Un schéma d'emprunt de ressources pour l'adaptation du comportement d'applications distribuées

Narkoy Batouma

N° d'ordre : 2011ISAL0090
Année : 2011

THESE

Un schéma d'emprunt de ressources pour l'adaptation du comportement d'applications Distribuées

Présenté devant
L'Institut National des Sciences Appliquées de Lyon

Pour obtenir
Le grade de Docteur

Formation doctorale
Architectures distribuées et collaboratives
Ecole doctorale
Informatique et Mathématiques (Info-Maths)

Par
BATOUMA Narkoy
(Ingénieur)
Soutenu le 30 septembre 2011 devant la commission

Jury MM.

Christian PERCEBOIS	Professeur à l'IRIT Toulouse	Rapporteur
Pham CONGEDUC	Professeur au LIUPPA Pau	Rapporteur
Marc DALMAU	MCF-HDR au LIUPPA Pau	Examinateur
Denis TRYSTRAM	Professeur au L'INP Grenoble	Examinateur
Jean-Louis SOURROUILLE	Professeur au DISP-LIESP Lyon	Directeur de thèse

Laboratoire de recherche : **L**aboratoire d'**I**nformatique pour L'Entreprise et les **S**ystèmes de **P**roduction (**LIESP**), Lyon, France.

Un Schéma d'Emprunt de ressources pour l'Adaptation du Comportement d'Applications Distribuées

Résumé

L'objectif de cette thèse est de contrôler l'utilisation des ressources par les applications qui s'exécutent dans un environnement où la loi d'arrivée des applications est inconnue. Nous visons à maximiser cette utilisation pour optimiser la QoS.

Les systèmes d'exploitation utilisent très généralement une politique de « meilleur effort » (*Best-effort*) pour exécuter les applications. Tant que les ressources sont suffisantes, les applications s'exécutent normalement mais quand les ressources deviennent insuffisantes, des mécanismes de contrôle (*graceful degradation*) sont nécessaires pour continuer à fournir des services de qualité acceptable. Les architectures pour la gestion de la Qualité de Service (*QoS*) ont pour but de prendre en compte les propriétés non-fonctionnelles des applications (disponibilité des ressources, sécurité, etc.) et de contrôler l'exécution des applications dans leur environnement.

Plusieurs architectures centralisées pour la gestion de la QoS d'applications distribuées ont été expérimentées dans notre équipe. Cette thèse a pour objectif de proposer une architecture décentralisée. Une première étude des architectures de gestion de la QoS a montré que les approches décentralisées ont un coût non négligeable en termes de messages échangés, même en se limitant dans chaque nœud à une vue partielle, des ressources disponibles dans le système.

La première partie de la thèse propose un middleware totalement décentralisé pour contrôler l'utilisation des ressources des applications distribuées. Cette approche se fonde sur une planification approximative et un schéma d'emprunt de ressources afin d'améliorer la QoS globale du système. Via ce schéma d'emprunt, chaque nœud construit localement une vue partielle de la disponibilité des ressources dans le système. La connaissance locale de la disponibilité des ressources permet à chaque nœud de prendre des décisions et de planifier l'exécution des applications. Ainsi des messages ne sont échangés que lorsqu'une information manque localement.

Pour un contrôle plus fin de l'exécution, la deuxième partie ajoute un support pour l'adaptation du comportement des applications. Le middleware utilise un modèle général des applications sous forme de graphe d'exécution décoré avec les besoins en ressources.

Chaque application doit être conçue selon ce modèle pour être gérable par le middleware (approche intrusive). Dans ce graphe, les nœuds sont des points de décision et les arcs des actions à exécuter avec les besoins en ressources correspondants et une utilité quantifiant la perception du service rendu par cette action. Un chemin dans le graphe est une exécution possible de l'application avec une certaine utilité, et ce sont ces chemins qui fournissent les degrés de liberté dont le middleware a besoin pour adapter la consommation des ressources au contexte.

Les applications coopèrent avec le *middleware* dans le processus de gestion de la QoS lors de l'admission puis durant toute l'exécution. Le *middleware* est le chef d'orchestre et c'est lui qui pilote l'exécution des actions des applications (arcs dans le graphe d'exécution). Pour valider notre approche, un prototype à base d'agents a été réalisé à l'aide de JADE. Ce prototype simule un environnement avec plusieurs nœuds et plusieurs applications et compare différents modes d'exécution, en particulier le best effort et notre solution, avec et sans adaptation. Les résultats démontrent l'intérêt de notre approche.

Mots clés : Qualité de Service, Gestion de la QoS, Gestion des Ressources, Schéma d'Emprunt, Planning, Architecture, Approche Décentralisée, Agent, Adaptation, Coordination de l'Adaptation, Modèle d'Application, Systèmes Multi-Agents, Applications Distribuées.

A Resource borrowing schema for behavior adaptation of distributed applications

Abstract

The objective of this thesis is to control the use of resources by applications which run in an environment where applications' arrival law is unknown. We aim to maximize resource use for optimizing the delivered QoS.

Generally, Operating Systems control and execute applications based on a *best effort* policy. As long as resources are sufficient, applications are normally executed but when resources become scarce, control mechanisms (graceful degradation) are necessary to maintain acceptable QoS. The purpose of QoS management architectures is to take into account non-functional properties of applications (performance, safety, resource availability, etc) and to control execution evolution of applications in their environment.

Several centralized architectures for QoS management of distributed applications were investigated in our team. This thesis aims to propose a decentralized architecture. Our team study related to architectures of QoS management has shown that decentralized approaches have a notable cost in terms of exchanged messages, even when each node hold a partial view of the system and available resources.

The first part of the thesis proposes a fully decentralized *middleware* to control the use of resources of distributed applications. This approach is based on a resource borrowing schema and an approximate scheduling in order to improve the overall QoS provided by the system. Using this borrowing schema, each node locally constructs a comprehensive view of resources availability in the system. The local knowledge of resources availability makes it possible for each node to make decisions and to schedule the execution of the applications. Thus, messages are exchanged only when information misses locally.

To finely control execution, the second part of the thesis adds a support for behavior adaptation of applications. The middleware interprets a general model of applications as an execution graph enriched with resources requirements. Each application must be designed according to this model to be manageable by the *middleware* (intrusive approach). In this graph, vertexes are decision points and edges represent actions to be executed, the corresponding resources

requirements, and the utility indicates the satisfaction related to the action choice. A path in the graph is a possible execution of application with a certain utility. These paths provide degrees of freedom needed by the middleware to adapt resource consumption to the context.

During the QoS management process, applications cooperate with the middleware at admission and execution phase. The middleware guides and controls the execution of applications actions (edges in the execution graph). To validate our approach, a prototype based on agents has been implemented using JADE. This prototype simulates an environment with several nodes and several applications and compares various policies, in particular the *best effort* and our solution, with and without adaptation. The results show the interest of our approach.

Keywords: Quality of Service (QoS), QoS Management, Resource Management, Borrowing Schema, Scheduling, Architecture, Decentralized Approaches, Agent, Adaptation, coordination Adaptation, Application Model, Multi-Agents Systems, Distributed Applications.

Dédicace

A ma sœur **Wandou-Ounon** Narkoy
Que ton âme repose en paix.

Remerciements

Je souhaite remercier et exprimer ma gratitude la plus profonde à mon directeur de thèse, **Jean-Louis SOURROUILLE**, Professeur à l'INSA de Lyon, pour les précieux conseils et l'expérience dont il a su me faire profiter. Un grand merci pour votre aide, inoubliable.

Je tiens à remercier chaleureusement les membres du Jury pour l'honneur qu'ils m'ont fait en acceptant de participer à mon jury de thèse.

Un merci particulier à tous les membres de l'équipe et plus précisément mes collègues du bureau 502-303 du bâtiment Blaise Pascal avec qui nos relations ont été sympathiques et cordiales. Je pense à **Alida Esper**, **Francis Waidrago**, **Yang**, **Zee**, **Guy.**

Que mes collègues de la FSEA, puissent également recevoir l'expression de mes meilleurs sentiments. Une pensée spéciale à **Djibet Mbaiguessé**.

Mes remerciements s'adressent également à l'ensemble des services de l'INSA, qui permettent à chacun de travailler dans les meilleures conditions. Je pense particulièrement à **Valérie Lebeye.**

Ces années n'auraient pas été possibles sans le soutien permanent et les prières de ma famille : **papa**, **maman**, **etc.** Vous m'avez soutenu malgré la distance qui nous sépare. Je vous aime tous**.**

A Dieu le tout puissant, Créateur du ciel et de la Terre.

Merci à tous.

Sommaire

Liste des Figures

Liste des Tableaux

Chapitre 1 Introduction

En général, les environnements actuels utilisent une politique basique appelée *best effort* pour contrôler l'exécution des applications. Cette politique de gestion est générique et indépendante de toute application. Si les spécificités des applications ne sont pas prises en compte, l'exécution des applications n'est pas garantie dans les conditions pour lesquelles elles ont été conçues.

Des applications distribuées diverses ayant des besoins en ressources différents, qui s'exécutent dans un environnement réparti composé de plusieurs nœuds de caractéristiques logicielles et matérielles spécifiques, sont en compétition pour l'utilisation des ressources dans le système. Ces ressources étant en quantité limitée, un support d'exécution est nécessaire pour contrôler l'utilisation des ressources par les applications. La conception des applications pose de réels défis quand les environnements dans lesquels elles sont déployées ne sont pas toujours adaptés à leur fonctionnement. Les développeurs d'applications doivent non seulement se focaliser sur l'aspect métier des applications mais aussi prendre en compte les aspects non-fonctionnels des applications, par exemple la disponibilité des ressources ou la sécurité, pour que l'application s'exécute dans les meilleures conditions. La gestion de l'ensemble de ces aspects définit la manière dont les applications se comportent face aux variations de l'environnement d'exécution.

La qualité de service en abrégé QoS est définie comme un ensemble de contraintes de qualité lié au comportement collectif d'une ou plusieurs applications [Int96]. Tant que les ressources sont disponibles, le comportement des applications conduit à fournir le meilleur service possible. Mais quand les ressources sont rares, le comportement doit être modifié pour réduire la consommation de ressources et fournir un niveau de service acceptable.

1.1 Contexte

Cette thèse s'est effectuée au sein du laboratoire LIESP (Laboratoire d'Informatique pour l'Entreprise et les Systèmes de Production). L'un des thèmes de recherche dans ce laboratoire concerne les architectures distribuées et collaboratives dont le but est de gérer par auto-adaptation les ressources d'applications distribuées et/ou embarquées, d'optimiser l'utilisation de ressources de production, d'adapter conjointement l'infrastructure informatique (réseau et SI) et les proces-

sus métiers reflétant l'organisation de l'entreprise. Dans ce thème, un des axes concerne les systèmes autonomes industriels soumis à des contraintes. Dans ce cadre, notre équipe a mené plusieurs travaux qui ont en commun une gestion dynamique des ressources, la maximisation de la QoS fournie par le système géré, et le contrôle d'applications dont le comportement est adaptable. La plupart de nos travaux introduisent un logiciel intermédiaire appelé communément *middleware*, entre les applications et le système sous-jacent (système d'exploitation, réseau…) pour contrôler l'utilisation des ressources. Dans ce contexte, les applications distribuées sous le contrôle du *middleware* s'exécutent en fournissant la meilleure QoS possible selon la disponibilité des ressources.

Cependant, les expériences passées se sont basées sur des architectures centralisées dans lesquelles une seule entité contrôle l'utilisation de toutes les ressources de l'environnement. Bien que les architectures centralisées soient efficaces, elles souffrent néanmoins de la fragilité du système et du goulot d'étranglement potentiel constitué par le gestionnaire unique. L'enjeu est donc de proposer et d'analyser une approche totalement décentralisée dans laquelle le contrôle est réparti entre des entités coordonnant leurs décisions afin de maîtriser la QoS des applications.

1.2 Objectif de la thèse

Dans cette thèse, notre objectif est de proposer un système autonome et décentralisé pour la gestion de la QoS par adaptation du comportement des applications en fonction de la disponibilité des ressources.

Les applications gérées doivent être capables de modifier leur comportement et d'évoluer pour s'adapter à leur environnement. Dans leur fonctionnement, les applications doivent présenter un certain degré de liberté. En s'appuyant sur la connaissance des informations sur l'environnement et sur les applications, la répartition des ressources entre les applications doit maximiser l'utilité globale du système, et par voie de conséquence optimiser l'utilisation des ressources.

Dans la recherche de notre solution QoS, la première étape consiste à décentraliser une architecture centralisée existante [VSM05] et à évaluer les conséquences d'une planification décentralisée. L'étape suivante consiste alors à proposer une approche qui réduit le plus possible le nombre de messages échangés dans le système.

1.3 Organisation du document

Ce mémoire est organisé en chapitres de la manière suivante :

Le chapitre 2 définit un certain nombre de notions et introduit la problématique générale de la QoS. Dans ce chapitre les différentes approches et architectures pour la gestion de la QoS seront explorées et nos différents choix seront précisés.

Le chapitre 3 compare trois architectures basées sur la planification de l'utilisation des ressources dans un système pour la gestion de la QoS. Il s'agit d'une architecture centralisée et de deux variantes d'une architecture décentralisée. Selon le niveau de centralisation/décentralisation, les principales fonctionnalités sont présentées, le critère principal de comparaison étant le nombre de communications. Cette étude justifie notre première motivation.

Le chapitre 4 propose une solution complètement décentralisée pour la gestion de la QoS basée sur un schéma d'emprunt de ressources. Ce chapitre est une solution au problème soulevé lors de notre étude au chapitre 3 : comment minimiser le nombre de communications dans un système décentralisé ?

Le chapitre 5 utilise un modèle générique d'applications pour proposer un support pour l'adaptation des applications distribuées. Des stratégies d'adaptation sont construites au-dessus de notre système d'emprunt et des heuristiques montrent comment maximiser l'utilité globale du système.

Au chapitre 6, la technologie adoptée et l'environnement de simulation sont présentés. Des règles d'implémentation et les conditions de simulations sont précisées. Ce chapitre montre la faisabilité de nos propos. Les résultats de nos simulations sont présentés et discutés.

Pour finir, le chapitre 7 conclut ce mémoire en récapitulant le travail réalisé et propose des perspectives pour des travaux futurs.

Chapitre 2 Notions, Concepts de Base et Etat de l'Art

Les applications s'exécutent dans un environnement en constante évolution. D'un coté, les applications distribuées sont de plus en plus complexes et ont besoin de plus en plus des ressources pour fournir leurs services. D'un autre coté, le nombre croissant des applications qui s'exécutent simultanément fait que la disponibilité des ressources dans l'environnement varie rapidement dans le temps. Ces modifications continuelles du contexte peuvent perturber le fonctionnement des applications. Par exemple, la bande passante disponible peut perturber le fonctionnement d'une application de visioconférence, ce qui se répercute sur la satisfaction de l'utilisateur et par conséquent, modifie la qualité de service (QoS) perçue par ce dernier. Un faible débit entraîne la lenteur des transmissions : voix de l'interlocuteur pas nette, dégradation de l'image vidéo, non continuité de la vidéo. Dans ces conditions, il est difficile d'exécuter les applications qui n'acceptent pas de fluctuations de leur consommation de ressources. Une solution est de prendre en compte les évolutions du contexte lors de l'exécution des applications.

Les systèmes informatiques utilisent pour la plupart un système basique de gestion des ressources appelé communément *best effort*. Selon cette stratégie d'allocation des ressources, les applications obtiennent les ressources qu'elles demandent si celles-ci sont disponibles indépendamment des besoins des autres applications. Si les applications s'exécutent dans un environnement où les ressources sont toujours disponibles, le problème de gestion des ressources ne se pose pas. Malheureusement, les ressources comme le temps processeur, la mémoire ou la bande passante sont disponibles en quantité limitée et les applications sont en compétition pour leur utilisation. En plus, les contraintes en termes de ressources sont variables d'une application à l'autre, ce qui fait que cette stratégie basique de gestion de ressources ne peut convenir à toutes les applications.

Ainsi, la politique *best effort* a l'inconvénient de gérer les ressources indépendamment des besoins des applications. Avec cette politique, lorsque les ressources viennent à manquer les applications subissent des dégradations non maîtrisées et il est difficile de prévoir leur comportement ou de savoir lesquelles seront perturbées par tel ou tel manque de ressources. En plus, pour libérer des ressources il

serait très difficile de choisir quelles applications annuler ou dégrader. Par conséquent, les conditions d'exécution donc les services rendus par les applications sont inconnus et imprévisibles.

Pour de nombreuses applications, la dégradation du service rendu est envisageable pourvu que cette dégradation soit maîtrisée. Prenons l'exemple d'une application de type vidéo à la demande. Regarder cette vidéo dans une résolution de 1920x1080 à 25 images par seconde suppose que les exigences de l'application en termes de ressources sont satisfaites. Cependant, en cas de surcharge, l'utilisateur peut continuer à regarder ce film si la résolution ou le nombre d'images par seconde sont réduits à un certain niveau. Par contre il n'est pas envisageable pour le lecteur de regarder une séquence vidéo avec des ralentissements dans l'affichage des images entraînant une désynchronisation de l'image et du son, ou encore des arrêts intempestifs lors de la lecture. Selon la classe d'application ciblée, des mécanismes doivent donc être implémentés pour continuer à fournir un service acceptable.

Nous venons de voir que l'une des problématiques actuelles pour les systèmes informatiques est la gestion ou la répartition des ressources entre applications qui s'exécutent dans un environnement ouvert. Le domaine de la gestion de la Qualité de Service (QoS) s'attaque au problème de la répartition de ressources dans le système afin de maîtriser la dégradation des services rendus.

Dans ce chapitre, nous faisons un tour d'horizon des concepts, des notions et des approches autour de la gestion de la QoS afin de proposer un système autonome distribué pour cette gestion. Un système autonome est vu comme un système capable de modifier son comportement et d'évoluer pour s'adapter à son environnement. L'autonomie est construite en réduisant le plus possible l'intervention extérieure, le plus souvent l'intervention humaine. L'autonomie se retrouve dans le processus de prise de décision qui permet l'adaptation du comportement des applications et l'optimisation de la QoS.

Dans la suite, les définitions autour de la notion de QoS sont précisées (sections 2.1.1). Puis, différents problèmes dans le domaine de la QoS sont décrits (section 2.1.2), ce qui permettra de situer notre travail et d'aborder la gestion de la QoS dans le cadre général (sections 2.2). Ensuite, comme moyen de gestion de la QoS, la notion d'adaptation et différentes approches seront présentées (section 2.3). Enfin, après avoir présenté les différentes architectures de gestion de la QoS existantes (section 2.4), nous terminons ce chapitre par une conclusion (section 2.5).

2.1 Définition et Problématique Générale de la QoS

2.1.1 Notion de QoS

Initialement, la notion de QoS a été utilisée dans le cadre de la performance des réseaux basés sur l'utilisation des routeurs. Elle était définie comme l'aptitude à garantir un niveau acceptable de perte de paquets, de bande passante, du débit, défini contractuellement, pour un usage donné [Int95]. Dans cette perspective, la QoS a pour objectif d'optimiser l'utilisation des ressources d'un réseau et de garantir de bonnes performances aux applications. Mais dans la pratique cette définition ne fait pas l'unanimité et n'est pas applicable à tous les domaines. De ce fait, elle a donné lieu à plusieurs variantes en fonction du domaine [Jun03].

La QoS est le terme utilisé pour représenter l'ensemble des contraintes imposées par un usager sur la performance d'une application lors de son exécution [Siq98]. Cependant, la performance n'est pas le seul critère à prendre en compte pour caractériser la QoS. Cette définition doit être étendue à d'autres propriétés. Ainsi, pour pouvoir fournir les services demandés (aspects fonctionnels) de manière satisfaisante, une application a besoin de gérer des aspects complémentaires (non-fonctionnels) comme le type de communication, la gestion d'état partagé, la sécurité etc. Pour cette raison, certains auteurs définissent la qualité de service comme les propriétés autres que le comportement fonctionnel d'une application [Sch94]. Par exemple, un composant multimédia traitant et affichant des flux vidéo peut offrir à l'utilisateur différents niveaux de QoS, niveaux qui seront définis en fonction de la disponibilité de la ressource réseau, et des ressources disponibles au sein de l'environnement d'exécution du composant [Sig98]. La QoS consisterait alors à réaliser une réservation des ressources nécessaires à l'application, afin de permettre sa réalisation.

Le premier standard ISO autour de la problématique de la QoS [Int95], appelé QoS Framework, avait pour but de fournir une base commune pour spécifier les exigences et les mécanismes de QoS dans un environnement technologique. C'est un document qui décrit les moyens de caractérisation de la QoS, de la spécification des demandes de QoS et de la gestion de la QoS. Le concept de qualité de service est défini comme « un ensemble des qualités relatives au comportement d'ensemble d'un ou de plusieurs objets ». Cette définition est enrichie par [VKB+95] qui définit la QoS comme l'ensemble

des caractéristiques qualitatives et quantitatives de tout système qui a des contraintes sur le temps de réponse, l'accès à ses ressources ou encore sur la qualité du flux de données sortant nécessaires à l'accomplissement des fonctionnalités d'une application. La qualité de service est un concept orienté utilisateur. Il a pour rôle de fournir des garanties quant aux conditions d'exécution d'une application. Dans cette thèse, notre définition de la QoS va dans le même sens que [VKB+95]. Notre but est de créer un environnement dans lequel les applications s'exécutent en fonction de la disponibilité des ressources.

Dans un environnement ouvert, l'enjeu est de garantir l'exécution de l'application conformément aux attentes et en tenant compte du contexte d'exécution. Des mécanismes QoS sont nécessaires [DR01][GHR+04][LY97] pour rendre le système prédictible. Ce support QoS peut faire face à un ou plusieurs aspects du problème dans le domaine de la QoS. La section suivante décrit ces aspects.

2.1.2 Problématique de la QoS

La variation des conditions d'exécution implique une variation de la qualité de service fourni par les applications. La QoS varie en fonction des besoins des applications et de la perception des usagers.

Les recherches autour de la qualité de service proposent des moyens pour garantir ou assurer un niveau de QoS. Il est très important de pouvoir garantir le niveau de QoS malgré les changements ou les variations de l'environnement d'exécution, plus précisément la variation de la disponibilité des ressources. Pour arriver à maintenir le niveau de la QoS, il faut prendre en compte ces différentes variations et pouvoir réagir pour adapter la QoS en conséquence.

Lors de la conception des applications réparties, la prise en compte de la qualité de service peut se faire sous différents aspects du problème : la disponibilité des ressources, la sécurité, la tolérance aux pannes, la persistance…

Dans cette thèse nous nous intéressons à l'aspect disponibilité des ressources dans le système, qui peut conduire à une dégradation des services fournis, à un rejet ou à une interruption de l'application. Les ressources peuvent être des ressources réseau (bande passante, latence, etc.), des ressources du système (CPU, mémoire, puissance de batterie d'un ordinateur portable etc.). La variation de la disponibilité des ressources affecte directement le fonctionnement des applications. Pour assurer un niveau de QoS donné, le système dans sa totalité doit pouvoir prendre en compte l'aspect va-

riable de la disponibilité des ressources. Il faut donc mettre en œuvre des mécanismes pour garantir l'accès aux ressources, par prédiction ou par planification de l'exécution. Une optimisation de l'utilisation des ressources rendra encore meilleur le niveau de QoS.

La gestion de la QoS implique avant tout de connaître l'environnement d'exécution et de se baser sur un ou plusieurs des aspects ci-dessus. Selon [Fou00], cette gestion met en œuvre une politique de mesure et de gestion de la qualité de service

Dans cette thèse, notre but est la maîtrise du système malgré l'insuffisance des ressources en regard des demandes des applications. Ce support QoS peut être introduit dans le système selon plusieurs approches et plusieurs politiques de gestion.

2.2 Gestion de la QoS

2.2.1 Approches de Gestion de la QoS

Le domaine de gestion de la QoS peut être organisé en deux sous catégories : l'approche de gestion de la QoS à bas niveau et l'approche de gestion de la QoS à haut niveau

- Pour la première approche, nous avons deux cas possibles. Premièrement, des mécanismes QoS sont introduits dans les protocoles de communication pour garantir l'exécution des applications dans des domaines précis. Et deuxièmement, des mécanismes sont ajoutés au système d'exploitation afin de modifier sa politique de gestion de ressources, offrant aussi des mécanismes adaptés aux besoins spécifiques (contrainte temps réel, latence etc.). En ce qui concerne la gestion ou l'accès réseau, un certain nombre de travaux ont été faits. C'est le cas du protocole POC [PMC98] qui est un service de la couche transport, indépendant de tout porteur de données spécifiques. C'est aussi le cas du protocole de réservation de ressources (RSVP) [RLS+97] ou encore des protocoles RTP, RTCP dont le premier définit un format de paquets standardisés pour dispenser audio et vidéo en temps réel [Hen97] et le second est utilisé conjointement pour surveiller les statistiques de transmission et de qualité de service (QoS) et des aides à la synchronisation de plusieurs flux. Un réseau Ethernet commuté est aussi utilisé dans [GDR05], [GDR+05] pour assurer les communications d'applications à fortes contraintes tempo-

relles. Ainsi, un mécanisme de « classes de service » permet une meilleure gestion des applications critiques, et des techniques pour l'évaluation de performances permettent de vérifier que les contraintes sont bien respectées. La stratégie de gestion est implémentée au niveau des couches réseau pour l'acheminement des messages.

- Pour la deuxième approche, une description à haut niveau est fournie [LLR+99] et [VSM05]. Les mécanismes de contrôle de la QoS sont introduits soit dans l'application, ce qui rend la gestion spécifique à chaque application au prix d'une gestion relativement lourde, soit dans un logiciel intermédiaire entre les applications et les systèmes d'applications pour rendre la solution générique. En général, cette approche permet de mieux prendre en compte les besoins réels des applications. Le comportement des applications est adapté en conséquence avec un niveau de qualité de service acceptable.

La solution que nous proposons s'inscrit dans le cadre de la deuxième approche et propose un *middleware* qui se base sur une description détaillée des besoins des applications pour contrôler leur comportement. Le comportement des applications est adapté pour fournir un niveau de service acceptable même en cas de variation des ressources disponibles.

2.2.2 Niveau de QoS

Gérer la QoS c'est gérer les propriétés non-fonctionnelles des applications. Ces propriétés ont des valeurs spécifiques selon le contexte d'exécution. Elles définissent un niveau de QoS. C'est le niveau d'exigence pour la capacité d'un système à fournir un service. Dans le domaine de la QoS, il existe trois niveaux de QoS :

- *Meilleur effort* (en anglais best effort) : c'est la politique basique des systèmes d'exploitation. Elle ne fournit aucune différenciation entre plusieurs applications et ne permet aucune garantie. Ainsi, un flux temps réel (comme le streaming vidéo) et la messagerie sont traités de la même manière. Ils sont stockés dans la file d'attente selon le principe FIFO (First in First Out). Selon cette politique, tant que les ressources exigées par l'application sont disponibles, le niveau de QoS fourni est satisfaisant. Mais quand les ressources viennent à manquer, des mécanismes spécifiques doivent être introduits pour améliorer la QoS fournie.

- *Service différencié* : Comme dans tout système d'exploitation, il permet de définir des niveaux de priorité aux différentes applications sans toutefois fournir une garantie. L'ajout d'un support tel qu'une politique d'optimisation des ressources par planification est nécessaire pour garantir le niveau de la QoS.
- *Service garanti* : consiste à réserver des ressources pour certains types d'applications. Les conditions sur les ressources exigées par les applications sont respectées. Dans certaines situations, des protocoles spécifiques peuvent être utilisés. Par exemple, RSVP (Resource reSerVation Protocol), qui est un protocole de signalisation pour allouer dynamiquement la bande passante aux applications orientées réseaux. Il est très utile pour les applications multimédia.

Tous les systèmes d'exploitation satisfont les deux premiers points. Notre solution se situe entre le *best-effort* et le *service différencié*. Elle se substitue au *best-effort* pour fournir un niveau de QoS acceptable. La planification est utilisée pour contrôler l'utilisation des ressources et garantir l'exécution des applications avant leur échéance. En ce qui concerne les contraintes de temps des applications, elles sont de type temps réel mou, c'est-à-dire des contraintes qui nécessitent des garanties moins contraignantes que dans le domaine du temps réel dur où elles doivent être respectées quel que soit le contexte.

La section suivante présente les manières d'exécuter et de contrôler les applications distribuées dans les processus de gestion de la QoS.

2.2.3 Modalités d'exécution et Contrôles

Les applications s'exécutent selon deux grands modèles : *flux de données périodiques* et *flux de contrôle*. Le premier modèle gère des applications à tâches périodiques. Dans [AAS00], ce modèle a été utilisé pour offrir des niveaux de services différents et dans [HWC99], des applications basées sur le modèle de *flux de données périodiques* sont prises en compte avec un mécanisme de pré-allocation de ressources pour les applications jugées critiques. Les applications multimédias sont basées sur ce modèle car des tâches identiques sont exécutées à intervalle régulier. Dans [ACH98], la plupart des applications utilisent le modèle de *flux de données périodiques*. Il est important de noter ici qu'avec cette stratégie, on peut prévoir la consommation des ressources à long terme en s'appuyant sur la prévisibilité des

exécutions. Pour le second modèle, c'est-à-dire *le flux de contrôle*, les applications réagissent plutôt à un stimulus ou à l'arrivée d'un événement qui déclenche l'exécution d'un ensemble d'opérations en totalité et dans un ordre préétabli. La loi d'arrivée des événements n'étant généralement pas connue, il n'est pas possible de prévoir la consommation des ressources au-delà de la durée de l'exécution de cet ensemble d'opérations appelé transaction. Dans [VSM05] une approche de gestion de la QoS des applications basée sur ce modèle est proposée.

En plus du modèle d'exécution des applications, il convient de choisir une approche pour contrôler les applications. Deux principales approches sont utilisées pour la mise en application des décisions au niveau du gestionnaire : le *contrôle continu* et le *contrôle discret*. Avec *un contrôle continu,* les indicateurs (disponibilité des ressources [BNB+98] [CJC+00], respect des échéances [BNB+98], etc.) sur l'état du système sont surveillés à tout moment. Si certaines variations sont observées (dépassement du seuil, rareté des ressources, disponibilité des ressources, etc.), le processus d'adaptation est déclenché. Par contre avec un *contrôle discret*, le contrôle ne peut être effectué qu'à des points prédéfinis pendant l'exécution des applications. A chacun de ces points prédéfinis, les ressources consommées par l'application peuvent être ajustées par adaptation de comportement comme dans [CSS+97] [VSM05].

En ce qui concerne l'exécution des applications, un *flux de données périodiques* peut être vu comme un cas particulier de *flux de contrôle*. En effet, une application périodique est une application qui exécute une séquence d'opérations sur l'arrivée, à intervalle régulier, d'un événement. Concernant le contrôle des applications, si les points prédéfinis dans l'application sont de plus en plus proches dans un *contrôle discret*, nous nous approchons du *contrôle continu*. Ainsi, un *contrôle continu* est un cas particulier d'un *contrôle discret.*

Dans cette thèse, la solution de gestion de la QoS proposée s'appuie sur le modèle d'exécution *en flux de contrôle* puisque c'est un modèle plus général. Le *contrôle discret* est adopté puisque les applications changent leur consommation des ressources à des points précis.

2.2.4 Stratégies de Gestion de la QoS

Une architecture QoS dépend avant tout de la manière dont les ressources de l'environnement sont contrôlées et plus précisément du rôle des applications dans cette gestion.

