	16	7	7	28	9
			Green	14	507	13	16	0	7	2	8
			Black	116	3886	10	20	10	7	2	8
			Red			16	21	0	8	12	5
			Yellow	16	4460	3	26	6	8	21	10

Suite sur la page suivante

				direct		M1		M2		M3	
EE	AA	nœud X	nœud Y	moy.	h. p.	moy.	h. p.	moy.	h. p.	moy.	h. p.
		Yellow	Blue	58	347	23	23	3	13	5	13
			Green	2	158	23	25	2	7	0	14
			Black	119	721	20	22	18	10	7	13
			Red	18	240	26	27	4	12	3	13
			Yellow			14	35	0	4	1	6
		moyenne		175	907	15	22	5	9	9	8
Ants	Multicast	Blue	Blue			10	23	0	3	11	12
			Green	242	56	17	19	9	10	17	13
			Black	483	76	21	22	6	9	16	12
			Red	516	63	13	23	1	5	9	11
			Yellow	401	51	17	21	16	14	12	13
		Green	Blue	170	146	6	22	11	12	5	16
			Green			14	23	0	4	10	8
			Black	169	174	18	23	2	5	9	14
			Red	193	113	10	24	8	10	7	15
			Yellow	112	182	14	24	7	6	7	11
		Black	Blue	126	120	7	21	10	13	7	16
			Green	63	92	15	28	1	3	9	14
			Black			19	28	0	4	8	9
			Red	9	123	11	25	6	8	6	12
			Yellow	21	146	15	25	7	6	8	10
		Red	Blue	124	246	10	23	3	5	7	14
			Green	66	155	17	28	7	5	13	11
			Black	8	322	21	21	6	6	12	8
			Red			14	28	0	3	7	13
			Yellow	28	354	18	24	13	8	15	14
		Yellow	Blue	133	136	1	26	6	16	7	12
			Green	53	103	9	24	4	12	5	12
			Black	27	152	13	31	6	3	5	14
			Red	38	145	5	24	8	7	10	11
			Yellow			9	27	0	6	15	10
		moyenne		149	148	13	24	6	7	9	12
Magician	Ping	Blue	Blue			99	23	9	15	1	13
			Green	136	110	12	23	6	3	8	18
			Black	147	157	11	28	15	25	5	14
			Red	48	108	13	26	6	25	2	12
			Yellow	3	88	9	24	3	20	8	17
		Green	Blue	376	202	15	36	10	25	13	12
			Green			3	23	3	12	1	9
			Black	31	172	23	30	3	34	6	16
			Red	244	146	25	29	1	33	14	15

Suite sur la page suivante

				direct		M1		M2		M3	
EE	AA	nœud X	nœud Y	moy.	h. p.	moy.	h. p.	moy.	h. p.	moy.	h. p.
Magician	Ping		Yellow	367	136	21	25	1	31	2	13
		Black	Blue	311	323	6	25	10	16	9	14
			Green	23	215	21	33	4	40	4	19
			Black			4	23	0	9	4	7
			Red	210	115	6	21	3	7	7	6
			Yellow	304	325	2	26	0	9	5	18
		Red	Blue	92	217	4	27	1	43	8	15
			Green	169	188	19	23	2	57	6	16
			Black	191	120	6	21	0	42	8	6
			Red			9	24	7	42	2	4
			Yellow	86	159	4	25	3	46	11	15
		Yellow	Blue	3	148	10	23	4	9	4	11
			Green	137	59	25	31	1	9	4	16
			Black	149	173	1	23	1	20	4	12
			Red	46	108	3	22	15	19	9	10
			Yellow			2	22	2	15	0	11
		moyenne		154	163	14	25	4	24	6	13
Magician	Route	Blue	Blue			0	25	0	17	1	11
			Green	353	464	17	23	14	40	12	16
			Black	182	227	11	19	11	13	9	10
			Red	273	233	2	25	1	16	3	12
			Yellow	123	626	13	21	13	3	7	19
		Green	Blue	140	143	20	25	11	16	4	7
			Green			10	21	0	26	1	4
			Black	68	91	28	36	6	14	6	5
			Red	31	106	21	33	3	15	7	5
			Yellow	91	121	30	37	5	22	6	5
		Black	Blue	222	134	5	16	12	22	8	15
			Green	209	127	24	32	4	50	9	19
			Black			6	26	0	11	2	9
			Red	111	179	4	17	10	17	10	14
			Yellow	72	179	8	26	4	9	4	15
		Red	Blue	158	325	6	22	1	14	10	14
			Green	46	418	24	35	16	34	5	17
			Black	53	245	6	20	9	10	5	9
			Red			4	24	0	11	1	9
			Yellow	87	600	8	21	13	2	10	12
		Yellow	Blue	528	65	6	20	15	34	13	18
			Green	983	49	25	24	35	66	10	15
			Black	251	63	6	21	3	21	4	14
			Red	642	104	4	21	14	27	5	19

