ARFANET : Une nouvelle approche pour les réseaux actifs

Amdjed Mokhtari

To cite this version:

Amdjed Mokhtari. ARFANET : Une nouvelle approche pour les réseaux actifs. Modélisation et simulation. Migration - université en cours d'affectation, 2005. Français. NNT : . tel-00010867v1

HAL Id: tel-00010867
https://theses.hal.science/tel-00010867v1

Submitted on 4 Nov 2005 (v1), last revised 5 Nov 2005 (v2)

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.

UNIVERSITÉ DE VERSAILLES SAINT-QUENTIN-EN-YVELINES

U.F.R DE SCIENCES

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE VERSAILLES
SAINT-QUENTIN-EN-YVELINES

Discipline : INFORMATIQUE

Présentée et soutenue publiquement par

Amdjed MOKHTARI

le 30 Septembre 2005

TITRE

ARFANet : Une nouvelle approche pour les Réseaux Actifs

ARFANet : A new approach for Active Networks

JURY

Gérard HEBUTERNE	Professeur à l'INT d'EVRY	Président
Ken CHEN	Professeur à l'Université Paris XIII	Rapporteur
CongDuc PHAM	Professeur à l'Université de PAU	Rapporteur
Jean-Michel FOURNEAU	Professeur à l'Université de Versailles	Directeur
Leïla KLOUL	MCF à l'Université de Versailles	Co-directeur
Ahmed SERHROUCHNI	MCF à l'ENST de PARIS	Examinateur

Remerciements

Cette thèse n'aurait jamais vu le jour sans la confiance, la patience, la générosité et la contribution de mon encadrant de recherche, Leïla Kloul, que je veux vivement remercier et lui exprimer ma gratitude.

Je veux également remercier Jean-Michel Fourneau de m'avoir accueilli au sein de son équipe, de son soutien et de ses encouragements.

Mes plus chaleureux remerciements vont à Ken Chen et CongDuc Pham pour avoir accepté d'évaluer mes travaux en qualité de rapporteurs. Je leur suis très reconnaissant pour leurs pertinentes remarques et questions qui m'ont permis de clarifier quelques points.

Des remerciements tout particuliers sont adressés à Gérard Hébuterne et Ahmed Serhrouchni pour avoir accepté d'être mes examinateurs et pour leurs commentaires et remarques.

Je tiens spécialement à remercier Jane Hillston et Mokrane Bouzeghoub pour leurs précieuses contributions dans ce travail.

J'adresse mes vifs remerciements à tous mes amis et collègues du laboratoire PRiSM (Université de Versailles) pour leurs présence, aide et encouragements.

Je ne remercierai jamais assez ma famille, surtout mon père pour son aide, conseils, remarques et encouragements qui m'ont accompagné tout au long de l'élaboration de cette thèse.

Enfin, sans oublier tous ceux qui ont contribué à l'aboutissement de ce travail par leur précieuse aide et assistance.

Abstract

The internet reveals this last decade a metamorphosis at the high level of its hierarchy, specially on the nature of the client application, protocols, services and flows. Only the low layers are stiff and adhere faithfully to an obsolete mechanisms as old as the Internet creation. The economic sake and the financial market which wove itself around the actual infrastructure make difficult the transition to worldwide scale of new protocols. The example of reference is IPv6 which finds difficulties to impose itself on the network, though there is several solutions allowing soft transition were envisaged like IPv6 over Ipv4 encapsulation. The active networks are a possible alternative which allow to exceed rapidly and efficiently this impediment in prevision of network saturation caused by the continuous increase of the Internet users and services. In an active network, the treatments to perform in the different nodes may be customised according to the user needs and/or the application requirements. The specific functionalities to an user or an application are loaded within the network nodes as methods or a small programs. Once a user data packet arrives at the specified nodes, its header is evaluated to choose an appropriated program to be executed on the packet content.

The needs to develop common programming model including a common models of network program coding, integrated primitives available in each node and node resources allocation showed the necessity of a common architecture able to recognise different packet languages and execution environments. The concept of an architecture based principally on three layers proved necessary and largely adopted in different projects in active networking area. Those layers are the *active application* layer which constitutes the application services for user data and network management, the *execution environment* layer which interprets the active packets and executes the active applications, and finally the *node operating system* which, as for it, manages several types of execution environments and the node resources allocation for those environments.

We developed a new active network infrastructure, called ARFANet (Active Rule Framework for Active Network), based on the notion of active rules. In this context, an application can be described as a set of active rules, where each rule is defined as a Event-Condition-Action (ECA) statement. The execution of an application consists, then, to detect the events, to evaluate the conditions and finally to launch the corresponding actions. The ECA form is exclusively event based concept and makes active rules a potential candidantes for network applications which must react to an arising events within the network, such as packet arrival, bottleneck, congestion, link breaking down, etc.

The two most important aspects in active networking are the code distribution and security. We defined the notion of Code Identification and Storage Server (CISS) which allows the storage of the code and its unique identification. With this server we can apply all security techniques and specially on all actors implied in code distribution process such as the developers, the users and the code itself. The security tasks are not any more within the competence of the nodes but delegated to this server, thus the performances of nodes are not further deteriorated. Among those techniques we used cryptophic mechanisms and public key systems for the authentication of all entities concerned by the deployment process including the routers and the codes. The code execution safety validation using PCC method (Proof Code Carrying) has been adapted to ARFANet and takes into account the code providers classification to défine check rules. Symmetric keys or credentials establishment allows to only authorised users to execute the codes on network nodes. The study of the security at the distribution level is not enough to guarantee the safe execution of the active programs in the nodes. Consequently, we enhanced the study of the security of run time program execution in order to avoid all dysfunctions of the nodes or all right violation access. Among those dysfunctions we

note : the occurrence of infinite cycles during the active rule execution, non authorised memory or resources access, excessive packets creation, ect. We defined an adequate methods to secure mainly active rules execution.

Moreover, we observed the necessity to introduce more than one code identification and storage server in large scale networks such as Internet. The distributed code base management must take into account all specifics of active networks. However, there are several ways to manage this base by using a distributed code bases over all the sites or a duplicated code bases on each site. We choose the first technique to avoid the redundancy and the frequent updating. In the context of ARFANet, the active code is composed of several active rules and can be divided to several disjoint modules. This modularity simplifies service composition and avoids unsecured and expensive composition mechanisms. The composition in ARFANet is performed anteriorly of the execution by referring events of a composable modules.

We also concentrated a large part of our efforts on performance evaluation of the nodes in ARFANet infrastructure which led us to model a node using the performance modelling method PEPA (Performance Evaluation Process Algebra), and after that to simulate it with SimJava toolkit.

In the purpose of the study of the possibility to introduce active nodes in the current networks in terms of performance measures, we compared the standard nodes performance, which the unique functionality is to forward the packets flowing through them, with the performance of an ARFANet active node. We studied, in particular, the influence of code packet introduction on the transfer of standard packets. The measures computed are mainly the active node throughput for the standard packets, the loss rates and the node latency, in particular, for this type of packets.

Once the platform implemented, we performed direct measures to validate the analytical and the simulation models results.

Résumé

Le réseau Internet enregistre ces dernières années une mutation au niveau haut de sa hiérarchie, spécialement sur la nature des applications clientes, des protocoles, des services et des flux. Seules les couches basses restent rigides et obéissent fidèlement à des mécanismes datant de la création du réseau Internet. L'enjeu économique et le marché financier qui s'est tissé autour de l'infrastructure actuelle rendent difficilement le passage à l'échelle mondiale de nouveaux protocoles. L'exemple phare est IPv6 qui trouve du mal à s'imposer même si des solutions de substitution pour un passage en douce ont été prévues, comme l'encapsulation d'un paquet IPv6 dans un paquet IPv4 le temps d'adaptation des applications actuelles au nouveau protocole. Les réseaux actifs sont une alternative envisageable qui permet de surpasser rapidement et efficacement ce blocage en prévision de la saturation du réseau à la suite de l'évolution incessante de l'Internet en termes du nombre croissant d'utilisateurs et de services. Dans un réseau actif, les traitements à effectuer dans les différents nœuds peuvent être adaptés aux besoins de l'utilisateur et/ou de l'application elle-même. Les fonctionnalités spécifiques à un utilisateur ou à une application sont chargées dans les nœuds du réseau sous forme de méthodes ou de petits programmes. Lorsqu'un paquet de données de l'utilisateur arrive aux nœuds spécifiés, son en-tête est évaluée et le programme approprié est exécuté sur le contenu du paquet.

Le besoin de développer un modèle de programmation commun comprenant des modèles communs de codage des programmes réseau, des primitives intégrées disponibles dans chaque nœud et l'attribution des ressources du nœud a montré la nécessité d'une architecture commune capable d'accepter différents langages de paquets et d'environnements d'exécution. L'idée d'une architecture basée principalement sur trois couches s'est avérée nécessaire et largement utilisée dans différents travaux dans le domaine des réseaux actifs. Ces couches sont la couche des *applications actives* qui comporte les services d'applications pour les données utilisateurs et la gestion du réseau, la couche des *environnements d'exécution* qui interprète les paquets actifs et exécute les applications actives, et finalement le *système d'exploitation* du nœud qui gère plusieurs types d'environnements d'exécution et l'attribution des ressources du nœud à ces environnements.

Nous avons développé une nouvelle infrastructure de réseaux actifs, appelée ARFANet (Active Rule Framework for Active Networks), basée sur la notion de règles actives. Dans ce contexte, une application peut être décrite comme un ensemble de règles actives, où chaque règle est définie sous la forme d'un tuple Evènement-Condition-Action (ECA). L'exécution de l'application consiste alors à détecter les évènements, évaluer les conditions et lancer les actions correspondantes. La forme ECA est fondée exclusivement sur un concept événementiel et rend les règles actives des candidates potentielles pour les applications réseau qui doivent réagir aux événements survenant au sein du réseau tels l'arrivée d'un paquet, la formation d'un goulot d'étranglement ou d'une congestion, la rupture d'un lien, etc.

Deux aspects très importants dans les réseaux actifs et leur émergence sont la distribution du code actif et la sécurité. Nous avons défini la notion de serveur de code et d'identification (CISS) qui permet le stockage de code et son identification. Avec ce serveur nous pouvons appliquer toutes les techniques de sécurité impliquant tous les acteurs liés au processus de la distribution tels que le développeur, les utilisateurs et le code lui-même. Les tâches de sécurité ne sont plus du ressort des nœuds mais déléguées à ce serveur, ainsi les performances des nœuds ne sont pas altérées. Parmi ces techniques nous avons employé les mécanismes de cryptage et les systèmes à clefs publiques pour l'authentification de toutes les entités concernées par le processus du déploiement y compris le code et les routeurs. Pour la validation de la sûreté des codes la technique PCC (Proof Code

Carrying) a été adaptée dans le contexte d'ARFANet et prend en compte désormais la classification des fournisseurs de codes pour définir les règles de contrôle. L'établissement de clefs symétriques (ou credentials) permet aux seuls utilisateurs autorisés d'exécuter un code sur les nœuds. L'étude de la sécurité au niveau de la distribution ne suffit pas à elle seule pour garantir la bonne exécution des codes actifs dans les nœuds. Par conséquent, nous avons étudié la sécurité des programmes en cours d'exécution pour éviter tout dysfonctionnement des nœuds ou la violation des droits. Parmi ces dysfonctionnements nous citons : la formation des cycles infinis dans l'exécution des règles actives, accès à des zones mémoires ou des ressources non autorisées, la création abusive de paquets, etc. Nous avons, de ce fait, défini des méthodes adéquates pour sécuriser principalement l'exécution des règles actives.

Par ailleurs nous avons constaté la nécessité d'introduire plusieurs serveurs d'identification et de stockage de code dans les réseaux de grande envergure. La gestion de la base de codes distribuée doit prendre en compte toutes les spécificités des réseaux actifs. Cependant, il existe plusieurs manières de gérer cette base, en utilisant des bases de codes réparties sur l'ensemble des sites ou des bases de codes répliquées sur chaque site. Nous avons opté pour la première technique car elle évite la redondance et la gestion des mises à jours fréquentes et convient le plus aux réseaux actifs et en particulier à ARFANet. Dans le cadre de notre infrastructure, le code actif est sous forme de plusieurs règles actives et peut se subdiviser en plusieurs modules disjoints. Cette modularité simplifie énormément la composition de services sans avoir recours à des mécanismes de composition coûteux et peu sécurisés. La composition dans ARFANet se fait en amont de l'exécution par la désignation des événements susceptibles de déclencher les règles de certains modules.

Nous avons également concentré une grande partie de nos efforts à l'évaluation des performances d'un nœud de l'infrastructure ARFANet qui nous a amenés à modéliser ce nœud, dans une première phase, en utilisant la méthode analytique PEPA, puis le simuler, dans une seconde phase, avec l'outil SimJava.

Dans le cadre de l'étude de la possibilité d'introduire les nœuds actifs dans les réseaux actuels en termes de mesures de performance, nous avons comparé les performances d'un nœud standard dont la seule fonctionnalité est de retransmettre les paquets le traversant, avec les performances d'un nœud actif ARFANet. Nous avons étudié, en particulier, l'influence de l'introduction des paquets de code sur le transfert des paquets standards. Les mesures faites sont principalement le débit du nœud actif pour les paquets standards, leurs taux de pertes et la latence du nœud pour ce type de paquets.

Une fois la infrastructure implémentée, nous avons fait des mesures directes pour valider les résultats du modèle analytique et de la simulation.

Table des matières

1 Introduction **1**
1.1 Règles actives & les réseaux actifs . . . 3
1.2 Distribution du code actif . . . 4
1.3 Sécurité dans les réseaux actifs . . . 5
1.4 Analyse des performances d'un routeur actif . . . 6
1.5 Organisation de la thèse . . . 7

2 L'État de l'art sur les réseaux actifs **9**
2.1 Motivations . . . 9
2.1.1 Qualité de service à l'intérieur du réseau . . . 10
2.1.2 Hétérogénéité des flux, des capacités des liens et des équipements . . . 10
2.2 Les types de réseaux actifs . . . 10
2.2.1 L'approche discrète ou nœuds actifs . . . 12
2.2.2 L'approche intégrée ou capsules . . . 12
2.3 Standardisation de l'architecture d'un nœud actif . . . 13
2.3.1 Application active . . . 13
2.3.2 Environnement d'exécution . . . 14
2.3.3 Le système d'exploitation d'un nœud (NodeOS) . . . 14
2.3.4 Le format de paquet . . . 15
2.4 Les projets courants . . . 16
2.4.1 ANTS : approche intégrée . . . 16
2.4.2 Switchware : approche discrète . . . 17
2.4.3 FAIN : approche compositionnelle . . . 18
2.5 Sécurité . . . 18
2.5.1 Les règles de protection . . . 19
2.5.2 Approches avec une plate-forme sécurisée . . . 20
2.5.3 Approche avec un langage restreint . . . 22
2.6 Performance des réseaux actifs . . . 24
2.7 Distribution & Identification du code : . . . 25
2.7.1 La distribution de code . . . 25
2.7.2 Identification de code . . . 26
2.7.3 Distribution & Identification dans les projets actuels . . . 26
2.8 La composition . . . 28
2.9 Conclusion . . . 29

3 Les Règles actives 31
3.1 Introduction . . . 31
3.2 Les composantes d'une règle active . . . 32
3.2.1 La composante événement . . . 33
3.2.2 La composante condition . . . 33
3.2.3 La composante action . . . 34
3.2.4 L'application active . . . 35
3.3 Les sémantiques d'exécution des règles actives . . . 35
3.4 La modélisation à base de règles actives . . . 37
3.5 Les avancées et les domaines de recherche sur les règles actives . . . 39
3.6 Conclusion . . . 40

4 ARFANet : Active Rules Framework for Active Networks 41
4.1 Introduction . . . 41
4.2 Les spécificités des réseaux actifs . . . 42
4.3 ARFANet : Une nouvelle infrastructure des réseaux actifs . . . 43
4.4 Le déploiement des services actifs . . . 46
4.5 Description d'un service d'application . . . 47
4.5.1 La spécification d'un événement . . . 47
4.5.2 La spécification d'une condition . . . 48
4.5.3 La spécification d'une action . . . 48
4.5.4 La spécification d'une règle . . . 49
4.5.5 Exemple de service d'application : Communications de groupe ou Multicast 49
4.5.6 Modularité des services d'applications . . . 52
4.5.7 La spécification des sémantiques d'exécution . . . 56
4.6 Description du moniteur actif . . . 57
4.7 Description de la machine virtuelle . . . 59
4.7.1 Les composants de la machine virtuelle . . . 60
4.8 Le format des paquets . . . 62
4.9 Conclusion . . . 63

5 Une représentation interne pour la Machine Virtuelle 65
5.1 Introduction : . . . 65
5.2 La modélisation d'un nœud ARFANet . . . 66
5.2.1 Le modèle PEPA . . . 68
5.2.2 Les Critères d'évaluation de performance . . . 70
5.2.3 Les résultats numériques . . . 71
5.2.4 L'impact des tailles des files d'attente . . . 75
5.2.5 L'impact de la génération des paquets actifs . . . 77
5.3 Amélioration du Modèle . . . 82
5.3.1 Le modèle PEPA . . . 82
5.3.2 Les résultats numériques . . . 84
5.4 Résultats analytiques *versus* mesures réelles . . . 86
5.4.1 Le modèle de Configuration 1 . . . 86
5.4.2 Le modèle de Configuration 2 . . . 87
5.5 ARFANet *versus* ANTS . . . 88

5.5.1 Variation du taux d'arrivée des paquets . . . 89
5.5.2 Variation de la taille des paquets . . . 90
5.6 Conclusion . . . 92

6 La distribution et le déploiement du code 95
6.1 Introduction . . . 95
6.2 Distribution du code actif . . . 96
6.2.1 Déploiement du code . . . 96
6.2.2 Identification du code . . . 97
6.3 Identification et déploiement de code dans ARFANet . . . 98
6.3.1 L'approche CISS . . . 99
6.3.2 L'approche multi-CISS . . . 100
6.3.3 L'approche mixte . . . 105
6.4 L'analyse de performance . . . 107
6.4.1 Le débit . . . 107
6.4.2 La Latence . . . 108
6.5 Les caractéristiques de l'approche CISS . . . 109
6.6 Conclusion . . . 110

7 Les mécanismes de sécurité 111
7.1 Introduction . . . 111
7.2 Les règles de sécurité dans les réseaux actifs . . . 112
7.3 La sécurité dans la distribution du code . . . 114
7.3.1 La notion de classes de développeurs et d'utilisateurs . . . 114
7.3.2 Sécuriser la phase de publication . . . 115
7.3.3 Sécuriser la phase de déploiement . . . 117
7.3.4 La sûreté du code . . . 119
7.4 La sécurité dans l'exécution du code . . . 120
7.4.1 Utilisation des ressources du Nœud . . . 120
7.4.2 Utilisation des ressources du Réseau . . . 122
7.5 Conclusion . . . 123

8 Des applications pour les Réseaux Actifs 125
8.1 Introduction . . . 125
8.2 Soulagement d'un Serveur Web en cas de surcharge . . . 125
8.3 Application P2P . . . 127
8.4 Conclusion . . . 129

9 Conclusion 131

Annexe 134
A PEPA . . . 135
A.1 La syntaxe de PEPA . . . 135
A.2 Le processus de Markov associé . . . 136
A.3 Résolution numérique . . . 137
A.4 Sémantique opérationelle structurée de PEPA . . . 139
B L'application ARFANet . . . 141

B.1 Le service d'application 141
B.2 Le moniteur actif 143
B.3 La machine virtuelle 144
B.4 Le serveur de code CISS 145
C L'interface Graphique 147
C.1 Construction d'une topologie réseau 147
C.2 Lancement des applications 147
C.3 Construction d'un moniteur actif 147
C.4 Construction d'un service d'application 148

Table des figures

2.1 L'architecture standard d'un nœud actif 14
2.2 Le format des paquets ANEP 16

3.1 Cycle d'exécution d'une règle. 36

4.1 Les composants d'un nœud actif. 44
4.2 Un nœud actif avec une hiérarchie de services d'application. 45
4.3 Un réseau actif avec plusieurs nœuds. 45
4.4 Les phases de déploiement des services active 46
4.5 Le module Multicast 51
4.6 Décomposition en modules 53
4.7 Identification d'un module 55
4.8 Une vue simple d'un moniteur actif. 58
4.9 L'architecture de la machine virtuelle 60

5.1 Réseau de files d'attente modélisant un nœud actif : *Configuration 1* 67
5.2 Impact de K sur le débit 73
5.3 Impact de K sur les taux de pertes 73
5.4 Pertes des paquets en entrée *versus* en sortie 74
5.5 La latence 75
5.6 Les taux de pertes 76
5.7 La latence 77
5.8 Nœud standard *versus* Nœud actif 77
5.9 Le débit des paquets standards en fonction de la génération de paquets actifs 78
5.10 Le taux de pertes des paquets standards en fonction de la génération de paquets actifs 79
5.11 La latence des paquets standards en fonction de la génération de paquets 79
5.12 Nœud standard *versus* nœud actif en fonction de la génération des paquets actifs . . 80
5.13 Proportion des paquets en entrée *versus* en sortie 81
5.14 Réseau de files d'attente modélisant un nœud actif : *Configuration 2* 82
5.15 Le débit 84
5.16 Les taux de pertes 85
5.17 La latence 85
5.18 ARFANet *versus* PEPA 87
5.19 ARFANet *versus* PEPA 88
5.20 ARFANet versus ANTS 89
5.21 ARFANet versus ANTS 90
5.22 ARFANet versus nœud standard 91

5.23 ARFANet versus Nœud standard en fonction de la taille des paquets 92

6.1 Publication et récupération du code actif 99
6.2 Répartition des *CISS* 101
6.3 Les phases de la méthode mixte 106
6.4 Comparaison des trois approches 108

7.1 Hiérarchisation des utilisateurs et des développeurs 114
7.2 les phases de distribution de code sécurisée 116
7.3 Utilisation de la fonction de Hashage 118

1 Chaîne de Markov 137
2 La sémantique opérationnelle de PEPA 139
3 Création d'une topologie 148
4 Lancement des applications 149
5 Construction d'un moniteur actif 150
6 Traitement des événements 151
7 Création des conditions et des actions 151
8 Création des règles 152

Liste des tableaux

3.1 Table Pilote . . . 32
3.2 Liste globale des paramètres sémantiques pour l'exécution des règles . . . 38

4.1 Exemple de la spécification d'un type d'événement. . . . 48
4.2 Spécification des types d'événement du module Multicast. . . . 52
4.3 Application du Multicast. . . . 54
4.4 Multicast : module 1. . . . 54
4.5 Multicast : module 2. . . . 55
4.6 Multicast : module 3. . . . 55
4.7 Exemple de module. . . . 57

5.1 Valeurs des paramètres . . . 72

8.1 Le module E-commerce *rush-hour filter* . . . 127
8.2 Le module *P2P covering* . . . 129
8.3 Le module *P2P research* . . . 129

Chapitre 1

Introduction

L'évolution des outils informatiques individuels a entraîné une modification des méthodes de gestion et de travail dans les entreprises. Le travail individuel est devenu collectif. Les besoins d'information et de communication dans les groupes de travail se sont alors développés de façon importantes. Le micro-ordinateur et le réseau local (LAN) sont au centre de ces changements.

Un réseau local est un ensemble de moyens autonomes de calcul et de traitement (postes et stations de travail ou autres) reliés entre eux pour échanger des informations comme la messagerie, la rédaction de documents en commun, la diffusion et la circulation de documents (workflow). Ceci afin de partager des ressources physiques et logicielles, telles que les imprimantes, les espaces disque, les agendas, les bases de données, les fichiers, etc. Le réseau local doit aussi fournir aux utilisateurs un ensemble d'applications permettant l'accès et l'utilisation des ressources mises en commun.

L'interconnexion des réseaux locaux est venue de la nécessité de mettre en relation différentes machines appartenant à des réseaux différents indépendamment de la distance les séparant et des protocoles utilisés. Les routeurs assurent la communication entre des réseaux distants.

Les routeurs appelés systèmes fermés, contrairement aux systèmes finaux comme les ordinateurs personnels ou les stations de travail dits systèmes ouverts, ont un fonctionnement prédéfini par les constructeurs. Ces derniers suivent les comités de standardisation des protocoles et des fonctionnalités pour changer ou introduire de nouveaux services [96].

Le traitement appliqué par les routeurs dans les réseaux à commutation de paquets traditionnels, tel que Internet, est extrêmement limité. Seules des modifications simples sur l'entête des paquets sont actuellement tolérées. Ces équipements ont des fonctionnalités fixées et limitées à la transmission des paquets.

L'évolution grandissante des applications réseaux, qui sont devenues de plus en plus sophistiquées, a rendu les équipements actuels obsolètes et non adaptés. Parmi ces applications nous comptons la vidéo conférence, la voix sur IP, la télévision en ligne, les jeux en réseaux et l'interconnexion des réseaux mobiles via Internet. Ces applications nécessitent une qualité de service qui ne peut être offerte par les réseaux d'aujourd'hui. La rigidité du réseau actuel entrave son évolution vers d'autres réseaux où plusieurs types d'application et de services peuvent coexister.

Le déploiement de nouveaux services sur IP est très lent. Un exemple très répandu de services dont le déploiement a pris beaucoup de temps (cinq ans) est RSVP (Resource ReSerVation Protocol). Le rôle de ce protocole est d'établir une certaine qualité de service pour différents types de flux et en particulier la diffusion du flux [44]. De plus, un protocole donné ne peut être performant dans tous les domaines (sans fil, satellite, temps réel,...). Le protocole IP n'est pas performant dans des domaines comme les réseaux mobiles et les applications temps réel.

En plus de la lenteur de déploiement de nouveaux services due à la standardisation et la normalisation, les inconvénients qui caractérisent les réseaux de télécommunication, sont comme suit :

- une sous-utilisation des informations liées à la nature et à la sémantique des flux (telles que les topologies logiques des flux qui nécessitent un traitement optimal comme dans le cas des flux multicast),
- la difficulté dans la conception, le développement et le déploiement de composants ouverts (codes portables et mobiles sur des architectures hétérogènes et distantes),
- une sous-exploitation des capacités des équipements du réseau comme les routeurs,
- la non adaptation des routeurs à la nature des flux, tels que les flux vidéos, la voix et les données d'application temps réel. Les routeurs considèrent tous les paquets de manière identique et leur appliquent le même routage,
- la difficulté d'introduire une qualité de service. La qualité de service induit une différenciation dans le traitement des flux. Une conséquence du point précédent est la difficulté pour les routeurs actuels de faire une distinction les flux.

L'ouverture des équipements de transmission et de routage pour qu'ils intègrent de nouveaux protocoles et services est une solution envisageable qui permettra de passer rapidement à une nouvelle génération de réseaux. Cette solution permettra d'améliorer le routage et rendre les réseaux capables d'intégrer de nouveaux services adaptés à des domaines spécifiques [28]. Les réseaux actifs sont conçus pour permettre une certaine dynamique dans le traitement des flux et une certaine personnalisation du mode d'acheminement des paquets.

Tandis que le réseau classique repose sur la philosophie store-and-forward c'est-à-dire récupérer et renvoyer, la philosophie d'un réseau actif est store-compute-forward c'est-à-dire récupérer, exécuter puis renvoyer [32]. Les routeurs des réseaux actifs exécutent des traitements supplémentaires sur les paquets les traversant. Ces traitements spécifiques permettent de passer d'un réseau fortement couplé, où le hardware et le software sont très liés, à un réseau virtualisé, où le hardware et le software sont découplés, et ceci afin de permettre à de nouveaux services de se déployer rapidement et aisément [85].

Les traitements à effectuer dans les différents nœuds peuvent être adaptés aux besoins de l'utilisateur et/ou de l'application elle-même. Ils peuvent être intégrés aux nœuds du réseau (*commutateurs programmables*) ou dans les paquets les traversant (*capsules*), sous forme de méthodes ou de petits programmes. Dans la première approche, l'utilisateur envoie d'abord ses paquets de programmes dans certains nœuds du réseau. Lorsqu'un paquet de données de l'utilisateur arrive aux nœuds spécifiés, son en-tête est évaluée et le programme approprié est exécuté sur le contenu du paquet. Dans la deuxième approche, chaque paquet ou capsule se compose d'un petit programme qui est transmis dans le flux de données (in-band) et exécuté dans chaque nœud traversé par le paquet.

Bien que ces approches semblent différentes, le concept de réseau actif utilisé derrière est fondamentalement le même. Dans les deux approches, le contenu des paquets est extrait et expédié à un environnement pour être exécuté immédiatement (approche des capsules) ou au bon moment (approche commutateurs programmables). Ceci suggère que les mécanismes et les primitives impliquées dans ces opérations sont non seulement indépendants de l'approche utilisée, mais également de l'application elle-même.

Le besoin de développer un modèle de programmation commun, comprenant des modèles communs de codage des programmes réseau, des primitives intégrées disponibles dans chaque nœud, et l'attribution des ressources du nœud a montré la nécessité d'une architecture commune capable d'accepter différents langages de paquets et d'environnements d'exécution. L'idée d'une architecture basée principalement sur trois couches a été suggérée dans [13] et depuis utilisée dans différents

travaux [50][82][74]. Ces couches sont la couche des *applications actives* qui comporte les services d'applications pour les données utilisateurs et la gestion du réseau, la couche des *environnements d'exécution* qui interprète les paquets actifs et exécute les applications actives, et finalement le *système d'exploitation* du nœud (NodeOS) qui gère plusieurs types d'environnements d'exécution et l'attribution des ressources du nœud à ces environnements.

Les réseaux actifs tentent d'apporter de nouvelles fonctionnalités aux réseaux telles que :

- une évolution rapide des services et protocoles : la mise à jour et la révocation d'un protocole ou d'un service actif doit être plus rapide que celle des protocoles actuellement déployés.
- une simplicité dans la conception d'objets mobiles qui trouveront une utilisation dans les télécommunications et dans d'autres domaines comme la programmation distribuée,
- une utilisation optimale des ressources des routeurs lors de traitements de tâches communes,
- réduire la difficulté d'introduire une qualité de service de bout en bout sur un flux de données,
- adapter le flux aux capacités des équipements finaux et des liens surtout pour les équipements mobiles qui sont raccordés au réseau global.

Les réseaux actifs s'avèrent, ainsi, une solution prometteuse pour l'application de nouveaux protocoles dans des délais raisonnables, en rendant le réseau plus intelligent et plus flexible.

L'objectif essentiel de cette thèse est le développement d'une architecture globale d'un réseau actif. Nous définissons une architecture du nœud actif implémentant les fonctionnalités des règles actives, appelée ARFANet (Active Rule Framework for Active Networks). De plus, nous proposons de construire une politique de publication et de déploiement uniformes du code en utilisant la notion de serveur de codes et en mettant en exergue tous les acteurs potentiels. Dans cette même phase, nous développons des mécanismes de sécurité intégrés aux différentes étapes du cycle de vie d'une application active à savoir : la publication, le déploiement et l'exécution. La concrétisation d'une telle architecture nous pousse à modéliser le système et à étudier son comportement et ses performances afin d'optimiser son fonctionnement et d'anticiper tout effet négatif sur les flux et les services qui ne nécessitent pas une gestion active.

Dans ce qui suit, nous présentons brièvement ces différentes parties.

1.1 Règles actives & les réseaux actifs

Les règles actives sont une extension des règles de production définies dans les systèmes experts et les systèmes de l'Intelligence Artificielle [20]. Elles ont été introduites dans les systèmes de gestion des bases de données (SGBD) pour améliorer les déclencheurs (triggers). Ces derniers exécutent une procédure de réparation ou d'alerte lorsqu'un état incohérent de la base se produit (une donnée non conforme), principalement par une mise à jour. Une règle active doit respecter un format bien défini :

On <nom de l'événement> *if* <expression de la condition> *then* <invocation de l'action>

Dans ce contexte, les événements sont tout signal discret détecté comme l'arrivée d'un paquet ou le changement d'état du système. Les conditions représentent des formules logiques dont les opérandes se réfèrent à des variables locales à la règle ou globales telles que l'état du système ou les données véhiculées ou stockées. Quant aux actions, elles sont les opérations autorisées sur les données et le système sous-jacent. Ainsi une application est définie comme un ensemble de règles

actives décrivant un comportement cohérent et significatif. La prise en charge des règles nécessite un système actif compétant et comportant des composantes spécialisées qui gèrent chacune une partie de la règle.

L'arrivée d'un paquet, la formation d'un goulot d'étranglement, la congestion ou la rupture d'un lien, sont tous des événements qui peuvent survenir dans un réseau. Cette nature événementielle du réseau implique des applications qui doivent réagir aux événements qui peuvent y survenir. Le modèle ECA (Evénement - Condition - Action) sur lequel sont basées les règle actives apparaît comme un modèle idéal pour formuler les applications à déployer dans un réseau actif.

Les applications actives actuelles sont des programmes compacts, compilés dans des langages dédiés aux réseaux actifs comme PLAN [44] ou généralistes comme JAVA. Cette forme peut-être performante dans le cas de petits codes, mais dans le cas de codes volumineux, il est plus intéressant d'avoir plusieurs modules de petites tailles. Les règles actives, par leur forme, offrent cette modularité. De plus, la décomposition d'une application en un ensemble de règles (modules) partageant le même *événement* permet une exécution optimale. Le déclenchement d'un événement permet le chargement et l'exécution du module concerné, les autres modules n'étant ni chargés, ni exécutés.

Les spécificités des réseaux actifs nous poussent à une adaptation intelligente des règles actives. L'implémentation des règles actives dans un réseau nécessite une architecture d'un nœud qui soit capable de gérer toutes les composantes d'une règle (*moniteur de règles actives*). Cette architecture doit pouvoir détecter les événements et déclencher toutes les règles concernées. Pour chaque règle déclenchée, tester si la condition est vérifiée. Les règles dont la condition est valide, le moniteur exécute leurs actions.

Dans un système à base de règles, l'exécution d'une règle peut engendrer la détection d'autres événements déclenchant à leur tour d'autres règles. Cette récursion dans le déclenchement des règles offre une grande flexibilité dans l'écriture des programmes et permet une large panoplie d'applications. Ainsi à l'image des règles, les composantes du moniteur d'exécution sont fortement connectées et interagissent entre elles. D'autre part, la prise en compte de plusieurs types d'applications actives en même temps nécessite une gestion parallèle des moniteurs.

Dans ce contexte, nous proposons une nouvelle infrastructure de nœuds actifs ARFANet basée sur les règles actives. Nous définissons une architecture à couches similaire à celle développée dans [13]. La première couche est liée aux services d'application qui regroupe les règles d'une même application. La seconde couche contient les moniteurs actifs qui gèrent les règles et les exécutent. Quant à la couche de base, c'est une machine virtuelle qui met à la disposition des moniteurs les ressources nécessaires disponibles dans les nœuds sous forme d'abstractions logicielles. La caractéristique fondamentale de cette architecture à couches est qu'elle met en exergue les propriétés de flexibilité et de modularité des règles actives indispensables aux réseaux actifs.

1.2 Distribution du code actif

La mise en place d'un réseau actif se divise en deux phases principales, la distribution et l'exécution du code. La première phase consiste à injecter du code et à l'acheminer jusqu'aux nœuds. La seconde phase se résume au chargement et à l'exécution de ce code orienté vers les environnements d'exécution dédiés. Même si tout l'effort de la recherche a principalement touché la deuxième phase par la définition de plusieurs environnements d'exécution qui acceptent différents langages et l'élaboration d'une architecture standard d'un nœud actif, cela ne diminue en rien l'importance de la première phase qui détermine comment le code arrive aux différents nœuds.

Dans ce contexte, nous nous intéressons à certains problèmes tels que l'universalité et l'unicité de l'identifiant d'une application, ainsi que l'indépendance du code vis-à-vis de son développeur pour permettre une réutilisabilité et un partage large de ce code. L'un des aspects déterminants pour un rapatriement uniforme du code par tous les nœuds est la source du code et sa localisation. Il existe plusieurs techniques en ce qui concerne la nature de la source du code. Ce point d'accès peut être l'utilisateur qui fournit en même temps le code et les données dans un seul flux ou séparément en les invoquant d'un ou de plusieurs serveurs qui sauvegardent le code et le mettent à disposition des nœuds.

Le choix d'une technique détermine la politique de publication et de déploiement des applications actives et les mécanismes appropriés. La politique où le code est dans le même flux que les données soulève quelques interrogations comme la grande taille des paquets, la réutilisabilité et le partage du code ou encore la sécurité. Tandis que la seconde politique (envoi du code et des données dans le même flux) nécessite de répondre à des questions telles que le nombre de serveurs de code et leur localisation dans le réseau. Il est impératif aussi de définir des mécanismes de contrôle de la base de codes dans le cas de plusieurs serveurs de codes distribués. L'unicité du code peut être mise en cause si deux serveurs tentent d'attribuer le même identifiant à des applications différentes. Un autre aspect important dans les réseaux actifs et plus généralement dans la programmation évolutive est la composition des applications. Si les modules sont distribués sur plusieurs serveurs, la composition d'une application doit prendre en compte cette répartition et offrir les mécanismes adéquats pour ce genre d'application.

La disponibilité des applications doit être à l'échelle de tout le réseau et pour tous les utilisateurs. A travers certains points d'accès, l'utilisateur peut prendre connaissance des différents services mis à disposition et notamment les composer pour créer de nouveaux services plus complexes. De même, les développeurs peuvent considérer la base de codes sur les serveurs comme une bibliothèque de modules qu'ils peuvent référencer dans leurs propres applications.

Dans cette thèse, nous proposons une architecture globale de distribution de codes qui regroupe toutes les phases, de la publication sur des serveurs au rapatriement du code par les nœuds. Nous proposons également des mécanismes de communication pour le maintien d'une base de codes cohérente, et étudions les emplacements potentiels des serveurs de codes dans le réseau. Par ailleurs, notre but principal est de définir des mécanismes d'identification unique qui respectent les conditions d'universalité sur tout le réseau.

1.3 Sécurité dans les réseaux actifs

Les réseaux actuels sont sujets à des attaques fréquentes par des utilisateurs ou des systèmes malveillants à cause des failles de sécurité. Parmi ces attaques, nous pouvons citer l'analyse du trafic, l'interception des données, l'accès non autorisé, l'usurpation d'identité ou l'interception et le contournement des mots de passe. Le déni de service reste l'une des attaques les plus dangereuses. Il est considéré comme un ensemble d'attaques provoquant la destruction et la dégradation des processus et du stockage, l'arrêt des systèmes, et l'appropriation et la falsification des adresses IP (IP spoofing) pour intercepter les données ou l'identité d'un tiers [49].

L'ouverture du réseau pour que des codes externes puissent être exécutés aux niveaux des routeurs augmente considérablement les failles du réseau en termes de sécurité. Parmi les failles qui peuvent survenir, nous pouvons citer la publication d'un code malveillant, l'usurpation de l'identité d'un développeur autorisé à publier un code malveillant, l'usurpation de l'identité d'un utilisateur

autorisé à exécuter un code actif déjà publié sur les propres données de l'intrus, la consommation abusive des ressources des routeurs (mémoire, temps d'exécution, bande passante, ...) jusqu'à altérer leur fonctionnement et par conséquent perturber tout le réseau et ses utilisateurs.

Sécuriser un réseau actif consiste d'abord à dénombrer les failles à tous les niveaux et à les renforcer par des mécanismes de sécurité. Il existe actuellement des systèmes à base de cryptographie utilisés par des infrastructures actives sécurisées. Cependant, des mécanismes de renforcement adaptés aux spécificités des réseaux actifs doivent être définis. Il est, par exemple, impératif d'authentifier et de certifier toutes les entités autorisées à publier ou à déployer un code. De plus, ces mécanismes doivent être les plus performants étant donnés l'enjeu et les conséquences sur le réseau et ses équipements.

Pour l'aspect sécurité, nous proposons une architecture qui accompagne les phases de distribution et d'exécution des codes dès leur conception et mise en place. Elle comprend des mécanismes d'identification et d'authentification des développeurs basés sur les techniques de cryptographie. Durant la phase de déploiement du code, elle authentifie les utilisateurs et définit un mécanisme d'autorisation de l'exécution par les nœuds pour les utilisateurs à base de clefs symétriques. Ce mécanisme épargne aux nœuds une connexion supplémentaire vers un serveur d'autorisation ou la détention d'une liste des utilisateurs autorisés. Pour l'exécution sûre des codes, la forme triangulaire des règles actives nous donne des moyens de contrôle adaptés sur les différentes parties.

1.4 Analyse des performances d'un routeur actif

La couche de base d'un routeur actif, sous-jacente à l'interface réseau, joue le même rôle qu'un routeur classique en retransmettant les paquets traditionnels. De plus, elle gère les environnements actifs et leur oriente les paquets actifs correspondants. Les performances de la couche de base doivent être optimales pour perturber le moins possible le routage des paquets traditionnels et permettre des exécutions optimisées des applications actives. Les perturbations éventuelles peuvent toucher la perte des paquets standards, leur latence, ainsi que le débit du routeur pour ce genre de paquets. La seule présence des paquets actifs peut engendrer des perturbations ; le traitement additionnel par les couches actives supérieures peut altérer les performances des routeurs.

L'étude des performances d'un nœud actif, et en particulier de la couche de base, permet d'analyser le comportement de chaque composante (file d'attente d'entrée, file d'attente de sortie, file d'attente de paquets actifs, ...) et les interactions entre ces composantes. Nous étudions plusieurs configurations de la couche basse et comparons leur performances. Nous nous intéressons également au comportement du nœud actif vis-à-vis du changement de la taille des files, la variation de la génération des paquets provenant de l'exécution des applications actives et la variation du taux d'arrivée des paquets. Pour cela, nous utilisons différentes techniques d'analyse des performances.

Modélisation analytique : nous modélisons analytiquement un nœud actif à base de réseau de files d'attente. Nous transcodons, ensuite, ce modèle vers un modèle de l'algèbre des processus en utilisant le formalisme PEPA (Performance Evaluation Process Algebra) [46]. Ce dernier permet d'étudier de manière abstraite le comportement et les performances des nœuds, tout en permettant une formalisation très simple des modèles à base de composantes.

Simulation : nous utilisons l'outil SimJava [25], qui représente chaque composante du système sous forme d'un objet. SimJava est basé sur une programmation orientée objet ce qui permet une

représentation avantageuse des systèmes complexes.

Mesures sur la plate-forme : des mesures effectuées directement sur ARFANet sont comparées aux résultats obtenus avec le modèle PEPA et la simulation.

1.5 Organisation de la thèse

Cette thèse est organisée comme suit. Nous présentons dans le prochain chapitre les réseaux actifs, leurs objectifs et leurs apports. Nous décrivons les différentes couches qui constituent un nœud actif et leurs interactions. La transmission des données actives et des codes est encapsulée par des protocoles et des formats de paquets spécifiques et standardisés. Ainsi, nous invoquons dans ce même chapitre les formats de paquets et les mécanismes actuels de déploiement de codes. Par ailleurs, nous survolons les principaux projets élaborés dans le domaine et les applications proposées dans la littérature.

Le troisième chapitre sera consacré à la présentation des règles actives et des systèmes qui les gèrent.

Dans le chapitre quatre, nous développons une nouvelle architecture d'un nœud actif ARFANet basée sur les règles actives. Nous présentons, par la suite, toutes les couches qui composent cette architecture et le format de paquet utilisé. Nous montrons, également, comment, avec la modularité que nous offrent les règles actives nous pouvons construire des applications composables. Nous illustrons notre technique de composition par un exemple d'application qui est le Multicast.

Dans le chapitre cinq, nous modélisons le nœud actif pour étudier l'attitude de ses composants vis-à-vis des différents types de paquets et dans plusieurs cas de figures. Cette étude nous permettra d'examiner le fonctionnement global des nœuds et le choix des paramètres afin de les améliorer.

Le chapitre six expose les techniques actuelles de distribution des programmes actifs au sein du réseau en termes d'identification, d'accessibilité, de réutilisabilité et de partage du code. Nous présentons également les solutions que nous préconisons pour un bon schéma de distribution de code.

La sécurité, qui est un point crucial dans ce type de réseaux, est abordée dans le chapitre sept. Nous exposons les mécanismes de sécurité proposés pour fiabiliser l'acheminement du code et son exécution et protéger les systèmes réseaux et les données utilisateurs des utilisations frauduleuses de l'activation du réseau.

Le chapitre huit montre l'intérêt des réseaux actifs à travers la formalisation par des règles ECA de deux applications originales. La première application consiste à soulager un serveur Web dédié au commerce électronique en période de pointe. Quant à la seconde, elle permet d'améliorer la détection des pairs voisins dans un réseau P2P (pair à pair) et la propagation des requêtes.

La conclusion générale et les perspectives futures pour ce travail sont présentées dans le chapitre neuf.

Chapitre 2

L'État de l'art sur les réseaux actifs

Étant donnée la multitude de projets dans le domaine des réseaux actifs, un chapitre ne saura suffire à présenter chaque projet et discuter leurs avantages et leurs inconvénients. Nous avons fait le choix de nous focaliser sur les aspects que nous aborderons tout le long de cette thèse. Nos travaux ont mis l'accent sur les domaines qui n'ont pas eu ou peu d'engouement de la part de la communauté des réseaux actifs tels que la sécurité, la composition des services et aussi la distribution et le déploiement du code. Dans ce chapitre nous présentons les réseaux actifs de manière générale, mais nous nous attarderons sur la présentation des aspects précédemment cités et les projets qui s'y sont intéressés.

La section suivante porte sur les motivations des réseaux actifs et leurs applications potentielles dans le domaine des télécommunications. La section 2.2 présente un historique des réseaux actifs et décrit les principales approches qui dominent ce domaine. L'architecture d'un nœud est présentée à la section 2.3 ainsi que les formats de paquets. Nous préférons présenter dans la section 2.4 un projet type par approche. La sécurité dans les réseaux actifs est discutée dans la section 2.5. Les approches développées par les projets actuels pour que leurs plates-formes soient performantes sont présentées dans la section 2.6. Notre infrastructure ouvre le champ à la composition d'applications actives, ce qui nous a poussé à nous intéresser plus particulièrement à ce point. La composition a dès le début de l'évolution des réseaux actifs attiré l'attention de la communauté et abouti à des projets que nous présentons dans la section 2.8. Dans la section 2.9, nous conclurons ce chapitre avec nos remarques et commentaires.

2.1 Motivations

Très nombreuses sont les applications des réseaux actifs pour l'amélioration du service Internet pour certains types de flux, la gestion du réseau, etc. Les applications répandues dans la littérature sont divisées en catégories. La première catégorie concerne essentiellement les applications de gestion de réseau, comme le contrôle actif de la congestion où le routeur aura pour rôle la détection et l'élimination de la congestion ou encore la communication en groupe. Dans le dernier point le routeur aura un rôle dans la gestion du sous-arbre du multicast et élimination de l'implosion des demandes de retransmissions (NACK). La seconde catégorie concerne plus les applications Web à caractère commercial où les informations sensibles échangées doivent respecter des délais raisonnablement petits, et une mise à jour fréquente et spontanée même si le serveur est surchargé. La cotation en direct et les enchères sont des exemples typiques de ce genre d'applications. D'autres domaines

peuvent tirer parti des réseaux actifs, tels que les réseaux mobiles pour l'adaptation du flux à la dégradation des liens ou les réseaux ad hoc pour permettre un recouvrement et une configuration dynamique des chemins. Le développement récent des réseaux de capteurs et la collaboration étroite avec les réseaux mobiles offrent des perspectives nouvelles dans beaucoup de domaines (météo, sismologie, géographie, médecine pour des patients en déplacement, ...). Les réseaux actifs peuvent aussi contribuer dans ce grand projet par des traitements adéquats sur les données urgentes et sensibles avant leur arrivée au serveur. Dans ce cas des mécanismes de mixage et d'agrégation qui améliorent considérablement ces applications seront exécutés par les routeurs proches des capteurs.

2.1.1 Qualité de service à l'intérieur du réseau

Le réseau IP actuel n'offre, essentiellement, qu'un seul type de services : *best effort*. La simplicité du protocole rend difficile la distinction entre plusieurs types de flux, et par conséquent des traitements adéquats. Les flux de vidéo et de téléphonie requièrent un caractère temps réel. Privilégier ce genre de flux au cours du routage permet d'améliorer la latence de bout en bout. Nous pensons qu'un traitement actif pour ce type d'application consiste à privilégier le chemin le plus court lors du routage. Comme le routeur n'a qu'une vue locale, il ne peut pas prédire le meilleur chemin à prendre. Si le système explore, en amont de l'envoi des paquets de la voix par exemple, les chemins potentiels entre la destination et la source, pourra choisir le plus court et le transcrire dans les paquets actifs. Le routeur actif, à la rencontre des flux de l'application, ne doit pas leur appliquer le routage classique pour choisir le prochain routeur, mais ce dernier sera indiqué par le paquet.

2.1.2 Hétérogénéité des flux, des capacités des liens et des équipements

Les différents équipements (PDA, téléphone portable, ...) et liens (hertzien, sattelitaire, ADSL, ...) ont des capacités très variées. Considérons deux utilisateurs dans des situations différentes et équipés avec des moyens différents. Le premier utilisateur en mouvement est équipé d'un PDA ou téléphone cellulaire qui a une capacité de traitement très limité et une bande passante réduite. Le second est un utilisateur derrière son ordinateur personnel relié à Internet par un câble ADSL. Ces deux utilisateurs ne peuvent pas avoir le même protocole appliqué à leur flux. Les mécanismes de traitements d'erreurs, de la perte des paquets et du changement incessant des capacités des liaisons doivent varier, selon les caractéristiques du lien et des équipements de l'utilisateur final, afin de garder une bonne qualité service. Un protocole comme IP qui tente de satisfaire un large spectre d'utilisateurs ne peut, en aucun cas, être performant pour tout type de flux. Les réseaux actifs viennent expressément pour combler ce genre de défaut en adaptant le protocole aux caractéristiques du flux, à la demande du client ou de manière autonome.

2.2 Les types de réseaux actifs

La notion de réseau actif est née des réseaux programmables promulgués par l'interface programmable P1520. Ces réseaux sont des réseaux de transmission de données ouverts et extensibles disposant d'une infrastructure programmable dédiée à l'intégration et la mise en œuvre rapide de nouveaux services sur l'ensemble de ses composants [28].

P1520, modèle de référence pour les réseaux programmables, définit trois couches programmables. La première couche en relation directe avec les applications utilisateur est la couche *V*

Interface qui fournit une librairie riche en services à valeur ajoutée. La deuxième couche *U interface* permet d'établir des connexions point à point ou point à multi-points. *L Interface* fournit une librairie de services qui gère l'accès direct aux ressources du nœud et leur gestion comme les tables de routage. La couche de base est *CCM* (Connection Control and Managment) qui n'est pas un interface programmable, mais un ensemble de protocoles de bas niveau. Ce genre d'interfaces programmables engendre une certaine dynamique dans les réseaux.

P1520 a ouvert la voie à d'autres familles de réseaux programmables. Il existe actuellement deux grandes familles d'approches :

1. La signalisation ouverte (OPENSIG) [42] : c'est une approche centrée autour de l'ouverture des réseaux via la définition d'interfaces de programmation offrant l'accès au plan de signalisation des équipements. Dans cette approche, le plan de signalisation est remplacé par un environnement logiciel distribué acceptant de nouveaux protocoles et services.
2. Le réseau actif : cette approche est plus ouverte que l'approche précédente (réseau programmable pur) car l'ouverture touche l'ensemble des plans (signalisation, supervision et données).

Le concept des réseaux actifs a vu le jour en 1994 par les travaux de DARPA (Defence Advanced Research Projects Agency). Les réseaux actifs sont des réseaux de commutation de paquets, où la partie données du paquet peut contenir des fragments de code exécutés dans les nœuds intermédiaires. Après 1998, plusieurs groupes de travail ont été constitués et ont défini une architecture standard d'un nœud actif et un format de paquet. Les réseaux actifs ont depuis connu une deuxième phase qui a donné un nouveau souffle à leur évolution.

Par opposition à un réseau classique qui repose sur la philosophie Store-and-Forward c'est-à-dire récupérer et renvoyer, la philosophie d'un réseau actif est Store-Compute-Forward c'est-à-dire récupérer, exécuter puis renvoyer [32]. Pour ce faire, tous ou une partie des composants du réseau, dans les différents plans (signalisation, supervision, données), sont programmables dynamiquement par d'autres entités (opérateur, fournisseurs de services, applications et même les usagers).

Le but des réseaux actifs est de réduire la rigidité des équipements réseaux actuels afin qu'ils puissent suivre l'évolution rapide des applications. Par les progrès énormes faits au niveau de la vitesse des processeurs et la capacité des mémoires vives, il n'est plus justifiable d'implanter seulement de simples fonctions fixes de routage dans les routeurs. Ces derniers sont d'autant plus capables d'effectuer plusieurs tâches en parallèle sans perturber les fonctions de base. Les réseaux actifs viennent en tant que solution incontournable pour le passage vers les réseaux de la nouvelle génération tant attendue. De notre avis, les événements socio-économiques, qui ont survenu au début de ce siècle, ont aussi atténué l'enthousiasme et l'appui des opérateurs des télécommunications et les fournisseurs de services pour l'émergence des réseaux actifs. Ce qui a bloqué le processus de standardisation commencé depuis 1998 par la communauté des réseaux actifs.

Les réseaux actifs permettent, entre autre, de :

- éliminer la phase de standardisation, car les protocoles sont directement déployés sur les routeurs,
- étendre et modifier l'infrastructure actuelle du réseau IP,
- augmenter la flexibilité et la spécialisation des réseaux,
- augmenter les performances des réseaux en éliminant les opérations redondantes au niveau des couches protocolaires (comme les opérations de contrôle des erreurs, de cryptage,...).
- adapter le flux aux capacités des équipements finaux et des liens surtout pour les équipements mobiles qui sont raccordés au réseau global,
- avoir n grand choix de protocoles de communication sur le routeur,

- créer d'autres types d'application pour la nouvelle génération de réseaux,
- profiter de l'évolution rapide des capacités de traitement et de stockage caractérisant les équipements informatiques. Avec cette évolution, il est tout à fait possible dans le futur proche d'appliquer des traitements par les routeurs sans affecter leurs performances.

Les recherches avancées ont divisé les réseaux actifs en deux sous catégories. La première catégorie est l'*approche discrète* connue sous le nom de nœuds actifs. L'autre approche plus dynamique, est appelée *approche intégrée* ou paquets actifs. Les paragraphes suivants définissent ces approches.

2.2.1 L'approche discrète ou nœuds actifs

Dans les nœuds actifs, les services sont déployés dynamiquement sur les nœuds du réseau et téléchargés à partir d'un serveur prédéfini (un petit rapprochement avec les réseaux programmables) [86].

Les nœuds programmables exécutent des programmes en les injectant (téléchargeant) hors bande. En d'autres termes, les programmes ne sont pas inclus dans les paquets de données, donc le téléchargement des programmes dans les routeurs est indépendant du traitement et de l'acheminement des messages. Une phase de sélection des routeurs et de pré-programmation est nécessaire. La sélection peut porter sur plusieurs critères (appartenance au chemin des données, compatibilité du matériel et du logiciel avec le code, disponibilité de certains dispositifs, ...).

Le concept des nœuds programmables permet de garder et de maintenir le format actuel des paquets et le protocole qui permet de les transmettre (en occurrence IP). Lors de l'arrivée de chaque paquet à un routeur actif, son entête est examinée et le programme approprié est identifié et chargé pour traiter le paquet selon le comportement souhaité par l'application cliente. Ainsi le code est appliqué sur les paquets standards qui vérifient certaines caractéristiques sur leur type (UDP, TCP, RTP, ...) ou sur leur entête (adresse source ou destination). Un filtre est nécessaire à l'acheminement des paquets jusqu'à l'application à installer au niveau bas du routeur.

Le téléchargement de nouveaux programmes (protocoles) sur le routeur nécessite la vérification du code et l'identification de son propriétaire.

2.2.2 L'approche intégrée ou capsules

La notion de capsules ou *paquets actifs* repose sur les paquets actifs. Ces capsules contiennent de petits programmes, de la taille d'un paquet, transmis en bande comme tout paquet de données et exécutés sur chaque nœud du chemin parcouru par les paquets de la même application cliente. En théorie les données peuvent accompagner les paquets de code, mais la taille des paquets résultants empêchent cette éventualité [86].

Le code à appliquer sur les données est chargé sur le même flux et sa désignation est explicite par les paquets de données. L'avantage majeur issu de cette caractéristique est sa flexibilité et son adaptabilité vis-à-vis du chemin des paquets qui, de plus, est dynamique dans le réseau IP. La pré-programmation de certains nœuds n'est pas nécessaire s'ils ne sont traversés par aucun paquet. La pré-programmation, principe de base de la technique précédente, peut s'avérer une solution adéquate dans les réseaux à commutation de circuits (ATM) quand les commutateurs sont prédéfinis.

En termes d'exécution des programmes actifs, ces deux approches sont presque identiques. La différence principale entre ces deux tendances réside dans l'acheminement du code et sa correspondance avec les données à traiter. Dans la première technique, c'est le service actif qui reconnaît les

paquets, tandis que dans la seconde ceux sont les données qui doivent désigner le service adéquat.

Pour des raisons de portabilité et de sécurité, le code exécuté dans les deux approches doit être interprété. D'où l'utilisation, au niveau des nœuds actifs, d'une machine virtuelle qui interprète le code et restreint l'espace d'adressage.

Donner aux routeurs ce genre de fonctionnalités nécessite la construction d'un système d'exploitation capable de charger les programmes et gérer des ressources partagées qui leurs seront allouées.

Dans ce qui suit nous présentons les composantes permettant l'activation d'un nœud.

2.3 Standardisation de l'architecture d'un nœud actif

Un réseau actif est un ensemble de nœuds dont certains sont actifs. La notion d'actif dans les nœuds est leur possibilité d'exécuter un programme qui peut modifier leur comportement vis-à-vis d'un flux donné. L'architecture proposée pour ces derniers repose sur un modèle à trois couches [13] (voir Figure 2.1) :

- La couche des applications actives qui représentent les programmes à exécuter,
- La couche des environnements d'exécution qui permettent d'interpréter les classes de programmes constituant une application,
- et le Système d'Exploitation du Nœud (NodeOS) qui gère plusieurs types de EE, la mémoire qui leur est allouée, le CPU, le système de fichiers, etc.

2.3.1 Application active

Une application active (AA) est une application dédiée à un ou plusieurs usagers qui réalise des fonctions de communication et opère dans le plan de données, dans le plan de contrôle ou dans le plan de supervision. Une AA s'exécute toujours dans le contexte d'un environnement d'exécution d'où elle puise toutes les ressources physiques et logicielles nécessaires. Une applications active est caractérisée par un nom, une ou plusieurs classes du code et le nom de la première classe à instancier.

Le code exécuté au niveau des nœuds actifs est soumis aux contraintes de flexibilité, portabilité, mobilité, sécurité et performance :

- flexibilité : le code doit être extensible et évolutif selon les besoins des services et leur dynamique. Il doit pouvoir aussi se combiner avec d'autres codes pour composer de nouveaux services.
- mobilité : elle se résume dans le transfert du code d'un nœud à un autre, selon un chemin bien déterminé,
- portabilité : exécution du code actif sur plusieurs plate-formes (systèmes et matériels différents),
- sécurité : est la restriction d'accès aux ressources afin de réduire les erreurs et le comportement inattendu, de préserver la confidentialité et l'intégrité des données et de réduire les risques d'attaque contre les nœuds et le réseau,
- performance : définit les caractéristiques nécessaires pour avoir des programmes actifs optimisés nécessitant le moins de ressources possibles.

Il existe plusieurs formats pour assurer la mobilité, la portabilité et la sûreté du code, comme les scripts TCL ou en format binaire comme les byte-code de Java et CAML.

2.3.2 Environnement d'exécution

La composante de base d'un nœud actif est son environnement d'exécution (EE). Un EE est une machine virtuelle capable d'interpréter un type de paquets actifs (ex. ANTS, PLAN, ...) et d'exécuter des AA. Elle offre des API spécifiques (librairies et services) aux codes contenus dans les paquets actifs, ainsi qu'un moteur d'exécution. Le EE met à disposition des AA les ressources autorisées du nœud avec les mécanismes d'ordonnancement pour ne pas laisser une application monopoliser une ressource donnée.

Les EEs, en termes de sécurité, doivent isoler les AAs entre elles pour éviter toute interférence dans leur exécution ou l'accès aux données. Pour des raisons de performances les EEs doivent être capables de gérer l'exécution parallèle de plusieurs AA. Parmi les EE qui ont été développés dans le cadre de projets nous citons ANTS [93], ASP EE [9][12] et CANE [14].