2.2.4.a Stratégie Non-Intrusive

Dans cette stratégie, la gestion de la QoS est indépendante des applications. C'est l'environnement d'exécution de l'application constitué du système d'exploitation pourvu des couches de communications qui implémente les mécanismes de gestion de la QoS. Tout système d'exploitation possède déjà une forme basique de gestion de la QoS. C'est le cas précisément quand le temps processeur est réparti entre les différentes tâches suivant un algorithme donné [Bla05] ou par utilisation des priorités sur des processus. L'avantage essentiel est que toutes les applications qui s'exécutent sur la même machine bénéficient des mêmes services de gestion de QoS, aucune modification n'étant faite au niveau des applications car la gestion de la QoS est à l'extérieur de l'application. En contrepartie, les fonctionnalités de gestion de la QoS sont limitées par leur caractère très général et dépendent de chaque système d'exploitation même si des couches communes sont envisageables. Pour une gestion fine de la QoS, les fonctionnalités du système d'exploitation peuvent être spécialisées ou il suffit de lui en ajouter d'autres. Dans [AAS+02] les auteurs ajoutent des entrées pour la gestion de la QoS des serveurs Web. Ainsi, les pages Web hébergées sont affichées en modifiant dynamiquement leur contenu (en réduisant la taille de la page par exemple) selon la charge du système.

Une autre façon de mettre en œuvre cette stratégie est de confier la gestion de la QoS à un logiciel médiateur entre les applications et le système d'exploitation. Dans ce cas, pour éviter des conflits dans la gestion de la QoS, ce logiciel médiateur doit communiquer avec le système d'exploitation afin que ce dernier lui délègue totalement cette gestion. Cette solution a l'avantage par rapport à la précédente d'être plus portable. Car, seules les interactions entre le logiciel intermédiaire et le système d'exploitation nécessitent des modifications.

2.2.4.b Stratégie Intrusive

Dans cette approche, les applications participent au processus de gestion de la QoS. Le principal avantage de l'approche intrusive est de permettre des adaptations très appropriées. En effet, seules les applications possèdent la connaissance nécessaire pour modifier de façon cohérente leurs besoins en ressources selon les disponibilités. Une application multimédia saura diminuer le nombre d'images par seconde, par exemple de 25 à 12, pour réduire sa consommation de ressources tout en fournissant un service acceptable. Si cette dégradation

était laissée à la charge du système d'exploitation nous pourrions obtenir une dégradation inacceptable, par exemple un ralentissement incontrôlé du flux d'images ou un très mauvais affichage des images. L'inconvénient de cette approche est que les applications doivent être conçues spécifiquement pour cette gestion, comme dans [HWC99].

Dans [LS01] et [SC02] une stratégie intrusive de gestion est adoptée pour les objets autonomes. Au niveau local, chaque objet s'appuie sur ses spécifications pour prendre des décisions et les mettre en œuvre, tandis qu'à un niveau global l'utilisation des ressources est optimisée en commun.

2.2.4.c Stratégie Mixte

Pour tirer parti des avantages des deux précédentes stratégies, un compromis entre la stratégie intrusive et non-intrusive est adopté. Dans cette stratégie mixte, les applications sont conçues spécifiquement et exportent leurs besoins en ressources. Chaque application expose publiquement une interface pour permettre la modification de son comportement par un gestionnaire de QoS.

Dans [CK00], un cadre général est proposé pour l'adaptation dynamique des applications en fonction des disponibilités des ressources et des préférences spécifiées par les utilisateurs. Les applications annoncent leurs différentes configurations associées à des besoins en ressources, et la configuration la plus adaptée est sélectionnée depuis l'extérieur de l'application.

Avec cette stratégie, en implémentant le gestionnaire de QoS sous forme de *middleware* qui fournit les fonctionnalités générales et réutilisables pour la gestion de la QoS et qui se sert des services offerts par le système d'exploitation, cette solution a le mérite de faciliter le portage du gestionnaire de QoS sur d'autres systèmes d'exploitation. En plus, il y a séparation des préoccupations : les fonctionnalités générales et réutilisables pour la gestion de la QoS se trouvent du coté du *middleware* alors que des fonctionnalités spécifiques sont implémentées du coté des applications.

Dans [AAS00], [SC00], [HWC99], des *middlewares* basés sur un compromis entre la stratégie intrusive et non-intrusive ont été proposés. La gestion des ressources est implémentée au-dessus du système d'exploitation et de la couche de communication. Les applications négocient les quantités de ressources qui leur sont affectées avec le *middleware* et adaptent leur comportement en conséquence.

Dans l'approche intrusive que nous adoptons dans cette thèse les applications sont spécifiquement développées pour participer au processus de gestion de la QoS. Le *middleware* détient les informations nécessaires pour gérer la QoS des applications en coopérant avec celles-ci. Les informations relatives à la QoS de chaque application sont prises en compte pendant la conception de l'application pour permettre une gestion efficace des ressources.

2.2.5 Gestion des ressources

[Iss97] définit la qualité de service comme une propriété dépendant de fonctions de gestion des ressources mises en œuvre de manière transparente pour l'application par le système. Par cette définition, nous abordons la problématique de la gestion de ressources. Cette problématique cherche à régler le problème de la gestion efficace et/ou optimale des ressources en quantité limitée et la garantie d'accès à ces ressources.

L'exécution des applications réparties dans un environnement nécessite une gestion de ressources adaptée pour offrir aux utilisateurs des garanties de performance [AGR98]. Cette gestion doit prendre en compte toutes les ressources partagées par les applications. Un schéma global de gestion doit être établi, comprenant aussi bien les ressources réseau (par exemple la bande passante) que les ressources locales (par exemple le temps processeur ou la mémoire).

Le niveau de la qualité de service dépend des mécanismes mis en place pour la gestion des ressources requises par des applications. Une question fondamentale se pose : comment assurer que ces applications réparties fournissent la QoS souhaitée ou au moins une QoS acceptable ? Une solution est de réserver les ressources pour chaque application [SLR+03]. Une autre est d'adapter les services rendus aux caractéristiques de l'application afin d'obtenir un niveau de service correct [VS05] [KYO+99]. La réservation des ressources est efficace dans un environnement parfaitement maîtrisé, ce qui permet de garantir strictement la QoS des applications avec des performances optimales.

Dans un environnement ouvert, la disponibilité des ressources varie constamment. Par conséquent, il est difficile de faire des hypothèses sur l'environnement dans lequel une application répartie sera exécutée. Il devient alors nécessaire d'adapter dynamiquement une application répartie en fonction des caractéristiques de l'environnement d'exécution pour une utilisation efficace des ressources.

Avant de définir l'adaptation et de décrire les différents mécanismes mis en œuvre pour réaliser l'adaptation des applications distribuées (section 2.3), il nous semble utile de connaître la manière de spécifier les informations relatives à la QoS (section 2.2.6), de préciser les caractéristiques d'un mode de fonctionnement pour l'application (section 2.2.7) et d'examiner le choix des modes de fonctionnement des applications pour optimiser l'utilité du système (section 2.2.8).

2.2.6 Spécification des Informations QoS

Le besoin de construire des applications dont la QoS est gérée n'est pas nouveau. Cependant, pour les concepteurs d'applications ou pour les concepteurs des systèmes de gestion de la QoS, une question importante se pose : comment prendre en compte les informations non-fonctionnelles ou comment modéliser celles-ci ?

Une solution possible serait d'intégrer directement des informations QoS dans le code métier de l'application. D'un point de vue génie logiciel, cette orientation n'est pas satisfaisante puisque le code métier et le code lié à la gestion de la QoS sont mélangés. Avec cette politique, le développement d'applications devient complexe, les possibilités d'adaptation sont limitées et surtout la réutilisation de la partie métier est réduite. Une autre solution plus satisfaisante consiste à séparer les aspects métiers des aspects non-fonctionnels. Cette approche inspirée par les concepts de séparation des préoccupations [HL95] et programmation par aspects [KLM+97] nous invite à repenser le cycle de vie des applications et à considérer l'adaptation aux évolutions du contexte comme un aspect [CDD]. Un aspect est un ensemble de sous-aspects (temps de réponse, taux d'erreurs, utilisation de ressources etc.). Toujours dans la deuxième catégorie de solution, l'utilisation des profils UML [OMG07] permet de modéliser les informations liées à la qualité de service. Un profil d'UML est un ensemble d'extensions d'UML pour des besoins communs. [AFJ02] propose une extension pour la prise en compte des aspects de la qualité de service lors de la phase de conception. Les auteurs associent à chaque élément de modélisation (classe, interface, composant, objet….) un profile (*QoSProfile*) qui fournit les spécifications QoS associées à l'élément. [VSM05] a étendu les diagrammes d'activités d'UML, et a proposé une solution plus légère [OMG03], pour prendre en compte les aspects tels que le temps processeur, la mémoire, la bande passante et l'échéance.

Concrètement, dans cette thèse, il y a séparation des préoccupations. Les applications sont conçues à part en prévoyant des points de contrôle. Le middleware contient le code nécessaire pour gérer la QoS. Un profil UML contient des éléments nécessaires pour modéliser le fonctionnement de l'application en exhibant différentes possibilités d'exécution avec différents besoins en ressources. Les informations non-fonctionnelles des applications sont spécifiées dans un fichier, qui décrit l'exécution de l'application selon plusieurs modes de fonctionnement.

2.2.7 Mode de Fonctionnement et Utilité

L'adaptation de comportement d'une application est sous la supervision du gestionnaire QoS qui communique avec cette application via une interface commune. L'application doit prévoir dans sa structure des points où des choix de comportement sont proposés. Chaque choix correspond à un mode de fonctionnement et à une consommation de ressources, par exemple 12 millisecondes de CPU, 30 Mo de mémoire.

A chacun des modes, nous pouvons associer une valeur numérique appelée utilité indiquant la perception que l'utilisateur a du service rendu. Cette utilité évalue l'intérêt du choix d'un mode. Plus l'utilité est grande, plus la perception est bonne. Par exemple, nous pouvons dans l'intervalle [0..100] donner une valeur 100 à l'utilité si nous regardons une vidéo avec 25 images par seconde et une résolution de 1920*1080. Ainsi, l'utilité ne définit pas la qualité de service fournie par une application mais représente sa quantification. Dans [Int99], l'utilité est une synthèse des indicateurs tels que le respect des échéances, taux d'erreur etc.

Le système de gestion de la QoS agit sur l'utilité. Nous sommes donc face à un problème d'optimisation de l'utilité de toutes les applications présentes dans le système.

2.2.8 Optimisation de la QoS

La gestion de la QoS concerne tout le système dans lequel plusieurs applications s'exécutent. Les différents niveaux de services fournis par une application dépendent du mode de fonctionnement qui est associé à l'utilité. L'optimisation de la QoS consiste à maximiser l'utilité globale lors du choix des modes de fonctionnement, ce qui

revient au choix de la répartition des ressources entre les différentes applications.

Le problème d'optimisation de la QoS dépend des ressources et des caractéristiques QoS qui peuvent être vues comme des dimensions : le délai, le niveau de sécurité, la fiabilité etc. En général, nous avons N applications, $\{App_1, App_2, \ldots App_N\}$, en compétition pour l'utilisation de M ressources, $\{R_1, R_2, \ldots R_M\}$. Chaque application est en réalité une succession de tâches mais pour simplifier le problème, nous supposons qu'une application (App_i) est une tâche (T_i). Les ressources sont disponibles en quantité limitée, $\{R_1^{max}, R_2^{max}, \ldots R_M^{max}\}$. L'exécution d'une tache T_i est sujette à K caractéristiques QoS, $\{C_{i,1}, C_{i,2}, \ldots C_{i,k}\}$ avec $C_{i,j}$ un ensemble de choix possibles pour le choix de la caractéristique j de l'application i. Une fonction $U_{i,j}$ associe une utilité à chaque $C_{i,j}$.

Dans cette formulation, détaillée dans [LS98], les valeurs de chacune des caractéristiques $C_{i,j}$ définissent un jeu de caractéristiques qui correspond à l'un des modes de fonctionnement de la tâche T_i. L'utilité d'une tâche T_i est une somme pondérée pour chacune des caractéristiques $C_{i,j}$. Notons $U(T_i)$, l'utilité de la tache T_i.

L'utilité globale du système (U_s) est définie sous la forme générale suivante, sous la contrainte que les ressources sont disponibles [LS98] :

$$U_s = \sum_{i=1}^{N} \alpha_i U(T_i) \quad (1)$$

N est le nombre d'applications et α_i un coefficient de pondération. L'objectif est de maximiser l'utilité du système.

Les auteurs de [LS98], [Kel03] et [RLL+97], avec des formulations différentes, démontrent que le problème d'optimisation de la QoS est *NP-difficile*. Par conséquent, nous ne pouvons pas résoudre ce problème en temps réel.

Certains auteurs ont réduit la complexité du problème en se limitant à une sous-catégorie du problème général défini en (1). Dans [BNB+98] et [DG01], seule la ressource processeur est prise en compte. Dans [RLL+97], plusieurs ressources sont prises en compte mais une seule caractéristique QoS est considérée. D'autres trouvent ces simplifications insuffisantes et proposent d'introduire des heuristiques ou se contenter de solutions approchées. Dans [RLL+97], une solution sous-optimale est recherchée. Dans [AAS00] [BS11], une heuristique permet d'optimiser la QoS localement.

De toute façon, même si la recherche d'une solution de (1) est possible à un instant, le fait que dans un environnement les événements sont imprévisibles nous conduit à des optimums locaux. Et

comme la somme des optimums est différente de l'optimum global du système, il est impossible de calculer l'optimum théorique, et l'utilisation d'une heuristique reste la seule alternative envisageable.

Dans notre thèse, une bonne connaissance du système a permis l'utilisation d'une heuristique pour maximiser l'utilité du système. Des détails seront donnés plus tard.

2.2.9 Les Etapes de la Gestion de la QoS

En général, la gestion de la QoS s'effectue en deux étapes principales :

- **La phase de contrôle d'admission**. Cette étape a lieu avant l'exécution de toute application dans le système et permet de vérifier si les quantités de ressources disponibles sont suffisantes pour exécuter l'application et le cas échéant de réserver les ressources. Les exigences de l'application sont traduites en informations compréhensibles par le gestionnaire QoS (*mapping*) avant le déclenchement du processus de contrôle d'admission. Par exemple, pour une application vidéo, l'utilisateur peut exprimer ses besoins en QoS pour le son ou la vidéo en termes de qualité qui peut être très bonne, bonne, acceptable etc. alors qu'au niveau du gestionnaire QoS, cette information subjective est traduite par exemple en codec vidéo, taille de l'image, quantité de mémoire, temps processeur, etc. Si la réservation ne peut être effectuée, c'est-à-dire si un contrôle d'admission échoue, une nouvelle phase d'admission peut être débutée si des mécanismes de négociations sont implémentés. Ceci permet de vérifier si une dégradation de la qualité permettrait d'accepter l'application.
- **La phase de contrôle de la QoS**. Une fois la réservation des ressources effective, l'exécution peut commencer. Lors de l'exécution l'utilisation des ressources est contrôlée en permanence. Si une difficulté survient, une renégociation du niveau de la QoS peut être effectuée afin de continuer l'exécution d'une application. Dans certains cas, l'exécution est annulée.

Il est à noter que ces deux phases ne sont pas obligatoires dans tous les processus de gestion de la QoS. Par exemple DART [RL98] implémente des mécanismes de contrôle d'admission et des stratégies d'adaptation alors que [NQ96] n'utilise pas la phase d'admission mais adapte les applications. Notre solution utilise les

deux étapes : le contrôle d'admission a pour but de vérifier la disponibilité des ressources avant l'acceptation de toute application dans le système. La phase de contrôle de la QoS permet d'adapter le comportement des applications selon le contexte d'exécution.

2.3 Adaptation des Applications

2.3.1 Définition de l'adaptation

L'adaptation se définit comme étant la capacité d'une application à fournir un service de différentes façons [BH97]. Une application adaptable est une application capable d'adopter plusieurs comportements au regard des variations de ressources disponibles dans son environnement. Ce qui signifie que l'application définit des points où un comportement alternatif est proposé et qu'une tâche est exécutée en tenant compte des ressources disponibles.

2.3.2 Quelques Approches d'Adaptation

L'approche à base d'adaptation est utilisée car elle convient parfaitement dans un environnement où la maîtrise de la gestion des ressources n'est pas totale ou impossible. Elle a l'avantage de contrôler l'exécution de l'application conformément à la disponibilité des ressources. Les mécanismes d'adaptation ont pour objectif l'utilisation efficace des ressources de l'environnement afin de permettre aux applications de fournir une qualité de service acceptable et d'adapter leur comportement au changement des conditions du système.

[CJC+00] propose une architecture de gestion de ressources fournissant des services intégrés permettant d'adapter le comportement des applications distribuées temps réel. Dans [KYO+99] et [MPT98] des systèmes multi-agents permettent de repartir les différentes tâches de gestion de la QoS des applications multimédia distribuées. Les agents collaborent et négocient entre eux pour adapter le comportement des applications. Dans [DL02] [LN00] [LN99], l'adaptation du comportement repose sur des règles fournies au système par le concepteur. Pour [DL02], les règles permettent d'associer dynamiquement les composants fonctionnels (liés au domaine d'application) et les composants non-fonctionnels (liés aux mécanismes d'exécution ou d'adaptation) des applications. Pour [LN00] [LN99], les applications exportent une interface pour leur reconfiguration et les règles permettent de sélectionner la configuration adéquate en fonction de la valeur d'indicateurs. Dans [SA09], les auteurs

proposent un modèle pour l'adaptation dynamique qui peut être personnalisé par les concepteurs d'applications afin de satisfaire les besoins d'adaptation. Le modèle est basé sur un ensemble de fonctionnalités obligatoires et optionnelles. Les mécanismes d'adaptation sont associés aux entités telles que application, composant, service etc. [VD05] est basé sur un environnement d'exécution et/ou des compilateurs pour rendre adaptables les applications. [Lou10] décrit une approche pour l'adaptation dynamique au contexte en proposant une plate-forme pour la reconfiguration et le déploiement contextuel d'applications en environnement contraint. Cette plate-forme permet des reconfigurations des applications basées composants et exploite le plus possible les ressources de l'application en permettant d'utiliser tous les périphériques disponibles comme supports des composants logiciels de l'application. L'approche utilisée dans [VS03] est basée sur des techniques d'apprentissage pour adapter le comportement des applications.

L'adaptation n'est utile que si l'application présente des possibilités pour changer son comportement en fonction du contexte. Ces possibilités sont prises en compte lors de la construction de l'application.

2.3.3 Constructions d'Applications

Pour fournir des capacités d'adaptation, les concepteurs d'applications doivent développer les aspects métiers des applications en prévoyant des comportements alternatifs. Les applications doivent avoir un certain degré de liberté pour le choix de leur comportement au moment de leur exécution afin de prendre en compte les fluctuations des ressources pour gérer leur QoS.

Lors de la construction de l'application, toutes les possibilités d'exécution de l'application doivent être prises en compte. Certains travaux, comme par exemple [Vie05] [ASB02] [LN00] [LS01] [VSM05] [BS10b], utilisent des modèles d'applications adaptables qui proposent différents chemins d'exécution des applications en fonction des ressources disponibles. D'autres se fondent sur des environnements qui contiennent les infrastructures nécessaires pour construire des applications adaptables, par exemple dans [KC03] [VD05] [RFS02].

Concrètement, dans cette thèse, le *middleware* pilote l'adaptation des applications en se basant sur leur description. C'est le concepteur d'applications qui doit fournir, séparément du code mé-

tier, les différentes possibilités d'exécution de l'application et les besoins en ressources correspondants.

Après la prise en compte des possibilités d'adaptation de l'application, il convient de choisir un modèle d'adaptation : statique ou dynamique.

2.3.4 Modèles d'adaptation

Cette sous-section répond aux questions suivantes : comment l'adaptation est-elle spécifiée ? Et comment est-elle effectuée par les applications ?

2.3.4.a Modèle d'adaptation statique

Ce cas correspond à une adaptation effectuée avant l'exécution de l'application en fonction des connaissances détenues de l'environnement de déploiement. Il peut s'agir d'une adaptation de la gestion pour assurer un niveau de QoS (c'est-à-dire de choisir l'implémentation qui convient le mieux) ou d'une adaptation de la nature de QoS (choix de propriétés non-fonctionnelles considérées). C'est le cas par exemple dans AspectJ [AsJ] où la prise en compte de certains aspects est faite à la compilation. Dans [DHT+98], le fait de choisir d'implémenter les interfaces dédiées fait que le type de communication est choisi (c'est-à-dire adapté) statiquement en fonction des besoins de l'application.

Dans le standard des EJB (Enterprise Java Bean) [EJB], c'est également avant le déploiement des composants appelés *beans* que le constructeur décide de propriétés comme la persistance. Dans la même catégorie nous mettons les approches où l'adaptation est faite pendant une phase d'initialisation.

2.3.4.b Modèle d'adaptation dynamique

Cette solution consiste à prendre en compte pendant la construction de l'application les différentes possibilités d'exécution de l'application. La spécification des modes de fonctionnement est faite de manière statique (avant le lancement) mais les adaptations sont effectuées dynamiquement.

C'est l'approche utilisée dans le cas des applications réparties classiques où les constructeurs utilisent un protocole pour la gestion d'un problème relié aux fluctuations de l'environnement. Le choix est statique alors que le fonctionnement de ce protocole prend en compte des informations dynamiques. Dans [SAW94] l'approche a été choisie pour effectuer des adaptations en fonction du contexte où

des règles spécifiées statiquement guident les paramètres d'exécution. Dans [NSN+97] l'approche proposée concerne les systèmes mobiles. Elle met en place des solutions choisies statiquement de gestion de la déconnexion. Dans cette même catégorie se trouvent les projets utilisant le principe de la réflexivité permettant un choix dynamique entre les implémentations existantes comme dans DART [RL98].

Cependant, l'adaptation dynamique peut être plus complexe. Elle peut aller encore plus loin et permettre non seulement d'effectuer de l'adaptation pendant l'exécution mais aussi la définition et la mise en place dynamique de la stratégie d'adaptation.

Dans la solution que nous proposons, les comportements sont spécifiés statiquement par le concepteur d'applications mais la décision d'adaptation est prise dynamiquement par le *middleware* en choisissant un mode opératoire approprié pour chaque application gérée.

Apres avoir fait un tour d'horizon des notions et concepts utilisés dans cette thèse, nous parlons dans la suite de l'ossature sur laquelle se base notre approche. La section suivante décrit les différentes architectures.

2.4 Architectures QoS

Les stratégies de gestion de la QoS sont requises dans tous les domaines du système d'exploitation. Cela comprend l'ordonnancement des tâches pour la consommation du CPU, la communication, la gestion des périphériques et la gestion de la mémoire. Un ensemble de mécanismes de gestion de ressources regroupés autour d'un même objectif qui est la gestion de la QoS est appelé architecture de gestion de la QoS [Vil02].

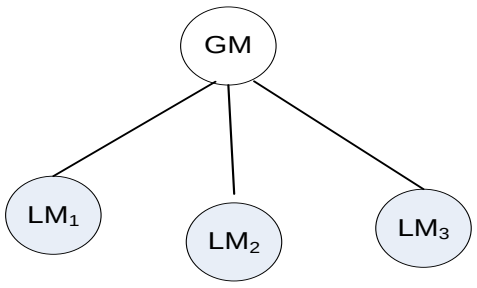

Figure 2-1: Architecture Centralisée

2.4.1 Différentes architectures QoS

L'architecture d'un système de gestion de la QoS a un impact sur la QoS fournie par le système entier. Une fois l'application et les ressources caractérisées, il est nécessaire de choisir une architecture pour mettre en place la solution QoS. Cette architecture guide le développement de tous les composants du système. Le degré de centralisation de cette architecture est important du point de vue de l'efficacité, de la tolérance aux pannes, ainsi que de la possibilité du système de passer à l'échelle. Nous présentons dans la section suivante un bref aperçu des différentes architectures qui sont utilisées pour mettre en place une solution QoS.

2.4.1.a Architectures centralisées

La majorité des systèmes sont conçus en suivant l'architecture centralisée (par exemple [VS04] [SC02] [LS01] [VSM05]). Dans cette architecture (Figure 2-1), le système global est contrôlé par une seule entité appelée gestionnaire global. Ce gestionnaire se trouve sur un nœud particulier (*GM*, Figure 2-1). Sur les autres nœuds, se trouvent des gestionnaires locaux (*LM*, Figure 2-1) qui sont sous le contrôle du gestionnaire global. Cette architecture est performante et efficace. De plus, le temps de latence est réduit. Cependant, pour des grands systèmes qui traitent d'énormes quantités de données, les principaux inconvénients sont la fragilité et le goulot d'étranglement dû à la présence de cette unique entité qui constitue un point critique pour le système. En effet, si cette entité tombe en panne, l'ensemble du système est inutilisable. En plus, il faut suffisamment de ressources au gestionnaire global pour traiter simultanément un grand nombre de connexions ou requêtes et satisfaire toutes les demandes.

2.4.1.b Architectures semi-centralisées

Une architecture semi-centralisée est caractérisée par la présence d'un agent central qui sert d'intermédiaire entre les autres nœuds (*GM*, Figure 2-2). Dans les systèmes pair-à-pair tels que les grilles de calcul, il est appelé agent de négociation. Un cas particulier d'architecture semi-centralisée est l'architecture trois

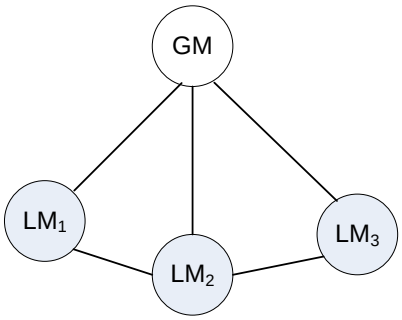

Figure 2-2: Architecture Semi-Centralisée

tiers [AAB+02], [CDF+04] où les nœuds sont séparés en deux catégories : les *clients* et les *serveurs*. Les *clients* soumettent les calculs dans la grille par l'intermédiaire de l'agent central et attendent les résultats. Les *serveurs* sont des ressources de calcul. Dans [AAB+02], les serveurs exécutent des services particuliers (fonctions mathématiques par exemple).

Dans [CDF+04], ils sont en mesure d'accepter les applications transmises par le client et l'agent médiateur est chargé de traiter les requêtes des clients. La principale différence avec l'architecture centralisée réside dans les communications entre les clients et les serveurs : dans [AAB+02], les clients sont mis en relation directement avec les serveurs alors que dans [CDF+04], toutes les communications passent nécessairement par l'agent médiateur.

Dans la même catégorie, se trouvent les architectures dans lesquelles les ressources locales sont gérées par des entités se trouvant sur des nœuds alors que les ressources partagées ou globales sont gérées par une entité spécifique sur un nœud particulier.

A part le fait que la gestion globale du système n'est pas à la charge exclusive d'une seule entité, les inconvénients sont les mêmes qu'avec l'architecture centralisée. En plus, le fait d'autoriser la communication entre les nœuds peut entraîner un coût supplémentaire en termes de messages.

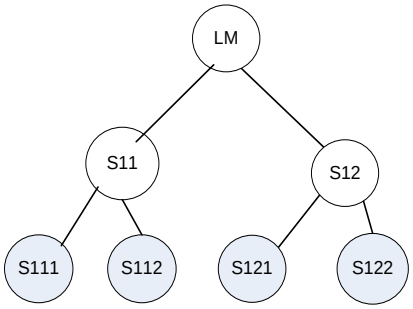

Figure 2-3: Architecture Logicielle Hiérarchique

2.4.1.c Architectures hiérarchiques

Dans une architecture semi-centralisée, l'agent central peut être un point critique, aussi bien au niveau de la tolérance aux pannes qu'au point de vue de la capacité de passer à l'échelle. Les solutions hiérarchiques (Figure 2-3) proposent de diviser le système entier en niveaux. Chaque niveau se décompose en sous-niveaux facilement manipulables. Les entités de niveaux supérieurs (par exemple, *S11* Figure 2-3) utilisent les services des entités du niveau inferieur (*S111*, *S112* dans la même figure). Les fonctions de gestion de la QoS sont décomposées en composants organisés de manière hiérarchique. Les composants enfants échangent des informations avec les composants parents pour traiter les requêtes. Dans [SA09], les auteurs proposent un modèle pour une adaptation dynamique qui peut être personnalisé par les concepteurs d'applications afin de satisfaire des besoins d'adaptation. Le modèle est basé sur un ensemble de fonctionnalités obligatoires et de fonctionnalités optionnelles. Des mécanismes d'adaptation sont associés aux entités telles que l'application, les processus, les services, les composants ou données, et chaque entité est vue comme un ensemble de composants, conduisant à une organisation hiérarchique des mécanismes d'adaptation. [CJC+00] est basée sur une architecture hiérarchique de gestion des ressources dans un environnement hétérogène. Le modèle définit une entité structurelle récursive appelée gestionnaire de service qui encapsule un ensemble de ressources et leur mécanisme de gestion. Au bas de la hiérarchie, des services fournissent des fonctions de gestion pour les ressources de base telles que le processeur et les ressources réseau et contrôlent directement l'utilisation des ressources par les composants d'application. Des services de haut niveau sont assemblés au-dessus

des services de bas niveau conduisant ainsi à une hiérarchie de services.

Une architecture hiérarchique a une tolérance aux pannes supérieure aux deux types d'architectures précédents. En plus, elle équilibre la charge. Cependant, une grande hiérarchisation peut avoir un impact sur la flexibilité du système et peut entraîner une grande distance entre les entités en haut de la hiérarchie et celles d'en bas créant ainsi un temps de latence pour certaines fonctions du système qui exigent les communications. En plus, la racine ou tout nœud contenant des sous-branches est un point de fragilité comme dans les architectures centralisées.

2.4.1.d Architectures Décentralisées

Les architectures décentralisées (Figure 2-4) sont utilisées pour tenter de remédier à certains inconvénients comme la fragilité et le goulot d'étranglement. Dans les architectures totalement décentralisées, tous les nœuds du système possèdent les mêmes capacités. Les fonctions de gestion de la QoS sont distribuées sur le réseau entre les nœuds, ce qui répartit la charge du système. Cette architecture convient particulièrement dans des environnements distribués où des composants négocient et coopèrent fortement pour atteindre un objectif.

Dans le cas des systèmes multi-agents par exemple [KYO+99] [BS10a], les agents coopèrent entre eux dans le but de fournir la QoS demandée par les applications. Des modules de contrôles sont distribués dans le système et exécutés par des agents, permettant d'avoir une maîtrise locale de la situation. Bien que cette architecture soit tolérante aux pannes, il est cependant difficile voire même complexe de la mettre en œuvre. Par exemple, dans un système basé sur la planification des ressources, les mécanismes pour le contrôle, la coordination des décisions et la synchronisation des plannings ne sont pas aisés à implémenter. En plus, la gestion du système requiert un nombre important de messages [ANA+08].

Différentes technologies peuvent être utilisées pour implémenter la gestion de la QoS. Par exemple, [FK98] utilise des objets, [MPT98] [BS10a] [HWC99] [KYO+99] sont basées sur des agents, [BCS02] [DL03] adoptent la technologie de composants, [CJC+00] hiérarchise les services etc. Chaque solution définit un modèle d'un système de gestion de la QoS et répond à des besoins spécifiques.