Suite sur la page suivante

				direct		M1		M2		M3	
EE	AA	nœud X	nœud Y	moy.	h. p.	moy.	h. p.	moy.	h. p.	moy.	h. p.
			Yellow			7	23	0	20	2	15
		moyenne		231	225	12	25	8	21	6	12
moyenne ANTS						14	23	5	8	9	10
moyenne Magician						13	25	6	23	6	13
moyenne				177	361	14	24	6	15	8	11

TAB. E.1: Comparaison détaillée des pouvoirs de prédiction des 3 modèles

Résumé

Fruit des progrès réalisés dans plusieurs domaines de la science informatique ces dernières années, le concept de réseau actif est actuellement à l'étude pour permettre l'introduction de nouveaux services réseau de façon plus dynamique que dans les réseaux actuels. Il donne aux applications la possibilité de définir la pile de protocoles qu'elles souhaitent voir employée pour le traitement de leurs paquets aux nœuds du réseau. Dans ce nouveau contexte très variable, pour qu'un nœud puisse gérer intelligemment ses ressources en calcul il faut que les applications actives lui déclarent leurs besoins. Malheureusement, s'il existe des unités bien établies pour exprimer les besoins en mémoire (octet) ou en bande passante (bit/seconde) d'un programme, il n'existe pas d'unité pour exprimer les besoins en calcul (temps processeur) qui puisse se traduire sur les machines aux performances parfois très différentes composant un réseau actif.

Dans l'hypothèse où les différents comportements possibles d'une application sont identifiés et peuvent être répétés dans une phase initiale de traçage, nous proposons de modéliser une application par une série de scénarios, où un scénario a une certaine probabilité d'occurrence et est caractérisé par une suite de services que les paquets de l'application demandent aux nœuds sur lesquels ils s'exécutent. Les performances des nœuds actifs sont évaluées par des *benchmarks* dédiés. La combinaison de ces deux informations permet à un nœud de prévoir *a priori* combien de temps CPU l'exécution d'une application active est susceptible de demander. Nous montrons que le nœud peut alors utiliser cette information pour réaliser une « gestion renseignée », plus efficace que la « gestion arbitraire » classique. Nous proposons également de faire évoluer les modèles afin que notre approche reste satisfaisante avec des hypothèses moins restrictives.

Mots-clés: réseaux actifs, hétérogénéité, ressources de calcul, modélisation de code, contrôle, prédiction

Abstract

Active network technology envisions deployment of virtual execution environments within network elements, such as switches and routers, so that inhomogeneous processing can be applied to network traffic associated with services, flows, or even individual packets. To use such a technology safely and efficiently, individual nodes must have a meaningful understanding of the resource requirements for specific network traffic. Well-accepted metrics exist for expressing bandwidth requirements (bits per second) and memory requirements (bytes) in units independent of the capabilities of particular nodes. Unfortunately, no well-accepted metric exists for meaningfully expressing processing (i.e., CPU time) requirements in a platform-independent form.

Based on its execution trace, we model an active application as a set of scenari. Each scenario has a probability of occurrence and is characterized by the set of node services called by the packets of the application during their execution. Node performances are captured by specific benchmarks. The combination of these information allows a node to predict how many CPU time an active application will need to execute. We show that the node can then use this prediction to manage more efficiently its resources. We also propose model evolutions to validate the approach in more general contexts.

Keywords: active network, heterogeneity, computational resources, code modelization, control, prediction