2.3.3 Le système d'exploitation d'un nœud (NodeOS)

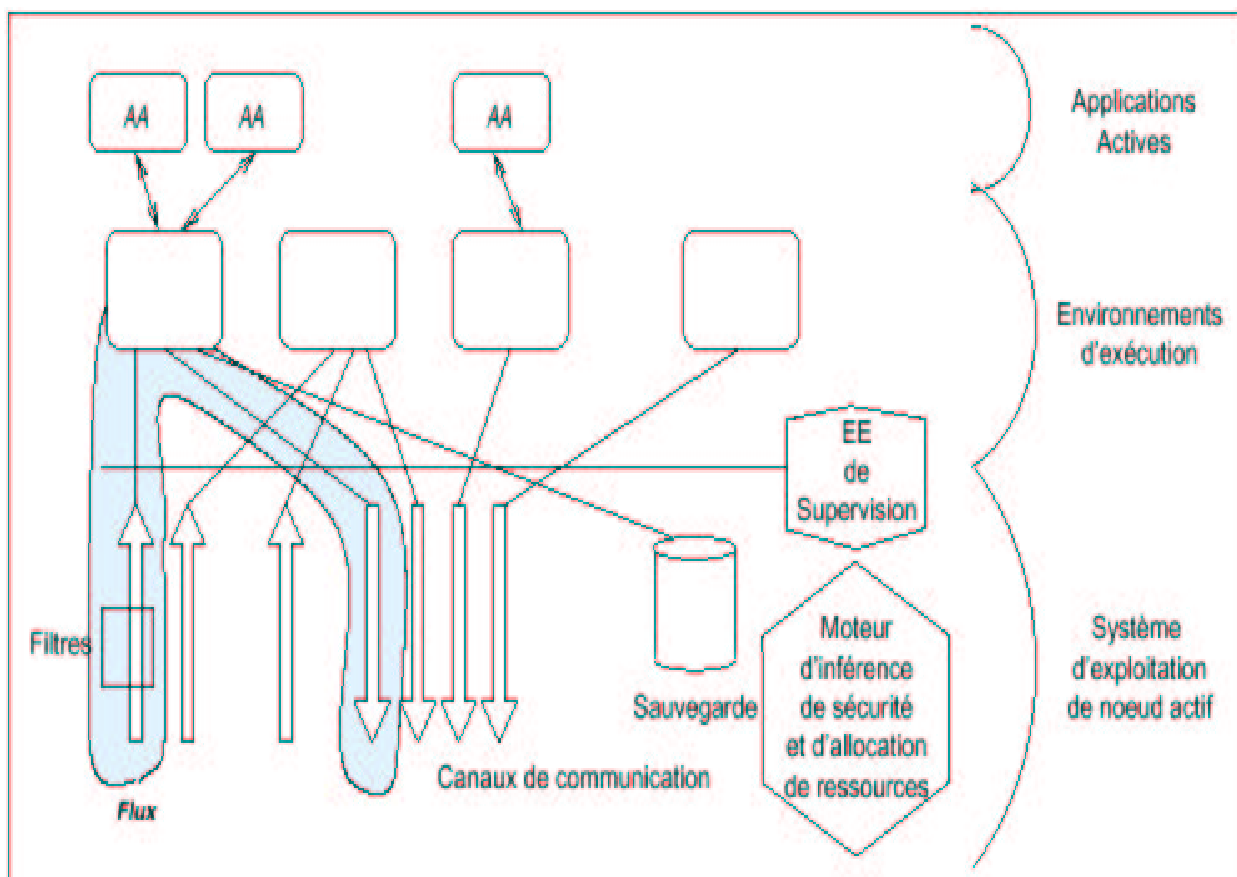


FIG. 2.1 – L'architecture standard d'un nœud actif

Le rôle d'un NodeOS est de rendre un nœud programmable au travers des interfaces et des abstractions qui représentent les ressources nécessaires à la programmation réseau (interfaces de communications, interaction avec les protocoles des différentes couches réseau, ...). Ces abstractions sont comme suit [39] :

- Exécution ⟶ file d'attente des processus (Thread pool),
- Mémoire ⟶ gestion de l'espace mémoire (Memory pool),
- Communications ⟶ les canaux d'entrée/sortie (Channel pool)
- Stockage ⟶ fichiers
- Domaine ⟶ regroupe les quatre abstractions précédentes pour gérer un flux donné.

Cette composition d'un NodeOS est le résultat du groupe de travail [39] dont le but était de normaliser l'architecture d'un nœud pour promouvoir une compatibilité entre les différentes plates-formes existantes.

La présence de plusieurs EE dans le même nœud induit forcément à une exécution parallèle et une orientation des données vers le EE correspondant. Le NodeOS doit être capable de gérer l'exécution parallèle des EE en les isolant individuellement et en les séparant de la gestion directe des ressources critiques. Si cette tâche est parfaitement accomplie, aucun EE (ou une AA à travers un EE) ne peut s'interférer avec les autres [60] (voir Figure 2.1).

L'ajout de programmes dans les nœuds actifs ne doit pas les dévier de leur vocation initiale, à savoir : le routage des paquets standards. Le rôle majeur d'un NodeOS est de minimiser l'impact négatif que peut avoir l'activation du nœud sur les paquets lorsque ceux-ci n'exigent pas un traitement particulier en fournissant des mécanismes optimisés de routage.

Pour la sécurité du nœud contre la mauvaise exécution des EE et AA, le NodeOS doit en premier lieu protéger les ressources du nœud.

Parmi les systèmes d'exploitation conçus pour la gestion des nœuds actifs nous pouvons citer JANOS [74], JOUST [50] , BOWMAN [82] ...

2.3.4 Le format de paquet

Toujours dans le cadre de la normalisation des réseaux actifs, quelques formats de paquets ont été retenus pour la transmission des codes et des données actives. Les formats de paquets doivent être reconnus essentiellement par les NodeOS pour pouvoir orienter les paquets vers les bons EE.

ANEP (Active Network Encapsulation Protocol) est un mécanisme d'encapsulation de paquets actifs. Il définit un format de paquet actif indépendant du protocole de communication utilisé par le réseau sous-jacent (IP, UDP, TCP, Trames Ethernet, ...) et s'adapte à tous les types de paquets actifs (ANTS, PLAN, SmartPackets, ...) [84][32] (voir Figure 2.2).

ANEP permet le multiplexage de plusieurs paquets actifs de différents types pour les transmettre sur le même canal, et leur démultiplexage se fait à l'arrivée [83].

Un paquet ANEP est composé d'une en-tête et une charge utile (payload) contenant le code actif. L'en-tête du paquet contient les champs suivants :

- la version ANEP,
- Flags : ce champ indique le traitement à faire (rejet, retransmission, ...) lorsque le routeur ne reconnaît pas le type du EE ou le routeur est passif,
- TypeID : identificateur du type du EE ciblé (ANTS, PLAN, ...) et ,
- un certain nombre d'options représentées sous forme : Type/Longueur/Valeur.

Le champ 'Option Type' est utilisé pour interpréter le champ 'Option Payload'. Les valeurs par défaut définies par l'agence de normalisation des réseaux actifs (ANANA) [83], sont :

- (1) : Identificateur source,
- (2) : Identificateur destination dans un schéma adressage donné (IPv4, IPv6, ...),
- (3) : Contrôle d'intégrité dans un schéma donné (MD5, SHA-1, ...)

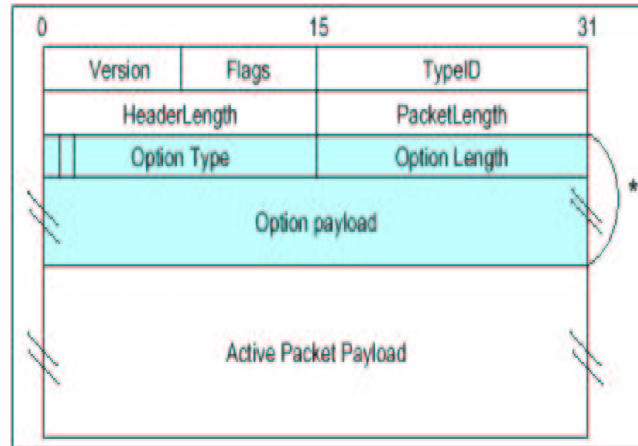


FIG. 2.2 – Le format des paquets ANEP

- (4) : Authentification non négociée (non-negocied authentication) pour fournir une authentification à sens unique.

Les mécanismes de cryptographie pour l'authentification, l'intégrité et la confidentialité peuvent être assurés par un paquet ANEP. Ceci permet d'éviter une redondance de ce traitement par les différentes couches supérieures.

SAPF (Simple Active Packet Format) est un autre format normalisé pour l'encapsulation de paquets actifs. SAPF est plus petit (en nombre de champs) que ANEP, mais moins répandu dans la communauté de recherche sur les réseaux actifs.

2.4 Les projets courants

Vu le grand nombre de projets et leur particularité, nous présentons dans cette section trois projets types. Le premier est un exemple de l'approche intégrée. Le second reprend le principe de l'approche discrête. Quant au dernier, il présente est une approche un peu différente car elle se base sur une technique compositionnelle et un déploiement dynamique du code.

2.4.1 ANTS : approche intégrée

Active Network Transfert System (ANTS), mis en œuvre par D. Wetherall, est l'extension des travaux du Massachusetts Institute of Technology (MIT) sur Active IP. Le projet Active IP, utilisant l'option 63 pour encapsuler un code actif dans un paquet IP normal, est resté un prototype et une base pour les travaux de MIT sur les réseaux actifs.

ANTS se bas sur l'approche intégrée (capsules) dont le code mobile est intégré dans les paquets IP et exécuté sur les routeurs IP sélectionnés et étendus [91] [92].

Les composantes principales d'ANTS sont :

- le protocole : il englobe le traitement à effectuer sur le flux de données. Il est défini par un identificateur (signature numérique du code), un ensemble de classes (codes) et les unités de

distribution des codes (capsules).
- les capsules : elles véhiculent les données et le code entre les nœuds actifs. Elles ont la structure suivante : l'adresse source et destination, champ TTL (Time to live idem que IP) pour éviter que les paquets restent indéfiniment dans le réseau, la version ANTS, l'adresse du dernier nœud traversé, et la charge utile (code ou données).
- le nœud actif : contient un EE incluant une machine virtuelle Java qui s'occupe de la réception, l'exécution et l'envoi des capsules.

Dans ANTS, un nouveau service est sous forme d'une hiérarchie à trois niveaux :
- Le Protocole : c'est l'unité logique et le traitement global à appliquer sur le réseau. Il est représenté par un ensemble de groupes de code et forme une unité de protection.
- Un groupe de code : c'est une fermeture transitive de routines de routage, en d'autres termes, il représente les routines appartenant au même flux et qui se référencent mutuellement.
- Une routine de routage : c'est l'unité exécutée et transmise sur une capsule, les routines appartenant au même protocole peuvent partager les mêmes données.

Le déploiement du code est opéré de proche en proche le long du chemin des paquets. Lorsque les données arrivent au nœud, le code à appliquer est soit extrait du paquet lui même ou indiqué en référence et téléchargé du nœud précédent. L'unité élémentaire pour un rapatriement n'est pas seulement la classe demandée mais tout le groupe de code.

2.4.2 Switchware : approche discrête

Le projet Switchware [6][52] regroupe l'ensemble des activités de l'université de Pennsylvanie autour des réseaux actifs. Il développe une approche à commutateur programmable qui permet à des modules, numériquement signés (par les mécanismes des signatures digitales) et à type contrôlé, d'être chargés dans les nœuds. L'idée est de définir une architecture homogène incluant les concepts de nœud actif, paquet actif et leur sécurité. Les couches du système SwitchWare sont [6] :

1. Le paquet actif : il remplace le paquet traditionnel, contenant des codes exécutables légers qui interagissent avec l'environnement du nœud et des données qui remplacent la charge utile (payload) du paquet traditionnel. Le code est écrit dans un langage restreint et dédié appelé PLAN pour assurer la sécurité du nœud [69].
2. Les extensions actives : ceux sont des programmes écrits dans des langages évolués (CAML). Contrairement aux paquets actifs, elles ne sont pas mobiles, mais font partie des fonctionnalités du routeur, appelés aussi Switchlets.
3. L'infrastructure du routeur actif : elle forme la couche basse de l'architecture. Elle est statique contrairement aux couches précédentes. Son but est de fournir une fondation sûre sur laquelle les deux couches précédentes vont être construites. Elle inclue le chargement et l'interprétation des paquets actifs et des Switchlets et garantit leur exécution.

SwitchWare admet trois composants de base :
- ALIEN : il se compose d'un chargeur actif (Active Loader) dont le rôle est le chargement des extensions actives (Switchlets) de manière sécurisée, d'un ensemble de librairies et des Switchlets spécialisées.
- PLAN (Programming Language for Active Networks) : c'est un langage de programmation des codes actifs proche de Scheme et CAML et se base sur le λ-Calcul, dont les programmes sont fortement typés et de petites tailles transportables dans des paquets [44].

- PLANet : un réseau couplant ALIEN et PLAN pour offrir un environnement d'exécution dédié. Les paquets PLAN sont augmentés de deux champs (numéro de la session et identificateur de flux). Ceci dans le but d'associer une application à un paquet (numéro de la session) et à un flux (identificateur de flux). PLANet s'appuie particulièrement sur le protocole IP [45].

2.4.3 FAIN : approche compositionnelle

Un autre type de distribution est basé sur une pré-programmation sélective des nœuds et une composition des modules de codes au moment de l'exécution par le nœud. Le projet phare dans ce type de réseaux actifs est FAIN (Future Active IP Networks) [67]. FAIN est une approche compositionnelle. Les modules de codes sont stockées dans un serveur de code, appelé Service Repository. Ces modules participent à la composition d'un service. Chaque service contient un descripteur qui est stocké dans un serveur, appelé Service Registry. Les nœuds potentiels sont sélectionnés selon un système de correspondance entre les besoins en ressources des services et les caractéristiques des nœuds (logiciels, langages, processeurs, ...).

Au niveau du nœud, le sous-système de composition, selon les spécificités du service formulées par le descripteur, récupère les modules de codes nécessaires à partir du Service Repository et les compose pour n'avoir qu'un seul module actif à charger en mémoire. Il installe ensuite un filtre pour la reconnaissance des paquets passifs à traiter, même esprit que l'approche nœud actif.

Par ailleurs, dans cette plate-forme active chaque fournisseur est capable de changer le code ou de le retirer (révoquer) complètement du réseau (le supprimer des nœuds et du serveur de code).

2.5 Sécurité

En raison de l'ouverture des ressources partagées des commutateurs aux programmes utilisateur, les services existants pourraient souffrir d'une dégradation de leurs performances, et plus dangereusement, l'arrêt de service dû aux exceptions, aux erreurs de programmation ou aux programmes malveillants.

Les risques en matière de sécurité induits par les réseaux actifs touchent essentiellement les accès non autorisés aux données en cours de traitement ou stockées, la consommation excessive de la mémoire, de cycles d'exécution et de la bande passante. Sécuriser les réseaux actifs revient à protéger par ordre croissant d'importance, les utilisateurs certifiés, les applications systèmes et les ressources des nœuds contre tout dépassement intentionnel (utilisation malveillante) ou non (erreur de programmation) des limites imposées [19].

La sécurité peut être considérée comme l'établissement d'un certain nombre de droits. Comme par exemple donner la bonne information, à la bonne personne, au bon endroit et au bon moment. Il existe aussi d'autres approches dans le domaine de la sécurité. Une première vue, plutôt pessimiste, considère la sécurité comme l'ensemble des mauvaises situations ou opérations à éviter. Tandis qu'une autre plus optimiste se limite aux seules situations et opérations à admettre.

Le groupe de travail dédié à la sécurité pour les réseaux actifs [40] a adressé un inventaire des différentes attaques que peut subir un réseau actif. Il a également adressé les techniques potentielles de sécurité pour certains types d'attaques.

Le but des mécanismes de sécurité est d'arriver à un degré de sûreté au moins égal à celui du réseau IP actuel. Le passage de paradigme d'un réseau configurable à un qui soit programmable mène inévitablement à un compromis sur la sûreté. Le reste de cette section discute de ces compromis.

2.5.1 Les règles de protection

Comme déjà mentionné dans la sous-section 2.3.3, l'architecture standard d'un nœud actif confie au NodeOS la gestion des ressources sous forme de files d'attente représentant les canaux de communication, la mémoire, l'espace disque et les cycles d'exécution. Ce rôle rend le NodeOS le plus habilité à protéger ces ressources. Dans ce qui suit, nous présentons les points nécessitant un effort particulier en matière de sécurité et les techniques utilisées dans les projets courants.

1. Protection des ressources : si les paquets standards consomment peu de bande passante, de mémoire pour la recopie de leur contenu et de temps d'exécution pour leur routage, les paquets actifs peuvent consommer davantage de ressources. Les réseaux actifs sont exposés à des risques plus onéreux car les programmes actifs consomment plus de ressources pour s'exécuter (CPU), se propager (bande passante) et stocker les données (la mémoire). La nature fortement programmable des réseaux actifs les rend susceptibles à une consommation excessive des ressources allouées. Les utilisateurs ou les EE malveillants tentent de dépasser les capacités allouées pour chaque ressource du nœud. Cette consommation de ressources peut être limitée de plusieurs façons :
   (a) Limite fixée par le paquet : quelques EE, tel que ANTS, assignent un TTL (time to live) à chaque paquet actif. Le TTL suggéré est 64 nœuds, ce qui correspond au diamètre d'Internet.
   (b) Limite fixée par le nœud : un nœud peut limiter l'espace mémoire, espace disque et la largeur de bande qu'un paquet peut consommer, mais il est difficile de limiter le temps CPU parce que les paquets doivent accomplir leurs tâches [31].
   (c) Limite fixée par le langage de programmation : la programmation des codes actifs par des langages restreints peut freiner la consommation excessive des ressources.
   (d) Approche commerciale : cette approche donne le coût au paquet traité, et selon la participation financière de l'utilisateur ses paquets auront un temps CPU proportionnel [31].
   (e) Ordonnancement : retire la ressource à l'utilisateur courant et l'accorde à un autre selon la politique et le type de la ressource [80] :
      - Pour la largeur de bande : par les algorithmes d'ordonnancement "best-effort", "par priorité" ou "garanti",
      - pour le calcul : les threads sont l'abstraction primaire d'exécution par le processeur. Le NodeOS gère et partage le temps CPU entre les différents threads selon la politique d'ordonnancement. L'ordonnanceur interrompt de manière préemptive les threads dès que leur quantum est expiré, avec des algorithmes tels que le round robin, le premier traité dont le temps d'exécution restant est le plus court (earlist-deadline-first) ou par priorité [80].
2. Confidentialité : le rôle du réseau est de mettre à disposition des utilisateurs des moyens de transmission pour leur permettre le partage et la communication des informations en toute sécurité et confidentialité. Si le réseau actif rompt cette contrainte tout le réseau est alors mis en cause. Les techniques de cryptographie constituent une stratégie satisfaisante à condition que les mécanismes d'authentification, d'attribution et de gestion des clefs et des droits soient aussi fiables.

Plus particulier, la protection de la mémoire fait partie intégrante de la protection de la

confidentialité. A la base, les systèmes d'exploitation actuels protègent l'accès mémoire. Il est d'autant de la responsabilité du langage d'exécution que du NodeOS de protéger l'utilisation de la mémoire. Java, qui est considéré comme un langage sûr, ne permet pas à une classe d'accéder aux données (les attributs) d'une autre classe si elle ne possède pas les permissions requises. De même pour la plupart des langages restreints conçus pour les réseaux actifs. Les accès non autorisés peuvent être une faiblesse dans les réseaux actifs si ces derniers ne fournissent pas des mécanismes de protection de haut niveau. En raison de l'importance des données des utilisateurs ou des informations propres au nœud et au réseau lui-même comme architecture matérielle et logicielle, la topologie et les tables de routage, le nœud doit se protéger contre des accès non autorisés.

3. Qualité de la protection : dans [97], la notion de QoP (qualité de la protection) est définie en tant qu'un ensemble minimal de règles que le système de sécurité (en grande partie le NodeOS) doit respecter. Cet ensemble comprend :
   - la longueur de la clef de l'algorithme de cryptage. La longueur de la clef symétrique dans les algorithmes à clef unique est de l'ordre de 56 octets, tandis que le couple des clefs publique et privée est autour de 1024 octets,
   - la robustesse et l'efficacité de l'algorithme de cryptage. Les différents types d'algorithmes de cryptage ont des caractéristiques et des degrés de fiabilité différents.
   - un mécanisme de sécurité pour l'authentification et la confidentialité,
   - le degré de confiance des développeurs ou fournisseurs de code,
   - le niveau d'assurance d'un NodeOS,
   - la sûreté des codes,
   - les permissions d'exécution sur les nœuds

Les avancées récentes dans les réseaux actifs ont vu l'introduction des plates-formes réseaux actifs de sûreté et de sécurité pour vérifier formellement qu'un système est sûr et le demeurera, ou pour renforcer la sécurité sur une architecture existante.

Les protections existantes contre les accès non autorisés consistent à des restrictions par le langage dès la compilation ou lors de l'exécution par les EE ou le NodeOS. La deuxième technique nécessite des mécanismes lourds à déployer dans le EE et le NodeOS.

Dans ce qui suit, nous présentons quelques projets dont le but est de pourvoir des réseaux actifs sécurisés. En premier lieu, les projets qui placent des mécanismes dans le nœud pour le protéger contre les attaques externes, en utilisant généralement les outils cryptographiques, tels que SANE et SANTS. Puis, ceux qui sont basés sur des restrictions du langage tels que PLAN et SafetyNet.

2.5.2 Approches avec une plate-forme sécurisée

Deux types de contrôles peuvent être opérés sur une plate-forme sécurisée : les contrôles statiques et les contrôles dynamiques. Les contrôles statiques sont employés pour vérifier l'état et l'intégrité du système après démarrage, par l'intermédiaire d'une architecture sécurisée de chargement. En ce qui concerne les contrôles dynamiques, ils sont exécutés par la plupart des plates-formes actives sécurisées comme SANE et effectuent essentiellement des vérifications sur l'authenticité des utilisateurs et les permissions d'exécutions.

2.5.2.1 SANE

Parmi les travaux de D. Scott Alexander sur la sécurité, nous citons l'environnement réseau actif sécurisé SANE (Secure Active Network Environment) qui fournit une architecture à couche pour la construction des infrastructures réseaux actifs sécurisés. En utilisant un ensemble de règles, la plate-forme charge et démarre le système dans un environnement sécurisé. SANE fournit également des services d'authentification pour le traitement fiable des paquets actifs. Pour maintenir un service fiable, la plate-forme SANE a été conçue pour fournir les contrôles dynamiques rapides, car ils sont exécutés fréquemment. Les contrôles statiques prennent considérablement plus de temps, mais ce genre de contrôles est normalement exécuté une fois seulement, à la mise en marche de système.

Les efforts de recherches sur la plate-forme SANE se sont concentrés pour atteindre un compromis entre des contrôles statiques rares et coûteux et les contrôles dynamiques fréquents, mais peu coûteux. Le projet Switchware emploie cet environnement comme base pour tous les services de sécurité, de la vérification de l'état initial du réseau et de l'authentification des paquets. De plus, la plate-forme SANE est assez générique pour être utilisée par d'autres architectures comme ANTS [19].

Secure Quality of Service Handling (SQoSH)[3] est une architecture globale qui emploie l'environnement d'exécution sécurisé SANE et Piglet un système d'exploitation léger spécialisé dans l'allocation de ressources. SQoSH contrôle l'accès aux ressources gérées et intègre ce contrôle avec les mécanismes de gestion de ressources fournis par le système d'exploitation Piglet [5] .

SQoSH considère deux types de violation : la violation du contrôle d'admission lors d'un accès non autorisé vers les ressources du nœud, et la seconde violation due à l'utilisation abusive et malsaine des ressources par un utilisateur autorisé. Pour la violation d'admission, SQoSH propose l'authentification forte pour le contrôle d'accès et la vérification des permissions à base de jetons d'autorisation ("credentials") pour l'accès aux ressources.

Pour vérifier la conformité entre les éléments des listes de contrôle d'accès, SQoSH génère des communications vers des serveurs qui détiennent ces listes ou les stocke sur les nœuds. Les deux solutions ne sont pas adaptées aux réseaux actifs car la première augmente la latence globale du flux actif par des accès supplémentaires pour l'autorisation et la deuxième est improbable étant données l'espace de stockage alloué et les mises à jour fréquentes des listes au niveau des nœuds.

2.5.2.2 ActiveSpec

ActiveSpec est une plate-forme pour la vérification formelle des politiques de sécurité pour les réseaux actifs et leurs services [24]. Par la définition formelle des services, des politiques et des ressources, les interactions dans un réseau actif peuvent être contrôlées selon les règles de sécurité. ActiveSpec demande des spécifications formelles simples des nœuds actifs (et de leurs ressources) et des politiques de sécurité, et vérifie par PVS (Prototype Verification System) [57] que les paquets ont accès aux ressources sécurisées seulement s'ils présentent les permissions requises. Ceci est réalisé en surveillant les canaux entrants et sortants pour tout paquet et les ressources qu'il consomme (l'état des nœuds actifs relatifs est également vérifié après traitement du paquet). Si une violation des ressources se produit, le procédé de vérification renvoie une erreur. La plate-forme ActiveSpec a été employée pour modéliser et vérifier formellement l'intégrité du service de chargement pour l'environnement SANE, garantissant un processus sécurisé au démarrage pour les architectures de réseaux actifs qui ne mettent pas en application leurs propres mécanismes de sécurité [19].

2.5.2.3 SANTS

Les travaux de Sandra Murphy ont abouti à l'élaboration de Secure ANTS (SANTS) pour les plates-formes ANTS sécurisées [71]. SANTS regroupe des mécanismes basés sur l'intégrité, l'authentification et l'autorisation qui emploient la cryptographie en utilisant :

- des certificats X.509v3 sauvegardés par DNS CERT et protégés par DNSEC pour la gestion des "credentials" (ou preuves de permission),
- Java api crypto et des extensions cryptographiques de Java,
- un système de politique (Keynote Policy System), qui est un langage et un évaluateur de politique qui permet aux utilisateurs d'établir leurs propres politiques de sécurité vérifiées au niveau du nœud.
- une authentification avec HMAC-SHA1 entre nœuds voisins.
- un système d'autorisation qui se base sur l'architecture de sécurité fournie par Java 2. Le chargement et l'exécution d'une classe Java nécessite une permission qui est stockée dans le domaine de protection de chaque classe. Le chargeur de classe vérifie la concordance entre les permissions requises et celles détenues.

Les inconvénients de SANTS sont la nécessité d'un traitement et des connexions supplémentaires faits par chaque nœud pour vérifier les règles de sécurité et les certificats. En plus le système d'autorisation n'est pas générique car il est limité seulement à Java.

Sandra Murphy définit la protection d'un réseau actif selon plusieurs points de vue :

- *Utilisateur* : par l'utilisation d'algorithmes cryptographiques de bout-en-bout contre des attaques touchant la confidentialité et l'intégrité des données par le EE ou tout autre code actif et même le NodeOS.
- *Nœud* : doit contrôler les demandes des EE et répartir les ressources y compris le temps d'exécution.
- *EE* : les menaces venant d'un code actif malveillant ou erroné peuvent être contrées par le EE en imposant des contrôles d'accès et des attributions valides de ressources. Cependant, le NodeOS et les EE peuvent collaborer pour mieux protéger le nœud par le partage de mécanismes de sécurité (librairies internes), des informations ou des états.
- *Code actif* : il peut faire l'objet de menaces de la part du EE, du NodeOS ou des autres codes actifs. L'analyse d'un code et l'accès à son espace d'adressage constituent les majeures attaques contre un code.

2.5.2.4 Seraphim

Seraphim est une plate-forme active qui utilise les travaux de [97] sur l'intégration des politiques de sécurité au niveau du NodeOS. Les politiques de sécurité sont évaluées et renforcées par le *Gardien Actif de Sécurité* (Active Security Guardian). Tout accès aux ressources du nœud doit être contrôlé par le Gardien de sécurité. Les politiques sont typiquement des autorisations d'accès fournis par le paquet actif lors de son passage par le nœud. L'authentification des politiques d'accès est faite par un mécanisme de cryptage fourni par le NodeOS sous forme de librairies de sécurité sollicitées par le Gardien.

2.5.3 Approche avec un langage restreint

Pourquoi un langage restreint ? Pour maîtriser les actions des paquets dans les nœuds, pour minimiser la sécurité au niveau du code du paquet actif, ou pour minimiser la sécurité au niveau

du EE ou du NodeOS lors de l'exécution des codes.

Beaucoup de langages de programmation ne fournissent pas assez d'informations pour déterminer si le programme est sûr avant l'exécution [4] [7]. Par ailleurs, les langages de programmation à but général (non spécialisés dans les réseaux actifs) vérifient la sûreté et la sécurité en temps réel comme Java, lors de l'exécution du code en déclenchant des exceptions. Ceci peut engendrer une dégradation des performances. Un langage restreint comme PLAN opère des vérifications de type et des restrictions d'accès aux ressources du nœud. L'inconvénient majeur de ces langages est la nécessité d'installer au préalable des extensions actives au niveau du EE ou NodeOS à appeler par le programme actif.

2.5.3.1 SafetyNet

La protection en cours d'exécution réduit considérablement les performances. Pour cela, SafetyNet dans le cadre d'un projet de l'université de Sussex essaie de définir un langage qui protège les ressources du nœud [90]. SafetyNet est un langage dédié aux réseaux actifs et caractérisé par un typage fort, une programmation adaptée à la communication et à l'hétérogénéité des systèmes (portable). Il définit, également, des politiques appropriées pour protéger l'intégrité du réseau, comme de fixer une limite TTL sur les paquets. Dans [90], il y est supposé que tout programme bien typé est sûr et ne permet pas de violer les politiques de sûreté. En outre, en utilisant un langage fortement typé, les mêmes auteurs pensent aussi qu'il est possible de supprimer quelques contrôles dynamiques très fréquents exigés par d'autres modèles de langage, et ainsi améliorer les performances.

Cela est justifié par les récentes avancées dans la théorie des langages qui ont fourni des modèles mathématiques dans la distribution et la migration de code, la limitation de ressource et la sécurité. Ces techniques sont employées pour fournir un modèle formel sûr et valide de programmation de réseaux actifs. Un compilateur et un environnement d'exécution pour le langage SafetyNet ont été développés.

2.5.3.2 PLAN

Programming Language for Active Networks (PLAN), développé dans le cadre du projet Switchware par l'université de Pennsylvanie, est un langage de programmation basé sur le λ-Calcul et conçu spécialement pour que le code soit très sûr, portable et surtout de taille petite qui utilise des ressources limitées du réseau [44][52]. Pour cela, PLAN fournit un ensemble restreint de primitives et de types de données [84][32].

Ces limites permettent principalement au code PLAN de faire appel à d'autres services (extensions actives) au niveau du nœud et qui sont écrits dans un langage plus évolué (CAML).

Les extensions actives combinées avec les paquets actifs donnent des résultats appréciables en termes de programmabilité du réseau [44]. Elles permettent aussi de fournir aux programmes PLAN des informations sur l'environnement du nœud actif comme l'adresse du nœud, son état, la résolution d'adresses et le routage.

Programming Language for Active Networks and Protocols (PLAN-P) a été proposé par l'IRISA dans le projet COMPOSE. C'est une extension de PLAN avec des EE optimisés par l'introduction de la notion de protocole (un programme non mobile téléchargé sur un équipement réseau) [87]. PLAN-P utilise une version orientée objet de CAML (OCAML).

Un effort considérable a été fait pour authentifier l'utilisateur et le code au lieu de composer une architecture globale sécurisée [19]. Ceci constitue l'inconvénient majeur des systèmes de sécurité existants.

2.6 Performance des réseaux actifs

Les performances des réseaux actifs est le deuxième aspect après la sécurité qui rend difficile leur émergence dans les réseaux actuels. Pour palier à ce problème plusieurs projets tels que ANN [21], TAMANOIR [33] se sont lancés dans la définition des nœuds actifs Hautes Performances.

Les caractéristiques visées pour ce genre de routeur à haute vitesse sont une fonction de routage matérielle pour optimiser le routage hors traitement actif et une exécution optimisée du code actif. Dans la majorité des réseaux actifs le code est typiquement un code interprété qui nécessite une machine virtuelle. L'interprétation d'un code Java est cinq fois plus lente dans les meilleurs des cas comparée à l'exécution d'un code binaire en C. Pour cette raison ces mêmes projets ont opté pour une transformation du code interprété vers un code binaire correspondant aux systèmes d'exploitation et au matériel en utilisant des compilateurs JIT (Just In Time compiler).

Quand le code est composé de plusieurs modules, un système de composition, comme dans les projets ANN et FAIN [67], est utilisé pour avoir un seul code afin d'optimiser le temps d'exécution.

Il y a d'autres projets qui sont allés plus loin en définissant des routeurs assistants. Un routeur assistant est une machine déléguée au traitement actif. Elle implémente un environnement d'exécution des codes. Dans ce cas un routeur continue de s'occuper de sa fonction habituelle de routage. A l'arrivée d'un paquet actif au routeur classique, ce paquet est directement routé à travers un lien haut débit (en Giga bit/s) vers l'assistant qui exécutera le code correspondant sur le paquet et le renvoie au routeur classique appelant. Le routeur est alors épargné du traitement actif (chargement et exécution du EE et des AA). L'avantage de cette technique est que les nœuds actifs peuvent être placés partout dans le réseau car leurs performances ne seront pas altérées. Néanmoins, elle peut présenter quelques inconvénients d'ordre financier dus au coût élevé de la mise en place de plusieurs équipements informatiques et d'ordre conceptuel dus à la difficulté de connecter plusieurs liens hauts débits vers les assistants. Ces deux points constituent un compromis qui est difficile à atteindre ; la présence de plusieurs assistants diminue le nombre de liens des routeurs vers les assistants et inversement.

L'analyse des performances : en dépit d'un manque d'étude et d'évaluation des performances, les résultats obtenus montrent qu'il y a une dégradation des performances des plates-formes actives comparées à celles d'un réseau classique. Les expériences faites sur les architectures réseaux actifs telles que PLANet ou même Hautes Performance comme TAMANOIR [33] et ANN [21] ont donné des débits sur des réseaux FastEthernet à 100 Mbps autour de 70 Mbps ou 700 Mbps sur des réseaux Ethernet Giga bps. Il est évident qu'un seuil est imposé aux performances du nœud, indépendamment des caractéristiques matérielles et logicielles lors de l'exécution (tels que le type d'environnement d'exécution, capacité du processeur, taille mémoire, ...). Pour avoir une idée de la grandeur de ces résultats, la même expérience sur le même type de réseaux a été faite sur un nœud passif. Le débit enregistré avoisine les 95 Mbps alors que la capacité du lien est de 100 Mbps. Une explication plausible pour ce comportement est l'impact de la recopie multiple des données du noyau système vers le domaine utilisateur pour les traiter, la commutation du contexte et l'exécution multi-threadée. Les projets implémentant des nœuds Hautes Performances, comme TAMANAOIR,

tentent d'éviter une recopie des paquets vers l'espace utilisateur.

Traitement des paquets actifs : le temps du traitement des codes actifs n'a pas été évalué ou il est dur de réaliser une telle étude en raison du grand nombre de paramètres, qu'on doit prendre en compte. Parmi ces paramètres il faut compter le temps de rapatriement des codes, le temps pris par le NodeOS pour orienter les données vers le EE, le temps pris par le EE pour orienter les données vers l'application active, le temps éventuel de chargement du EE, le temps éventuel de chargement du AA et le temps d'exécution du code. Parmi tous les temps cités, seul le temps d'exécution du code est variable d'un code à un autre et il est difficile à calculer car cela dépend de la nature du code (interprété ou binaire). Tous les autres temps dépendent aussi des caractéristiques matérielles et logicielles des nœuds, mais ils restent inchangés quelque soit le code pour une configuration donnée. Néanmoins, il est très intéressant d'avoir une grandeur abstraite pour ces temps. Les travaux de [30] ont permis d'avoir des approximations du temps d'exécution dans les plates-formes actives.

Traitement des paquets passifs : comme souligné au début de cette section, les architectures hautes performances fournissent des mécanismes optimisés pour le traitement des paquets non actifs. A part la proposition d'établir un routage physique rapide pour les paquets standards, il existe des solutions qui restent dans le domaine du logiciel en proposant des canaux rapides au niveau du NodeOS (comme le système d'exploitation BOWMAN [82]). Ceci évite la recopie des données de l'espace système vers l'espace utilisateur, la commutation de contexte, et le traitement multi-threadé pour ce genre de paquet.

2.7 Distribution & Identification du code :

La distribution du code et son identification sont deux phases dans le cycle de vie d'un service actif et sont étroitement liées. Il est impossible de présenter une phase sans l'autre. Les sous-sections suivantes tentent de décrire ces phases et les projets qui ont suscité un intérêt particulier.

2.7.1 La distribution de code

La majorité des projets que nous avons étudiés ignorent l'aspect de la distribution de code et son identification pour se focaliser sur l'implémentation d'un nœud actif performant. La plupart des projets à base de nœuds actifs (approche discrète) comme TAMANOIR [33], ANN [21], ASP EE [9] et FAIN [67] utilisent la notion de *serveur de code* pour le rapatriement des programmes actifs ou des modules de code. Dans ce genre de réseau actif, le code est téléchargé en dehors du flux de données. Néanmoins, dans tous les projets, le code est supposé au préalable au niveau du serveur. Toute la phase de l'injection du code par le développeur avec ses mécanismes de gestion de l'identifiant, la sûreté du code et l'authentification ne sont pas prises en compte. Dans le cas, des projets basés sur l'approche intégrée comme ANTS, le code est fourni avec les données. Dans ce cas il n'est pas nécessaire d'utiliser un serveur de code pour le stockage et le rapatriement du code vers les nœuds. Pour éviter les grandes tailles des paquets, [93] suggérait un autre schéma de déploiement de code. Cette solution, partageant aussi le flux entre codes et données, consiste à envoyer le code de proche en proche ou d'un nœud à un autre si ce dernier ne possédait pas le code. La technique se déroule en deux phases. Dans la première les paquets de données véhiculant la référence du code arrivent aux nœuds qui chargent et exécutent le code en question s'il est présent dans sa base. La

seconde phase intervient seulement si le code est absent dans les nœuds qui doivent le télécharger du nœud précédemment visité.

2.7.2 Identification de code

L'identification des applications reste floue dans la majorité des projets. Aucune standardisation n'a pu être faite dans ce domaine. Le problème ne se pose pas pour les EE car une autorité de normalisation ANANA (Active Network Normalisation Authority) qui attribue à chaque EE un identifiant unique et universel (reconnu par les NodeOS) a été définie. L'absence de standardisation est justifié par le fait que chaque EE est laissé libre dans la manière de gérer l'identification des AA. Cet identifiant doit, à notre avis, être universel, unique et délivré de manière uniforme. Pour montrer l'importance de ce problème, dans la plate-forme active d'expérimentation Abone, la résolution des conflits pour le nommage des AA se fait manuellement sur le serveur de code. Ce problème sera, sans aucun doute, plus épineux lors du passage à l'échelle du réseau Internet où il y a plusieurs serveurs de code ou si le code se déploie à travers le flux de données. Dans ces cas l'identifiant sera mal maîtrisé et des conflits seront plus difficiles à résoudre.

L'approche nœud programmable est orienté flux. Cela implique que le service est installé sur le nœud pour traiter un flux de paquets IP spécifique, identifié par son type (UDP, RTP, TCP, ...) ou par une classe d'adresses IP (l'adresse source ou destination). Cette solution nécessite d'installer, en même temps que le service, un filtre au niveau du NodeOS qui permet de reconnaître et d'orienter les paquets. L'inconvénient est que la classe des clients est figée au départ dès l'installation des filtres et aucune technique de mises à jours des filtres n'est explicitée. Par contre, son point fort est qu'elle peut aussi traiter des paquets passifs.

Puisque dans l'approche capsules le code est transmis dans le flux des données et n'est pas stocké de manière permanente dans un serveur globale, la résolution des noms des AA reste donc locale au routeur. Ainsi une clef, sous forme d'empreinte numérique, constituée à partir du contenu du code et la clef unique du développeur sont largement suffisantes pour garantir l'unicité de l'identifiant.

2.7.3 Distribution & Identification dans les projets actuels

Nous pouvons diviser les contributions courantes en deux principales catégories selon l'utilisation ou non d'un *serveur de code*. Dans la première catégorie les projets sont principalement du type discret (ou hors bande). Parmi ces projets, la technique du *cache de code distribué* pour les réseaux actifs (DAN) [22] dans le cadre de la plate-forme haut débit ANN [21] est une contribution qui définit la notion de serveur de code dont le rôle est seulement le stockage et la publication des modules de codes. DAN ne propose aucun mécanisme d'identification de code ou de techniques de vérification de sûreté de l'exécution des programmes et d'authentification de leurs développeurs.

FAIN [67] peut être aussi considéré de cette catégorie car les composants des services sont stockés dans un serveur de code. En ce qui concerne l'identification, chaque service est identifié par concaténation du nom du *fournisseur* et le *nom du service*. Au niveau du nœud actif, les données actives interceptées sont orientées au service correspondant par un canal créé au niveau du NodeOS. Cette correspondance ne nécessite pas la référence du service car ce dernier définit le type de paquets filtrés par le canal. Cette façon limite les utilisateurs du service à seulement ceux qui appartiennent à l'ensemble des clients du fournisseur du service. Nous devons noter qu'avec ce type d'identification le service reste fortement lié à son fournisseur ou développeur qui doit, par conséquent, garantir l'unicité du nom du service. L'unicité de l'identifiant du code est garantie sur le serveur de code

par plusieurs tentatives de la part du fournisseur qui peuvent échouer sinon par la création de plusieurs versions. Cette dernière façon de gérer l'unicité peut difficilement maintenir la cohérence des différentes versions déployées sur les nœuds. La différence principale entre les différents systèmes de distribution de code et celle de FAIN est que cette dernière est dédiée à une architecture basée sur un déploiement de composantes tandis que les autres architectures sont basées sur un déploiement d'un code invoqué par des paquets.

Le projet TAMANOIR et les travaux de Jean-Patrick Gelas [33], qui portent principalement sur la définition d'une architecture d'un nœud actif haute performance, projètent l'utilisation d'un serveur de code appelé *service repository* pour le déploiement de services actifs dans un réseau à hauts débits. Ces travaux ne présentent aucune motivation d'établissement d'un schéma global de distribution et d'identification des codes. Les codes sont supposés dans le serveur de code.

Dans le projet ASP EE [9], les AA sont identifiées par un triplet composé d'un nom global unique, d'une liste d'URL vers le serveur de code hébergeant les modules ou les classes de l'application et finalement le nom de la première classe à instancier.

Le schéma de distribution de code pour les réseaux actifs (CDS), proposé dans [98], est un ensemble de recommandations pour la conception d'un protocole de distribution de code (CDP). Les auteurs recommandent une transmission hors bande du code (dans un flux séparé de celui des données) plutôt qu'une transmission en-bande (dans le même flux que les données) en raison des grandes tailles des capsules. Ils suggèrent également la sauvegarde des AA dans un serveur de code. Ces mêmes auteurs considèrent la méthode de "nommage" des AA et de leur "résolution de nom" par le EE comme fonction de base d'un CDP. Par conséquent, CDS recommande que le nom d'une AA devrait être universel et identifié de manière unique une seule AA.

Les projets de la seconde catégorie sont principalement basés sur le déploiement de code en bande. Le projet propulseur de cette catégorie est ANTS [93]. Il se base sur une migration de proche en proche du code dans le chemin des données. La technique d'identification des AA employée par ANTS [94] définit un condensât MD5 pour identifier et authentifier le code des utilisateurs. La clef privée du certificat PKI de l'utilisateur et le code actif lui-même sont combinés ensemble pour créer une signature numérique ou un code de hashage de longueur fixe de 128 bits. Ce code de hashage est propre à un seul code et à un seul développeur. Ce fort couplage entre la clef privée du propriétaire et l'identifiant rend le partage et la réutilisation du code entre les utilisateurs impossible par le cryptage MD5 et oblige le nœud à authentifier la clef public du certificat pour vérifier la correspondance.

D'autres techniques telles que PromethOS [54] et les travaux de Robert Haas [43] ont défini de nouveaux mécanismes de déploiement de service pour les réseaux actifs. PromethOS définit un routage basé sur le chemin, Path-Based Routing (PBR), qui établit un chemin explicite par flue à travers une liste de nœuds prédéfinie. Un algorithme, qui calcule le chemin le plus court parmi les nœuds intermédiaires fournissant des fonctionnalités spécifiques, a été défini dans PromethOS. Dans [43], un mécanisme composé de cinq étapes de calculs hiérarchiquement distribués à travers le réseau et au niveau de chaque nœud. Ces étapes sont la sollicitation, l'agrégation, la dissémination, l'installation, et l'annonce. Des algorithmes sont proposés pour calculer ces chemins comportant les nœuds potentiels. Ces deux techniques utilisent des algorithmes de pré-selection des nœuds et du calcul du chemin le plus court.

2.8 La composition

Comme la sécurité, la composition a eu la considération de la communauté réseaux actifs qui lui a même consacré un groupe de travail [41]. Les travaux de ce groupe n'ont pas porté sur la spécification d'une méthode uniforme dans la composition des services, mais ils ont établi les critères et les règles pour la définition, selon les spécificités de chaque EE, d'une méthode efficace.

De même pour la sécurité et la distribution de code, la composition trouvera du mal à imposer une standardisation dans les réseaux actifs. La multitude des langages de programmation pour les AA et l'hétérogénéité des architectures des environnements d'exécution constituent un frein à la normalisation. En guise d'exemple, l'utilisation des classes Java où l'héritage et la portabilité facilitent la création des classes composites. Alors que les langages qui ne permettent ni l'héritage ni la portabilité des modules rendent difficile, voire même impossible la conception de services composites invoqués sur différentes plates-formes.

Le besoin de la composition n'est pas limité aux réseaux en général ou aux réseaux actifs en particulier. Pour son évolution et extension, tout système fournissant des services doit présenter des mécanismes pour composer de nouveaux services à partir de services existants. Dans les réseaux actifs, caractérisés par leur haute programmabilité, le besoin en composition des services s'accentue et devient même un concept de base à prendre en compte dès la conception d'un tel réseau.

Afin de mesurer l'importance de la composition, nous énumérons ci-après ses avantages les plus importants :
- la flexibilité du système,
- la disponibilité de services,
- le grand nombre de services qui peut être créé à partir de services plus simples,
- la réduction du temps de développement et de tests,
- l'augmentation de la réutisabilité des services,
- la réduction de la probabilité d'erreur et l'augmentation de la fiabilité par l'utilisation de composants validés,
- la réduction du temps du déploiement des modules de codes et
- la réduction du temps de contrôle de la sécurité sur les services (vérifier seulement le service composite au lieu de la vérification de l'ensemble),

Chaque méthode de composition doit, selon [41], vérifier les spécificités élémentaires suivantes :
- un contrôle de séquence : c'est la capacité de décrire la hiérarchie et l'enchaînement de base pour un service composé. Il doit définir aussi l'ordre de l'exécution des composantes selon la politique choisie ou les capacités des EE (exécution séquentielle et parallèle). Ce contrôle devient complexe lorsque les composantes interagissent entre elles par des appels mutuels.
- un contrôle du partage des données : il définit les droits d'accès et la manière avec laquelle sont partagées les informations entre services (partage par paramètres, par objets, par héritage, ...). Lors que l'exécution du service composite, les données partagées doivent demeurer intègres, même si elles subissent des modifications par l'une des composantes.
- un contrôle de la liaison dynamique entre services : il détermine le temps de la création du service (au niveau développement et compilation) et le temps de son exécution (instanciation des différentes composantes au niveau du nœud),
- des méthodes d'invocation : elles représentent les événements qui déclenchent l'exécution du service composite (lors de l'arrivée d'un paquet, par le nœud, par un changement d'état, ...),
- une fonctionnalité de division : c'est la spécification du service composite et la définition de la logique entre les éléments transmis par le paquet (code, script) et les éléments résidant dans

le nœud (les extensions actives, procédures systèmes).

Parmi les projets qui se sont intéressés à la composition des services actifs on note CANEs (Composable Active Network Environments) [14]. CANEs a été construit spécialement pour la composition de services par la spécification de mécanismes de composition robustes et expressifs pour ne pas corrompre les performances et la sécurité des nœuds.

Le système CANEs définit deux types de programmes, les *programmes injectés* par les utilisateurs et les *programmes de base* fournis par le système. La création d'un service se fait par l'intermédiaire d'une interface de programmation qui définit clairement les interactions entre les programmes injectés et les programmes de base. Les interactions sont typiquement les appels, le flux de données et le partage des états, etc. Le partage des variables entre programmes, point critique en raison de la vulnérabilité qu'il crée au sein du système, est rendu possible sous CANES. Les variables peuvent désormais être accessibles par plusieurs programmes (injectés ou de base) à la fois. Une variable est déclarée et exportée seulement par les programmes de base. A leur tour, pour utiliser une variable précédemment exportée par un programme de base, les programmes injectés doivent importer cette variable localement et lui créer une référence. Il est tout à fait possible qu'un programme injecté puisse modifier le contenu d'une variable.

Dans le cadre du projet AMARAGE [35], en utilisant une extension de la méthodologie ODAC (Open Distributed Applications Construction) développée pour la composition de services sur ANTS qui, à cause de la séparabilité des protocoles, rend difficile la construction des services composites.

ODAC définit un ensemble de concepts et leurs propres règles et spécifications telles que la spécification de comportements (définit l'objectif, les informations à traiter et des tâches à accomplir), de l'ingénierie (décrit les EE cibles) et finalement les spécifications opérationnelles (projettent un comportement sur un EE spécifique). La composition des services revient, alors, à la composition des spécifications.

2.9 Conclusion

Le nombre formidable de chercheurs qui constitue la communauté s'intéressant au domaine des réseaux actifs comparé à sa nouveauté reflète l'originalité et le potentiel que peut détenir la philosophie des réseaux actifs. L'alternative qu'elle propose aux réseaux classiques actuels rend plausible le passage vers les *réseaux de la nouvelle génération* ou communément appelé Next Generation Network (NGN) tant attendus. L'architecture active devrait permettre au réseau (structure, équipements et protocoles) de suivre l'évolution des applications Web et de profiter du progrès continu des capacités de calcul et de l'espace de stockage.

L'idée des réseaux actifs est l'ouverture des équipements de routage en les rendant programmables et dynamiques. Ceci entraîne inévitablement une architecture nouvelle d'un nœud actif et d'un réseau actif. Le défi à relever par cette architecture est sa capacité à faire face aux problèmes de sécurité et de performances.

Tout le long de ce chapitre nous avons essayé de répondre à cette question au travers des différents projets.

Cette étude bibliographique nous a permis de faire ressortir les axes importants de recherche dans le domaine des réseaux actifs. Ces axes sont la distribution du code, la sécurité, la composition des services et finalement la construction d'un nœud haute performance.

Le point commun à tous ces axes est la non existence d'une normalisation même si plusieurs groupes de travail ont été constitués. Ces groupes de travail proposent des règles ou des modèles

à prendre en compte sans pour autant exiger ou fournir une méthodologie spécifique et commune. Ceci est la cause de la multitude des projets qui a amené à la formation de plusieurs langages de codes actifs, de plusieurs environnements d'exécution et des architectures hétérogènes des nœuds.

L'harmonisation dans les réseaux actifs, à notre avis, doit d'abord commencer par la définition d'un langage adapté aux applications réseau, simple et à la porté de tout développeur. Les applications dérivées doivent comporter les caractéristiques suivantes : composabilité, modularité, sécurité, mobilité et performance.

Dans les chapitres suivants nous proposont une architecture nouvelle d'un réseau actif basée sur la notion de règles actives. Par leur formulation adaptée aux applications actives, les règles actives peuvent donner une nouvelle impulsion et peut être une nouvelle conception des réseaux actifs.

Le prochain chapitre sera consacré à la présentation des règles actives, de leurs composantes et du système actif qui doit se charger de leur exécution.

Chapitre 3

Les Règles actives

3.1 Introduction

Dans les bases de données conventionnelles, seules les requêtes SQL ou les transactions sont exécutées et explicitement soumises par des utilisateurs ou des applications. Certaines requêtes ou transactions, principalement des requêtes d'insertions et de mises à jour, peuvent altérer la cohérence de la base de données. Dans ces situations, il est impératif de redonner à la base un état cohérent dans les temps impartis en déclenchant des procédures spécifiques. Par exemple, si après une mise à jour d'un champ, la nouvelle valeur n'est pas cohérente, une procédure de correction ou d'alerte est alors déclenchée. Ce genre de procédures, appelée déclencheurs ou *triggers*, appliquées dans les bases de données passives permet des fonctionnalités avancées dans les systèmes de gestion des bases de données (SGBD). Les règles actives ont été présentées comme une extension aux déclencheurs traditionnels.

Les règles actives ont été introduites au début des années 80 dans les systèmes d'Intelligence Artificielle (IA) et les systèmes experts [20], en tant que notion de règles de production :

$$<condition> \rightarrow <action>$$

L'introduction des règles actives dans les bases de données a permis la définition d'un nouveau paradigme communément appelé *les bases de données actives*. Parmi les applications qui ont poussées à intégrer un traitement actif aux bases de données, nous comptons l'application de vérification complexe d'intégrité des données, les traitements d'exceptions et des actions de réparation (comme la maintenance de données et la propagation de mises à jour dérivées dans un système à cache). En plus de leur utilisation pour résoudre ces problèmes techniques à l'intérieur d'un SGBD, les règles actives s'appliquent aussi à la gestion des contraintes économiques et financières, l'établissement des workflows, des modèles complexes de transaction, et plus généralement à chaque domaine d'application où la sémantique d'exécution dépend des politiques définies par l'utilisateur [95]. Elles sont aussi utilisées pour renforcer les contraintes d'accès, d'admission et d'intégrité et dans les contrôles de version.

Une règle active est définie sous la forme "Evénement-Condition-Action" (ou ECA). Les *événements* peuvent être tout signal ou changement d'état défini par l'application tel qu'un événement temporel, un flux de données, un événement du système d'exploitation ou un événement propre à l'application. Les *conditions* sont des formules logiques ou des fonctions booléennes qui sont évaluées sur des instances d'un événement ou sur un état donné de la base de données ou du système. Les *actions* peuvent être toutes les opérations concernant le traitement d'une application. Un système

actif de base de données fournit un modèle de connaissance qui permet de définir des règles ECA et un ensemble de sémantiques selon lesquelles les règles actives vont être exécutées. Ainsi, le système actif comporte un moniteur actif admettant une sémantique paramétrable ou plusieurs moniteurs où chacun est dédié à une sémantique d'exécution spécifique. Le *moniteur actif* a pour rôle la détection des instances d'un événement, l'évaluation des conditions et l'exécution (ou déclenchement) des actions. Une fois exécutées, ces actions peuvent produire à leur tour d'autres événements qui lanceront d'autres actions dans un processus en cascade.

Les systèmes actifs offrent un formalisme simple, mais très puissant qui peut facilement s'appliquer dans des domaines autres que les bases de données et l'intelligence artificielle. Les événements, les conditions et les actions peuvent concerner le dialogue dans une interaction homme-machine, la synchronisation des flux d'activités (workflow), la communication dans un système de surveillance ou dans un réseau informatique. Le premier avantage des règles actives est leur modularité ; elles décrivent un comportement du système avec une granularité fine qui peut évoluer facilement selon l'évolution du domaine d'application. Le deuxième avantage est la flexibilité de leur exécution ; elles concentrent des politiques d'exécution dans une description uniforme et non redondante. Cela permet d'éviter la diffusion répétée de ces politiques d'exécution dans les applications des utilisateurs, sinon des comportements redondants et potentiellement contradictoires peuvent apparaître.

Dans les bases de données, les règles actives sont comme tous les autres objets. Elles peuvent faire l'objet de création, de suppression et aussi de modification. En plus, les règles ont d'autres opérations propres à elles telles que : *fire* pour déclencher une règle, *enable* pour l'activer et *disable* pour la désactiver [20].

La section 3.2 donne une définition détaillée des règles actives et de leurs composantes. La section 3.3 fournit l'ensemble des sémantiques qui régissent l'exécution d'une règle. Les directives nécessaires à la modélisation des règles actives sont présentées dans la section 3.4. La section 3.5 permet d'avoir une idée sur la contribution de la communauté des bases de données dans le domaine des règles actives. Et finalement la section 3.6 conclut ce chapitre.

3.2 Les composantes d'une règle active

Une règle active est constituée de trois composantes et respecte une forme bien précise. La forme générale d'une règle active est la suivante :

On <nom de l'événement> *if* <expression de la condition> *then* <invocation de l'action>

L'ordre d'évaluation de la règle suit exactement l'ordre de définition de toutes ses composantes. A la détection d'un évènement le système actif évalue la condition et selon le résultat, l'action est alors lancée. Cette section décrit toutes ces composantes et les illustre à travers un exemple d'une règle active appliquée sur une table d'une base de données. La table contient des informations relatives à un ensemble de pilotes. Les champs de la table "*Pilote*" sont montrés par la table 3.1.

Nom du Pilote	Prénom du Pilote	Age	Nombre d'heures de vol	Catégorie

TAB. 3.1 – Table Pilote

3.2.1 La composante événement

L'événement joue un rôle central dans un système actif. Les systèmes actifs traitent seulement des événements discrets. Nous distinguons entre les types d'événements (ou les classes d'événements) et les occurrences d'événements (ou les instances). Des types d'événements peuvent être structurés et organisés dans un diagramme de classe. Nous supposons que les instances d'événements sont, habituellement, identifiées au moment où elles se produisent (appelées horodateurs), ayant pour résultat un ordre total des instances de l'événement. Un environnement distribué peut poser quelques problèmes concernant cette hypothèse à moins d'admettre l'existence d'une horloge universelle au préalable. Les types d'événements sont principalement classés dans deux catégories : *événements internes* et *événements externes*. Les événements internes sont ceux produits par le système d'exploitation ou le moniteur actif gérant les règles actives. Les événements temporels sont des événements en général internes dont la source est l'horloge du système. Dans un système transactionnel, on peut trouver les types suivants d'événement : BeginTransaction, CommitTransaction, AbordTransaction, RollbackTransaction. Une erreur d'entrée/sortie et la division par Zéro, en tant qu'exceptions ou erreurs d'exécution du programme, peuvent être récupérées et considérées comme événements internes. Les événements externes sont ceux provenant des applications d'utilisateurs, telles que les flux de données, les interactions avec les utilisateurs ou l'environnement externe. Les paquets de synchronisation, par exemple, représentent des événements externes.

Les événements sont considérés comme *primitifs* s'ils proviennent d'une source et ne sont pas agrégés par une fonction de composition. Ils sont considérés comme *composés* (ou complexes) s'ils se composent d'événements primitifs en utilisant la fonction d'agrégat dont les opérandes sont produites par la même source ou par différentes sources.

La plupart des événements déclencheurs des règles actives dans les bases de données sont les changements dans les données de la base. Dans les bases de données relationnelles par exemple les commandes telles que **insert**, **update** et **delete** modifient les données et aussi déclenchent les événements.

Voici un exemple d'une définition d'une règle active $Change_Nombre_Heures_de_Vol$ qui réagit à la modification du champ $Nombre_Heures_de_Vol$ de la table "*Pilote*". L'évènement est déclenché à l'application d'une mise à jour du champ par la commande **update**.

```
define rule Change_Nombre_Heures_de_Vol
On update Pilote.Nombre_Heures_de_Vol
if ...
then ...
```

3.2.2 La composante condition

Cette composante de la règle active peut être une formule logique (par exemple une requête SQL sur une base de données) ou une fonction booléenne. Les variables dans la composante condition peuvent se rapporter aux données dans l'instance de l'événement ou au contenu d'une base de données ou gérées par le système de fichiers. Par exemple, le résultat :

- d'une *sélection : Select * From Pilote Where Nom_Pilote="DuPont"*. L'existence d'un tuple lors de l'application de la requête peut être considéré comme une condition vraie. Si aucun tuple ne correspond à la requête alors la condition est fausse.
- d'une *fonction Booléenne : Si (Nom_Pilote="DuPont")*,

– ou d'une expression logique (Condition1 *et* Condition2) *Ou* Condition3.

Si l'expression de la condition est une formule logique, les conditions sont directement indiquées sous une forme déclarative plutôt que sous une forme encapsulée.

Dans exemple "*Change_Nombre_Heures_de_Vol*", la même règle ne peut se déclencher que, par exemple, le nombre d'heures de vol est supérieur à 1500 heures.

```
define rule Change_Nombre_Heures_de_Vol
On update Pilote.Nombre_Heures_de_Vol
if Pilote.Nombre_Heures_de_Vol > 1500
then ...
```

La composante condition est facultative ou optionnelle ; dans ce cas nous parlons de règles de type Evénement-Action (E-A).

3.2.3 La composante action

L'action d'une règle active peut être une action de réparation, qui corrige une valeur incohérente d'un champ. Par exemple, si la valeur du champ *Nombre_Heures_de_Vol* est négative le système doit signaler cette incohérence à l'administrateur de la base de données ou appliquer directement la correction. Une action peut être une propagation de mise à jour s'il y a des dépendances entre plusieurs champs de différentes tables. par exemple, la suppression d'une entrée dans une table implique la suppression de toutes les entrées dans les tables en relation de jointures. Elle peut être aussi une notification à un utilisateur ou à un autre système, un message à un objet ou un appel de procédure (invocation locale ou à distance).

Dans les bases de données relationnelles, cela se traduit par l'exécution d'une requête d'insertion, de suppression ou de mise à jour comme :

– *Insert into table Pilote (Matricule, Nom_ Pilote, Prénom_ Pilote, Age, Nombre_ Heures_ de_ Vol) values (235254, "DuPont", "Jean", 34,1500, "Expert").*

La partie action d'une règle active peut être composée d'une ou de plusieurs opérations primitives. De même que la condition, la partie action est exprimée dans une forme déclarative. L'implémentation correspondante peut être encapsulée sous forme, par exemple, d'une méthode d'un objet.

L'exemple suivant rajoute à la règle "*Change_Nombre_Heures_de_Vol*" l'action correspondante sous forme de requête SQL de mise à jour.

```
define rule Change_Nombre_Heures_de_Vol
On update Pilote.Nombre_Heures_de_Vol
if Pilote.Nombre_Heures_de_Vol > 1500
then UPDATE Pilote SET Catégorie ='Expert' WHERE Age>=30
```

3.2.4 L'application active

Une *application active* est un ensemble de règles actives, généralement organisées en modules, et des spécifications d'une sémantique d'exécution pour chaque module de règle ou pour toute l'application active.

Un ensemble de règles ayant le même événement de déclenchement est appelé *ensemble de conflit*. L'exécution d'un ensemble de conflit doit obéir à un certain ordre. L'application d'un ordre arbitraire peut donner plusieurs résultats à l'exécution de toutes les règles.