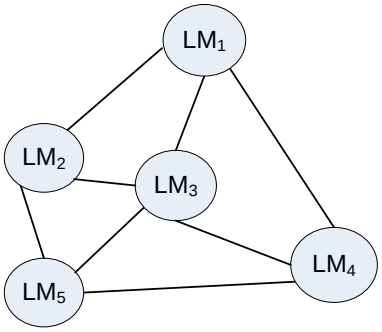

Figure 2-4: Architecture Décentralisée

En général il est souhaitable d'avoir un cadre générique pour la gestion de la QoS car selon [XXH00] :

> *"It is useful to design a general QoS management framework, which can easily change the QoS strategies and arithmetic without wholly reprogramming and redesigning for different type of applications".*

Cependant une solution générique est au détriment de la finesse de la gestion de la QoS. C'est pourquoi certaines architectures proposent des solutions pour des domaines précis. Par exemple les solutions dans [CJC+00] [KYO+99] [NQ96] gèrent des applications multimédias distribuées alors que les auteurs dans [VSM05] [BS10b] s'intéressent aux applications conçues spécifiquement.

Dans cette thèse, une architecture de gestion de la QoS à base d'agents logiciels est proposée. Le choix des agents est justifié par leur propriété d'autonomie et le caractère décentralisé inhérent à l'approche proposée.

2.5 Conclusion

A travers cette revue de la littérature, il ressort clairement le besoin pour les systèmes distribués d'avoir un support QoS pour contrôler l'exécution des applications.

La gestion de la QoS consiste à prendre en compte des aspects non-fonctionnels des applications pour fournir un service de niveau acceptable. L'aspect principalement concerné dans cette thèse est la fluctuation imprévisible de la disponibilité des ressources.

Comme moyen de gestion de la QoS, la technique d'adaptation permet de prendre en compte dynamiquement les changements du contexte d'exécution. L'adaptation permet aux applica-

tions de fournir des services en fonction de la disponibilité des ressources dans l'environnement.

Dans la dernière partie de ce chapitre, nous avons présenté plusieurs architectures de gestion de la QoS. L'architecture d'un système guide l'organisation de ses différents composants et a un impact sur la QoS fournie.

En plus du choix d'architecture, le système de gestion de la QoS dépend d'autres choix précisant le type de la solution. Tout au long de ce chapitre, nous avons précisé nos choix.

Dans le chapitre suivant, nous décrivons trois politiques de gestion de la QoS pour trois architectures différentes. Une comparaison de ces politiques est présentée.

Chapitre 3 Centralisé vs. décentralisé

L'une des principales questions auxquelles doit répondre la recherche dans l'investigation sur la problématique de gestion de la QoS concerne les choix organisationnels qui accompagnent le choix d'architecture. Plusieurs options s'offrent à elle : centraliser et/ou décentraliser.

Ce chapitre est la première étape dans l'étude que nous avons menée dans la recherche d'une solution décentralisée de gestion de la QoS. Ce chapitre est subdivisé en cinq parties. Dans la première section (3.1), nous présentons les grands choix adoptés pour la mise en œuvre de la version centralisée. Puis (section 3.2), en conservant ces choix, nous décrivons trois architectures de gestion de la QoS : une version centralisée et deux variantes d'une solution décentralisée. Ensuite (section 3.3), les différentes fonctionnalités de gestion de la QoS nécessitant des communications seront prises en compte pour estimer le nombre de messages échangés, dans chaque architecture. Enfin, avant de conclure ce chapitre (section 3.5), la section 3.4 va comparer les trois architectures en termes de nombre de messages échangés.

3.1 Choix adoptés

Cette section résume brièvement les choix adoptés dans la version centralisée détaillée dans [VSM05]. L'objectif était de gérer la QoS

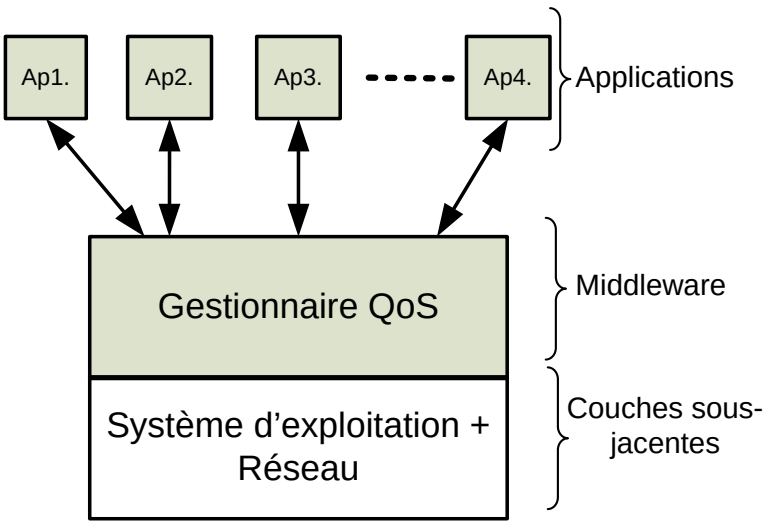

Figure 3-1: Gestion de la QoS sur un Nœud

par adaptation du comportement de l'application en fonction de l'évolution du contexte d'exécution. L'environnement est un ensemble de nœuds qui communiquent via un réseau. Les applications s'exécutent dans un contexte distribué dans lequel toutes les ressources sont déclarées : ressources de l'environnement et besoins en ressources de chaque application.

Le gestionnaire de la QoS est un intergiciel (Figure 3-1) appelé communément *middleware* s'appuyant sur des couches existantes c'est-à-dire, le système d'exploitation et les couches de communication réseau, qui fournissent les services de base nécessaires pour la gestion des ressources : processeur, mémoire, bande passante.

Les applications disposent d'une interface commune pour communiquer avec le *middleware*. Elles sont conçues spécifiquement pour être gérées par le middleware.

En se basant sur la connaissance précise des ressources de l'environnement et des besoins des applications en ressources, le middleware construit dynamiquement un planning exact d'utilisation des ressources du système. Par conséquent, le middleware doit détenir des informations sur les applications.

Chaque application est associée une description et une échéance qui fait partie de la description. L'objectif du *middleware* est d'exécuter chaque application avant son échéance. Une planification et une réservation de ressources sont nécessaires pour contrôler

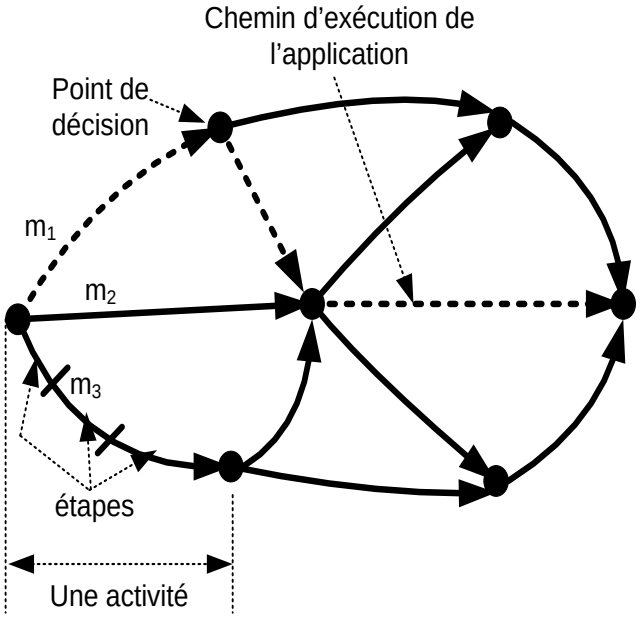

Figure 3-2: Modèle Général des Applications

l'utilisation des ressources dans le système. Le *middleware* contrôle l'exécution des applications selon le paradigme Maitre/Esclave : à la fin de l'exécution de chaque étape, l'application attend un ordre du middleware avant de continuer, ce qui permet un contrôle total de l'ordonnancement sur tous les nœuds. La sous-section suivante décrit le modèle général des applications manipulé par le middleware.

3.1.1 Modèle d'application

Chaque application est modélisée sous forme de *graphe* (Figure 3-2). Dans ce graphe, les *nœuds* sont des points de *décision* d'où partent des *arcs* qui sont des comportements possibles de l'application avec divers besoins en ressources. Un *arc* dans le graphe est une activité que le *middleware* choisit à chaque point de décision pour que l'application l'exécute. Une activité est un ensemble d'étapes exécutées séquentiellement. Chaque étape est associée à des besoins en ressources (temps CPU, consommation en mémoire ...). Chaque activité est associée à un mode d'exécution (m_i). Ce mode est un entier et est unique pour chaque activité à chaque point de décision. Choisir un mode m_i signifie choisir une activité de mode m_i pour être exécutée. Une étape est exécutée sur un nœud alors qu'une activité peut être distribuée sur plusieurs nœuds. Un point central est que les étapes d'une même activité sont exécutées selon le même mode.

Les applications s'adaptent dynamiquement à leur contexte en exécutant une activité à chaque point de décision selon la disponibilité des ressources. Un chemin dans le graphe (pointillé Figure 3-2) est une exécution de l'application. Ce chemin est une succession d'activités que le *middleware* a choisies à chaque point de décision. La consommation en ressources peut être ajustée à chacun de ces points.

Plus de détails seront donnés plus tard dans le chapitre 5 car ce modèle servira de base et sera complété pour modéliser les applications adaptables. Pour l'instant, il suffit de s'en tenir au fait que l'application propose des points où des choix de modes de fonctionnement sont possibles. Pour une application distribuée, les étapes sur des nœuds différents qui s'exécutent simultanément en utilisant une ressource partagée doivent être synchronisées. Par exemple les étapes *Transfert* et *Reception* doivent être synchronisées pour la transmission des données en utilisant une communication réseau.

3.1.2 Exemple d'application

Pour illustrer notre propos, nous donnons Tableau 3-1 un exemple d'application distribuée sur deux nœuds, *Node$_1$* et *Node$_2$*. Cet exemple simple servira de base pour compter le nombre de messages dans ce chapitre.

Tableau 3-1: Exemple d'Application

Activité	Mode	étapes	Nœud
Initialisation	1	Load	Node$_1$
		Load	Node$_2$
Acquisition	1	Acquisition	Node$_1$
Transmission	1	Transmission1	Node$_1$
		Reception1	Node$_2$
	2	Compression	Node$_1$
		Transmission2	Node$_1$
		Reception2	Node$_2$
		Decompression	Node$_2$
Display	1	Display1	Node$_2$
	2	Display2	Node$_2$

Cette application consiste à acquérir les données sur un nœud et à les afficher sur l'autre. Après le chargement des parties locales de l'application sur les deux nœuds (*Load*), l'acquisition (*Acquisition*) des images est faite sur le nœud *Node$_1$*. Après l'acquisition, les images sont transmisses (*Transfert*) au nœud *Node$_2$* pour y être affichées (*Display*). La transmission peut être faite selon deux modes : sans compression (mode 1), avec compression (mode 2). Ces deux modes de transmission permettent de définir des variations dans les besoins en ressources de l'application.

Cet exemple montre par exemple que les étapes *Transmission1*, *Réception1* de l'activité distribuée *Transmission* dans *Table 1*, sont toujours exécutées selon le même mode sur les deux nœuds. Le choix d'un mode d'exécution dépend de la disponibilité des ressources.

3.1.3 Principe de la planification

Pour contrôler l'exécution des applications, le middleware utilise une planification exacte des ressources du système. La planification utilise un algorithme possédant les propriétés suivantes :

- Planification au plus tard en fonction de l'échéance.
- Exécution au plus tôt.
- Apres l'admission d'une application, les étapes sont planifiées et les ressources réservées, ce qui garantit l'exécution.
- Les propriétés QoS (temps processeur, mémoire, bande passante etc.) sont estimées au pire des cas, Worse Case Execution Time (WCET).
- L'algorithme est une heuristique basée sur EDF (Earliest Deadline First).
- Les ressources réservées non utilisées sont disponibles pour les autres applications.

Nous supposons dans la suite de ce chapitre, que les choix sommairement expliqués dans cette section restent valables pour les autres architectures. Cependant, les mécanismes pour les mettre en

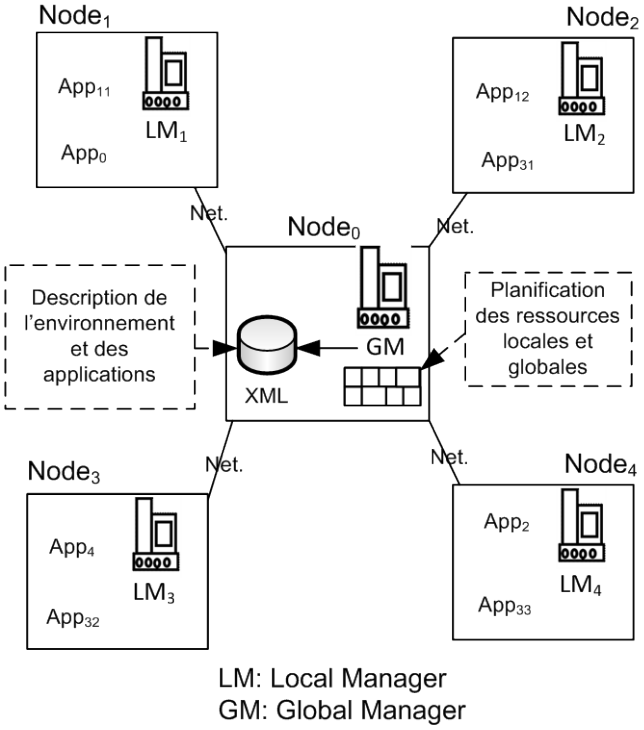

Figure 3-3: Gestion Centralisée

œuvre peuvent être différents d'une architecture à l'autre. Dans la section suivante, les principales fonctions de chaque architecture seront décrites.

3.2 Gestion centralisée et décentralisée

Cette section présente une architecture centralisée (*Centralized*) et deux variantes d'une architecture décentralisée : une architecture partiellement décentralisée (*Partial-Decentralized*) et une architecture totalement décentralisée (*Fully-Decentralized*).

3.2.1 Architecture centralisée

Dans cette architecture Figure 3-3, le gestionnaire global (*Global Manager*, *GM*), sur un nœud particulier $Node_0$, contrôle le fonctionnement du système entier. Les applications sur les autres nœuds reçoivent des ordres d'exécution par l'intermédiaire des gestionnaires locaux (*Local Manager*, *LM*) qui communiquent avec le *GM* via le réseau (*Net.*). Dans la Figure 3-3, les App_{ij} sont des parties j d'application distribuées App_i : par exemple App_{11} sur $Node_1$ et App_{12} sur $Node_2$ sont des parties d'une même application distribuée App_1 sur ces nœuds. Les App_i sont des applications locales, par exemple App_0 sur le nœud $Node_1$.

Le *GM* détient la description de toutes les applications et de l'environnement (fichier XML). Il exécute les tâches d'admission, planifie l'utilisation des ressources pour toutes les applications et contrôle leur exécution.

3.2.2 Architecture totalement décentralisée

Dans l'architecture totalement décentralisée, représentée dans la Figure 3-4, tous les nœuds sont d'égale importance. C'est une décentralisation des fonctions de gestion de la QoS assurée par le *GM* dans l'architecture centralisée (Figure 3-3). La gestion de toutes les ressources, locales et globales, est confiée à chaque gestionnaire (*LM*). En effet, en plus du planning d'utilisation des ressources locales, chaque *LM* détient une copie du planning d'utilisation des ressources globales.

Les informations sur les applications et l'environnement (fichier XML) sont stockées sur chaque nœud pour limiter les communications entre les nœuds.

Pour coordonner les traitements des applications distribuées, les gestionnaires locaux doivent communiquer entre eux afin de synchroniser leurs plannings. Etant donné que chaque gestionnaire local a une vision partielle du système, il peut communiquer avec n'importe quel autre gestionnaire local pour avoir des informations supplémentaires sur la disponibilité des ressources dans le système.

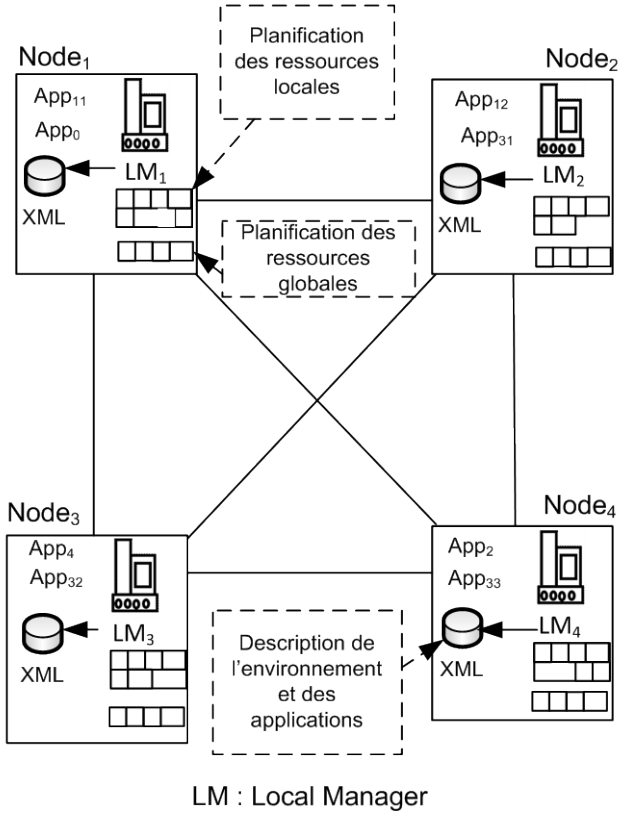

Figure 3-4: Gestion Totalement Décentralisée

3.2.3 Architecture partiellement décentralisée

Dans l'architecture partiellement décentralisée montrée Figure 3-5, le *GM* s'occupe de la gestion des ressources globales alors que les *LM* se chargent de gérer les ressources locales.

Les gestionnaires QoS, *GM* et *LM* sur leur nœud respectif, utilisent les informations sur les applications et l'environnement stockées sur leur nœud pour construire le planning d'utilisation des ressources.

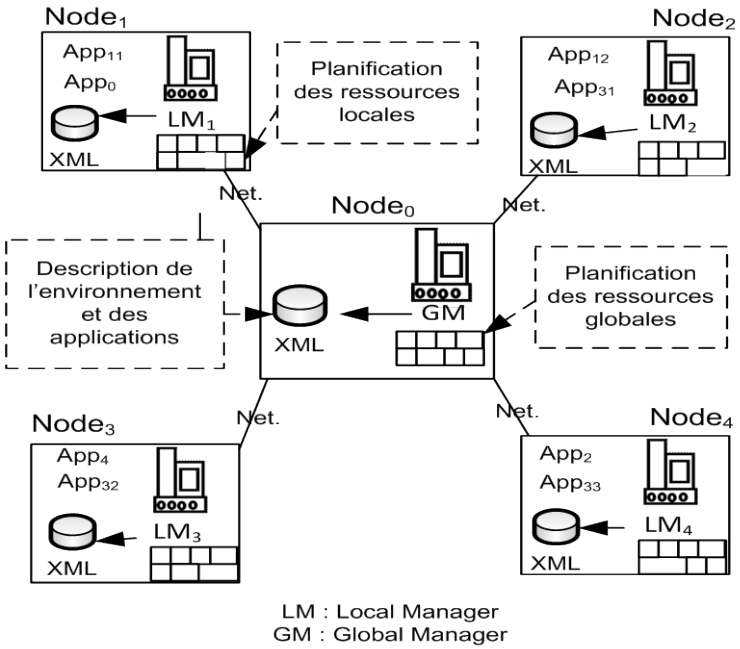

Figure 3-5: Gestion Partiellement Décentralisée

Si une application locale arrive, l'admission se fait uniquement sur le nœud concerné alors que pour une application distribuée, les *LM* des nœuds concernés communiquent avec le *GM* pour trouver une plage commune d'utilisation des ressources globales afin d'admettre l'application. Une fois l'admission effectuée, le *GM* doit mettre à jour le planning d'utilisation des ressources globales tandis que les *LM* concernés doivent mettre à jour sur leur nœud le planning d'utilisation des ressources locales.

De plus, lorsqu'une ressource globale est utilisée dans un traitement, le gestionnaire *GM* se charge de synchroniser les plannings des différents nœuds concernés.

Cette approche est un compromis entre l'approche centralisée et l'approche totalement décentralisée

3.3 Evaluation du nombre de messages

L'objectif principal de cette section est de comparer les trois architectures présentées. Pour simplifier la comparaison, nous nous limitons au compte du nombre de messages réseaux échangés dans chaque architecture. En effet, nous supposons que le coût pour gérer la planification varie peu d'une approche à l'autre et que ce coût est très négligeable devant le temps nécessaire pour l'échange de mes-

sages entre les nœuds. Mais avant cela des hypothèses sont nécessaires pour formuler le calcul.

3.3.1 Hypothèses

Nous donnons ci-après une liste d'hypothèses sur lesquelles nous nous basons pour calculer le nombre de messages dans chaque architecture :

- Dans chaque approche, la planification est utilisée pour contrôler l'utilisation des ressources par les applications. Le planning peut être modifié lors de l'admission d'une nouvelle application, ce qui peut entraîner un ou des décalages d'étapes sur un ou plusieurs nœuds (voir sous-section suivante).
- Nous supposons que les applications distribuées sont réparties sur deux nœuds. Toute synchronisation ne concerne que deux étapes sur deux nœuds différents.
- Pour simplifier la planification, seule la ressource processeur est prise en compte pour chaque nœud.
- Le coût de la planification est évalué en nombre de messages échangés entre les nœuds.
- Le décalage d'une étape synchronisée dans le planning entraîne le décalage de son étape associée et le cas échéant des étapes qui les précèdent.
- La planification des étapes d'application sur chaque nœud est faite au plus tard.
- L'insertion d'étapes synchronisées peut entraîner des décalages successifs sur les nœuds dont les étapes sont déjà synchronisées. Les décalages s'effectuent nécessairement (voir point précédent) en remontant dans le temps donc, les modifications déjà effectuées sur un nœud ne sont pas remises en cause par les nouvelles modifications. Par conséquent l'algorithme converge.
- Toutes les communications locales, entre le *LM* et les applications, sont négligées. Sont prises en compte les communications entre les *LM* : pour l'admission des applications, pour la synchronisation des étapes et pour la mise à jour du ou des plannings sur les nœuds.
- Les messages sont de type point à point. Il n'y a pas négociation entre les *LM* : il n'y a pas de réponse impliquant un nouveau message.

3.3.2 Décalage d'étapes sur les nœuds

Cette section explique comment l'admission d'une application peut entraîner des décalages d'étapes sur un ou plusieurs nœuds.

La Figure 3-6 montre un exemple des plannings pour le CPU de plusieurs nœuds et une ressource globale le réseau (*Net*). Les chiffres (*1*) à (*5*) dans les plages de temps désignent des étapes qui utilisent des ressources globales et qui nécessitent une synchronisation. Les autres étapes (*loc*) utilisent uniquement des ressources locales.

Le planning des nœuds avec lesquels les étapes (2) et (5) doivent être synchronisées avec (*2*) ou (5) n'est pas représenté pour ne pas surcharger la Figure 3-6.

L'admission de l'application représentée par son étape (3) Figure 3-6 ne nécessite aucun décalage sur les axes CPU_1, CPU_2 et *Net*. Par contre, l'admission de l'application représentée par (*4*) entraîne un décalage de (*1*) sur l'axe *Net*. Et par conséquent, les plages (*1*) de CPU_1 doivent être décalées. En plus, le décalage vers la gauche de (*1*) sur CPU_n nécessite le décalage de (*loc*) et celui de (*5*) qui lui-même etc.

Dans l'approche centralisée, comme toute la planification se fait sur un seul nœud, les décalages d'étapes sont effectués en local sur ce nœud, aucune communication distante n'est nécessaire. Dans

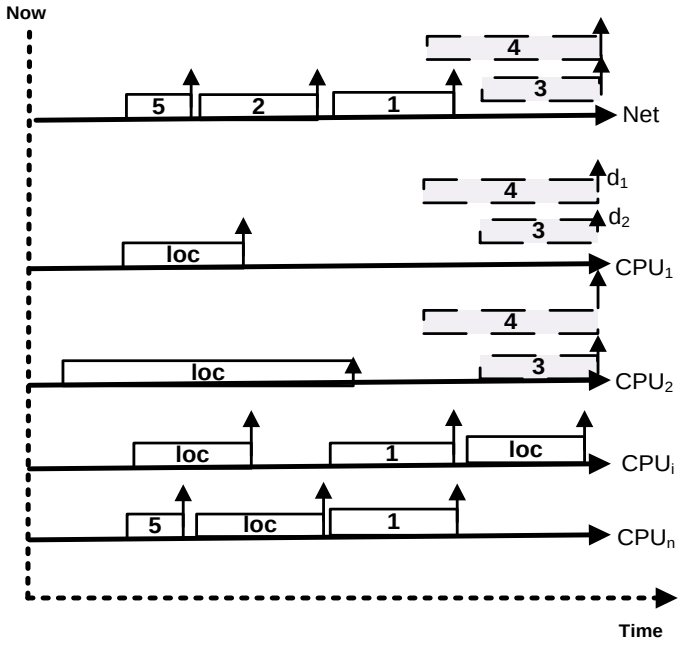

Figure 3-6: Décalage d'étapes

les autres architectures, où chaque nœud détient les plannings ou une partie des plannings, un décalage concernant plusieurs nœuds exige des communications réseau entre ces nœuds.

Dans la section suivante, nous estimons le nombre de messages pour chaque architecture. Mais avant cela, le Tableau 3-2 résume les variables utilisées :

Tableau 3-2 : Variables utilisées

Variable	Description
N_Nodes	Le nombre de nœuds dans le système
N_Steps	Le nombre total d'étapes exécutées
N_Sync	Le nombre d'étapes qui utilisent des ressources globales (celles qui sont concernées par des synchronisations)
N_Shift	Le nombre d'étapes qui utilisent des ressources globales déplacées durant le test d'admission.

3.3.3 Comptage du nombre de messages

Pour le comptage des messages, nous considérons un type d'application distribuée comme celle de la *Table* 1 et son exécution suivant un mode sur les deux nœuds dans les trois architectures.

Pour l'exécution de cette application, nous considérons deux phases : la *phase d'admission* pour décider l'acceptation de l'application dans le système et le cas échéant planifier l'utilisation des ressources, et la *phase d'exécution* pour le contrôle de l'exécution des étapes de l'application. La *phase d'admission* est constituée de deux étapes : l'étape de *déclaration* qui permet à un nœud d'informer les autres de l'arrivée d'une nouvelle application sur son nœud et l'étape de test d'*admission* qui vérifie si l'application peut être acceptée dans le système. Une application peut s'exécuter sur n'importe quel nœud du système. L'admission d'une application locale débute le plus tôt possible dès son arrivée. Pour une application distribuée, la déclaration de toutes ses parties est nécessaire pour débuter la phase d'admission le plus tôt possible.

3.3.3.a Architecture centralisée

Les messages échangés dans l'approche centralisée se calculent comme suit :

- **Phase d'admission** : à l'arrivée d'une application locale sur un nœud, le gestionnaire local (*LM*) sur ce nœud informe (1 message) le gestionnaire global (*GM*) qui procède à l'admission le plus tôt possible. Ce dernier informe le *LM* de la réussite ou pas de l'admission (1 message). Nous avons au total 2 messages pour une application locale. Pour une application distribuée sur deux nœuds, les *LM* concernés doivent envoyer chacun un message (1) au *GM* pour l'informer de l'arrivée des parties locales sur leur nœud. Le *GM* à son tour, répond en envoyant un message (1) à chacun de ces *LM*. Nous avons donc 2 messages par partie locale d'application distribuée. Ainsi, 4 messages sont échangés pour une application distribuée sur deux nœuds. Au total, nous avons 4 messages au plus échangés pour l'admission d'une application.
- **Phase d'exécution** : Dans cette approche, c'est le *GM* qui décide l'exécution de toute étape dans le système. Pour chaque étape de l'application, le *GM* envoie un message au *LM* concerné pour débuter l'exécution de cette étape. A la fin de l'exécution de chaque étape, le *LM* informe le *GM* (1 message) de sa disponibilité pour exécuter d'autres étapes. Nous avons au total 2 messages pour exécuter d'une étape.

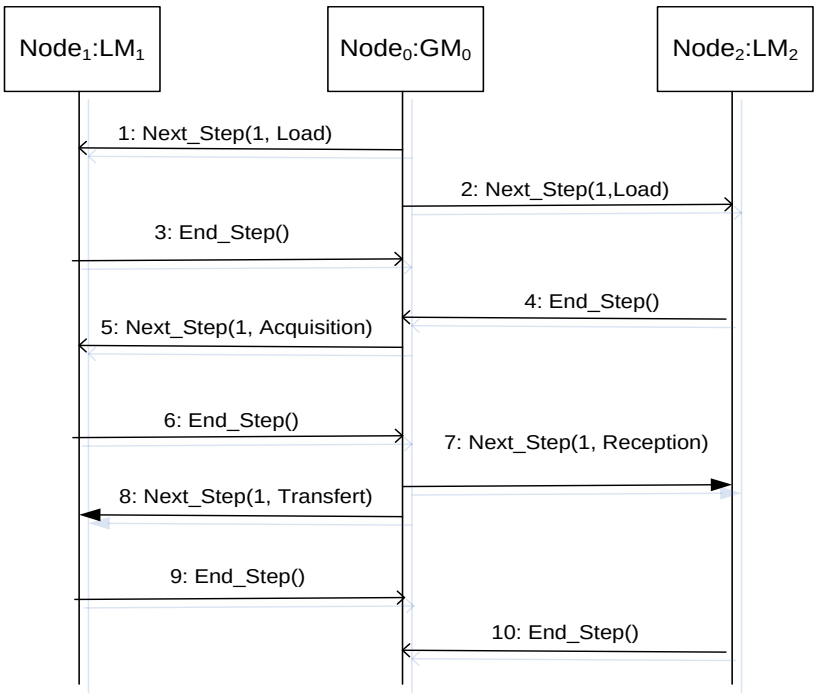

Figure 3-7: Séquence d'Exécution d'une Application

La Figure 3-7 montre les séquences d'exécution de l'application *Table* 1 suivant le mode 1 sur les nœuds $Node_1$ et $Node_2$ sous le contrôle du *GM* sur le nœud $Node_0$. Nous voyons aisément que 2 messages sont nécessaires pour exécuter chaque étape (*Next_Step()* et *End_Step()*). Si *N_Steps* est le nombre total des étapes exécutées dans le système, nous avons 2**N_Steps* messages dans la phase d'exécution.

3.3.3.b Architecture Totalement Décentralisée

Dans cette approche (Figure 3-4), chaque *LM* détient le planning d'utilisation des ressources locales et une copie du planning d'utilisation des ressources globales. L'exécution d'une étape qui utilise une ressource globale implique deux nœuds (Figure 3-6) qui doivent synchroniser leurs plannings respectifs. Pour conserver la cohérence du système, un protocole à jeton est utilisé : seul le nœud qui détient le jeton est autorisé à modifier le planning pour l'utilisation des ressources globales.