Les règles sans composante événement (règles du type Condition-Action) ne sont pas des règles actives mais des règles de production. Les règles de production peuvent aider à modéliser un certain comportement de système d'une manière déclarative comme elle a été faite pendant longtemps dans les système experts et l'intelligence artificielle. Les règles de production ne sont pas considérées en tant qu'élément ou sous partie des règles actives et leur utilisation demeure indépendante dans de tels systèmes. Il est à noter que les éléments essentiels qui constituent réellement une règle active sont l'événement et l'action. L'absence de l'un de ces deux éléments enlève le caractère actif à cette règle.

3.3 Les sémantiques d'exécution des règles actives

Afin de comprendre les mécanismes liés à la sémantique d'exécution des règles actives, nous définissons un cycle de vie simple à l'exécution d'une règle. Ce cycle de vie comporte sept phases indépendantes, comme l'illustre la figure 3.1. La première phase détecte les instances d'événement produites par l'application de l'utilisateur, l'interface utilisateur ou le moniteur actif. La deuxième phase est la signalisation des événements qui diffère de la détection. En effet, la détection est le processus qui permet de détecter et d'analyser les événements ; ce qui peut être fait localement ou dans un système à distance. La signalisation d'événement est le processus qui met l'événement détecté dans la file de type événement (autrement dit la prise en charge de l'événement). L'ordonnancement des événements selon la politique de leur composition et de leurs priorités est opéré en troisième lieu. Une fois les événements ordonnés le système déclenche, dans une quatrième phase, les règles concernées par chaque événement intercepté. L'ensemble des règles déclenchées par chaque événement seront organisées selon leurs priorités (ordonnancement des règles) ou un ordre implicite dans la cinquième phase de ce cycle. La sixième phase évalue la condition de chaque règle déclenchée. Et finalement, la dernière phase exécute l'action de chaque règle déclenchée pour laquelle la condition est vérifiée.

L'état d'une règle est caractérisé par trois phases distinctes selon l'état de ses composantes : l'état de l'événement, l'état de la condition et l'état de l'action. Une règle dont le type d'événement est instancié prend la désignation de règle déclenchée. Une règle dont la condition est vérifiée prendra la désignation de règle exécutable. Et enfin, une règle, dont l'action a été exécutée avec succès, est appelée règle exécutée.

Définir une sémantique d'exécution pour une règle consiste à fixer les valeurs des paramètres qui définissent les transitions de son cycle de vie. Définir la sémantique d'exécution d'un ensemble de règles consiste, en plus, à considérer quelques paramètres globaux tels que l'interaction entre les règles (règles concourantes ou règles en cascade).

Le comportement des règles dans tout système actif, et en particulier une base de données active, est caractérisé par le modèle d'exécution des règles. Chaque modèle d'exécution est caractérisé par un ensemble de paramètres qui peuvent être instanciés avec différentes valeurs pour une application

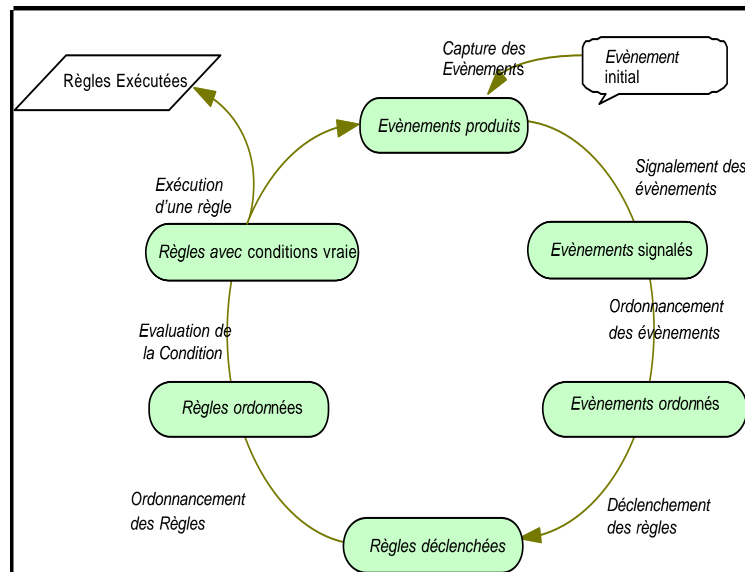


Fig. 3.1 – Cycle d'exécution d'une règle.

donnée. Ces paramètres formalisent une liste de choix importants qui devraient être déterminés avant d'exécuter un programme actif. Les différentes combinaisons de ces paramètres caractérisent le modèle employé par chaque système actif. En conséquence, il y a autant de modèles que de combinaisons de paramètres, et la plupart des systèmes courants ont différents modèles [29]. Les catégories de paramètres les plus importantes, qui définissent ces modèles, sont synthétisées dans les paragraphes suivants :

1. **Les sémantiques d'exécution d'une règle :**

   – **paramètres de gestion des événements** : un modèle d'exécution doit explicitement déclarer quand les événements sont détectés (la fréquence d'écoute), où ils sont détectés (la source), comment ils sont détectés (le processus) et quand ils sont signalés au système d'exécution des règles (la mise en file d'attente). Il doit également indiquer comment des événements peuvent se composer et comment ils sont organisés. Selon chaque type d'événement, une fonction d'agrégation peut produire un événement composé qui servira réellement à déclencher des règles. La composition peut concerner des événements de différentes classes (événements complexes) ou des événements de la même classe (effet réseau). La consommation d'événement indique si des événements sont détruits. Elle doit aussi préciser le moment où les événements seront détruits ou s'ils seront réutilisés après leur considération.

   – **paramètres d'évaluation d'une condition** : avant d'évaluer une condition, le système actif doit savoir sur quelles données devrait porter l'évaluation. La condition peut concerner les objets indiqués dans la composante événement, les objets persistants dans le système ou l'objet dans le moniteur actif. Une condition peut-être évaluée juste après le signalement de l'événement ou reportée à un autre moment qui devrait être indiqué par un paramètre spécifique. Ce paramètre est défini par le couplage $E - C$. Deux modes à considérer le mode

immédiat et le mode *différé*.

- **paramètres d'exécution d'une action** : comme pour la condition, une action devrait avoir un contexte dans lequel elle sera exécutée (les données liées à la condition ou une autre donnée). Un mode de couplage $C - A$ (à quel moment l'action est exécutée si la condition est vérifiée) est alors défini. Dans ce type de couplage, deux modes sont considérés : le mode *immédiat* et le mode *différé*, qui précisent le moment d'exécuter l'action par rapport à l'évaluation de la condition.
  En plus des deux modes de couplage $E - C$ et $C - A$, il existe un autre mode qui est le mode *découplé*. Dans ce mode la prise en charge de la condition est découplée par rapport à l'événement. De même, la prise en charge de l'action est découplée de la condition.

2. **Les sémantiques d'exécution d'un module de règles :**

- **paramètres globaux** : l'exécution d'une action peut produire plusieurs événements qui à leur tour déclencheront d'autres règles (exécution en cascade des règles). Dans ce cas, il est nécessaire d'indiquer si les règles déclenchées sont considérées juste après la génération de chaque événement (exécution *récursive* ou en *profondeur*) ou à la fin de l'action (exécution *itérative* ou en *largeur*). Un autre paramètre global concerne la gestion d'un ensemble de conflits déclenché par une occurrence donnée d'un événement. Les règles composant l'ensemble de conflits peuvent être exécutées en parallèle ou dans un ordre séquentiel. Le premier cas stipule une indépendance entre les règles pour assurer une exécution saine. Dans le dernier cas, il est impératif d'indiquer dans quel ordre (induit par la priorité attachée aux règles, à un choix aléatoire ou à l'ordre d'écriture pour garder une cohérence de la base de données ou tout autre système actif).

Ces paramètres sont récapitulés dans Table 3.2. Ils peuvent servir de directive pour définir la sémantique opérationnelle pour tout moniteur actif. D'autres paramètres peuvent être spécifiés comme les sémantiques orientées instance/ensemble et l'*effet réseau* des instances d'événements n'apparaissent pas dans cette table. Ces paramètres sont considérés comme des méthodes spécifiques pour définir des événements complexes en utilisant des fonctions d'agrégation définies par le concepteur.

3.4 La modélisation à base de règles actives

Si les systèmes basés sur les règles actives sont puissants, il reste au concepteur de définir et de maintenir un ensemble de règles cohérent. En effet, le comportement d'une application peut être décrit par des dizaines ou des centaines de règles ayant leur propre sémantique et pouvant agir l'une sur l'autre. Le concepteur de l'application active doit faire face à beaucoup de difficultés :

(i) comprendre la diversité des sémantiques d'exécution qui sont fournies par différents systèmes de règles actives et leur incidence sur l'exécution globale des règles,
(ii) indiquer un ensemble de règles cohérentes en ce qui concerne les besoins de l'application,
(iii) garantir l'exécution correcte, en particulier son arrêt définitif et la production des résultats prévus,

	Nom du Paramètre	**Valeurs du Paramètre**	**Définition**
Gestion de l'événement	Détection de l'événement	Défini par l'utilisateur	
	Signalisation de l'événement	Défini par l'utilisateur	
	fonction Agrégation	Défini par l'utilisateur	
	Consommation de l'événement	Toujours	Chaque occurrence de l'événement est détruite après sa prise en compte, indépendamment du résultat de la vérification de la condition
		Jamais	Les occurrences d'événements restent indéfiniment dans la file d'attente des événements et sont toujours signalés
		Vérifié	Les occurrences d'événements sont détruites seulement quand des conditions sont vérifiées, autrement elles restent indéfiniment
Évaluation de la Condition	Mode couplage E-C	Immédiat	La composante condition d'une règle est immédiatement évaluée après signalisation de l'événement
		Différé	La composante condition d'une règle est évaluée à la fin du programme d'application ou de la transaction
Évaluation de l'Action	Mode couplage C-A	Immédiat	La composante action d'une règle est exécutée quand la condition est vérifiée.
		Différé	La composante action d'une règle est exécutée à la fin du programme d'application ou la transaction si la condition est vérifiée.
Paramètres globaux	Priorités des règles	Aucune	Aucune priorité est affectée aux règles, elles sont traitées selon leur ordre arrivée.
		Globale	Une priorité globale est affectée aux règles, en respectant un ordre global entre elles.
		Relative	Une priorité relative est affectée aux règles, en respectant un ordre local entre elles.
		Aléatoire	Les règles sont commutatives, donc un ordre indépendant.
	Exécution en cascade	Itérative	Une exécution en cascade des règles obéissant un ordre en largeur (à la fin d'une action toutes les règles déclenchées sont considérées en même temps)
		Récursive	Une exécution en cascade des règles obéissant un ordre en profondeur (après chaque génération d'un événement ou à la fin d'une action, seules les règles déclenchées par le même événement sont considérées)

TAB. 3.2 – Liste globale des paramètres sémantiques pour l'exécution des règles

(iv) fournir une exécution optimale qui réduit la surcharge dans le moniteur actif,
(v) éviter tout cycle dans le déclenchement mutuel des règles,
(vi) prévoir un ordre ou une priorité d'exécution des règles qui ne perturbe pas l'état des données ou du système.

Le comportement d'un ensemble de règles peut être imprévisible et souvent non déterministe. L'arrêt du processus des règles est l'un de ces problèmes ; l'exécution en cascade des règles peut produire des cycles infinis. La gestion des conflits est un autre problème complexe ; si l'ensemble de conflits de règles n'est pas ordonné, beaucoup de plans d'exécution pourraient être envisagés et le résultat peut différer de ce qui est prévu, surtout en ce qui concerne l'ordre des lectures et des écritures qui peut donner des états du système très différents.

Les problèmes fondamentaux de l'uniformité, de l'arrêt et de la concurrence demeurent difficiles dans les règles actives aussi bien que dans les programmes parallèles. Ce sont encore des problèmes complexes, et même des problèmes indécidables. Les solutions sont souvent données pour les cas très restreints qui, en général, pourraient être résolus sans le besoin d'une théorie sophistiquée comme l'utilisation des verrous ou l'exclusion mutuelle dans les sections critiques. Les programmes écrits en langages classiques (C, Java, ...) s'exécutent correctement dans la plupart des cas. Ils sont considérés comme stables et souvent commercialisés. Cependant, leur comportement n'a pas été formellement démontré. Ils sont habituellement programmés en utilisant la programmation procédurale (programmation structurée, ou orientée objet) et sont alors informellement validés par des tests. De même, des programmes à base de règles actives pourraient être informellement validés par simulation et correction du type essai-erreur. Ces deux modes de validation peuvent être intégrées dans un environnement qui fournit un ensemble d'outils pour visualiser, vérifier, simuler et corriger les programmes actifs.

Des approches plus pragmatiques pour l'analyse et la conception des règles actives ont été proposées dans [63]. Elles visent à soutenir le processus itératif de la définition des règles en :

(i) mettant en place la sémantique d'exécution des règles,
(ii) traçant l'exécution des règles,
(iii) isolant les sections critiques avec des conflits potentiels ou des cycles infinis,
(iv) changeant la définition de la règle ou de la sémantique d'exécution et
(v) refaisant des tests pour les nouvelles spécifications.

Comme mentionné précédemment, nous devons noter que ces problèmes ne sont pas spécifiques aux règles actives, mais concernent tout type de programmation et tout type de langage.

Grâce à leur modularité et à la flexibilité de leurs modèles d'exécution, les systèmes actifs basés sur les règles actives fournissent un appui idéal pour l'évolution des applications ; leur sémantique explicite facilite l'analyse et le raisonnement sur le comportement des applications et la cohérence de leur exécution.

3.5 Les avancées et les domaines de recherche sur les règles actives

Beaucoup de résultats valables ont été obtenus à partir des recherches effectuées dans la communauté de bases de données. De la fin des années 80 et jusqu'au milieu des années 90, beaucoup de systèmes actifs de bases de données ont été conçus [95]. Chacun ayant son propre modèle de données (relationnel ou orienté objet), son propre langage de spécifications de règle et son propre modèle d'exécution. Dans la deuxième partie des années 90, les systèmes paramétrables ont été proposés par [29] pour alléger la disparité entre les besoins des applications et des systèmes actifs spécifiques. Cette approche a été critiquée pour son manque de performance. Une approche, qui a succédé à l'approche des systèmes paramétrables est celle de la composition de modules actifs qui fournit un ensemble de modules de base permettant à chaque créateur d'application de définir le

système actif qui répond le mieux aux besoins des applications. Une directive sur les systèmes à base de règles actives a été convenue dans le réseau de recherche européen ACTNET et publié dans [55]. Différentes expériences ont évalué l'utilisation des règles actives dans des domaines d'application spécifiques [27] et les publications récentes ont montré comment des règles d'actives peuvent être intégrées dans les documents XML [51]. Une bibliographie avancée sur les systèmes à base de règles actives a été constituée dans [95][76] et dans la série d'ateliers RIDS (Rules in Database Systems), [77][81][34].

3.6 Conclusion

Les règles actives sont un modèle de programmation adapté à tout système événementiel ou toute application qui peut être découpée en plusieurs modules déclenchables en leur associant un événement précis.

Une règle active peut être exprimée sous forme de document XML. Il faut noter que ce type de document trouve des applications dans plusieurs domaines tels que le web et les bases de données, en passant par les fichiers de configuration des systèmes.

L'un des domaines potentiels où l'application des règles actives peut apporter un progrès est le domaine des réseaux et en particulier les réseaux actifs. La particularité de ce type de réseau est la possibilité d'exécuter des programmes en réaction à des situations qui se produisent au sein du réseau.

Dans le chapitre suivant nous montrons comment nous procédons à l'intégration des règles actives dans les réseaux actifs afin de donner une nouvelles formalisation des applications pour quelles s'adaptent mieux aux spécificités du réseau.

Chapitre 4

ARFANet : Active Rules Framework for Active Networks

4.1 Introduction

Dans le chapitre 2 nous avons pu constaté que les architectures actuelles offrent un simple environnement d'exécution aux programmes actifs. Ces programmes sont écrits dans des langages dédiés sous formes d'objets ou de procédures. Cette forme compacte des programmes est dépouvue de tout caractère événementiel et ne permet aucun contrôle par le nœud. Le code actif, par sa nature, n'est déclenchable que par l'arrivée d'un paquet de données. Ainsi des événements survenants au sein du réseau comme la formation d'un goulot d'étrangement sont difficiles à intercépter dans ce genre d'environnement.

Créer une infrastructure qui englobe la notion d'événement autour de codes actifs écrits dans les langages actuels n'est pas suffisante et risque d'être peu performante. Il est, de notre avis, plus judicieux de créer ou d'adapter un formalisme existant qui soit sensible aux événements.

Dans le chapitre précédent nous avons présenté un formalisme basée sur les événements. Les règles actives, créées dans des domaines tels que les bases de données et les systèmes experts, ont fait l'objet d'études poussées qui ont abouti à une structuration harmonieuse à base d'événements. Cette structure comporte, pour toutes les applications, trois entités liées : l'événement comme source de déclenchement, une condition pour tester l'état et enfin une action pour appliquer le traitement adéquat. Les études menées dans le domaine ont abouti à la définition d'un moniteur actif qui doit être capable de traiter chaque partie d'une règle. Pour l'événement le moniteur fournit des mécanismes de détection, d'agrégation et l'ordonnancement. Pour la condition il fournit un environnement d'évaluation. Enfin pour l'action il implémente un environnement d'exécution. Ceci suggère que le moniteur actif est une infrastructure qui englobe des environnements d'exécution capable de traiter des événements avec des formalismes bien définis qui sont les règles actives.

Les caractéristiques des règles actives telles que la *séparabilité*, la *modularité*, l'*évolutivité* et l'*extensibilité* sont importantes dans un réseau actif. La séparabilité est au niveau de la distinction des différentes parties de la règle et concerne la connaissance mise dans les règles actives et leurs sémantiques d'exécution. La modularité réside dans la définition de modules indépendants qui peuvent coopérer pour la création d'autres applications. L'évolutivité est permise par la considération de la plus petite granularité d'une application qui est la règle. Par la séparabilité et la modularité, les applications à base de règles actives sont facilement extensibles.

Ce chapitre vise à exploiter les résultats de recherche sur les règles actives obtenus dans les domaines des bases de données actives et de l'intelligence artificielle, et à appliquer les techniques correspondantes au développement des réseaux actifs. Les règles actives, telles que définies dans ces domaines, ne peuvent être appliquées directement dans les réseaux actifs. Il est nécessaire d'adapter la technologie des règles actives au contexte des réseaux actifs.

La première adaptation concerne la composante condition, qui est habituellement considérée comme une requête dans le contexte des bases de données : dans le contexte des nœuds actifs, la composante condition sera considérée comme une fonction booléenne définie à partir des méta-données contenues dans le nœud ou dans les données transportées par les paquets du réseau.

La deuxième adaptation réside au niveau du contexte transactionnel qui garantit l'atomicité des opérations dans les bases de données : ce concept devrait être remplacé par la notion de *session active* dans le nœud.

La troisième adaptation réside dans la base de données elle-même : bien qu'un nœud actif puisse avoir un certain système de cache pour contrôler la persistance des données, il n'a pas nécessairement toutes les fonctionnalités de stockage d'un système de base de données.

Enfin les types d'événements et les types d'actions seront définis en accord avec les services fournis par les nœuds actifs et les besoins des utilisateurs.

Dans la prochaine section nous décrivons les quelques spécificités qui caractérisent un réseau actif, et qui doivent être prises en compte lors de la conception de tels réseaux. La section 4.3 est une description globale de notre infrastructure ARFANet et les phases de son déploiement sont résumées dans la section 4.4. La couche application de notre architecture est définie dans la section 4.5. La section 4.6 et la section 4.7 définissent respectivement les deux couches de base à savoir le moniteur actif et la machine virtuelle. Le format de paquet propre à ARFANet est présenté dans la section 4.8. Et enfin la section 4.9 conclut ce chapitre.

4.2 Les spécificités des réseaux actifs

Un réseau actif est un réseau où une nouvelle fonctionnalité propre à un utilisateur ou spécifique à une application peut être pré-chargée dans les nœuds du réseau ou transmise dans des paquets les traversant, sous forme de méthodes ou de petits programmes. Ces méthodes sont semblables aux méthodes utilisées dans la programmation orientée objet qui sont des procédures exécutables incluses dans des objets.

L'extension des réseaux standards ou *passifs* vers des réseaux plus intelligents vise à transformer des réseaux dont le rôle est uniquement l'acheminement des données vers ceux qui, en plus du transport des données, appliquent un traitement actif sur les données.

Un réseau actif est tout réseau de transmission qui devrait répondre aux critères suivants :

1. Un réseau actif doit au moins fournir les mêmes services que n'importe quel réseau de transmission. Chaque nœud devrait pouvoir envoyer, recevoir et router des paquets de données.
2. Un réseau actif doit fournir des outils et des mécanismes nécessaires aux nœuds pour définir et exécuter des règles actives.
3. Un réseau de transmission est considéré comme actif si au moins un de ses nœuds est actif, donc équipé de services actifs.
4. Un réseau actif doit fournir les équipements flexibles afin d'implémenter dans ses nœuds tout service d'application distribué. Il devrait en particulier permettre à des fournisseurs de services

de publier et de déployer leurs programmes actifs et de les adapter selon un modèle d'exécution et une interface spécifiques.

5. Un réseau actif doit fournir les fonctionnalités qui permettent à des utilisateurs de choisir les services réseaux qui sont appropriés à leurs besoins d'applications.
6. Un réseau actif doit fournir des fonctionnalités de réadaptabilité qui devraient en particulier faciliter la redéfinition et l'évolution des services, et la reconfiguration des ressources.

Ces critères sont nécessaires pour la définition de tout réseau actif et en particulier celui qui est basé sur les règles actives. Néanmoins, l'absence d'un critère peut violer les propriétés fondamentales d'un réseau actif.

4.3 ARFANet : Une nouvelle infrastructure des réseaux actifs

ARFANet (Active Rule Framework for Active Networks) est une infrastructure réseau actif qui implémente les fonctionnalités des règles actives ou ECA (Evénement-Condition-action). Pour réaliser une telle architecture, nous avons besoin de définir les composantes nécessaires au traitement des règles actives.

Le principe de base d'un réseau actif consiste à fournir les équipements de déclenchement qui permettent à certains nœuds (les nœuds actifs) de réagir aux événements externes véhiculés dans des paquets de données ou produits par l'environnement du système d'exploitation du nœud. Pour réaliser un tel réseau, il faut mettre en place des moyens implémentant dans chaque nœud actif un moniteur actif basé sur la sémantique des règles ECA. Cependant, comme nous avons pu le voir précédemment, il est possible d'avoir autant de moniteurs actifs que de combinaisons des valeurs des paramètres sémantiques. Développer un moniteur actif paramétrable et unique, qui implémente différentes sémantiques, n'est pas viable car cela ne permet pas, en général, des performances satisfaisantes [29]. Par conséquence, nous considérons qu'un nœud actif peut être équipé d'un ou de plusieurs moniteurs actifs, chacun d'eux étant consacré à une classe d'applications exigeant la même sémantique d'exécution. Les moniteurs actifs sont dynamiquement chargés dans un ou plusieurs nœuds actifs. Un réseau actif dont les nœuds sont équipés de moniteurs actifs est alors prêt à traiter les règles actives qui définissent le comportement des applications.

Un *service d'application* est modélisé comme un ensemble de règles actives qui seront exécutées par un moniteur actif spécifique. Comme les moniteurs actifs, les services d'application sont dynamiquement chargés (publiés) dans les nœuds actifs. Le lien entre un service d'application et le moniteur actif correspondant est établi au moment où le service d'application arrive au nœud. Le lien entre un service d'application et un moniteur n'est pas figé au moment de leur définition. Cette mise en correspondance différée permet un découplage entre le déploiement des services d'application et le déploiement des moniteurs actifs. Ce découplage peut être exploité pour garantir une meilleure sécurité, car il nous permet de concentrer le mécanisme de sécurité sur les services actifs.

A ce stade, nous avons vu qu'un nœud actif peut contenir un ou plusieurs moniteurs actifs et un ou plusieurs services d'applications. Afin de permettre le chargement de ces deux genres de composantes, nous avons besoin d'une autre composante qui doit être assez générique pour ne pas dépendre d'un certain type de moniteur actif ou de service d'application. Elle doit aussi cacher aux deux précédentes couches les spécificités matérielles et logicielles des nœuds. Cette composante est considérée comme une machine virtuelle qui a les mêmes fonctionnalités dans chaque nœud actif, mais qui dépend de l'environnement système du nœud actif. La *machine virtuelle* est mise en place une fois pour toutes dans chaque nœud du réseau au-dessus du système d'exploitation

du nœud. Un nœud équipé d'une machine virtuelle n'est pas automatiquement un nœud actif. Il peut-être considéré comme nœud standard aussi longtemps qu'il ne reçoive pas un paquet de service le rendant actif. Ce paquet de service peut être un paquet de moniteur actif. L'utilisation d'un tel concept de machine virtuelle nous permet de définir une architecture où les nœuds actifs peuvent être déployés dynamiquement, en fonction, par exemple, de l'état de congestion du réseau.

Nous pouvons également généraliser la notion de système actif à la machine virtuelle. En effet, la machine virtuelle d'un nœud reçoit des événements (divers paquets de données) et exécute des actions (charge et implémente des moniteurs actifs, lie des services d'application aux moniteurs actifs ou juste route les paquets) sous certaines conditions (résultats de l'analyse des en-têtes des paquets). Un moniteur actif apparaît bien comme un service d'application spécifique et constitué d'un ensemble de règles actives. Cette vue est la base de notre infrastructure. Elle fournit une flexibilité élevée pour la reconfiguration du réseau et une large ouverture pour son évolution.

En résumé, un nœud actif se compose de trois types de ressources (Fig 4.1) :
- une machine virtuelle,
- un ensemble de moniteurs actifs
- un ensemble de services d'application pour chaque moniteur.

Fig. 4.1 – Les composants d'un nœud actif.

Il faut noter que l'architecture proposée est une architecture à trois couches comme celle proposée par [13]. Le service d'application peut être considéré dans l'architecture standard comme l'application active. L'environnement d'exécution a pour rôle la gestion des AA, de même le moniteur actif de notre architecture a pour rôle la gestion d'une classe de services d'application. Notre machine virtuelle peut être étendue pour englober toutes les fonctionnalités du NodeOS comme la gestion de plusieurs environnements d'exécution.

Au moment où un nœud du réseau devient actif, tout moniteur dans sa base devient capable d'être lié aux services d'application par la machine virtuelle. Une fois le lien établi, le service d'application est prêt à détecter des événements d'application et à réagir à ces événements selon la sémantique d'exécution des règles choisie et mise en place par le moniteur après la mise en correspondance différée avec le service d'application.

Bien que le nœud actif de base soit défini par trois couches, il est toujours possible d'ajouter d'autres couches. Chaque couche inférieure est vue comme une sorte de moniteur actif pour la couche supérieure. Ceci nous permet de définir une hiérarchie des services d'application, s'étendant du niveau le plus primitif jusqu'au niveau le plus élevé (Fig 4.2). Si le flux de données peut être hiérarchisé en plusieurs classes, chaque classe peut alors jouer le rôle d'un genre de service applicatif pour la classe supérieure.

Un réseau actif est alors un réseau de transmission ayant un ou plusieurs nœuds actifs. Chaque nœud est équipé d'un ou de plusieurs moniteurs actifs. À chaque moniteur actif, un ou plusieurs services d'application peuvent lui être liés. Le figure Fig 4.3 décrit un exemple de réseau actif où certains nœuds sont seulement des nœuds de communication (des nœuds passifs implémentant

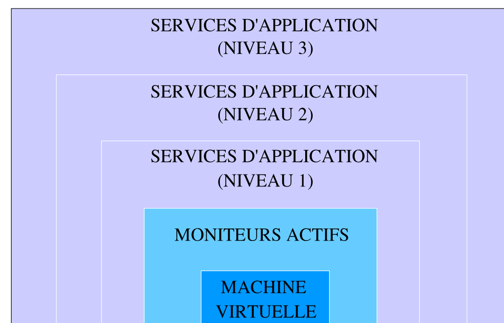


Fig. 4.2 – Un nœud actif avec une hiérarchie de services d'application.

seulement une machine virtuelle), certains sont actifs (ils contiennent au moins un moniteur actif) et d'autres sont actifs et liés (dans ce cas, il existe au moins un service d'application auquel un moniteur actif est assigné). Le statut d'un nœud n'est pas indéfiniment fixe, il peut évoluer au cours du temps, étant alternativement non actif, actif, ou actif et lié, selon l'utilisation et l'état du réseau ou le besoin des utilisateurs.

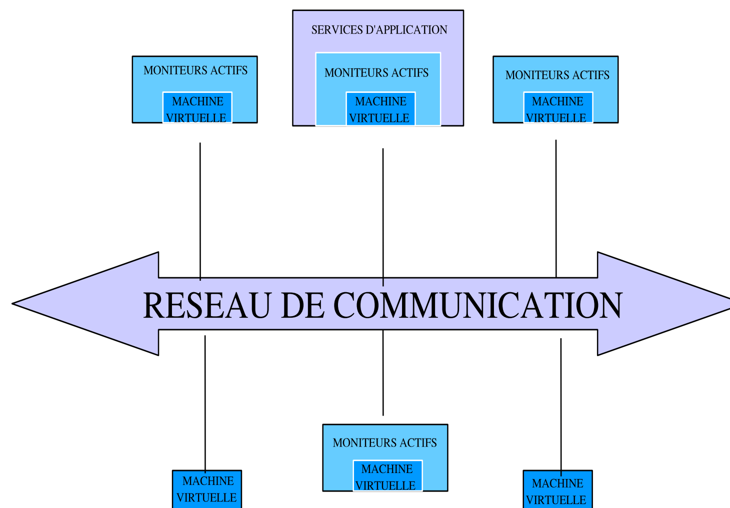


Fig. 4.3 – Un réseau actif avec plusieurs nœuds.

Dans un réseau actif, nous pouvons distinguer trois types d'information traversant le réseau :
- les paquets encapsulant les moniteurs actifs (m-paquets),
- les paquets encapsulant les services d'application (s-paquets),
- les paquets encapsulant les données régulières (d-paquets).

Les paquets qui ne sont pas actifs, autrement dit les paquets passifs ou qui n'appartiennent pas aux trois précédentes catégories de paquets actifs, sont simplement routés.

La publication des ressources se rapporte au processus d'ajout des moniteurs actifs ou des services d'application au réseau. La publication d'une ressource signifie mettre en application cette ressource dans un emplacement approprié (ou une liste d'emplacements) dans le réseau. L'une des principales problématiques des réseaux actifs est la sécurité qui est rendue plus sensible par le

processus de publication. Bien que, notre architecture de nœuds actifs, ne résout pas le problème de sécurité par lui-même, elle fournit une hiérarchie des rôles (éditeurs des services actifs, éditeurs des services d'application et utilisateurs finaux). Cette hiérarchie peut renforcer la sécurité au niveau de l'interface entre deux couches de l'architecture en appliquant des contrôles avec un genre de pare-feu spécifique. Cela dit, nous proposons dans le chapitre 7 une architecture globale qui permet d'avoir une publication du code actif plus sure et une exécution plus fiable des programmes actifs dans les nœuds du réseau.

4.4 Le déploiement des services actifs

La publication de ressources consiste à définir et à enregistrer des services actifs dans le réseau. Ce processus peut être fait en quatre phases (voir figure Fig 4.4) :

- Phase 0 : publier la machine virtuelle. Ceci correspond à l'installation initiale de réseau avec les fonctionnalités minimum d'acheminement ainsi que les possibilités d'analyser des paquets, de charger et lier des programmes.
- Phase 1 : publier les moniteurs actifs en envoyant les paquets définissant les moniteurs actifs à travers le réseau. Le processus de publication déclenchera la machine virtuelle et, selon les en-têtes des paquets (conditions), installera le moniteur actif transporté par le paquet d'information.
- Phase 2 : publier les programmes des utilisateurs (services d'application) et répandre les s-paquets à travers le réseau. Le processus de publication déclenchera le moniteur actif correspondant dans chaque nœud et installera les services d'application dans les emplacements appropriés.
- Phase 3 : publier les données qui seront utilisées par les services d'application. Le processus de déploiement répandra les données à travers le réseau et déclenchera tous les services d'application appropriés concernés par ces données.

Chaque phase de ce processus de publication peut-être faite par un acteur différent ou le même acteur jouant différents rôles. Un acteur peut-être une personne (un administrateur de réseau, un fournisseur d'accès Internet ou un programmeur) ou un processus (un programme d'application).

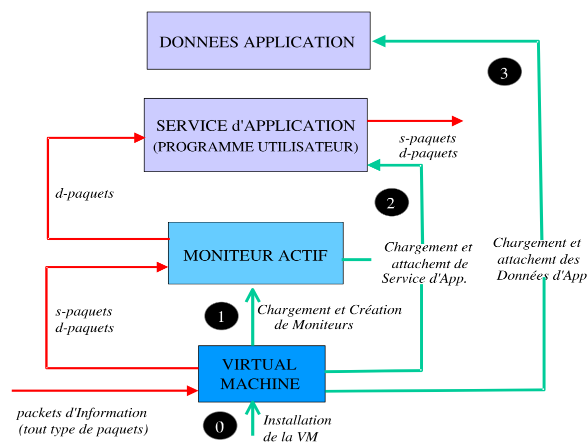


FIG. 4.4 – Les phases de déploiement des services active

Comme elles sont définies actuellement, l'approche discrète et l'approche intégrée peuvent être facilement mises en œuvre en utilisant notre architecture. Nous pouvons noter que la différence entre ces dernières approches apparaît seulement dans la façon avec laquelle le code est déployé dans le réseau. En effet, les phases du déploiement d'un service sont les mêmes dans les deux approches et diffèrent seulement dans l'ordre dans lequel elles sont exécutées. Ces phases sont semblables à celles que nous avons décrites ci-dessus.

Les prochaines sections détailleront les trois couches qui composent un nœud actif dans ARFA-Net.

4.5 Description d'un service d'application

Chaque service d'application est défini comme un ensemble de règles actives pour s'exécuter sous une certaine sémantique indiquée comme méta-données sur les règles.

Les paragraphes suivants définissent, dans le contexte des réseaux actifs, les composantes principales d'une règle active (événement, condition et action) ainsi que les méta-données qui définissent leur sémantique et leur organisation au sein même d'une application en plusieurs modules.

4.5.1 La spécification d'un événement

Définition 1 : un événement est tout signal discret qui permet de déclencher une réaction d'un nœud actif en exécutant un service d'application correspondant. Nous distinguons les types d'événement (ou les classes) et les occurrences d'événement (ou les instances). Un type d'événement est défini par son nom, ses attributs et les contraintes possibles sur ses valeurs. Une instance d'un événement est définie par une référence d'événement (*evid*) et une valeur (ou les valeurs assignées à la liste de ses attributs).

Exemples : l'arrivée d'un paquet à un nœud (ses attributs sont la description du paquet), le ACK, NACK ou l'ACK-Timeout qu'un nœud peut recevoir.

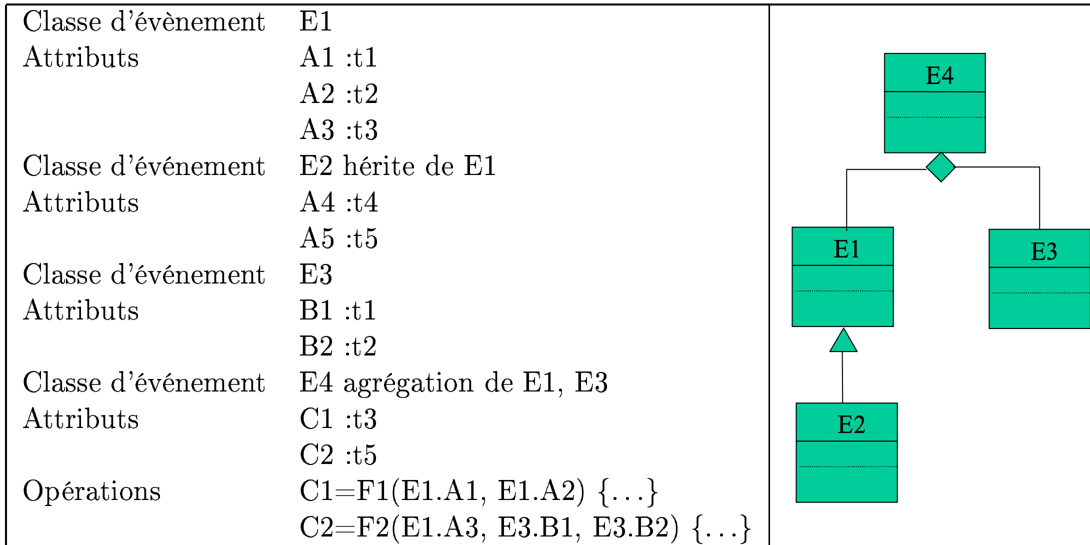
Plus généralement, les événements peuvent être de différentes catégories : événements temporels, événements de flux de données, événements du système, événements de transition d'état du nœud ou du réseau, etc.

Définition 2 : l'identifiant d'un événement (evid) : pour identifier les instances d'un événement, nous employons habituellement des horodateurs, qui est le moment auquel une occurrence d'événement a été détectée ou signalée au moniteur actif. Cette manière d'identification introduit un ordre total des instances d'un événement indépendamment de leurs types et des sources qui les ont produites. Nous combinons l'identifiant de la source ou l'identifiant de la classe d'événement avec le moment de l'instantiation de l'événement pour avoir l'*evid*. Cette hypothèse d'avoir un ordre total est tout à fait raisonnable car il est toujours possible, quoique difficile dans un réseau, de définir une échelle de temps (ou une horloge) locale ou globale.

La définition des types d'événement est semblable à la modélisation d'un ensemble de classes d'objets. Des types d'événement peuvent être organisés en hiérarchie et peuvent être agrégés pour former un nouvel événement de type complexe. Pour plus d'exemples de spécifications de type d'événements, reportez vous à la Table 3.2.

Dans la composition des événements, nous utilisons en plus de la composition ou l'hiérarchie de classes, une composition par expression booléenne selon la grammaire suivante :

- $E \leftarrow E \;\&\; E$: événement1 et événement2



Classe d'évènement	E1
Attributs	A1 :t1
	A2 :t2
	A3 :t3
Classe d'événement	E2 hérite de E1
Attributs	A4 :t4
	A5 :t5
Classe d'événement	E3
Attributs	B1 :t1
	B2 :t2
Classe d'événement	E4 agrégation de E1, E3
Attributs	C1 :t3
	C2 :t5
Opérations	C1=F1(E1.A1, E1.A2) {...}
	C2=F2(E1.A3, E3.B1, E3.B2) {...}

TAB. 4.1 – Exemple de la spécification d'un type d'événement.

- $E \leftarrow E \mid E$: événement1 ou événement2
- $E \leftarrow !E$: non événement1
- $E \leftarrow (E)$: (événement1)
- $E \leftarrow A$: opérande (initialement un événement)

Pour avoir des expressions complexes telles que : $((e1 \ \& \ e2) \mid (!e1 \ \& \ e3))$, où e1, e2 et e3 sont des événements.

4.5.2 La spécification d'une condition

Définition 3 : une expression conditionnelle d'une règle ECA est une expression logique ou une fonction booléenne qui opère sur différentes variables telles que des attributs d'événement, des données persistantes dans le nœud actif ou toute variable de l'environnement (variable ou méta-données du système).

Exemple : une condition simple peut examiner si un paquet a été envoyé ou non, s'il se trouve toujours dans le cache ou non, etc.

Si la condition est une formule logique, elle doit être définie directement dans la règle active. Si la condition est une fonction booléenne, seule son invocation est mentionnée dans la règle active ; nous supposons que la définition de la fonction qui calcule la condition est faite en amont en utilisant un langage de programmation adapté ou dédié aux réseaux actifs.

Nous utilisons la grammaire de l'expression booléenne des événements pour la composition des conditions.

4.5.3 La spécification d'une action

Définition 4 : une action d'une règle peut être toute action que le nœud actif peut exécuter pour la communication de données, la gestion de réseau, fournir un service d'application ou simplement

un traitement sur les données de l'utilisateur.

Exemples : envoi d'un paquet, envoi d'un ACK ou un NACK, appel d'une procédure d'attribution de ressource, application d'une fonction de reconfiguration de réseau, une invocation locale ou à distance d'un service d'application, etc.

Les actions sont définies en utilisant des langages de programmation évolués tels que C et Java, ou des langages dédiés à l'application des règles actives aux réseaux actifs. C'est seulement une invocation des actions qui est indiquée dans les règles actives.

Contrairement à l'application des règles actives dans les bases de données, une condition ou une action ne peuvent être formulées dans un format déclaratif, seulement dans un format encapsulé. Car dans les bases données il est possible de lancer une requête SQL qui est par définition dans un format déclaratif. Toutefois, il est possible que des actions soient constituées d'un ensemble d'actions élémentaires et que ces actions peuvent utiliser les mêmes actions élémentaires. Les sous-actions peuvent s'exécuter en série comme en parallèle. La formulation A_1 ; A_2 représente deux sous-actions qui s'exécuteront en série suivant l'ordre, tandis que $A_1 \,||\, A_2$ est utilisée pour les sous-actions parallèles. Les actions peuvent être limitées pour prévenir des exécutions engendrant des dysfonctionnements dans les nœuds. Ces actions couplées peuvent être formulées sous cette forme déclarative et c'est le moniteur qui doit organiser l'exécution des sous-actions selon le mode défini.

4.5.4 La spécification d'une règle

Il est commun de trouver deux règles ou plus qui partagent des conditions ou des actions. Ceci est déjà valable pour les événements car dans la spécification des événements nous avons introduit la notion d'événements complexes qui est le résultat de l'agrégation des événements primitifs. Une condition définie en tant qu'expression booléenne est considérée comme complexe. Les actions aussi peuvent être composées d'actions plus élémentaires. Pour cela, nous avons préféré définir un service d'application en tant que liste d'événements, une liste de conditions, une liste d'actions et enfin une liste de règles. Chaque événement, condition, action et règle ont un identifiant unique. Chacun d'eux peut être utilisé plusieurs fois et combiné avec d'autre éléments du même type dans des fonctions d'agrégation et des expressions. Ainsi, une règle n'est autre qu'un triplet contenant les identifiants de l'événement, de la condition et de l'action ou des fonctions manipulant ces identifiants.

Comme chaque règle possède un identifiant, il est tout a fait possible qu'une règle déclenche une autre en spécifiant simplement sont identifiant.

Exemple de règle complexe :

```
define rule R1
On   E1 & E2
If   C1 | C2
Then A1 ; A2
```

4.5.5 Exemple de service d'application : Communications de groupe ou Multicast

Pour comprendre comment appliquer les règles actives dans les réseaux actifs, nous considérons un exemple de traitement actif de l'implosion des acquittements négatifs dans les groupes du

multicast. Une communication de groupe est une communication qui fait intervenir plus de deux participants désirant échanger des informations [28], [61].

Les réseaux actifs peuvent être utilisés pour fiabiliser les protocoles du multicast et permettent de résoudre de façon élégante les problèmes dus :
- à l'implosion d'acquittements négatifs (NACK),
- aux retransmissions inutiles,
- à la concentration de trafic due aux retransmissions,
- aux duplications de paquets et,
- au fait que le nombre des membres du groupe est fortement dynamique.

Dans la solution "Active Reliable Multicast" (ARM), proposée dans [62], les nœuds vont jouer un rôle actif dans le mécanisme de fiabilité, comme :
- copier les données transmises afin de pouvoir les réémettre au plutôt en cas de requête de retransmission ;
- traiter les NACKs afin d'optimiser les demandes de retransmission et de collecter les informations concernant l'auteur de la demande de retransmission (son débit de réception, ...) ;
- et aussi la détection par le routeur de la perte d'un paquet pour anticiper toutes demandes de retransmission.

Dans un système de multicast basé sur une gestion active, quand un paquet arrive dans un nœud, si le nœud a un cache, ce paquet est stocké avant d'être communiqué à tous les abonnés du groupe. Si un paquet est perdu pendant la transmission, le récepteur du paquet envoie un message NACK qui indique que le paquet n'est pas arrivé et devrait donc être retransmis.

L'arrivée d'un tel paquet (NACK) au nœud actif déclenchera la destruction du NACK si le paquet absent a été déjà retransmis, ou la retransmission de ce paquet s'il est toujours dans le cache du nœud. Dans le cas où le paquet n'est pas dans le cache, le nœud note l'identité de l'utilisateur et transmet le premier NACK qu'il a reçu à la source du paquet perdu. Le paquet est retransmis par la source elle-même ou par le nœud actif sur le lien. Le paquet est supprimé du nœud s'il reçoit tous les ACKs des clients ou le temps aller-retour maximal (RTT) est atteint.

Selon ce modèle de multicast, nous pouvons définir cinq événements possibles qui peuvent se produire dans le système (Table 4.2). Ces événements sont décrits ci-dessous.

1. Evénement NACK : cet évènement correspond à l'arrivée d'un paquet de non acquittement envoyé de la part du client. Le traitement d'un tel évènement déclenchera l'évaluation de deux types de condition :
- Booléen *Presence* : spécifie si le paquet est toujours dans le cache :
  - si le paquet est toujours dans le cache du nœud, alors le nœud le retransmettra au client qui a envoyé le NACK. Cette action est notée $A_1()$.
  - si le paquet a été détruit par le nœud, l'identité du client est enregistrée par le nœud ($A_2()$) et le premier NACK reçu est alors transféré à la source qui a signalé la perte du paquet ($A_3()$).

  Dans les deux cas, le NACK reçu est détruit à la fin du traitement par le nœud.
- Booléen *Sent* : à l'arrivée du NACK, le nœud vérifie si le paquet correspondant a été déjà retransmis au client. Ceci est vérifié en utilisant le booléen *Sent* :
  - si c'est le cas, le NACK n'est pas pris en compte et il est immédiatement détruit ($A_4()$).
  - si le paquet n'a pas encore été retransmis, le traitement correspondant au booléen *Presence* est alors déclenché ($A_5()$).

2. Evénement ACK : cet événement correspond à l'arrivée d'un accusé de réception du paquet de la part d'un client. Le traitement à déclencher dépend du nombre de ACK que le nœud a reçu

pour le même paquet. Soit *total*, ce nombre.

- Entier *total*
  - si le nœud a reçu un ACK de tous les membres du groupe de multicast concernés par le paquet, alors le nœud détruit le paquet ($A_6()$),
  - sinon, le nœud ajoute le nom de l'expéditeur du ACK à sa liste des membres qui ont accusé le paquet ($A_7()$).

3. Evénement PAQUET : un événement de ce type n'a aucune condition à tester pour déclencher le traitement correspondant. A l'arrivée de cet événement, le nœud envoie le paquet, selon l'algorithme de routage utilisé, à la liste des inscrits au groupe se trouvant dans la sous-branche de l'arbre du multicast dont la racine est le nœud ($A_8()$).

4. Evénement ACK-Timeout : cet événement est un exemple d'événements internes que peut produire le système. Il est déclenché par le système d'exploitation du nœud. Cet événement n'a aucune condition à tester pour déclencher le traitement correspondant. Quand un certain délai est écoulé, le nœud suppose que le client n'est plus apte à reçevoir d'autres paquets pour différentes raisons telles que l'interruption ou la congestion du lien, ou simplement parce que le client est déconnecté. Cependant, le nœud doit supprimer le client de sa liste des inscrits ($A_9()$) et transférer cette information au serveur ($A_{10}()$). L'événement *Départ volentaire* du client provoque le même traitement.

5. Evénement GroupJoin : cet événement correspond à l'arrivée, de la part d'un nouveau client, d'une requête pour faire partie du groupe. Le traitement à déclencher dépend du nombre maximal d'inscrits que peut accepter le serveur à un instant donné.

- Integer *MaxClient*
  - si le nombre maximal des membres du groupe de multicast est atteint, alors le nœud rejette la requête du nouveau client ($A_{11}()$),
  - autrement, le nœud ajoute le nouveau client à sa liste ($A_{12}()$) et envoie cette information au serveur ($A_{13}()$).

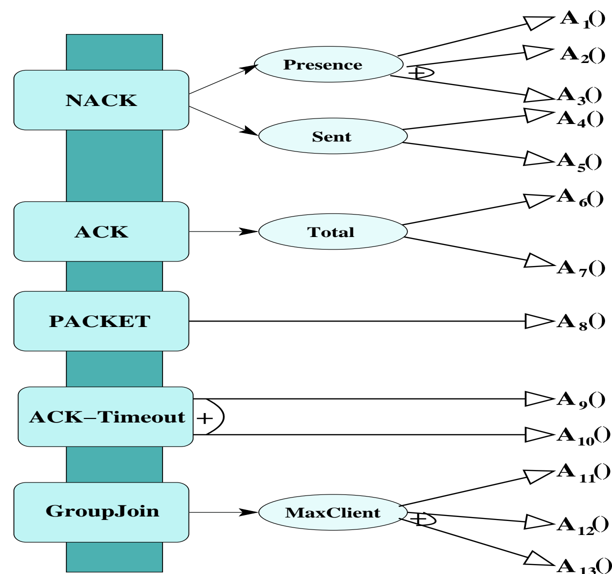


FIG. 4.5 – Le module Multicast

Les règles actives décrivant le modèle du multicast et ses sémantiques d'exécution sont données

Event class	Event **NACK**
Attributes	@source : Adr,
	@multicast-server : Adr,
	id-packet (lost packet) : Int.
Event class	Event **ACK**
Attributes	@source : Adr,
	@multicast-server : Adr,
	id-packet (arrived packet) : Int.
Event class	Event **PACKET**
Attributes	id-packet : Int,
	@multicast-server : Adr,
	@multicast-group : Adr.
Event class	Event**ACK-Timeout**
	(depends on Round Time Trip)
Attributes	@source : Adr,
	RTT : Int.
Event class	Event**GroupJoin**
	(client joins multicast group)
Attributes	@source : Adr,
	@multicast-group : Adr.

TAB. 4.2 – Spécification des types d'événement du module Multicast.

dans Table 4.3. Dans cet exemple, les règles R_1, R_2, R_3 et R_4 ont la même priorité.

Notons que la stratégie de toute application peut être facilement modifiée. Ainsi, nous pouvons, par exemple, ajouter des règles ou inhiber d'autres. Nous pouvons aussi changer les sémantiques d'exécution, par exemple, en changeant la priorité des règles ou la valeur d'un paramètre comme remplaçer la valeur *vérifié* du paramètre *Consommation d'événement* dans l'exemple du multicast par la valeur *Toujours*. Dans ce cas, au lieu de détruire chaque occurrence de l'événement seulement quand la condition est vérifiée, elle est détruite après sa considération même si la condition n'est pas vérifiée.

4.5.6 Modularité des services d'applications

Les règles sont organisées en modules et un service d'application est défini par un ou plusieurs modules. Nous supposons que la sémantique d'exécution est définie de manière identique pour toutes les règles du même module. Ainsi, l'en-tête de chaque module fournit l'ensemble des paramètres sémantiques ainsi que leurs valeurs respectives (Table 4.7).

Toutefois, avoir quelques petits modules spécialisés ou regrouper des règles du même ensemble de conflit dans un seul module peuvent avoir des incidences positives du point de vue performance au lieu d'avoir un seul module global pour toute l'application. En effet le chargement des modules implique le chargement de toutes les règles. Cependant, les règles qui peuvent être exécutées sont uniquement celles dont l'événement a été détecté et les conditions sont vérifiées. Vue la nature discrète des événements, un seul événement est détecté à la fois. En conséquence, toutes les règles seront chargées en mémoire, mais seules quelques unes seront exécutées.

La division d'une application en un ensemble de petits modules doit, au moins, obéir à l'une

des règles suivantes : le respect de la dépendance entre règles actives, le déclenchement des règles appartenant au même ensemble de conflit (autrement dit partageant le même évènement) et les critères de conception définis par l'utilisateur sur les règles à un niveau logique.

La politique que nous avons choisie pour l'établissement des modules est liée aux performances. Ce choix dépend de la granularité du chargement des règles. La granularité du chargement en mémoire est le module, mais la granularité du téléchargement ou le déploiement peut être le module ou l'application toute entière. Nous pouvons dire que chaque module est un ensemble de règles déclenchées par le même événement et également les règles déclenchées par une exécution en cascade (dans le modèle des règles actives, une règle peut déclencher une autre en générant l'évènement de cette dernière ou en rendant sa condition valide si son évènement a déjà détecté). Autrement dit, *une règle ne peut déclencher qu'une règle du même module.*

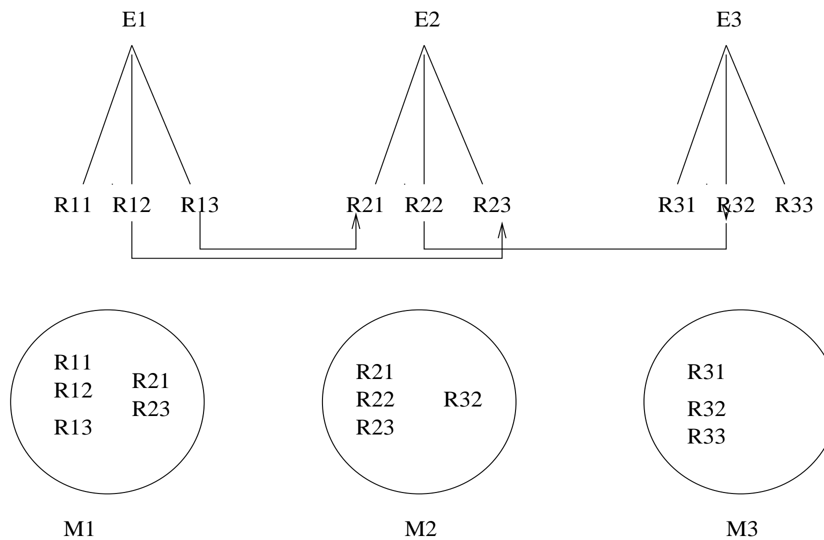


FIG. 4.6 – Décomposition en modules

L'arrivée d'un événement déclenche seulement les règles du module correspondant. Les règles du module dont l'événement correspondant est l'événement qui a été détecté seront exécutées d'abord avant les règles déclenchées par ces règles. Ceci permet de maintenir une concordance et une cohérence dans l'exécution globale. Par exemple, dans la figure Fig 4.6 la règle **R13** doit être traitée avant **R21**, car l'exécution de l'action de la règle **R13** déclenchera la règle **R21**. La modularité doit, de ce fait, respecter les dépendances entre les règles. Nous avons vu, plus haut dans ce chapitre, que les événements (simples) peuvent être composés pour donner un événement plus complexe. C'est la fonction d'agrégation qui assure ce rôle au niveau du moniteur actif. Cette possibilité de composer des événements permet une réutilisabilité des modules actifs par d'autres développeurs. En effet, un développeur souhaitant intégrer dans son application active un module à base de règles actives d'une autre application dans une bibliothèque dédiée à cette fin, peut utiliser l'événement déclenchant ce module externe. Ce même événement peut participer à la composition d'un événement complexe qui fait appel aux règles du module externe pour assuer la composition.

A- Exemple de décomposition :

Nous appliquons la décomposition de l'application Multicast en utilisant :
- un seul module : la table 4.3 montre le module de l'application.
- plusieurs modules : les tables 4.4, 4.5 et 4.6 indiquent les modules qui composent selon l'événement l'application du multicast.

Module Multicast
Les sémantiques d'exécution Consommation d'événement = vérifié Mode de couplage E-C = immédiat Mode de couplage C-A = immédiat Ppriorités des règles = (R3, R4, R1, R2) ; R5 ; R6 ; R7 ; R8 ; R9 ; R10 Exécution en cascade = itérative
Les règles actives R1 : On NACK If *Presence* Then A1 R2 : On NACK If ¬*Presence* Then {A2,A3} R3 : On NACK If *Sent* Then A4 R4 : On NACK If ¬*Sent* Then A5 R5 : On ACK If $total = Max$ Then A6 R6 : On ACK If $total < Max$ Then A7 R7 : On PACKET If *True* Then A8 R8 : On ACK-Timeout If *True* Then {A9,A10} R9 : On GroupJoin If $nbr = MaxClient$ Then A11 R10 :On GroupJoin If $nbr < MaxClient$ Then {A12,A13}

TAB. 4.3 – Application du Multicast.

Multicast Module 1
Les sémantiques d'exécution Consommation d'événement = vérifié Mode de couplage E-C = immédiat Mode de couplage C-A = immédiat Priorités des règles = (R3, R4, R1, R2) Exécution en cascade = itérative
Les règles actives R1 : On NACK If *Presence* Then A1 R2 : On NACK If ¬*Presence* Then {A2,A3} R3 : On NACK If *Sent* Then A4 R4 : On NACK If ¬*Sent* Then A5

TAB. 4.4 – Multicast : module 1.

La notion du module est une valeur ajoutée pour une bonne exécution. La modularité est une décomposition logique de l'application. Cette décomposition n'est pas physique car les modules peuvent contenir les mêmes règles (voir figure 4.6), si celles-ci ont des dépendances entre elles.

Cette modularité par évènement suivie par la contrainte de règles directement déclenchables est la décomposition de base. Le programmeur du service d'application peut en choisir d'autres en plus. On peut aussi définir des décompositions qui prennent en compte un ensemble d'évènements comme facteur de modularité afin d'augmenter la granularité de la décomposition.

Module Multicast 2
Les sémantiques d'exécution
Consommation d'événement = vérifié
Mode de couplage E-C = immédiat
Mode de couplage C-A = immédiat
Priorités des règles = R5 ; R6 ; R7
Exécution en cascade = itérative
Les règles actives
R5 : On ACK If $total = Max$ Then A6
R6 : On ACK If $total < Max$ Then A7
R7 : On PACKET If *True* Then A8

TAB. 4.5 – Multicast : module 2.

Module Multicast 3
Les sémantiques d'exécution
Consommation d'événement = vérifié
Mode de couplage E-C = immédiat
Mode de couplage C-A = immédiat
Priorités des règles = R8 ; R9 ; R10
Exécution en cascade = itérative
Les règles actives
R8 : On ACK-Timeout If *True* Then {A9,A10}
R9 : On GroupJoin If $nbr = MaxClient$ Then A11
R10 :On GroupJoin If $nbr < MaxClient$ Then {A12,A13}

TAB. 4.6 – Multicast : module 3.

Dans les modules 2 et 3 du multicast il existe une règle qui ne partage pas le même événement avec les deux autres. Il faut noter que pour des raisons de performance le processus de décomposition ne doit pas créer des modules dont le nombre de règles est inférieur à un certain seuil (dans l'exemple le seuil a été fixé à trois).

L'identification doit inclure la référence du moniteur, du service d'application, ainsi que celle des modules actifs. La référence du module peut être définie en tant que méta-donnée, sur toute l'application active.

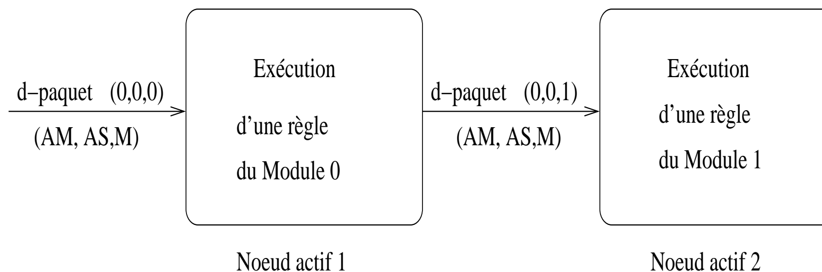


FIG. 4.7 – Identification d'un module

B- Référencement entre modules :

Dans Fig 4.7, l'arrivée d'un d-paquet (0,0,0) qui identifie le module **0**, le service d'application **0** et le moniteur actif **0** déclenche une règle du module **0**. Le résultat de l'exécution de cette règle est l'envoi au prochain nœud actif d'un d-paquet (0,0,1) qui référence le même service d'application et moniteur actif, mais le module **1**. Les modules peuvent se mettre en référence entre deux nœuds actifs différents, mais pas dans le même nœud actif afin de respecter la règle de modularité : *une règle ne peut déclencher qu'une règle du même module dans le même nœud.*

Cette manière de composer ne nécessite aucun temps additionnel, contrairement à ce qu'on peut reprocher aux techniques actuelles qui requièrent une composition en amont ou une composition en aval. La composition en amont nécessite une pré-compilation en intégrant les composantes dès le départ avant l'envoi du code. Par contre, une composition en aval nécessite une composition par le nœud actif avant d'exécuter le code, ce qui est le cas de nombreux projets dans le domaine des réseaux actifs comme CANEs [14] et FAIN [67]. Cette dernière approche pénalise beaucoup les performances du nœud en lui ajoutant une autre tâche. Notre approche ne nécessite ni une composition en amont, ni en aval. C'est une composition par déclenchement ou agrégation d'événements appartenant à d'autres modules actifs accessibles à partir du serveur de code (voir chapitre 6) ou des modules systèmes fournis par le moniteur actif pour le traitement des événements systèmes.

4.5.7 La spécification des sémantiques d'exécution

Les sémantiques d'exécution permettent de définir les modes sous lesquels les règles vont être exécutées. Le choix des paramètres est laissé au développeur de l'application active qui doit prendre toutes les précautions pour garantir une cohérence entre les valeurs choisies.

Dans l'exemple de la table 4.7, la valeur *vérifié* du paramètre "consommation d'événement" indique qu'une occurrence ou l'instance d'un événement est détruite seulement quand la condition est vérifiée, autrement elle reste indéfiniment jusqu'à ce que la condition soit vérifiée. La valeur *immédiat* pour le mode du couplage E-C indique que la partie condition d'une règle est immédiatement évaluée après la signalisation de l'événement. De même, la valeur *immédiat* pour le mode de couplage C-A indique que la partie action d'une règle est immédiatement exécutée quand la condition est vérifiée.