Le comptage des messages dans cette architecture se fait comme suit :

- **Phase d'admission** : Avant d'accomplir le test d'admission proprement dit, la déclaration de toutes les parties d'une application distribuée est nécessaire. Ainsi (Figure 3-8), la partie d'application qui arrive la première sur un nœud (*Part1* Figure 3-8) se déclare à son gestionnaire local (LM_1 sur $Node_1$) qui à

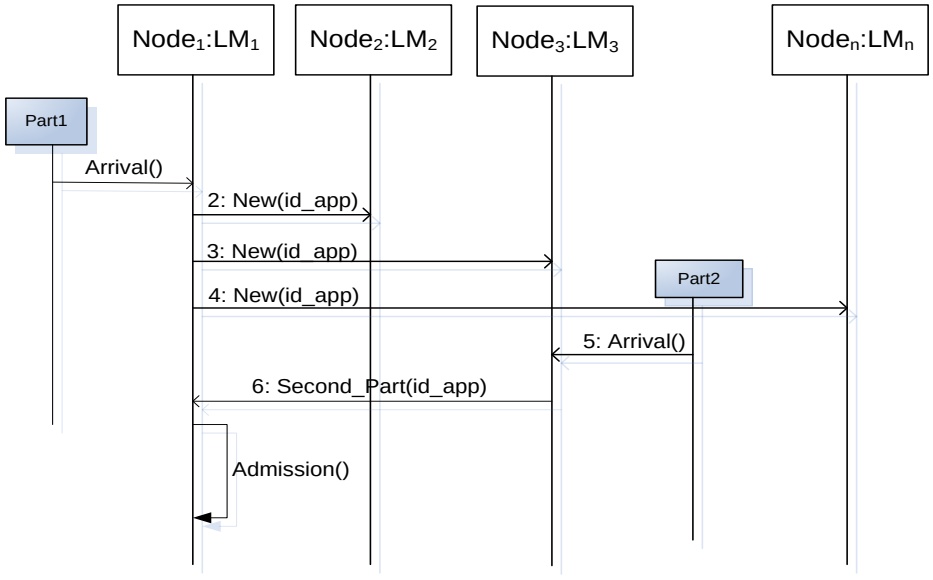

Figure 3-8: Déclaration d'une Application

son tour est chargé d'en informer tous les autres nœuds du système (messages *2*, *3* et *4* dans la Figure 3-8). Si *N_Nodes* est le nombre total de nœuds dans le système, nous avons (*N_Nodes* - 1) messages envoyés. Lorsque la deuxième partie de l'application distribuée arrive sur un second nœud ($Node_3$ Figure 3-8), après sa déclaration à son gestionnaire local (LM_3 Figure 3-8), ce dernier répond en envoyant 1 message au gestionnaire associé à la première partie (message *6* Figure 3-8). Nous supposons dans la suite que c'est le nœud qui détient la première partie locale de l'application distribuée qui est autorisé à admettre l'application. Pour procéder à l'admission, ce nœud demande d'abord le jeton en envoyant 1 message au nœud qui le détient. Ce dernier lui répond par 1 message. Ainsi, pour l'étape de déclaration, nous avons (*N_Nodes -1 + 1+1+1*), soit (*N_Nodes+2*) messages. Pendant le traitement de l'admission, la planification des étapes peut nécessiter des décalages d'étapes sur certains nœuds. Dans la Figure 3-6, le décalage de l'étape (1) sur l'axe *Net* suite à l'admission de l'étape (4) sur $Node_2$ nécessite le décalage de (1) sur les nœuds $Node_i$ et $Node_n$. Ces derniers doivent recevoir chacun 1 message (messages 1 et 3 Figure 3-9) soit 2 messages au total pour les en informer. Apres la modification de leurs plannings respectifs, $Node_i$ et $Node_n$ répondent chacun par 1 message (messages 2 et 4 Figure 3-9), soit 2 autres messages. Nous déduisons que 2 messages sont nécessaires pour chaque nœud concerné par le décalage d'une étape sur l'axe *Net* (Figure 3-9). Pour *N_Shift* étapes concernées, nous avons (*2*N_Shift*) messages au total pour les décalages d'étapes lors de l'admission. En plus, après le décalage de toutes les étapes impliquées, 1 message est envoyé à chacun des nœuds du système (messages 5 et 6 Figure 3-9) pour diffuser la mise à jour des plannings, soit (*N_Nodes-1*). Ainsi, pour la phase d'admission comprenant l'étape de déclaration et l'étape d'admission nous avons au total, (*N_Nodes+2*) + (*2*N_Shift*) + (*N_Nodes-1*) soit *(2*N_Nodes + 2*N_Shift + 1*) messages échangés entre les nœuds pour l'admission d'une application distribuée.

- **Phase d'exécution** : Dans la phase d'exécution, chaque *LM* contrôle l'exécution des étapes sur son nœud en respectant ses plannings. L'exécution des étapes qui utilisent des ressources locales ne nécessite aucun message entre les nœuds. Cependant, pour les étapes qui utilisent des ressources globales, un contrôle est nécessaire pour éviter qu'une ressource globale ne fasse l'objet de plusieurs synchronisations. Ainsi, pour chaque synchronisation des étapes, l'un de deux nœuds concernés doit au préalable demander et obtenir le jeton. La demande et l'obtention du jeton requièrent 2 messages : message 1 et 2 dans la Figure 3-10. En plus, le nœud qui a obtenu le jeton doit informer les autres pour qu'ils mettent à jour l'adresse du détenteur du jeton, soit (*N_Nodes-2*) messages. Nous supposons (Figure 3-10) que la synchronisation concerne $Node_i$ et $Node_n$ et que $Node_2$ détenait le jeton. En ce qui concerne l'exécution des étapes synchronisées sur leur nœud respectif, le nœud qui détient le jeton envoie un message au second qui répond (messages 3 et 4 Figure 3-10). 2 messages sont nécessaires pour préparer la synchronisation. Au total, nous avons 4 messages pour une synchronisation qui implique deux étapes (Figure 3-10), soit 2 messages par étape synchronisée. Si *N_Sync* est le nombre total d'étapes concernées par la synchronisation, nous avons au total au plus

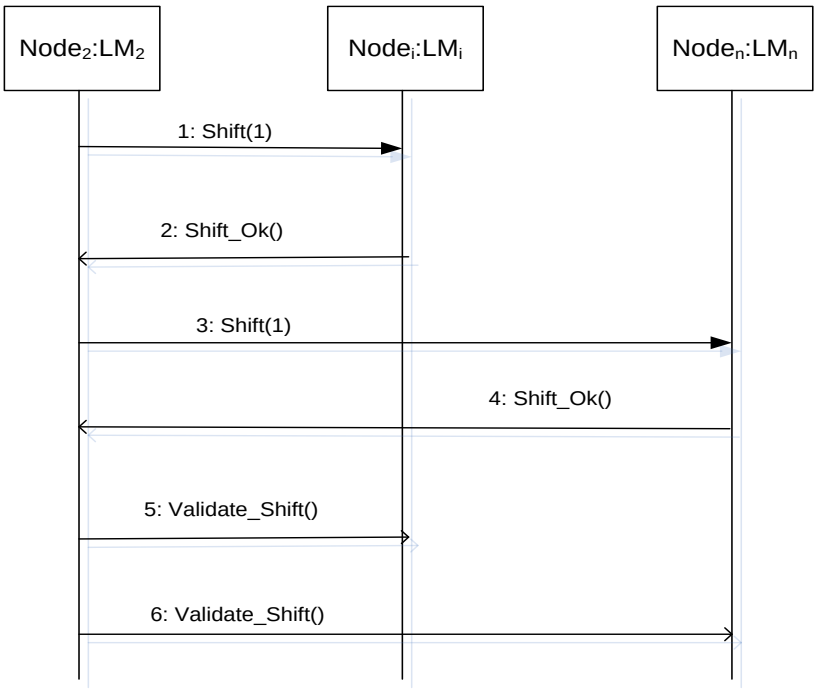

Figure 3-9: Déplacement des étapes

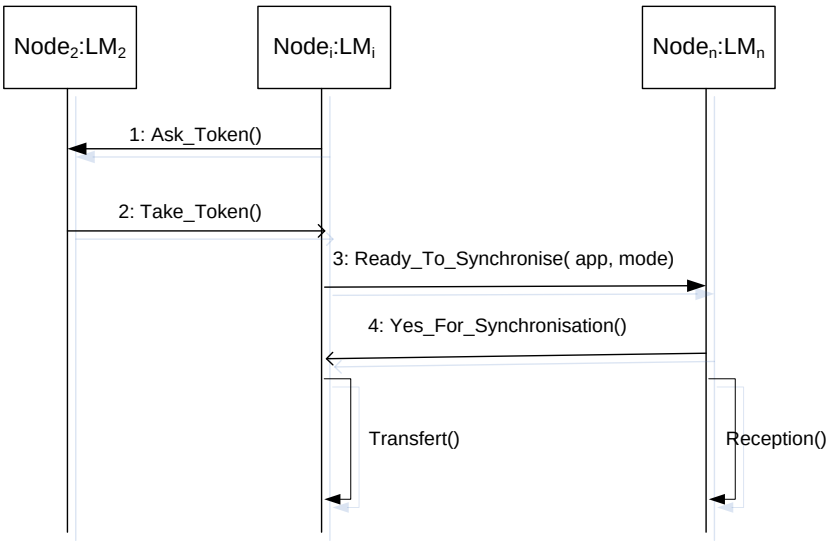

Figure 3-10: Demande du Jeton

(*2*N_Sync+N_Nodes-2*) messages échangés dans la phase d'exécution.

3.3.3.c Architecture partiellement décentralisée

Dans l'approche partiellement décentralisée (Figure 3-5), l'admission d'une application locale est faite uniquement en utilisant la disponibilité des ressources du nœud sur laquelle l'application arrive. Aucun

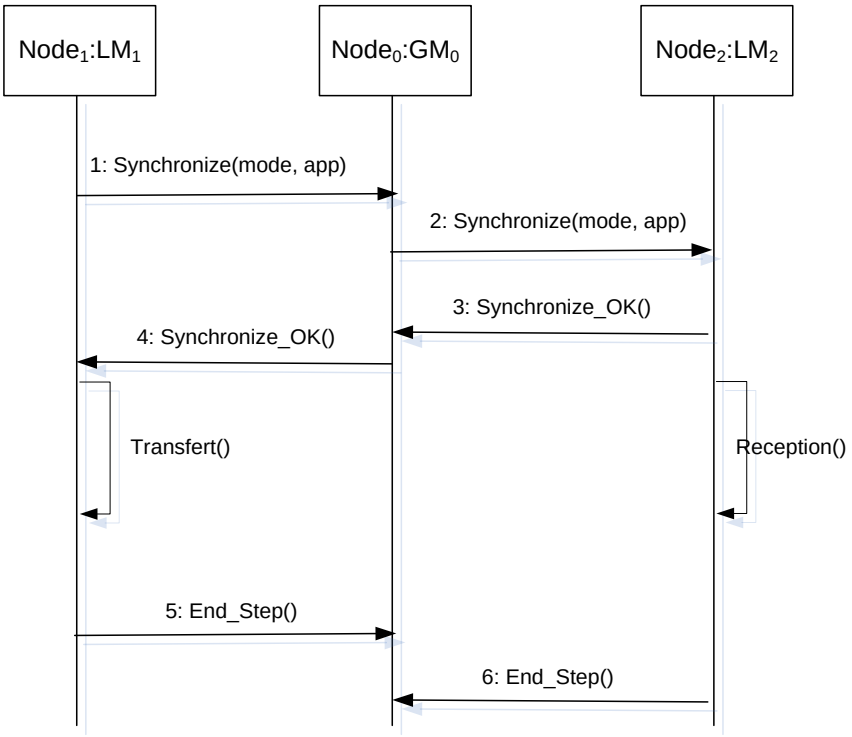

Figure 3-11: Synchronisation des étapes

message entre les nœuds n'est requis pour une application locale. Par contre pour l'admission d'une application distribuée, le planning pour l'utilisation des ressources globales doit être validé par le gestionnaire global (*GM*). En effet, chaque nœud propose des plages pour l'utilisation des ressources globales pour cette application et le *GM* choisit la plage qui convient le mieux. Les messages sont comptabilisés de la façon suivante :

- **Phase d'admission** : l'admission d'une application distribuée débute si toutes ses parties se sont déclarées à leur *LM* respectif et déclarées aux autres *LM* du système. La partie qui arrive la première se déclare à son *LM* qui, à son tour envoie (*N_Nodes -1*) messages pour informer les autres nœuds. Après l'arrivée de la seconde partie, celle-ci se déclare également à son *LM*. Après cette déclaration, 1 message est envoyé au nœud supportant la première partie locale de l'application distribuée. Ainsi, nous avons (*N_Nodes -1 +1*) soit *N_Nodes* messages nécessaires dans l'étape de déclaration. Pour le traitement de l'admission proprement dite, deux cas se distinguent. Le premier est le cas sans décalage d'étapes, comme par exemple l'étape (3) dans la Figure 3-6. Après l'admission d'une partie locale de l'application distribuée, chaque *LM* concerné envoie un message au gestionnaire global qui valide la plage commune en répondant par 1 message à chacun des nœuds concernés. Ainsi 4 messages sont nécessaires admettre une application distribuée qui n'implique pas de décalage. Le second cas concerne l'admission avec décalage d'étapes. C'est le cas de l'étape (4) dans la Figure 3-6. Le *GM* dans ce cas joue le rôle d'un *LM* qui aurait obtenu le jeton pour procéder à l'admission d'une application distribuée, dans l'approche totalement décentralisée. Dans la Figure 3-6, nous voyons que si une étape utilisant une ressource globale est décalée (par exemple l'étape 1), deux nœuds sont concernés ($Node_i$ et $Node_n$). Chaque nœud doit en être informé (1 message) et doit y répondre (1 message) soit 2 messages par nœud. Ainsi, si *N_Shift* est le nombre total des étapes concernées par les décalages, nous avons *2*N_Shift* messages dans l'étape de traitement de l'admission. A cela, il faut ajouter les (*N_Nodes-1*) messages au maximum pour la validation à la fin du traitement. Au final, nous avons *(2*N_Shift + N_Nodes -1 +4)* messages pour la phase d'admission avec décalages et 4 messages pour le cas d'admission sans décalage.

- **Phase d'exécution** : dans cette phase aucun message n'est requis pour l'exécution d'une application locale puisque celle-ci utilise uniquement des ressources locales. Pour l'exécution des étapes qui utilisent des ressources globales, les nœuds concernés doivent en être informés avant de débuter la synchronisation. Figure 3-11, le nœud qui rencontre le premier une étape qui nécessite une synchronisation ($Node_1$) envoie 1 message (message 1) au gestionnaire global ($Node_0$) pour savoir si une synchronisation est possible. Ce dernier à son tour informe le second nœud concerné $Node_2$, pour savoir s'il est prêt (message 2). Apres la réponse du nœud $Node_2$ (message 3, Figure 3-11), le nœud $Node_1$ est notifié (message 4 Figure 3-11) pour débuter la synchronisation. A la fin de la synchronisation, les deux nœuds informent le *GM* (message 5 *et 6* Figure 3-11). Nous en déduisons (Figure 3-11) que 3 messages sont nécessaires pour exécuter une étape qui utilise une ressource globale sur un nœud. Si *N_Sync* est le nombre total d'étapes qui utilisent des ressources globales, nous avons *3*N_Sync* messages au total générés par la synchronisation.

Au final, pour l'architecture partiellement décentralisée, que nous soyons dans le cas simple (sans décalage) ou complexe (avec décalage), nous avons *3*N_Sync* messages échangés dans la phase d'exécution.

3.3.3.d Résumé du nombre de messages

Le Tableau 3-3 récapitule le nombre de messages échangés dans chaque architecture :

Tableau 3-3: Résumé du Nombre de Messages

Approche utilisée	Phase d'admission	Phase d'exécution
Centralisée	*4*	*2*N_Steps*
Partiellement décentralisée (cas simple)	*4*	*3*N_Sync*
Partiellement décentralisée (cas avec décalage)	*2*N_Shift+N_Nodes+* 3	*3*N_Sync*
Totalement décentralisée	*2*N_Shift+N_Nodes+* 1	*2*N_Sync+N_Node s-2*

3.4 Résultats Graphiques

Dans cette section, nous représentons graphiquement le nombre de messages échangés dans les trois architectures décrites dans ce chapitre.

Les variables prises en compte sont le nombre de nœuds (*N_Nodes*), le nombre d'étapes synchronisées (*N_Sync*) et le nombre d'étapes décalées lors de l'admission (*N_Shift*).

La variable *N_Shift* indique la charge du système. En effet, plus le système est chargé, c'est-à-dire le planning est rempli, plus il

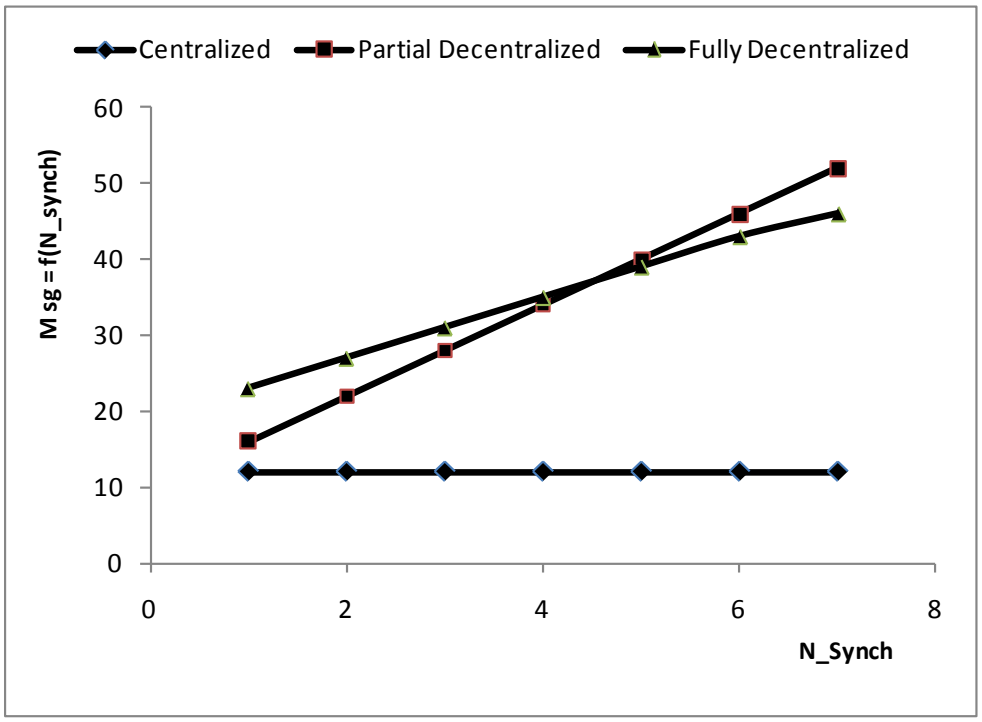

Figure 3-12: Messages échangés en fonction de N_Sync

est probable que le nombre d'étapes décalées augmente.

Les courbes Figure 3-12 montrent le nombre de messages échangés (*Msg*) dans chaque architecture pendant la phase d'exécution en fonction du nombre d'étapes synchronisées (*N_Sync*).

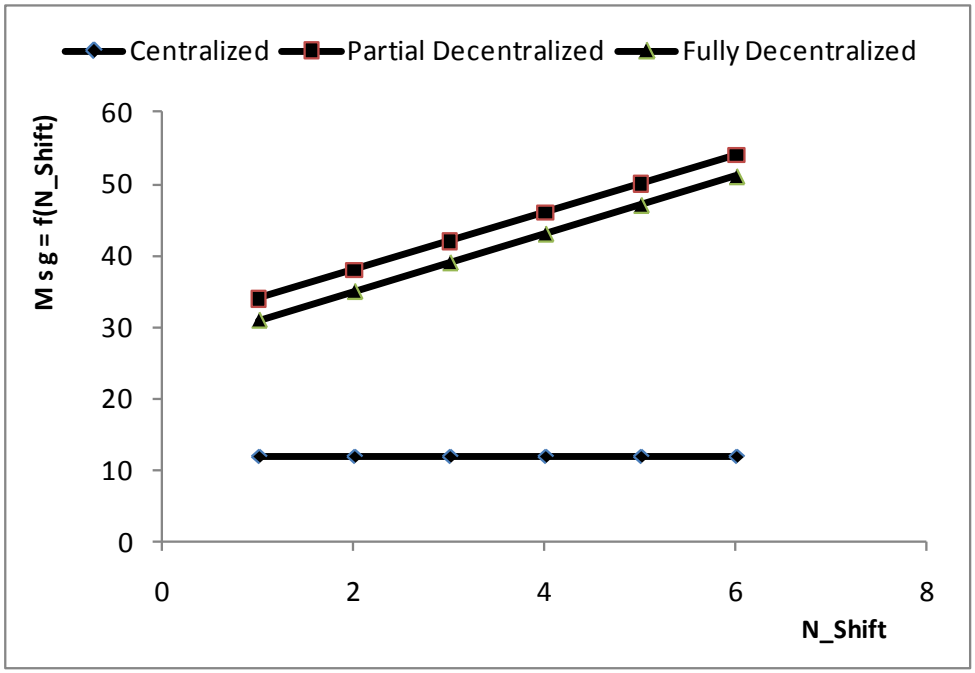

Figure 3-13: Nombre de Messages pendant la phase d'admission

La variable *N_Sync* est liée aux applications distribuées : quand le nombre d'applications distribuées croît, *N_Sync* augmente. Figure 3-12, on suppose que *N_Shift* est égal à 1, *N_Nodes* <= 6 et *N_Steps* est constant. Dans cette figure, la gestion des étapes dans les approches décentralisées (*Partial Decentralized* et *Fully Decentralized*) requiert plus de messages que dans l'approche centralisée (*Centralized*) dont le nombre de messages est constant. En effet, d'après *Table 3*, le nombre de messages échangés dans la phase d'exécution est indépendant du nombre d'étapes synchronisées si le nombre total des étapes exécutées dans le système (*N_Steps*) reste constant.

La Figure 3-13 représente le nombre de messages échangés pendant la phase d'admission en fonction du nombre d'étapes décalées (*N_Shift*) pour *N_Nodes* <= 10. Il ressort que le nombre de messages dans les deux approches décentralisées (*Partial decentralized* et *Fully decentralized*) est supérieur au nombre de messages dans l'approche centralisée (*Centralized*) qui reste encore constant. En effet, dans l'approche centralisée, la gestion des plannings est locale et effectuée uniquement sur le nœud qui abrite le *GM*. Les seuls messages échangés (*Table 2*) dans la phase d'admission concernent la déclaration des applications.

Les deux graphiques représentés dans cette section montrent que les approches décentralisées sont peu efficaces en considérant le nombre de communications comme critère de comparaison.

Comme le nombre de messages est non négligeable, le temps de latence est un autre paramètre que l'on n'a pas considéré dans cette comparaison. Dans l'approche centralisée, chaque *LM* qui envoie un message attend la réponse avant de continuer. Le *GM* gère une file des messages provenant de tous les *LM* du système. Le temps de latence doit être géré efficacement. Dans l'approche décentralisée, les traitements sont en parallèles, un message envoyé ne concerne que deux nœuds. Le temps de latence bien que réduit par rapport à l'approche centralisée, doit aussi être géré.

3.5 Conclusion

Ce chapitre a donné un aperçu de trois architectures et leur conséquence sur la planification de l'utilisation des ressources pour garantir l'exécution des applications dans le système. Nous avons comparé les trois architectures en fonction du nombre de messages échangés entre les nœuds du système.

Dans cette thèse nous souhaitons expérimenter une approche totalement décentralisée pour la gestion de la QoS. Les résultats montre le principe proposé pour la planification de l'utilisation des ressources est inacceptable. Le premier pas à franchir est de proposer une approche qui minimise le nombre de communications entre les nœuds. Le chapitre suivant décrit une solution fondée sur des mécanismes d'emprunt.

Chapitre 4 Schéma d'emprunt de ressources pour la gestion de la QoS

Ce chapitre propose et étudie une approche de gestion de ressources distribuées. Le *Scheduler* (ordonnanceur) représente un niveau d'abstraction qui décide de l'ordonnancement de tâches afin de maximiser certains critères. Ce *Scheduler* est une partie d'un gestionnaire QoS qui, dynamiquement, planifie les tâches pour maximiser l'utilité globale du système. Dans cette approche décentralisée, il y a un *Scheduler* sur chaque nœud du système. En plus, les *Schedulers* partagent des informations afin d'améliorer la QoS globale du système.

Dans ce contexte, chaque *Scheduler* doit maximiser l'utilisation des ressources à la fois locales et globales et doit réduire le coût induit par les communications entre Schedulers.

Notre but est de gérer de façon totalement décentralisée la QoS des applications. La planification est utilisée pour contrôler l'utilisation des ressources et garantir leur exécution. Les objectifs principaux de ce chapitre sont :

- Réduire le plus possible le nombre de communications entre les *Schedulers* ;
- Augmenter la performance du système en réduisant le plus possible le temps global inutilisé ;
- Une fois l'application admise et planifiée, de garantir l'exécution autant que possible.

Le chapitre est organisé comme suit : la première partie (section 4.1) décrit l'environnement d'exécution. La deuxième partie (section 4.2) présente un modèle général d'application. Ensuite, la troisième partie (section 4.3) décrit une approche totalement décentralisée, utilisant des mécanismes d'emprunts, pour la gestion des ressources des applications distribuées. Enfin, la quatrième et dernière partie (section 4.4) termine ce chapitre par une conclusion.

4.1 Environnement d'exécution

Un nœud ($Node_i$) est une entité séparée possédant au moins un processeur comme défini en UML [OMG07]. Notre environnement

(Figure 4-1) est constitué d'un ensemble de nœuds reliés par un réseau (*Net*.). Chaque nœud a ses propres ressources (Mémoire, CPU etc.). Sur chaque nœud, une entité locale appelée gestionnaire local (*LM*) gère les ressources.

Nous distinguons deux types de ressources : les ressources locales sont celles qui sont associées au nœud et gérées par un gestionnaire local (*LM*) sur ce nœud ; les ressources partagées ou globales, comme la bande passante, sont des ressources qui sont gérées collectivement par tous les gestionnaires locaux (*LM*).

L'ensemble des gestionnaires locaux sur leur nœud respectif constitue notre *middleware* chargé de contrôler l'utilisation des ressources.

Pour gérer efficacement les ressources, le *middleware* doit détenir une description complète à la fois des ressources disponibles dans l'environnement et des ressources requises par les applications pour leur exécution. Dans le processus de gestion des ressources pendant l'exécution, un protocole commun définit la manière dont les applications communiquent avec le *middleware*.

Pour qu'une application soit acceptée dans le système, une étape d'admission est nécessaire pour vérifier la disponibilité des ressources et décider si l'application peut être exécutée avant son échéance. Pour chaque application acceptée dans le système, les étapes sont planifiées et les ressources nécessaires à l'exécution sont réservées.

L'exécution d'activités d'une application sur tout nœud dans le système est décidée par le *middleware* : les *LM* donnent des ordres

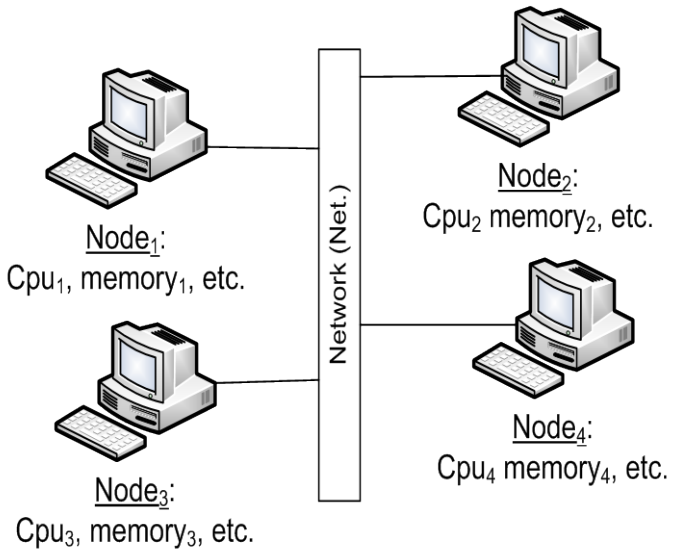

Figure 4-1: Environnement d'Exécution

et les applications les exécutent. Cette coopération impose que les applications soient conçues de manière spécifique. La section suivante précise le modèle général des applications gérées par le *middleware* dans ce chapitre.

4.2 Modèle d'Application

Pour gérer la QoS, un modèle général d'application est nécessaire. Une application est conçue comme une succession d'étapes (*Step,* Figure 4-2) dans un graphe et son exécution est un chemin dans ce graphe. A la fin de l'exécution de chaque étape, c'est-à-dire à chaque nœud du graphe, les applications informent leur gestionnaire local (*LM*) et attendent un ordre pour continuer. Chaque étape est associée à des exigences en ressources, nécessaires pour exécuter cette étape.

Une unité de planification (*Scheduling Unit*, SU_i Figure 4-2) est une séquence d'étapes associée à une échéance (*deadline*). Une application est composée d'une ou de plusieurs unités de planification. Exécuter une application c'est exécuter ses différentes unités de planification en respectant leur deadline.

Quand les étapes de différentes parties d'une application distribuée communiquent, elles doivent être synchronisées. Le premier nœud qui arrive au point de synchronisation attend l'autre pour débuter simultanément l'exécution. Par exemple dans la Figure 4-3, les parties App_{11} sur le nœud $Node_1$ et App_{12} sur le nœud $Node_2$ d'une application distribuée incluent un transfert de données. Si App_{11} et App_{12} arrivent respectivement aux nœuds *1* et *2* appelés points de rendez-vous, une synchronisation entre les étapes *Transfert* et *Recep-*

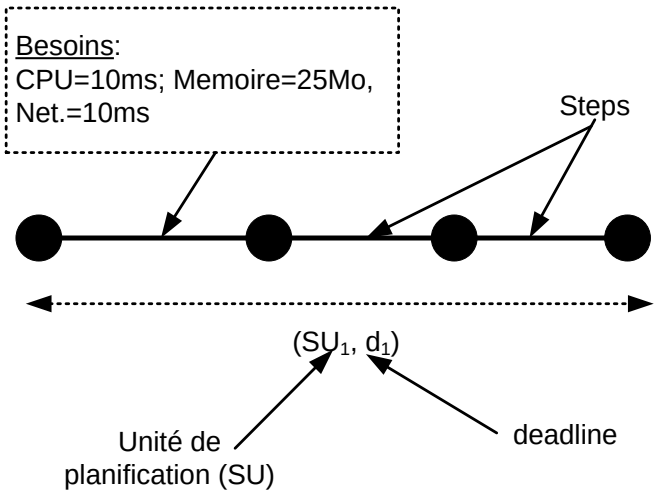

Figure 4-2: Modèle de l'application

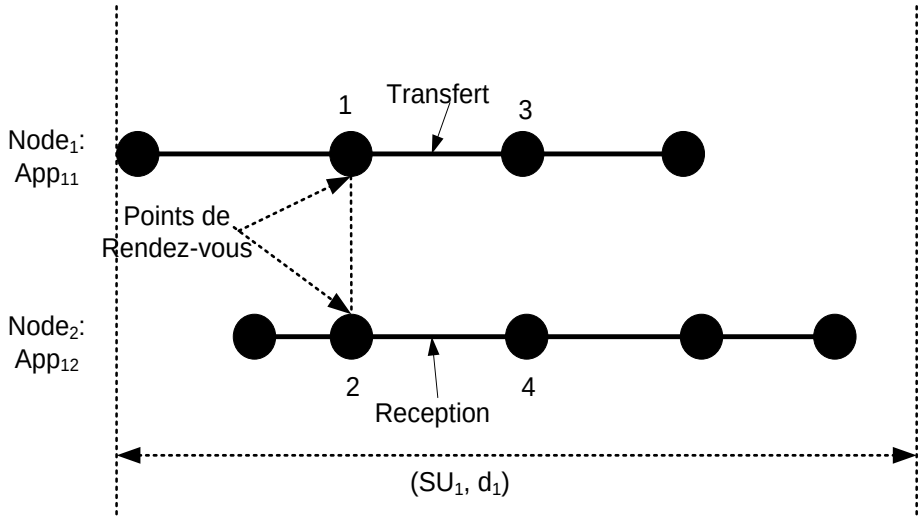

Figure 4-3: Synchronisation des étapes

tion apparaît. A la fin de la synchronisation de ces étapes, c'est-à-dire aux nœuds *3* et *4*, chaque partie de l'application distribuée exécute séparément ses étapes jusqu'à la prochaine synchronisation ou la fin de l'application.