Le mode *différé* est sous-entendu lorsqu'il n'y a aucune contrainte entre la vérification de la condition et l'exécution de l'action. Dans ce cas, les actions peuvent être exécutées à la fin du traitement d'un module ou d'un service d'application, après la vérification des conditions de toutes les règles. La flexibilité procurée par le mode *différé* nous permet d'envisager une exécution globale des actions qui obéit à un ordre global qui privilégie une action vis-à-vis d'une autre si toutes les deux sont sous le mode différé (une action d'une règle système plus prioritaire). Ce mode peut aussi dépendre du mode "exécution en cascade".

De plus, dans cet exemple, le paramètre "exécution en cascade" est défini comme étant *itératif*, ce qui signifie que l'exécution en cascade des règles est faite par une approche en largeur (à la fin de l'exécution de chaque action, toutes les règles déclenchées sont considérées en même temps). Pour plus de détails au sujet des paramètres et leurs valeurs, voir la table 3.2.

L'en-tête de chaque module sera employée au cours du processus de correspondance fait par la machine virtuelle pour attribuer un moniteur actif approprié au module (ou au service d'application de manière générale) en question (voir section 4.7).

Module M1
Les sémantiques d'exécution Consommation d'événement = vérifié Mode de couplage E-C = immédiat Mode de couplage C-A = immédiat Priorités des règles = aucune Exécution en cascade = itérative
Les règles actives R1 : On E1 If C1 Then A1() R2 : On E1 If C2 Then A2() ... Rn : On Ep If Cn Then An()

TAB. 4.7 – Exemple de module.

4.6 Description du moniteur actif

Comme nous avons vu dans la première partie du Chapitre 3, les règles peuvent être exécutées différemment selon la sémantique opérationnelle fournie par le moniteur actif, ou définie pour les règles actives dans le cas d'un moniteur actif paramétrable. Comme nous avons vu aussi, un moniteur actif paramétrable ne répond pas généralement aux exigences d'exécution des applications. Par ailleurs, avoir quelques systèmes restreints ne satisfait pas pleinement aux exigences sémantiques des applications. En conséquence, il est intéressant de fournir un ensemble de modules qui nous permettent de définir des moniteurs actifs spécifiques qui répondent le mieux aux besoins des applications en termes de sémantique, performance et flexibilité [64].

Un environnement d'exécution ou un moniteur actif implémente la sémantique opérationnelle d'une classe de services d'application, chacune étant définie par un ensemble de règles ECA. C'est un service logiciel personnalisé par opposition à un service d'application. Un moniteur actif se compose d'un ensemble de composants qui contrôlent les différentes parties de règles ECA et implémentent la sémantique d'exécution (Figure 4.8). Parmi ces composants, nous distinguons l'écouteur d'événements, l'ordonnanceur des événements, l'ordonnanceur des règles et l'exécuteur des règles (comprenant l'évaluateur des conditions et le lanceur des actions). Chacun de ces principaux composants peut être décomposé en services plus élémentaires.

Le composant "écouteur d'événements" : il est constitué d'un ensemble de détecteurs d'événements, chacun étant dédié à la détection des occurrences d'un type donné d'événements produits par une source donnée. Dans un réseau de capteurs, chaque capteur est considéré comme étant un détecteur. Au niveau du nœud, il est possible de définir des services systèmes qui sont à l'écoute des informations externes et capables de les interpréter, comme le taux de pertes des paquets ou le débit du nœud. A partir de ces informations les écouteurs systèmes peuvent les interpréter pour les traduire en événements. Par exemple, si le taux de pertes dépasse un certain seuil on peut en déduire qu'une congestion a été formée et un événement de congestion est alors signalé.

Les utilisateurs peuvent aussi installer leurs propres services d'écoute, qui peuvent récupérer des

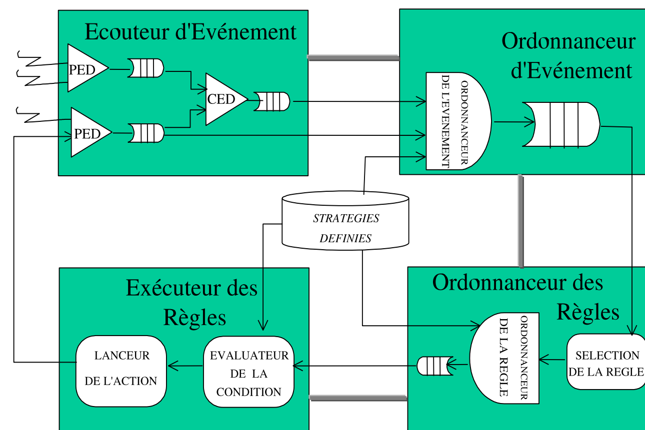


FIG. 4.8 – Une vue simple d'un moniteur actif.

informations systèmes ou propres à l'application pour les traduire en événements internes. Cette partie n'a pas était traité car elle nécessite une étude approfondie sur la machine virtuelle (section suivante) et sur la sécurité pour permettre d'installer au niveau bas de l'architecture des “daemons” utilisateurs.

Nous distinguons entre les détecteurs d'événements primitifs (PED) et les détecteurs d'événements composés (CED). La détection des événements primitifs peut être faite par diverses manières : interception d'un signal externe (un paquet de données par exemple), dés-encapsulation des flux de données, récupération des exceptions en cours d'exécution des règles ou celles générées par le système, etc. La détection d'un événement composé est faite par une fonction globale spécifique dont les opérandes sont des événements primitifs ou d'autres événements composés.

On suppose que les détecteurs d'événements d'un moniteur actif donné sont locaux aux nœuds actifs correspondants. Les événements distribués sont encapsulés dans des paquets de données traversant le réseau. Ils sont détectés seulement une fois que les paquets sont traités par le nœud actif. Une occurrence d'un événement ne peut exister qu'après son signalement par un détecteur. C'est également le rôle du détecteur d'assigner un *evid* à chaque occurrence d'événement.

Nous différencions la période de détection du moment de signalisation d'un événement. La période de détection est le moment où une occurrence d'événement est produite par une source (par exemple, le temps où un NACK est envoyé par un client vers le nœud). Le temps de signalisation est le moment où une occurrence d'événement est mise dans la file d'attente de sortie d'un détecteur (le temps où ce NACK arrive au nœud actif cible). Les deux temps peuvent se recouvrir pour quelques types et sources d'événements.

Le composant “ordonnanceur des événements” : il est consacré à l'organisation des événements de différents types en se basant sur une stratégie d'ordonnancement qui indique dans quel ordre les occurrences d'événements sont considérées pour déclencher les règles actives. La stratégie d'ordonnancement des événements est un paramètre pour chaque moniteur actif ; elle implique la définition d'une structure de données appropriée pour l'organisation d'événements. Ce composant peut être considéré comme un module de *multiplexage d'événements*. Il détermine la manière avec laquelle les événements de différents types seront considérés, selon la logique qu'un service d'application veut implémenter, la priorité entre événements ou le niveau de complexité de chacun

(l'événement le plus simple est plus prioritaire que celui issu d'une fonction d'agrégat), ...

Le composant "ordonnanceur de règles" : après la sélection des règles concernées par les événements transmis par l'étage précédent, l'ordonnanceur de règles définit un plan d'exécution pour l'ensemble de règles déclenchées par le même événement. La manière d'organiser ce plan d'exécution dépend de l'existence de priorités qui sont accordées aux règles. Cela dépend également de la sémantique que le service d'application veut implémenter ou les dépendances entre celles-ci.

Le composant "exécuteur des règles" : ce composant évalue les conditions et lance les actions selon le mode de couplage mis en œuvre dans le moniteur actif. L'exécuteur de règles peut être équipé de quelques composants qui donnent régulièrement des rapports permettant au moniteur actif d'être informé sur l'exécution régulière ou exceptionnelle de l'action de la règle. Le lanceur d'action peut être considéré comme un détecteur spécifique d'événements dans le cas où les événements en sortie (produits par les règles) sont réutilisés dans le comportement du moniteur actif. On associe à chaque moniteur un environnement d'exécution pour exécuter ou interpréter les actions et les conditions. Cet environnement dépend du langage de programmation, des actions et des conditions. Cela nous permet de ne pas se limiter à un seul type d'environnement ou de langage.

Définir les composants d'un moniteur actif n'est pas suffisant pour indiquer le comportement du moniteur actif. D'autres éléments tels que les flux de données, les flux de contrôle et les files d'attente doivent être également définis afin d'indiquer les modèles de communication entre les composants. Dans la figure 4.8, nous avons représenté les flux de données par des flèches fines tandis que les flux de contrôle (éléments du modèle de communication) sont représentés par des lignes "en gras".

Dans le cas du couple de composants *écouteur* et *ordonnanceur d'événements*, le flux de données correspond à l'ensemble des événements primitifs ou complexes. En ce qui concerne les flux de contrôle, ce sont en particulier des signaux de synchronisation qui garantissent une cohérence dans le partage de la ressource critique représentée par la file d'attente des événements. Cette file obéit au modèle consommateur-producteur. Comme le moniteur actif est apte à gérer plusieurs services d'application à la fois, parmi les flux de contrôle nous comptons la gestion du basculement dans l'exécution des services d'application et le changement de contexte.

De même, les interactions entre les différents composants sont basées sur le partage d'une même file d'attente tout en respectant la synchronisation. Cette synchronisation permet de garantir une forte autonomie des différents composants et un haut degré de parallélisme au niveau du moniteur actif.

4.7 Description de la machine virtuelle

La machine virtuelle est la couche entre le système d'exploitation du nœud et les moniteurs actifs. Nous supposons que chaque nœud dans le réseau de transmission est équipé d'une telle machine virtuelle. Elle est mise en place une fois pour toutes dans les nœuds et fournit les fonctionnalités minimales de :

- acheminement des paquets (envoi, réception et routage des paquets),
- analyse des paquets pour différencier entre ceux correspondant avec les moniteurs actifs (m-paquets), ceux correspondant avec les services d'application (s-paquets) et ceux correspondant

aux données utilisateur (d-paquets). Les autres types de paquets sont simplement routés,
- amorçage des paquets des moniteurs actifs afin de rendre le nœud capable de traiter des services actifs,
- chargement des services d'application et leur mise en correspondance aux moniteurs actifs afin d'être exécutés dans le nœud actif,
- mise en correspondance des données d'application aux services d'application dans les nœuds actifs.

L'implémentation d'une machine virtuelle dans un nœud nous impose d'avoir une bonne connaissance de tous les dispositifs qui caractérisent le nœud : type de machine, type de système d'exploitation, des protocoles, la capacités des liens, les langages, les compilateurs, etc. Les conditions, sous lesquelles une machine virtuelle a été créée, dépendent des données décrivant cet environnement. En conséquence, en plus des méta-données décrivant les moniteurs actifs dans les nœuds actifs, nous supposons l'existence d'une base de méta-données décrivant l'environnement du nœud.

4.7.1 Les composants de la machine virtuelle

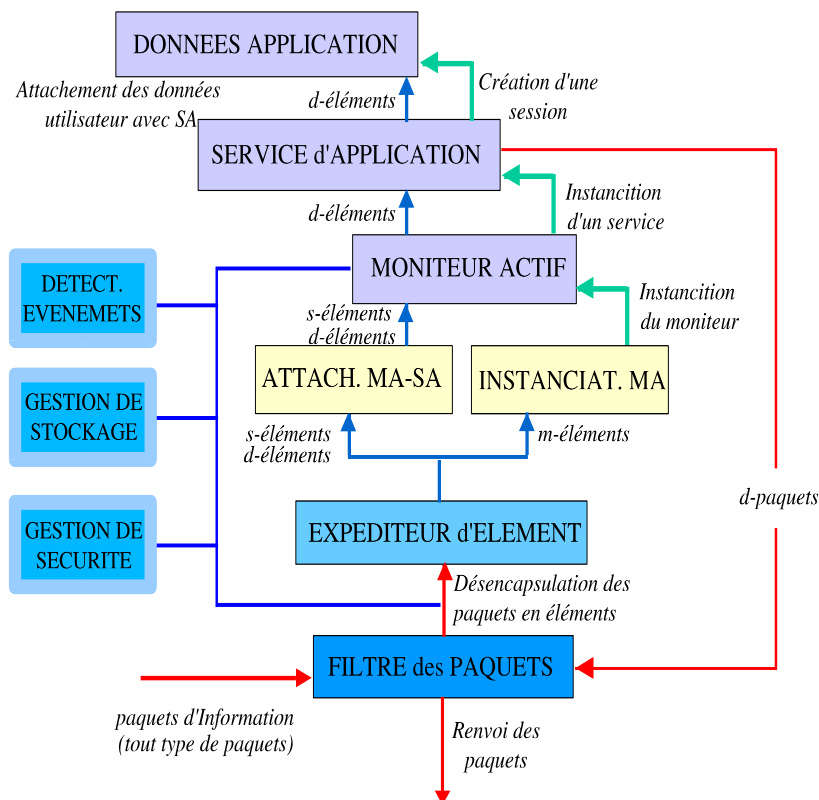


FIG. 4.9 – L'architecture de la machine virtuelle

Afin d'assurer pleinement son rôle, la machine virtuelle comporte un ensemble de composants spécialisés et fortement connectés, que nous présentons dans ce qui suit.

Filtre de paquets : quand un paquet arrive à un nœud, le filtre de paquets analyse son type. S'il identifie un d-paquet, un m-paquet, ou un s-paquet, il enlève l'en-tête IP et forme respectivement un d-element, un m-element et un s-element afin de le communiquer à l'expéditeur d'éléments. Si le paquet n'est pas reconnu en tant qu'un de ces types de paquets, il est expédié selon la table de routage standard. Par contre, quand un paquet résulte d'un traitement effectué dans les couches actives supérieures du nœud, le filtre de paquets forme le paquet correspondant selon le format de l'interface réseau sous-jacent et l'envoie à travers le réseau. Dans les architectures hautes performances, il est recommandé que ce composant soit très optimisé, même une implémentation physique est suggérée.

Expéditeur d'élément : le rôle de l'expéditeur est d'orienter les éléments ou les paquets reçus du filtre à la file d'attente correspondante.

Gestionnaire de Stockage : il permet aux applications des utilisateurs et aux moniteurs actifs de stocker leurs données provisoires ou persistantes dans le disque ou le cache du nœud. Il contrôle l'accès mémoire et donne une abstraction basée sur des appels systèmes comme la création, la suppression, l'ouverture ou la fermeture d'un fichier et des fonctions comme la lecture et l'écriture dans les fichiers.

Gestionnaire de moniteurs : son rôle est de permettre à plusieurs moniteurs actifs d'être exécutés d'une façon parallèle et sécurisée. Cependant, il doit pouvoir activer un moniteur concerné par un événement déclenché. Il doit aussi donner au moniteur actif la capacité de contrôler ses propres processus fils (ceux qui correspondent aux services d'application) comme les fonctions d'interruption, de mise en attente, de sauvegarde du contexte et du rechargement du processus fils.

Gestionnaire de sécurité : il vérifie les propriétés de sécurité au niveau du nœud pour l'exécution des services d'application et des moniteurs actifs quand ils sont chargés. Il est également chargé de vérifier l'utilisation et les droits d'accès aux ressources demandées par un service d'application ou un moniteur actif.

Gestionnaire d'événements : la détection de tous les événements est opérée dans la machine virtuelle indépendamment des moniteurs actifs et des services d'application. Ces événements peuvent être internes ou externes. La liste de ces événements est communiquée au gestionnaire des moniteurs. L'installation d'un filtre de paquets passifs au niveau de la machine virtuelle est liée à ce gestionnaire d'événements, car, par le biais du filtre, les événements vont être déclenchés.

Instanciation de moniteurs : le rôle de cette partie de la machine virtuelle est l'instanciation d'une classe de moniteurs actifs avec la base d'information de stratégies (SIB-Strategy Information Base) correspondante. Cette base contient l'information fournie par les m-paquets tels que le mode de consommation d'événements, le mode de couplage E-C, le mode de couplage C-A, les priorités des règles et l'exécution en cascade. Le type du langage ou l'environnement d'exécution des actions et des conditions dépendent de leur disponibilité sur le nœud. Avant l'instantiation effective et complète des moniteurs, ce composant de la machine virtuelle doit projeter les ressources du nœuds avec les besoins formulés par les moniteurs et les associer.

L'attachement S-AM : le rôle de cet attachement est d'associer les s-éléments avec le moniteur actif approprié afin de charger un service d'application.

L'attachement D-UA : cet attachement associe un d-élément avec l'application appropriée de l'utilisateur. Cette application est chargée par le service d'application qui contrôle les applications des utilisateurs. La hiérarchisation de plusieurs services de données d'application ne change en rien le fonctionnement de cet attachement.

Dans ce qui suit, nous décrivons la structure des différents types de paquets : le d-paquet, le s-paquet et le m-paquet.

4.8 Le format des paquets

Les paquets se composent d'une en-tête réseau (IP, Ethernet, ATM, ...) et d'une charge utile. La charge utile est un d-élément, un s-élément ou un m-élément ou tout autre type de paquet (paquet de données simple ou un paquet actif appartenant à une autre infrastructure active). Si le paquet actif est encapsulé dans un paquet ANEP, la charge utile correspond dans ce cas à la charge utile ANEP. Le champ "type de paquet", qui est commun à tous les paquets, indique si la charge utile du paquet est un d-élément, un s-élément ou un m-élément. Les valeurs de ce champ sont comme suit :

- 00 : d-paquet,
- 01 : s-paquet,
- 10 : m-paquet.

Les autres champs des paquets sont indiqués ci-dessous selon le type du paquet.

– d-élément : pour pouvoir identifier une application d'un utilisateur parmi toutes les applications qu'un service d'application gère, un d-paquet doit référencer le flux de données, le service d'application et le moniteur. L'identifiant du flux d'application doit être un identifiant de la session courante qui est propre à un utilisateur et à une connexion.

<table>
<tr><td>Type du paquet</td><td colspan="2">Identifiant de l'application utilisateur</td></tr>
<tr><td>Identifiant du service d'application</td><td>Identifiant du moniteur</td><td>Données</td></tr>
</table>

Un d-paquet est consommé par le nœud actif dès son interception. Cela ne se traduit pas par son envoi automatique vers les autres nœuds sauf si c'est indiqué explicitement par le paquet. Concrètement, ceci est fait pour éviter l'inondation inutile du réseau par des d-pquets puisque chaque d-paquet déclenche le rapatriement et l'exécution d'un code. Ceci évite que le paquet actif boucle indéfiniment dans le réseau.

– s-élément : en plus de l'identifiant du service d'application et du moniteur actif, le s-élément contient tous les composants du service d'application : les règles, les modules, les événements, les conditions et les actions. Il doit également contenir les paramètres qui définissent les interactions entre les règles (les sémantiques d'exécution) et leur composition en modules.

<table>
<tr><td colspan="2">Type du paquet</td><td colspan="2">Identifiant du service d'application</td></tr>
<tr><td>Identifiant du moniteur</td><td colspan="2">Nombre de règles</td><td>Nombre de modules</td></tr>
<tr><td>Nombre d'événements</td><td colspan="2">Nombre de conditions</td><td>Nombre d'actions</td></tr>
<tr><td colspan="2">Liste des règles</td><td colspan="2">Liste des modules</td></tr>
<tr><td>Liste des événements</td><td colspan="2">Liste des conditions</td><td>Liste des actions</td></tr>
</table>

La taille des différentes listes d'entités (événement, condition, action) est variable puisque le nombre de chacune de ces entités est également variable. Cependant, toutes ces entités ont la même description :

identifiant de l'entité	Taille de l'entité	Valeur de l'entité

– m-élément : il contient la référence du moniteur et les paramètres de la sémantique d'exécution pour définir un moniteur adapté aux besoins du client. Les champs suivants représentent un m-element.

<table>
<tr><td>Type du paquet</td><td>identifiant du moniteur</td><td>Consommation de l'événement</td></tr>
<tr><td colspan="2">Mode de couplage E-C</td><td>Mode de couplage C-A</td></tr>
<tr><td colspan="2">Exécution en cascade</td><td>Priorités des règles</td></tr>
<tr><td colspan="3">Langage, l'environnement d'exécution et besoins en ressources</td></tr>
</table>

4.9 Conclusion

Ce chapitre était consacré à la mise en œuvre d'une architecture d'un nœud actif ARFANet qui implémente les caractéristiques des règles actives.

Nous avons pris conscience des spécificités des réseaux et en particulier les réseaux actifs pour pouvoir tirer profit de la formalisation des programmes actifs en se basant sur le formalisme des règles actives. Notre but est de tirer avantage des caractéristiques des règles actives comme leur flexibilité et modularité. De plus, la structure événementielle des règles actives s'adapte parfaitement à l'aspect événementiel du réseau lui-même.

Cette implémentation nous a poussé à définir des composants actifs d'une architecture à trois couches permettant de prendre en charge les parties constituant la règle, à savoir l'événement, la condition et l'action.

Nous avons implémenté dans cette étape de notre travail toutes les couches qui composent ARFANet en Java. Ce langage, avec les caractéristiques d'héritage, de portabilité et de multi-threading, nous a aidé à la réalisation rapide d'une implémentation fiable et sûre. Cependant, par souci de performance, il est impératif de créer une autre version à base d'un langage plus rapide comme le langage C, surtout pour la machine virtuelle. Actuellement les classes d'action et de condition sont écrites en Java. Pour étendre notre infrastructure et pouvoir accepter plusieurs langages, généralistes ou actifs, nous sommes amenés à intégrer plusieurs EE existants.

Notre approche se veut un mélange entre l'approche discrète et intégrée, pour cela les deux techniques d'orientation des données actives sont utilisées. Pour rappel, une des techniques utilise un référencement explicite avec une référence vers le service actif, c'est ce que nous réalisons avec les d-paquets. Mais il est tout à fait possible d'intercepter des paquets passifs en installant des filtres qui déclencheront directement des événements spécifiques.

Le chapitre suivant sera consacré à la modélisation d'un nœud actif ARFANet par un réseau de files d'attente. Cette modélisation nous permettra d'évaluer les performances et d'extraire des conclusions sur le choix des paramètres et des architectures sous-jaçantes à un nœud.

Chapitre 5

Une représentation interne pour la Machine Virtuelle

5.1 Introduction :

Un routeur classique est typiquement représenté sous forme de files d'attente, la file d'entrée qui reçoit les paquets arrivant au nœud et la file de sortie qui reçoit les paquets après leur traitement (la détermination du prochain nœud). Un nœud actif ARFANet est un nœud qui étend les fonctionnalités d'un nœud classique.

Dans ce chapitre, nous nous intéressons à la représentation interne un nœud ARFANet, en particulier la machine virtuelle (la couche basse), sous forme d'un réseau de files d'attente. Nous considérons deux configurations et pour chacune de ces configurations, nous analysons ses performances en termes de débit, taux de pertes et de latence.

Pour cela, nous modélisons un nœud implémentant les fonctionnalités d'ARFANet en utilisant le formalisme de l'algèbre des processus PEPA [46]. Nous comparons les performances d'un nœud "passif" ou standard dont la seule fonctionnalité est de retransmettre les paquets le traversant avec les performances d'un nœud actif ARFANet pour chacune de ces configurations.

PEPA (Performance Evaluation Process Algebra) est un formalisme, développé par Hillston en 1994 [46], qui étend l'algèbre des processus classiques en associant, à chaque action, une variable aléatoire, représentant la durée. Ces variables aléatoires sont supposées être exponentiellement distribuées, ce qui établit clairement une relation entre le modèle de l'algèbre des processus et un processus de Markov en temps continu. Via ce processus de Markov, des mesures de performances peuvent être extraites à partir du modèle.

L'un des attraits de ce formalisme est sa claire structure compositionnelle qui permet de construire un modèle à partir d'éléments ou composantes qui reflètent la composition du système à modéliser. Ceci permet, d'une part, une meilleure appréhension de la complexité des modèles à construire et d'autre part, une meilleure exploitation des propriétés structurelles lors de la résolution du modèle.

Par ailleurs, PEPA est une approche formelle qui comprend un *calculus*, basé sur des relations d'équivalence formellement définies, pour la manipulation et l'analyse des modèles. Celui-ci comprend une technique de simplification des modèles qui exploite l'équivalence des comportements pour partitionner l'espace d'états, c'est la technique d'*agrégation* [46]. Pour répondre au problème de l'explosion de l'espace d'états, une seconde solution a été développée. Elle consiste à faire un mapping d'un modèle PEPA vers une représentation compacte du générateur associé à la chaîne

de Markov sous-jacente [47]. Cette technique est basée sur l'utilisation de l'algèbre de Kronecker. Ces deux techniques ont été intégrées à l'outil PEPA Workbench [36], offrant ainsi à l'utilisateur la possibilité de choisir la technique optimale pour son modèle et même la possibilité de combiner de manière appropriée ces techniques.

Bien que l'on retrouve l'aspect compositionnel ainsi que les techniques d'agrégation et/ou de représentation tensorielle dans d'autres formalismes tels que les réseaux d'automates stochastiques (RAS) de Plateau [78][10], les réseaux de Petri stochastiques superposés (Superposed GSPN) de Donatelli [26] ou les réseaux d'activités stochastiques (Stochastic Activity Networks) de Sanders [79], PEPA a en plus le mérite d'être très simple à utiliser et, comme nous le verrons plus loin, il permet d'avoir une description intuitive du système à modéliser grâce à son petit, mais néanmoins riche, ensemble d'opérateurs.

De plus, plusieurs autres outils utilisent PEPA comme formalisme de description. Ainsi, il est l'un des langages en entrée de, entre autres :

- l'outil de modélisation *Möbius* développé par le groupe PERFORM [17], qui grâce à sa technique de sauvegarde sur disque permet de repousser les limites de la modélisation en termes d'espace d'états.
- *PRISM model checker* [59] qui permet l'analyse des systèmes stochastiques et probabilistiques en offrant à l'utilisateur la possibilité de vérifier une propriété logique pour un modèle donné. Plusieurs types de logiques peuvent être considérés dont CSL [8] pour les chaînes de Markov à temps continu.

Pour plus de détails sur le formalisme PEPA se reporter à l'annexe A.

Les résultats obtenus avec PEPA sont comparés aux mesures effectuées sur l'infrastructure ARFANet et à la simulation avec l'outil SimJava [25]. Ce dernier associe à chaque composante du système à simuler une classe Java. Le corps d'une classe définit exactement le fonctionnement de la composante correspondante. Les composantes peuvent communiquer entre elles. Les interactions entre les classes définissent le comportement de tout système.

Par ailleurs, nous comparons les performances d'un nœud ARFANet à celle d'un nœud d'une référence dans la littérature des réseaux actifs, à savoir ANTS. Cette étude nous permettra de mesurer la surcharge de l'exécution des règles actives en comparaison avec un nœud passif.

Le chapitre est structuré comme suit. La section 5.2 est consacrée à la modélisation de l'infrastructure ARFANet et la présentation de la première configuration. Elle sera consacrée également à décrire le modèle PEPA développé pour cette configuration, à définir les critères de performance nécessaires à notre étude, et enfin, à discuter les résultats numériques obtenus. L'amélioration du premier modèle engendre la définition d'un deuxième modèle dont nous définirons le modèle PEPA et étudierons les performances dans la section 5.3. La validation des résultats obtenus par la modélisation, nous pousse à les comparer aux mesures réelles effectuées sur de notre infrastructure dans la section 5.4. Tandis que nous comparons les performances de notre infrastructure avec celle d'ANTS (exemple d'une architecture type d'un réseau actif) dans la section 5.5. Enfin, nos conclusions sont présentées dans la section 5.6.

5.2 La modélisation d'un nœud ARFANet

La modélisation de l'architecture à couches du nœud actif et son mode d'exécution suppose que le nœud est vu comme un ensemble de files d'attente de capacités finies. Nous nous intéressons à la configuration du réseau de files d'attente décrite par la figure 5.1.

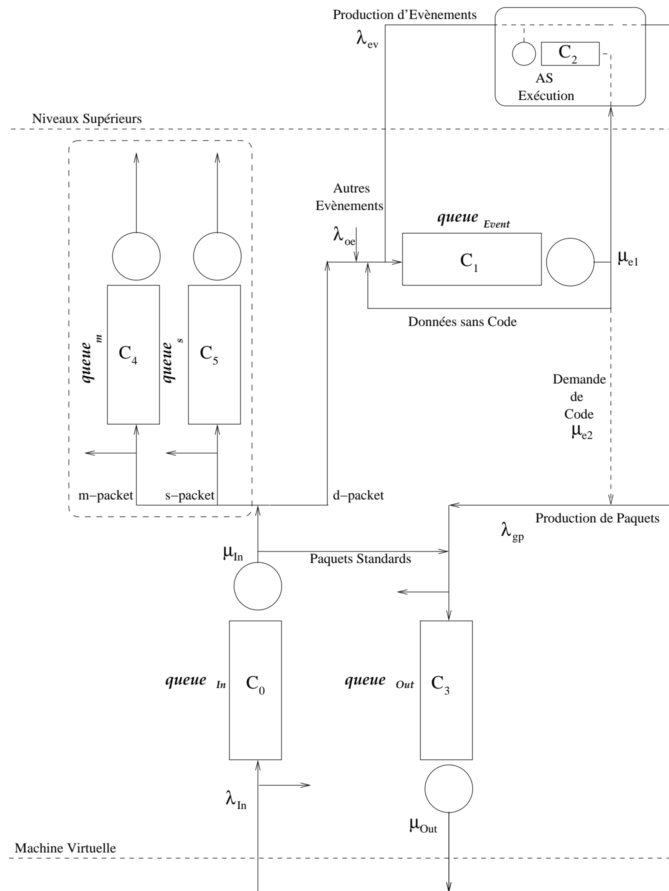


FIG. 5.1 – Réseau de files d'attente modélisant un nœud actif : *Configuration 1*

Dans cette configuration, les paquets arrivant au nœud sont mis dans la file d'entrée $queue_{in}$, avant d'être répartis selon leur type :

- les paquets standards (à forwarder) sont directement envoyés à la file de sortie $queue_{out}$,
- les paquets actifs de données (*d-paquets*) sont orientés vers la file des évènements $queue_{event}$. Si un *d-paquet* en service dans $queue_{event}$ référence un paquet de code actif (*s-paquet* or *m-paquet*) absent du nœud, le paquet est remis dans la file $queue_{event}$ et une requête concernant le ou les paquets absents est envoyée au réseau via la file de sortie.
- les paquets contenant le code actif des services d'applications (*s-paquets*) sont orientés vers la file $queue_s$,
- les paquets contenant le code actif des moniteurs (*m-paquets*) sont orientés vers la file $queue_m$.

En plus des *d-paquets*, $queue_{event}$ reçoit également des évènements internes tels que les évènements produits par le système et l'exécution des applications. A la fin de son service dans la file $queue_{event}$, le *d-paquet* est envoyé à la file d'attente de la couche application pour déclencher la règle active correspondante. Par ailleurs, l'exécution de l'application peut générer de nouveaux paquets actifs envoyés à la file d'attente de sortie.

L'objectif de cette étude est de donner, d'une part, une représentation sous forme de réseau de files d'attente. Cette représentation nous permettra d'étudier analytiquement le comportement de la machine virtuelle et ses relations avec les autres couches et notamment l'extérieur du nœud.

D'autre part, cette étude permet l'analyse de l'impact de la présence des paquets actifs sur les performances des paquets standards. Comme ces paquets sont directement orientés de la file d'entrée vers la file de sortie, seules les couches ayant un lien direct, donc un impact, avec ces deux files sont explicitement représentées. Dans cette configuration, en plus de la machine virtuelle, la couche services d'application est représentée par la file $queue_{as}$ qui reçoit de la machine virtuelle les données nécessaires à l'exécution d'une application et qui peut produire à son tour des paquets vers la file de sortie du nœud. La couche moniteurs d'exécution n'est pas explicitement représentée car elle n'a aucun lien direct avec la file de sortie.

La modélisation d'un routeur actif doit prendre en compte l'exécution parallèle par un seul processeur. La représentation par la figure 5.1 ne doit pas exprimer une exécution parallèle des différentes files par différents processeurs. Dans PEPA les composantes concurrentielles sont traitées une à la fois et doivent coopérer sur un certain ensemble d'actions. Cette manière correspond à la coopération des différentes tâches exécutées au niveau de l'implémentation réelle de la couche machine virtuelle.

5.2.1 Le modèle PEPA

Comme nous nous intéressons principalement à l'impact des nouvelles fonctionnalités des nœuds du réseau sur les paquets traditionnels ou passifs, en termes de taux de pertes et de latence du nœud, le modèle PEPA développé pour cette configuration du réseau de files d'attente ne prend pas en compte les files d'attente des paquets contenant le code actif, c'est-à-dire $queue_s$ et $queue_m$. Ce modèle comprend quatre composantes : $Buffer_0^{(in)}$, $Buffer_0^{(out)}$, $Buffer_0^{(event)}$ et $Buffer_0^{(as)}$ modélisant respectivement les files $queue_{in}$, $queue_{out}$, $queue_{event}$ et $queue_{as}$. Les composantes de ce modèle ont pour forme $Buffer_{Nb_paquet}^{(file)}$, où $file$ désigne la file correspondante et Nb_paquet représente le nombre de paquets présents dans cette file. De même pour les autres files. Ces composantes sont définies comme suit :

- **Composante $Buffer_0^{(in)}$:** Les arrivées du réseau à la file d'attente d'entrée du nœud sont modélisées en utilisant le type d'action in. Comme nous faisons une distinction entre les différents types de paquets, le service d'un *d-paquet*, *s-paquet*, *m-paquet* et d'un paquet passif est modélisé en utilisant respectivement le type d'action $service_d$, $service_s$, $service_m$ et $service_p$. La probabilité de servir un paquet de type $x = d, s, m, p$ est notée q_x. Si C_0 est la capacité de la file d'entrée, alors son comportement complet est comme suit :

$$
\begin{array}{lcl}
Buffer_0^{(in)} & \stackrel{def}{=} & (in, \lambda_{in}).Buffer_1^{(in)} \\
Buffer_1^{(in)} & \stackrel{def}{=} & (in, \lambda_{in}).Buffer_2^{(in)} + (service_d, q_d \times \mu_{in}).Buffer_0^{(in)} \\
 & & +(service_p, q_p \times \mu_{in}).Buffer_0^{(in)} + (service_s, q_s \times \mu_{in}).Buffer_0^{(in)} \\
 & & +(service_m, q_m \times \mu_{in}).Buffer_0^{(in)} \\
\vdots & & \vdots \\
Buffer_{C_0}^{(in)} & \stackrel{def}{=} & (in, \lambda_{in}).Buffer_{C_0}^{(in)} + (service_d, q_d \times \mu_{in}).Buffer_{C_0-1}^{(in)} \\
 & & +(service_p, q_p \times \mu_{in}).Buffer_{C_0-1}^{(in)} + (service_s, q_s \times \mu_{in}).Buffer_{C_0-1}^{(in)} \\
 & & +(service_m, q_m \times \mu_{in}).Buffer_{C_0-1}^{(in)}
\end{array}
$$

- **Composante $Buffer_0^{(event)}$:** L'arrivée d'un évènement interne à la file d'attente $queue_{event}$ est modélisée en utilisant l'action in_{oe}. Les autres arrivées sont les événements résultants

de l'exécution des services d'application, modélisés avec l'action $generate_{ev}$, et les *d-paquets* après leur service dans la file d'entrée ($service_d$). Le service au niveau de la file $queue_{event}$ est modélisé par deux types d'actions, $service_{e2}$ qui modélise le service d'un *d-paquet* qui référence un paquet de code absent et $service_{e1}$ qui modélise le service d'un *d-paquet* pour lequel tout se passe normalement avec une probabilité notée q. Soit C_1 la capacité de la file d'événements. Le taux indéfini $\top$ (appelé "Top") est utilisé lors de la coopération entre deux composantes. C'est la composante active qui détermine le taux effectif de l'action.

$$\begin{aligned}
Buffer_0^{(event)} &\stackrel{def}{=} (service_d, \top).Buffer_1^{(event)} + (in_{oe}, \lambda_{oe}).Buffer_1^{(event)} \\
&\quad +(generate_{ev}, \top).Buffer_1^{(event)} \\
Buffer_1^{(event)} &\stackrel{def}{=} (service_d, \top).Buffer_2^{(event)} + (service_{e1}, q \times \mu_{e1}).Buffer_0^{(event)} \\
&\quad +(generate_{ev}, \top).Buffer_2^{(event)} + (in_{oe}, \lambda_{oe}).Buffer_2^{(event)} \\
&\quad +(service_{e2}, (1-q) \times \mu_{e2}).(generate_{req}, \mu_{e2}).Buffer_1^{(event)} \\
&\vdots \\
Buffer_{C_1}^{(event)} &\stackrel{def}{=} (service_d, \top).Buffer_{C_1}^{(event)} + (service_{e1}, q \times \mu_{e1}).Buffer_{C_1-1}^{(event)} \\
&\quad +(generate_{ev}, \top).Buffer_{C_1}^{(event)} + (in_{oe}, \lambda_{oe}).Buffer_{C_1}^{(event)} \\
&\quad +(service_{e2}, (1-q) \times \mu_{e2}).(generate_{req}, \mu_{e2}).Buffer_{C_1}^{(event)}
\end{aligned}$$

– **Composante** $Buffer_0^{(as)}$ **:** Pour modéliser le comportement de la file d'attente des applications en exécution, on considère trois types d'actions : $service_{e1}$, $generate_{ev}$ et $generate_p$. La première décrit l'arrivée d'un événement qui déclenche l'exécution d'un service d'application. Cette exécution peut avoir comme conséquence la génération de nouveaux évènements ($generate_{ev}$) avec le taux λ_{ev} et de nouveaux paquets ($generate_p$) avec le taux λ_{gp}. Si C_2 est la capacité de la file d'attente $queue_{as}$, son comportement est comme suit :

$$\begin{aligned}
Buffer_0^{(as)} &\stackrel{def}{=} (service_{e1}, \top).Buffer_1^{(as)} \\
Buffer_1^{(as)} &\stackrel{def}{=} (service_{e1}, \top).Buffer_2^{(as)} + (generate_{ev}, \lambda_{ev}).Buffer_0^{(as)} \\
&\quad +(generate_p, \lambda_{gp}).Buffer_0^{(as)} \\
&\vdots \\
Buffer_{C_2}^{(as)} &\stackrel{def}{=} (service_{e1}, \top).Buffer_{C_2}^{(as)} + (generate_{ev}, \lambda_{ev}).Buffer_{C_2-1}^{(as)} \\
&\quad +(generate_p, \lambda_{gp}).Buffer_{C_2-1}^{(as)}
\end{aligned}$$

– **Composante** $Buffer_0^{(out)}$ **:** Les paquets arrivant à la file d'attente de sortie sont soit des paquets passifs ($service_p$), soit des requêtes de code actif ($generate_{req}$) ou les paquets résultants de l'exécution d'une application ($generate_p$). L'activité $service_{out}$ modélise le service. Si C_3 est la capacité de la file d'attente de sortie, son comportement est comme suit :

$$\begin{array}{lcl} Buffer_0^{(out)} & \stackrel{def}{=} & (generate_p, \top).Buffer_1^{(out)} + (generate_{req}, \top).Buffer_1^{(out)} \\ & & +(service_p, \top).Buffer_1^{(out)} \\ Buffer_1^{(out)} & \stackrel{def}{=} & (generate_p, \top).Buffer_2^{(out)} + (generate_{req}, \top).Buffer_2^{(out)} \\ & & +(service_p, \top).Buffer_2^{(out)} + (service_{out}, \mu_{out}).Buffer_0^{(out)} \\ \vdots & & \vdots \\ Buffer_{C_3}^{(out)} & \stackrel{def}{=} & (generate_p, \top).Buffer_{C_3}^{(out)} + (generate_{req}, \top).Buffer_{C_3}^{(out)} \\ & & +(service_p, \top).Buffer_{C_3}^{(out)} + (service_{out}, \mu_{out}).Buffer_{C_3-1}^{(out)} \end{array}$$

Le réseau : Le comportement complet du réseau de files d'attente de cette configuration est donné par l'équation suivante :

$$System \stackrel{def}{=} \left(\left(\left(Buffer_0^{(in)} \underset{\mathcal{K}'}{\bowtie} Buffer_0^{(event)}\right) \underset{\mathcal{L}'}{\bowtie} Buffer_0^{(as)}\right) \underset{\mathcal{M}'}{\bowtie} Buffer_0^{(out)}\right)$$

Où les ensembles de coopération sont $\mathcal{M}' = \{service_p, generate_p, generate_{req}\}$, $\mathcal{L}' = \{service_{e1}, generate_{ev}\}$ et $\mathcal{K}' = \{service_d\}$.

5.2.2 Les Critères d'évaluation de performance

Les mesures de performance qui nous intéressent sont le débit du nœud actif pour les paquets standards, les taux de pertes de ces paquets et la latence du nœud, en particulier, pour ce type de paquets. Le calcul du taux de pertes prend en compte les pertes enregistrées au niveau des deux files d'entrée ($queue_{in}$) et de sortie ($queue_{out}$). Dans le premier cas, nous considérons les états où l'activité in échoue, états qui correspondent à une file d'attente d'entrée pleine. Dans le second cas, nous considérons les états où l'activité $service_p$ échoue, c'est-à-dire, les états qui correspondent à une file de sortie pleine.

De même, la latence du nœud actif pour les paquets standards est la latence au niveau de la file d'entrée et celle de sortie. Pour calculer ce critère de performance, les deux files d'attente sont considérées comme des files $M/M/1/C$ et la formule de calcul du temps de réponse moyen correspondante est utilisée. Soit C la capacité des files d'attente.

Si :

$$K_{in} = q_p, \;\; \rho_{in} = \frac{\lambda_{in}}{\mu_{in}} \;\; et \;\; \rho_{out} = \frac{\lambda_{out}}{\mu_{out}}$$

tel que $\lambda_{out} = K_{in} \times \mu_{in} \times (1 - P(Buffer_0^{in})) + \lambda_{gp} + q \times \mu_{e2}$ et $P(Buffer_0^{in})$ est la probabilité d'avoir la file $queue_{in}$ vide, alors :

- Le nombre moyen de paquets dans les deux files est :

$$L_{in} = \frac{\rho_{in}[1 - (C+1)\rho_{in}^C + C\rho_{in}^{C+1}]}{(1-\rho_{in})(1-\rho_{in}^{(C+1)})} \;\; et \;\; L_{out} = \frac{\rho_{out}[1 - (C+1)\rho_{out}^C + C\rho_{out}^{C+1}]}{(1-\rho_{out})(1-\rho_{out}^{(C+1)})}$$

- Le temps moyen de séjour des paquets dans les deux files est :

$$W_{in} = \frac{L_{in}}{\lambda_{in}(1 - P(Buffer_C^{in}))} \quad et \quad W_{out} = \frac{L_{out}}{\lambda_{out}(1 - P(Buffer_C^{out}))}$$

où $P(Buffer_C^{in})$ et $P(Buffer_C^{out})$ sont respectivement les probabilités d'avoir la file $queue_{in}$ et $queue_{out}$ pleine.

- Le temps d'attente moyen global des paquets passifs : $W_p = W_{in} + W_{out}$.

- Le calcul des taux de pertes Tp est comme suit :

$$Tp = K_{in} \times \lambda_{in} \times P(Buffer_C^{in}) + K_{out} \times \lambda_{out} \times P(Buffer_C^{out})$$

tel que

$$K_{out} = \frac{K_{in} \times \mu_{in} \times (1 - P(Buffer_0^{in}))}{\lambda_{out}}$$

est la proportion des paquets passifs dans le file de sortie.

- Le débit (D) des paquets passifs à la sortie du nœud se calcule comme suit :

$$D = K_{out} \times \mu_{out} \times (1 - P(Buffer_0^{out}))$$

Dans ce qui suit, nous présentons les résultats numériques obtenus pour ces critères de performance après la résolution du modèle à l'aide de l'outil PEPA Workbench [36] et PRISM [59].

5.2.3 Les résultats numériques

Pour calculer les critères de performance que nous avons définis ci-dessus, nous devons définir les paramètres d'entrée. Considérons d'abord l'arrivée externe des paquets au nœud. Sachant que ces arrivées se composent de paquets actifs et de paquets standards, nous supposons dans nos expériences que $\lambda_{in} = \lambda_{in,a} + \lambda_{in,s}$, où $\lambda_{in,a}$ est le taux d'arrivée des paquets actifs et $\lambda_{in,s}$ le taux d'arrivée des paquets standards. Comme nous nous intéressons aux performances du nœud en ce qui concerne les paquets traditionnels, nous définissons $\lambda_{in,s} = K \times \lambda_{in}$ où K est la proportion de paquets standards qui varie entre 0.2 et 0.8. On notera $\overline{K} = 1 - K$ la proportion de paquets actifs arrivant au nœud. Indépendamment de la valeur de K, les paquets actifs sont divisés entre les différents types de paquets actifs selon les proportions définies dans Table 5.1. Le code actif, représenté par les *m-paquets* et les *s-paquets*, n'intervient que dans l'exécution des règles, pas dans leur gestion ou leur lancement. La persistance du code actif dans le cache et la modularité des règles actives permettent une intense réutisabilité, induisant peu de circulation du code dans le réseau en

comparaison avec les données actives : d'où la répartition des paquets actifs définie dans la table 5.1.

Par ailleurs, en supposant que la vitesse moyenne d'un routeur est de $10Mbits/s$ et que la taille moyenne des paquets est de 250 octets, on obtient un taux de service au niveau des files du nœud de 5000 $paquets/s$, sauf pour la file d'attente $queue_{as}$ de la couche application dont le serveur est supposé un peu plus rapide avec un taux de 7000 $paquets/s$.

Dans nos expériences initiales, nous supposons que toutes les files d'attente ont la même capacité $C_i = 5$, $i = 0, \ldots, 3$. C'est la taille minimale considérée dans cette étude ; afin d'étudier l'influence de la taille des buffers sur le délai et les pertes des paquets, d'autres tailles seront considérées dans la sous-section 5.2.4. Pour limiter le délai, il est recommandé d'avoir des tailles aussi petites que possibles, quelques douzaines de paquets [70] selon le débit du système. Faire varier la valeur de cette taille à partir de 5 pour évaluer son impact sur les mesures de performance qui nous intéressent rentre tout à fait dans ce contexte [70]. A noter que pour cette capacité des buffers, la chaîne de Markov résultante a 26136 états et 214632 transitions pour le modèle 1.

Les paramètres du modèle sont récapitulés dans la table 5.1.

Répartition des paquets actifs	Taux d'arrivée (p/ms)	Taux de service (p/ms)
d-paquet : 75%	$\lambda_{ev} = 1.0$	$\mu_{in} = 5.0$
m-paquet : 12.5%	$\lambda_{oe} = 1.0$	$\mu_{out} = 5.0$
s-paquet : 12.5%	$\lambda_{gp} = 1.0$	$\mu_e = 5.0$; $\mu_{as} = 7.0$

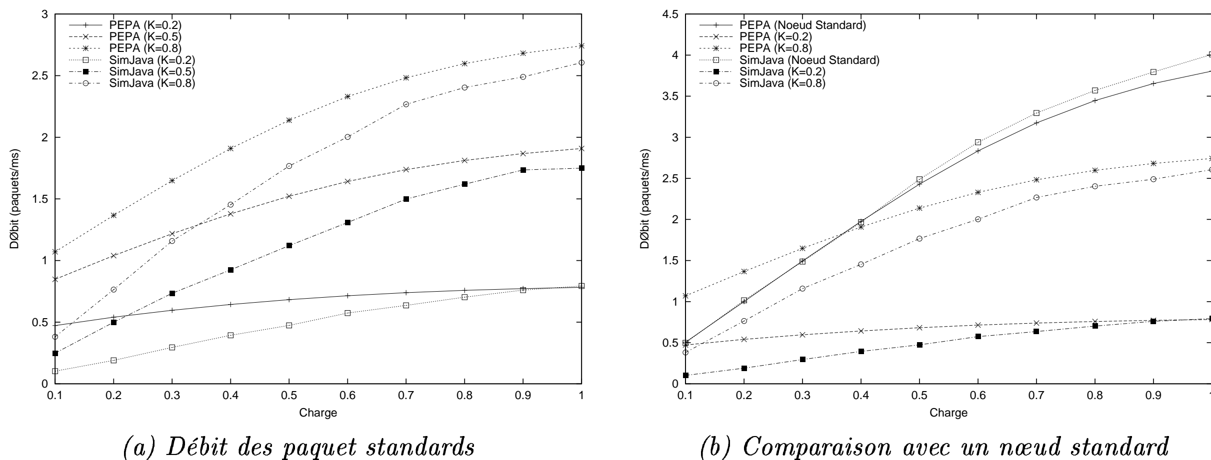
TAB. 5.1 – Valeurs des paramètres

On suppose que parmi les *d-paquets* qui arrivent au nœud 50% ($q = 0.5$) de ces paquets référencent un code non présent dans le nœud. Les résultats obtenus pour ce modèle sont décrits dans les figures 5.2, 5.3 et 5.5. Toutes les courbes sont tracées en fonction de la charge de la file d'entrée ; nous diminuons le temps des inter-arrivées des paquets ($1/\lambda_{in}$), augmentant ainsi la charge du nœud.

1. Le débit : Les premiers résultats pour ce modèle concernent le débit du nœud actif pour les paquets passifs et l'impact de la proportion de paquets actifs dans le nœud sur ce débit. Les deux figures suivantes montrent les débits mesurés par PEPA et SimJava. Les courbes des deux méthodes sont proches les unes des autres et ont les mêmes formes. Ainsi elles mènent aux mêmes conclusions que nous discutons ci-après.

- La figure 5.2(a) montre que le débit du nœud pour les paquets passifs augmente proportionnellement au pourcentage K de paquets passifs dans le nœud, ainsi qu'au taux d'arrivée λ_{in} à la file d'entrée. On note également qu'à partir d'une certaine charge, ce débit a tendance à se stabiliser et ce pour toutes les valeurs de K.

- Pour avoir une idée de l'impact des nouvelles fonctionnalités du nœud sur le débit pour les paquets passifs, nous comparons le débit d'un nœud standard avec celui du nœud actif pour $K = 0.2$ et $K = 0.8$ (figure 5.2(b)). On note que pour de très faibles charges, le débit du nœud passif et celui du nœud actif ne sont pas très différents, en particulier, pour $K = 0.8$. Mais plus la charge devient importante, plus ces deux débits diffèrent. Ce phénomène est dû au fait qu'une augmentation de la charge pour un nœud standard signifie l'arrivée de plus de paquets qui sont tous passifs. Alors que pour un nœud actif, cela signifie plus d'arrivées des deux types de paquets actifs et passifs. Cette différence est plus évidente lorsque la proportion



(a) Débit des paquet standards *(b) Comparaison avec un nœud standard*

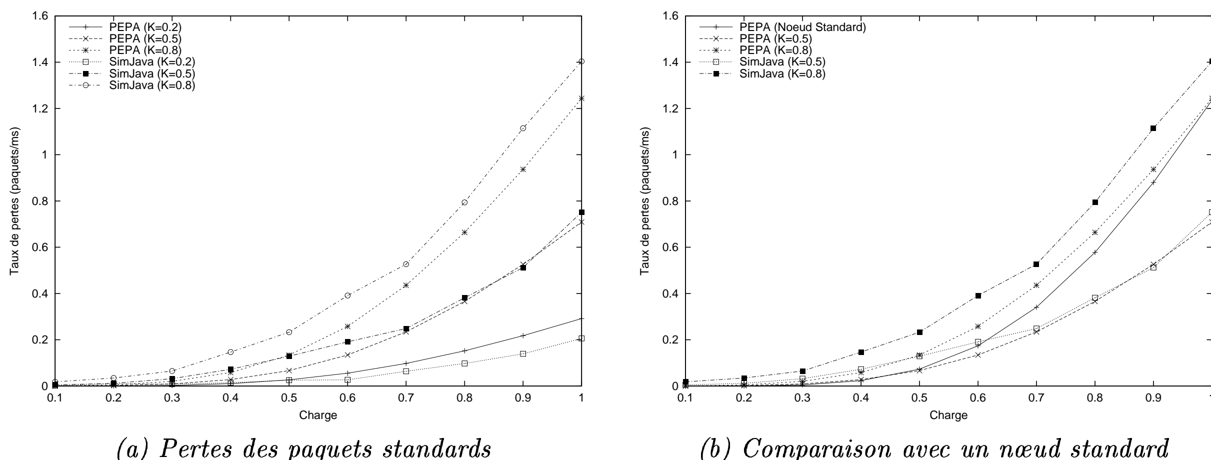
FIG. 5.2 – Impact de K sur le débit

de paquets passifs est très petite, c'est-à-dire lorsque $K = 0.2$.

Par ailleurs, la figure 5.2(b) montre que quand la charge est faible, la courbe du nœud standard est bornée par les deux autres courbes.

2. Les pertes : L'objectif de cette partie est de montrer l'impact de la présence de paquets actifs sur les pertes des paquets standards. Les figures suivantes montrent les taux de pertes mesurés par PEPA et SimJava. Comme le débit, les courbes des taux de pertes des deux méthodes sont proches les unes des autres et ont les mêmes formes.

- La figure 5.3(a) montre que quelque soit la proportion de paquets actifs dans le nœud, le taux de pertes des paquets passifs croît proportionnellement à la charge du nœud. De plus, plus K est important, plus les pertes sont importantes. Cependant, on note que lorsque la charge est faible $(0.1 - 0.5)$ les pertes sont similaires pour les différentes valeurs de K.



(a) Pertes des paquets standards *(b) Comparaison avec un nœud standard*

FIG. 5.3 – Impact de K sur les taux de pertes

Par ailleurs, on peut noter que, dans tous les cas, le taux de pertes augmente de manière exponentielle. La figure 5.4 nous montre ce phénomène qui est dû à la file d'entrée. En effet,

contrairement à la file de sortie où le taux de pertes tend à devenir constant à partir d'une certaine charge, les taux de pertes dans la file d'entrée sont exponentiels. Cette différence est due au fait que les paquets actifs arrivant à la file d'entrée restent dans le nœud et ne sont pas envoyés directement à la file de sortie.

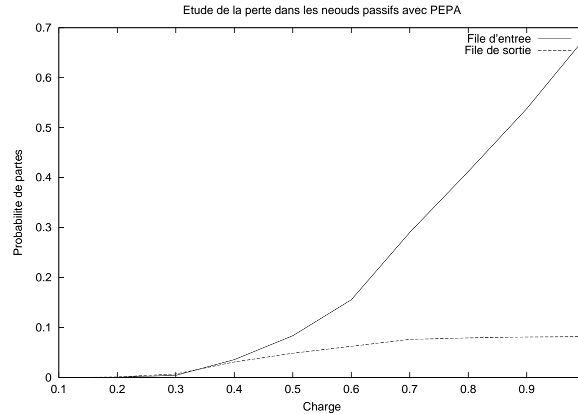


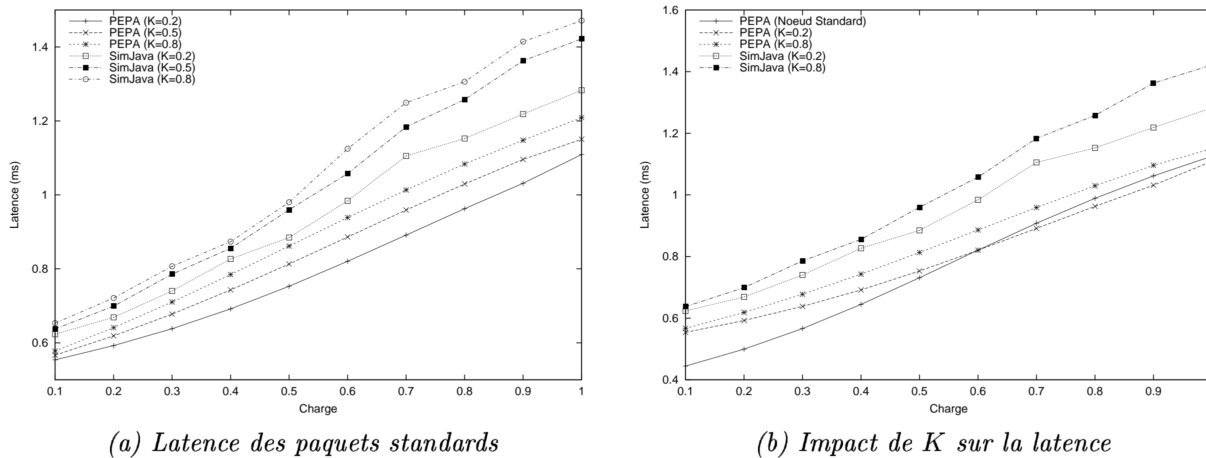
FIG. 5.4 – Pertes des paquets en entrée *versus* en sortie

- Les résultats décrits dans la figure 5.3(b) sont particulièrement intéressants. Dans cette figure, nous comparons le taux de pertes des paquets standards dans un nœud actif avec celui obtenu pour le nœud standard. La figure montre que les taux de pertes pour le nœud standard sont bornés par les taux de pertes quand $K = 0.5$ et ceux quand $K = 0.8$. La première peut être considérée comme la borne inférieure et la seconde comme la borne supérieure. Ce résultat suggère que ces pertes peuvent être contrôlées pour ne pas dépasser une certaine limite en choisissant la bonne proportion de paquets actifs à admettre dans le nœud.

3. La latence : Les derniers résultats concernent la latence des paquets passifs dans un nœud actif mesurée analytiquement avec PEPA et par simulation avec SimJava. Les courbes résultantes sont très proches.

- Dans la figure 5.5(a), nous comparons la latence dans un nœud actif pour différentes valeurs de K. Cette figure montre que plus la proportion de paquets passifs augmente, plus la latence augmente. Comme pour les pertes, le calcul de la latence considère les deux files $queue_{in}$ et $queue_{out}$. Contrairement à la file d'entrée, la file de sortie ne reçoit des paquets qui arrivent au nœud que les paquets à retransmettre (standards). Aussi, si on diminue la proportion de paquets actifs arrivant au nœud, augmentant ainsi celle des paquets standards qui y arrivent, la latence au niveau de la file de sortie augmente. Alors que cela ne change rien à la latence au niveau de la file d'entrée puisque le taux total d'arrivées reste inchangé. Cependant, la différence entre les résultats obtenus pour les différentes valeurs de K n'est pas très significative.
- Dans la figure 5.5(b), nous comparons la latence d'un paquet passif dans un nœud actif pour $K = 0.2$ et $K = 0.8$ avec la latence dans un nœud standard. Bien que cette figure suggère que la latence dans un nœud actif est toujours supérieure pour $K > 0.2$ à forte charge et pour $K \geq 0.2$ à faible charge, la différence avec un nœud standard n'est pas très importante.

Pour voir si les résultats obtenus pour les nœuds actifs avec une configuration de type *Configuration 1* de ce type se confirment, nous étudions ci-dessous l'impact de la capacité des files d'attente



(a) Latence des paquets standards

(b) Impact de K sur la latence

FIG. 5.5 – La latence

sur les pertes et la latence.

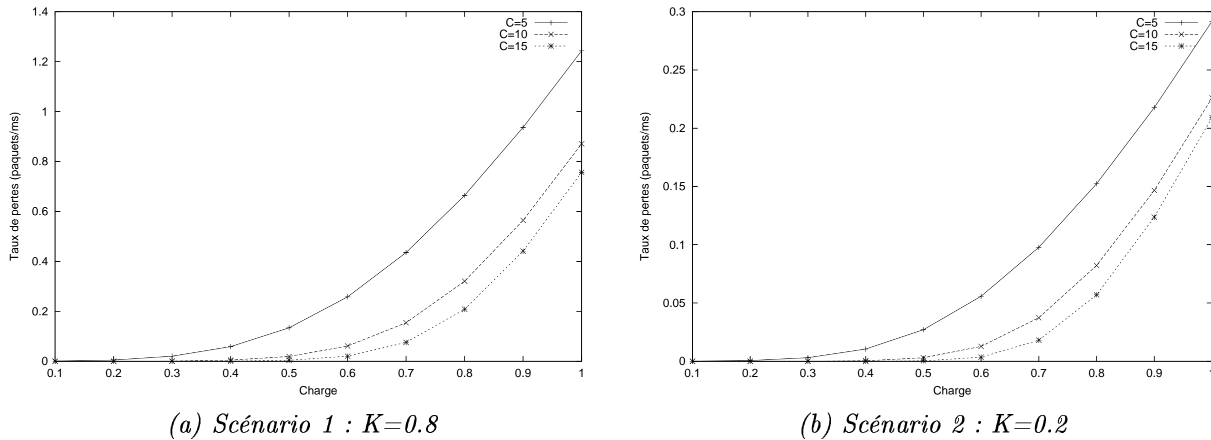
5.2.4 L'impact des tailles des files d'attente

Dans cette partie, nous étudions l'impact de l'augmentation de la taille des files d'attente sur le taux de pertes des paquets standards et la latence d'un nœud actif pour ce type de paquets. Pour cela, nous considérons deux cas, $C_i = 10$ et $C_i = 15$, $i = 0, \ldots, 3$. Comme toutes les files ont la même capacité, nous notons C la taille de ces files. Les résultats obtenus sont présentés dans les figures 5.6, 5.7 et 5.8. A noter qu'avec des files de grandes tailles telles que 15 on arrive à de très grands nombres d'états pour le modèle 1 (126976 états).

1. Les pertes : la figure 5.6 montre l'impact de la taille des buffers sur la perte des paquets standards pour $K = 0.2$ et $K = 0.8$.

- Dans la figure 5.6(a), nous comparons le taux de pertes des paquets standards pour les trois tailles des buffers $C = 5, 10$ *et* 15 et pour $K = 0.8$. Cette figure montre que les taux de pertes croient exponentiellement en fonction de la charge, et cela quelque soit la taille des buffers. De plus, ces taux diminuent logiquement à mesure que la capacité des buffers augmente. Par ailleurs, on note que la différence entre les taux de pertes quand $C = 10$ et $C = 15$ est négligeable, en particulier pour une charge entre 0.1 et 0.6. Cette différence est toujours plus petite que la différence entre les pertes quand $C = 10$ et $C = 5$. Cela suggère que si nous continuons à augmenter la taille des buffers, le taux de pertes peut continuer à diminuer. Cependant, il semble probable que nous atteindrons une capacité des buffers à laquelle ce taux ne diminuera plus de manière aussi significative.