4.3 Schéma d'Emprunt de ressources pour la Gestion des Ressources

Cette section décrit notre solution. Elle propose un schéma d'emprunt totalement décentralisé pour gérer les ressources d'applications distribuées afin d'améliorer la QoS globale du système. Ce schéma consiste principalement à gérer les ressources par pourcentages et par périodes. Il permet à chaque gestionnaire local (*LM*) de construire localement une vue détaillée de la disponibilité des ressources des nœuds distants afin d'admettre les applications et de contrôler leur exécution. Utilisant ce schéma, des décisions sont prises localement, évitant ainsi certains messages. La sous-section suivante explique le principe général.

4.3.1 Principe

Dans la suite de ce chapitre, l'utilisation de toute ressource par les applications dans le système est représentée par son temps d'utilisation sur un axe de temps. Pour simplifier le modèle, nous ne prenons en compte que le temps processeur sur chaque nœud $Node_i$ symbolisé par CPU_i, et le temps d'utilisation de la bande passante,

symbolisé par *Net*. Considérons les trois nœuds $Node_1$, $Node_2$, $Node_3$ avec leurs ressources respectivement CPU_1, CPU_2, et CPU_3 et un réseau *Net*. La Figure 4-4 représente un exemple d'utilisation des ressources sur chaque nœud selon un axe de temps *Time*.

Il est important de noter que les nœuds peuvent avoir des caractéristiques différentes, par exemple le type et/ou la vitesse du processeur peut être différent d'un nœud à l'autre. Par conséquent, le temps d'utilisation du processeur pour exécuter une étape sur un nœud peut être différente de celui pour cette même étape sur un autre nœud.

Utilisant une approche simpliste du problème, chaque nœud $Node_i$ serait associé aux ressources locales et le *LM* a uniquement une vision locale des ressources qu'il gère, c'est-à-dire uniquement des ressources sur son nœud. La disponibilité des ressources sur les nœuds distants n'est pas connue, c'est-à-dire le nœud $Node_1$ par exemple ne connaît pas la disponibilité des ressources sur le nœud $Node_2$. De cette façon, si une application locale, c'est-à-dire qui utilise uniquement des ressources locales, arrive, la connaissance des informations sur la disponibilité des ressources en local sur n'importe quel nœud est suffisante pour prendre des décisions concernant l'admission de cette application. Par contre, si une application distribuée, qui utilise des ressources sur différents nœuds, arrive, le *LM* qui gère les ressources sur chaque nœud n'a pas assez d'informations pour décider localement. Deux possibilités se présentent alors :

- Admettre l'application sans se soucier de la disponibilité des

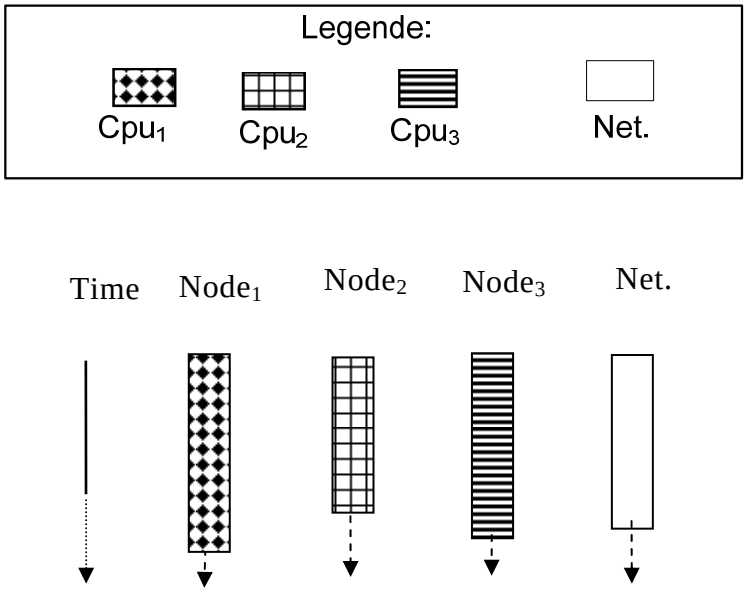

Figure 4-4: Utilisation des Ressources

ressources sur les nœuds distants. Cette stratégie ne garantit pas l'exécution de l'application car seule la vue locale est utilisée. C'est une sorte de politique basique *best effort*.

- Vérifier la disponibilité des ressources sur les nœuds impliqués avant d'admettre l'application. Cette politique a pour objectif de garantir l'exécution de l'application sur les nœuds concernés.

Dans le premier cas, aucun message n'est généré alors que dans le second des messages sont nécessaires pour vérifier la disponibilité des ressources sur les nœuds distants. Nous pourrons éviter certains messages en utilisant notre schéma d'emprunt.

Pour illustrer notre propos, supposons les points suivants :

- Le temps est subdivisé en périodes de longueur T, et sur chaque nœud l'utilisation d'une ressource est exprimée comme un pourcentage de T.
- Au début de chaque période, chaque *LM* réserve une partie (αT) de ses ressources locales et prête le reste (βT) aux autres *LM*.
- Les ressources globales ou partagées sont réparties de la même façon (σT).

La Figure 4-5 représente le pourcentage de disponibilité des ressources sur trois nœuds pour deux périodes successives avec un pourcentage de répartition égal pour la ressource partagée *Net* : chaque nœud a droit à T/3 (σ=1/3). Pour les ressources locales (CPU_i), chaque nœud garde T/2 (α=1/2) et distribue T/4 (β=1/4) aux autres nœuds.

Ainsi, chaque *LM* sur son nœud détient un quota de ressources disponibles sur les autres nœuds en addition à ses propres ressources. Chaque *LM* se base sur cette vue pour décider l'admission d'une application dans le système.

L'originalité de ce schéma est la possibilité pour un *LM* de garder en local une vue détaillée de la disponibilité des ressources dans le système. Utilisant des mécanismes d'emprunt, les *LM* construisent cette vue pour éviter des messages entre les nœuds. C'est l'un des avantages de notre schéma d'emprunt.

Les sous-sections suivantes détaillent les fonctionnalités de notre proposition. Mais avant cela, il est nécessaire de définir quelques termes ou notations utilisés dans la suite de ce chapitre.

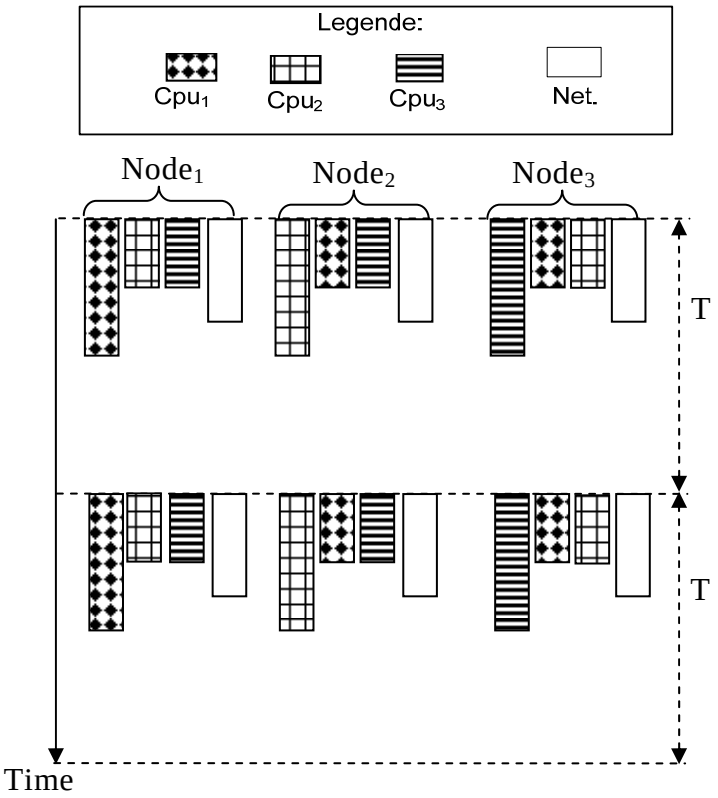

Figure 4-5: Exemple d'un schéma d'emprunt

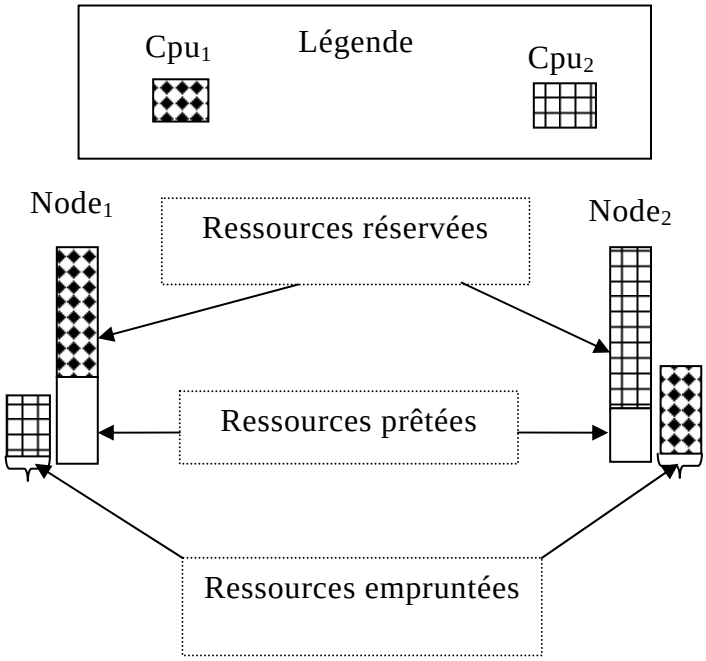

Figure 4-6: Notions utilisées

4.3.2 Quelques notations

Dans une période T (Figure 4-6), les nœuds $Node_1$ et $Node_2$ détiennent des ressources CPU_1 et CPU_2 respectivement. Utilisant notre schéma d'emprunt, sur chaque nœud, le *LM* garde une partie de ses ressources et prête le reste aux autres. Les ressources que chaque nœud prête aux autres sont appelées des *ressources prêtées*. D'un autre coté, les ressources que chaque nœud emprunte sont appelées des *ressources empruntées*.

Les *ressources réservées* sont des ressources locales qui restent après avoir prêté aux autres nœuds.

4.3.3 Emprunt Statique et Utilisation des Ressources

Pour répartir les ressources sur les nœuds, il faut une politique initiale d'emprunt de ressources. Les pourcentages des ressources locales prêtées et les pourcentages des ressources partagées à emprunter doivent être définis et connus par tous les nœuds au lancement du système. C'est la politique d'emprunt statique des ressources.

Selon notre politique d'emprunt, à chaque début de période, des ressources sont associées à chaque nœud. Les différentes ressources (Figure 4-6) sont utilisées de la manière suivante :

- Les *ressources réservées* sont utilisées par les applications admises par le nœud. Supposons qu'une application arrive. Si les quantités des *ressources réservées* disponibles sont au moins égales aux quantités des ressources locales exigées par l'application alors nous avons deux situations. Dans le cas d'une application locale, l'admission est effective. Dans le cas d'une application distribuée, c'est une condition nécessaire pour continuer le processus d'admission de l'application.
- *Les ressources empruntées* sont utilisées par les applications distribuées admises par le nœud. Elles donnent une représentation locale de la disponibilité des ressources sur les nœuds distants. Si une application distribuée, par exemple entre le nœud $Node_1$ et le nœud $Node_2$, arrive sur le nœud $Node_1$, les quantités des ressources requises par la partie de l'application distribuée qui s'exécute sur le nœud distant $Node_2$, doivent être au plus égales aux quantités des *ressources empruntées* disponibles pour continuer le processus d'admission de l'application distribuée.

Nous nous apercevons via ces deux points que les *ressources réservées* sont utilisées à la fois par les applications locales et les applications distribuées qui arrivent sur un nœud tandis que les *ressources empruntées* sont utilisées uniquement par les applications distribuées.

Pour qu'une application distribuée soit admise les deux conditions précédentes doivent être satisfaites.

Pour l'échange des données entre les nœuds, la ressource partagée, par exemple la ressource bande passante *Net*, est utilisée conjointement avec d'autres ressources. En effet, si $Node_1$ transfère des données au nœud $Node_2$, le nœud $Node_1$ utilise CPU_1 (ressource réservée sur $Node_1$) et la ressource partagée *Net*. Evidemment, le nœud qui reçoit les données, le nœud $Node_2$, utilise aussi la ressource CPU_2 (ressource réservée sur $Node_2$) et la ressource partagée *Net*.

Les mécanismes utilisés dans notre schéma d'emprunt entraînent les propriétés suivantes :

- Pour chaque ressource et pour chaque période, la somme des quantités des *ressources réservées* et des *ressources prêtées* est égale à la période T ;
- Pour une ressource partagée, la somme sur tous les nœuds du système des quantités des *ressources empruntées* est égale à T.
- Pour chaque période, la somme des durées des étapes planifiées sur un nœud est inférieure ou égale à T. En effet, il n'y a pas nécessairement une application à exécuter dans une période.

Dans notre schéma d'emprunt, chaque nœud peut définir sa propre politique d'emprunt c'est-à-dire définir son propre pourcentage d'emprunt. Dans ce cas toutes les politiques doivent être connues par tous les nœuds du système pour pouvoir les appliquer sans échange de messages. Dans la *Figure 4-5*, chaque *LM* utilise la même politique d'emprunt des ressources, c'est-à-dire, tous les nœuds empruntent la même quantité. C'est donc juste une simplification du problème.

Les sous-sections suivantes décrivent les différentes phases que le *middleware* utilise dans le processus de gestion des ressources.

4.3.4 Les phases de contrôle des applications

Dans ce schéma d'emprunt, le *middleware* gère les applications en deux phases : la *phase d'admission* et *la phase d'exécution*. Dans la

phase d'admission, le *middleware* utilise la vue de la disponibilité des ressources du système construite localement pour procéder à l'admission de l'application. Si les ressources disponibles suffisent pour exécuter l'application, celle-ci est admise, sinon elle est rejetée.

Pendant la *phase d'exécution*, le *middleware* contrôle sur chaque nœud l'exécution des applications. Dans cette phase les *LM* communiquent avec les applications ou entre eux via différents messages :

- Des *LM* aux applications : pour débuter l'exécution d'une étape, pour charger une application, pour annuler l'exécution d'une application etc.
- Des applications aux *LM* : pour signaler la fin de l'exécution d'une étape, pour informer le *LM* sur le nœud local de l'arrivée d'une nouvelle application ;
- D'un *LM* vers un autre : pour informer de l'arrivée d'une partie locale d'application distribuée, pour synchroniser des étapes d'applications distribuées sur plusieurs nœuds etc.

4.3.5 Avantage / Inconvénient de la politique d'emprunt statique

En utilisant notre schéma d'emprunt, les *LM* peuvent admettre les applications en utilisant une vue locale de la disponibilité des ressources du système. Ainsi, si la disponibilité des ressources sur un nœud est suffisante pour admettre une application, celle-ci est acceptée, sinon elle est rejetée. L'admission se fait sans messages. C'est un avantage de la politique d'emprunt statique. Pour l'instant, les nœuds n'ont le droit d'utiliser que les ressources qui ont été mises à leur disposition, c'est-à-dire les quantités des *ressources empruntées* et *réservées* selon la politique adoptée.

Si l'objectif était de réduire uniquement le nombre de messages générés, notre schéma d'emprunt statique est une solution. Cependant, nous pouvons nous retrouver dans une situation où un nœud n'utilise pas la totalité de ses ressources réservées ou empruntées alors qu'un autre en a effectivement besoin. La gestion de l'utilisation des ressources n'est donc pas efficace dans ce cas. C'est un inconvénient de notre politique d'emprunt statique. Que faire dans cette situation ?

Réduire le nombre de messages tout en augmentant l'utilisation des ressources dans le système est un compromis.

Dans la sous-section suivante, quelques éléments de réponse sont donnés dans ce sens.

4.3.6 Emprunt / prêt dynamique des ressources

Pendant *la phase d'admission*, les *LM* ont la possibilité de formuler des requêtes pour demander des ressources additionnelles. Ces requêtes sont appelées des requêtes d'emprunt ou prêt dynamique de ressources.

4.3.6.a Ressources locales

Les situations dans lesquelles les ressources locales font l'objet des requêtes d'emprunt dynamique sont les suivantes :

- Quand une application locale arrive sur un nœud, si les quantités des *ressources réservées* disponibles sont insuffisantes pour l'admettre et l'exécuter, le *LM* demande aux autres des ressources additionnelles. Les ressources concernées sont des *ressources prêtées* par un nœud aux autres. Cette demande peut être envoyée à un ou plusieurs nœuds si la quantité manquante est importante. D'une part, cette procédure peut réduire la quantité des *ressources empruntées* disponibles sur un ou des nœuds distants. D'autre part, elle peut augmenter la quantité des *ressources réservées* disponibles sur le nœud qui a formulé et envoyé la requête.
- Quand une application distribuée arrive sur un nœud, si les quantités des *ressources empruntées* disponibles sont insuffisantes pour l'admettre et l'exécuter, le *LM* peut formuler une requête pour demander des ressources additionnelles afin d'accepter l'application dans le système. Cette procédure concerne les ressources locales des nœuds distants et peut réduire la quantité des *ressources réservées* sur ces nœuds. Par conséquent, elle augmente les quantités des *ressources empruntées* disponibles sur le nœud qui a formulé la demande.
- Cette troisième situation concerne l'utilisation à la fois de deux points précédemment décrits lors de la phase d'admission. Ce cas s'applique si les *ressources réservées* et *ressources empruntées* disponibles sont insuffisantes pour admettre une application distribuée.

Par exemple, supposons qu'une application distribuée sur deux nœuds $Node_1$ et $Node_2$ arrive avec des besoins en ressources CPU_1 (α) et CPU_2 (β) respectivement. $Node_1$ procède à l'admission de l'application. Si l'application nécessite une quantité de CPU_2 supérieure à celle disponible, $Node_1$ peut demander la quantité manquante (β-σ) de CPU_2 à $Node_2$, σ est la quantité disponible de CPU_2 pour l'utilisation de $Node_1$. Nous sommes dans la deuxième situation expliquée ci-dessus.

Une requête qui augmente les *ressources empruntées* disponibles est appelée requête d'emprunt dynamique par le nœud qui formule la demande et requête de prêt dynamique par le nœud qui traite la requête. Les ressources empruntées dynamiquement peuvent faire l'objet d'une requête formulée par le nœud qui les a précédemment prêtées.

4.3.6.b Ressources Partagées

Supposons qu'une application distribuée arrive sur un nœud. Si les quantités de ressources disponibles sur le nœud (excepté les ressources partagées) sont suffisantes pour accepter l'application dans le système, le nœud qui procède à l'admission peut envoyer une requête pour demander des ressources partagées additionnelles.

Cependant, le nœud qui a besoin des ressources partagées additionnelles ne connaît pas la disponibilité de ces ressources sur les nœuds distants. Ainsi la requête est envoyée à tous les nœuds. Il est possible de mettre en place des mécanismes permettant à un nœud d'informer tous les autres à chaque fois qu'une application distribuée est admise. Ainsi, chaque nœud saura à chaque instant la disponibilité des ressources partagées sur les nœuds et d'envoyer la requête à un nœud donné ou certains si la demande est importante. Ces deux possibilités ont un coût en termes de messages.

Dans cette thèse, l'utilisation des ressources partagées est gérée via un protocole à jeton expliqué dans la section 4.4. Pour exécuter une étape qui utilise une ressource partagée, il faut demander et obtenir le jeton. Le nœud qui a obtenu le jeton détient aussi une liste de toutes les requêtes demandant l'utilisation du jeton. Ce nœud a une connaissance de l'utilisation des ressources partagées sur les nœuds du système. En envoyant un message au nœud détenteur du jeton, les nœuds sur lesquels les ressources partagées sont disponibles sont connus et c'est ceux là qui recevront des messages pour la demande des ressources partagées additionnelles. Cependant, envoyer un mes-

sage point à point aux nœuds concernés a un coût. En utilisant des mécanismes qui permettent d'envoyer un message multicast, ce coût serait diminué.

Les procédures pour la demande des ressources partagées additionnelles n'ont pas été implémentées.

4.3.6.c Durée de validité de l'emprunt/Prêt dynamique

Les requêtes pour la demandes des ressources additionnelles peuvent être acceptées ou pas selon la disponibilité des ressources sur les nœuds concernés par les requêtes.

Une question importante se pose : si une requête de demande des ressources additionnelles, formulée par exemple par $Node_2$, a été acceptée, par exemple par $Node_1$, quelle est la durée de validité de cet accord ? Autrement dit, à quel moment $Node_1$ a droit de reprendre ces ressources s'il en a besoin ?

Notre *middleware* gère des applications qui doivent a priori être exécutées avant leur échéance. Ainsi, plusieurs situations peuvent se présenter, entre autres :

- Le nœud $Node_2$ continuera d'utiliser les ressources additionnelles obtenues jusqu'à l'échéance de l'application concernée par la requête ou plus précisément la *SU* concernée. Après la deadline, $Node_2$ rend les ressources à $Node_1$. $Node_2$ peut les redemander plus tard si nécessaire. L'idée n'est pas mauvaise puisque ces ressources sont utiles jusqu'à l'échéance de l'application, après quoi les ressources sont automatiquement libérées, sans messages supplémentaires. Un autre avantage est que le nœud $Node_1$ peut utiliser les ressources libérées pour admettre une autre application. Un inconvénient est que si $Node_1$ ne les utilise pas et que $Node_2$ en a besoin un instant après, des messages sont nécessaires. De manière générale, cette solution est favorable si les besoins en ressources sont fluctuants dans le système.
- Une deuxième situation serait de dire que les suppléments accordés sont valables tant que ceux-ci ne font pas l'objet d'une demande ultérieure. En effet, une requête n'est formulée que s'il y a besoin. Ce deuxième point est avantageux si les besoins en ressources se stabilisent à partir d'un certain moment.

La deuxième possibilité est le cas général. Elle permet l'utilisation des ressources selon le besoin de chaque nœud. En effet,

dans la deuxième situation, pour retrouver la première, il suffit que $Node_1$ formule une requête à la fin de l'échéance de l'application concernée par la précédente requête formulée par $Node_2$. C'est la deuxième solution qui est adoptée dans cette thèse parce que nous avons supposé qu'à un certain moment, les besoins changent durablement.

4.3.7 Evolution de la politique d'emprunt

Les procédures pour la demande d'emprunt dynamique modifient la politique d'emprunt des ressources. Par exemple, supposons qu'à chaque début de période $Node_1$ a 10ms de temps processeur sur $Node_2$. Par la suite, si $Node_1$ demande à $Node_2$ un supplément α pour admettre une application distribuée ayant pour échéance d_1 alors $Node_1$ se retrouverait chaque période avec $(10+\alpha)$ ms pendant que $Node_2$ réduit ses ressources réservées de α un certain nombre de périodes.

Nous appelons propriétaire d'une ressource, le gestionnaire local (*LM*) qui a prêté cette ressource. D'autres procédures peuvent être définies pour automatiquement modifier la politique d'emprunt. Par exemple, après un temps Δt ($\Delta t < T$) ou K périodes ($K > 1$) depuis le début d'une période, si un nœud n'utilise pas *les ressources empruntées*, son propriétaire récupère automatiquement une partie (β%) ou la totalité de ses ressources après accord, K et β sont des paramètres d'implémentation.

Cette politique de libération est-elle profitable pour le système ? Elle est avantageuse si les ressources libérées sont utilisées par leur propriétaire. L'inconvénient est que lorsque les propriétaires n'ont pas besoin des ressources et que les nœuds qui les ont libérées en ont besoin, des messages sont nécessaires (requête d'emprunt dynamique).

Ne pas admettre une application pour éviter des messages n'est pas une bonne idée si les ressources concernées ne sont pas utilisées quelque part. N'oublions pas que l'un de nos objectifs est d'optimiser l'utilité globale du système. Cette politique est utile pour le système. Elle serait encore très utile si des mécanismes permettraient d'estimer la charge ou la fréquence d'arrivée des applications dans le système. Ainsi, cette politique serait activée en deçà d'une certaine charge ou d'une fréquence donnée. En effet, plus le système est chargé ou plus les applications arrivent sur les nœuds, plus la possibilité d'admettre une application augmente sur les nœuds.

Dans cette thèse les procédures de libération des ressources n'ont pas été implémentées. Une expérimentation permettra d'en savoir plus.

4.4 Gestion des Ressources partagées

Les ressources locales sont gérées par chaque *LM* sur son nœud. Chaque *LM* utilise la vue locale construite sur son nœud pour admettre et planifier l'utilisation des ressources. L'exécution d'une étape qui utilise uniquement des ressources locales ne pose pas de problème puisque le contrôle est local.

La situation est toute autre pour l'exécution des étapes qui utilisent des ressources globales ou partagées. Des mécanismes sont nécessaires pour s'assurer qu'une seule synchronisation d'application distribuée utilise une ressource partagée à un moment donné.

Ainsi, pour éviter des conflits d'utilisation des ressources globales telles que la bande passante, un protocole à base de jeton contrôle l'utilisation des ressources globales. Ce protocole est basé sur les points suivants :

- Initialement dans le système, un nœud détient le jeton. Les autres nœuds savent quel nœud a le jeton.
- Le nœud qui détient le jeton utilise une ressource globale sans le notifier aux autres. Si un autre nœud veut exécuter une étape qui exige l'utilisation des ressources globales, il doit demander et obtenir le jeton. Il envoie un message au nœud qui détient le jeton et l'attend. Pendant ce temps, le nœud qui a demandé le jeton exécute toute autre étape qui n'exige pas l'utilisation d'une ressource globale. Quand le nœud reçoit le jeton, l'étape qui attend le jeton débute son exécution.
- Le nœud qui détient le jeton détient aussi la liste de toutes les requêtes pour la demande du jeton. Lorsque ce nœud reçoit une requête, plusieurs situations se présentent :
    - Si le nœud n'utilise pas le jeton, il le donne au nœud qui l'a demandé. Ce cas implique que la liste des requêtes est vide ;
    - Si le nœud utilise le jeton, c'est-à-dire, est en train d'exécuter une étape qui exige la présence du jeton, la requête est insérée dans la file d'attente selon EDF. A la fin de l'utilisation du jeton, celui-ci est envoyé au nœud prioritaire qui est le premier dans la liste. En effet, la liste est gérée selon l'urgence de chaque application. L'urgence de chaque application est symbolisée

par un temps, qui est une échéance signifiant la date au plus tard pour l'utilisation du jeton. Chaque requête est associée à une échéance et une durée qui représente le temps d'utilisation de la ressource globale. Le principe d'EDF est utilisé pour retrouver l'application la plus prioritaire. Cette façon permet d'éviter l'annulation de l'exécution de certaines applications dont l'exécution est urgente alors que leur requête se retrouve à la fin de la liste.

  - Si la requête arrive après l'envoi du jeton à un autre qui l'a demandé, cette requête est transférée vers le nouveau nœud détenteur du jeton. Ce dernier insère cette requête dans la file d'attente pour être traitée le moment venu.

- Le nœud qui reçoit le jeton informe tous les autres nœuds sauf le précédent détenteur. Ainsi, l'adresse du détenteur du jeton est mise à jour sur chaque nœud.

L'utilisation de ce protocole a un coût en termes de messages échangés entre les nœuds. Selon ce scenario, pour obtenir le jeton dans un système ayant *N* nœuds, 1 message est envoyé pour demander le jeton, 1 autre pour la réponse. En plus, (*N-2*) messages sont envoyés pour informer les autres nœuds afin de mettre à jour l'adresse du nouveau détenteur du jeton. Au total, *N* messages sont nécessaires pour obtenir et utiliser le jeton. L'envoi d'un message multicast réduirait le coût.

4.5 Echéances des Applications

Les applications gérées sont composées d'une succession d'unités de planification ayant chacune une échéance. Chaque unité de planification doit être exécutée avant son échéance et en théorie, toute valeur d'échéance serait acceptée. Mais, comme les ressources sont gérées

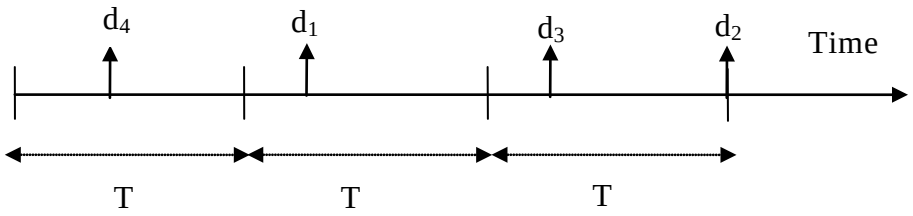

Figure 4-7: Echéances des Applications

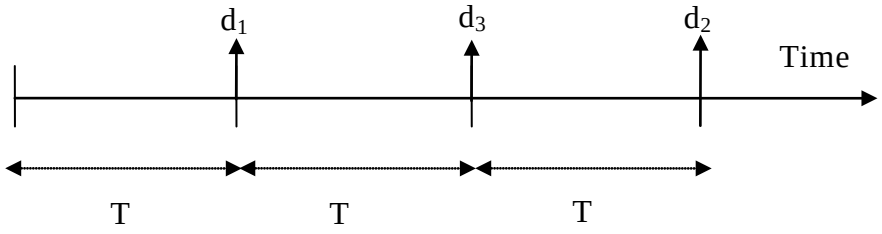

Figure 4-8: Alignement des Echéances

sur la base d'une période, que faire si les échéances sont à l'intérieur des périodes, par exemple d_4, d_1 et d_3 sur la Figure 4-7 ?

Puisque dans une période, l'ordre d'exécution n'est pas connu, il n'est pas possible de garantir les échéances à l'intérieur d'une période. Par conséquent, toute échéance sera alignée sur les périodes (Figure 4-8) de la manière suivante :

- Toute échéance de l'application d_i est arrondie à βT (Figure 4-8) avec $\beta T < d_i$, β un entier et T la période ;
- Toute échéance inférieure à la période telle que d_4 sur la Figure 4-7 est rejetée (Figure 4-8).

Cependant, la taille de la période doit être judicieusement choisie pour éviter trop de rejets d'application lors de l'admission. En effet, Figure 4-7, si la taille de la période était 2*T par exemple, les échéances d_4, d_1 seraient rejetées, alors que si la taille de la période était inferieure à d_4, toutes les échéances seraient acceptées.

4.6 Conclusion

Dans ce chapitre, un *middleware* décentralisé pour la gérer des ressources d'applications distribuées a été proposée. La contribution principale est la proposition d'un schéma d'emprunt pour la gestion des ressources. Ce *middleware* a pour but de minimiser le nombre de communications entre les nœuds et d'augmenter l'utilisation des ressources dans le système.

Utilisant ce schéma d'emprunt statique, les gestionnaires locaux (*LM*) disposent localement des informations nécessaires pour admettre les applications et contrôler leur exécution. Le contrôle des applications s'appuie sur leur description et celle de l'environnement. Les applications sont admises si les ressources disponibles sont suffisantes pour les exécuter. Leur exécution suit un chemin dans un graphe.