- La figure 5.6(b) montre les résultats obtenus pour $K = 0.2$, c'est-à-dire lorsque le trafic des paquets standards est très perturbé. Comme dans la figure 5.6(a), nous pouvons voir que le taux de pertes est inversement proportionnelle à la taille des buffers. Cependant, nous pouvons voir aussi que les taux de pertes sont très petits comparés à ceux obtenus pour $K = 0.8$. Ceci est dû aux pertes enregistrées dans la file de sortie. Comme tous les paquets standards sont dirigés vers la file d'attente de sortie, quand $K = 0.8$ il est plus probable que cette file d'attente soit pleine et plus de paquets standards soient perdus.



(a) Scénario 1 : K=0.8 *(b) Scénario 2 : K=0.2*

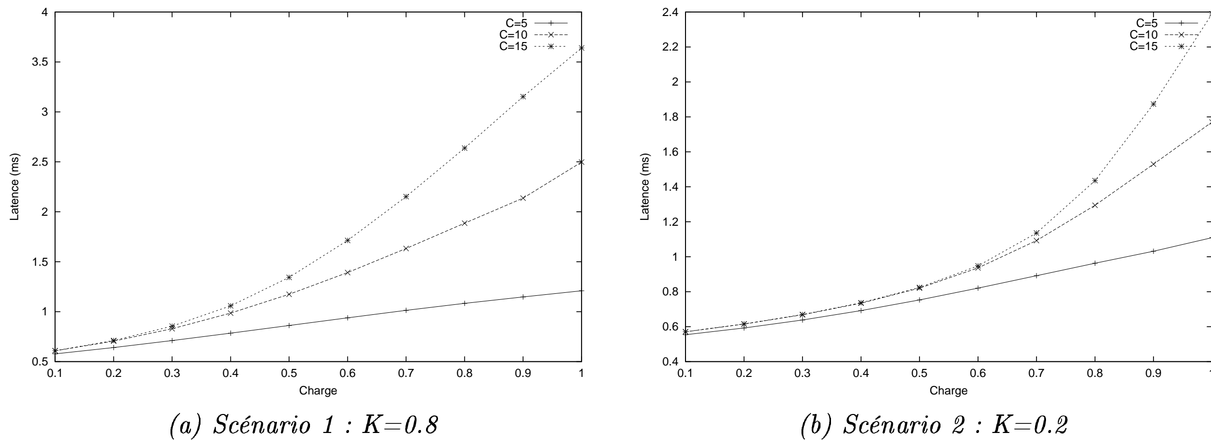
FIG. 5.6 – Les taux de pertes

2. La latence : La figure 5.7 montre l'impact de la taille des buffers sur la latence. Comme précédemment, nous considérons deux scénarios : $K = 0.2$ et $K = 0.8$.

- Dans la figure 5.7(a), nous comparons la latence obtenue pour les différentes tailles de buffers. La proportion des paquets actifs est $\overline{K} = 0.2$, c'est-à-dire que le trafic des paquets standards est légèrement perturbé. Cette figure montre que pour de faibles charges, la taille des buffers n'a pas un grand impact sur la latence, mais dès que la charge devient plus importante la latence augmente proportionnellement à la taille des buffers.
- La figure 5.7(b) présente les résultats obtenus quand la proportion des paquets actifs est $\overline{K} = 0.8$, c'est-à-dire quand le trafic des paquets actifs est beaucoup plus important que le trafic des paquets standards. Elle montre que la capacité des buffers semble avoir un impact plus modéré que pour $\overline{K} = 0.2$. En comparant les deux scénarios, on remarque que plus la taille des buffers augmente, plus la différence entre les résultats des deux scénarios est importante. Cela est dû au fait que les paquets actifs arrivant au nœud vont devoir rester dans le nœud pour un traitement ultérieur, alors que les paquets passifs sont tout de suite acheminés vers la file de sotie. Lorsque $K = 0.8$, 80% du trafic est orienté vers la file de sortie, ce qui explique une plus grande latence, comparé au cas où uniquement 20% ($K = 0.2$) du trafic est immédiatement acheminé vers cette même file. De plus, plus la taille des buffers augmente, plus ce phénomène s'accentue.

3. Nœud actif ***versus*** **nœud passif :** Dans ce qui suit, nous comparons, d'une part, le taux de pertes dans un nœud standard avec le taux de pertes dans un nœud actif, et d'autre part, la latence d'un paquet standard dans les deux types de nœuds. Ces comparaisons sont faites pour une capacité des buffers $C = 10$ et pour différentes valeurs de K : 0.2, 0.5, 0.8.

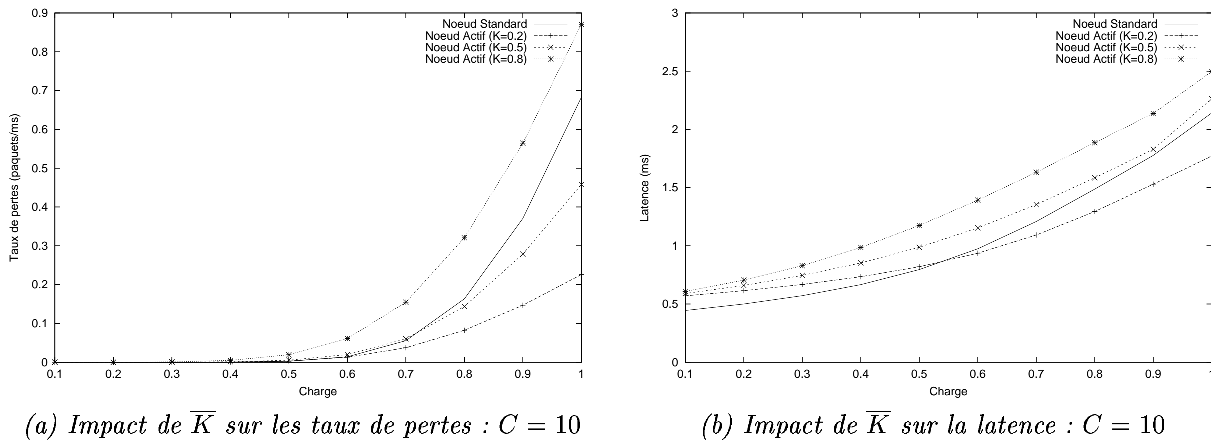
- Les résultats obtenus dans la figure 5.8(a) pour les pertes confirment les résultats décrits dans la figure 5.3(b) pour $C = 5$. A savoir que le taux de pertes dans un nœud standard est borné par le taux de pertes dans un nœud actif. De nouveau ceci suggère que la perte de paquets passifs dans un nœud actif dépend de la proportion des paquets actifs arrivant au nœud. Et ces pertes peuvent être contrôlées pour ne pas dépasser une certaine limite si nous choisissons la



(a) Scénario 1 : K=0.8 *(b) Scénario 2 : K=0.2*

FIG. 5.7 – La latence

bonne proportion de paquets actifs à admettre dans le nœud. Nous pouvons voir que $K = 0.8$ peut être utilisé comme une borne supérieure comme le montre la figure 5.3(b). Cependant la figure 5.8(a) montre qu'il est préférable d'utiliser $K = 0.5$ plutôt que $K = 0.2$ comme borne inférieure.



(a) Impact de $\overline{K}$ sur les taux de pertes : $C = 10$ *(b) Impact de $\overline{K}$ sur la latence : $C = 10$*

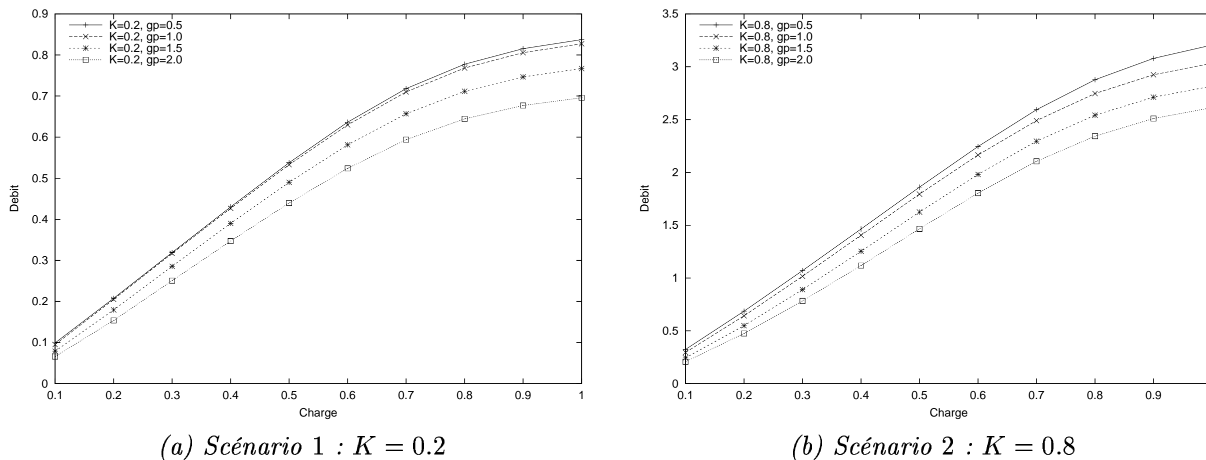
FIG. 5.8 – Nœud standard *versus* Nœud actif

- Comme pour le cas $C = 5$ (Fig 5.5), les nouvelles fonctionnalités du nœud pénalisent très peu les paquets standards et ce quelque soit la valeur de K. On peut noter que pour de fortes charges, la latence dans le nœud standard est bornée par la latence quand $K = 0.2$ (borne inférieure) et $K = 0.5$ (borne supérieure).

5.2.5 L'impact de la génération des paquets actifs

Pour étudier l'impact des paquets actifs résultant de l'exécution des règles actives dans la couche services d'application, nous considérons différentes valeurs du taux des arrivées λ_{gp} à la file de sortie :

0.5, 1.0, 1.5, et 2. Les résultats obtenus pour la capacité des files $C = 10$ sont présentés dans les figures 5.9-5.12.



(a) Scénario 1 : $K = 0.2$ *(b) Scénario* 2 : $K = 0.8$

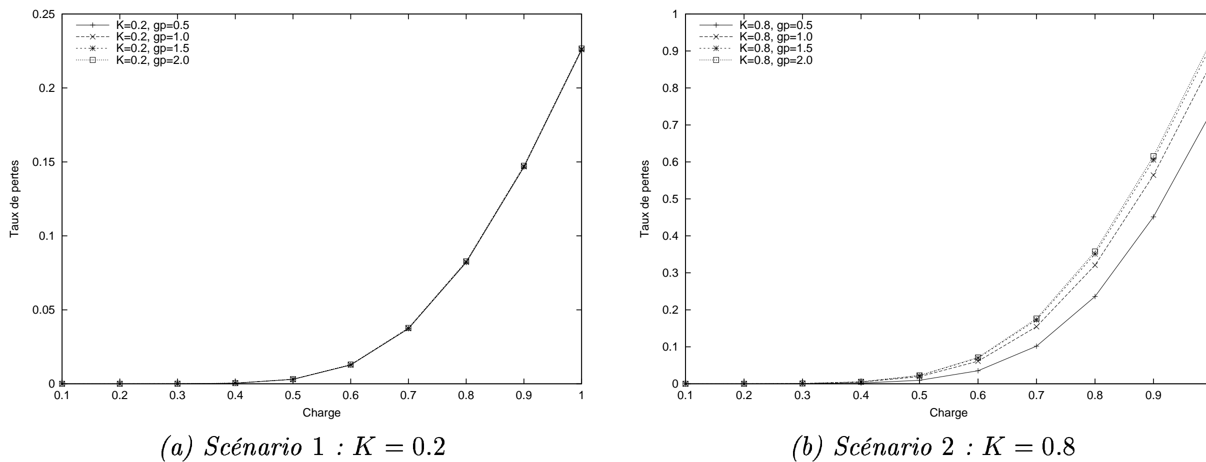
FIG. 5.9 – Le débit des paquets standards en fonction de la génération de paquets actifs

1. Le débit : Les figures 5.9(a) et (b) montrent le débit obtenu pour $K = 0.2$ et $K = 0.8$ respectivement. Dans les deux scénarios, le débit diminue à mesure que λ_{gp} augmente. D'ailleurs cette diminution devient plus significative à mesure que ce taux augmente, en particulier, quand le nœud est chargé et le trafic des paquets standards est très perturbé ($K = 0.2$).

La génération de paquets est proportionnelle au nombre de paquets actifs dans le nœud. Plus les paquets actifs sont nombreux dans le nœud, plus la génération des paquets actifs dans ce nœud est plus probable.

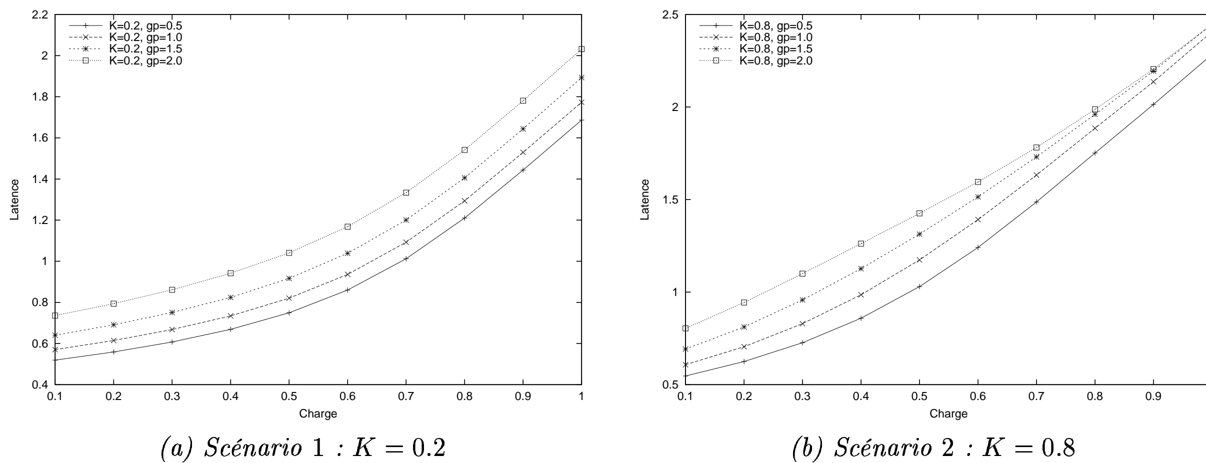
2. Le taux de pertes : La figure 5.10(a) montre le taux de pertes des paquets passifs quand $K = 0.2$ pour toutes les valeurs de λ_{gp}. Dans cette figure, toutes les courbes sont superposées pour toutes les valeurs de la charge et de λ_{gp}. Ce résultat implique que la génération des paquets actifs à l'intérieur du nœud n'a aucun effet sur les pertes quand la proportion des paquets passifs est très petite. On observe le même phénomène quand $K = 0.8$ (Fig 5.10(b)), mais seulement quand la charge est petite (en-dessous de $0,5$). Cependant, la figure 5.10(b) prouve qu'à mesure que λ_{gp} augmente, l'effet sur les pertes devient rapidement insignifiant.

3. La latence : Comme pour les deux mesures de performance précédentes, nous analysons la latence du nœud pour les paquets passifs pour toutes les valeurs de λ_{gp}. La figure 5.11(a) montre la latence quand la proportion des paquets passifs est $K = 0.2$. Dans cette figure, toutes les courbes ont le même comportement, mais lorsque λ_{gp} augmente, la latence augmente aussi. L'augmentation de la latence devient plus significative à mesure que λ_{gp} augmente. Quand $K = 0.8$, on peut observer le même phénomène dans la figure 5.11(b). Cependant celui-ci prouve que lorsque le nœud devient chargé la différence entre les latences devient peu significative à mesure que λ_{gp} augmente.



(a) Scénario 1 *:* $K = 0.2$ *(b) Scénario* 2 *:* $K = 0.8$

FIG. 5.10 – Le taux de pertes des paquets standards en fonction de la génération de paquets actifs

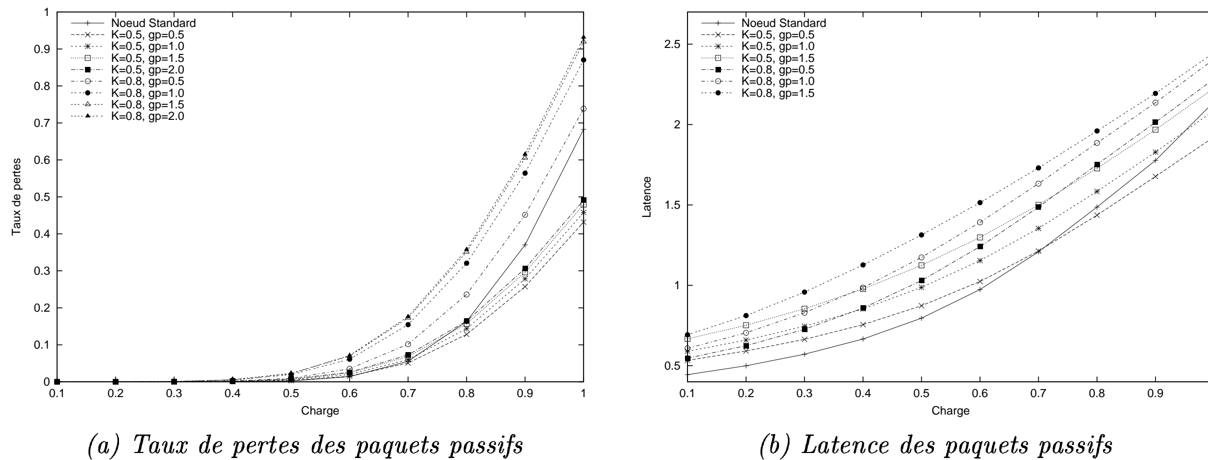


(a) Scénario 1 *:* $K = 0.2$ *(b) Scénario* 2 *:* $K = 0.8$

FIG. 5.11 – La latence des paquets standards en fonction de la génération de paquets

4. Nœud standard *versus* **nœud actif :** Afin de comparer le taux de pertes et la latence des paquets passifs dans un nœud d'ARFANet avec ceux dans un nœud standard, nous faisons varier le taux de génération des paquets actifs pour les deux proportions $K = 0.5$ et $K = 0.8$.

- Dans la figure 5.12(a) nous pouvons voir que la variation de λ_{gp} n'a presque aucun impact sur les pertes quand $K = 0.5$. Cette proportion peut être considérée comme une borne inférieure, en particulier, pour $\lambda_{gp} = 0.5$. Ceci confirme les résultats précédents que nous avons obtenus (section 5.2.3). De même, cette figure montre que $K = 0.8$ peut être considéré comme une borne supérieure pour les pertes, mais pour $\lambda_{gp} = 0.5$. Si une plus grande valeur de λ_{gp} est considérée, nous pouvons choisir une valeur plus petite que $K = 0,8$.
- La figure 5.12(b) donne la latence pour les trois premières valeurs de λ_{gp} (0.5, 1.0, 1.5) et des proportions des paquets standards $K = 0.5$ et $K = 0.8$. Cette latence est comparée à celle d'un nœud standard. Nous pouvons voir que pour les deux valeurs de K, les meilleurs résultats sont obtenus quand $\lambda_{gp} = 0.5$. Une autre valeur de λ_{gp} peut être employée, comme



(a) Taux de pertes des paquets passifs *(b) Latence des paquets passifs*

FIG. 5.12 – Nœud standard *versus* nœud actif en fonction de la génération des paquets actifs

1.0, mais nous devrons alors réduire la valeur de K.

La proportion des paquets passifs à l'entrée du nœud n'est pas forcément identique à la sortie. La figure 5.13 le montre bien. Lorsque la proportion des paquets en entrée est fixe, la proportion en sortie varie en fonction des taux d'arrivée et de la proportion en entrée. On remarque que pour $K = 0.8$ la proportion des paquets passifs à la sortie du nœud n'atteint pas 65% ($K_{out} = 0.65$). Cela veut dire que le nœud génère plus de paquets actifs qu'il n'en reçoit. D'après la figure nous pouvons remarquer que lorsque la proportion des paquets passifs est petite à l'entrée du nœud et puisque tous les paquets actifs se dirigent vers la file des événements, dans la file de sortie il n'y aura que les paquets passifs et les paquets générés. Pour cette raison, la proportion des paquets passifs augmente. Cependant, ce phénomène s'inverse lorsque la proportion en entrée des paquets passifs est grande. L'augmentation du taux de génération des paquets actifs influe sur cette proportion en sortie en la diminuant. Cette figure nous permet de conclure que ce n'est pas le taux des paquets actifs à l'entrée qu'il faut maîtriser, mais plutôt le taux de génération des paquets actifs vers la file de sortie. Ce taux de génération est évidemment indépendant du taux les paquets actifs à l'entrée du nœud puisqu'il est lié exclusivement à l'exécution des applications. Le contrôle doit donc être fait au niveau des moniteurs actifs qui doivent maintenir ce taux à un certain seuil.

Ces expériences nous permettent de montrer que la génération des paquets actifs dans le nœud a un impact sur les performances d'un nœud pour des paquets standards. Cependant, cet impact n'est pas significatif pour les pertes parce que la plupart d'entre elles se produisent dans la file d'entrée, tandis que les paquets actifs produits sont mis dans la file d'attente de sortie. La génération de nouveaux paquets actifs par le nœud influence de manière significative la latence des paquets passifs parce que la présence de plus de paquets dans la file d'attente de sortie retarde le service des paquets standards.

Nous pouvons conclure que le taux de génération des paquets actifs et la proportion de ces paquets arrivant au nœud sont étroitement liés et le choix de l'un a un impact sur le choix de l'autre. D'ailleurs tous les deux doivent être limités à une certaine valeur afin de réaliser des performances

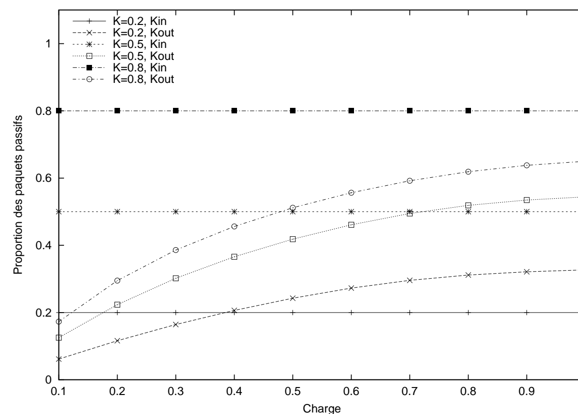


FIG. 5.13 – Proportion des paquets en entrée *versus* en sortie

comparables à celles d'un nœud standard.

Bilan :

A partir de cette étude des performances d'un nœud actif, nous pouvons conclure que l'introduction des paquets actifs dans un réseau a une incidence sur le comportement des paquets passifs. Nous avons pu remarquer que la proportion des paquets actifs a une influence sur les paquets standards mais la taille des files d'attente et la génération des paquets ont aussi une influence sur ce genre de paquets. Pour la taille des files, l'augmentation de la taille jusqu'à une certaine valeur à laquelle ce taux ne diminuera plus de manière aussi significative, mais elle augmente la latence. Le compromis entre le taux de pertes et la latence est lié au choix de la taille. Nous remarquons que la taille $C = 10$ est une taille qui satisfait ce compromis. En ce qui concerne la génération des paquets actifs, notre étude nous a révélés qu'elle n'influe pas beaucoup sur la perte des paquets car les paquets générés sont orientés directement vers la sortie. De même pour la latence. Néanmoins, un contrôle du nombre de paquets générés par les couches supérieures, comme les moniteurs actifs, pour qu'il ne dépasse pas un certain seuil est recommandé.

Dans la configuration de la figure 5.1, tous les *d-paquets* sont acheminés vers la file des évènements ($queue_{event}$). Lorsqu'un de ces paquets est servi dans la file, s'il référence un code non présent dans le nœud, une requête de code actif est générée et envoyée vers la file de sortie, et le *d-paquet* est remis dans la file. Cela peut avoir pour conséquence la perte de ce genre de paquet car la file $queue_{event}$ est rapidement saturée. De plus, la génération de plusieurs requêtes de code par *d-paquet* provoque la présence d'un plus grand nombre de paquets de requêtes dans la file de sortie. Ceci peut être la cause de la perte de certains paquets standards et la cause d'une plus grande latence dans la file de sortie.

Dans ce qui suit, nous proposons de modifier partiellement notre configuration pour tenir compte de cet aspect. Pour distinguer cette nouvelle configuration de la précédente, elle sera notée *Configuration 2*.

5.3 Amélioration du Modèle

Dans cette configuration (Fig 5.14), une file supplémentaire $queue_{dw}$ est introduite. Elle permet la sauvegarde des *d-paquets* qui référencent un code actif *s-paquet* ou *m-paquet* non présent dans le nœud. L'absence des paquets de code référencés par certains *d-paquets* n'est plus détectée au niveau de la file des évènements, mais plutôt au niveau de la file d'entrée. Dans ce cas, le paquet est orienté vers la file $queue_{dw}$ et une requête des paquets absents est envoyée au réseau via la file de sortie $queue_{out}$. Les autres files de cette nouvelle configuration sont similaires à celles dans *Configuration 1*.

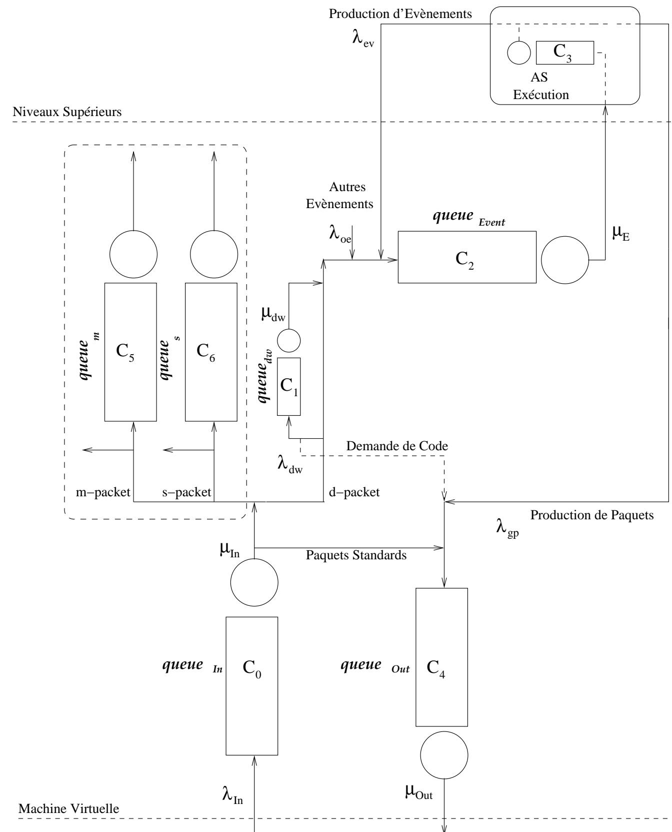


FIG. 5.14 – Réseau de files d'attente modélisant un nœud actif : *Configuration 2*

5.3.1 Le modèle PEPA

Le modèle PEPA correspondant à la seconde configuration comprend, en plus des composantes du modèle précédent, la composante $Buffer_0^{(dw)}$ qui modélise la file $queue_{dw}$. Les composantes $Buffer_0^{(out)}$ et $Buffer_0^{(as)}$ sont identiques à celles du modèle précédent. Cependant la présence de la file $queue_{dw}$ dans cette configuration apporte des changements au niveau des autres composantes, c'est-à-dire $Buffer_0^{(in)}$ et $Buffer_0^{(event)}$.

- **Composante** $Buffer_0^{(in)}$ **:** Nous utilisons $service_{dw}$ pour modéliser le service d'un *d-paquet* qui référence un paquet de code inexistant. A cette activité nous associons la probabilité

q_{dw}. La génération d'une requête pour un paquet de code absent est modélisée avec l'action $generate_{req}$.

$$\begin{array}{lcl}
Buffer_0^{(in)} & \stackrel{def}{=} & (in, \lambda_{in}).Buffer_1^{(in)} \\
Buffer_1^{(in)} & \stackrel{def}{=} & (in, \lambda_{in}).Buffer_2^{(in)} + (service_d, q_d \times \mu_{in}).Buffer_0^{(in)} \\
& & +(service_p, q_p \times \mu_{in}).Buffer_0^{(in)} + (service_s, q_s \times \mu_{in}).Buffer_0^{(in)} \\
& & +(service_{dw}, q_{dw} \times \mu_{in}).(generate_{req}, \mu_{rq}).Buffer_0^{(in)} \\
& & +(service_m, q_m \times \mu_{in}).Buffer_0^{(in)} \\
\vdots & & \vdots \\
Buffer_{C_0}^{(in)} & \stackrel{def}{=} & (in, \lambda_{in}).Buffer_{C_0}^{(in)} + (service_d, q_d \times \mu_{in}).Buffer_{C_0-1}^{(in)} \\
& & +(service_p, q_p \times \mu_{in}).Buffer_{C_0-1}^{(in)} + (service_s, q_s \times \mu_{in}).Buffer_{C_0-1}^{(in)} \\
& & +(service_{dw}, q_{dw} \times \mu_{in}).(generate_{req}, \mu_{rq}).Buffer_{C_0-1}^{(in)} \\
& & +(service_m, q_m \times \mu_{in}).Buffer_{C_0-1}^{(in)}
\end{array}$$

- **Composante** $Buffer_0^{(event)}$ **:** Comme tous les paquets dans la file sont du même type, à savoir des *d-paquets* dont le code actif correspondant est présent dans le nœud, le service dans $queue_{event}$ est modélisé par une seule action $service_e$. Soit C_1 la capacité de cette file, son comportement est comme suit :

$$\begin{array}{lcl}
Buffer_0^{(event)} & \stackrel{def}{=} & (service_d, \top).Buffer_1^{(event)} + (service_{ew}, \top).Buffer_1^{(event)} \\
& & +(generate_{ev}, \top).Buffer_1^{(event)} + (in_{oe}, \lambda_{oe}).Buffer_1^{(event)} \\
Buffer_1^{(event)} & \stackrel{def}{=} & (service_d, \top).Buffer_2^{(event)} + (service_{ew}, \top).Buffer_2^{(event)} \\
& & +(generate_{ev}, \top).Buffer_2^{(event)} + (in_{oe}, \lambda_{oe}).Buffer_2^{(event)} \\
& & +(service_e, \mu_e).Buffer_0^{(event)} \\
\vdots & & \vdots \\
Buffer_{C_1}^{(event)} & \stackrel{def}{=} & (service_d, \top).Buffer_{C_1}^{(event)} + (service_{ew}, \top).Buffer_{C_1}^{(event)} \\
& & +(generate_{ev}, \top).Buffer_{C_1}^{(event)} + (in_{oe}, \lambda_{oe}).Buffer_{C_1}^{(event)} \\
& & +(service_e, \mu_e).Buffer_{C_1-1}^{(event)}
\end{array}$$

- **Composante** $Buffer_0^{(dw)}$ **:** Les paquets de données dont le code actif correspondant est absent du nœud sont stockés dans la file d'attente $queue_{dw}$ ($service_{dw}$). L'action $service_{ew}$ modélise le service dans $queue_{dw}$ des *d-paquets* dont le code est finalement arrivé au nœud.

$$\begin{array}{lcl}
Buffer_0^{(dw)} & \stackrel{def}{=} & (service_{dw}, \top).Buffer_1^{(dw)} \\
Buffer_1^{(dw)} & \stackrel{def}{=} & (service_{dw}, \top).Buffer_2^{(dw)} + (service_{ew}, \mu_{dw}).Buffer_0^{(in)} \\
\vdots & & \vdots \\
Buffer_{C_4}^{(dw)} & \stackrel{def}{=} & (service_{dw}, \top).Buffer_{C_4}^{(dw)} + (service_{ew}, \mu_{dw}).Buffer_{C_4-1}^{(dw)}
\end{array}$$

Le réseau : Le comportement complet du réseau de files d'attente de cette configuration est donné par l'équation suivante :

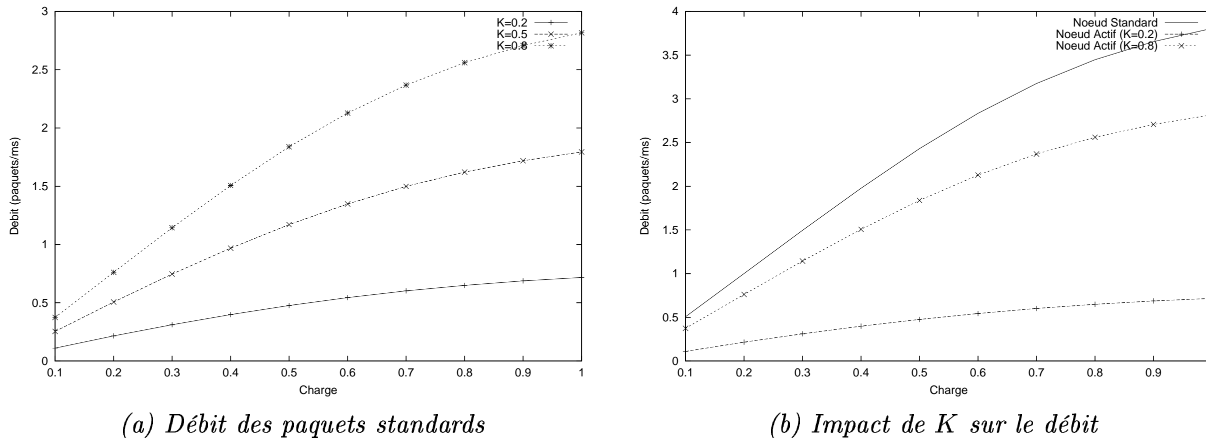
$$System \stackrel{def}{=} \left(\left(\left(Buffer_0^{(in)} \underset{\mathcal{K}}{\bowtie} Buffer_0^{(event)}\right) \underset{\mathcal{L}}{\bowtie} Buffer_0^{(as)}\right) \underset{\mathcal{M}}{\bowtie} Buffer_0^{(out)}\right) \underset{\mathcal{N}}{\bowtie} Buffer_0^{(dw)}$$

Où les ensembles de coopération sont $\mathcal{M} = \{service_p, generate_p, generate_{req}\}$, $\mathcal{K} = \{service_d\}$, $\mathcal{L} = \{service_{e1}, generate_{ev}\}$ et $\mathcal{N} = \{service_{dw}, service_{ew}\}$.

5.3.2 Les résultats numériques

On suppose que le taux de service dans la file d'attente $queue_{dw}$ est $\mu_{dw} = 7.0$. Comme dans le modèle précédent, on suppose que, parmi les *d-paquets* arrivant au nœud, 50% ($q_{dw} = 0.5$) sont orientés vers la file $queue_{dw}$. De plus, on suppose que $\mu_{rq} = 2.0$. Les résultats obtenus pour ce modèle sont décrits dans les figures 5.15, 5.16 et 5.17.

1. Le débit : La première partie de nos résultats concerne le débit du nœud actif pour les paquets standards. La figure 5.15 montre que ces résultats sont très similaires à ceux obtenus pour *Configuration 1*. Cependant, en comparant la figure 5.15(b) avec son équivalente dans le premier modèle (Fig 5.2(b)), on note une petite différence lorsque la charge est faible ($0.1 - 0.4$). Le débit est toujours inférieur à celui dans le nœud actif.



(a) Débit des paquets standards

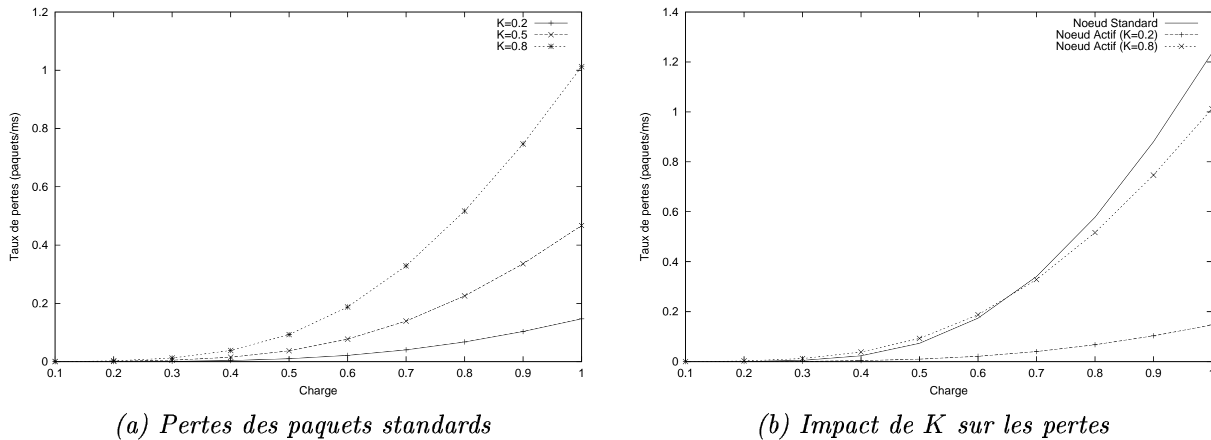
(b) Impact de K sur le débit

FIG. 5.15 – Le débit

2. Les pertes : La seconde partie de nos résultats concerne l'impact de la proportion de paquets actifs sur les taux de pertes des paquets passifs. La figure 5.16 illustre les résultats obtenus.

- Bien qu'ayant le même comportement que celles dans la figure 5.16(a), les courbes décrites dans la figure 5.3(a) ne sont pas de la même grandeur. On remarque en effet que moins de paquets sont perdus dans le second modèle (*Configuration 2*). Ainsi pour $K = 0.8$, le taux de pertes peut baisser de près de 30% lorsque la charge est très importante.

- Dans la figure 5.16(b), nous comparons les pertes dans un nœud standard avec les pertes dans un nœud actif pour $K = 0.2$ et $K = 0.8$. Lorsque la charge est faible ($0.1 - 0.5$) et que le trafic des paquets standards est très perturbé ($K = 0.2$), les pertes sont similaires à celles du nœud standard. Par contre, dès que la charge commence à être importante, on enregistre moins de pertes dans le nœud actif. Encore une fois, ceci est dû au fait qu'une augmentation de la charge pour un nœud actif implique une augmentation des paquets passifs et des paquets actifs, et en particulier ces derniers lorsque $K = 0.2$.



(a) Pertes des paquets standards *(b) Impact de K sur les pertes*

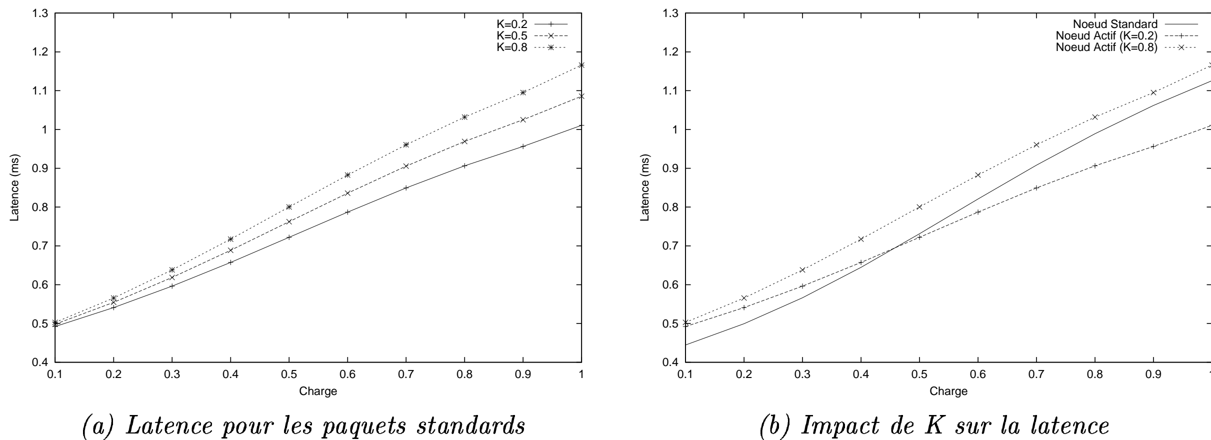
FIG. 5.16 – Les taux de pertes

Par contre, le comportement des pertes lorsque le trafic des paquets standards est peu perturbé ($K = 0.8$) est très similaire à celui des pertes dans le nœud standard. A très forte charge, on note cependant le même phénomène que pour $K = 0.2$, mais en beaucoup moins accentué.

3. La latence : Les derniers résultats concernent la latence du nœud pour les paquets passifs.

– La figure 5.17(a) montre que le taux des paquets actifs dans le nœud n'a pas un grand impact sur la latence du nœud pour les paquets passifs. Par ailleurs, on note que la latence des paquets standards est légèrement plus petite que dans *Configuration 1*. Comme pour les pertes, cette différence est due à l'introduction de la file $queue_{dw}$ dans le second modèle, mais contrairement aux pertes, cette différence n'est pas très significative.

– En comparant la latence du nœud passif avec celle du nœud actif (Fig 5.17(b)), on voit que, dans le nœud actif, les paquets passifs subissent une latence très proche de celle dans le nœud passif.



(a) Latence pour les paquets standards *(b) Impact de K sur la latence*

FIG. 5.17 – La latence

Bilan :

La différence entre les deux modèles réside dans le taux de génération des requêtes de codes et à quel niveau la vérification de la présence ou de l'absence d'un code actif référencé est opérée. Le premier modèle génère plus de demandes et augmente la charge de la file de sortie.

Les résultats du modèle 2 sont nettement meilleurs que ceux du modèle 1, pour les trois mesures de performances considérées : le débit, la latence et le taux de pertes. L'ajout de la file s'avère bénéfique pour la machine virtuelle. De ce fait, nous choisissons pour l'implémentation réelle la deuxième configuration.

5.4 Résultats analytiques *versus* mesures réelles

Dans cette partie nous comparons les résultats obtenus avec le modèle PEPA des deux configurations *Configuration 1* et *Configuration 2* avec les mesures effectuées sur un nœud ARFANet.

Pour pouvoir faire cette comparaison, nous avons dû calculer le taux de service réel de l'infrastructure et adapter les paramètres du modèle PEPA à ce nouveau taux service (26 $paquets\ actifs/s$, nécessitant un traitement actif). Dans le contexte de la comparaison d'ARFANet avec ANTS (section 5.5) nous avons pu atteindre 1550 $paquets\ actifs/s$.

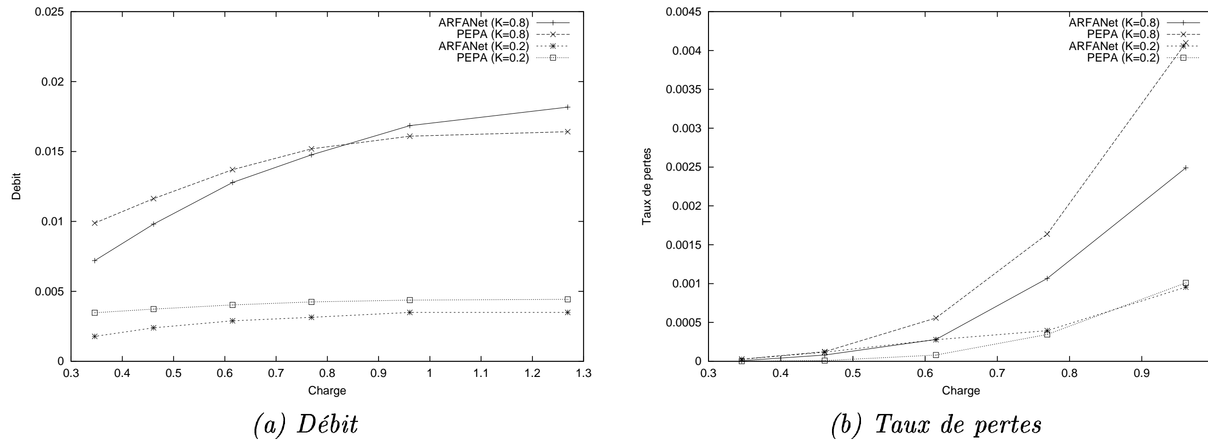
Nos tests ont été faits sur une machine Linux avec un processeur "AMD Athlon(tm) XP 1500+" de 1339 MHz, 256 Ko de mémoire cache et une mémoire vive de 256 Mo.

5.4.1 Le modèle de Configuration 1

La figure 5.18 montre les résultats obtenus pour le débit et le taux de pertes pour une taille des files C=10 et pour les deux proportions de paquets passifs $K = 0.2$ et $K = 0.8$. Les limites matérielles et logicielles nous ont contraint à limiter la charge de la file d'entrée entre 0.3 et 1.3.

- La figure 5.18(a) représente le débit du nœud pour les paquets standards obtenu avec le modèle PEPA, d'une part, et mesuré sur un nœud ARFANet, d'autre part. Cette figure montre que les deux débits sont très proches et ceci quelque soit le degré de perturbation du trafic des paquets standards (K). Lorsque $K = 0.2$, le débit du modèle PEPA et celui d'ARFANet ont le même comportement, le premier étant toujours légèrement supérieur au second. Par contre lorsque le trafic des paquets standard est très peu perturbé ($K = 0.8$), le débit du modèle PEPA est supérieur uniquement dans la première moitié de la charge. Dans la seconde moitié la tendance s'inverse.
- Pour les pertes, on voit, dans la figure 5.18(b), que lorsque $K = 0.2$ les pertes du modèle PEPA sont très similaires à celles de l'implémentation, et ceci quelque soit la charge en entrée du nœud. Par contre lorsque $K = 0.8$, les deux taux de pertes sont initialement très proches ($charge < 0.7$). Plus la charge augmente, plus l'écart entre les deux taux de pertes augmente, le taux de pertes du modèle PEPA étant plus important. A noter que, dans un souci de clarté de la première partie de la figure ($0.3 - 0.7$), on se limite dans cette figure à une charge maximale égale à 1.0.

Bilan :



(a) Débit *(b) Taux de pertes*

FIG. 5.18 – ARFANet *versus* PEPA

Si l'étude analytique et la simulation nous donnent plus de contrôle au niveau du choix des paramètres, surtout du taux de service, dans le cas de l'implémentation, celui-ci nous est imposé par la configuration matérielle qui ne correspond pas forcément à la réalité. Toute fois, de manière globale, les résultats des tests confirment ceux de la modélisation avec PEPA, et par conséquent confortent les conclusions de l'étude analytique sur l'influence de l'introduction des paquets actifs sur les paquets passifs.

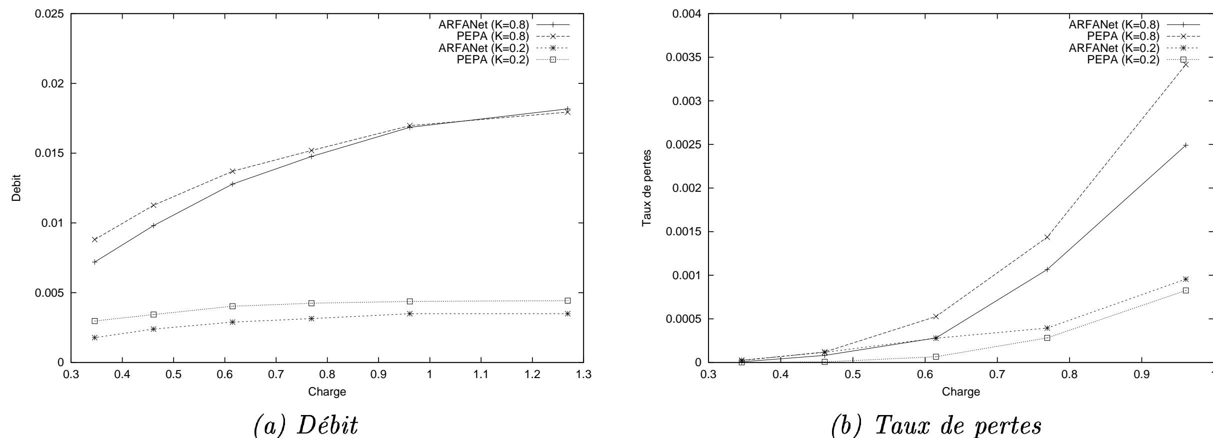
5.4.2 Le modèle de Configuration 2

La figure 5.19 montre les résultats obtenus pour le débit et le taux de pertes pour une taille des files C=10 et pour les deux proportions de paquets passifs $K = 0.2$ et $K = 0.8$.

- La figure 5.19(a) représente le débit du nœud pour les paquets standards obtenu avec le modèle PEPA, d'une part, et celui mesuré sur l'infrastructure ARFANet, d'autre part. Cette figure montre que les deux débits ont exactement le même comportement et leurs courbes sont presque confondues quelque soit la valeur de K. Dans le cas de *Configuration 1*, ceci n'est vrai qu'a moitié. Pour $K = 0.2$. les débits ont le même comportement, mais dans le cas de $K = 0.8$, leurs courbes s'intersectent.

- Comme pour *Configuration 1*, la figure 5.19(b) montre que lorsque $K = 0.2$ les pertes du modèle PEPA sont très similaires à celle de l'implémentation. Et lorsque $K = 0.8$, les deux taux de pertes sont initialement très proches, mais comme pour le premier modèle, plus la charge augmente, plus l'écart entre les deux taux de pertes augmente, le taux de pertes du modèle PEPA étant plus important. Cependant, contrairement à ce que nous avons vu pour *Configuration 1*, lorsque $K = 0.8$, l'écart entre les deux taux de pertes est nettement plus petit, puisque les pertes enregistrées dans *Configuration 2* sont plus petites que celles obtenues pour la première configuration.

Bilan :



(a) Débit *(b) Taux de pertes*

FIG. 5.19 – ARFANet *versus* PEPA

Bien que le débit dans la nouvelle configuration soit toujours inférieur au débit du nœud standard, l'introduction d'une file supplémentaire permet de réduire le taux de pertes des paquets standards de manière significative. De plus, la présence de cette file d'attente permet d'améliorer la latence dans le nœud pour les paquets passifs. Certes, cette amélioration est minime comparée à celle des pertes, mais la latence obtenue avec la seconde configuration se rapproche de celle d'un nœud standard. Cela a permis d'aboutir à une comparaison plus satisfaisante avec ARFANet.

5.5 ARFANet *versus* ANTS

Dans cette section, nous comparons un nœud ARFANet à un nœud ANTS et à un nœud standard.

Les paquets arrivant à un nœud ARFANet sont des d-paquets. L'arrivée de ce genre de paquets au nœud actif déclenche l'exécution des règles actives appropriées. Les paquets arrivant à un nœud ANTS sont des capsules qui déclenchent l'exécution du code actif. Le nœud standard est une application Java, comme les deux précédentes infrastructures, qui consiste à renvoyer les paquets. Les paquets arrivant à ce genre de nœuds sont des paquets simples et le traitement consiste seulement à les envoyer à travers le réseau selon le routage sous-jacent.

Les capsules dans ANTS et les règles actives dans ARFANet sont des programmes simples qui décrivent un simple renvoi des paquets actifs. Ces programmes simples sont choisis pour étudier les surcharges de l'exécution active comparée à un nœud standard.

Cette étude considère un environnement actif pur pour les infrastructures actives ARFANet et ANTS, ce qui signifie que tous les paquets sont actifs et exigent un traitement actif. Le taux d'arrivée des paquets actifs varie pour atteindre une valeur significative afin de voir le comportement des implémentations actives (ANTS et ARFANet). Nous sommes principalement intéressés par des mesures de performance telles que le débit, la probabilité de pertes et la latence.

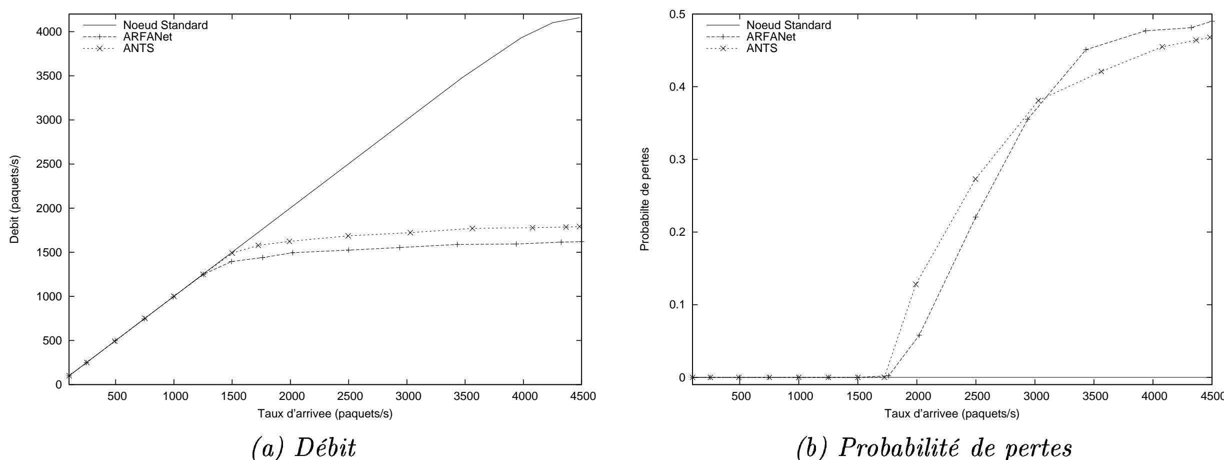
La taille du buffer est la taille du buffer UDP qui a comme valeur 64K octets. Si nous considérons 256 octets comme la longueur moyenne d'un paquet, cette taille du buffer peut être exprimée en termes de paquets en tant que 256 paquets.

5.5.1 Variation du taux d'arrivée des paquets

Dans la première expérience, nous comparons les mesures de performance d'ARFANet, d'ANTS et du nœud standard en changeant le taux d'arrivée des paquets.

La figure 5.20(a) montre que le débit d'un nœud ARFANet et ANTS sont très similaires. D'ailleurs, elle montre que le débit des deux infrastructures est très petit comparé au débit du nœud standard quand le taux d'arrivée commence à devenir important (supérieur à 1500 paquets/s). Le débit du nœud standard peut atteindre 4200 paquets/s. Les surcharges sont principalement dues à l'exécution des règles actives (ARFANet) et des capsules actives (ANTS). Elles sont également dues à la machine virtuelle de Java (JVM) car le débit de la version de C d'un nœud standard peut atteindre 15000 paquets/s.

La figure 5.20(b) montre les probabilités de pertes pour les trois implémentations. Nous notons que jusqu'à un taux d'arrivée de 4500 paquets/s, il n'y a aucune perte dans le nœud standard pour la taille du buffer considérée. Dans les deux autres implémentations, les pertes sont enregistrées seulement quand le taux d'arrivée atteint 1750 paquets/s. Alors la perte augmente exponentiellement dans les deux infrastructures actives. Notez que les résultats pour ARFANet et ANTS sont très similaires même si nous obtenons une probabilité légèrement plus élevée pour ARFANet quand le taux d'arrivée est très grand (supérieur à 3000 paquets/s). Nous observons le phénomène inverse quand le taux d'arrivée est inférieur.



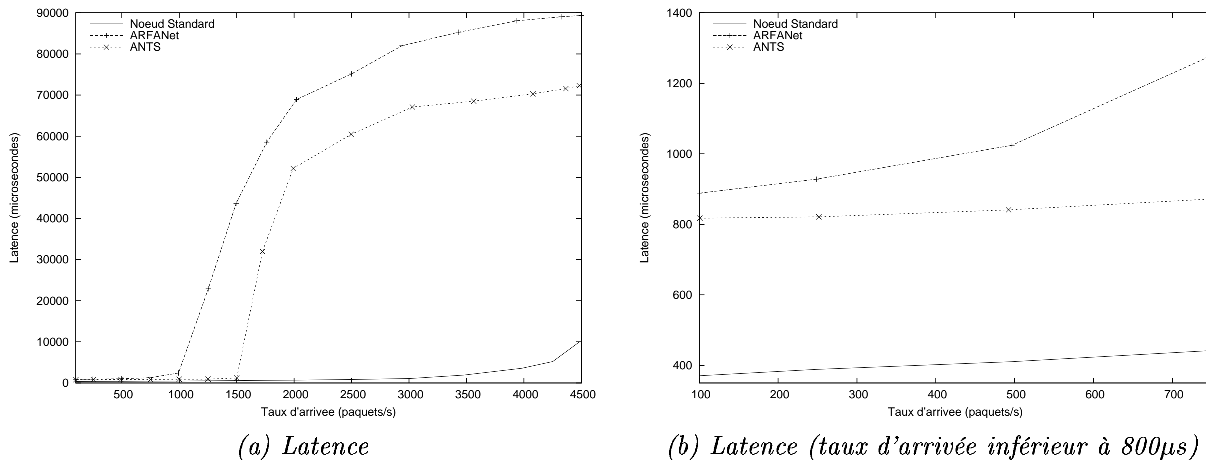
(a) Débit *(b) Probabilité de pertes*

FIG. 5.20 – ARFANet versus ANTS

La figure 5.21(a) montre la latence éprouvée par les paquets dans ARFANet, ANTS et le nœud standard. La latence dans le nœud standard semble être petite et stable parce que les paquets arrivant à ce nœud n'exigent pas un traitement additionnel comme le chargement et l'exécution d'un code actif. Ces paquets sont juste transmis. Dans le cas d'ARFANet et d'ANTS, la latence augmente de manière significative quand le taux d'arrivée approche le taux de service dans ces infrastructures. C'est-à-dire, quand le taux d'arrivée atteint 1000 paquets/s pour ARFANet et 1500 paquets/s pour ANTS.

Une meilleure latence dans ANTS est due à un taux de service plus élevé comparé à celui de notre infrastructure active. En effet le taux de service dans ANTS est 1750 paquets/s, tandis qu'il est égal à 1550 paquets/s dans ARFANet. Puisque le taux de service est plus grand dans ANTS,

le temps d'attente éprouvé par un paquet est inférieur. Pour voir le comportement de la latence quand le taux d'arrivée est petit, nous montrons dans la figure 5.21(b) la latence obtenue pour un taux entre 100 paquets/s et 800 paquets/s. Cette figure montre que pour un petit taux d'arrivée, ARFANet et ANTS ont des latences similaires et acceptables, autour de 850 μs pour ANTS et de 900 μs pour ARFANet. On note que la latence dans le nœud standard est autour de 400 μs.



(a) Latence *(b) Latence (taux d'arrivée inférieur à 800µs)*

FIG. 5.21 – ARFANet versus ANTS

5.5.2 Variation de la taille des paquets

Afin d'analyser le comportement de notre infrastructure active (ARFANet) pour différentes tailles de paquets, la deuxième partie de notre étude consiste à faire varier la taille des paquets entre 100 octets et 1000 octets. Nous considérons deux taux d'arrivée, un petit taux (100 paquets/s) et un plus significatif (1000 paquets/s).

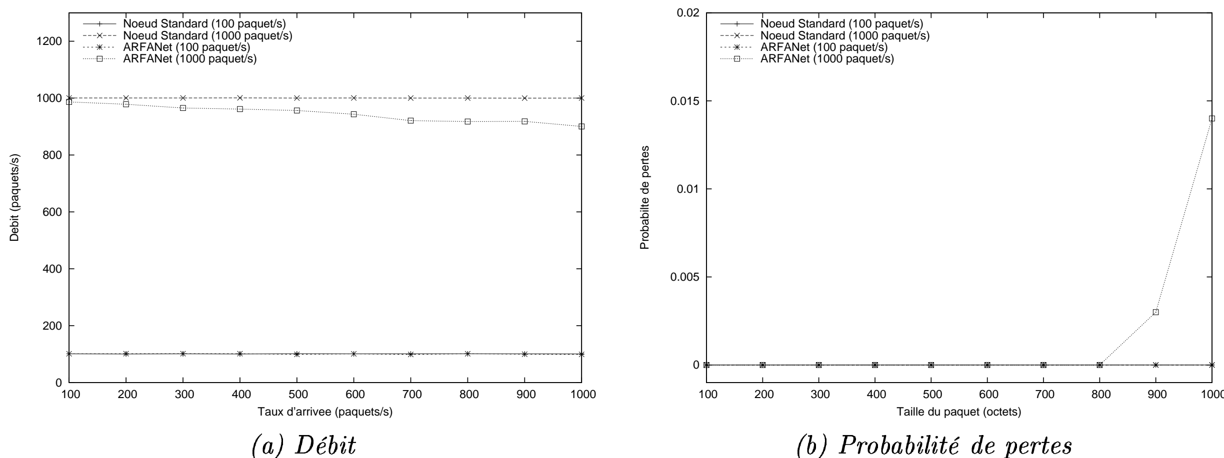
La figure 5.22(a) montre que, quand le taux d'arrivée est petit (100 paquets/s), le débit d'ARFANet et celui du nœud standard sont les mêmes pour toutes les valeurs des tailles de paquets. Quand le taux d'arrivée est grand (1000 paquet/s), le débit du nœud standard est toujours constant tandis que celui de l'infrastructure active diminue légèrement (linéairement) à mesure que la taille des paquets augmente. Cependant ces débits demeurent très similaires.

La diminution du débit dans ARFANet quand le taux d'arrivée est grand est due au fait que ce taux est près du taux de service (1550 paquets/s). L'augmentation de la taille des paquets augmente le temps de service et diminue ainsi le taux de service. La différence entre le taux d'arrivée et le taux de service devient plus petite quand la taille des paquets augmente.

La figure 5.22(b) montre le comportement de la probabilité de pertes dans ARFANet et le nœud standard quand la taille de paquets change. Nous notons qu'il n'y a aucune perte dans le nœud standard pour toutes les tailles de paquets et pour les deux valeurs du taux d'arrivée. Cette figure prouve également que lorsque le taux d'arrivée est petit, la probabilité de pertes pour ARFANet est nulle.

On observe le même phénomène quand le taux d'arrivée est grand et les paquets ont une taille inférieure à 800 octets. Cependant, à mesure que la taille des paquets augmente, la probabilité de perte augmente, mais ces probabilités restent très faibles. L'augmentation de la taille des paquets

signifie que le nombre de paquets qui peuvent attendre dans le buffer deviennent plus petits et le temps de service plus grand. Ainsi quand le taux d'arrivée est grand, à mesure que la taille des paquets augmente, la probabilité de pertes augmente de manière significative.



(a) Débit *(b) Probabilité de pertes*

FIG. 5.22 – ARFANet versus nœud standard

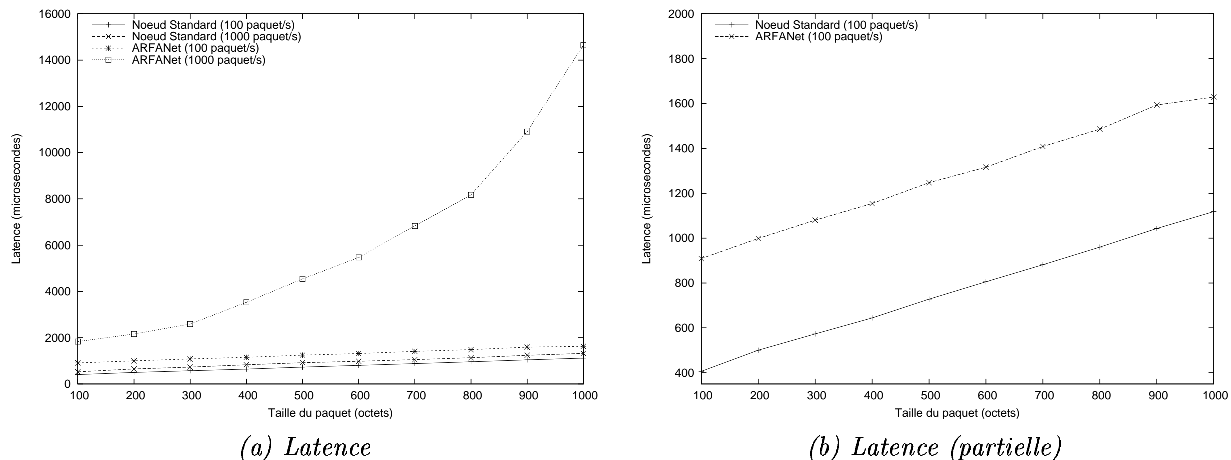
La figure 5.23(a) montre la latence mesurée sur ARFANet et le nœud standard quand la taille des paquets change. La latence dans le nœud standard augmente linéairement pour les deux taux d'arrivée et toutes les longueurs de paquets. Quand le taux d'arrivée est petit (100 paquets/s), la latence dans ARFANet est très proche de celle du nœud standard. Cependant, quand le taux d'arrivée est grand, la latence dans l'infrastructure active augmente exponentiellement à mesure que la taille des paquets augmente. Quant aux pertes, elles sont dues principalement à l'augmentation du taux d'arrivée des paquets. Cependant, l'augmentation de la taille des paquets rend le temps de service plus élevé contribuant ainsi à l'augmentation des taux de pertes. Par conséquence, la latence éprouvée par les paquets augmente à mesure que la taille des paquets augmente.

La figure 5.23(b) montre clairement le comportement de la latence dans le nœud standard et ARFANet quand le taux d'arrivée est égal à 100 paquets/s. Nous notons que les courbes sont parallèles et augmentent linéairement à mesure que la taille des paquets augmente, mais restent encore petites et acceptables.

Bilan :

Globalement les résultats que nous avons obtenus montrent que le comportement d'ARFANet et celui d'ANTS sont similaires. Même si les performances obtenues avec ANTS sont légèrement meilleures, elles demeurent très proches de celles obtenues avec ARFANet. Cette différence entre les deux infrastructures actives est due au taux de service qui est inférieur dans ARFANet, 1550 paquets/s au lieu de 1750 paquets/s pour ANTS. À la différence d'ANTS où un programme compact est chargé et exécuté, dans ARFANet, chaque règle active exige deux chargements et deux exécutions des classes Java puisque une règle active est en générale composée d'une condition et d'une action.

Cependant, la décomposition de la règle active ne pénalise pas beaucoup les performances du nœud actif. Ainsi, l'augmentation du taux de service dans ARFANet, par quelques optimisations



(a) Latence *(b) Latence (partielle)*

FIG. 5.23 – ARFANet versus Nœud standard en fonction de la taille des paquets

par exemple, aura certainement comme conséquence de meilleures de performances. Une implémentation en C de la couche Machine Virtuelle de l'infrastructure ARFANet permettra certainement d'améliorer considérablement ses performances. Cette couche étant installée une fois pour toutes ne nécessite pas une forte portabilité. D'autres optimisations peuvent être apportées à notre infrastructure pour permettre un routage efficace des paquets passifs et une exécution parallèle optimisée des moniteurs actifs et les services d'application.

5.6 Conclusion

L'objectif de ce chapitre était de modéliser la couche basse d'un nœud actif ARFANet pour pouvoir étudier le comportement de chaque composante et d'optimiser leur fonctionnement. Pour ce faire, nous avons effectué une étude des performances d'un nœud actif et comparé les résultats obtenus à ceux d'un nœud classique.

Dans un premier lieu, nous avons utilisé l'algèbre des processus stochastiques PEPA et la simulation avec SimJava pour étudier les perturbations que peut subir un trafic *normal*, qui doit coexister avec un trafic *actif*, dans un réseau adoptant la gestion active. Cette étude, qui est composée de deux parties, est réalisée au niveau d'un nœud du réseau actif où la gestion des paquets actifs utilise le principe des règles actives.

Dans la première partie de cette étude, une configuration de l'architecture du nœud a été considérée et le modèle PEPA correspondant a été utilisé pour évaluer, en termes de débit, de taux de pertes et de latence, l'impact de l'introduction de paquets actifs dans un nœud sur les paquets traditionnels le traversant. Nous avons montré que la présence des paquets actifs, exigeant un traitement plus long, a un impact négatif sur le débit du nœud pour les paquets standards.

Nous avons supposé que les paquets standards étaient perdus lorsque la file d'entrée ou celle de sortie est pleine. Nous avons observé que les paquets actifs ont moins d'impact sur les pertes de la file de sortie que sur ceux de la file d'entrée, parce que les paquets actifs sont retardés dans le nœud en attente d'une activation. Aussi, le taux de pertes des paquets standards est dominé par les pertes au niveau du buffer d'entrée.

Les résultats obtenus pour cette configuration suggèrent qu'il est possible de contrôler les pertes des paquets standards pour qu'elles restent similaires à celles enregistrées dans un nœud standard.

De plus, la latence obtenue pour le nœud actif est assez proche de celle du nœud standard. Différentes tailles de buffers ont été utilisées et des résultats similaires ont été obtenus. De manière générale, les résultats de la configuration suggèrent que le scénario de la gestion active du réseau est faisable.

Nous nous sommes intéressés également à l'impact de la génération des paquets actifs par le nœud sur le comportement des paquets passifs. Notre étude a révélé que l'impact est négligeable. Ceci s'explique par le fait que les paquets générés sont orientés vers la file de sortie du nœud. Il faut noter que le comportement des paquets passifs est très lié à la file d'entrée du nœud.

Par ailleurs, une comparaison des résultats analytiques avec ceux mesurés sur l'infrastructure ARFANet nous a permis de conclure que globalement ces résultats sont assez proches.

Dans un souci d'améliorer ces résultats, dans la seconde partie de cette étude, une seconde configuration de l'architecture du nœud actif a été proposée. Cette nouvelle configuration a permis d'améliorer le taux de pertes de près de 30% ainsi que la latence, même si l'amélioration de celle-ci n'est pas aussi significative.

En second lieu, nous avons orienté nos investigations sur la surcharge provoquée par l'exécution active des codes. Pour cette fin nous avons comparé le comportement de notre infrastructure à celui d'ANTS et à un nœud passif implémenté en Java. Nous avons également étudié l'influence de la taille des paquets sur leur latence et la probabilité de pertes dans ces plates-formes. Les résultats obtenus montrent que les performances de ces deux plates-formes sont proches. Nous avons noté les possibilités d'optimisations que nous pouvons apporter à ARFANet, notamment à la couche Machine Virtuelle. Une implémentation en C de cette couche et une amélioration du routage des paquets passifs permettent de réduire l'influence de l'activation du nœud sur les paquets standards. Au niveau des services d'application, une création d'une version simplifiée et optimisée du langage Java permet d'avoir une exécution rapide et sécurisée des actions et des conditions.

Dans le chapitre suivant nous présenterons des techniques de déploiement et de distribution des applications actives dans le contexte de l'infrastructure ARFANet.

Chapitre 6

La distribution et le déploiement du code

6.1 Introduction

Les chapitres précédents ont été consacrés à l'élaboration d'une architecture d'un nœud actif générique. Jusqu'à présent nous avons supposé que le code actif se trouve au niveau des nœuds actifs et il est prêt à être exécuté dès que les données actives atteignent les nœuds. La question à laquelle nous allons répondre dans ce chapitre est comment fournir aux nœuds le code dans le meilleur délai et de manière sécurisée.

Le déploiement de code est l'étape principale de la mise en œuvre d'un réseau actif car elle définit comment le code arrive dans les différents nœuds du réseau. Sans un bon schéma de déploiement de code, en termes de temps de mise en place (téléchargement, chargement et exécution du code) et de sécurité, la notion de réseau actif reste au stade de projet sans une vraie implémentation dans les réseaux tels que Internet. Comme le déploiement de code, l'identification n'a jusqu'à présent fait l'objet d'aucune normalisation ou standardisation dans les réseaux actifs contrairement à l'architecture d'un nœud ou du format de paquet. Dans la partie état de l'art, nous avons constaté le manque de standardisation dans ce domaine, en particuliers pour les AA et ainsi donc chaque projet définit sa propre méthode d'identification.

De plus, à notre connaissance, aucune des méthodes d'identification et de déploiement de code ne permet un partage pratique, une réutilisation et une composition efficace des services par d'autres utilisateurs.

Dans ce chapitre nous définissons une architecture globale pour le déploiement et l'identification de code dans les réseaux actifs. Notre objectif est de fournir un mécanisme qui garantit une identification unique du code et permet son utilisation par des utilisateurs autres que son propriétaire. Cette approche est basée sur l'utilisation du serveur de code [22]. En plus de son rôle de base de stockage de code, ce serveur peut accomplir d'autres tâches telles que l'attribution à chaque application d'un identificateur unique et la vérification de la conformité du code et de l'authenticité de son développeur.

Notre approche tire beaucoup avantage des caractéristiques de l'infrastructure ARFANet dans laquelle les règles actives introduisent une certaine flexibilité et modularité. Chaque module peut être défini et développé par une entité différente. De plus, il peut aussi être réutilisé dans la composition d'autres applications. Ces caractéristiques permettent de dissocier le développement, l'identification et le déploiement d'un module vis-à-vis des autres.

Ce chapitre est organisé comme suit. Dans la section 6.2, nous discutons du problème de dis-

tribution de code dans les plates-formes actives actuelles et les solutions actuellement considérées. L'approche que nous proposons est présentée dans la section 6.3. La section 6.4 est consacrée à l'évaluation des performances de notre approche en termes de débit et de latence. Les résultats obtenus sont comparés à ceux de la technique *hop by hop* développée dans le cadre de la plate-forme ANTS [94]. Nous résumons les caractéristiques de notre approche dans la section 6.5. Nos conclusions pour ce chapitre sont présentées dans la section 6.6.

6.2 Distribution du code actif

La distribution de code consiste à injecter du code dans un système semi-fermé que constituent les routeurs. Cette opération doit prendre en compte toutes les précautions de sécurité et de sûreté pour qu'un code malveillant n'altère ni le fonctionnement de base des routeurs (le routage des paquets), ni leurs performances. Elle doit aussi fournir un mécanisme de référencement homogène aux programmes actifs à l'intérieur et à l'extérieur de ce système.

En dépit de l'importance du processus de la distribution du code, les deux principales techniques, intégrée et discrète, dominent et orientent les projets dans le domaine des réseaux actifs. Il faut noter le peu de techniques proposées et de travaux élaborés dans le domaine.

On distingue deux étapes indépendantes dans la distribution d'un code. La première étape est le déploiement effectif dans les nœuds actifs de la nouvelle application ou le service actif. La deuxième est l'identification du code.

6.2.1 Déploiement du code

La politique choisie pour le déploiement du code dans le réseau définit la manière avec laquelle le code atteint les nœuds actifs de ce réseau. Les projets de recherche dans le domaine considèrent deux approches de déploiement de code, les capsules et la pré-programmation des commutateurs.

1. *Les capsules :* dans cette approche, le code est transmis dans le même flux que les données ; une capsule (paquet) peut contenir le code actif et les données [93]. Quand une capsule contenant des données arrive au nœud, celui-ci charge immédiatement la partie code du paquet et l'exécute dans l'environnement d'exécution correspondant. Dans cette approche, intégrer le code actif dans le même paquet que les données augmente considérablement sa taille [12]. Or comme le temps de réponse (temps de traitement et de propagation) est proportionnel à la taille des paquets, les protocoles réseau sont inefficaces pour le traitement des paquets de grande taille.

   Pour pallier à cet inconvénient, cette approche a été modifiée pour intégrer le mode *hop by hop* (nœud par nœud) [94]. Dans cette approche, le code et les données peuvent être envoyés séparément. Un nœud recevant une capsule extrait la référence du programme actif et vérifie sa présence au niveau de sa base de code. Si le programme est présent dans le nœud, il sera chargé en mémoire et exécuté sur les données. Dans le cas où le code est absent, le nœud envoie une demande au nœud précédemment visité par le paquet (un champ adresse du nœud précédent est mis à jour à chaque nœud). Ce processus sera répété nœud par nœud tout au long du chemin emprunté pas les paquets.

   Par ailleurs, seuls les nœuds traversés par le flux des capsules sont programmés. Ainsi la configuration des nœuds à programmer dépend du chemin emprunté par les paquets qui n'est pas toujours le même. Cette technique est purement active car elle ne peut pas appliquer un service actif sur les paquets standards.

2. *Les commutateurs pré-programmés :* dans cette approche, le code est envoyé avant et de manière séparée des données [85]. Le déploiement de code est effectué en deux phases distinctes. La première phase consiste à envoyer le code vers les nœuds choisis à travers tous les chemins possibles que les paquets de données peuvent traverser, généralement un seul chemin potentiel pour chaque session d'utilisateur. La deuxième phase est l'envoi des paquets de données. Dans ce cas, les nœuds sont pré-programmés et prêts à recevoir les données actives. L'inconvénient de cette approche est lié à l'une des spécificités du routage dans les réseaux IP actuels. Ce type de routage est orienté paquets et plusieurs chemins peuvent être pris par les paquets de données de l'utilisateur. Ainsi, il est difficile de déterminer à l'avance les nœuds à pré-programmer. Son application dans les protocoles à commutation de circuits (comme ATM) serait opportune car le chemin est prédéfini au départ. La particularité de cette technique est qu'elle était destinée à appliquer le service actif sur les paquets standards sans que ces derniers le mettent en référence comme dans la première technique.

En dépit de leurs différences fondamentales, ces deux approches partagent les mêmes particularités :

- Un code actif ne peut pas être stocké indéfiniment dans les nœuds. A un certain moment, il doit être supprimé de la mémoire du nœud. Le temps de sauvegarde d'un code peut être préalablement défini pour tous les nœuds du réseau ou décidé par chaque nœud selon l'état de son espace mémoire.
- Un code actif déployé dans le réseau ne peut être partagé par les autres utilisateurs. Ceux-ci doivent demander le code à son propriétaire ou en développer une autre version.
- Un nœud actif ne peut pas vérifier l'intégrité du code reçu et l'authenticité de son propriétaire. Les programmes de vérification consomment beaucoup de temps et d'espace mémoire et leur exécution pénalise les performances du nœud.
- La publication des environnements d'exécution est statique. Chaque environnement a sa référence prédéfinie et figée dans ANEP (Active Node Encapsulation Packet) et son déploiement est effectué statiquement au départ. Quant à sa référence, elle doit être connue par le NodeOS et tous les utilisateurs avant l'envoi des applications actives.
- Il est difficile de créer de nouvelles applications actives à partir de la composition de petits programmes actifs.
- Dans l'approche commutateurs pré-programmés, il faut considérer le temps requis par chacune des deux phases de la pré-programmation des nœuds et de l'exécution du code sur tous les nœuds du chemin, ou autrement dit, la session (réception des données actives de l'utilisateur et exécution du code correspondant). Le temps de session peut être petit, mais le temps global peut être considérable. De même, dans l'approche capsules (ou *hop by hop*), le temps du téléchargement du code est inclus dans le temps de la session. Ce temps qui correspond au temps global peut être également important.

6.2.2 Identification du code

Actuellement beaucoup de projets tels que ANTS [92] [93] et Switchware [6] utilisent l'architecture standard à trois couches développée dans [13]. Dans cette architecture, l'identification des données de l'application de l'utilisateur est propre à chaque session utilisateur. Un paquet de données doit contenir l'identifiant de l'AA ainsi que celui de l'environnement d'exécution (EE) correspondant. Ainsi à l'arrivée d'un paquet de données à un nœud actif, celui-ci charge l'EE dont l'identifiant

est spécifié dans le paquet. Le EE doit alors déclencher à son tour le chargement de l'application active correspondante.

La référence du EE est donnée une fois pour toutes par l'autorité de normalisation ANANA (Active Networks Assigned Number Authority) et ne peut être changée. Le paquet d'encapsulation ANEP, qui est une norme en termes de format de paquets, comprend un champ de 16 bits dédié à l'identifiant du EE. Ce champ est utilisé par le système d'exploitation du nœud (NodeOS) à l'arrivée d'un paquet de données pour identifier le EE correspondant et le charger en mémoire. La référence d'un EE est unique parce que chaque EE est défini par une seule entité et placé une seule fois au niveau du nœud.

Une des caractéristiques de base d'un EE est qu'il doit être capable de gérer plusieurs applications actives à la fois, de les charger en mémoire et de les décharger selon les besoins de l'utilisateur. Ceci implique, comme les EE, que chaque application doit avoir un identifiant unique. Cependant comme l'identifiant d'une AA est, généralement, défini par son développeur et que chaque application peut être développée par des utilisateurs indépendants, le problème des identificateurs multiples risque d'être inévitable. En effet, étant donnée la multitude des sources de code et leur indépendance, et l'architecture multi-niveaux du nœud, des situations où deux applications actives partagent le même identifiant ne sont pas impossibles.

Par ailleurs, comme l'identifiant d'une application est propre à son développeur, le code de celle-ci reste difficilement partageable et réutilisable par d'autres utilisateurs.

Dans les systèmes qui se basent sur un transfert du code dans le même flux de données, l'utilisation d'une *clef unique* [93] détenue par un seul développeur pour créer un identifiant unique est très répandue. Cette clef est fournie par l'administrateur du réseau, le fournisseur Internet ou une autorité de confiance. En utilisant des mécanismes de cryptage des PKI (infrastructure à clef publique), le développeur crypte avec sa clef la référence de son application. L'unicité de la clef du développeur assure par transitivité l'unicité de l'identificateur du code. Cette approche peut s'avérer utile car elle authentifie et limite le nombre des développeurs autorisés à déployer leurs codes dans le réseau. Toutefois, cette solution nécessite une autorité de confiance qui attribut les clefs et les certifie et rend l'identifiant de l'application fortement lié à son propriétaire.

Dans ce qui suit, nous présentons une nouvelle approche de déploiement et d'identification de code dans les réseaux actifs. Cette approche constitue une réponse aux problèmes, entre autres, d'identification et de partage du code. Bien que cette approche ait été développée dans le cadre de l'infrastructure ARFANet, de laquelle elle tire beaucoup avantages, elle reste parfaitement applicable dans d'autres contextes.

6.3 Identification et déploiement de code dans ARFANet

L'idée de base de notre approche consiste à utiliser une ou plusieurs autorités dont le rôle est d'associer un identifiant unique à une application. Ces autorités auront le rôle de sources permanentes des services pour les nœuds.

Dans les sous sections suivantes nous détaillons le fonctionnement de chaque autorité de manière isolée et son comportement en groupe.

6.3.1 L'approche CISS

Le *serveur d'identification et de stockage de code (CISS - Code Identification and Storage Server)* est une extension du serveur de code présenté dans [22]. En plus de la fonction de stockage, le CISS attribue automatiquement à chaque programme un identifiant unique.

Nous associons au CISS un *site web de publication* qui reflète l'image, sous forme de pages HTML, du contenu de la base de codes du CISS. Chaque page contient un descriptif détaillé sur le service d'application, son identifiant unique et le rôle de chacune de ses règles. A partir de ce site web chaque utilisateur peut prendre connaissance de l'ensemble des services d'application existants. Ce site permet à ses visiteurs de composer leurs applications en choisissant les différents modules. De même, les développeurs peuvent considérer la base des codes sur le CISS comme une bibliothèque de modules qu'ils peuvent référencer dans leurs propres applications. Ce site comporte des règles d'accès et de consultations sécurisées que nous aborderons dans le Chapitre 7.

Dans notre technique la distribution du code est composée de trois étapes, la publication, le référencement et la récupération du code.

1. *Publication du code :* cette étape est réalisée de manière indépendante par le *développeur* de l'application. Le développeur qui veut publier ses applications doit envoyer ses paquets de code (m-paquets et s-paquets) au CISS, ce qui correspond au message '1' de la figure 6.1. La réponse du *CISS*, après vérification de la sûreté du code et l'authentification de son fournisseur, est une référence unique au code actif ou un message négatif, englobé par le message '2'. Un message négatif survient si la tâche de la publication échoue. Cet echec peut être dû à une erreur de sauvegarde du code, de l'attribution de la référence ou si un code malveillant est détecté. A la fin de cette étape un identifiant unique, appelé ACI (Active Code Identifier), est attribué au code qui est sauvegardé dans la base. Le CISS publie, en envoyant le message '3', sur le site web une page HTML contenant l'identifiant du service, le nom de son fournisseur, l'ensemble de ses modules et règles et un descriptif détaillé de chaque règle.

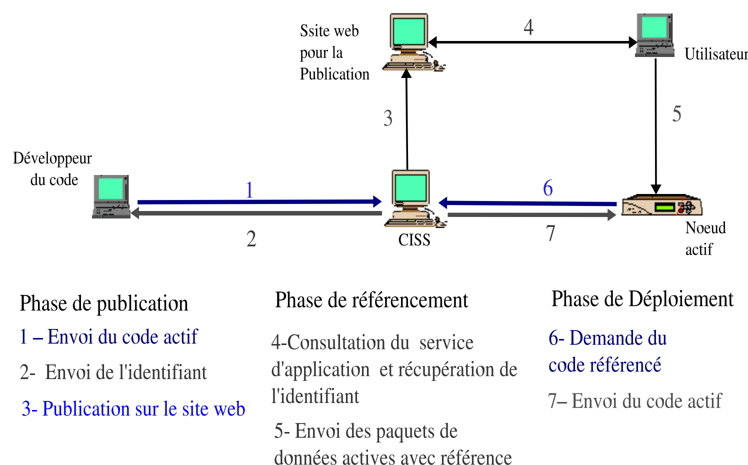


FIG. 6.1 – Publication et récupération du code actif

2. *Référencement du code :* quand l'*utilisateur* veut personnaliser le traitement des nœuds du

réseau sur le contenu de ses paquets, il se connecte au site de publication afin de choisir le service et de récupérer son identifiant (ACI). Cette phase est numérotée '4' dans FIG 6.1. Une fois l'identifiant récupéré, l'utilisateur envoie à travers le réseau ses paquets de données (d-paquets) avec le ACI (message '5').

3. *Récupération du code :* à l'arrivée d'un paquet de données à un nœud, si celui-ci ne possède pas le code référencé, il doit envoyer une demande de code au *CISS* (message '6'). Le CISS satisfait les requêtes des nœuds en leur envoyant le code demandé (message '7' dans FIG 6.1).

Avec cette approche, chaque application active est caractérisée par un identifiant unique. De plus, le partage et la réutilisation des codes deviennent possibles parce que toutes les applications (codes actifs) sont sauvegardées dans la base du CISS. Ainsi tout utilisateur qui veut utiliser un code existant aura juste à le référencer dans ses paquets de données.

Par ailleurs, cette approche constitue une solution au problème des chemins multiples empruntés par les paquets d'une même application, ce qui peut être le cas dans les réseaux IP. Les nœuds se trouvant sur le chemin de données traversé par les paquets actifs peuvent, au fur et à mesure, être pré-programmés en sollicitant le CISS. Cette approche permet aussi au nœud de ne pas garder les codes indéfiniment, mais sans pour autant les perdre. En effet, les codes seront toujours récupérables au niveau du CISS.

La présence d'un CISS évite au nœud tout mécanisme de vérification lourde des codes. Ce genre de tâches est désormais laissé au CISS. Nous détaillons ce point dans le prochain chapitre.

Il est important de noter que les paquets standards peuvent subir un traitement actif même s'ils ne référencent pas un service d'application. Comme notre approche est à base d'événements, l'arrivée d'un paquet, même d'un paquet standard, constitue un événement capable de déclencher une règle active. Ceci sous-entend l'installation au préalable d'un écouteur d'événements qui reconnaît le type de paquets standards ou une adresse source et/ou une destination (selon les adresses dans l'en-tête). Ce mode de référencement des applications actives est utilisé dans le cadre de l'approche discrète.

6.3.2 L'approche multi-CISS

La présence d'un seul serveur CISS peut concentrer tout le trafic du code actif et/ou des requêtes sur un nombre réduit de liens. Un goulot d'étranglement peut alors se former au niveau du CISS entraînant la congestion du réseau. Une solution pour éviter la congestion du réseau consiste à ajouter un ou plusieurs CISS selon la topologie du réseau.

La détection de tous les CISS présents dans un voisinage est l'une des premières tâches qu'un nœud *passif* ou standard doit accomplir au moment où il devient actif. Chaque nœud doit alors ordonner les CISS dans son voisinage dans une liste selon le temps RTT moyen par exemple.

Quand un nœud actif reçoit un d-paquet référençant un code actif absent du nœud, celui-ci envoie une requête de code au premier CISS dans sa liste. Si le premier CISS répond avec un message négatif ou aucun message n'a été reçu après le temps RTT associé à ce serveur, le nœud envoie une nouvelle demande au second CISS dans sa liste. Ce processus est répété jusqu'à ce que le nœud reçoive les modules actifs appropriés.

Pour éviter qu'un nœud reçoive plusieurs versions du même code de différents CISS, il peut également entamer un protocole de négociation en envoyant d'abord une *notification d'existence de code* à l'ensemble des CISS dans son voisinage. L'ensemble des CISS peut constituer un groupe

de multicast. Si un CISS est en mesure de satisfaire la requête du nœud, il répond par un équivalent ACK, sinon il envoie un équivalent NACK. Lorsque le nœud reçoit un ou plusieurs ACK, il sélectionne un seul CISS et lui envoie une *requête de téléchargement de code*. On suppose que les messages du protocole de négociation constituent des paquets de petite taille et leur transmission dans le réseau ne nécessite pas plus de temps que les messages *syn*.

Cette version de l'approche CISS soulève plusieurs points tels que le nombre de CISS, leur localisation dans le réseau et les politiques de gestion des bases de codes que nous devons étudier. Dans ce qui suit, nous présentons les solutions qu'on préconise pour répondre à ces points.

6.3.2.1 Choix du nombre et répartition des CISS

La répartition des CISS dans le réseau peut tenir compte de l'architecture imbriquée du réseau Internet. A un sous-réseau ou *domaine* contenant un certain nombre de nœuds actifs peut être affecté un CISS (FIG 6.2). La présence d'un CISS dans le même domaine de routage que les clients diminue les temps de téléchargement des codes et par conséquent le temps de session. Ceci est d'autant plus justifié par des études sur l'évolution du réseau Internet pour tout type de flux et d'applications [75][68], et en particulier pour le commerce électronique [58][89]. Ces études ont montré que la grande partie du trafic dans le réseau Internet actuel, qui est plus polarisée, inter-domaine et internationale, tend à se localiser et à devenir plus intra-domaine, régionale et nationale. Ainsi les modules de code dans la base d'un CISS ont plus de chance d'être demandés et utilisés par les utilisateurs qui appartiendraient au même domaine que ce CISS.

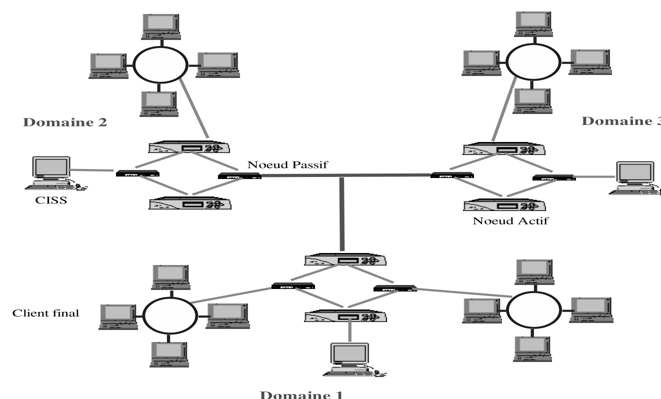


FIG. 6.2 – Répartition des *CISS*

Les CISS ne peuvent être placés au cœur du réseau (*Core Network*) tout comme les nœuds actifs. Ce genre de réseau est caractérisé par de hauts débits et ses routeurs sont moins susceptibles de faire un traitement actif sur les paquets les traversant. De ce fait, dans notre approche les CISS sont plutôt placés dans la périphérie des réseaux. Ils auront des positions intermédiaires entre les utilisateurs finaux et les nœuds actifs comme les pares-feu, par exemple. Nous pensons qu'il est plus judicieux de spécialiser les routeurs selon leur localisation. Les routeurs au contact des utilisateurs finaux doivent être plus spécialisés dans les applications orientées utilisateurs. Alors que les routeurs prochent du réseau cœur doivent être plus spécialisés dans les applications orientées réseau.

6.3.2.2 Gestion de la base de code des CISS

Deux politiques de gestion de la base de codes au niveau des CISS peuvent être considérées. La première politique se base sur le mode de répartition des CISS par domaine. Dans cette politique, chaque CISS gère une partie de la base de codes, celle contenant les modules et les programmes actifs les plus référencés par les nœuds actifs voisins (Bases de codes réparties). Le contenu de cette partie peut changer selon l'évolution de la demande de code et du réseau. La mise en œuvre et la gestion de cette solution semblent tout à fait réalisables puisque le propriétaire du code actif et ses utilisateurs appartiendraient au même domaine et partageraient le même CISS.

La seconde politique est basée sur la duplication de la base de code au niveau de chaque CISS (Bases de codes répliquées). Ceci suggère que les CISS doivent communiquer entre eux pour la mise à jour de leur base de codes respective. Cette solution est complexe par sa mise en pratique et sa gestion. De plus, elle peut être à l'origine d'une augmentation sensible du trafic dans le réseau.

1- Les bases de codes réparties :

En s'inspirant des bases de données réparties nous pouvons considérer chaque CISS comme une base de données séparée où nous pouvons appliquer les techniques de la distribution étudiées dans les domaines des bases de données réparties (BDR).

Les caractéristiques des BDR qui semblent adéquates pour les CISS répartis sont :

- du point de vue de l'utilisation, un système distribué doit paraître fonctionné exactement comme un système local.
- l'autonomie locale : le CISS fonctionne en local de manière homogène et complètement indépendante des autres CISS,
- une gestion distribuée (pas de site central) : un CISS en dysfonctionnement ne doit pas altérer le bon fonctionnement des autres CISS (une indépendance vis-à-vis de la gestion et du contrôle),
- un fonctionnement permanent : une meilleure tolérance aux pannes.
- une localisation transparente : l'accès des nœuds aux codes doit être uniforme quel que soit l'emplacement du site de stockage.
- une indépendance à la réplication : les codes répliqués doivent être maintenus en cohérence (la version, le fournisseur, ...),
- une exécution de requêtes distribuées : l'existence de modules sur plusieurs sites implique l'exécution d'une seule requête, mais qui doit prendre en compte la répartition de ces modules,
- indépendance vis-à-vis de matériel : le type du matériel du serveur qui abrite le CISS (le type, le nombre et la disposition des processeurs, ...) ne doit pas être restreint à un ensemble prédéfini.
- une indépendance vis-à-vis du système d'exploitation : il est tout a fait possible que les CISS soient implémentés avec différents systèmes d'exploitation.
- une indépendance vis-à-vis du réseau et des protocoles.

D'après les caractéristiques propres aux CISS, nous ajoutons d'autres fonctions telles que :

- l'identification d'un nouveau service d'application doit être unique sur l'ensemble des CISS (éviter que deux CISS donnent le même identifiant à différents codes),

- les modules utilisés dans la composition de services peuvent être placés dans différents CISS. Il est possible d'avoir des requêtes réparties.
- le CISS recevant la requête (répartie) de code avec des modules répartis, peut jouer le rôle du CISS central pour la requête. Il doit, de ce fait :
  + exécuter la (les) sous-requête(s),
  + récupérer les modules résultats,
  + assembler en local un ou plusieurs s-paquets (selon la taille du s-paquet résultant et la taille maximale tolérée par le protocole réseau sous-jaçant),
  + envoyer le(s) s-paquet(s) au nœud,
  + garder le s-paquet temporairement dans sa base dans le cas où d'autres nœuds de la même session le demande. Ceci permet d'éviter de répéter l'exécution de la même requête et la récupération des différents modules (pour optimiser le trafic et le traitement par les CISS).

Après la présentation des caractéristiques et des fonctionnalités d'un CISS réparti, nous pouvons définir les CISS répartis en tant que bases de codes réparties sur plusieurs serveurs distants, mais visible comme un CISS unique. L'utilisation d'un schéma local et un schéma global est nécessaire pour maintenir l'ensemble de la base dans un état cohérent.

Chaque CISS doit contenir une *table de références globales* ou *TRG* contenant dans chaque entrée la référence des services d'application et leur emplacement dans le réseau, autrement dit l'adresse du CISS qui les stocke (schéma global). Il doit contenir également une *table de références locales* ou *TRL* contenant les références des services d'application locaux au CISS (schéma local).

L'identification des codes dans le contexte de plusieurs CISS est plus compliquée, comparée à celle dans le cas d'un seul CISS. La règle de l'unicité des identifiants peut ne pas être satisfaite si deux CISS attribuent à deux codes distincts la même référence.

Il existe plusieurs manières de garantir l'unicité de l'identifiant dans un contexte distribué :

- **CISS central** : Une des solutions est de centraliser l'identification et tout contrôle de tout genre. Dans ce cas, un CISS central joue le rôle de coordinateur. A l'arrivée d'un nouveau service d'application à un CISS, celui-ci contacte le CISS central qui attribut l'identifiant et le lui envoie. A son tour le CISS demandeur renvoie l'identifiant au développeur. Le CISS central signale à tous les CISS de mettre à jour leurs tables de références globales.

- **Coopération entre CISS** : Cette technique stipule qu'il n'existe pas de CISS central. Les CISS doivent coopérer pour maintenir un identifiant global. Ceci nécessite beaucoup de communication entre les serveurs. Quand un nouveau code arrive à un CISS, celui-ci lui attribut un identifiant temporaire, ensuite déclenche une procédure de validation à deux phases. La première phase consiste à informer tous les CISS du nouvel identifiant. Dans le cas où tous les CISS acceptent ce nouvel identifiant, ils envoient un message de notification au CISS concerné pour le retenir. En retour, le CISS leur envoie, dans une deuxième phase, la validation de cet identifiant. Les autres CISS, doivent, de leur côté, mettre à jour leurs tables de références globales. Cependant, lorsqu'au moins un des CISS refuse de valider l'identifiant parce qu'il est aussi en cours d'attribution du même identifiant à un autre code, le CISS doit générer un autre identifiant et refaire le processus de validation jusqu'à résoudre tout conflit.

- **Concaténation du nom du CISS à l'identifiant** : C'est la technique la plus simple car

elle ne nécessite ni coopération, ni gestion centralisée. Son principe est très simple, le CISS recevant un nouveau code lui attribut un identifiant unique concaténé à l'adresse du CISS lui même (l'adresse IP par exemple, ou son nom d'hôte : *ciss1.prism.uvsq.fr*). Une fois cette étape accomplie, le CISS envoie à tous les CISS la référence du nouveau code pour qu'ils mettent à jour leurs tables de références globales. Malgré sa simplicité, son autonomie et le peu de trafic qu'elle engendre, cette solution reste rigide et sujette à une évolutivité très limitée. Ceci est dû à la dépendance du l'identifiant au nom du CISS qui peut changer au cours du temps.

2- Les bases de code dupliquées (répliques distribuées) :

La présence de serveurs répliqués [38] [2] aident à améliorer la latence (requêtes très rapides) car le nœud peut contacter le CISS le plus proche. Elle permet aussi de diminuer la charge du réseau et son trafic en diminuant le nombre de liens et de routeurs qui vont prendre en charge les messages. Mais cette technique comporte quelques défauts tels que la redondance et les mises à jour nombreuses qui peuvent être très lentes.

Il existe deux types d'algorithme de mise à jour des répliques. Le premier concerne les mises à jour à forte consistance et le second est dédié aux mises à jour à faible consistance.

L'algorithme de mise à jour des répliques à forte consistance garantit une cohérence au niveau des répliques de tous les sites à tout moment. Néanmoins, il présente quelques inconvénients puisqu'il est très coûteux, non évolutif, peu fiable et génère une grande latence et plus de trafic. Cet algorithme reste adapté aux systèmes synchrones avec peu de répliques.

Quant à l'algorithme de mise à jour des répliques à faible consistance, il vise à diminuer le trafic entre les différents sites en évitant de concentrer tout l'effort à maintenir une forte consistance. Le but de ce genre d'algorithme est d'assurer que les répliques convergent vers un état cohérent dans un temps fini. Dans un système à faible consistance chaque serveur choisit, de temps à autre, un voisin aléatoirement et lance une session de mise à jour. A la fin de chaque session, le contenu des deux bases est identique et consistant. Cette façon de faire est appelée *anti-entropie*. Il a été démontré que le choix aléatoire des serveurs voisins pour lancer des sessions donne les meilleurs performances pour maintenir la consistance des répliques car il génère peu de sessions [38]. Cet algorithme est très opportun pour les systèmes asynchrones. Parmi ses avantages, nous citons la faible génération de trafic, la faible latence et sa forte évolutivité.

Les CISS sont plutôt des systèmes asynchrones et leur coopération dans le cadre d'un réseau actif ne nécessite pas une forte consistance dans leur contenu. De ce fait, nous pouvons conclure que dans un système où les CISS sont répliqués, l'algorithme de mise à jour à faible consistance est le mieux adapté.

3- Un système de gestion de la base de codes pour ARFANet :

Le choix entre la méthode qui duplique la base de codes sur plusieurs sites et celle qui la répartit semble très difficile du fait leurs avantages et inconvénients. Par conséquence, nous avons choisi de nous baser sur les caractéristiques des réseaux actifs et de la distribution du code pour déterminer laquelle des deux approches correspond le mieux. Vue la spécialisation du contenu du CISS selon son emplacement et son domaine réseau, nous supposons que les utilisateurs vont plus référencer des codes qui existent dans le CISS le plus proche (voir la sous-section 6.3.2.1). La duplication dans ce cas n'est pas adaptée car le taux d'utilisation d'un code ne sera pas homogène sur les différents sites.

Dans la pratique, l'entité réalisant le code a de forte chance d'être un administrateur de réseau ou un fournisseur d'accès Internet. Dans le but d'améliorer un service déjà existant, ces deux types d'entité devront développer un code actif qui sera destiné à leurs clients qui appartiennent au même domaine. Ainsi, il nous semble plus approprié de considérer la répartition de la base que sa duplication pour sa simplicité de mise en œuvre et le gain en espace et en échanges de messages lors de la maintenance de la cohérence.

Quant à l'identification, nous adoptons la technique de concaténation pour la simplicité de sa mise en œuvre. D'autre part, il n'est pas judicieux de centraliser l'identification ou tout contrôle, car chaque CISS doit être capable de fournir les mêmes services. En ce qui concerne la coopération, elle génère beaucoup de trafic et nécessite plusieurs phases.

En cas de panne, la tâche du CISS après la reprise totale est de mettre à jour sa *table de références globales* à partir de celles des autres CISS. Le CISS doit importer au moins une *table de références* des CISS voisins pour la comparer avec la sienne. La taille des *tables de références* peut poser problème si elle est volumineuse. Dans ce cas le CISS peut seulement demander les entrées de la table à partir d'une date valide (après un *commit*).

Dans le cas où un nœud demande au CISS le plus proche un code qui n'est pas dans sa base, ce CISS peut aider le nœud, en utilisant sa table de références globales, à accéder directement et rapidement au CISS disposant du code en lui donner l'adresse du CISS.

Le prochain paragraphe présente une autre approche que nous considérons comme une solution mixte entre la méthode employant le CISS et la méthode du *Hop by Hop*.

6.3.3 L'approche mixte

Comme nous l'avons déjà souligné auparavant, il est tout à fait probable que les CISS ne soient pas placés dans le noyau du réseau (ou le cœur), mais plutôt dans les abords du réseau global et dans les sous-domaines. Pour les nœuds près des serveurs CISS (appartenant au même domaine), le nombre de routeurs les séparant est très petit et le temps RTT est réduit. Par ailleurs, si chemin d'un nœud quelconque dans le réseau au CISS est plus long, la demande de code du nœud le plus proche convient le plus dans ce cas.

L'approche CISS s'applique et se combine facilement avec d'autres techniques, en particulier avec la technique *hop by hop* [93]. Ceci rend notre approche plus flexible et autonome (réduire son attachement à un serveur). Dans cette combinaison, le CISS joue son rôle classique d'attribution des identificateurs et de sauvegarde des règles et des moniteurs actifs. Il a également le rôle de la première et permanente source de code. Dans ce cas, nous pouvons considérer un ou plusieurs CISS. Le seul changement réside dans le comportement des nœuds qui peuvent choisir entre plusieurs sources de codes. La migration nœud par nœud du code intervient pour réduire la charge du *CISS* et pour améliorer également le temps de transfert du code en choisissant la source la plus proche.

De plus, la combinaison peut être une alternative dans le cas où un nœud actif ne peut pas recevoir le module actif ou reçoit une réponse négative du CISS. Ceci peut être dû à l'une des raisons suivantes : le CISS est congestionné, le lien est provisoirement rompu, un goulot d'étranglement a été formé, le serveur est éteint ou n'a pas le bon code (cas de plusieurs CISS contrôlant chacun une sous-partie de la base de code distribuée).

Le nœud actif peut demander le code au nœud actif précédent dans le chemin du paquet de données. L'adresse du nœud actif précédent est donnée par un champ dans l'entête du paquet,

appelé *nœud précédent*.

L'adoption de cette approche exige de nous de suivre deux étapes et de faire la distinction entre le premier nœud actif visité et le reste des nœuds.

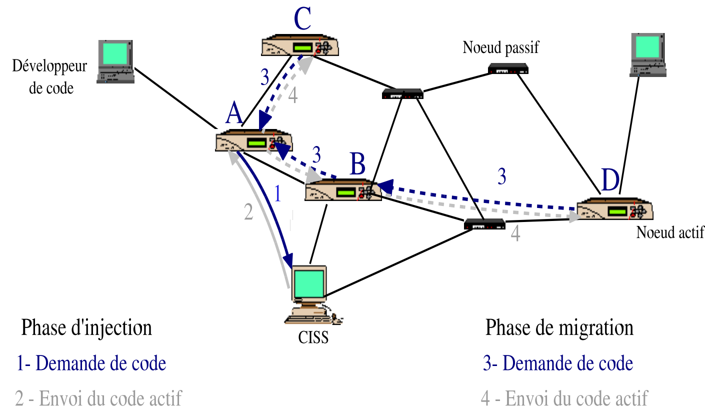


Fig. 6.3 – Les phases de la méthode mixte

1. *Injection du code* : au départ de chaque session, l'utilisateur envoie ses d-paquets avec les références ACI au nœud actif qui lui est attaché. À ce moment là le champ "nœud précédent" du paquet de données est vide. Ce nœud est le premier visité et doit télécharger le code du CISS si le code actif est absent. Cette phase est représentée par les étapes '1' et '2' de la figure 6.3. Avant le renvoi du d-paquet, ce premier nœud actif met son adresse dans le champ "nœud précédent" et par conséquent il devient le premier "nœud précédent" de la chaîne des nœuds actifs à visiter.

2. *Migration du code* : à partir du deuxième nœud actif visité, les nœuds recevant le d-paquet extraient le champ "nœud précédent" qui détermine la nouvelle source de code. A ce moment, toutes les demandes de code exprimées par chaque nœud devront être envoyées au nœud qui le précède dans le chemin des données. Les nœuds ne demandent le code au CISS qu'en cas d'erreur. Le champ "nœud précédent" devrait être mis à jour à chaque nœud. Cette phase est représentée par les étapes '3' et '4' de la figure 6.3.

Comme nous pouvons le constater, avec la technique *mixte*, le CISS est sollicité seulement une fois pour toute la session de données. Elle peut être employée avec un ou plusieurs CISS. Cette technique permet, entre autre, de rendre le processus de distribution plus flexible et moins dépendant des CISS. Néanmoins, elle exige des nœuds de faire partie des acteurs actifs dans ce processus, ce qui peut dégrader les performances des nœuds.

Gelas, dans [33], a cité succinctement la possibilité de combiner le déploiement de code à partir d'un serveur de code et la méthode *hop by hop* pour une raison de sécurité. L'application active ne doit pas être téléchargée du poste du client. Ainsi le premier nœud doit récupérer l'application du serveur. Néanmoins, il faut rappeler que les travaux dans [33] étaient plus orientés vers l'établissement d'un nœud actif à hauts débits que le déploiement de code lui-même.

6.4 L'analyse de performance

Dans cette partie, on s'intéresse à l'évaluation des performances des techniques de déploiement de code présentées précédemment. On compare les deux approches *CISS unique* et *multi-CISS* avec une base de codes répartie que nous avons développées dans le cadre de l'infrastructure ARFANet et l'approche *hop by hop*. Nous nous intéressons particulièrement au débit du réseau pour les paquets actifs et la latence de bout en bout pour ce type de paquets.

La topologie du réseau considérée est celle décrite dans la figure 6.3. Dans le cas de l'approche multi-CISS, un second CISS est ajouté entre le nœud A et le nœud C de ce réseau. On fait varier le taux d'arrivée des paquets à ce réseau de 250 $paquets/s$ à 4500 $paquets/s$ pour une taille des paquets de 200 $octets$.

Dans le cadre de l'analyse de l'impact de l'introduction des paquets actifs dans un réseau sur les paquets standards, plusieurs proportions de paquets actifs ont été considérées (voir le Chapitre 5). Les différentes évaluations des performances effectuées ont montré que la proportion qui permet d'introduire un ratio satisfaisant d'applications actives, tout en altérant le moins possible les performances des nœuds actifs vis-à-vis des paquets standards, est $K = 20\%$. Aussi, dans cette étude, nous avons retenu la même proportion de paquets actifs. Cette proportion inclut les paquets de données (d-paquets), de programmes (m-paquets et s-paquets) ainsi que les paquets de requêtes de code envoyées au(x) CISS. Les d-paquets envoyés dans le réseau peuvent référencer quatre différents types d'applications, elles-mêmes appartenant à deux classes de moniteurs actifs. Les moniteurs et les applications sont publiés au niveau du/des serveur(s) pour les approches CISS et sont fournis par la source pour l'approche *hop by hop*.

Nos tests ont été effectués sur des machines Linux avec un processeur "AMD Athlon(tm) XP 1500+" de 1339 MHz, 256 Ko de mémoire cache et une mémoire vive 256Mo.

6.4.1 Le débit

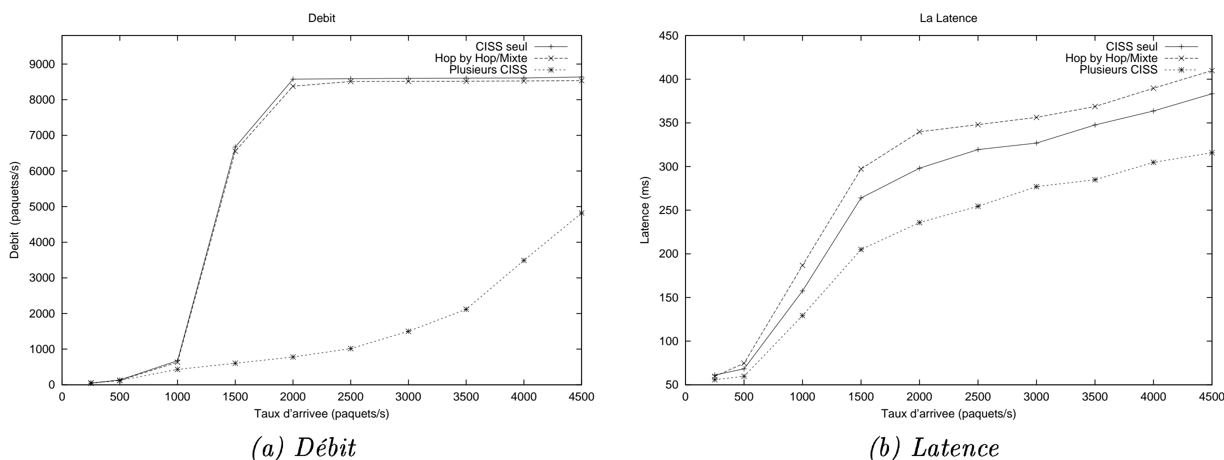
La figure 6.4(a) nous montre le débit du réseau pour les paquets actifs en fonction du taux d'arrivée global des paquets (actifs et passifs) pour les trois techniques. Ces résultats montrent que bien que le débit de la technique *CISS unique* soit légèrement supérieur à celui de la technique *hop by hop*, les deux débits restent très similaires. Nous remarquons que les deux débits se stabilisent à 8500 $paquets\ actifs/s$ à partir d'un taux d'arrivée élevé (2000 $paquets/s$). Ceci est intéressant à noter car la proportion de paquets actifs arrivant au réseau n'est que de 20% c'est-à-dire 400 $paquets\ actifs/s$ quand le taux d'arrivées global est 2000 $paquets/s$.

Ce débit élevé par rapport au taux d'arrivée réel des paquets actifs est dû au fait que les premiers paquets de données d'une application qui arrivent à un nœud vont devoir attendre que le nœud récupère le code correspondant du CISS (approche *CISS unique*) ou du nœud précédent (approche *hop by hop*). Une fois le code récupéré, un traitement immédiat est effectué sur les d-paquets accumulés dans la file provoquant un envoi successif des paquets traités vers le nœud suivant. Ce phénomène se répète pour tous les nœuds actifs traversés par les paquets, d'où un débit beaucoup plus élevé que le taux d'arrivée. Ainsi, tout au long du chemin des données, le rythme de migration des paquets actifs ne respecte pas forcément le rythme de la source.

Par ailleurs, on observe que le débit obtenu avec l'approche *multi-CISS* est nettement inférieur à ceux obtenus avec les deux précédentes approches, en particulier l'approche *CISS unique*. Cela est dû à la présence d'un CISS supplémentaire dans cette approche. Celui-ci permet de désengorger le premier CISS puisqu'il absorbe la partie des demandes de code émanant des nœuds dans son

voisinage immédiat. Ainsi les premiers paquets de données d'une application arrivant à un nœud vont attendre l'arrivée du code au nœud moins longtemps que dans l'approche *CISS unique*. Les paquets auront moins tendance à s'accumuler dans les files des nœuds. D'où un débit moins important que dans les autres approches, bien que encore supérieur au taux d'arrivée.

Le décalage entre le débit du réseau pour les paquets actifs et le taux d'arrivée de ce type de paquets reste nettement moins important que celui noté pour les autres approches. Ainsi pour un taux d'arrivée de 400 $paquets\ actifs/s$, le débit est de 800 $paquets/s$, c'est-à-dire 10 fois moins important que dans l'approche *CISS unique*. On notera cependant que plus le taux d'arrivée augmente, plus le débit augmente jusqu'à atteindre 5000 $paquets/s$ pour un taux d'arrivée global de 4500 $paquets/s$ (900 $paquets\ actifs/s$). Ainsi plus le taux d'arrivée augmente, plus les deux CISS sont sollicités et plus ces derniers sont sollicités, plus les paquets de données vont attendre et s'accumuler dans les nœuds. Ce qui explique l'augmentation de la différence entre le débit et le taux d'arrivée.



(a) Débit *(b) Latence*

FIG. 6.4 – Comparaison des trois approches

6.4.2 La Latence

Dans cette partie, nous étudions la latence des paquets actifs pour les trois approches. La latence est calculée de bout en bout à travers tous les nœuds du réseau.

La figure 6.4(b) montre la latence obtenue pour les différentes approches en fonction du taux d'arrivée des paquets au réseau. Bien que les nœuds doivent récupérer le code du serveur CISS qui peut être quelques routeurs plus loin du nœud sollicitant le code, la latence des paquets actifs est meilleure dans l'approche *CISS unique* que celle de l'approche *hop by hop*. Cela peut s'expliquer par le fait que les nœuds actifs dans la technique *hop by hop* doivent également gérer les requêtes de code émanant d'autres nœuds. Cette tâche supplémentaire a un impact visible sur la latence.

Cette figure montre également que l'introduction d'un CISS supplémentaire (approche muti-CISS) permet d'améliorer encore la latence de bout en bout. Comme expliqué précédemment pour le débit, le second CISS permet de soulager le premier CISS d'une partie des requêtes de code du réseau, celles provenant des nœuds actifs dans son voisinage.

Globalement, il est clair que la gestion du code par une ou plusieurs entités dédiées permet d'avoir de meilleures performances que lorsque cette tâche est laissée aux nœuds eux-mêmes

Les résultats de la méthode mixte (CISS-*Hop by Hop*) sont identiques à celle du *Hop by Hop* pure. La différence entre les deux techniques réside au niveau du premier téléchargement des codes par le premier nœud actif. Dans la méthode mixte, le premier nœud récupère le code du CISS tandis que dans la méthode *Hop by Hop* le premier nœud récupère le code de la source, autrement dit, l'utilisateur. Si le CISS et l'utilisateur sont équidistants du premier nœud les résultats sont forcément identiques. Pour des réseaux à grande échelle, cette condition n'est pas forcément vérifiée car le CISS doit être intermédiaire en les nœuds et les utilisateurs finaux.

6.5 Les caractéristiques de l'approche CISS

Nous résumons dans ce qui suit les caractéristiques de notre approche de distribution de code :

- *Identification :* consiste à faire appel à une autorité centrale qui fournit une référence unique à chaque service d'application et niveau de l'architecture d'un nœud ARFANet. Nous utilisons les deux modes de référencements des applications actives dans les nœuds définis dans le cadre des approches discrête et intégrée. Comme dans l'approche discrête nous installons des écouteurs d'événements pour filtrer les paquets passifs nécessitant un traitement actif. De même que l'approche intégrée, nous définissons la notion de d-paquets qui représentent des paquets actifs référençant explicitement une application active.

- *Sauvegarde du code :* dans le but de partager et réutiliser le code en le rendant accessible à tous les utilisateurs et à tous les nœuds, le code actif est sauvegardé et indexé dans une base. La persistance du code actif au niveau du CISS permet aux nœuds de le supprimer de leur mémoire.

- *Sécurité :* l'autorité centrale peut vérifier la sûreté des codes et authentifier les entités propriétaires et utilisatrices (voir Chapitre 7).

- *Composition de modules :* la modularité et la composition sont les principaux apports d'une programmation évolutive et ouverte. Les règles actives nous offrent ces caractéristiques. Le CISS doit être considéré comme une bibliothèque de modules actifs où chaque utilisateur peut composer lui-même son propre programme en choisissant les modules nécessaires parmi ceux publiés par le CISS. Un module est un ensemble élémentaire de règles actives qui constituent une entité logique et indivisible définissant un nouveau comportement au niveau du nœud ou du réseau ou garantissant la transition d'un état à un autre. Un service d'application est donc défini par une agrégation de modules.
  Une liste de tous les codes peut être publiée sur une page Web. A chaque module actif est associée une description de toutes ses règles et l'effet de l'exécution de leurs actions sur le réseau. Sur cette même page Web, l'utilisateur peut composer sa propre application en choisissant les modules appropriés, dont il a l'autorisation.

- *La répartition de la base de code* : pour les réseaux à grande échelle, il est préférable de répartir la base de code sur plusieurs CISS. Cela permet une meilleure latence et moins de trafic dans le réseau car les utilisateurs et les nœuds contacteront le CISS le plus proche.

- *Les phases de la distribution* : nous distinguons entre la phase de publication du code et la phase du déploiement. La publication concerne l'envoi du code au CISS. Quant au déploiement,

il concerne la phase de réception du code par les nœuds actifs.

6.6 Conclusion

Les procédés d'identification et de déploiement de code définissent respectivement comment le code est identifié par les différentes entités du réseau (les nœuds et les utilisateurs) et comment il atteint les nœuds. Dans ce chapitre, nous avons développé une nouvelle approche basée sur l'utilisation d'un serveur de code *CISS* (Code Identification and Storage Server) qui en plus de stocker le code permet de lui attribuer un identifiant unique. Ce nouveau mécanisme de distribution des codes actifs permet de résoudre le problème des multiples identifications des applications et des multiples chemins de données. De plus, ce serveur permet la vérification du code et l'authentification du développeur. Ce genre de tâches ne peut être affecté au nœud du réseau car ses performances seront altérées. Le *CISS* mixe les avantages des deux approches de distribution de code, l'approche basée sur la pré-programmation de certains nœuds actifs et celle du *hop by hop* qui utilise des capsules de code.

Par ailleurs, nous avons étudié le problème du nombre et de l'emplacement des CISS dans le réseau. Nous avons conclu que la décomposition par domaine du réseau orientera le choix sur l'emplacement et le nombre des CISS. Chaque CISS doit être près des utilisateurs et des nœuds actifs. Les CISS distribués implémenteront forcément une base de codes distribuée. Nous nous sommes inspirés des bases de données répliquées et des bases de données réparties pour choisir la bonne technique. Initialement, nous avons considéré les deux éventualités. Cependant, les caractéristiques des CISS et la décomposition du réseau en domaines, la base de codes distribuée est plus appropriée au contexte distribué des CISS.

Par souci de flexibilité et pour réduire la dépendance des nœuds aux CISS, nous nous sommes inspirés du modèle de déploiement *hop by hop* pour créer une nouvelle approche mixte qui combine les avantages de l'approche CISS avec le *hop by hop*. Cette combinaison a montré les capacités de notre approche CISS à s'intégrer à d'autres approches de distribution.

Nous avons comparé l'approche *CISS* unique avec l'approche *hop by hop* en termes de latence et de débit. Les résultats obtenus ont montré que l'utilisation du serveur de codes donne de meilleures performances. Par ailleurs, l'ajout d'un ou de plusieurs serveurs de codes dans le réseau permet d'améliorer considérablement ces résultats. Bien que notre approche ait été développée dans le cadre de ARFANet, elle peut être utilisée pour tout genre d'applications actives et dans plusieurs plates-formes existantes.

Dans le chapitre, suivant nous introduisons des techniques de sécurisation des différentes phases de la distribution du code. Ces techniques ont pour vocation d'authentifier les différents acteurs liés à la distribution du code tels que le développeur et l'utilisateur et aussi d'assurer la sûreté du code. Il proposera aussi quelques techniques employées par ARFANet pour sécuriser l'exécution des règles actives qui par leur propre format permettent de localiser et partager l'effort de sécurité sur l'ensemble des composantes d'une règle.

Chapitre 7

Les mécanismes de sécurité

7.1 Introduction

Au-delà des avantages d'ouvrir le réseau pour introduire rapidement de nouveaux protocoles et applications, l'injection d'un programme actif dans les nœuds du réseau peut affecter leur performance et avoir des répercutions négatives sur le réseau tout entier. Exécuter des programmes actifs par les routeurs peut détériorer les fonctions de routage des paquets standards et augmenter considérablement les failles du réseau en termes de sécurité.

Habituellement, les renforcements de sécurité sont actionnés dans les architectures de réseau actif en tant que couche supérieure ou comme un composant indépendant ajouté à la fin du processus de conception. Ces renforcements sont typiquement une interface de contrôle d'admission ou un simple mécanisme d'authentification basé sur les infrastructures à clef publique (PKI) ou sur la cryptographie. Cependant, les techniques de sécurité ne sont pas intégrées dans les couches inférieures comme la distribution et l'exécution de code. Notre but est d'établir une architecture globale où le système de sécurité est inclus dès la conception des processus de distribution et d'exécution des codes. Dans notre architecture, le mécanisme de sécurité agit sur tous les acteurs qui sont concernés par l'injection des programmes dans le réseau et leur exécution. Ces acteurs ou entités peuvent être identifiés comme le *fournisseur* ou le *développeur* de code, l'*utilisateur* du code et le *code* lui-même. Puisqu'ils sont externes au réseau, ces trois entités peuvent rendre le réseau plus vulnérable du point de vue sécurité et nous poussent à définir un mécanisme interagissant avec chacun d'eux selon ses spécificités. Cela se traduit par une identification unique, une forte authentification et une classification des entités citées précédemment selon leurs droits.

La sécurité dans la distribution du code consiste à renforcer toutes les phases du déploiement, c'est-à-dire, la publication, l'identification, le référencement et le téléchargement du code dans le réseau. Les renforcements de la sécurité dans les phases de publication et de référencement du code doivent garantir que seuls l'utilisateur et le développeur, ayant les droits nécessaires, peuvent injecter et déployer un programme à travers les nœuds du réseau.

La fonction de sécurité du côté développeur doit au moins vérifier si ce dernier est habilité à publier son code. Les développeurs de codes doivent être authentifiés et classés en différents groupes. La distinction entre les groupes est basée sur leur relation avec le réseau, l'accès aux ressources des nœuds et le champ d'application des utilisateurs de ce code. Différentes classes peuvent être définies afin de donner plusieurs niveaux de permissions. La distinction entre les développeurs crée un ensemble de catégories de codes. Chaque catégorie de programmes correspond à une classe

d'utilisateurs. Cette relation à trois pôles entre le code, le fournisseur et l'utilisateur nous mène à définir une architecture de sécurité à trois pôles également.