Dans ce *middleware*, les ressources locales sont gérées par chaque gestionnaire sur son nœud alors que les ressources globales sont gérées par tous les gestionnaires collectivement. A cet effet, un protocole de gestion des ressources globales a été proposé.

Cependant, les applications gérées ne s'exécutent que sous des conditions strictes de disponibilité des ressources pour fournir un même service. Le chapitre suivant ajoute un support pour gérer les applications qui ont un certain degré de liberté dans leur fonctionnement.

Chapitre 5 Adaptation du Comportement

Pour gérer finement la QoS, ce chapitre ajoute un support pour l'adaptation du comportement d'applications distribuées. Via un modèle, l'application expose tous ses comportements possibles et le comportement approprié sera choisi au moment de l'exécution en fonction de la disponibilité des ressources.

Autrement dit, s'il est impossible d'exécuter l'application pour fournir un service donné dans les conditions idéales, une dégradation va s'imposer.

Ce chapitre est subdivisé en deux parties. La première complète le modèle général d'application pour l'adaptation. La deuxième est consacrée à la description des mécanismes et stratégies construits au-dessus du schéma d'emprunt pour l'adaptation du comportement d'applications distribuées.

5.1 Modélisation d'application

5.1.1 Structure de l'application Adaptable

Le modèle d'application présenté dans cette section complète celui présenté aux chapitres 3 et 4. L'objectif est de donner un degré de liberté à l'application pour s'exécuter et s'adapter selon le contexte d'exécution.

Une application est modélisée sous forme de graphe (Figure 5-1). Dans ce graphe, les arcs représentent l'exécution des activités (*Activity*). Une activité est une séquence d'étapes (*Steps*). Chaque activité (A_i) est associée à un mode (m_i) et une utilité (U_i). Dans les applications les modes définissent des comportements alternatifs qui fournissent le même service avec des exigences en QoS différents. L'utilité (U_i) indique le bénéfice ou le degré de satisfaction relatif au mode choisi. Chaque étape a ses exigences en termes de ressources, par exemple, l'utilisation du CPU, la consommation en mémoire, etc. Les nœuds dans le graphe (Figure 5-1) sont des points de décision à partir desquels plusieurs comportements fournissent le même service avec des niveaux de QoS différents.

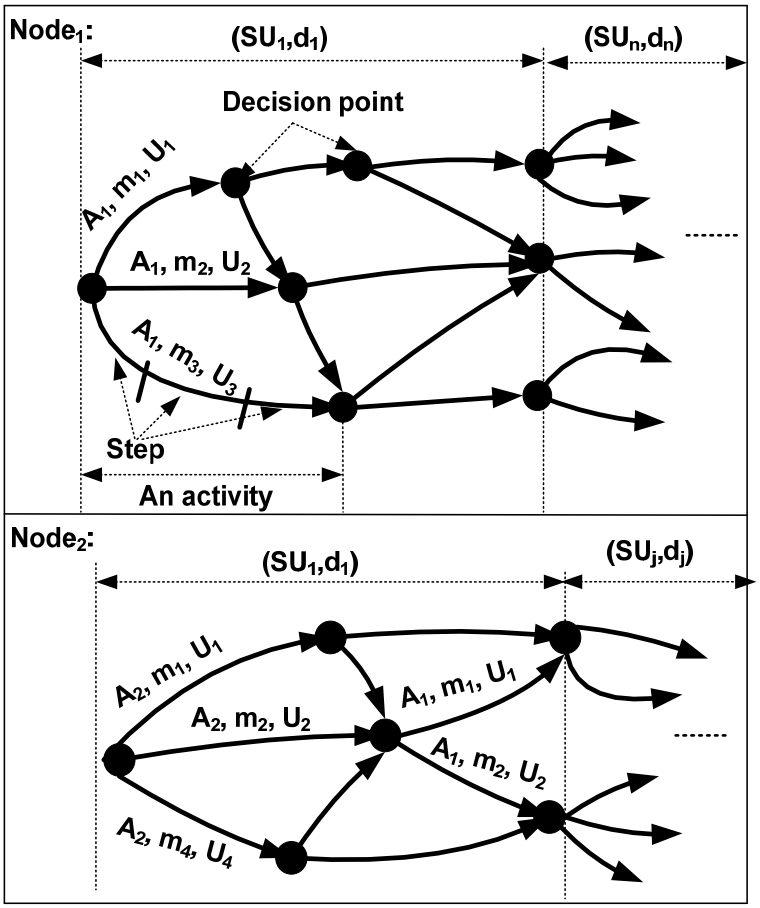

Figure 5-1: Exemple de modèle d'application distribuée

Une application locale est modélisée comme un seul graphe alors qu'une application distribuée est modélisée comme un ensemble de sous-graphes. Chaque sous-graphe correspond à une partie locale de l'application distribuée.

La Figure 5-1 représente un exemple d'une application distribuée sur deux nœuds $Node_1$ et $Node_2$. Cette application est une séquence d'unités de planification (*Scheduling Units*, SU_i). Chaque unité de planification est associée à une deadline (SU_i, d_i) et est composée d'un ensemble d'activités. Une unité de planification d'une application distribuée peut concerner un ou plusieurs nœuds. Par exemple, Figure 5-1, SU_1 sur le nœud $Node_1$ et SU_1 sur le nœud $Node_2$ appartiennent à la même *SU* distribuée sur ces nœuds tandis que SU_j est une unité de planification de cette application distribuée sur le nœud $Node_2$ uniquement. Si une unité de planification est distribuée sur plusieurs nœuds alors il existe au moins une activité qui est distribuée sur au moins deux nœuds. Cette activité est donc modé-

lisée comme des sous-activités avec le même mode et la même utilité sur les nœuds concernés. Par exemple, Figure 5-1, (A_1, m_1, U_1) sur le nœud $Node_1$ et (A_1, m_1, U_1) sur le nœud $Node_2$ sont des parties d'une même activité.

5.1.2 Modèle Statique de l'application

La Figure 5-1 représente le modèle d'une application pour qu'elle soit gérable par notre *middleware*. Toutes les applications sont basées sur une structure commune, le modèle statique de l'application.

Le modèle statique décrit les différents niveaux de décomposition de l'application. Cette structure doit contenir toutes les informations dont le *middleware* a besoin pour contrôler l'exécution de l'application.

Le modèle utilisé (Figure 5-2) [VSM05] s'appuie sur UML [OMG07]. UML est un langage qui offre de nombreuses notions et des capacités d'extension. Le formalisme proposé, semblable à [OMG03], décrit formellement les applications et leurs besoins en ressources. Ce formalisme fait ressortir clairement les concepts utilisés dans Figure 5-1: *Scheduling Unit*, *Activity*, *Step*. Une unité de planification est faite pour séparer les parties de l'application associées à une échéance (*delay*) qui est une durée maximale associée dynamiquement à la date de début d'exécution de l'unité de planification pour en spécifier l'échéance. Pour chaque unité de planification,

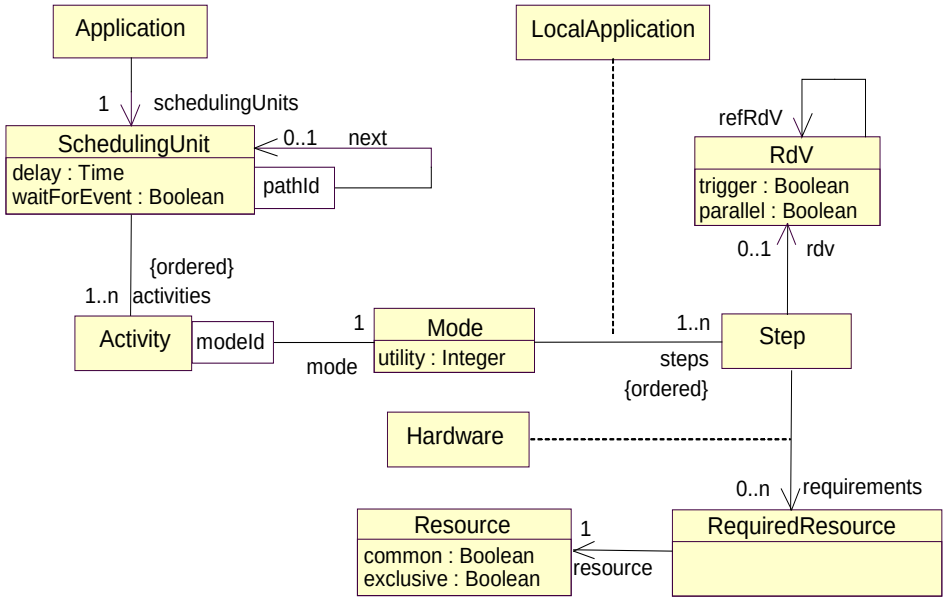

Figure 5-2: Modèle Statique de l'application

un chemin (*pathId*) indique l'unité suivante à exécuter. L'application choisit elle-même le chemin au moment de l'exécution. La Figure 5-2 permet l'interruption de l'exécution de l'application avant une unité de planification pour attendre un événement (*waitForEvent*). Le modèle montre aussi que chaque activité (*Activity*) de l'application est associée à un mode et à une utilité. Les étapes sont utilisées pour séparer dans une activité les traitements ayant des besoins en ressources différents. Une application locale (*LocalApplication*) représente un nœud logique qui sera associé à un nœud physique au moment de l'exécution. Une application locale est conçue pour s'exécuter sur n'importe quel nœud. Les ressources nécessaires pour une étape (*Step)* dépendent du matériel (*Hardware*) sur lequel il s'exécute.

L'étape est donc la décomposition la plus élémentaire de l'application. Une activité est exécutée dans le même mode et consiste à exécuter séquentiellement toutes ses étapes. Le contrôle de l'application est discret. Une fois que la première étape de l'activité a démarré son exécution, il n'est plus possible de revenir pour changer le mode d'exécution. Le changement de mode se fait uniquement au point de décision.

Le modèle (Figure 5-2) indique également qu'il y a une possibilité de rendez-vous entre les étapes (*RDV*). Un rendez-vous est séquentiel quand une étape se termine avant l'exécution d'une nouvelle étape de la même application alors que le rendez-vous est parallèle quand deux étapes de deux applications locales différentes sont exécutées au même moment sur deux nœuds différents en utilisant une même ressource globale.

5.1.3 Exemple d'application distribuée

Pour mieux illustrer notre propos, le Tableau 5-1 reprend l'exemple donné au chapitre 3 en le complétant.

L'application Tableau 5-1 est une succession de trois unités de planification : *Start, Sample* et *Analyse*. L'unité de planification *Start* concerne le chargement de l'application qui se fait en exécutant les étapes *Load* sur les deux nœuds concernés. L'unité de planification *Sample* fait l'acquisition, la transmission et l'affichage des données représentés respectivement par les activités *A_Transmission, A_Acquisition,* et *A_Display.* L'acquisition des données est faite sur $Node_1$ et l'affichage sur $Node_2$. La transmission est une activité distribuée sur les deux nœuds $Node_1$ et $Node_2$.

Le Tableau 5-1 montre aussi que les données sont transmises selon trois modes (1, 2 et 3) qui correspondent à trois niveaux de satisfaction caractérisés par leur utilité 100, 100, 80. Les deux premiers modes (1 et 2) ont la même utilité car nous supposons que la compression est sans perte, ce qui n'est pas le cas pour le mode 3 qui a une utilité inférieure. Après la réception des données puis leur affichage sur le nœud $Node_2$, la dernière unité de planification (*Analyse*) continue l'exécution de l'application.

Tableau 5-1: Application Distribuée

Unité de planification	Activité	Mode	étapes	Nœud	Utilité
Start	A_Start	1	Load	$Node_1$	100
	A_Start		Load	$Node_2$	
Sample	A_Acquisition	1	Acquisition	$Node_1$	100
	A_Transmission	1	Transmission1	$Node_1$	100
			Reception1	$Node_2$	
		2	Compression1, Transmission2	$Node_1$	100
			Reception2, Decompression1	$Node_2$	100
		3	Compression2, Transmission3	$Node_1$	80
			Reception3, Decompression2	$Node_2$	80
	A_Display	1	Display1	Node2	100
		2	Display2	Node2	100
		3	Display3	Node2	80
Analyse	……..	…….	…….	……..	…..

La section suivante montre comment nous pouvons utiliser UML pour décrire l'exécution de l'application sur plusieurs nœuds.

5.1.4 Description de l'application

Une application peut être décrite en utilisant des diagrammes décorés des propriétés de QoS. La description dans cette thèse est fondée sur des diagrammes d'activités d'UML enrichis avec des valeurs étiquetées (tagged values en UML). Au plus haut niveau un diagramme d'activité décrit le graphe d'exécution de l'application ou d'une unité de planification (*Scheduling Unit*).

La Figure 5-3 décrit l'exemple d'application donné Tableau 5-1 en ajoutant une boucle sur l'échantillonnage (*Sample*).

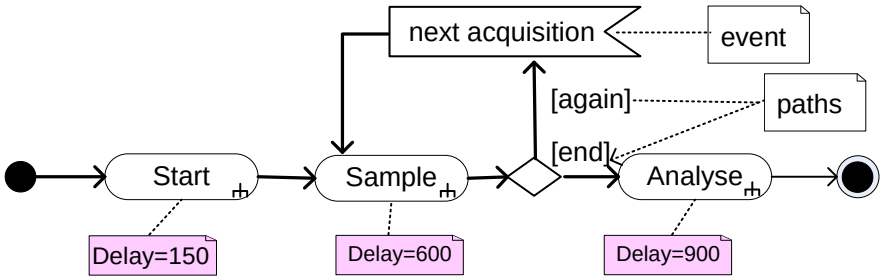

Figure 5-3: Exécution de l'application Tableau 5-1 sans boucle

Ensuite, chaque unité de planification est décrite en utilisant une hiérarchie d'activités. Dans la Figure 5-4, l'exécution détaillée de l'unité de planification *Sample* est montrée. Les étapes sont représentées par des activités d'UML. Nous avons choisi de ne décrire que deux modes pour simplifier la figure. Cette unité de planification est distribuée sur deux nœuds : $LApp_{11}$ est la partie locale de l'application qui s'exécute en faisant l'acquisition et le transfert des données sur un nœud et $LApp_{12}$ est la partie locale de l'application

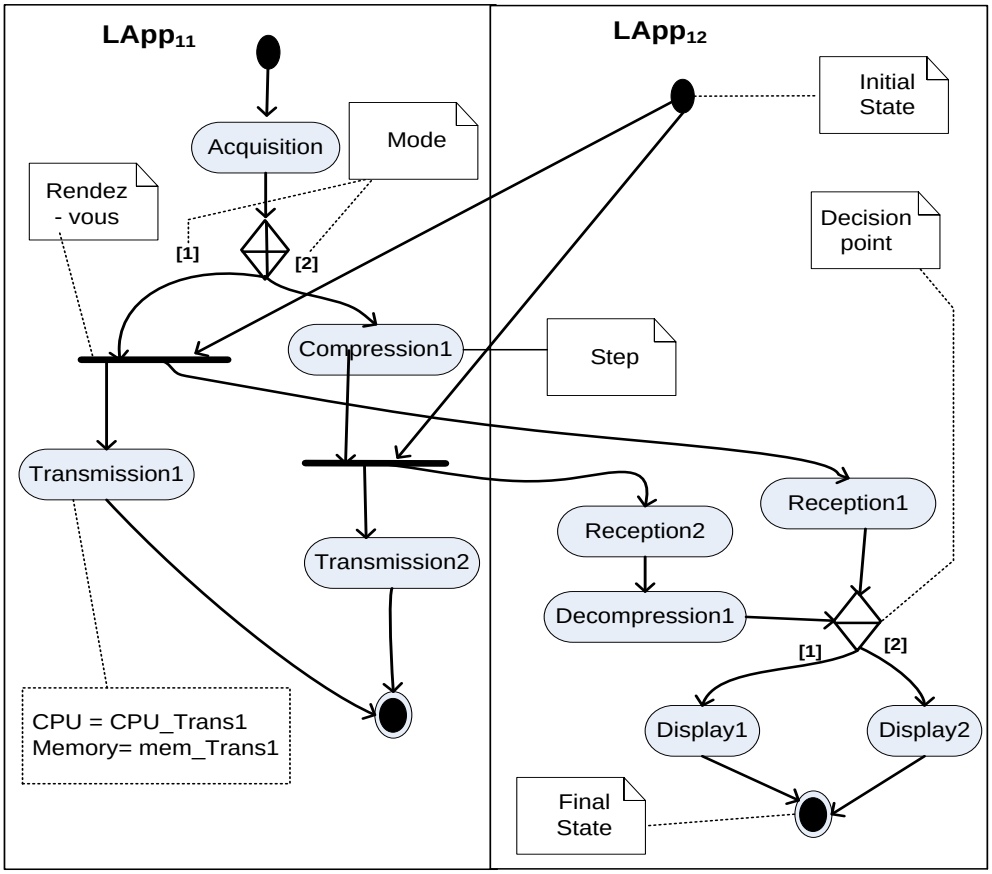

Figure 5-4: Graphe d'exécution détaillée

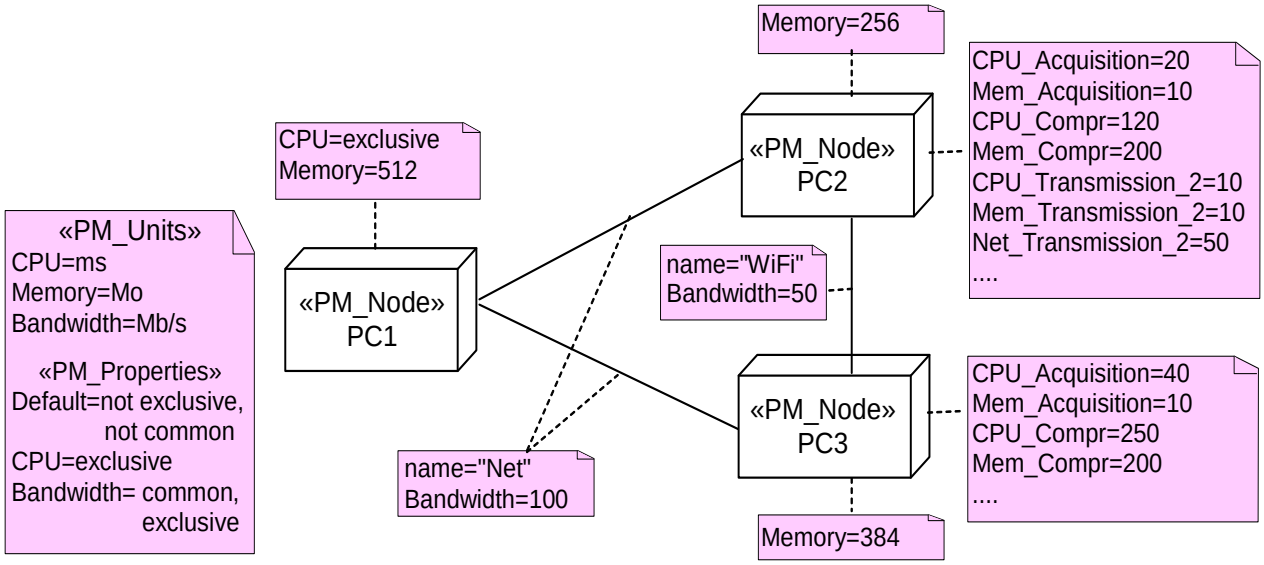

Figure 5-5: Diagramme de déploiement spécifiant l'environnement d'exécution

qui reçoit les données et les affiche sur un autre nœud. Le nœud physique d'exécution est inconnu. La Figure 5-4 indique en outre les ressources nécessaires pour exécuter chaque étape selon le nœud. Dans la Figure 5-4 seule *Transmission1* est associée aux ressources pour ne pas surcharger la figure. Il est important de noter que cette description est indépendante du matériel. Cette figure montre les rendez-vous entre les étapes. Par exemple, pour un rendez-vous entre *Transmission2* sur un nœud et *Reception2* sur un autre, les deux parties de l'application, doivent être prêtes au même moment (*rendez-vous*) pour débuter l'exécution sur leur nœud respectif.

La gestion de la QoS nécessite aussi la description de l'environnement dans lequel les applications s'exécutent, c'est ce que fait le digramme de déploiement dans la Figure 5-5. Les valeurs sont exprimées à l'aide de valeurs étiquetées (*tagged values*) dans des commentaires. C'est ici que sont données les valeurs réelles des besoins des applications en fonction du matériel. Le stéréotype *PM_Node* (Figure 5-5) étend la métaclasse *Node* d'UML et est appliqué au nœud dans la figure. De façon plus générale, un profile UML décrit toutes les extensions UML utiles pour représenter le modèle.

Nous avons présenté dans cette section le modèle général d'application. Nous avons aussi décrit les différents niveaux de décomposition de l'application et montré comment nous pouvons utiliser UML pour décrire l'exécution des applications et spécifier l'environnement d'exécution des applications. La section qui suit

présente les stratégies et les mécanismes pour mettre en œuvre l'adaptation d'applications distribuées.

5.2 Adaptation

Dans ce chapitre l'environnement d'exécution est le même que celui du chapitre précédent. Il est composé d'un ensemble de nœuds qui communiquent via un réseau. Chaque application adapte son comportement selon la disponibilité des ressources dans le système et sous le contrôle du *middleware*. L'adaptation est fondée sur le schéma d'emprunt présenté dans le chapitre précédent.

5.2.1 Phases pour l'adaptation

L'adaptation est effectuée en deux phases. Dans la *phase d'admission*, chaque *LM* utilise les informations sur les applications et une vue locale de la disponibilité des ressources du système construite en appliquant notre schéma d'emprunt pour décider de l'admission de l'application. Si le test d'admission réussit, un chemin d'exécution de la *SU* courante est choisi. Les étapes des activités appartenant à ce chemin sont planifiées. Dans la phase d'exécution, l'application exécute ses étapes (donc chaque activité) sous le contrôle du *middleware*. A chaque point de décision, le middleware choisit une activité avec la plus haute utilité possible en fonction des ressources disponibles afin d'améliorer la QoS fournie. Ainsi, le comportement de l'application peut changer aux points de décision.

5.2.2 Stratégies d'adaptation

L'idée basique de l'adaptation est de changer le comportement de l'application selon le contexte d'exécution. Dans le modèle général d'application présenté Figure 5-1, les activités ont différentes exigences en ressources. Une utilité évalue le niveau de satisfaction relatif au QoS fourni. Dans ce contexte, on peut imaginer au moins deux stratégies. Pour appuyer notre propos, nous supposons une application composée d'une seule *SU* avec trois activités *A1*, *A2*, *A3* avec des besoins en CPU respectivement 30, 20 et 10 ms et d'utilité respectivement 90, 50, 30.

- Dans *la phase d'admission*, le *middleware* choisit un chemin planifiable dans la *SU* avec la plus petite utilité et augmente l'utilité si possible pendant la *phase d'exécution*. Par exemple,

considérant l'exemple ci-dessus, l'activité *A3* d'utilité *30* est choisie pendant la *phase d'admission* mais pendant la *phase d'exécution* l'utilité augmente en choisissant d'exécuter l'activité *A1* d'utilité *90*.

- Dans la *phase d'admission*, le *middleware* choisit un chemin composé des activités avec la plus haute utilité possible selon la disponibilité des ressources (par exemple l'activité *A2* d'utilité *20* dans l'exemple ci-dessus). Dans la *phase d'exécution*, le niveau de la QoS est amélioré en choisissant d'exécuter les activités avec la plus haute utilité possible aux points de décision. Par exemple l'activité *A1* avec l'utilité *90*.
- La première stratégie admet l'application avec la plus petite utilité alors que la seconde admet l'application avec la plus haute possible en fonction du contexte d'exécution. Choisir l'une ou l'autre de deux stratégies dépend des objectifs fixés. Dans notre approche, la plus petite utilité correspond à un service dégradé qui nécessite moins de ressources. La dernière stratégie, exécute moins d'applications. Maximiser uniquement l'utilité des applications qui s'exécutent n'est pas suffisant, il faut tenir compte des applications rejetées. En effet, nous pouvons choisir d'exécuter une seule application avec l'utilité maximale et rejeter les autres. En supposant que les applications rejetées ont pour utilité - ∞, exécuter par exemple trois applications avec l'utilité totale de 50 est mieux que d'exécuter une application avec une utilité totale de 100.

Dans cette thèse, notre principe de base est que tout service dégradé est mieux que pas de service de tout. Donc, nous avons adopté la première stratégie puisqu'elle permet d'exécuter plus d'applications.

5.2.3 Coordination de l'adaptation

Une application est un ensemble d'unités de planification qui sont admises et exécutées séquentiellement avant leur échéance.

Pour une application distribuée, si une *SU* est distribuée sur plusieurs nœuds, le *middleware* choisit un chemin pour que l'application adapte son comportement en tenant compte des ressources disponibles sur ces nœuds.

Les *LM* des nœuds concernés par une unité de planification (*SU*) doivent coordonner leurs décisions afin d'exécuter les activités distribuées dans le même mode. Par exemple, Tableau 5-1, si le nœud

Node$_1$ compresse les données avant de les transférer selon le mode 3, le nœud *Node*$_2$ doit décompresser les données selon le même mode avant de les afficher.

De toutes les façons, que la *SU* courante contienne ou pas une activité distribuée, l'adaptation est pilotée par le *LM* qui a fait l'admission de la *SU*. Quand un mode est choisi, le *LM* envoie cette information à tous les *LM* des nœuds concernés par une activité distribuée.

5.2.4 Maximisation de l'utilité du système

Dans cette thèse nous gérons la QoS des applications en contrôlant l'utilisation des ressources du système. C'est le cas par exemple pour les systèmes industriels où l'objectif est la résolution conjointe des problèmes d'ordonnancement et de planification des moyens de transformation et de transport. C'est aussi le cas dans les systèmes, quand on cherche à paralléliser les programmes ou à faire la gestion des applications temps réel.

Etant donné un ensemble d'objets et de ressources, il s'agit d'affecter chaque objet à une ressource en satisfaisant un ensemble de contraintes et en optimisant une fonction objective donnée en mesurant la qualité de l'affectation.

Dans notre cas, le problème de la gestion des ressources consiste à affecter un ensemble de ressources à un ensemble d'applications dans l'espace de temps en satisfaisant un ensemble de contraintes. Ce problème comme un grand nombre de problèmes de planification est un problème d'optimisation combinatoire et d'affectation sous contraintes. C'est typiquement un problème de résolution de contraintes, NP-Complet [LLR+99] comme nous l'avons dit dans le chapitre 2. La solution n'est pas, a priori, connue dans le cas général.

Dans notre approche, une activité est associée à un mode et une utilité (Figure 5-1) qui détermine le degré de satisfaction de l'utilisateur. Plus l'utilité est grande, plus la satisfaction est bonne. Chaque mode d'exécution est associé à une combinaison de ressources. Notre *middleware* a donc pour objectif de maximiser l'utilité globale du système. En plus du fait que le problème est NP-Complet, notre *middleware* fait face à des applications dont la loi d'arrivée est inconnue, ce qui rend impossible la recherche d'un optimum. Au mieux, on pourrait essayer de trouver une succession d'optimums instantanés. Cependant, la somme des décisions optimales intermé-

diaires ne conduit pas à une décision optimale globale. Pour faire face à cette complexité, notre objectif est la recherche d'une solution approchée en utilisant un algorithme d'approximation.

Notre *middleware* utilise une heuristique basée sur EDF : s'il existe une solution, cet algorithme la trouve alors que si les étapes ne peuvent pas être planifiées, la solution est arbitraire.

Les *LM* choisissent une activité en fonction du contexte pour sélectionner la plus haute utilité possible. Supposons $u(i)$ et $u_m(i)$ respectivement l'utilité sélectionnée et l'utilité maximum au point de décision *i* et *N* le nombre de points de décision tout au long du chemin d'exécution de l'application. Evidemment $\frac{u(i)}{u_m(i)} \leq 1$. A chaque point de décision le *LM* choisit une activité avec une utilité $u(i)$ de telle sorte que $\frac{u(i)}{u_m(i)}$ soit le plus proche possible de 1. Définissons $U = \frac{\sum_{i \in N} u(i)}{\sum_{i \in N} u_m(i)}$. *U* définit la qualité d'exécution de l'application dans le système. Elle est égale à 1 si toutes les activités de l'application ont été sélectionnées avec la plus haute utilité à chaque point de décision. Autrement dit, toutes les activités ont été exécutées avec l'utilité maximale, c'est le cas idéal qui implique $\frac{u(i)}{u_m(i)} = 1$ à chaque point de décision. *U* est inférieure à 1 si au moins une activité n'est pas exécutée avec l'utilité maximale, $\frac{u(i)}{u_m(i)} < 1$ à certains points de décision. Notre algorithme tente d'optimiser l'utilité globale du système en choisissant à chaque point de décision l'activité avec la plus haute utilité possible.

5.3 Conclusion

Ce chapitre a présenté un support pour l'adaptation des applications. Pour une gestion fine de la QoS, un modèle d'application adaptable est décrit. Ce modèle expose des comportements possibles de l'application et le comportement approprié est sélectionné au moment de l'exécution en fonction de la disponibilité des ressources.

Via ce modèle, les concepteurs d'applications prennent en compte les spécificités de chaque application en proposant des modes de fonctionnement appropriés.

L'avantage principal de l'adaptation est l'utilisation la plus efficace possible des ressources de l'environnement. Elle permet d'exécuter au mieux l'application afin de fournir un service et de s'adapter dynamiquement aux changements du contexte.

Le *Framework* proposé utilise une heuristique qui combine la réservation des ressources et la planification des étapes des applications pour garantir leur exécution et maximiser l'utilité globale du système.

Chapitre 6 Implémentation

Ce chapitre présente les résultats de nos simulations et montre l'efficacité de l'approche proposée.

La première partie (section 6.1) introduit la technologie et la plate-forme utilisées pour nos simulations. La deuxième partie (section 6.2) définit l'architecture et introduit quelques principes et règles d'implémentation. La troisième partie (section 6.3) sera consacrée à la présentation des résultats. Avant de conclure (section 6.5), la section 6.4 discute quelques aspects de la solution.

6.1 Technologie et plate-forme utilisées

Les systèmes informatiques actuels sont bâtis à l'image que l'on se fait du fonctionnement d'une société humaine. Ils sont constitués d'entités réparties à travers le système qui communiquent entre elles afin de s'organiser et de coordonner leurs activités. Parmi ces entités, nous trouvons les agents.

Les agents et les systèmes multi-agents fournissent un cadre adéquat pour analyser, concevoir et implémenter des systèmes ou des applications [GMM98] [AAS99].

Cette section ne s'étend pas les différents systèmes à base d'agents mais choisit la définition qui convient et justifie le choix d'une plate-forme d'implémentation des agents.

6.1.1 Agents et systèmes multi-agents

Le terme agent englobe plusieurs aspects théoriques et pratiques différents *[MR95]*.

Le choix de cette technologie pour simuler notre approche est dû tout simplement au caractère inhérent décentralisé de nos besoins et à la propriété d'autonomie des agents [CBF03].

Dans cette thèse, le terme agent sera utilisé pour faire référence à un programme informatique qui a la capacité d'exécuter des actions indépendantes, pour réagir et communiquer. Les agents peuvent communiquer sur une seule machine ou sur le réseau dans l'objectif de coordonner leurs actions. Ils peuvent échanger des informations via des messages.