D'autre part, l'arrivée au nœud d'un code contrôlé et supposé sûr lors des phases de publication et de déploiement ne garantit pas le bon déroulement de son exécution et ne protège pas le nœud et le réseau de certaines exécutions exceptionnelles de codes. Les environnements d'exécution (EE) et les systèmes d'exploitation des nœuds (NodeOS) doivent jouer un rôle important afin de maintenir les ressources du nœud disponibles et opérationnelles. La sécurité lors de l'exécution doit prévenir de toute tentative des programmes actifs de corrompre les fonctionnalités des nœuds.

Dans ce chapitre, nous étudions la question concernant la sécurité dans le contexte de la plateforme ARFANet. Nous établissons quelques conditions minimales et nécessaires pour qu'un réseau actif soit sécurisé et nous proposons un nouveau mécanisme qui comprend des autorités centrales agissant en tant que serveurs de code pour le stockage des applications actives et une autorité de certification pour authentifier et classer les entités concernées. Nous affectons au serveur d'identification et de stockage de code (CISS), vue dans le chapitre précédent, les tâches de sécurité telles que l'identification et la vérification de l'unicité et de la sûreté des codes. Cette vérification, qui est très coûteuse en temps et en mémoire, ne peut pas être faite par les nœuds en raison de la dégradation des performances qui en résulterait.

Nous proposons une amélioration du mécanisme basé sur les clefs symétriques [66] pour définir un protocole d'autorisation d'exécution des programmes par les utilisateurs sans qu'il rajoute un surplus de traitement aux nœuds.

Pour une exécution plus sûre, nous proposons quelques techniques de sécurité inhérentes aux règles actives et leur cycle de vie. Ces mesures de sécurités sont prises au niveau de la machine virtuelle et des moniteurs actifs d'un nœud actif ARFANet.

Dans la section suivante, nous présentons les conditions essentielles à l'établissement d'un réseau actif sûr et fiable. La section 7.3 est consacrée à la définition des mécanismes de sécurisation lors de l'introduction du code dans le réseau et son acheminement jusqu'aux différents nœuds. Dans la section 7.4, nous présentons les faiblesses et failles des nœuds actifs lors de l'exécution d'une application et les mécanismes susceptibles de protéger le nœud pendant cette phase. La section 7.5, conclut ce chapitre.

7.2 Les règles de sécurité dans les réseaux actifs

Un réseau permet de relier des utilisateurs éloignés par des moyens de télécommunications. L'acheminement des données de la source à la destination est fait par les routeurs qui, en utilisant des protocoles bien définis, analysent l'entête des paquets pour déterminer leur chemin. Il faut noter que les routeurs classiques ne regardent pas le contenu des paquets.

Vu de l'extérieur, le réseau est comme une entreprise de distribution de lettres et de colis où les clients, à l'extrémité, ne font qu'envoyer et recevoir des lettres. Nous pouvons constater que les clients ne peuvent pas connaître les mécanismes de distribution de l'entreprise et ne peuvent pas agir ou intervenir dans ce processus. Mais l'entreprise peut déléguer pour certaines lettres et certains colis leur acheminement partiel à des entreprises de sous-traitance à la convenance du client afin d'améliorer le service et diminuer le délai.

Dans les deux cas, le réseau de transmission de données et le réseau du courrier, les clients ne peuvent pas, par eux mêmes, intervenir sur les paquets envoyés à travers le réseau. Néanmoins, ils peuvent faire appel aux services de fournisseurs intermédiaires.

Dans cette perspective, la mise à disposition d'applications actives doit être limitée à des sous-traitants certifiés par les autorités de normalisation ou les organismes qui gèrent les réseaux des télécommunications. Ces sous-traitants sont des *développeurs* d'applications actives pour offrir des services qui amélioreront, par exemple, le transfert ou établir des qualités de service pour les clients privilégiés. Les clients voulant personnaliser le traitement de leurs paquets, en choisissant un service donné, sont les *utilisateurs* du *code actif*. Ces entités régissent le nouveau comportement du réseau, et par conséquent le fragilisent encore plus du point du vue sécurité.

Toutes ces entités peuvent, à la fois, être victimes et source d'attaques. Le code peut être victime lorsqu'il subit une tentative d'analyse et de modification de son contenu. Il peut être dangereux et nuisible lorsqu'il tente de compromettre l'exécution d'un autre code, de l'environnement d'exécution ou du nœud. Le développeur peut être victime d'usurpation de son identité. Il peut être dangereux s'il essaie d'exécuter un code qui ne correspond pas à ses droits. L'utilisateur est victime s'il y a un autre utilisateur qui tente d'intercepter ses données en utilisant un autre code actif. Il peut être malicieux s'il tente de compromettre l'exécution d'un autre code, du EE, du nœud ou accéder à des données et des espaces mémoires non autorisés.

Les conditions minimales au déploiement de code doivent considérer la sécurité dans trois points : le développeur de code, l'utilisateur de code et le code lui-même.

- *Développeur :*
  - une identification unique du développeur,
  - une authentification du développeur,
  - des droits hiérarchisés pour publier leurs codes selon leur effet sur le réseau,
  - la non répudiation des codes publiés.
- *L'utilisateur :*
  - une identification unique de l'utilisateur,
  - une authentification de l'utilisateur,
  - des droits hiérarchisés pour exécuter le code,
  - la possession d'une preuve de l'autorisation à exécuter le code à fournir aux nœuds actifs.
- *Le code :*
  - une identification unique du code,
  - un mécanisme de contrôle de la sûreté du code,
  - une utilisation limitée des ressources des nœuds (en particulier, limiter l'espace mémoire alloué et le temps d'exécution et de séjour),
  - l'interdiction des accès à des zones mémoires non autorisées.

Par ailleurs, une classification au sein des développeurs et des utilisateurs séparément exige une association à chaque classe de développeurs d'une classe d'utilisateurs. A l'instar de l'adresse IP qui permet de distinguer un utilisateur, un routeur et même un serveur Web, chaque entité liée aux réseaux actifs (telle que les CISS, les codes actifs, les nœuds, les utilisateurs, les développeurs et le serveur web de publication) doit avoir un certificat ou au moins une clef (publique et privée) certifiée par une autre autorité de confiance. Cette attribution de clefs à chaque entité du réseau actif est indispensable pour authentifier les interlocuteurs et, est la base de toute communication sûre.

Nous distinguons deux principaux niveaux de sécurisation de l'ouverture du réseau aux applications utilisateurs. Ces deux niveaux sont liés à l'introduction du code actif dans le réseau et son exécution par les nœuds. Dans la prochaine section nous développons des techniques liées à

l'introduction des programmes dans le réseau.

7.3 La sécurité dans la distribution du code

Dans le chapitre 2, nous avons vu que tous les projets de recherche, qui se sont intéressés à la sécurité des réseaux actifs, se sont focalisés sur l'aspect exécution du code au niveau des nœuds en définissant des langages restreints comme PLAN [69] ou de plate-formes sécurisées comme SANTS [71]. Nous déplorons le manque de contributions pour la sécurité de la distribution du code,

L'architecture de la distribution de code que nous avons présentée dans le chapitre précédent, nous a poussés à intégrer, aux différentes phases de déploiement de code, des mécanismes de sécurité appropriés en se basant sur des techniques existantes dans différents domaines tels que l'analyse du code, génération clefs symétriques et génération de la preuve de validité d'un code.

Pour respecter les nouvelles contraintes de sécurité imposées par une infrastructure active qui recommandent une affectation de clefs aux différentes entités, nous adoptons une approche qui se base sur l'utilisation d'un serveur de certificats et de serveurs de code auxquels on assigne plusieurs tâches dont l'attribution des clefs aux codes et des clefs symétriques aux utilisateurs. Ces clefs sont valides durant une session active. Nous introduisons ci-après ces autorités.

CAAN (Certificate Authority for Active Network) : est une autorité de certificats pour le réseau actif. C'est une autorité autonome qui fournit un certificat PKI selon la norme des certificats X509 [1]. Les certificats doivent contenir des informations sur le développeur et l'utilisateur (identifiant, nom, adresse, organisation, rôle dans l'organisation, ...), sa classe, ainsi que sa clef publique. Le CAAN, fournit au souscripteur le couple de clef privée et publique.

CISS (Code Identification and Storage Server) : est un serveur de code auquel on peut affecter d'autres tâches de vérification et d'authentification en relation avec le code qu'il détient.

7.3.1 La notion de classes de développeurs et d'utilisateurs

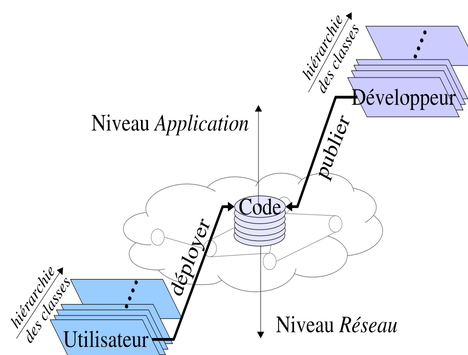


FIG. 7.1 – Hiérarchisation des utilisateurs et des développeurs

Dans les réseaux actifs, on s'attend à ce que la plupart des utilisateurs ne programment pas les nœuds du réseau. Cependant, le nombre de développeurs est limité, tandis que, le nombre d'utilisateurs est susceptible d'être grand et peut même s'exprimer en fonction du nombre des

utilisateurs des réseaux actuels (tels que Internet par exemple) si le taux d'utilisation des services actifs est significatif.

Le principe d'un administrateur qui gère les droits des utilisateurs, doit être adopté dans les réseaux actifs. Il faut noter qu'il n'est pas permis à tous les utilisateurs d'exécuter ou de publier tous les services d'application disponibles sur les CISS. Dans les réseaux actuels, la distinction entre les différentes classes du développeurs et d'utilisateurs existe belle et bien. Nous pouvons considérer les établissements et les entreprises qui participent au niveau transmission physique pour gérer les commutateurs ATM des réseaux cœur, les routeurs et les protocoles parmi les éléments de la classe prioritaire (la super classe). Au niveau fourniture d'accès ou service logiciel aux utilisateurs finaux et les réseaux locaux (réseaux d'entreprise ou universitaires), les fournisseurs d'accès peuvent être considérés comme une deuxième classe (la classe intermédiaire). Fondamentalement, l'exemple de ces développeurs est l'administrateur de réseau ou le fournisseur d'accès Internet (Internet Service Provider - ISP).

Ces entités ont déjà acquis des droits par des autorités de normalisation et sont déjà répertoriées (ou peuvent aisément l'être) par des organismes spécialisés et classées selon le degré de manipulation du réseau. Ces deux classes ont la permission d'opérer au niveau réseau et peuvent changer le comportement du nœud en présentant un nouveau protocole.

Néanmoins, il reste les utilisateurs simples qui n'appartiennent à aucune classe. Nous pouvons les grouper dans la classe la moins prioritaire (la classe utilisateur) où ils ne gèrent que leurs propres données. Chaque classe peut également être divisée en plusieurs sous-classes. Nous supposons que la classe la plus restrictive est la classe où l'utilisateur est autorisé à ne contrôler que ses propres paquets de données.

Cette distinction est un premier pare-feu contre un code malveillant et un contrôle fin sur l'exécution des codes. Elle est réalisée par le CISS en collaboration avec CAAN durant la phase de publication de code.

L'accès et l'attribution des ressources du nœud dépendent de la classe du développeur pour une utilisation restrictive. Les ressources convoitées du nœud sont typiquement : la table de routage, le système de fichiers, la mémoire et le temps d'exécution. La création de nouveaux paquets qui nécessitent une ressource réseau, est restreinte à certaines classes d'utilisateurs. Cette action n'est pas considérée par de nombreux projets comme un accès à une ressource et n'est pas du tout contrôlée.

Il est tout à fait possible que CAAN, à travers un site web, attribue des certificats temporaires ou renouvelables, avec des clefs, aux utilisateurs simples. Ce procédé est largement répandu dans les applications Internet où les transactions sécurisées sont exigées tel que le E-commerce, en utilisant par exemple SSL (Secure Socket Layer)[48]. En ce qui concerne les classes supérieures (la super classe et la classe intermédiaire), la démarche est plus réglementée. Les certificats, dans ce cas, sont délivrés aux organismes par la voie officielle.

7.3.2 Sécuriser la phase de publication

Cette étape se charge de garantir les règles minimales concernant le développeur. L'attribution d'un certificat au développeur garantit son identification et l'utilisation des mécanismes de cryptographie à clefs publiques permet de l'authentifier. Les sous-sections suivantes définissent ces mécanismes.

7.3.2.1 L'identification du développeur

L'identification du développeur doit nous permettre de déterminer exactement à quelle classe il appartient. Le CISS doit lui-même contrôler cette tâche. Dans ce cas, le CISS demandera la classe du développeur à CAAN, car le certificat du développeur contient la désignation de sa classe (message *Demande d'authentification* dans la figure 7.2). L'attribution de l'identifiant au développeur peut être effectuée par cette autorité de confiance ou en utilisant des mécanismes de PKI avec une autre autorité certifiée et reconnue par CAAN (comme les administrateurs de l'organisme réseau auquel est attaché le développeur).

7.3.2.2 L'authentification du développeur

Un type d'attaques possible est l'usurpation de l'identité d'un développeur certifié. Dans ce cas, un tiers mal attentionné peut vouloir prendre l'identité d'un développeur d'une certaine classe afin de publier un code malveillant. Le CISS doit, avant d'accepter tout code, authentifier son interlocuteur.

Le procédé d'authentification est également basé sur PKI. Au début de ce processus, le CISS envoie une information privée (un texte simple tel que le nom du développeur ou une donnée aléatoire pour éviter toute tentative de briser l'algorithme de cryptage derrière) au développeur (message (1) dans la Figure 7.2). Quand le développeur reçoit cette information, il la crypte avec sa propre clef privée et renvoie l'information chiffrée au CISS. En utilisant la clef publique du développeur, le CISS déchiffre l'information reçue. Si l'information déchiffrée et l'originale concordent alors le développeur est authentifié. Autrement, c'est une autre entité agissant au nom du vrai développeur. En fait, avec cette technique le développeur est fortement identifié et authentifié et ne peut pas nier l'envoi du code (éviter la répudiation et le déni du service). Ainsi, le développeur non certifié et non identifié ne peut ni envoyer ni publier un code.

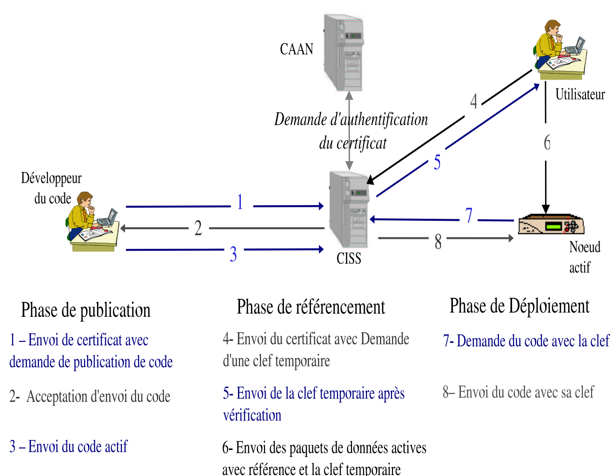


FIG. 7.2 – les phases de distribution de code sécurisée

7.3.3 Sécuriser la phase de déploiement

Les nœuds recevant un paquet nécessitant une exécution d'un programme actif doivent s'assurer que cet utilisateur possède l'autorisation qui lui permet d'avoir un traitement particulier. En plus, cet utilisateur doit authentifier son identité afin d'éviter toute usurpation de celle-ci.

7.3.3.1 L'identification de l'utilisateur

L'arrivée d'un paquet actif au nœud mettant en référence un code actif, ne signifie pas une exécution systématique du code. Il est exécuté seulement dans le cas où l'utilisateur fournit les preuves de la possession des droits et des permissions requis. Comme le développeur, l'utilisateur doit acquérir un certificat auprès de CAAN avec une clef publique à authentifier et une classe. Cette phase d'acquisition de certificat doit être antérieure à l'envoi des paquets. Avant l'envoi de ses paquets actifs, chaque utilisateur est contraint de fournir la preuve, sous forme de ticket ou clef, l'autorisant à exécuter le code, le message 4 dans la figure 7.2.

Dans la technique de génération symétrique et distribuée de “jeton de permission” (credential), employée par ROSA [66] [65], la clef est produite pour un utilisateur donné. Cette clef est la combinaison de l'adresse de la source émettrice (S), de la clef de code (Kc) à exécuter, de la clef de l'utilisateur, de la date à laquelle la demande de l'autorisation a été formulée par l'utilisateur (T) et d'une période de validité (P). La clef (K) est construite par le serveur d'autorisation, à la demande de l'utilisateur, et envoyée à ce dernier après vérification de ses droits sur le code. L'utilisateur envoie ses paquets contenant K. A la réception du paquet, le nœud actif peut, avec les informations concernant la session véhiculées par le paquet telles que la Source (S), la clef de l'utilisateur, la Durée (D) et la période (P), reconstituer la clef de code Kc à partir de la clef symétrique K extraite du paquet. Si la clef de code reconstituée et celle envoyée au nœud par le serveur de code en même temps que le code concordent, alors l'utilisateur est autorisé à exécuter le code. Avec cette technique, le nœud actif peut vérifier l'authenticité et les droits des utilisateurs sans demander à une autre autorité et sans procéder à des calculs supplémentaires ou détenir une liste additionnelle des utilisateurs autorisés.

Les limites de la technique ROSA

Dans cette technique [66], le code est supposé dans des serveurs de codes et contenant un identifiant unique. Elle ne définit pas comment gérer l'unicité des identifiants des codes et leur partage. La clef de code Kc est partagée entre les serveurs de codes et le serveur d'autorisation et doit être changée fréquemment pour garantir sa sécurité. Par conséquent, Kc n'est pas uniquement liée au code car sa valeur change régulièrement au cours du temps. Le routeur doit la télécharger du serveur de code avec le code. Donc, Kc doit être mise à jour régulièrement entre le serveur qui la génère (le serveur d'autorisation), les autre serveurs de codes et les nœuds. En ce qui concerne la clef symétrique K, elle n'est pas à sens unique car il est possible de lui extraire la clef de code qui la compose.

Les améliorations proposées

Nous pensons qu'il n'est pas très judicieux de changer ou de partager entre plusieurs serveurs la clef d'un code. Par contre, il est préférable de sécuriser le processus de génération et de transfert de la clef du code. La clef produite correspond à une clef liée au code lui-même. L'adaptation de cette

méthode dans le contexte d'ARFANet consiste à affecter au CISS le contrôle de la clef du code, sa génération et son stockage. Cette clef n'est pas partagée avec une autre autorité, comme défini par ROSA. Il n'existe pas un serveur d'autorisation, le CISS joue ce rôle. Ce qui permet de supprimer tout besoin de partager la clef du code et faciliter la mise en œuvre d'un mécanisme de contrôle de sécurité et diminuer la communication entre les serveurs.

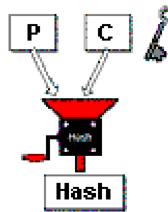


FIG. 7.3 – Utilisation de la fonction de Hashage

Pour sécuriser cette clef de code, nous proposons que la clef soit une signature numérique, voir figure 7.3. Nous proposons aussi que chaque CISS contienne un certificat avec un couple de clefs (privée, publique). Le CISS signe avec sa clef privée le code pour obtenir un condensât à sens unique de 128 octets. Ce genre de condensât permet au nœud de vérifier, d'un seul coup, que le code n'a pas été altéré en cours de route et aussi d'authentifier le CISS qui l'a envoyé. Donc la clef du code est indépendante de son développeur tout comme son identifiant. Par contre, la clef est liée au contenu du code et de l'autorité qui le détient (CISS).

Le reste du processus de la méthode ROSA est inchangé. L'utilisateur se connecte au CISS pour formuler une demande d'autorisation. Le CISS envoie la clef K à l'utilisateur (message 5 dans la figure 7.2). La tâche ne peut être accomplie que si l'utilisateur a les droits pour déployer ce code. A ce mement l'utilisateur peut envoyer ses paquets à travers le réseau (message 6 dans la figure 7.2). Tout nœud actif dans le chemin des paquets recevant les paquets doit extraire la clef symétrique K, l'identificateur du code et les informations concerant la session utilisateur. La clef symétrique K est aussi un condensât à sens unique. Le nœud ne doit pas extraire la clef du code Kc de la clef K, mais il doit reconstituer la clef K à partir de la clef de code Kc reçu du serveur de code (messages 7 et 8 de la figure 7.2) et les autres paramètres (la source, la date et la période).

Nous remarquons que la technique n'engendre aucun traitement supplémentaire, ni de connexions vers d'autres serveurs d'autorisations.

7.3.3.2 L'authentification de l'utilisateur

Nous utilisons le même système d'authentification que celui du développeur qui est basé sur la vérification à clef publique. L'utilisateur doit souscrire à CAAN pour obtenir un certificat. En retour, CAAN attribue à ce nouvel utilisateur une clef publique, une clef privée correspondante et une classe. Lors de la phase de consultation des service d'application, comme le montre la figure 6.1, le site web de publication authentifie l'utilisateur qui fournit son certificat. Seules les pages web qui définissent les services correspondant à la classe de l'utilisateur lui seront affichées.

7.3.3.3 En cas de plusieurs CISS

Du point de vue développeur ou utilisateur, il n'y a aucune différence entre un ou plusieurs CISS. Chaque CISS recevant un code doit suivre les mêmes consignes de sécurité et générer la clef du code

signée par sa propre clef privée. Pour les demandes d'utilisation des codes, le CISS qui reçoit une demande de clef symétrique temporaire pour un code de sa base suivra les mêmes démarches. Par contre, dans le cas où le code n'est pas dans sa base, il est dans l'obligation de demander au CISS possédant le code, en consultant sa *table de références globales*, de lui générer la clef temporaire, qu'il communiquera par la suite à l'utilisateur.

7.3.3.4 En cas de distribution mixte

La distribution mixte consiste à appliquer le mécanisme de distribution de code en combinant le CISS avec la méthode *Hop by Hop*. La méthode *Hop by Hop*, rappelons-le, se base sur un déploiement de proche en proche où chaque nœud doit demander le code au nœud précédent. Le nœud recevant le code reçoit aussi la clef du code et la désignation du CISS le détenant. Puisque la clef du code est un condensât du code signé par la clef du CISS, le nœud peut vérifier qu'il est bien certifié par le CISS signataire et qu'il n'a pas été changé par le nœud précédent.

Le premier nœud du chemin des données procédera au rapatriement du code ainsi que sa clef à partir du CISS. Les prochains nœuds vont devoir les demander aux nœuds précédents. Pour cette raison, certifier la clef du code et le code lui-même est la condition minimale pour se prémunir contre de "faux" nœuds précédents qui essaient d'injecter un code malveillant. Dans les réseaux classiques les routeurs doivent se faire confiance pour mettre à jour leur table de routage. Cette façon ne sera plus possible dans les réseaux actifs. Dorénavant les routeurs doivent s'authentifier mutuellement avec leurs clefs respectives.

7.3.4 La sûreté du code

Le CISS peut tout à fait appliquer plusieurs méthodes pour garantir une exécution sûre des codes actifs. Parmi ces méthodes, nous pouvons citer la technique Proof-Carrying Code (PCC) et les systèmes d'analyse des programmes.

7.3.4.1 Vérification du code avec PCC

Proof-Carrying Code (PCC) est utilisé pour une exécution sûre d'un code d'une source distante et inconnue ou non authentifiée. Dans un exemple typique de l'application du PCC, un récepteur de code établit un ensemble de règles de sûreté qui garantissent le comportement fiable des programmes. De son côté, le producteur (développeur) de code crée une preuve formelle de sûreté qui confirme, pour le code en question, son adhésion aux règles de sûreté formulées par le récepteur. Puis, le récepteur, à la réception d'un code et de sa preuve, peut employer une simple et rapide validation de la preuve pour avoir la certitude qu'elle est valide. Ceci garantit que le code reçu est sûr lors de son exécution [73].

Dans notre contexte, le CISS correspond au récepteur de code et définit les règles pour la sûreté de code. Selon la classe du développeur, le CISS exige différentes règles. Les règles pour le développeur de code de priorité faible (niveau données) sont plus restrictives que les règles pour le développeur de code d'une classe prioritaire (niveau réseau).

7.3.4.2 Analyse du code

Il existe des systèmes qui permettent d'analyser le contenu d'un programme actif et d'en estimer la consommation en espace mémoire et en temps d'exécution (surtout pour les byte-code interprétés

comme Java [15] [16], largement utilisés dans les réseaux actifs) [23]. Ce genre de systèmes d'analyse de programmes est très coûteux en temps et en mémoire et ne peut pas être effectué par le NodeOS.

Ils permettent aussi de signaler toutes les demandes en ressources effectuées par les programmes et d'interdire celles qui ne sont pas conformes.

7.4 La sécurité dans l'exécution du code

La tâche de la vérification de l'exécution et les demandes en ressources par une application active doivent-elles être du ressort du NodeOS ou de celui de l'EE ? Est ce que le NodeOS fera suffisamment confiance au EE pour contrôler ses AA afin de respecter la sécurité du nœud et ainsi lui léguer le contrôle de la sécurité ? La réponse à ces questions reste toujours ambiguë dans les réseaux actifs. Selon un autre point de vue, le NodeOS ne fait pas confiance aux EE et juge plus serein d'assurer lui-même la gestion des ressources allouées aux AA.

Nous avons vu dans le chapitre 2, qu'au niveau de l'exécution des programmes de manière sécurisée, les projets de recherche se sont focalisés sur deux techniques. La première technique définit une plate-forme sécurisée, quant à la deuxième, elle consiste à définitir un langage restreint. Notre contribution dans la sécurisation des exécutions touche ces deux aspects. L'utilisation des règles activs s'inscrivent dans le cadre des langages restreints, car leur forme supprime les instructions de boucle et de test qui sont en général la cible des attaques. De plus, nous avons sécurisé notre plate-forme au niveau du moniteur actif et de la machine virtuelle en ce qui concerne les accès aux données, la génération des paquets et l'allocation de l'espace mémoire.

Nous traitons dans cette partie des risques dans les exécutions de codes et les solutions apportées par ARFANet. Comme la machine virtuelle partage quelques fonctionnalités du NodeOS et le moniteur actif peut être considéré comme un EE, nous attribuons à chacun, selon son rôle et selon les caractéristiques propres aux règles actives les mécanismes de sécurité à appliquer.

7.4.1 Utilisation des ressources du Nœud

Les deux ressources critiques d'un routeur sont la mémoire et le temps CPU pour l'exécution des applications.

7.4.1.1 L'accès mémoire

Les deux violations de cette ressource sont l'accès à des espaces mémoires non autorisés et les stockages abusifs ou le dépassement de l'espace de stockage autorisé.

L'accès non autorisé

Il se produit lorsqu'un programme actif essaie d'accéder à un espace mémoire qui ne lui appartient pas. Par exemple, une application active tente de lire les données d'une autre application.

Dans le cadre de l'infrastructure ARFANet, une application active ne peut pas accéder aux données des autres applications. Les données des services d'application sont représentées en mémoire sous forme d'une table indexée par le triplet (la session active, le service d'application, le moniteur actif). Cette représentation mémoire est gérée par la *machine virtuelle* et cachée par des appels systèmes. Chaque action ou condition d'une règle voulant récupérer une donnée correspondante ne doit préciser que le numéro de la session au service d'application. Celui-ci encapsule cette demande

en ajoutant son propre identifiant au moniteur actif. De même, le moniteur actif, lorsqu'il reçoit une demande de données, doit l'encapsuler en ajoutant son propre identifiant et l'envoyer à la machine virtuelle qui récupère les données. Un niveau ne sait pas comment le niveau supérieur gère les demandes en données. Ce qui constitue une protection contre des accès interdits. Le système refuse toute autre façon de récupérer les données par les règles actives.

Le stockage abusif

Ce cas se produit lorsqu'un programme actif ne respecte pas l'espace de stockage qu'il lui est réservé. Dans ce cas, il arrive que le nœud trouve sa mémoire saturée même pour recevoir ou envoyer les paquets simples de données (ses files attentes sont saturées). Dans ARFANet, les données de l'application chargées en mémoire sont récupérées du paquet. La taille allouée est en fonction de la taille du paquet (hormis l'entête). Si les paquets sont fragmentés (au passage d'un sous-réseau dont le protocole exige des tailles de paquets plus petites), nous pouvons utiliser l'indicateur de la fragmentation et le numéro séquentiel des fragments pour allouer d'autres espaces mémoire sans pour autant détruire ou remplacer l'espace existant. La prise en compte des fragments permet d'avoir toutes les données nécessaires de la session si les règles actives doivent appliquer un traitement global. Cette permission ne doit pas être considérée comme une faille car dans tous les cas et indépendamment du/des paquet(s) de l'application utilisateur, la machine virtuelle limite à une taille max qui ne doit pas être dépassée.

7.4.1.2 Le temps d'exécution

Les actions des règles actives sont, par définition, des actions primitives ou composées d'un ensemble de sous actions primitives. Les restrictions au niveau du langage d'écriture des conditions et des actions sont comme suit :

- *L'exécution des boucles* : les boucles de la forme : *for (...)*, *while (expression)...*, *do ...while (expression)* ne sont pas acceptées pour éviter toute boucle infinie.
- *L'exécution des tests* : les tests de la forme : *if (expression) then ..., switch(cas) ...*, ne sont pas acceptés dans les conditions et les actions car la forme d'une règle active permet d'exprimer des conditions complexes.

L'absence des boucles et des tests n'exprime pas une faiblesse des règles actives. Le mode ECA permet, avec sa façon de formuler les applications en utilisant des déclenchements récursifs ou mutuels des règles, d'exprimer de manière exhaustive les boucles et les conditions. Vue la multitude des domaines d'application des règles actives (bases de données, systèmes experts, l'intelligence artificielle, ...) et le nombre et le type d'applications à base des règles actives, telles que le multicast (voir Chapitre 4), cotations directes et enchères [11], surcharge d'un serveur Web et P2P (voir Chapitre 8), il est tout à fait possible d'exprimer tout genre d'application sous forme d'ECA.

L'utilisation du PCC pour la preuve de la conformité des codes permet, en formulant des règles spécifiques interdisant l'expression des boucles et des conditions, de prouver que le code ne contient pas ce genre d'instructions.

Bien que les boucles ne sont pas autorisées, les cycles infinis peuvent se former dans le cadre des règles actives par des déclenchements récursifs de la même règle ou des déclenchements mutuels entre plusieurs règles. Un cycle est le retour à la même règle au bout d'un certain nombre de pas.

Comme les règles sont traitées séparément, il est facile de détecter des cycles infinis et de les rompre. Puisque le rôle du moniteur actif est de gérer l'exécution des règles (détecter l'événement, la

vérification de la condition et l'interprétation ou l'exécution des actions), il doit également détecter les cycles infinis.

Deux politiques sont envisagées et basées soient sur le temps de séjour en mémoire ou le nombre de cycles d'un service d'application. Une action qui déclenche une autre règle du même service d'application allonge le temps de présence de ce dernier en mémoire et son temps d'exécution. Si ce temps dépasse un certain seuil prédéfini, le moniteur actif se voit dans l'obligation de retirer ce service de la mémoire.

Dans la deuxième politique, on considère le nombre de cycles d'un service d'application comme critère d'arrêt des déclenchements récursifs infinis. A la détection d'un événement, le moniteur déclenche toutes les règles concernées par cet événement. L'exécution de cet ensemble de règle est considérée comme un cycle. Les règles déclenchées de l'exécution des actions des règles de premier ensemble appartiennent à un deuxième cycle, et donc à un deuxième ensemble de règles. Chaque déclenchement augmente le nombre de cycles des services d'application. Au bout d'un certain nombre de cycles, le moniteur supprime le service d'application. Il existe deux types de cycles.

Les cycles de longueur un : ce sont les cycles formés après un autodéclenchement. Ils sont simples à détecter à l'exécution comme à la compilation. Dans ce cas, une action déclenche sa propre règle. Le moniteur actif, en attribuant un compteur à chaque règle et en l'incrémentant à chaque passage, peut constater si le compteur dépasse le seuil de tolérance. Si c'est le cas, le moniteur peut arrêter l'exécution de la règle seulement, sans pour autant arrêter le service d'application tout entier.

Les cycles de longueur supérieure à un : ceux sont les cycles qui font intervenir plus d'une règle. Cela nécessite que le moniteur actif sauvegarde pour chaque service d'application l'ensemble des déclenchements des règles pour pouvoir vérifier si un cycle infini est formé en faisant plusieurs parcours. Cette solution semble difficile à réaliser, nous nous contentons d'utiliser l'une des politiques précédemment citées où le temps de séjour global d'un service d'application ou son nombre de cycle est considéré comme critère de la suppression du service.

7.4.2 Utilisation des ressources du Réseau

Les deux ressources critiques d'un nœud sont la capacité des liens et la table de routage dans le contexte réseau.

7.4.2.1 La capacité des liens

Le réseau est inondé lorsque plusieurs paquets actifs se dupliquent ou génèrent des paquets de données plusieurs fois sur chaque nœud. Cela peut être évité si on contrôle le débit des paquets actifs à la sortie des nœuds. Ce débit est proportionnel au taux d'arrivée de ces paquets à l'entrée du nœud et au taux de génération de nouveaux paquets qui est liée à la nature des applications.

La proportion des paquets actifs à l'entrée d'un nœud doit être maintenue à la sortie. En effet, sur une suite de nœuds, ayant le même comportement, si la proportion des paquets actifs augmente à la sortie de chaque nœud, la proportion des paquets actifs à la sortie du dernier nœud peut atteindre une valeur très élevée même si celle de l'entrée du premier nœud était petite. Il faut, par conséquent, maintenir la même proportion des deux extrémités des nœuds. Une étude sur le choix de la proportions des paquets actifs est présentée dans le Chapitre 5. Avec la comparaison entre la proportion des paquets actifs à l'entrée et à la sortie en fonction du taux d'arrivée global, nous avons

pu démontré que la proportion des paquets actifs à la sortie du nœud est peu liée à celle en entrée. Ceci s'explique tout particulièrement par le fait que les paquets actifs remontent vers les couches supérieures. Néanmoins, la génération des paquets peut influencer considérablement la proportion à la sortie. Donc, il faut faire un contrôle assez strict par les moniteurs actifs lorsque les demandes d'envoi de paquets actifs augmentent.

7.5 Conclusion

Le chapitre a traité l'aspect sécurité dans le domaine des réseaux actifs. Il a établi quelques règles rudimentaires à la conception d'un réseau sécurisé. A cette fin, nous avons défini des mécanismes globaux sur les phases principales du cycle de vie d'un programme actif. Ces phases sont principalement la phase de l'introduction dans le réseau et l'acheminement de codes jusqu'aux nœuds et la phase d'exécution du programme par les nœuds.

Les mécanismes de sécurité de la publication du code et son déploiement proposent la mise en place d'une architecture d'ensemble et complète mettant en avant trois autorités complémentaires. Le premier composant est une autorité de certificats (CAAN) qui conserve, génère et authentifie les certificats des développeurs et des utilisateurs. Le second composant est le serveur de code (CISS) qui est la pièce maîtresse de notre architecture car il est présent lors des différentes phases et interagit avec les différents acteurs. Lors de la publication, il sauvegarde et attribue un identifiant et une clef aux codes après avoir vérifié les droits des développeurs et la sûreté de ces codes. Et pendant la phase du déploiement, il génère une clef symétrique pour l'autorisation des utilisateurs après avoir vérifié leurs droits. Dans ce cas, les nœuds n'ont à effectuer qu'une simple reconstitution des clefs symétriques à partir des clefs de codes et vérifier leur concordance sans aucun surplus de traitement ou de communication. Ces mécanismes reposent sur l'hypothèse que toutes les entités possèdent des clefs mêmes les routeurs et les serveurs de codes. Quant à la dernière entité, elle est le serveur Web de publication.

Quant aux mécanismes de sécurité lors de l'exécution des programmes, ils veillent au respect de la consommation des ressources des nœuds et du réseau. Ces mécanismes développés sont liés à la nature des règles actives.

Dans le chapitre suivant nous présenterons de nouvelles applications, en utilisant les règles actives implémentées dans le contexte de l'infrastructure ARFANet.

Chapitre 8

Des applications pour les Réseaux Actifs

8.1 Introduction

La recherche de solutions de substitution à base de réseaux actifs aux problèmes existants met en exergue l'intérêt de ce type de réseaux dans le domaine des télécommunications. Les problèmes sont au niveau des différents plans (signalisation, supervision, données), tels que la congestion, le multicast, l'adaptation des flux et la qualité de service. Cette recherche est, actuellement, indispensable aux réseaux actifs pour leur émergence dans les réseaux actuels.

Les applications qui ont connu un essor considérable ces dernières années sont les applications Pair à Pair ou P2P (Peer to Peer), les raccordements massifs entre équipements mobiles (réseau ad hoc) et à l'Internet et les jeux en réseaux.

L'objectif de ce chapitre est de proposer de nouvelles applications actives et de les modéliser en utilisant les règles actives.

La première section est dédiée à la résolution du problème de la surcharge d'un serveur Web commercial en période de pointe. La section 8.3 présente la façon d'améliorer la détection des pairs voisins d'un réseau P2P et la recherche de fichiers en utilisant les routeurs actifs tout en évitant d'inonder le réseau avec des messages multicast ou ping. La conclusion de ce chapitre est présentée dans la section 8.4.

8.2 Soulagement d'un Serveur Web en cas de surcharge

L'événement phare qui a caractérisé ce début d'année est la déclaration des Impôts en ligne. Mais cette nouveauté, qui se devait d'être une amélioration et une automatisation de cette tâche si onéreuse pour le contribuable français, a vite tourné au fiasco. A l'approche des dates limites fixées à chaque fois par le *fisc*, le nombre de connexions vers le site des impôts devenait faramineux, ce qui a conduit à une surcharge du serveur provoquant ainsi un effondrement brutal du serveur. Le serveur ne pouvait répondre à aucune demande, sans distinction, mêmes celles qui sont arrivées à un point avancé dans le processus de déclaration.

Cet incident prouve qu'on est loin d'une solution efficace, même si on s'attendait à ce que les services des impôts prévoient une telle situation.

Ceci n'est qu'un exemple parmi des millions qui se passent chaque jour pour les sites commerciaux durant les périodes de pointes. Nombreuses sont les agences de voyages qui proposent de temps à autre des promotions à travers leur portail web. Le pic des demandes de connexions peut

survenir à tout moment et de manière instantanée. Le service est alors interrompu pour tous les types d'utilisateurs.

Le serveur E-commerce doit garantir une qualité minimale de service qu'il se trouve ou non dans une période de pointe. Un service différencié est une valeur ajoutée dans ce cas. Le serveur E-commerce peut, par exemple, donner à chaque client une priorité selon son progrès dans les étapes de navigation.

Cette priorité peut être définie (de manière croissante) comme suit :
- nouveau client (faible priorité),
- client en navigation,
- client qui a au moins choisi un produit,
- client qui a commencé une transaction financière,
- client attendant une confirmation (haute priorité).

Au moment de la surcharge, le serveur doit rejeter les nouveaux clients, donc les nouvelles connexions. Il peut également rejeter ceux en cours de navigation, n'acceptant que ceux qui attendent une confirmation. Cependant cette amélioration ne suffit pas car les demandes de connexions TCP peuvent saturer la file d'entrée du serveur (le serveur ouvre une connexion TCP juste à la réception d'un paquet *syn*) et former un goulot d'étranglement autour du serveur.

Un service actif pour améliorer le E-commerce en période de surcharge consiste à déléguer aux nœuds actifs avoisinant le serveur Web les tâches de refus et d'acceptation des requêtes selon le type de l'utilisateur pour alléger le serveur. Ce service est donc un genre de *filtre* qui s'adapte à la densité du trafic. En utilisant les informations embarquées dans le flux HTTP, en particulier la *session*, le serveur et les routeurs peuvent connaître exactement à quelle catégorie le client appartient sans avoir au préalable répertorié les clients.

Par l'utilisation des nœuds actifs ARFANet en collaboration avec le serveur E-commerce se trouvant dans son voisinage, le gestionnaire du serveur lui-même peut créer l'application à base règles actives au niveau *donnée utilisateur* (le plus bas niveau, voir Chapitre 7). Le gestionnaire envoie ensuite au CISS le service d'application correspondant.

Lorsque le serveur Web détecte qu'il n'arrive pas à satisfaire les clients, parce qu'un goulot se forme progressivement ou parce que le nombre de demandes de retransmissions (NACK) augmente de manière vertigineuse, il envoie au CISS le service d'application *rush-hour filter*. Cet envoi est fait seulement si le service d'application n'a pas été envoyé avant, sinon le gestionnaire du service Web envoie ses paquets actifs aux nœuds voisins avec la référence de ce service. De même pour les nœuds, à la réception des paquets actifs, ils doivent récupérer le code sur le serveur CISS s'ils ne le possèdent pas.

Les règles composant l'application définissent le comportement du nœud vis-à-vis de chaque classe et le degré de surcharge du serveur. L'application doit être aussi petite que possible avec un temps d'exécution minimum et peu d'information à stocker sur le nœud. Le degré de la surcharge indiquera le seuil de filtrage des clients. Nous définissons cinq niveaux reflétant l'état du serveur et du réseau.
- 0 : admettre tous les clients, même les nouveaux,
- 1 : admettre uniquement les clients en cours de navigation,
- 2 : admettre uniquement les clients qui ont au moins choisi un article,
- 3 : admettre uniquement les clients qui ont commencé une transaction financière,
- 4 : admettre uniquement les clients qui attendent une confirmation. C'est l'étape finale (haute priorité),

Le degré de surcharge du serveur associé à la catégorie du client détermine le traitement du

nœud. Si le degré de surcharge est supérieur à la catégorie, la connexion sera rejetée. La catégorie du client peut être extraite du flux HTTP, ainsi seul le degré de la surcharge est gardé dans le cache du nœud. La mise à jour du degré est faite par le serveur Web au moment où son état change.

Le service d'application *rush-hour filter* Dans la table 8.1 nous présentons le module du service d'application *rush-hour filter* :

Module M1 : E-commerce rush-hour filter
Sémantiques d'Exécution Consommation d'événement = vérifié Mode de couplage E-C = immédiat Mode de couplage C-A = immédiat Priorités des règles = R5 ; R4 ; R1 ; R2 ; R3 Exécution en cascade = itérative
Les règles actives R1 : On *Arrivée du Degré* If *true* Then *Mise à jour de la valeur du Degré local* R2 : On *Arrivée d'un Client* If *catégorie du client* $<$ *Degré* Then *Rejeter le client* R3 : On *Arrivée d'un Client* If *catégorie du client* $\geqslant$ *Degré* Then *Renvoyer la requête au serveur* R4 : On *Fin* If *true* Then *Libérer la mémoire (détruire le Degré)* R5 : On *Début* If *true* Then *Commencer l'application (créer un Degré)*

TAB. 8.1 – Le module E-commerce *rush-hour filter*

8.3 Application P2P

Le principe de ce genre d'architecture réseau est de partager des ressources (principalement des fichiers) entre simples utilisateurs sans passer par un serveur dédié. Cette topologie est une extension du modèle Client-Serveur. Elle consiste en un ensemble de nœuds, où chaque nœud peut agir à la fois comme serveur et comme client. Les systèmes P2P sont caractérisés par le partage des ressources (espace de stockage, cycles processeur, bande passante), et par une décentralisation et une auto-organisation (autonomie, recouvrement du voisinage, propagation des requêtes).

Il existe deux types de systèmes P2P. Les systèmes P2P purs, comme GnuTella [56] et Freenet [18], qui n'admettent aucun nœud centralisé. Chaque nœud communique et découvre lui-même les autres nœuds. L'existence d'un ou de plusieurs serveurs qui gardent trace des autres participants rentre dans le cadre du deuxième type de systèmes P2P comme KaZaa [53] et Napster [72]. La communication entre nœuds est directe, i.e. ne passe pas par un serveur. Néanmoins, même dans les P2P purs comme GnuTella, il est question d'initialisation de la table de nœuds connus à partir d'un serveur Web hors protocole (par exemple, www.gnutella.com) et à compléter lors de la connexion.

La recherche des nœuds voisins et la recherche de fichiers à base de mots clefs dans les systèmes P2P purs se basent sur l'inondation du réseau (par ping généralement). Lorsqu'il y a un nombre

important de nœuds P2P, le nombre de requêtes augmente de manière colossale, pouvant causer ainsi des perturbations dans tout le réseau.

L'utilisation des nœuds actifs pour l'agrégation de requêtes de recouvrement et les requêtes de recherche peut être appréciable. Un nouveau client qui se connecte, au lieu d'inonder le réseau à la recherche des autres nœuds voisins en ligne, envoie une *annonce de présence* au nœud actif auquel il est raccordé. Si le nœud actif détient une information qu'un autre client, rattaché aussi à ce même nœud, a été connecté, il envoie aux deux clients une note de leur présence. Dans le cas contraire, il remonte cette information au nœud actif voisin. Ce processus est répété jusqu'à ce qu'un nœud contient la notification de la présence d'un client rattaché. Avec cette technique où les routeurs actifs prennent part au repérage des voisins P2P, il est tout à fait possible d'atteindre des clients non détectables avec une simple inondation. Le nœud actif envoie aux autres nœuds actifs cette information et de même, chaque nœud actif la retransmet aux clients rattachés afin de mettre à jour leur table des voisins.

En ce qui concerne la recherche à mot clef, il est possible de profiter des réseaux actifs pour ne pas aussi inonder le réseau avec des requêtes de fichiers. Le nœud peut enclencher un processus de recherche sans pour autant inonder le réseau, et cela par transmission de la requête de voisins à voisins par des connexions point-à-point. Dans le système GnuTella, chaque nœud propage la requête à plusieurs voisins (en général 4) et limite la propagation à un TTL fixe (en général 7) avec détection et élimination des cycles.

Les inconvénients de cette méthode sont la difficulté de son application à grande échelle (sature à plus de 10000 utilisateurs), l'inondation par requêtes et le TTL qui peut être atteint sans que les requêtes aboutissent à des résultats. De plus, cette technique est sa lenteur et les réseaux actifs peuvent améliorer la recherche à mot clef en donnant des résultats plus rapides et plus exhaustifs en atteignant des pairs éloignés.

Il est évident que les réseaux actifs transforment un P2P pur en un P2P partiellement hybride et rendant les routeurs des serveurs stables dans ce type de P2P. Dans une logique pure d'un P2P, la notion de voisin n'est pas réelle car il n'y a aucune vue globale de la topologie. Deux voisins dans cette logique peuvent être aux deux extrémités du réseau Internet. Par contre, avec les réseaux actifs deux clients rattachés au même routeur sont forcément voisins au sens physique. L'avantage majeur de l'utilisation des réseaux actifs est la réduction du trafic, sachant que le trafic entre pairs est difficile à mesurer et souvent sous-estimé. Ceci est la cause de la multitude des systèmes P2P utilisant plusieurs ports [88].

Le service d'application *P2P covering* La table 8.2 montre le module ECA du service d'application du recouvrement intelligent des utilisateurs P2P. Il faut noter que dans la règle R4 deux événements sont composés, l'événement *Déconnexion* et *TimeOut*.

Le service d'application *P2P research* Au lieu que le client envoie plusieurs requêtes à différents voisins, il envoie une requête au routeur qui la retransmet aux autres clients sous sa couverture. Si l'un des clients détient l'information recherchée, il envoie au client demandeur une notification.

Le rôle du routeur consiste à éviter une augmentation du trafic. Il est à noter que le routeur peut remonter la requête à d'autres routeurs pour toucher plus de clients possibles, et peut être tous. Ceci constituera un graphe de multicast où chaque routeur est le nœud racine d'un arbre de multicast formé par les pairs qu'ils lui sont rattachés. L'envoi des requêtes est groupé entre nœuds actifs tandis que chaque nœud les diffuse aux clients sous sa couverture.

Module M1 : P2P covering
Sémantiques d'Exécution Consommation d'événement = vérifié Mode de couplage E-C = immédiat Mode de couplage C-A = immédiat Priorités des règles = R5 ; R4 ; R3 ; R2 ; R1 Exécution en cascade = itérative
Les règles actives R1 : On *Arrivée d'une Nouvelle connexion* If *true* Then *Enregistrer le client (Adresse IP, capacité du lien)* R2 : On *Arrivée d'une Nouvelle connexion* If *existe clients rattachés* Then *Envoyer à tous les clients* R3 : On *Déconnexion* Or *TimeOut* If *true* Then *Détruire le l'enregistrement client*

TAB. 8.2 – Le module *P2P covering*

Module M1 : P2P research
Sémantiques d'Exécution Consommation d'événement = vérifié Mode de couplage E-C = immédiat Mode de couplage C-A = immédiat Priorités des règles = R2 ; R1 Exécution en cascade = itérative
Les règles actives R1 : On *Arrivée d'une Requête* If *existe clients rattachés* Then *Décrémenter TTL ;* *Envoyer à un autre routeur actif* R2 : On *Arrivée d'une Requête* If *true* Then *Envoyer au groupe multicast client la requête*

TAB. 8.3 – Le module *P2P research*

8.4 Conclusion

Nous avons pu voir à travers ces deux exemples qu'il est extrêmement facile d'exprimer et de formaliser des applications actives variées par les règles ECA. Ces dernières explicitent la séparation d'une application en différentes règles déclenchables de manière autonome ou bien des modules fondamentalement distincts dans leur déploiement et chargement comme le montrent la section 8.3 avec les deux modules pour le traitement respectif des requêtes de recouvrement et des requêtes de recherche dans un réseau P2P. Par conséquence, il devient aisé et presque trivial de modéliser

différents genres d'applications. De plus, leur formulation sous forme ECA facilite la maintenance future et leur extension.

L'expression des actions et des conditions est formelle, mais il appartient au développeur d'utiliser d'autres types de langages. Des langages restreints aux réseaux actifs, comme des langages généralistes, peuvent être employés pour le codage des actions et des conditions.

Chapitre 9

Conclusion

Nous avons considéré la réalisation d'un réseau actif comme un projet global qui est composé de plusieurs sous-projets. De chaque sous-projet découlait une question principale : Quel langage actif utiliser pour les applications actives ? Quelle politique de distribution de code employer pour assurer une cohérence dans ce système distribué constitué par les routeurs ? Quel plan de sécurité envisager pour garantir la continuité des services et la confidentialité des données ? Et finalement quel niveau de dégradation des performances admettre ?

Nous avons remarqué que ces sous projets ne pouvaient être réalisés séparément, dans des laps de temps distincts et sans aucune interaction les uns avec les autres. Nous avons constaté, par exemple, que le plan de sécurité et la politique de distribution étaient étroitement liés. La définition d'une méthode de sécurité exigeait de connaître les mécanismes de distribution du code et les acteurs externes impliqués. D'autre part, les mécanismes de déploiement de codes devaient prendre en compte la technique employée pour autoriser l'exécution d'un code.

Ce que nous avons souhaité à travers cette thèse est de mettre en évidence cette interaction entre les différentes parties et l'exploiter dès la phase de la conception du réseau actif.

Dans l'état de l'art, nous avons mentionné l'existence de plusieurs groupes de travail séparés sur l'architecture d'un nœud, la composition des services et la sécurité. Nous déplorons, cependant, l'inexistence d'un groupe de travail qui engloberait tous les aspects précédemment cités pour définir une architecture conforme à tous les besoins et à toutes les caractéristiques d'un réseau actif. Ce groupe de travail devrait permettre de définir des méthodologies de conception et des préceptes normatifs à prendre en compte à la réalisation de chaque étape, et énumérer les interactions possibles entre ces étapes.

La concrétisation des réseaux actifs doit, cependant, passer par la remise en cause de l'architecture et des mécanismes actuels, qui ne répondent pas pleinement aux exigences et caractéristiques d'une architecture émergeante. Chaque architecture active doit au moins comporter les caractéristiques suivantes : composabilité, modularité, extensibilité, mobilité, sécurité et enfin performance.

Nous avons, par ailleurs, mis l'accent sur ces points que nous avons voulu reproduire avec les règles actives sur notre propre plate-forme ARFANet. Ainsi, par l'introduction des règles actives ECA dans les réseaux actifs, notre but était de rendre le réseau plus *réactif*. En réagissant aux événements internes et externes, les nœuds du réseau seront capables d'exécuter des applications sensibles à l'événement. Il en découle une simplicité de formalisation et de modélisation des protocoles et des programmes comme le montre le Chapitre 8.

D'ailleurs, la décomposition de l'application active en plusieurs structures bien formées sous forme d'Événement-Condition-Action nous permet d'avoir des applications bien formulées qu'on

peut facilement introduire dans le réseau. En effet, cette décomposition nous permet d'une part, de savoir exactement ce que doit contenir chaque couche de l'architecture active. Et d'autre part, la distinction entre les parties principales de l'application active facilite la développement de services composites et le contrôle de la sécurité en opposition à celle d'un programme actif compact.

Nous avons développé des méthodes de distribution et de sécurisation de code complètes qui tiennent compte de toutes les phases et de toutes les entités concernées à savoir le développeur, les utilisateurs et les routeurs. Cela a nécessité la création d'une autorité de certification (CAAN), dont le rôle est de classifier et d'authentifier chaque entité, et d'un serveur de codes (CISS) qui se place au cœur de cette architecture, dont le rôle est d'identifier, de sauvegarder et de vérifier le code et l'identité du développeur. Dans la phase de déploiement du code, ce serveur a un rôle supplémentaire. Il doit, en outre, vérifier les permissions d'exécution des codes par les utilisateurs en leur attribuant des jetons temporaires qui permettent aux nœuds, aux moyens de simples opérations de comparaison, de les authentifier et de les autoriser à exécuter le programme. Et enfin, nous avons créé un serveur web pour reproduire sur des pages HTML le contenu de la base de codes des serveurs CISS.

Par ailleurs, nous avons réalisé une étude de performances poussée et exhaustive. Pour arriver à cette fin, nous avons modélisé l'architecture d'un nœud actif à bases de files d'attentes en ne gardant que les couches essentielles. L'architecture à trois couches que nous avons développée dans ARFANet est assez proche de celle retenue dans l'architecture standard. Pour cette raison et des choix de modélisation que nous avons faits, ceci place notre étude à un niveau assez général pour représenter un nœud actif quelconque. Notre étude pointe la faisabilité de la gestion active dans un réseau et son influence sur les flux dits passifs qui n'exigent aucun traitement personnalisé. Pour comprendre le comportement du nœud avec les différents types de flux, nous l'avions observé dans plusieurs situations : différentes charges, différentes tailles des files, différentes proportions des flux et finalement différents taux de générations de paquets actifs après exécutions des applications actives. Les résultats obtenus nous poussent à conclure que les paquets actifs n'ont pas une grande influence sur la perte et la latence des autres paquets dû au fait de l'acheminement interne des paquets actifs. Toute la saturation étant concentrée dans la file d'entrée, ce qui laisse les paquets générés peu encombrants pour les paquets standards. Nous avons aussi fait des tests réels pour valider nos résultats analytiques, ainsi que des simulations.

Comme nous l'avons souligné dans le Chapitre 5, nous avons trouvé que PEPA est un formalisme de modélisation pratique pour réaliser une étude des performances. Une extension future de ce travail consiste à considérer la configuration d'un réseau de nœuds. D'ailleurs, dans notre recensement des études de performances dans la littérature, plusieurs auteurs déplorent l'inexistence d'études d'évaluation des performances entre plusieurs nœuds, les seuls études se focalisent sur le débit, les pertes et la latence observés pour un nœud. Pour une telle étude, un formalisme comme PEPA nets serait plus approprié. PEPA nets [37] est une combinaison de PEPA et des réseaux de Petri stochastiques colorés où chaque jeton est une composante PEPA. Ce formalisme combiné modélise de manière naturelle des applications telles que les systèmes de codes mobiles où les composantes PEPA sont utilisées pour représenter le code qui se déplace entre les réseaux hôtes (places dans PEPA nets). L'utilisation de PEPA nets serait une suite logique après l'étude effectuée avec PEPA.

Par ailleurs dans le futur, nous projetons, au niveau des couches de l'architecture d'un nœud, d'opérer des optimisations en implémentant la machine virtuelle dans le langage C. Actuellement, toutes les couches sont écrites en Java, ce qui donne à la machine virtuelle une certaine compatibilité et portabilité, mais la désavantage du point de vue performance. De plus, les codes des actions et des conditions sont des classes Java. Puisque la formalisation des règles actives dans ARFANet doit être

indépendante et accepter plusieurs langages, nous proposons d'intégrer plusieurs environnements d'exécution des actions et des conditions que le développeur doit spécifier dans l'en-tête du module du service d'application.

En ce qui concerne la distribution de code, nous sommes tentés par des tests à grande échelle, sur des réseaux d'envergure, des techniques que nous avons mises au point. De plus, nous retenons la nécessité de développer des algorithmes efficaces de mises à jours des bases de codes sur des serveurs distants. Notre méthode mixte, qui mélange une migration de proche en proche et l'utilisation d'un serveur de code n'a pas donné les performances attendues sur le réseau de test de petite taille. Cela dit, nous pensons qu'elle aura de meilleurs résultats lorsqu'elle sera appliquée sur des réseaux de grandes tailles.

Dans le domaine de la sécurité, nous ne pouvons pas prétendre que notre tâche est achevée car il reste beaucoup à faire surtout dans le cadre du contrôle en cours d'exécution des actions et des conditions. Il faut dire que nous avons profité de la séparabilité des codes qu'offrent les règles actives pour jeter tout le poids sur la sécurité dite "de pré-exécution", en d'autres termes toutes les phases de la distribution du code. Étant donnée la capacité du serveur de codes à procéder à de lourds traitements de vérification et d'analyse des codes, nous sommes tentés pour creuser plus cet aspect en combinant plusieurs techniques récentes [16] [23] et les adapter dans le contexte des réseaux actifs.

Annexe A

A PEPA

A.1 La syntaxe de PEPA

Un modèle PEPA est décrit comme l'interaction d'un ensemble de *composantes*. Chaque composante peut exécuter un ensemble d'actions : une action $a \in \mathcal{A}ct$ est décrite comme une paire (α, r), où $\alpha \in \mathcal{A}$ est le *type* de l'action et $r \in \mathbb{R}^+$ est le paramètre de la distribution exponentielle négative régissant sa durée.

PEPA dispose d'un ensemble réduit, mais suffisamment puissant, d'opérateurs pour modéliser un comportement complexe à partir de comportements simples. Ces opérateurs sont ceux de l'algèbre des processus classiques : le préfixe, le choix, la composition parallèle et l'abstraction. Dans ce qui suit, nous expliquons brièvement chacun de ces opérateurs. La sémantique opérationnelle formelle de PEPA est décrite dans l'Annexe A.4.

Préfixe : Une composante peut avoir un comportement purement séquentiel, entreprenant à plusieurs reprises une action après une autre, retournant éventuellement par la suite au début de son comportement. Dans ce cas, l'opérateur préfixe, noté ".", est utilisé pour indiquer la première action à effectuer. Par exemple, $(\alpha, r).p$ effectuera une activité de type α avec une durée moyenne de $1/r$, puis se comportera comme la composante P. Dans certains cas, le taux d'une action est en dehors du contrôle d'une composante. Celle-ci joue alors un rôle passif et le taux de l'activité pour cette composante est dénoté par le symbole $\top$ (appelé " top "). Une telle action est effectuée en coopération avec une autre composante qui elle contrôle le taux.

Choix : Un choix entre deux comportements possibles est représenté comme la somme des différentes possibilités, par exemple, $(\alpha, r).P + (\beta, s).Q$. La condition de compétition (*race condition)* est supposée gouverner le comportement des actions simultanément déclenchables. Ainsi cet opérateur représente un choix préemtif avec re-échantillonage. La nature continue des distributions de probabilités garantit que les actions ne peuvent pas se produire simultanément. Quand une action a plus d'un résultat possible, elle est représentée par un choix de différentes actions, une pour chaque résultat possible. Les taux de ces actions sont choisis de manière à refléter leurs probabilités relatives.

Composition parallèle : La composition parallèle est utilisée pour représenter les cas où deux composantes du système *coopèrent* pour réaliser une certaine action. Par exemple, $P \underset{L}{\bowtie} Q$ se compose de deux composantes P et Q qui doivent coopérer pour réaliser les actions qui sont dans l'*ensemble de coopération* L. Les actions dont le type n'est pas dans cet ensemble peuvent être effectuées de manière indépendante et concurrente par les deux composantes. Les actions dans L exigent la participation simultanée des deux composantes. L'action résultante ou *l'action partagée* aura le même type que les deux actions de la contribution et un taux reflétant le taux de l'action de la composante la plus lente des deux. Notez que cela signifie que le taux d'une action passive deviendra le taux de l'action avec laquelle elle coopère.

Abstraction : Il est souvent commode de cacher quelques actions, les rendant privées à la composante ou aux composantes impliquées. La durée des actions est inchangée, mais leur type devient

caché, apparaissant à la place comme le type inconnu τ. Les composantes ne peuvent pas synchroniser ou coopérer sur τ. L'utilisation de l'opérateur d'abstraction a deux implications. Premièrement, elle permet d'éviter qu'une composante ajoutée ultérieurement au modèle puisse interagir ou interférer avec une action privée. Deuxièmement, les actions privées ne peuvent contribuer au calcul des mesures, ce qui, par conséquent, suggère des simplifications du modèle.

En utilisant des constantes pour nommer les composantes et des définitions récursives nous pouvons décrire des composantes avec des comportements infinis sans utiliser un opérateur explicite de récursion.

Exemple : On considère un simple système composé d'un serveur et d'une file d'attente de capacité finie N. Le nombre total de clients dans le système ne peut pas dépasser N. On suppose que les arrivées et les services sont exponentiellement distribués de paramètres λ et μ, respectivement.