Pour plus d'informations, d'autres définitions sont proposées dans [Fer95], [MR95], [Yoa93], [HVD87], [DUR], [HD99] et [Hya96].

Dans [Nwa96], une typologie a été utilisée pour différencier les sortes d'agents et de les séparer en classes. La catégorie des *agents réactifs* nous intéresse particulièrement. Ces agents ne possèdent pas de représentation symbolique interne de leur environnement. Ils agissent et répondent à leur environnement à la suite de stimulis. Ces agents sont relativement simples, mais lorsqu'un groupe est vu globalement, des concepts complexes émergent. Un agent réactif peut aussi être vu comme une collection de modules qui ont chacun des tâches très spécifiques.

Le système dans lequel les agents évoluent est appelé système multi-agent (*SMA*). Un système multi-agent est composé d'un ensemble d'agents situés dans un environnement [AG92]. Un ensemble de relation entre les agents de l'environnement est défini. Une application multi-agent se compose d'un environnement et des agents. L'environnement contient des informations que les agents utilisent et manipulent pour exécuter leurs tâches.

Chaque agent a une structure c'est-à-dire qu'il possède une architecture logicielle lui permettant de s'exécuter dans un système informatique. La structure de chaque agent dépend de la plate-forme d'exécution choisie. Dans cette thèse la plate-forme JADE est choisie.

6.1.2 Plate-forme JADE

Une plate-forme multi-agent est un ensemble d'outils nécessaires à la construction et à la mise en service d'agents au sein d'un environnement spécifique. Ces outils peuvent servir également à l'analyse et au test du *SMA* ainsi créé. Ces outils peuvent être sous la forme environnements de programmation (API) et d'applications permettant d'aider le développeur. Nous allons brièvement présenter et justifier dans cette section le choix de la plate-forme JADE (Java Agent DEvelopment framework) pour simuler notre approche.

6.1.2.a La plate-forme multi-agent JADE

JADE est une plate-forme multi-agent développée par le laboratoire TILAB et décrite dans [BEL00] [BEL99]. Elle est entièrement développée en Java. Cette plate-forme fournit à la fois un modèle pour développer les systèmes multi-agent et un environnement pour exécuter les applications distribuées à base d'agents.

Dans le domaine des systèmes multi-agents, l'efficacité est un problème crucial. L'efficacité d'un système à base d'agents dépend à la fois de sa conception et de la plate-forme agents sous-

jacente. La plate-forme JADE [LES04] [KDS04] est largement connue, facile à utiliser et bien documentée. La conception de l'agent dans JADE et son implémentation en Java offrent une bonne efficacité d'exécution. Chaque agent JADE détient une collection de comportements qui sont planifiés et exécutés pour effectuer des actions de l'agent. Chaque agent JADE fonctionne dans un thread Java, ce qui satisfait les propriétés d'autonomie, et tous les comportements de l'agent sont exécutés dans un seul thread.

Pour l'interopérabilité entre les agents ou les systèmes multi-agent, la plate-forme JADE utilise la spécification FIFA [FIF09] dont le but est de produire des standards pour l'interopération d'agents hétérogènes. Trois rôles obligatoires sont présents dans JADE : l'AMS (Agent Management Systèm) qui contrôle l'accès et l'utilisation de la plate-forme, DF (Directory Facility) qui fournit un service de pages jaunes et ACC (Agent Communication Channel) qui fournit un service de transport de messages.

Des conteneurs (*containers*) peuvent être utilisés pour séparer des groupes d'agents dans un système multi-agent même si les agents peuvent être exécutés sur une seule machine. Les agents du service FIFA (par exemple l'AMS et le DF) sont localisés dans le conteneur principal (*main container*) qui est toujours lancé au démarrage de la plate-forme. JADE implémente un système de communication synchrone et asynchrone entre les agents.

6.1.2.b Performance de JADE

Le choix de JADE est basé sur de nombreux travaux sur la performance des plateformes multi-agents existantes (par exemple [KGM06] [LJJ06] [KNS04] [CTG+04]). Selon ces travaux, JADE est un environnement efficace, limité seulement par la limitation standard du langage de programmation Java, qui est interprété et exécuté dans une machine virtuelle : vitesse du processeur, quantité de mémoire disponible, vitesse de la connexion réseau. L'environnement lui-même n'introduit pas de surcoût (*overhead*) important. Avec JADE, on peut expérimenter des milliers d'agents distribués sur plusieurs machines communicant et échangeant plusieurs milliers de messages.

6.1.2.c Overhead dans JADE

Le surcoût (*overhead*) est défini comme étant le temps passé par un système à se gérer. Ce temps est important pour l'efficacité d'un système. Nous expliquons comment JADE réduit ce temps. Comme toute

plate-forme agent, JADE introduit un surcoût puisqu'elle utilise des couches additionnelles (couches de communication et autres) pour se gérer. Cependant, des précautions peuvent être prises pour réduire le surcoût durant les simulations [CQG02], par exemple grouper des agents dans un conteneur réduit le surcoût induit par les messages échangés entre ces agents.

Dans notre implémentation, un nœud est simulé par un conteneur et tous les agents représentant les applications résident dans un même conteneur. Selon [KGM06], envoyer un message dont le contenu est de taille importante augmente le surcoût. Dans notre *middleware*, les messages échangés sont de petite taille, sérialisés et avec peu de caractères, ce qui réduit le surcoût. Cependant, en raison de l'implémentation particulière de JADE, certaines actions peuvent avoir un impact sur le surcoût global du système, spécialement quand des changements ont lieu dans un conteneur distant tels que la création d'agents, la destruction d'agents, la localisation d'un agent, la recherche des services d'un agent etc. Toutes ces actions requièrent des messages entre le conteneur où l'action a eu lieu et le conteneur principal où sont logés les agents AMS et DF. Grâce au système de caches distribués dans la plate-forme JADE, le surcoût lié à la localisation d'un agent et la recherche d'un service est réduit. En plus, dans notre approche, les agents sont des objets logiciels à longue vie. Un agent est créé pour chaque application et est détruit à la fin de l'exécution. Toutefois, pour évaluer le surcoût lié à l'utilisation du CPU, nous avons mesuré les temps d'exécution moyens de toutes les fonctionnalités de notre *middleware* (création de l'agent, destruction de l'agent, temps d'admission, temps de préparation des messages, temps de traitement des messages etc.). Vu le temps d'exécution des étapes, nous avons par hypothèse négligé ces temps. Par exemple le temps d'exécution des étapes est au moins 300 fois le temps de création d'un agent.

Dans un système, le temps d'attente (de latence) des messages peut ne pas être négligeable. Cependant, dans notre approche, la plupart des messages sont asynchrones. Les messages synchrones apparaissent seulement quand les *LM* veulent synchroniser leurs étapes.

Comme nous simulons un système décentralisé, il est aussi important de parler de la quantité de données qui sont transférées sur le réseau. Bien que nos messages contiennent peu de caractères, le nombre de messages échangés entre les nœuds nécessite une attention puisque ce nombre est important dans un système décentralisé comme

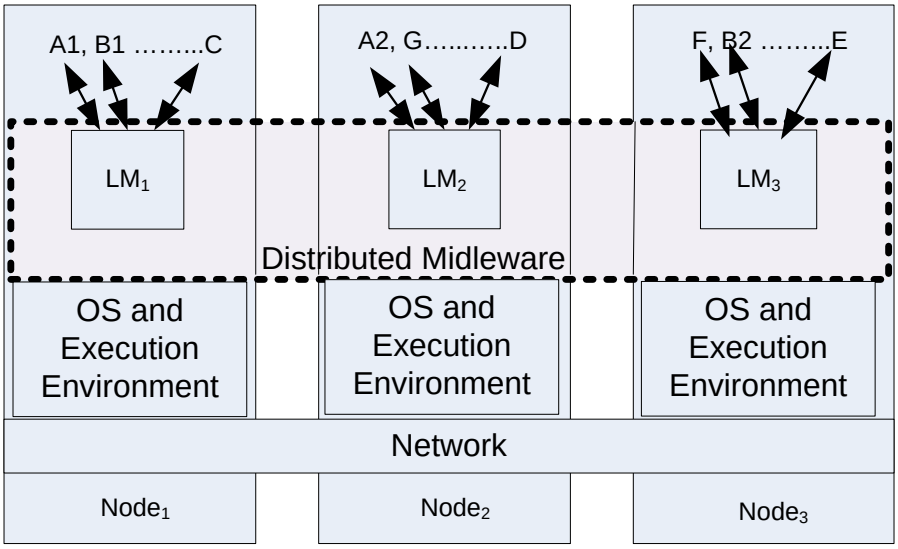

Figure 6-1: Architecture du Middleware

nous l'avons montré au chapitre 3. Un des objectifs du schéma d'emprunt présenté dans le chapitre 4 était la réduction du nombre de messages échangés entre les nœuds, ce qui diminue la quantité de données transférées.

6.2 Implémentation de notre approche

Pour évaluer notre approche des simulations ont été réalisées. Cette section décrit l'architecture générale de notre système multi-agent pour la gestion de la QoS et quelques détails d'implémentation.

6.2.1 Architecture du middleware

Le *middleware* proposé (Figure 6-1) est formé d'un ensemble de gestionnaires locaux (LM_i) sur les nœuds. Ce *middleware* utilise les services fournis par le système d'exploitation (OS), par le réseau et par l'environnement d'exécution (JADE) pour simuler notre approche. Les A_i, B_i, etc. sont des applications distribuées qui s'exécutent sur des nœuds et les *C*, *D*, *E*, *G* etc. sont des applications locales.

6.2.2 Architecture du gestionnaire local

Notre environnement est constitué d'un ensemble de nœuds reliés par un réseau. Un nœud est supposé être une machine ayant ses propres ressources telles que le CPU et la mémoire. La bande passante considérée comme une ressource globale est utilisée pour la communica-

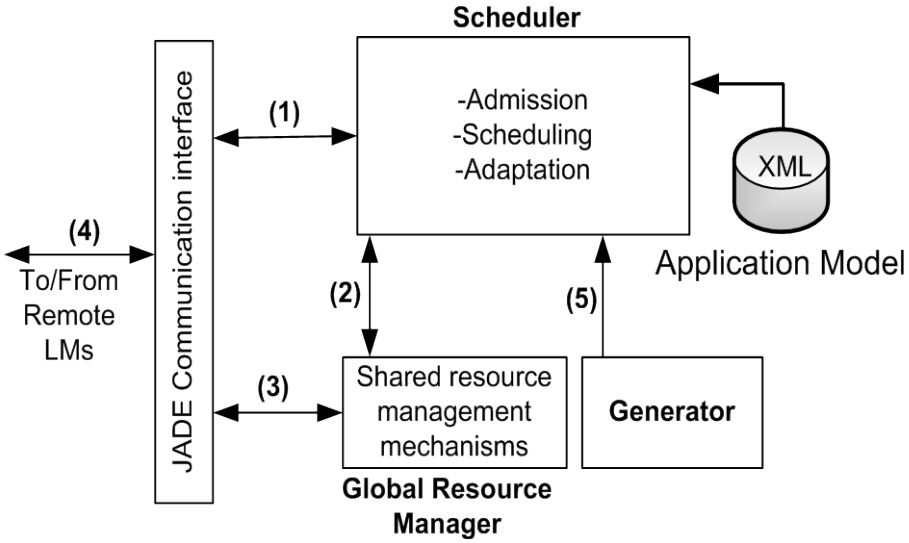

Figure 6-2: Architecture d'un LM sur un nœud

tion inter-nœuds. Sur chaque nœud, un gestionnaire local de QoS (*LM*), composé d'un ensemble d'agents dont l'architecture est représentée dans la Figure 6-2.

Un *LM* est composé de trois agents (Figure 6-2). L'agent *Scheduler* exécute les tâches d'admission et planifie l'utilisation des ressources. Selon l'approche utilisée (adaptation ou pas), il implémente le schéma d'emprunt de ressources et/ou les stratégies d'adaptation. L'agent *Scheduler* prend des décisions en se basant sur les informations du contexte d'exécution et les informations sur les applications (les modèles d'applications, fichier XML).

Le *Scheduler* communique (1) avec les autres *Schedulers* sur les nœuds distants pour exécuter des tâches d'admission, pour la synchronisation des étapes et pour coordonner l'adaptation de l'application distribuée sur plusieurs nœuds. Avant l'exécution de toute étape utilisant une ressource globale, il communique (2) avec l'agent *Global Resource Manager* qui implémente des mécanismes pour contrôler l'utilisation des ressources globales qui sont gérées collectivement (3) par tous les agents *Global Resource Manager* dans le système, via un protocole basé sur l'utilisation du jeton expliqué dans le chapitre précédent. Sur chaque nœud, un agent *Generator* est chargé de générer des applications selon une loi d'arrivée donnée (cf. section 6.2.5). Les agents utilisent une même interface (4) pour communiquer avec des agents distants sur d'autres nœuds.

Chaque application générée envoie un message au *LM* sur son nœud pour l'informer de son arrivée. Le *LM* enregistre l'application et sa description. La description contient toutes les informations dont le *LM* a besoin pour admettre l'application. L'admission d'une application commence le plus tôt possible. Celle d'une application locale dès son enregistrement alors que celle d'une application distribuée après que toutes ses parties soient arrivées et enregistrées sur leur nœud respectif. Si l'application est admise, le *LM* planifie les étapes de l'application et celle-ci attend un ordre du *LM* pour exécuter une activité. L'exécution d'une application distribuée suppose que toutes ses parties s'exécutent en parallèle sur leur nœud respectif sous le contrôle de leur *LM*. Par exemple, l'application de transfert de données, après avoir fait l'acquisition des données, les transfère pendant que l'autre partie sur un autre nœud reçoit les données puis les affiche.

En ce qui concerne les applications, elles sont intégrées dans une architecture d'agents comme suit : un seul agent implémente les comportements d'une application locale tandis que plusieurs agents implémentent ceux d'une application distribuée. Sur un nœud, plusieurs modèles d'applications sont générés (cf. section 6.2.5).

6.2.3 Planification des étapes

L'algorithme utilisé pour la planification est une heuristique basée sur EDF (Earliest Deadline First) pour ses propriétés : il est facile à implémenter, et s'il y a une solution au problème de planification, celle-ci est trouvée, sinon, la solution est arbitraire. Selon cette heuristique, les ressources sont réservées à l'admission d'une application. Chaque activité fournit ses besoins en ressources au pire des cas et son échéance. A partir de ces informations, un planning d'utilisation des ressources est construit dynamiquement.

Dans la *phase d'admission*, le *middleware* recherche dans le graphe de l'application un chemin d'exécution selon la disponibilité des ressources et les étapes des activités de ce chemin sont planifiées si le chemin est trouvé.

Le planning est modifié chaque fois qu'il y a de nouveaux événements, par exemple, admission d'une nouvelle application, arrivée du jeton pour l'exécution d'une étape synchronisée, fin de l'exécution d'une étape. Dès qu'un nœud est libre, l'étape suivante de la planification commence son exécution. La planification est approximative puisqu'elle ne garantit pas l'exécution d'une application distribuée.

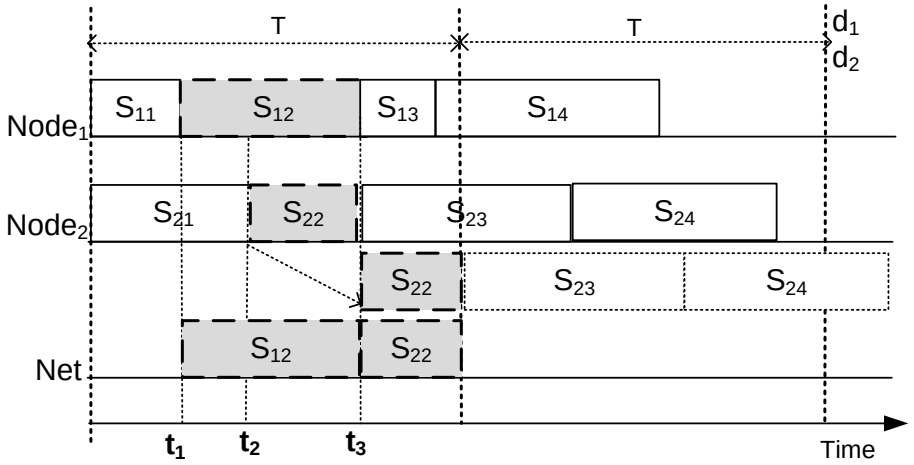

Figure 6-3: Conflits de synchronisation

La Figure 6-3 montre une situation conflictuelle dans laquelle l'exécution d'une application pourrait être annulée. Cette figure montre deux applications distribuées sur les nœuds *Node₁* et *Node₂* avec la planification de leurs étapes successives S_{11}, S_{12}, S_{13}, S_{14} et S_{21}, S_{22}, S_{23}, S_{24} respectivement et leur échéance d_1 et d_2 respectivement. Les étapes en pointillée S_{12} et S_{22} utilisent des ressources globales, ici la bande passante (*Net*). Les autres étapes utilisent uniquement des ressources locales. Ces deux applications veulent synchroniser leurs étapes S_{12} et S_{22} avec d'autres nœuds qui ne sont pas représentés ici par souci de simplicité. L'axe *Net* indique le temps d'utilisation de la ressource globale pour les deux nœuds.

Dans la Figure 6-3, au temps t_1, *Node₁* demande le jeton et l'obtient pour exécuter l'étape S_{12}. Pendant que *Node₁* utilise le jeton, *Node₂* à son tour demande le jeton au temps t_2 et l'attend. *Node₁* libère le jeton au temps t_3 et *Node₂* obtient le jeton pour commencer l'exécution de S_{22}, en supposant qu'il est prioritaire dans la liste d'attente. Dans tous les cas, *Node₂* n'exécutera pas l'étape S_{24} avant son échéance d_2. En effet, bien que les ressources soient disponibles, le décalage de S_{22} décale S_{23} et S_{24} et par conséquent, S_{24} ne sera pas exécutée avant la deadline d_2.

Cette situation est une conséquence de notre contrôle d'admission. Admettre les applications sur la base de périodes assure que les ressources disponibles sont suffisantes pour exécuter l'application en suivant un chemin d'exécution dans le graphe. Cependant, cette admission ne garantit pas l'ordre d'exécution dans une période. En effet, la planification est faite au moment de l'exécution pour choisir une étape à exécuter et des situations comme celle de la

Figure 6-3 peuvent faire que l'échéance ne soit pas respectée. Généralement, cette situation apparaît quand le système est chargé, ce qui augmente les étapes synchronisées avec des échéances rapprochées, ce qui diminue les degrés de liberté.

6.2.4 Choix d'implémentation

Dans nos simulations quelques décisions ont été prises :

- Chaque *LM* gère une file d'attente d'applications pour l'admission. L'admission a lieu à la fin de chaque étape;
- Les parties locales d'une application distribuées sont numérotées. Pour une application distribuée sur les *nœuds* $Node_i$ et $Node_{i+1}$, c'est le nœud $Node_i$ qui gère l'admission ;
- Pour les applications distribuées, le nœud qui procède à l'admission informe ceux qui ont les autres parties si l'admission de l'application a réussi ;
- Chaque *LM* exécute les étapes selon sa planification. Si l'étape est synchronisée, un message pour planifier la synchronisation est envoyé à l'autre *LM* qui a l'autre étape synchronisée correspondante ;
- Toute application dont l'admission a échoué est rejetée ;
- Si une étape a commencé, son exécution continue jusqu'à la fin sauf si un message de synchronisation arrive. Dans ce cas, l'étape courante est suspendue jusqu'à la fin de la synchronisation. Cette préemption ne concerne pas la mémoire car suspendre une étape et prendre une partie de sa mémoire pour une nouvelle étape est complexe à gérer.
- Chaque partie locale d'application est un agent logiciel.

6.2.5 Conditions expérimentales

Cette section donne des conditions sous lesquelles nos simulations ont été réalisées. Pour chaque *LM*, une description de l'environnement et de toutes les applications est disponible sur son nœud. Les résultats obtenus sont mesurés dans les conditions suivantes :

- La deadline de chaque application générée est définie comme égale à *2*SW + random [1..2*SW]*, avec *SW* la somme au pire cas d'exécution (*WCET : Worse Case Execution Time*) de toutes les durées des étapes constituant le chemin choisi dans l'unité de planification courante, du graphe de l'application ;

- Le temps d'exécution réel de chaque étape est égal à *WCET/3 + random [0…2*WCET/3]* ;
- La loi d'arrivée des applications est simulée par un temps aléatoire *random [0…C*], où *C* est un nombre positif et est l'écart entre deux arrivées. Plus C est petit, plus les applications arrivent fréquemment et plus la charge du système augmente ;
- Le temps au pire cas d'exécution des étapes des applications est défini de telle sorte que la somme des durées d'exécution en WCET de toutes les applications distribuées soit trois fois la somme des *WCET* de toutes les étapes des applications locales sur les deux nœuds;
- Pour une application distribuée respectivement sur les nœuds $Node_i$ et $Node_{i+1}$, l'admission commence dès l'arrivée de deux parties locales sur;
- Les nœuds sont numérotés de 0 à *N-1* où *N* est le nombre de nœuds dans le système. Pour chaque application distribuée générée, un nombre aléatoire i, $0 \leq i \leq N-1$, indique le nœud sur lequel sera exécutée la partie locale suivante de l'application distribuée ;
- Plusieurs modèles d'applications sont utilisés. Un modèle d'application correspond à un graphe d'exécution particulière. Les modèles sont numérotés de 0 à *M-1* où *M* est le nombre de modèles. Une application peut avoir 1, 2 ou 3 unités de planification Pour chaque application générée, un nombre j, $0 \leq j \leq M-1$, indique son modèle correspondant.
- Les simulations ont été faites sur une machine. Les nœuds sont simulés par des conteneurs JADE.

6.2.6 Scénario typique d'exécution d'une application

Cette section décrit l'exécution d'une application distribuée pour montrer les différents traitements dans les deux phases. Cette application, est distribuée sur les nœuds $Node_1$ et $Node_2$. L'admission et l'exécution de cette application sur les deux nœuds se passent de la manière suivante :

1) A l'arrivée de la première partie locale (*Part1*) de l'application sur le nœud $Node_1$, ce dernier l'enregistre, accède à sa description et procède à l'admission de l'application en utilisant une vue de la disponibilité des res-

sources du système sur son nœud. Si les quantités de ressources disponibles sont supérieures à celles exigées par l'application, celle-ci est admise, sinon des procédures pour demander des ressources additionnelles sont exécutées et des requêtes sont envoyées aux nœuds concernés par les ressources manquantes. Dans la suite nous supposons que l'application est admise. Un message est envoyé au nœud ($Node_2$) qui contrôle la seconde partie (*Part2*). Chacun des deux nœuds planifient l'exécution de l'unité *Start* qui correspond à l'exécution de *Load*, c'est-à-dire, le chargement des parties locales sur les deux nœuds. Dans nos simulations, l'exécution d'une étape *Load* correspond à la création d'un agent.

2) Après la création de ces deux agents représentant les parties locales de l'application distribuée, LM_1 sur le nœud $Node_1$, s'il y a des ressources, choisit un chemin dans l'unité de planification suivante, *Sample*, après son admission. LM_1 planifie les étapes des activités du chemin choisi. Supposons que dans l'unité *Sample*, les chemins n'ont qu'une seule activité qui peut être exécutée selon les modes 1, 2 ou 3. Nous supposons que l'activité de mode *3* et d'utilité *80* est choisie. Les étapes *Acquisition*, *Compression2*, *Transfert3* sont planifiées sur le nœud $Node_1$. Le nœud $Node_2$ est informé pour planifier à son tour les étapes *Reception3*, *Decompression2*. Cependant, *Node2* n'exécutera pas *Decompression2* tant que la synchronisation entre *Transfert3* et *Reception3* n'a pas eu lieu.
3) Dans la phase d'exécution, le *LM* du $Node_1$ vérifie si les ressources disponibles ne permettent pas d'augmenter l'utilité de l'application en essayant de planifier un autre mode. Dans l'affirmative le gestionnaire LM_1 sur $Node_1$ re-planifie les étapes de la nouvelle activité, par exemple l'activité de mode 1 et d'utilité 100. Les étapes *Acquisition*, *Transfert1* sont planifiées sur $Node_1$ et *Acquisition* est exécutée. Quand le moment d'exécuter *Transfert1* arrive, c'est-à-dire que $Node_1$ a demandé et obtenu le jeton, $Node_1$ envoie un message à $Node_2$ pour préparer la synchronisation entre *Transfert1* sur $Node_1$ et $Reception_1$ sur $Node_2$. Ce dernier remplace les activités *Decompression2*, *Reception3* et *Affichage3* (les étapes de l'ancien mode 3) par *Reception1*, *Affichage1* (les étapes du nouveau mode 1). Dans ce cas l'utilité de l'activité augmente et passe de 80 à 100.

4) A la fin de l'exécution des étapes synchronisées, *Part1* sur $Node_1$ et *Part2* continuent leur exécution ou la termine.
5) Un cas simple de l'application est celle qui termine son exécution après l'exécution de l'unité *Sample*. Dans ce cas *Part1* se termine après avoir exécuté *Transfert1* sur $Node_1$ et *Part2* après avoir exécuté l'étape *Display1* sur $Node_2$. Les deux agents correspondants sont supprimés.

Le cas expliqué concerne l'exécution d'une seule application distribuée dont le graphe d'exécution modélise les unités *Start* et *Sample*. Chaque étape est exécutée en respectant le planning sur chaque nœud et sur chaque nœud plusieurs applications sont en exécution. Le générateur d'applications génère des applications locales ou distribuées sur chaque nœud.

6.3 Résultats des simulations

Cette section montre les résultats de nos différentes simulations. Nous subdivisons cette section en deux parties. La première partie présente les résultats obtenus lors des simulations de notre schéma d'emprunt sans adaptation, la seconde partie avec adaptation du comportement des applications.

Les résultats concernent principalement l'utilisation du CPU, le nombre d'applications admises (*Nbre_App*) et le nombre de messages échangés (*Nbre_messages*).
Pour l'utilisation du CPU nous distinguons trois aspects :

- Le temps CPU utilisé par toutes les applications dans le système (*CPU_utilisé*), c'est-à-dire toutes les applications admises et exécutées ;
- Le temps CPU utilisé par les applications qui ont été admises et finalement interrompues parce qu'elles n'ont pas été exécutées avant leur deadline (*CPU_perdu*) ;
- Et le temps utile (*CPU_utile*), utilisé par les applications qui ont été exécutées avant leur deadline. C'est la différence entre le temps utilisé (*CPU_utilisé*) et le temps perdu (*CPU_perdu*).

Pour prouver l'efficacité de notre approche, celle-ci est comparée à la politique de best-effort (*Best_Effort*) dans laquelle les tâches d'admission ne sont pas exécutées et toutes les applications générées sont admises.

6.3.1 Approche sans adaptation

La Figure 6-4 compare notre approche (*Notre_Approche*) qui implémente le schéma d'emprunt et la politique de *best effort (Best_Effort)*. Les résultats montrent l'utilisation du temps CPU en fonction de la charge du système. Dans l'approche proposée (*CPU_Utilisé_Notre_Approche*), le pourcentage de *CPU* n'est pas égal à 100% parce que la *phase d'admission* rejette les applications si les ressources disponibles ne sont pas suffisantes pour les accepter et les exécuter dans le système. Cette figure montre que dans la politique du *best effort*, le CPU utilisé (*CPU_Utilisé_Best-Effort*) est plus grand puisque cette politique accepte toutes les applications générées. Cependant, plus d'applications ont été interrompues parce qu'elles n'ont pas respecté leur deadline, entrainant ainsi un temps perdu (*CPU_Perdu_Best_effort*) plus grand que dans notre approche (*CPU_Perdu_Notre_Approche*).

La Figure 6-5 montre le CPU utile pour les deux politiques comparées dans la Figure 6-4. Le CPU utile est plus petit en utilisant la politique *Best-effort (CPU_Utile_Best_Effort)*. Tout temps passé depuis le début de l'exécution de l'application jusqu'à son interrup-

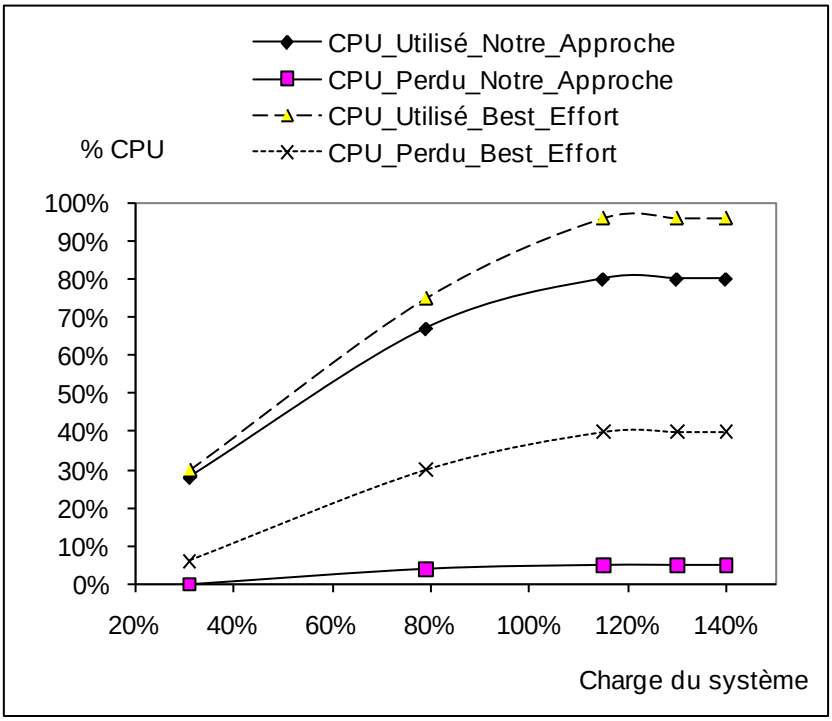

Figure 6-4: Utilisation du temps CPU

tion est considéré comme perdu. Il est possible de réduire le temps perdu dans notre approche en appliquant un fort contrôle d'admission, ce qui réduit le nombre d'applications admises et le nombre des applications interrompues, mais cela réduit aussi le CPU utile. La Figure 6-6 montre donc que notre approche basée sur le schéma d'emprunt sans adaptation est plus efficace que le *best effort*.