Le système peut être représenté comme l'interaction de deux composantes *Serveur* et *Buffer*.

$$Serveur \stackrel{\text{def}}{=} (service, \mu).Serveur$$

$$\begin{aligned} Buffer_0 &\stackrel{\text{def}}{=} (arrive, \lambda).Buffer_1 \\ Buffer_i &\stackrel{\text{def}}{=} (arrive, \lambda).Buffer_{i+1} + (service, \top).Buffer_{i-1} \quad 0 < i < N \\ Buffer_N &\stackrel{\text{def}}{=} (service, \top).Buffer_{N-1} \end{aligned}$$

La composante du modèle *Système* est :

$$\mathbf{Syst\grave{e}me_0} \stackrel{\text{def}}{=} Serveur \underset{\{service\}}{\bowtie} Buffer_0$$

Remarque : Contrairement à des formalismes de l'algèbre de processus tel que μCRL, PEPA n'est pas un formalisme paramétré. Tous les paramètres du modèle doivent être connus au départ. Ainsi dans l'exemple ci-dessus, la valeur de N doit être fixée lors de la description du modèle. Cependant, une composante peut avoir des activités dont le taux est défini comme une fonction de l'état d'une ou de plusieurs autres composantes du modèle [47].

A.2 Le processus de Markov associé

Dans un modèle PEPA, si une composante P exécute une activité (α, r) puis se comporte comme P', alors celle-ci est appelée *composante dérivée* de P.

A partir de n'importe quelle composante PEPA P, on peut construire de manière récursive l'ensemble des dérivées de P, noté $ds(P)$. Cet ensemble décrit l'ensemble des dérivées (comportements) possible pouvant résulter de P.

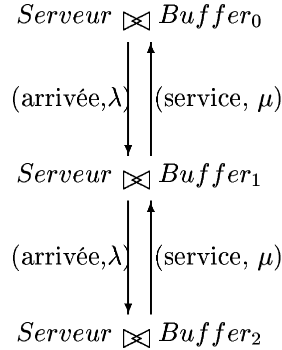
A partir de l'ensemble des dérivées de la composante du modèle $ds(C)$, on peut construire le *graphe de dérivation* qui est un multigraphe dirigé dont l'ensemble des nœuds est $ds(C)$ et dont les arcs sont les transitions possibles entre ces nœuds.

Le processus de Markov associé à un modèle PEPA est un processus de Markov à temps continu. La génération de ce processus est basée sur le graphe de dérivation du model. Un état est associé à chaque nœud du graphe et les transitions entre les états sont obtenues à partir de celles du graphe. Le taux de la transition entre deux états de la chaîne de Markov est la somme des taux de l'activité sur les arcs entre les deux nœuds correspondant du graphe.

Exemple : Considérons le modèle PEPA décrit précédemment (section A.1) et dont la composante est :

$$\mathbf{Syst\grave{e}me_0} \stackrel{\text{def}}{=} Serveur \underset{\{service\}}{\bowtie} Buffer_0$$

Si on suppose que $N = 2$, l'ensemble de dérivation de la composante du modèle est $ds(\mathbf{Système_0}) = \{Serveur \underset{\{service\}}{\bowtie} Buffer_j, 0 \leq j \leq 2\}$ et le graphe de dérivation est le suivant :



La chaîne de Markov sous-jacente au modèle (figure 1) est générée à partir du graphe de dérivation. Il est clair que $j = 0, \ldots, 2$ représente évidemment le nombre de clients dans le système.

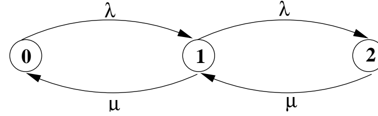


FIG. 1 – Chaîne de Markov

A.3 Résolution numérique

Un outil expérimental appelé PEPA Workbench [36] a été développé pour le formalisme PEPA. A partir de la description d'un modèle PEPA, cet outil fournit le comportement stationnaire du système modélisé en calculant la distribution stationnaire (vecteur de probabilités). Plusieurs techniques de résolution ont été intégrées à cet outil, parmi lesquelles on trouve les deux méthodes itératives *sor* (successive over-relaxation) et *linear biconjugate gradient*.

Par exemple, la résolution de la chaîne de Markov (figure1) associée à notre modèle PEPA consiste à résoudre l'équation suivante :

$$\Pi \times Q = 0$$

où $\Pi = (\pi_0, \pi_1, \pi_2)$ représente la distribution stationnaire que l'on veut calculer et Q est le générateur infinitésimal correspondant à la chaîne de Markov.

$$Q = \begin{pmatrix} -\lambda & \lambda & 0 \\ \mu & -(\lambda + \mu) & \lambda \\ 0 & \mu & -\mu \end{pmatrix}$$

Une fois obtenu, Π représente le facteur clé nécessaire au calcul des mesures de performance recherchées par l'utilisateur. Le calcul de ces mesures peut être effectué grâce à l'outil PEPA *State finder* qui fait partie intégrante du PEPA Workbench. Une fois le modèle résolu, PEPA *State finder* permet d'identifier les états qui satisfont certaines propriétés et à partir de ça l'utilisateur peut calculer les mesures de performance qui l'intéressent.

Dans note exemple, on peut être intéressé par le calcul du taux de perte des clients qui arrivent dans le système. Ce taux est donné par :

$$Taux_perte = \lambda \times \pi_2$$

π_2 étant la probabilité d'être dans l'état 2 de la chaîne de Markov, c'est-à-dire la probabilité que la file soit saturée.

A.4 Sémantique opérationelle structurée de PEPA

Les règles sémantiques, dans le style opérationnel structuré, sont décrites dans la figure 2 [46]. Les règles sont lues comme suit : si la ou les transitions au dessus de la ligne d'inférence peuvent être inférées, alors la transition sous la ligne peut être inférée aussi. La notation $r_\alpha(E)$ définie le taux apparent de α dans E.

Préfixe

$$\frac{}{(\alpha, r).E \xrightarrow{(\alpha,r)} E}$$

Coopération

$$\frac{E \xrightarrow{(\alpha,r)} E'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,r)} E' \underset{L}{\bowtie} F} \ (\alpha \notin L) \qquad \frac{F \xrightarrow{(\alpha,r)} F'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,r)} E \underset{L}{\bowtie} F'} \ (\alpha \notin L)$$

$$\frac{E \xrightarrow{(\alpha,r_1)} E' \quad F \xrightarrow{(\alpha,r_2)} F'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,R)} E' \underset{L}{\bowtie} F'} \ (\alpha \in L) \qquad \text{où } R = \frac{r_1}{r_\alpha(E)} \frac{r_2}{r_\alpha(F)} \min(r_\alpha(E), r_\alpha(F))$$

Choix

$$\frac{E \xrightarrow{(\alpha,r)} E'}{E + F \xrightarrow{(\alpha,r)} E'} \qquad \frac{F \xrightarrow{(\alpha,r)} F'}{E + F \xrightarrow{(\alpha,r)} F'}$$

Abstraction

$$\frac{E \xrightarrow{(\alpha,r)} E'}{E/L \xrightarrow{(\alpha,r)} E'/L} \ (\alpha \notin L) \qquad \frac{E \xrightarrow{(\alpha,r)} E'}{E/L \xrightarrow{(\tau,r)} E'/L} \ (\alpha \in L)$$

Constante

$$\frac{E \xrightarrow{(\alpha,r)} E'}{A \xrightarrow{(\alpha,r)} E'} \ (A \stackrel{def}{=} E)$$

FIG. 2 – La sémantique opérationnelle de PEPA

Annexe B

B L'application ARFANet

Dans cette annexe, nous présentons les programmes essentiels de notre implémentation.

B.1 Le service d'application

La classe abstraite d'un service d'application résume les méthodes que peuvent invoquer les autres classes des règles. Le rôle de ces méthodes est de permettre aux règles d'accéder et de changer les valeurs des attributs et les données vihiculées et de demander de renvoyer les paquets.

```
public abstract class AbstractAppliServ {

    public int getEveAttrVal (int eventId, int attrId) {return -1; }
    public void setEveAttrVal (int eventId, int attrId, int val) {}
    public Data getData (int eventId) {return null;}
    public void setData (int eventId, int val) {}
    public void forward (Packet p) {}
 }
```

Nous explicitons dans ce qui suit les classes liées à un service d'application.

B.1.1 La règle

La règle comporte son propre identificateur, le service d'application dont elle fait partie, le moniteur actif et sa propriété. Elle contient également l'événement, la condition et l'action sous forme binaire et les classes correspondantes après instantiation.

```
public class Rule {

    int asId, amId, ruleId;
           // l'identifiant de la règle, du service d'application et du moniteur actif

    public Event event;
    public Condition condition;
    public Action action;
    byte [] binEvent, binCondition, binAction;
           // le flux binaire des classes reçu des paquets de code

    public int priority;
           // cette  information dépend du champ rulePriority du moniteur

    public void instatiation  (){
```

```
    }     // instantiation des classes de codes binaires
}
```

B.1.2 L'événement

La classe événement est constituée de l'identificateur de l'événement, son état courant, son type (interne, externe, système et composé), le nombre et enfin la liste d'attributs où chaque entrée correspond à la valeur de l'attribut et la valeur de déclenchement. Ces variables sont suivies d'un certain nombre de méthodes qui permettent d'accéder et de mettre à jour les valeurs des attributs.

```
public class Event {

int eventIdent; // identificateur de l'événement
int eventState;  // état de l'événement
int eventType;  // type de l'événement qui correspond aux valeurs suivantes :
// 0 : interne,
// 1 : externe,
// 2 : système.
// 3 : composé.
int att_nb; // nombre d'attributs
int  attrList [][]; // liste d'attributs

public long currentTime () {
return 0;
}

public void setState (int state) {
this.eventState = state;
}

public int getAttrVal (int attrId) {
return -1;  // donne la valeur de l'attribut
}

public int getAttrTrig (int attrId) {
return -1; // donne la valeur de déclenchement de l'événement par cet attribut
}

public void setAttrVal (int attrId, int valattr) {
}  // met à jour l'attribut

}
```

B.1.3 La condition

La classe abstraite *Condition* que le développeur doit redéfinir, et particulièrment sa méthode *CondValue*, doit donner l'expression de la condition.

```
public abstract class  Condition {
    public abstract boolean CondValue(int eventId, AbstractAppliServ as);
}
```

B.1.4 L'action

Comme la condition, la méthode *ActionToLaunch* de la classe abstraite *Action* doit être redéfinie et donner le comportement voulu de l'action.

```
abstract public class Action {
    int nb_action;
    public abstract  void ActionToLaunch (int idEvt, AbstractAppliServ as);
}
```

B.2 Le moniteur actif

Le moniteur actif doit gérer une classe de services d'application. Il est responsable de la gestion et de la composition des événements, de leur ordonnancement, du déclenchement des règles concernées, de leur ordonnancement et en fin de leur exécution.

```
public class  ComposedMonitor extends AbstractActivMonitor  implements Runnable {

int AMident;    // l'identifiant du moniteur

public  AbstractVirtMachin getVirtualMachine (){
return VirtualMachine;    // retourne l'objet de la classe machine virtuelle
}

public class EventMonitoring  implements Runnable {  // gestion des événements
}

public class EventScheduling  { // ordonnancement des événement
}

public class RuleScheduling  implements Runnable { // ordonnancement des règles
}

public class RuleHandling  implements Runnable {  // exécution des règles

boolean condiontionEvaluation (Rule r){  // évaluation de la condition
}

boolean actionLaunching (Rule r) {   // lancement de l'action
}
}
}
```

B.3 La machine virtuelle

La machine virtuelle joue le rôle d'un NodeOS en routant les paquets passifs vers leur destination et les paquets actifs vers les moniteurs.

```
public class VirtualMachine  extends AbstractVirtMachin {

int [][] RoutingTable; // correspond à la table de routage
int CISS [];             // est la liste des adresses des CISS attachés

Queue dElementQueue ;  // file des d-paquets
Queue sElementQueue ;  // file des s-paquets
Queue mElementQueue ;  // file des m-paquets
Queue req4CodeQueue ;  // file des demandes de code

Queue OutpuQue;    // file de sortie
Queue InputQue ;  // file d'entrée

public void forward (Packet p) {
// fonction de routage appelée par la règle via le service d'application et le moniteur
}

public  void  setData (int ad, int as, int am, Data d) {
        // mise à jour des données transportées par les paquets
}

public  Data  getData (int ad, int as, int am) {
        // récupération des données  transportées par les paquets
}

public void  Object2Save (Queue file, Object objet){
        // sauvegarde un objet dans une file donnée
}

class NetworkListener  implements Runnable { // écoute les paquets à l'arrivée
}

class PacketFilter implements Runnable {  // filtre les paquets selon leur type
void d_processing () {}  // traitement des d-paquets
void req4code_processing(){
// traitement des requêtes de codes dans le cade de la méthode mixte
void s_processing (){}  // traitement des s-paquets
void m_pocessing (){}   // traitement des m-paquets
}

class SecurityManager {
// gère la sécurité lors de la réception des paquets actifs et de leur exécution
```

```
}

class EventListenning implements Runnable {
// écoute les événements définis par le système ou par les applications utilisateurs
}

class MonitorsManager {  // gère l'exécution parallèle des moniteurs
}

class MonitorInstantiater {
        // instancie la classe moniteur à la réception d'un m-paquet
}

class DUABinder {  // lie une donnée active à une application
}

class SAMBinder {  // lie un serveur d'application à un moniteur actif
}

class  NetworkSender { // envoie les paquets vers le réseau
}
}
```

B.4 Le serveur de code CISS

Le CISS partage une composition presque semblable à une machine virtuelle.

```
public class CISS {

Queue sElementQueue ;  // file des s-paquets
Queue mElementQueue ;  // file des m-paquets
Queue req4CodeQueue;   // file des requêtes de code

Queue InputQueue ;   // file d'entrée
Queue OutputQueue ;  // file de sortie

public class NetListener  implements Runnable { // écoute les paquets arrivés

public class PacketFilter implements Runnable { // filtre les paquets selon leur type

void req4code_processing(){}  // traitement des requêtes de codes dans le cade de
                                       // la méthode mixte
void s_processing (){}  // traitement des s-paquets
void m_pocessing (){}   // traitement des m-paquets
void identifier_assign (){}  // attribution des identificateurs
}
```

```
public class SecurityManager {
// effectue des traitements de sécurité en applicant des algorithmes de sureté
}

 public class StorageManager implements Runnable {
// gère le stockage des objets sur le disque
}

public static class  NetSender {  // envoie les paquets vers le réseau
}
}
```

Annexe C

C L'interface Graphique

Pour permettre à tout développeur de tester son application active, nous avons mis en place une plate-forme de tests qui offre la possibilité de construire sa propre topologie, de créer l'application active et de l'envoyer au serveur de code et finalement de lancer les applications selon la topologie réseau choisie.

C.1 Construction d'une topologie réseau

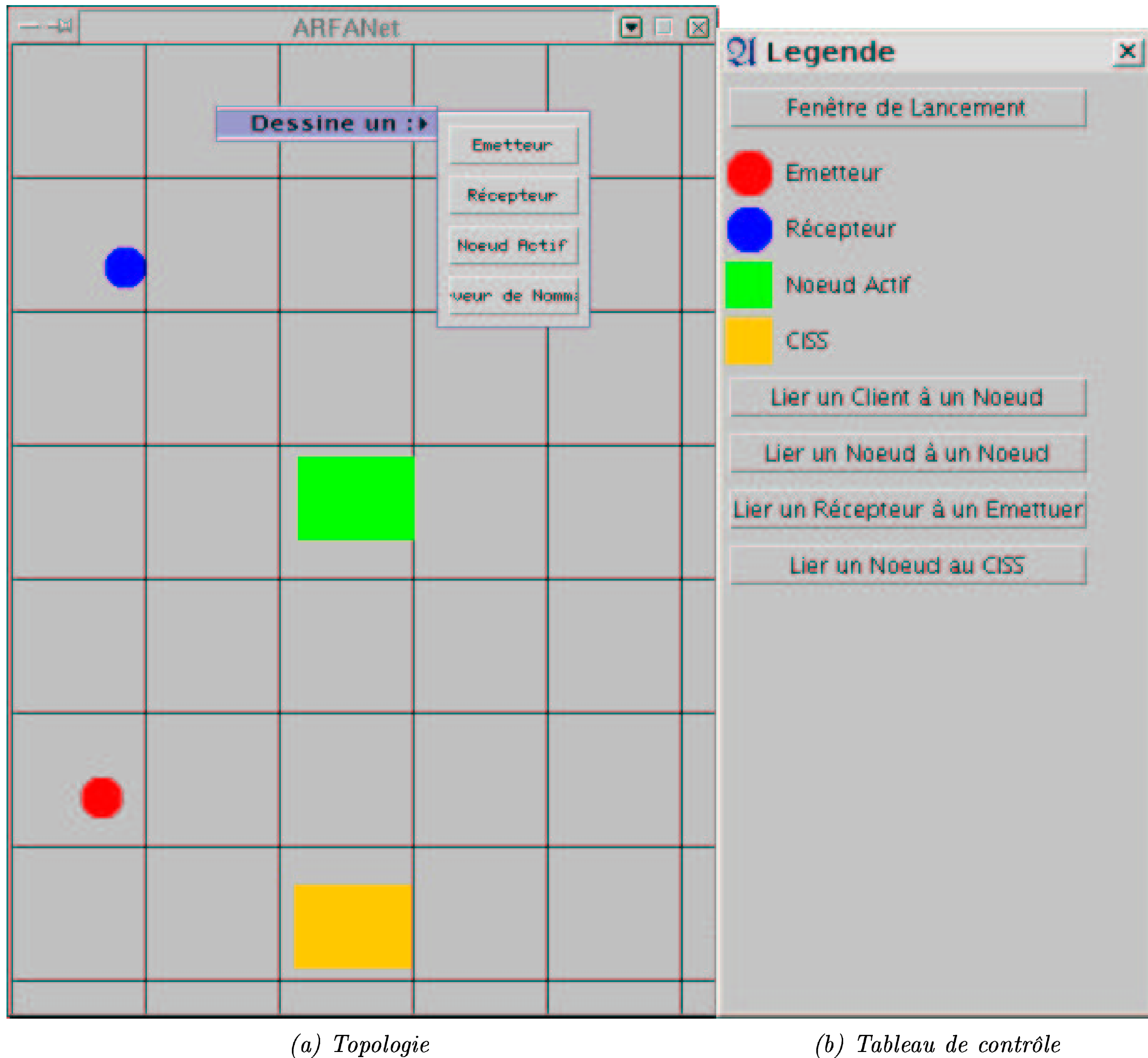
Les deux figures suivantes montrent deux fenêtres de notre interface graphique. La fenêtre 3 correspond à un panneau, au départ vide, qui permet à l'utilisateur de placer des émetteurs, des récepteurs, des nœuds actifs et des serveurs de codes. Une fois cette tâche accomplie, l'utilisateur doit attacher chaque émetteur et récepteur à un nœud actif, de même pour le ou les serveurs de codes. Le nœud construit sa table de routage au fur et à mesure que des machines lui soient attachées. L'ensemble des routeurs sera ensuite raccordé selon la topologie choisie. Durant cette étape, les routeurs s'échangent et complètent leurs tables de routage. A ce moment, l'utilisateur peut lancer les applications. Il est important de noter que ce n'est pas une simulation, mais chaque entité consiste en une application réelle correspondant au type choisi qui s'exécute en parallèle. A titre d'exemple, chaque nœud actif est une instantiation de la classe "VirtualMachine" et doit router les paquets et lancer les règles actives.

C.2 Lancement des applications

La fenêtre 4(a) est un exemple de topologies qu'on peut construire avec notre système. Avant de lancer les applications, l'utilisateur peut construire des applications actives qu'il intègre en temps réel au serveur CISS (voir la section suivante). La fenêtre 4(b) est un tableau de contrôle et contient aussi la légende. Par le changement de couleurs des entités, comme indiqué par la légende, il est possible de suivre toutes les étapes et les événements qui se produisent dans le réseau tels que l'envoi et le transfert d'un paquet passif, d'un paquet actif, d'une demande de code ou d'un paquet de code du serveur CISS. A leur tour, les nœuds actifs réagissent aux événements par des couleurs différentes. L'arrivée d'un paquet, la détection d'un événement (ici le seul événement est l'arrivée d'un paquet actif) et l'exécution d'une règle. Tous les commentaires peuvent être affichés à partir du tableau de contrôle.

C.3 Construction d'un moniteur actif

L'utilisateur peut créer un nouveau moniteur en spécifiant les sémantiques d'exécution correspondantes telles que la consommation d'événements, la source d'événements (interne, externe ou



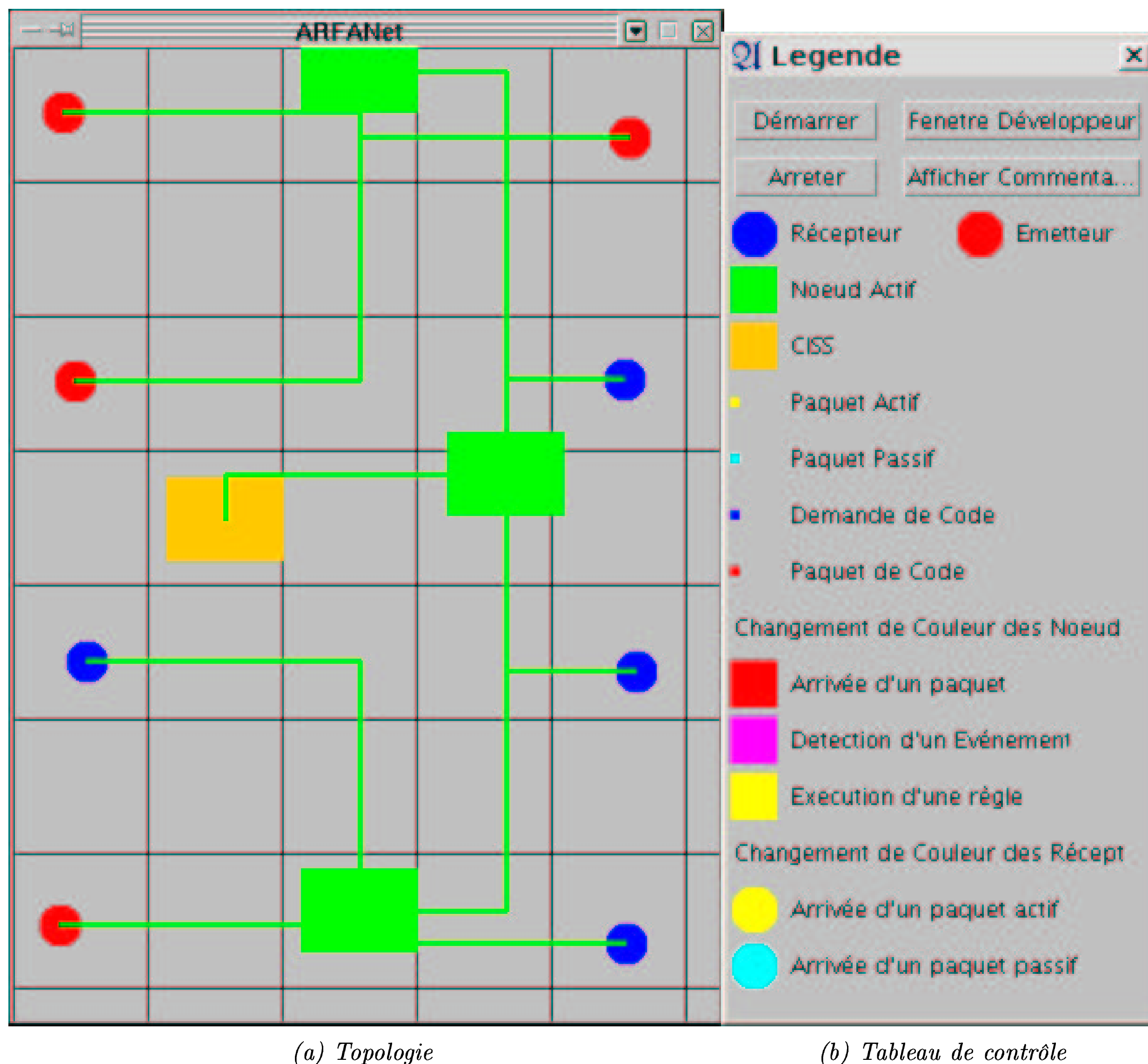
(a) Topologie *(b) Tableau de contrôle*

FIG. 3 – Création d'une topologie

système), le couplage E-C (immédiat et différé), le couplage C-A (immédiat et différé), la priorité des règles (aucune, globale, relative ou aléatoire) et finalement le type de l'exécution en cascade des règles (itérative ou récursive), comme le montre la fenêtre 5. A partir de ce point, l'utilisateur pourra envoyer au serveur CISS son nouveau moniteur qui lui retourne l'identifiant attribué.

C.4 Construction d'un service d'application

Le service d'application nécessite plusieurs étapes que nous présentons dans les sous-sections suivantes.



(a) Topologie *(b) Tableau de contrôle*

FIG. 4 – Lancement des applications

C.4.1 Création et combinaison des événements

La première étape que doit effectuer un développeur de service d'application est la définition de tous les événements. La fenêtre 6(a) permet de définir chaque événement et tous ses attributs. Pour l'événement, il faut spécifier son identifiant et son type. Chaque attribut est défini par un identifiant, un type (entier, réel, char, booléen, octet), sa valeur initiale et la valeur de déclenchement (le système déclenche l'événement lorsque la valeur de l'un de ses attribut a atteint la valeur de déclenchement). Le rôle de la seconde fenêtre (6(b)) est d'offrir un moyen de composer des événements complexes. Les opérateurs logiques et les parenthèses permettent toutes les expressions possibles. Les expressions résultantes sont vérifiées avant d'être validées.

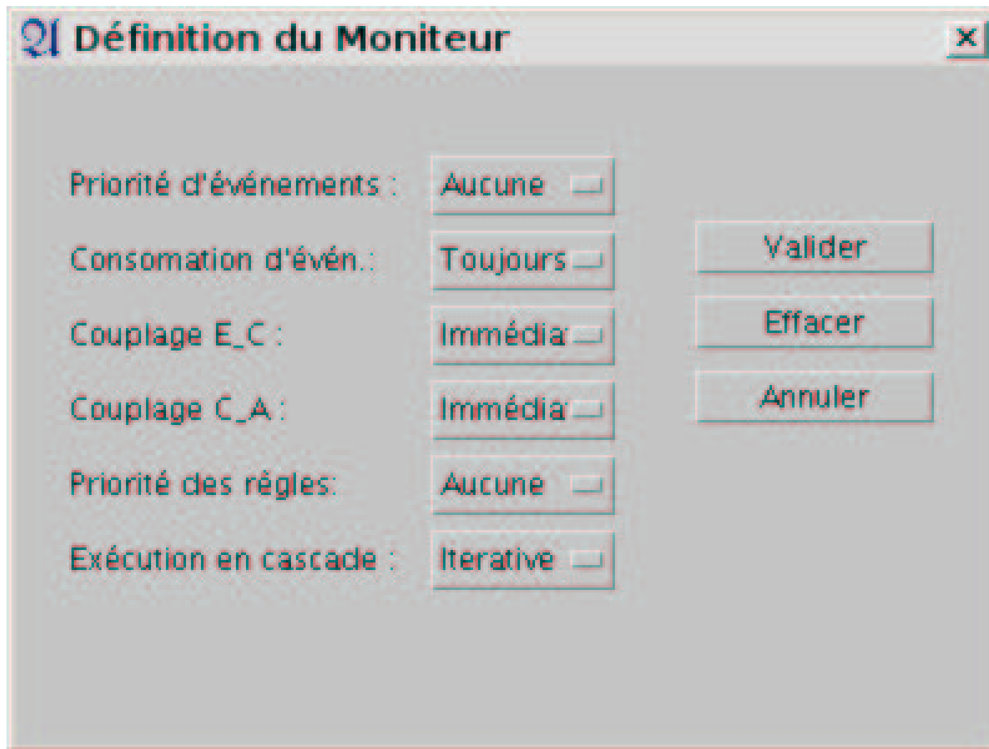


FIG. 5 – Construction d'un moniteur actif

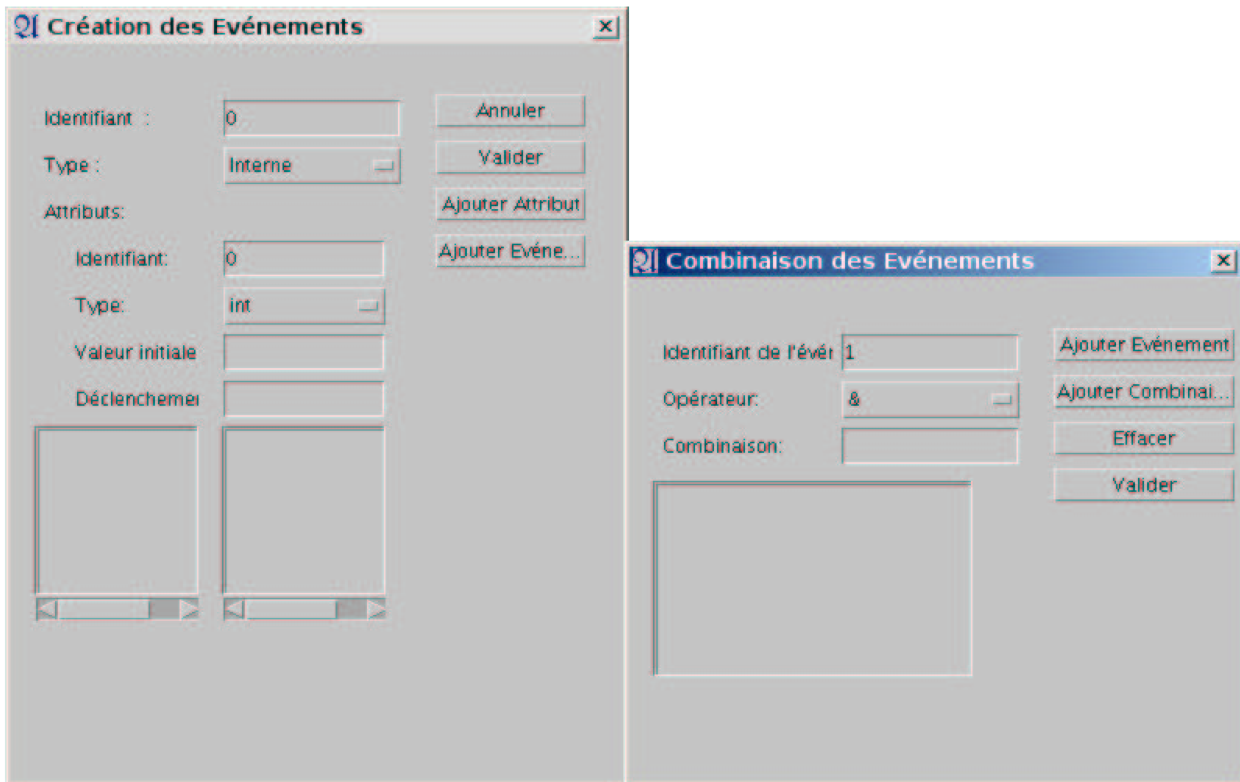
C.4.2 Création des conditions et des actions

Les conditions et les actions sont des classes Java. Le développeur doit écrire le corps de chaque classe en utilisant des fonctions précises, qui sera ensuite compilée. Si le système arrive à compiler la classe, alors la condition ou l'action est ajoutée à l'ensemble correspondant. Le développeur ne doit donner que le corps de la méthode, comme le montre la fenêtre 7, car le système se charge de l'encapsuler et de lui ajouter l'entête exacte pour éviter toute fausse signature.

Pour cacher la sauvegarder et la gestion des données liées aux événements, deux fonctions de récupération et de mises à jour ont été définies : *int getEvAttrVal(int Attr_Indent, int Event_Ident)* et *void setEvAttrVal(int Attr_Indent, int Event_Ident, int valeur)*. De même les données sur les paquets peuvent être manipulées à leur arrivée. Les fonctions correspondantes sont : *int getData(int Event_Ident)* et *void setData(int Event_Ident, int valeur)*.

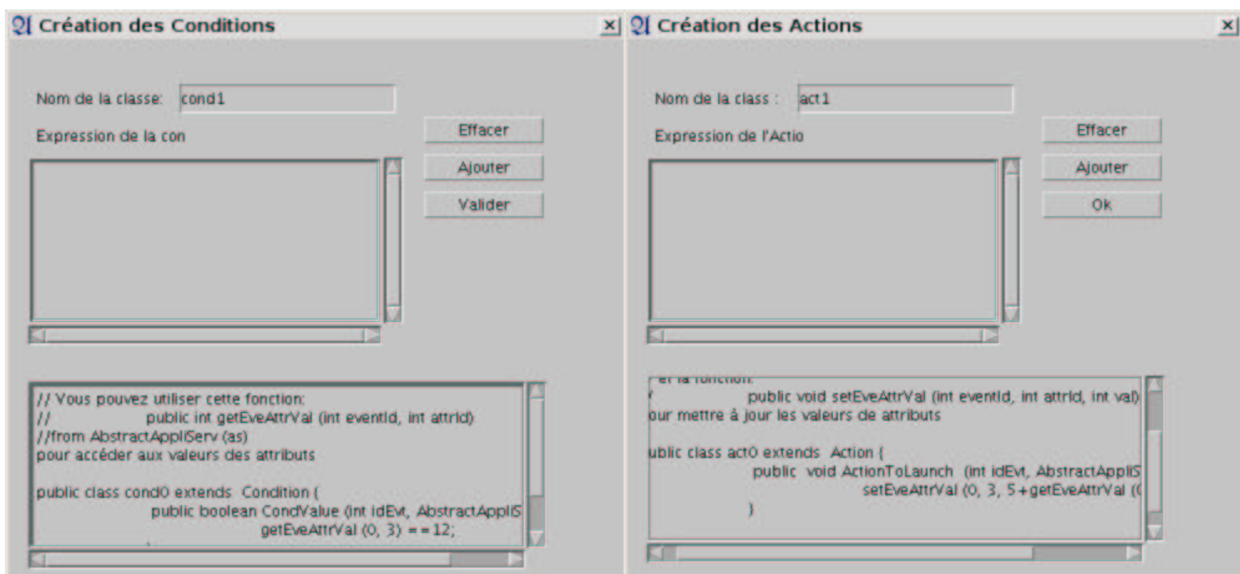
C.4.3 Création des règles

A la fin des étapes précédentes, le système détient une liste d'événements, de conditions et d'actions. La règle étant constituée d'un événement, d'une condition et d'une action, l'utilisateur est amené à définir chaque règle en lui associant l'identifiant de la liste pour chaque type et sa priorité, voir fenêtre 8(a). Le sens de la priorité est lié à la sémantique d'exécution définie par le moniteur.



(a) *Création des événements* (b) *Combinaison des événements*

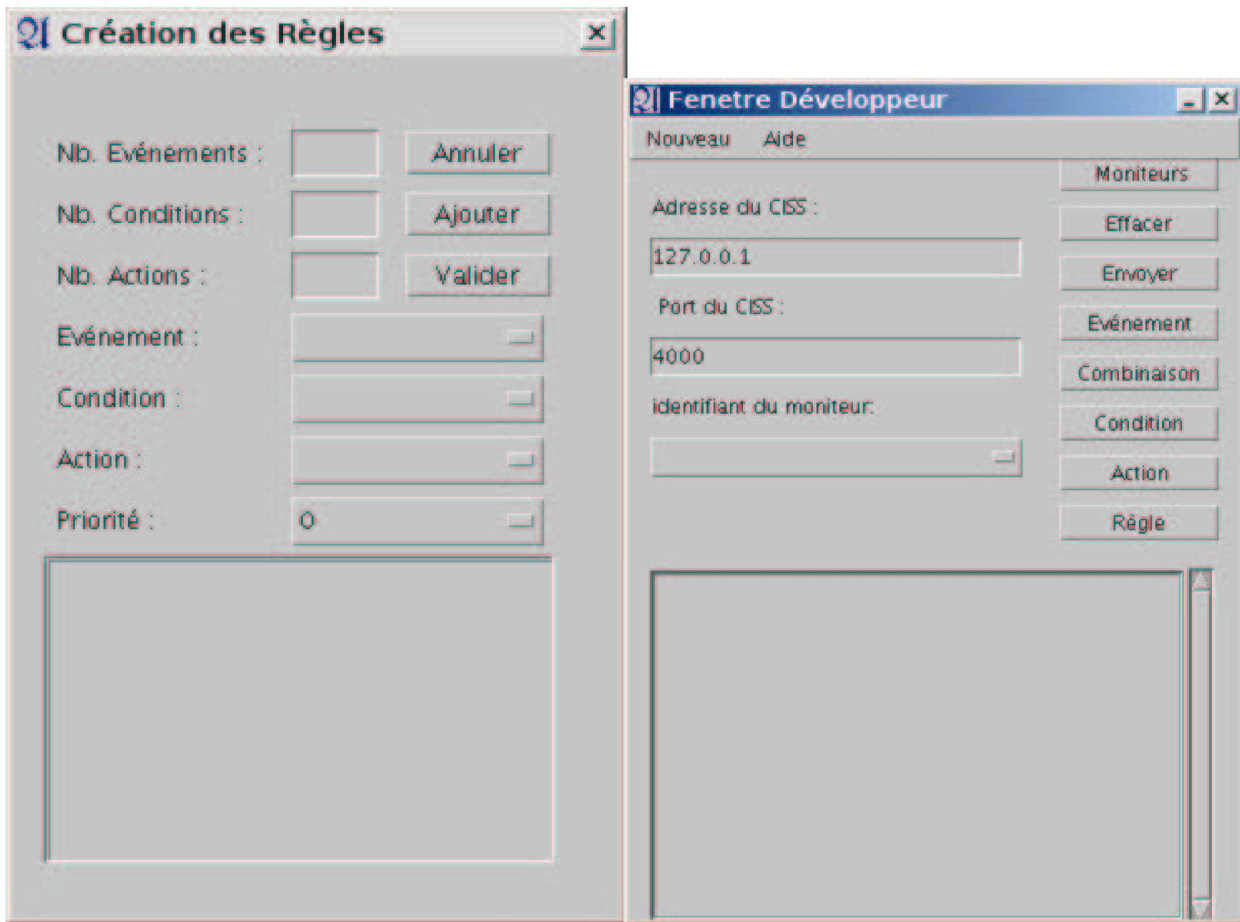
FIG. 6 – Traitement des événements



(a) *Création des conditions* (b) *Création des actions*

FIG. 7 – Création des conditions et des actions

Après la définition de toutes les règles, le service d'application est prêt à être envoyé au serveur CISS. Il est impératif de désigner à quel moniteur actif appartient le service d'application.



(a) Création des règles *(b) Envoi du service d'application*

FIG. 8 – Création des règles

Bibliographie

[1] RFC 2510. Internet x.509 public key infrastructure certificate management protocols.

[2] Jesús Acosta-Elías and Leandro Navarro-Moldes. A demand based algorithm for rapid updating of replicas. In *ICDCS Workshops*, pages 686–694, 2002.

[3] D. Alexander, W. Arbaugh, A. Keromytis, S. Muir, and J. Smith. Secure quality of service handling : Sqosh, 2000.

[4] D. Alexander, W. Arbaugh, A. Keromytis, and J. Smith. Safety and security of programmable network infrastructures, 1998.

[5] D. Scott Alexander, Kostas G. Anagnostakis, William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith. The price of safety in an active network. In *SIGCOMM '99*, 1999.

[6] D. Scott Alexander, William A. Arbaugh, Michael Hicks, Pankaj Kakkar, Angelps D. Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott Nettles, and Jonathan M. Smith. The switchware active network architecture. Juillet 1998. Departement of Computer and Information Science University of Pennsylvania.

[7] D. Scott Alexander, William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith. Security in active networks. pages 433–451, 1999.

[8] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time Markov chains. In *Computer-Aided Verification*, volume 1102 of *LNCS*, pages 169–276. Springer-Verlag, 1996.

[9] T. Faber B. Lindell G. Phillips B. Braden, A. Cerpa and J. Kann. Asp ee : An active execution environment for network control protocols. *Technical report USC/ISI*, Décembre 1999.

[10] A. Benoit, L. Brenner, P. Fernandes, and B. Plateau. Aggregation of stochastic automata networks with replicas. In *Proceedings of the International Conference on the Numerical Solution of Markov Chains (NSMC'03), Illinois*, pages 215–234. CLUT, Septembre 2-7 2003.

[11] M. Bouzeghoub, L. Kloul, and A. Mokhtari. A new active network framework based on active rules. Rapport de Recherche 21, PRiSM, 2002.

[12] R. Braden, B. Lindell, S. Berson, and T. Faber. The asp ee : An active network execution environment. Ieee cs press, Proceedings of DARPA Active Networks Conference and Exposition (DANCE'02), 2002.

[13] K. Calvert. Architectural framework for active networks. Rapport de recherche, AN Architecture Working Group, Juillet 1998.

[14] Y. Chae and E. Zegura. Canes : An execution environment for composable services. In *DANCE '02 : Proceedings of the 2002 DARPA Active Networks Conference and Exposition, year = 2002, isbn = 0-7695-1564-9, pages = 255, publisher = IEEE Computer Society, address = Washington, DC, USA,*.

[15] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA*, pages 322–328, 2002.

[16] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing : The JML and JUnit way. In Boris Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Máalaga, Spain, Proceedings*, volume 2374 of *LNCS*, pages 231–255, Berlin, Juin 2002. Springer-Verlag.

[17] G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. Webster. The Möbius modeling tool. In *Proceedings of the 9th International Workshop on Petri Nets and Performance Models*, pages 241–250, Aachen, Germany, September 2001.

[18] Ian Clarke. Freenet, http ://www.freenetproject.org.

[19] Carl Cook. *The Design and Evaluation of a Multiple Language Active Network Architecture Enabled via Middleware*. PhD thesis, University of Canterbury, 2001.

[20] Umeshwar Dayal, Eric Hanson, and Jennifer Widom. Active database systems. pages 434–456, 1995.

[21] D. Decasper, G. Parulkar, and B. Plattner. A scalable, high performance active network node, 1999.

[22] D. Decasper and B. Plattner. Dan - distributed code caching for active networks. In *IEEE INFOCOM*, San Francisco, Avril 1998.

[23] D. Deharbe and S. Ranise. Light-weight theorem proving for debugging and verifying units of code, 2003.

[24] Darryl Dieckman, Perry Alexander, and Philip A. Wilsey. ActiveSPEC : A framework for the specification and verification of active network services and security policies. In Nevin Heintze and Jeannette Wing, editors, *Workshop on Formal Methods and Security Protocols*, Indianapolis, IN, June 1998. Informal proceedings available at.

[25] Institute for Computing Systems Architecture Division of Informatics, University of Edinburgh. Simjava.

[26] S. Donatelli. Superposed generalised stochastic petri nets : Definition and efficient solution. In M. Silva, editor, *Proc. of 15th Int. Conf. on Application and Theory of Petri Nets*, 1994.

[27] Angela Kotz-Dittrich Eric Simon. Active database systems : Expectations, commercial experience and beyond. In N. Paton, editor, *Monographs in Computer Science*, pages 367–402. Active Rules /in Database Systems, 1998.

[28] Olivier Festor. *Les Réseaux Programmables*. LORIA, NANCY,FRANCE, Mars 2000.

[29] P. Fraternali and L. Tanca. A structured approach for the definition of the semantics of active databases. ACM Transactions On Database Systems, Décembre 1995.

[30] Virginie Galtier, Craig Hunt, Stefan Leigh, Kevin L. Mills, Doug Montgomery, and Mudumbai. How much cpu time ? expressing meaningful processing requirements among heterogeneous nodes in an active network.

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

[32] Rory Gavino and Vidal Burgoa. *A Flexible Client-Driven Service for Active Network*. Licence, Université de Genève, Mai 2000.

[33] Jean-Patrick Gelas. *Vers la conception d'une architecture de réseaux actifs apte à supporter les débits des réseaux gigabits*. PhD thesis, ENS Lyon, décembre 2003.

[34] Andreas Geppert and Mikael Berndtsson. (eds.) : Rules in database systems. In ISBN 3-540-63516-5 Springer, editor, *Lecture Notes in Computer Science, Vol. 1312*. Third International Workshop, RIDS ́97, Sk:ovde, Sweden, Juin 26-28, 1997.

[35] M. P. Gervais. Composition of active services : A methodological approach. In *Proceedings of the Fourth Annual International Working Conference on Active Networks (IWAN'02)*, Zürich, Décembre 2002. Poster Session.

[36] S. Gilmore and J. Hillston. The pepa workbench : A tool to support a process algebra-based approach to performance modelling. In *Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, volume 794, pages 353–368, Vienna, 1994. Lecture Notes in Computer Science, Springer-Verlag.

[37] S. Gilmore, J. Hillston, L. Kloul, and M. Ribaudo. PEPA nets : A structured performance modelling formalism. 54 :79–104, 2003. Performance Evaluation, Elsevier Science.

[38] Richard Andrew Golding. *Weak-consistency group communication and membership*. PhD thesis, Santa Cruz, CA, USA, 1992.

[39] AN NodeOS Working Group. Nodeos interface specification.

[40] AN Security Working Group. Security architecture for active nets, 2001.

[41] Composable Services Working Group. An composable services working group for active networks, Septembre 1998.

[42] Open Signalling Working Group. Url :- http ://comet.columbia.edu/opensig/.

[43] Robert Haas. *Service Deployment in Programmable Networks*. PhD thesis, Zurich, 2003.

[44] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. Plan : A packet language for active networks. 12(3) :29, 36, Juillet 1998. Departement of Computer and Information Science University of Pennsylvania.

[45] Michael Hicks, Jonathan T. Moore, D. Scott Alexander, Carl A. Gunter, and Scott Nettles. Planet : An active internetwork. 1999. Departement of Computer and Information Science University of Pennsylvania.

[46] J. Hillston. *A Compositional Approach to Performance Modelling.* PhD. Thesis, The University of Edinburgh, 1994.

[47] J. Hillston and L. Kloul. An efficient kronecker representation for pepa models. In *Proceedings of Process Algebra and Probabilistic Methods : Performance Modelling and Verification*, volume 2165, pages 120–135, Aachen, 2002. Lecture Notes in Computer Science, Springer-Verlag.

[48] Frederick J. Hirsch. Introducing ssl and certificates using ssleay. In *The Open Group Research Institute.* Published in World Wide Web Journal, Summer 1997.

[49] Allen Householder, Kevin Houle, and Chad Dougherty. Computer attack trends challenge internet security. *IEEE Computer Supplement on Security and Privacy*, pages 5–7, Avril 2002.

[50] A. Bavier P. Bigot P. Bridges B. Montz R. Piltz T. Proebsting J. Hartman, L. Peterson and O. Spatscheck. Joust : A platform for liquid software. pages 50–56. IEEE Computer Magazine, Avril 1999.

[51] Alexandra Poulovassilis James Bailey and Peter Wood. Analysis and optemisation of event-condition-action rules on xml. pages Vol 39, 239–259. Computer Networks Journal - Special Issue on Web Dynamics, 2002.

[52] Pankaj Kakkar, Michael Hicks, Jonathan T. Moore, and Carl A. Gunter. Specifying the plan network programming language. In *HOOST'99*, 1999.

[53] KaZaa :. http ://www.kazaa.com/.

[54] R. Keller, L. Ruf, A. Guindehi, and B. Plattner. Promethos : A dynamically extensible router architecture for active networks, 2002.

[55] Stella Gatziu Klaus R. Dittrich and Andreas Geppert. The active database management system manifesto : A rulebase of adbms features. Athens, 1995. 3-20, International Workshop on Rules in Database Systems (RIDS).

[56] T. Klingberg and R. Manfredi. Gnutella 0.6, http ://rfc-gnutella.sourceforge.net/draft.txt, 2002.

[57] Cindy Kong, P. Alexander, and Darryl Dieckman. Formal modeling of active network nodes using pvs. In *FMSP '00 : Proceedings of the third workshop on Formal methods in software practice*, pages 49–59, New York, NY, USA, 2000. ACM Press.

[58] S Kuchinskas. A decade of e-commerce. Site web, http ://www.internetnews.com/ec-news/article.php/3422901, Octobre 2004.

[59] M. Kwiatkowska, G. Norman, and D. Parker. PRISM : Probabilistic symbolic model checker. In T. Field, P. Harrison, J. Bradley, and U. Harder, editors, *Proc. 12th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'02)*, volume 2324 of *LNCS*, pages 200–204. Springer, 2002. http ://www.cs.bham.ac.uk/ dxp/prism/.

[60] M. Hibler P. Tullman J. Lepreau S. Schwab H. Dandekar A. Purtell L. Peterson, Y. Gottlieb and J. Hartman. An os interface for active router. 2001.

[61] Ulana Legedz, David J. Wetherall, and John V. Guttag. on active networking. *IEEE*, 1998.

[62] L. Lehman, J. Garland, and D. Tennenhouse. Active reliable multicast. *Proc. IEEE INFO-COM'9*, Aout 1998.

[63] F. Llirbat E. Simon M. Bouzeghoub, F. Fabret and M. Matulovic. Active-design : A generic toolkit for deriving specific rule execution models. Third International Workshop on Rules in Database Systems, RIDS'97 , LNCS ,Skovde, Sweden , Vol. 1312, Juin 1997.

[64] F. Llirbat M. Matulovic M. Bouzeghoub, F. Fabret and E. Simon. Active-design. In A. Geppert editor Springer, editor, *Lecture Notes in Computer Science*. Proceedings of The 3rd International Workshop on Rules in Database Systems (RIDS'97), Sweeden, Juin 1997, Sept 1997.

[65] Maria Calderon Marcelo Bagnulo, Bernardo Alarcos and Marifeli Sedano. Provinding authentication & authorization mechanisms for active service charging. *Lecture Notes in Computer Science*, Zurich, Switzerland(2511) :337,346, Octobre 2002.

[66] Maria Calderon Marcelo Bagnulo, Bernardo Alarcos and Marifeli Sedano. Rosa : Realistic open security architecture for active networks. In *IWAN '02 : Proceedings of the IFIP-TC6 4th International Working Conference on Active Networks*, pages 204–215, London, UK, 2002. Springer-Verlag.

[67] Hideki Otsuki Matthias Bossardt, Takashi Egawa and Bernhard Plattner. Integrated service deployment for active networks. *In Proceedings of Fourth Annual International Working Conference in Active Networks (IWAN 2002), Zürich, Switzerland*, Décembre 2002.

[68] W. Miao. Worldwide ip traffic patterns and network evolutions. http ://www.atmforum.com/meetings/bbx-july01.html, Advanced Consumer Service Research-Probe Research, 2001.

[69] Jonathan T. Moore, Michael Hicks, and Scott Nettles. Chunks in plan : Language support for programs as packets. Mars 1999. Departement of Computer and Information Science University of Pennsylvania.

[70] R.T. Morris. *Scalable TCP congestion Control*. PhD. Thesis, Harvard University, Cambridge, Massachussets, 1999.

[71] S. Murphy, E. Lewis, R. Puga, R. Watson, and R. Yee. Strong security for active networks, 2001.

[72] Napster. http ://www.napster.com/.

[73] G. C. Necula and P. Lee. Research on proof-carrying code for untrusted-code security. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, 1997.

[74] M. Hibler P. Tullmann and J. Lepreau. Janos : A java-oriented os for active networks. In *Vol. 19, N. 3*. IEEE Journal on Selected Areas of Communication, Mars 2001.

[75] S. Paltridge. Internet traffic exchange and the development of end-to-end international telecommunication competition. Organisation for economic co-operation and development, Mars 2002.

[76] Norman W. Paton. (eds.) : Active rules in database systems. In *Edinburgh, Scotland, 30 Août - 1 Septembre 1993. Workshops in Computing, Springer 1994, ISBN 3-540-19846-6 and 0-387-19846-6*. Proceedings of the 1st International Workshop on Rules in Database Systems, 1994.

[77] Norman W. Paton and M. Howard Williams. (eds.) : Rules in database systems. proceedings of the 1st international workshop on rules in database systems. Edinburgh, Scotland, 30 Août - 1 Septembre 1993. Workshops in Computing, Springer 1994, ISBN 3-540-19846-6 and 0-387-19846-6.

[78] B. Plateau. On the Stochastic Structure of Parallelism and Synchronisation Models for Distributed Algorithms. In *Proc. ACM Sigmetrics Conference on Measurement and Modelling of Computer Systems*, 1985.

[79] W.H. Sanders and J.F. Meyer. Reduced base model construction methods for stochastic activity networks. *IEEE Journal on Selected Areas in Communications*, 9(1) :25–36, January 1991.

[80] S. Schmid. Lara++ design specification, 2000.

[81] Timos K. Sellis. (ed.) : Rules in database systems. In ISBN 3-540-60365-4 Springer, editor, *Lecture Notes in Computer Science, Vol. 985*. Second International Workshop, RIDS 9́5, Glyfada, Athens, Greece, Septembre 25 - 27, 1995.

[82] E. Zegura S.Merugu, S. Bhattacharjee and K. Calvert. Bowman : A node os for active networks. Tel Aviv, Israel, Mars 2000. Proceedings of IEEE Infocom 2000.

[83] J. M. Smith, K. Calvert, S. Murphy, H. Orman, and L.L. Peterson. Activating networks. Novembre 1998.

[84] J.M. Smith, K. Calvert, H. Orman, and L. L. Peterson. Activating networks. 1998.

[85] David L. Tennenhouse, Jonarhan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, Janvier 1997.

[86] David L. Tennenhouse and David J. Wetherall. Towards an active network architecure. *http ://www.tns.lcs.mit.edu*, Janvier 1996.

[87] Scott Thibault, Charles Consel, and Gilles Muller. Safe and efficient active network programming. *17th IEEE Symposium on Reliable Distributed Systems*, West Lafayette, Indiana :135, 143, Octobre 1998.

[88] Nevil Brownlee kc claffy Thomas Karagiannis, Andre Broido and Michalis Faloutsos. File-sharing in the internet : A characterization of p2p traffic in the backbone. Technical report, Novembre 2003.

[89] UNCTAD. E-commerce and development report 2003. United nations-new york and geneva, 2003.

[90] Ian Wakeman, Alan Jeffrey, Tim Owen, and Damyan Pepper. Safetynet : A language-based approach to programmable networks. *Computer Networks*, 36(1) :101–114, 2001.

[91] David Wetherall and David Tennenhouse. The active ip option. *the 7th ACM SIGOPS European Workshop*, Connemara, Ireland :135, 143, Septombre 1996.

[92] David J. Wetherall. Active network vision and reality : lessons from a capsule-based system. *Operating Systems Review*, 17th ACM Symposium on Operating Systems Principles (SOPS'99)(34) :64, 79, Décembre 1999.

[93] David J. Wetherall. *Service Introduction in an Active Network*. PhD thesis, Massachusetts Institute of Technology, Février 1999.

[94] David J. Wetherall, John V. Guttag, and David L. Tennenhouse. Ants : A toolkit for building and dynamically reactive network vision and reality : lessons from a capsule-based system. In *IEEE OPENARCH'98*, San Francisco,CA, Avril 1998.

[95] J. Widom and S. Ceri. Active database systems, triggers and rules for advanced database processing. San Fransisco, 1996. ISBN 1-55860-304-2. Morgan Kauffmann Publishers INC.

[96] Yechiam Yemini and Sushil da Siliva. Towards programmable network. *http ://www.cs.columdia.edu/*, Avril 1996.

[97] Roy H. Campbell Zhaoyu Liu and M. Dennis Mickunas. Securing the node of an active network. Septembre 2000.

[98] Y. Zhou, Y. Zhang, and J. Lu. Cds : a code distribution scheme for active networks. *Computer Communications*, 27(3) :315,321(7), 15 Février 2004.

Abstract

We developed a new active network infrastructure, called ARFANet (Active Rule Framework for Active Network), based on the notion of actives rules. In this context, an application can be described as a set of active rules, where each rule is defined as a Event-Condition-Action (ECA) statement. The execution of an application consists to, then, detect the events, evaluate the conditions and finally launch the corresponding actions. The ECA form is exclusively event based concept and makes active rules a potential candidantes for network applications which must react to an arising events within the network, such as packet arrival, bottleneck, congestion, link breaking down, etc.

The two most important aspects in active networking are the code distribution and security. We defined the notion of Code Identification and Storage Server (CISS) which allows the storage of the code and its unique identification. With this server we can apply all security techniques and specially on all actors implied in code distribution process such as the developers, the users and the code itself. The security tasks are not any more within the competence of the nodes but delegated to this server, thus the performances of nodes are not further deteriorated. Among those techniques we used cryptophic mechanisms and public key systems for the authentication of all entities concerned by the deployment process including the routers and the codes. The code execution safety validation using PCC method (Proof Code Carrying) has been adapted to ARFANet and takes into account the code providers classification to défine check rules. Symmetric keys or credentials establishment allows to only authorised users to execute the codes on network nodes. The study of the security at the distribution level is not enough to guarantee the safe execution of the active programmes in the nodes. Consequently, we enhanced the study of the security of run time program execution in order to avoid all dysfunctions of the nodes or all right violation access. Among those dysfunctions we note : the occurrence of infinite cycles during the active rule execution, non authorised memory or resources access, excessive packets creation, ect. We defined an adequate methods to secure mainly active rules execution.

Moreover, we observed the necessity to introduce more than one code identification and storage server in large scale networks such as Internet. The distributed code base management must take into account all specifics of active networks. However, there are several ways to manage this base by using a distributed code bases over all the sites or a duplicated code bases on each site. We choose the first technique to avoid the redundancy and the frequent updating. In the context of ARFANet, the active code is composed of several active rules and can be divided to several disjoint modules. This modularity simplifies service composition and avoids unsecured and expensive mechanisms. The composition in ARFANet is performed anteriorly of the execution by referring events of a composable modules

We also concentrated a large part of our efforts on the performance evaluation and the modelling of nodes in ARFANet infrastructure in the purpose to study of the possibility to introduce active nodes in the current networks and the influence of code packet introduction on the transfer of standard packets. Thus, we compared the standard nodes performance, which the unique functionality is to forward the packets flowing through them, with the performance of an ARFANet active node. Once the platform implemented, we performed direct measures to validate the analytical and the simulation models results.

Résumé

Nous avons développé une nouvelle infrastructure de réseaux actifs, appelée ARFANet (Active Rule Framework for Active Networks), basée sur la notion de règles actives. Dans ce contexte, une application peut être décrite comme un ensemble de règles actives, où chaque règle est définie sous la forme d'un tuple Evènement-Condition-Action (ECA). L'exécution de l'application consiste alors à détecter les évènements, évaluer les conditions et lancer les actions correspondantes. La forme ECA est fondée exclusivement sur un concept événementiel et rend les règles actives des candidates potentielles pour les applications réseau qui doivent réagir aux événements survenant au sein du réseau tels l'arrivée d'un paquet, une congestion, etc.

Deux aspects très importants dans les réseaux actifs sont la distribution du code actif et la sécurité. Nous avons défini la notion de serveur de code (CISS) qui permet le stockage de code et son identification. Avec ce serveur nous pouvons appliquer toutes les techniques de sécurité impliquant tous les acteurs liés au processus de la distribution tels que le développeur, les utilisateurs et le code lui-même. Les tâches de sécurité ne sont plus du ressort des nœuds mais déléguées à ce serveur, ainsi les performances des nœuds ne sont pas altérées. Parmi ces techniques nous avons employé les mécanismes de cryptage et les systèmes à clefs publiques pour l'authentification de toutes les entités concernées par le processus du déploiement y compris le code et les routeurs. Pour la validation de la sûreté des codes la technique PCC (Proof Code Carrying) a été adaptée dans le contexte d'ARFANet et prend en compte désormais la classification des fournisseurs de codes pour définir les règles de contrôle. L'établissement de clefs symétriques (ou credentials) permet aux seuls utilisateurs autorisés d'exécuter un code sur les nœuds. L'étude de la sécurité au niveau de la distribution ne suffit pas à elle seule pour garantir la bonne exécution des codes actifs dans les nœuds. Par conséquent, nous avons étudié la sécurité des programmes en cours d'exécution pour éviter tout dysfonctionnement des nœuds ou la violation des droits. Parmi ces dysfonctionnements nous citons : la formation des cycles infinis dans l'exécution des règles actives, accès à des zones mémoires ou des ressources non autorisées, la création abusive de paquets, etc. Nous avons, de ce fait, défini des méthodes adéquates pour sécuriser principalement l'exécution des règles actives.

Par ailleurs nous avons constaté la nécessité d'introduire plusieurs serveurs d'identification et de stockage de code dans les réseaux de grande envergure. La gestion de la base de codes distribuée doit prendre en compte toutes les spécificités des réseaux actifs. Cependant il existe plusieurs manières de gérer cette base en utilisant des bases de codes réparties sur l'ensemble des sites ou des bases de codes répliquées sur chaque site. Nous avons opté pour la première technique car elle évite la redondance et la gestion des mises à jours fréquentes et convient le plus aux réseaux actifs et en particuliers à ARFANet. Dans le cadre de notre plate-forme, le code actif est sous forme de plusieurs règles actives et peut se subdiviser en plusieurs modules disjoints. Cette modularité simplifie énormément la composition de services sans avoir recours à des mécanismes coûteux et peu sécurisés. La composition dans ARFANet se fait en amont de l'exécution par la désignation des événements susceptibles de déclencher les règles de certains modules.

Nous avons également concentré une grande partie de nos efforts à l'évaluation des performances et la modélisation d'un nœud de l'infrastructure ARFANet afin d'étudier la possibilité d'introduire les nœuds actifs dans les réseaux actuels et l'influence des paquets de code sur le transfert des paquets standards. Pour cela nous avons dû comparer les performances d'un nœud standard dont la seule fonctionnalité est de retransmettre les paquets le traversant avec les performances d'un nœud actif ARFANet. Une fois la plate-forme implémentée, nous avons fait des mesures directes pour valider les résultats du modèle analytique et de la simulation.