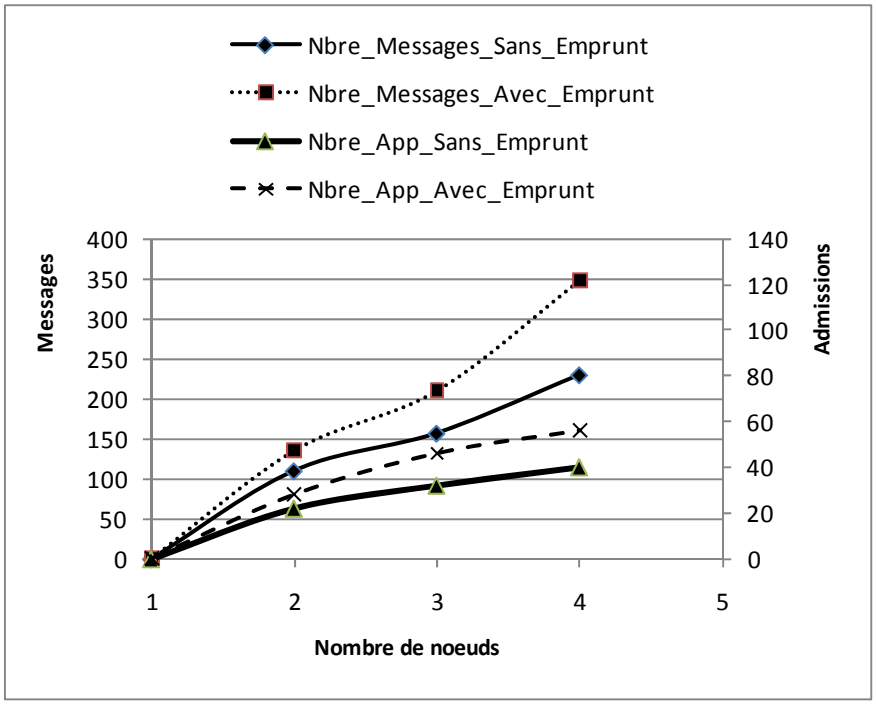

Figure 6-5: Nombre d'applications admises et nombre de messages en fonction du nombre de nœuds

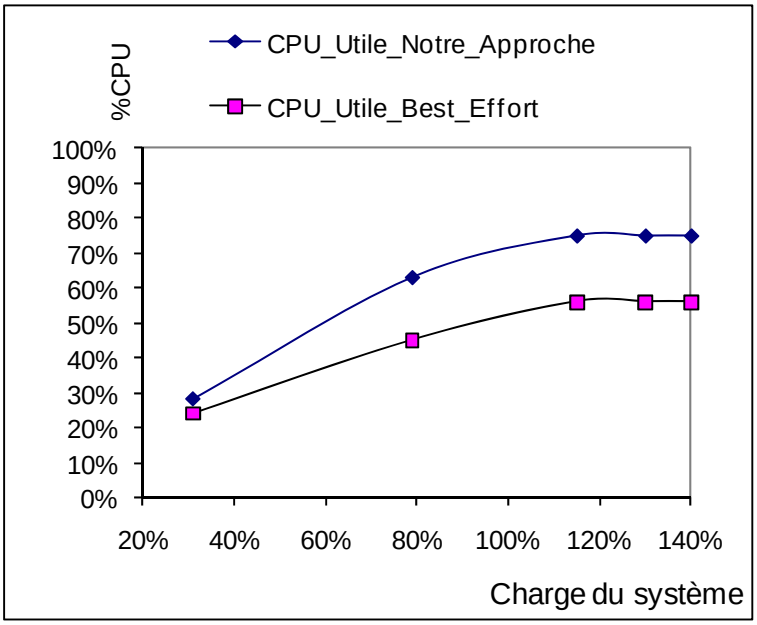

Figure 6-6 : Temps CPU Utile

Dans la Figure 6-6, deux politiques particulières de notre approche ont été comparées : l'approche proposée avec emprunt dynamique (*Avec_Emprunt*) et l'approche proposée sans emprunt dynamique (*Sans_Emprunt*). Dans la première politique, dans la phase d'admission, le LM formule des requêtes (emprunt dynamique) pour demander des ressources additionnelles si les quantités de ressources empruntées disponibles ne sont pas suffisantes pour admettre l'application. La Figure 6-6, montre le nombre d'applications distribuées admises et le nombre de messages échangés dans les deux situations en fonction du nombre de nœuds. Cette figure montre que les emprunts dynamiques augmentent le nombre d'applications distribuées admises (*Nbre_App_Avec_Emprunt* > *Nbre_App_Sans_Emprunt*, Figure 6-6). Ce résultat était attendu parce que les emprunts dynamiques implémentés concernent uniquement les applications distribuées, par conséquent ils favorisent les admissions de ces applications. Cependant, l'augmentation du nombre d'applications admises a un coût en nombre de messages (*Nbre_Message_Avec_Emprunt* > *Nbre_Message_Sans_Emprunt*, Figure 6-6).

Pour finir cette section, la Figure 6-7 illustre l'influence du pourcentage des *ressources empruntées* dans notre approche pour une charge de 60% et pour trois nœuds. Le pourcentage d'emprunt de *X%* signifie que chaque nœud ou *LM* prête X% de ses ressources locales à chacun des autres nœuds (*LM*) et garde le reste (*100-N_Nodes*X*)% pour ses utilisations, où *N_Nodes* est le nombre de nœuds dans le système. Quand *X* augmente, les pourcentages des *ressources empruntées* augmentent pendant que les pourcentages des *ressources réservées* diminuent.

Dans la Figure 6-7, l'utilisation de l'emprunt dynamique augmente le nombre d'applications distribuées admises, donc le nombre total d'applications admises (distribuées et locales). Cette même figure montre aussi que quand le pourcentage d'emprunts atteint 40%, il y a moins d'applications admises. Ceci s'explique par le fait que la quantité des *ressources réservées* est trop petite pour admettre à la fois des applications locales et distribuées, et les requêtes pour la demande de ressources additionnelles (emprunts dynamiques) sont envoyées quand les ressources locales sont suffisantes seulement, c'est-à-dire que l'emprunt dynamique concerne uniquement les ressources distantes pour simplifier notre implémentation. L'implémentation des procédures pour augmenter les quantités de ressources locales devrait peut être changer la donne.

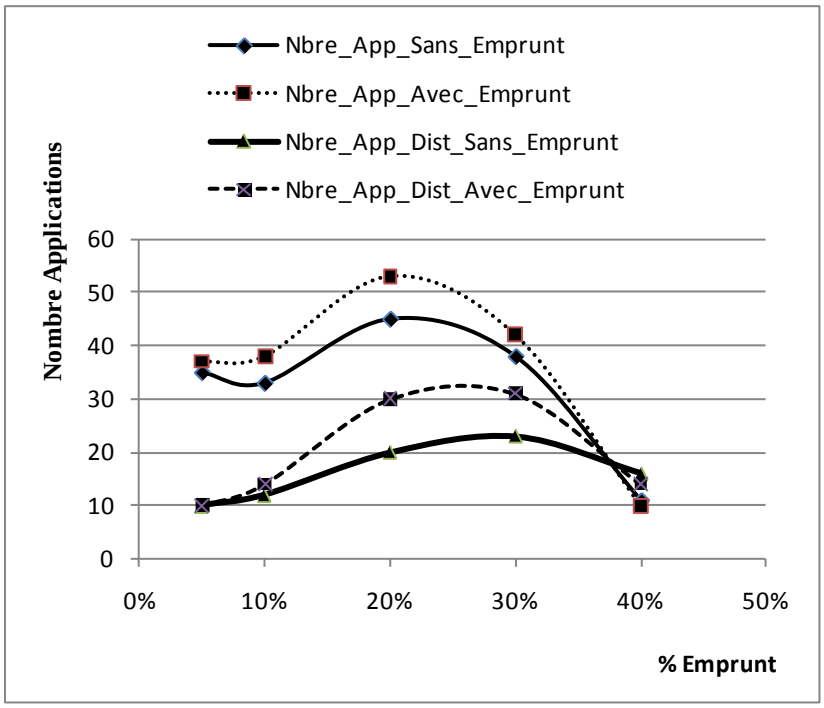

Figure 6-7: Nombre d'applications en fonction du pourcentage d'emprunt

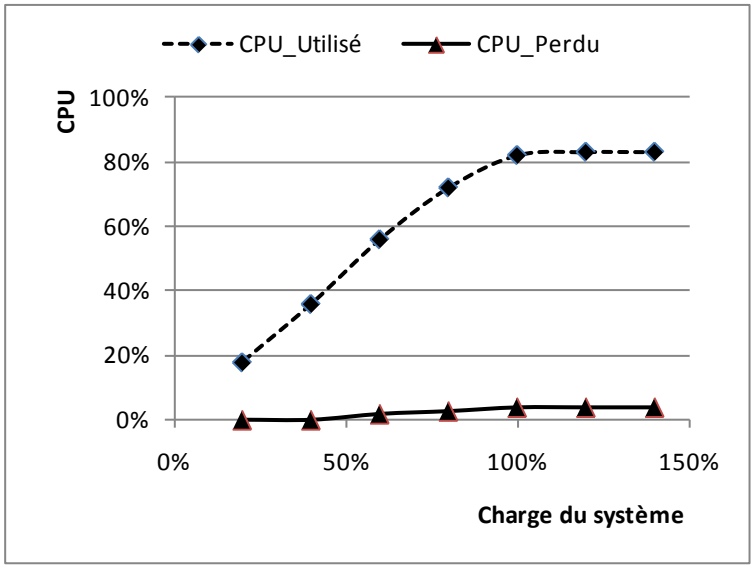

Figure 6-8: Temps CPU utilisé dans l'approche avec adaptation

6.3.2 Approche avec adaptation

Cette section a pour objectif de montrer l'intérêt de l'adaptation. Pour chaque application, un chemin d'exécution est choisi en fonction des ressources disponibles.

La Figure 6-8 montre l'utilisation du CPU en fonction de la charge du système. Le CPU utilisé (*CPU_Utilisé*) par toutes les applications se stabilise à ≈ 83% pendant que le CPU perdu (*CPU_Perdu*) est ≈3%.

La comparaison avec la politique *best-effort* (*CPU_Utilisé_Best_Effort* et *CPU_Utile_Best_Effort*), Figure 6-9, montre que l'approche avec adaptation (*CPU_Utilisé_Adaptation* et *CPU_Utile_Adaptation*) est efficace. En effet, le *best-effort* qui admet toutes les applications générées a un CPU utile très inferieur car beaucoup d'applications sont interrompues parce qu'elles ont violé leur échéance.

La Figure 6-10 compare deux variantes de notre approche. La première effectue l'adaptation, c'est-à-dire admet les applications avec la plus petite utilité et augmente cette utilité selon la disponibilité des ressources (*CPU_Utile_Adaptation*) dans la *phase d'exécution*. La seconde, notre approche sans adaptation (uniquement la politique d'emprunt), consiste à admettre et à exécuter les applications avec la

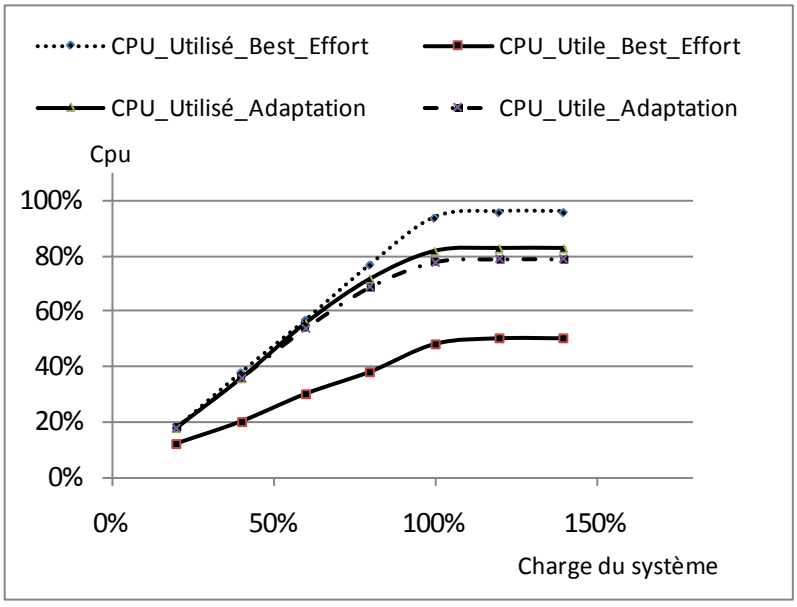

Figure 6-9: Best-effort vs. Notre approche avec adaptation

plus haute utilité (*CPU_Utile_Emprunt*). C'est une sorte de *best effort* qui effectue un test d'admission. La comparaison est basée sur le temps CPU utile dans ces deux approches. Nous remarquons, Figure 6-11, qu'à partir d'environ 50% l'utilité du système diminue avec la politique *Adaptation* (*Q_Adapt* < 1, Figure 6-11) quand le système devient de plus en plus chargé alors qu'avec la politique *Emprunt*, les applications sont toujours exécutées avec la plus haute utilité (*Q_Emprunt* = 1, Figure 6-11). L'utilité globale Q du système, est calculée en faisant la somme des utilités des applications divisée par le nombre d'applications dans le système. Figure 6-10 indique que l'adaptation (*CPU_Utile_Adaptation*) améliore le temps CPU utile dans le système (≈6% de différence).

En plus, la Figure 6-12, le nombre d'applications exécutées

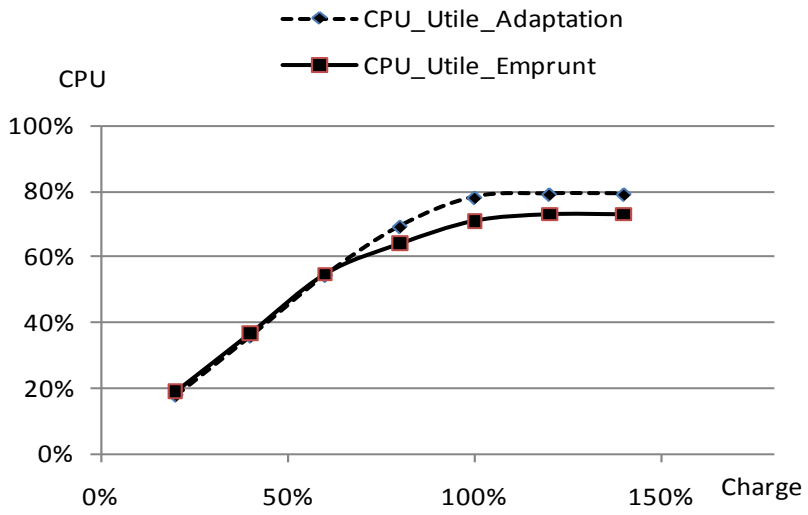

Figure 6-10: Adaptation vs. Sans adaptation

avant leur échéance dans l'approche avec adaptation (*Completed_Adapt*) est supérieur à celui dans l'approche sans adaptation (*Completd_Emprunt*) quand le système devient chargé, à partir ≈50%. Cela s'explique par le fait que, pendant la *phase d'admission*, la politique *Emprunt* rejette des applications qui pouvaient être exécutées si la politique *Adapt* avait été adoptée, ce qui explique aussi la différence dans la Figure 6-10.

Ces résultats montrent que l'approche avec adaptation est efficace.

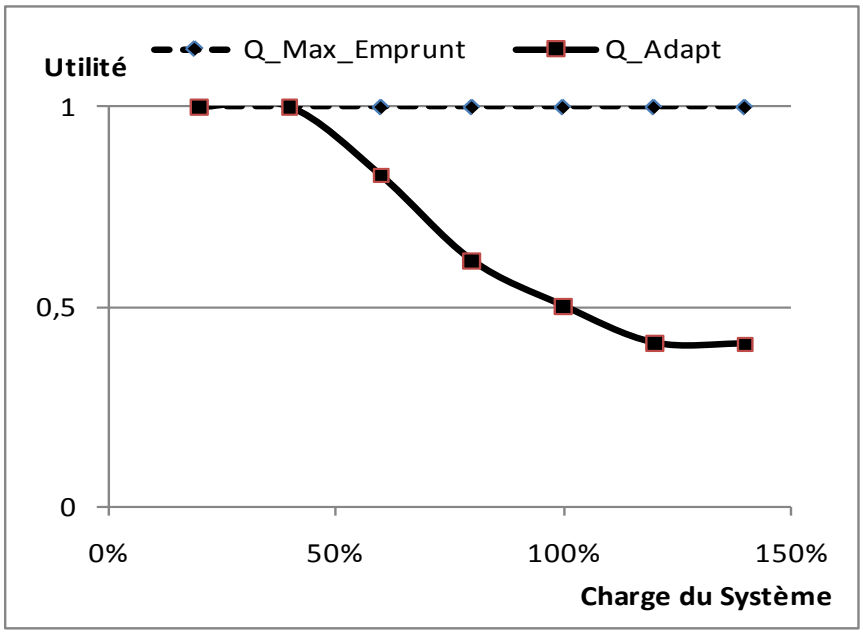

Figure 6-11: Utilité du système

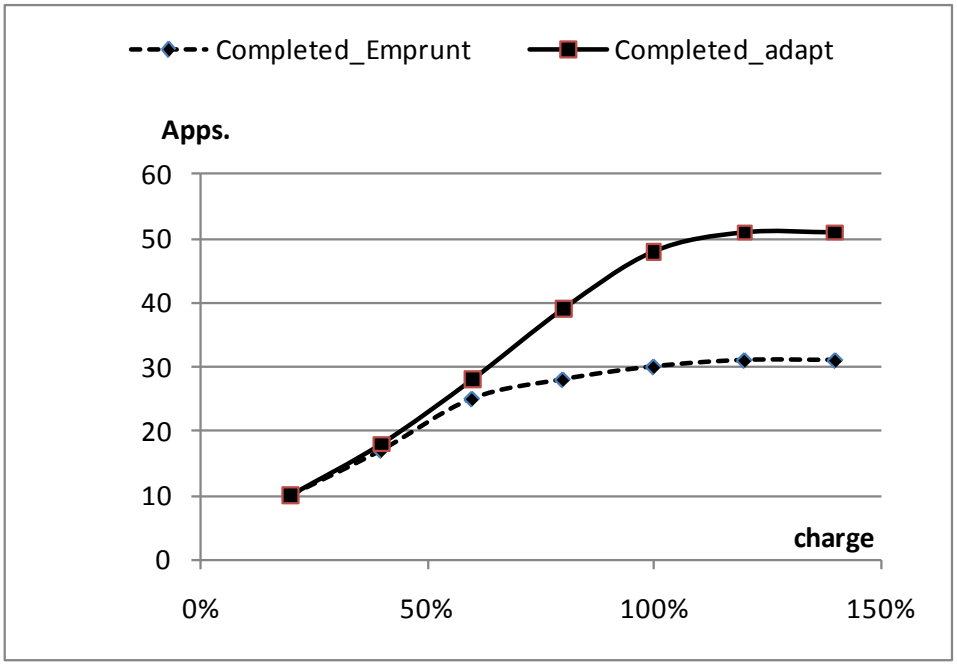

Figure 6-12: Nombre d'applications admises: Adaptation vs. sans adaptation

6.4 Conclusion

Dans ce chapitre, nous avons présenté la technologie et l'environnement pratique pour simuler notre solution. Une architecture physique et logicielle de l'approche a été décrite. Plusieurs poli-

tiques ont été simulées et comparées. Les résultats montrent l'efficacité de l'approche.

Chapitre 7 Conclusion et perspectives

Les travaux que nous avons présentés dans ce mémoire s'inscrivent dans la problématique générale de gestion de la QoS des applications distribuées. L'objectif principal de cette thèse était de proposer et d'examiner une approche totalement décentralisée pour répondre aux besoins en QoS des applications distribuées s'exécutant dans un environnement où la disponibilité des ressources varie de façon imprévisible. Dans cette optique, nous avons abordé le problème de l'adaptation dynamique des applications possédant un certain degré de liberté dans leur fonctionnement.

Dans la suite, la section 7.1 fait le bilan de notre travail et la section 7.2 liste quelques perspectives.

7.1 Bilan

Dans notre démarche pour la recherche d'une solution de gestion de la QoS, nous avons franchi plusieurs étapes décrites comme suit.

Dans un premier temps nous avons fait une revue des notions et des solutions existantes pour la gestion de la QoS afin de cerner notre domaine de recherche. Dans un environnement d'exécution ouvert où la loi d'arrivée des applications n'est pas connue, l'exécution des applications dans des conditions optimales est incertaine. La variation du contexte d'exécution impose la prise en compte de nouveaux aspects. Cela englobe non seulement des aspects purement fonctionnels mais également des aspects non-fonctionnels tels que la disponibilité de ressources, la sécurité etc. La gestion de l'ensemble de ces aspects définit la QoS.

Afin de fournir une approche décentralisée de gestion des ressources, trois architectures de gestion de la QoS dans les environnements distribués ont été étudiées. La question était de savoir quelles sont les conséquences d'une planification décentralisée. Pour cela, nous sommes partis d'une architecture centralisée existante et nous avons considéré deux approches décentralisées. La comparaison a été faite en fonction du nombre de messages échangés entre les nœuds. Cette étude nous a permis de mettre en évidence le nombre de messages élevé dans les approches décentralisées. En effet, les gestionnaires n'ont qu'une vue locale du système, et pourtant ils doivent coordonner leurs décisions pour optimiser la QoS globale du système. Ainsi, une contribution a consisté à proposer une solution de gestion de la QoS qui minimise le nombre de messages.

Rendre locale une vue détaillée de la disponibilité des ressources du système dans une approche décentralisée n'est a priori pas évidente et le chapitre 4 a proposé une solution. En se basant sur un schéma d'emprunt de ressources, les gestionnaires locaux construisent localement cette vue pour admettre les applications et contrôler leur exécution. Tout au long du processus de gestion des ressources les messages ne sont envoyés que si c'est nécessaire.

Dans le chapitre 5, la solution a été complétée en ajoutant un support pour l'adaptation du comportement des applications pour gérer finement la QoS et faire face aux variations de disponibilité des ressources de l'environnement. Dans ce contexte, nous avons décrit un modèle général des applications distribuées. Ce modèle expose différents comportements des applications pour fournir le même service avec différents besoins en ressources. La richesse du modèle proposé permet aux concepteurs d'applications de prendre en compte les spécificités des applications pour fournir différents modes de fonctionnement. Le contrôle des applications s'appuie sur leur description et de celle de l'environnement. Le modèle d'une application est fourni en utilisant le formalisme UML pour décrire de façon détaillée l'exécution des applications et de leurs besoins en QoS. La solution ainsi construite permet aux applications de s'adapter dynamiquement en fonction de la disponibilité des ressources. L'adaptation est basée sur les modes de fonctionnement et est effectuée aux points prédéfinis où une activité avec la plus haute utilité est choisie. Une coordination entre les nœuds assure une adaptation globale cohérente de l'application. Cependant, dans cette solution, un effort du coté des concepteurs d'applications est fondamental. En effet, les concepteurs doivent fournir des informations très précises sur les applications pour que la gestion soit la plus efficace et/ou la plus fine possible. Sachant que toute application peut être modélisée sous forme de graphe ou du moins un chemin dans un graphe, et que toute modification dans l'environnement d'exécution, par exemple les caractéristiques des nœuds et les spécificités des applications, sont prises en compte par le concepteur lors de la construction des applications, la solution proposée peut être utilisée dans un autre domaine.

La faisabilité de nos propos a été montrée dans le chapitre 6. Les résultats trouvés montrent l'efficacité de l'approche et l'intérêt de l'adaptation.

Néanmoins le travail réalisé ici n'est qu'une première approche et ne prétend en rien être la solution de la gestion de la QoS dans les systèmes distribués

7.2 Perspectives

De nombreux aspects n'ont pas été pris en compte mais mériteraient d'être approfondis. Nous restons persuadés que plusieurs perspectives peuvent être envisagées.

Il faudrait faire une implémentation complète afin d'évaluer dans leur intégralité les performances. Des simulations intensives et de nombreux paramètres doivent être testés. Par exemple, la prise en compte des mécanismes de libération automatique des ressources, la variation automatique de la taille des périodes, la variation du nombre de nœuds, etc.

En l'état actuel des choses, la simulation a été faite en supposant une architecture à un niveau et un nombre réduit de nœuds. Il serait intéressant de vérifier l'application de cette approche dans de grands systèmes. Par exemple, diviser le système en sous-réseaux ou domaines et pouvoir appliquer l'approche dans chaque regroupement et dans l'ensemble des regroupements qui composent le système.

Le protocole de gestion des ressources globales peut être amélioré afin de réduire encore le nombre de messages générés. Par exemple, proposer des mécanismes qui permettent l'envoi d'un message à un groupe de nœuds (multicast) plutôt que d'envoyer des messages point à point, plus coûteux.

Pour pouvoir adapter le comportement des applications notre *middleware* utilise un fichier XML qui décrit tous les services que les applications offrent aux utilisateurs. Il serait aussi utile de proposer un outil pour aider les concepteurs d'applications à prendre en compte et évaluer les exigences QoS des applications et fournir une spécification QoS correspondante. Par exemple, introduire dans le *middleware* des évaluateurs pour obtenir des informations quantitatives et qualitatives permettant de supprimer certains arcs dans le graphe et de ne garder que des arcs utiles pour le système et de générer de façon automatique ou semi-automatique la description correspondante.

Dans l'approche, tout service dégradé est mieux que rien du tout. Un problème voisin serait de voir s'il n'est pas intéressant d'intégrer l'utilisateur dans le processus d'adaptation des applications pour qu'il puisse modifier des décisions prises de façon automatique. Dans ce cas, il serait utile, par exemple, d'introduire des points (au début d'un *SU* par exemple) où le système communique avec l'utilisateur sur le niveau de service qui sera potentiellement fourni

afin qu'il puisse décider l'acceptation ou non de l'exécution de l'application pour fournir ce niveau de service.

Bibliographie

[AAB+02] **Dorian Arnold, Sudesh Agrawal, Susan Blackford, Jack Dongarra, Michelle Miller, Kiran Seymour, Kiran Sagi, Zhiao Shi, Sathish Vadhiyar** « Users' Guide to NetSolve V1.4.1 Innovative Computing Dept », Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, Juin 2002.

[AAS00] **Tarek F. AbdelZaher, Ella M. Atkins, Kang G. Shin** « QoS Negotiation in Real-Time System and Its Application to Automated Flight Control », IEEE Transactions on Computers, pp. 1170-1183, 2000.

[AAS+02] **Tarek F. Abdelzaher, Ella M. Atkins, Kang G. Shin, Nina Bhatti** « Performance Guarantees for Web Server End Systems: A Control-Theoretical Approach » IEEE Transactions on Parallel and Distributed Systems, pp. 80-96, 2002.

[AAS99] **E.M Atkins, T.F. Abdelzaher, K.G. Shin**, « Planning and Resource Allocation for Hard Real-time, Fault-Tolerance Plan Execution » Proceedings of the International Conference on Autonomous Agents (Agents'99), ACM Press, pp. 244-251, 1999.

[ACH98] **C. Aurrecoechea, A.T. Campbell, L. Hauw** « A survey of QoS Architectures, Multimedia Systems » Vol. 6, Spring-Verlag, pp.138-151, 1998.

[AFJ02] **Jan Oyvind Aagedal, Earl F., Ecklund Jr.**, « Modeling QoS: Towards a UML profile », Springer-Verlag, LNCS 2460, pp. 275-289, 2002.

[AG92] **N. A. Avouris, L. Gasser** « Distributed Artificial Intelligence: Theory and praxis », chapter Object Oriented Concurrent Programming and Distributed Artificial intelligence, Kluwer Academic Publisher, pp. 81-108 1992.

[AGR98] **Hafid A., Bochmann G. et Dssouli R.**, « Distributed Multimedia Application and Quality of Service: A Review », Electronic Journal on Networks and Distributed Processing, vol. No.6, pp. 1-50, 1998.

[AsJ] http://.eclipse.org/aspectj

[ANA+08] **F. Alhalabi, B. Narkoy, R. Aubry, M. Maranzana, L. Morel and J.-L Sourrouille,** « Centralized vs. Decentralized QoS Management Policy », third IEEE Inter. Conf. on Information and Communication Technologies: From Theory to Applications, pp. 1-6, ICCTTA, 2008.

[BCS02] **E. Bruneton, T. Coupaye et J.B Stefani**, « Recursive and Dynamic Software Composition with Sharing » **In** Seventh International Workshop on Component-Oriented Programming (WCOP02), ECOOP2002, Malaga, Spain, Juin 2002.

[BEL 00] **Bellifemine F., Giovani C., Tiziana T. Rimassa G**., « Jade Programmer's Guide », Jade version 2.6, http://sharon.cselt.it/projects/jade/, 2000.

[BEL 99] **Bellifemine F., Poggi A., Rimassa G.**, « JADE: A FIPA-compliant agent framework », CSELT internal technical report. Part of this report has been also published in Proceedings of PAAM'99, London, pp.97-108, April 1999.

[BH97] **Gregor v. Bochmann, and Abdelhakim Hafid**, « Some Principles for Quality of Service Management », Distributed Systems Engineering Journal, Vol. 4, pp. 16-27(12), 1997.

[Bla05] **Christophe Blaess**, « Programmation système en C sous Linux », Eyrolles, Paris, seconde édition, 964 pages, 2005.

[BNB+98] **S. Brandt, G. Nutt, T. Berk, J. Mankovich,** « A Dynamic Quality of Service Middleware Agent for Mediating Application resource usage », RTSS, pp. 307-317, 1998.

[BS10a] **Narkoy Batouma, Jean-Louis Sourrouille**, « A Decentralized Resource Management Using a Borrowing Schema », Proceedings of the eighth ASC/IEEE International Conference on Computer Systems and Applications (AICCSA), pp. 1-8, 2010.

[BS10b] **Narkoy Batouma, Jean-Louis Sourrouille,** « Framework for Behavior Adaptation of Distributed Application », Proceedings of the 2010 International Conference on Grid and Distributed Computing, Control and Automation, Jeju Island, south Korea, Vol. 121, pp. 42-53., 2010.

[BS11] **Narkoy Batouma, Jean-Louis Sourrouille,** « Dynamic Adaptation of Resource Aware Distributed Application », International Journal on Grid and Distributed Computing (IJGDC), Juin 2011, A paraitre.

[CBF03] **Casmin Carabelea, Olivier Boissier et Adina Florea**, « Autonomy in multi-Agents Systems : A classification Attempt » **In**: Agents nad Computational Autonomy, Lecture Notes in Computer Science, pp. 103-113, 2003.

[CDD] **Kendra Cooper, Lirong Dai, Yi Deng,** « Modeling performance as an aspect: a UML based Approach » In the 4th AOSD Modeling With UML Workshop, 2003.

[CDF+04] **Franck Cappello, Samir Djilali, Gilles Fedak, Thomas Herault, Frederic Magniette, Vincent N´eri et Oleg Lodygensky,** « Computing on Large Scale Distributed Systems : XtremWeb Architecture, Programming Models, Security, Tests and Convergence with Grid », In FGCS Future Generation Computer Science, volume 21, pp. 417-437, mars 2004.

[CJC+00] **I. Cardei, R. Jha, M. Cardei, and A. Pavan,** « Hierarchical Architecture for Real-Time Adaptive Resource Management », Middleware'02, LNCS 1795, Springer Verlag, pp. 415-434, 2000.

[CK00] **Fangzhe Chang et Vijay Karamcheti,** « Automatic Configuration and Run-time Adaptation of Distributed Applications », In: High Performance Distributed Computing, Pittsburg, PA., IEEE Computer Society, pp. 11-20, 2000.

[CQG02] **E. Cortese, F. Quarta, G. Vitaglione,** « Scalability and Performance of JADE Message Transport System », Centri Direzionale isola F7, Telecom Italia, 2002.

[CS03] **V. G. Claudia and Z. Shlomo,** « Optimizing information exchange in cooperative multi-agent systems », ACM, International Conference on Autonomous Agents, Proceedings of the 2th international joint conference on Autonomous agents and multi-agent systems, pp. 137 – 144, 2003.

[CSS+97] **Saurav Chatterjee, Jerry Sydir, Bikash Sabata et Thomas Lawrence**, « Modeling Application for Adaptive QoS-based Resource Management », In: Proceedings of the 2nd High-Assurance Systems Engi-

neering Workshop, Washington, DC, USA, IEEE Computer Society, pp. 194-201, 1997.

[CTG+04] **K. Chmiel, D. Tomiak, M. Gawinecki, P. Karczmarek, M. Szymczak, M. Paprzycki,** « Testing the Efficiency of JADE agent platform », Proceedings of the 3th Inter. Symposium on Parallel and Distributed Computing, IEEE, pp.49-56, 2004 .

[DG01] **Hans Domjan et Thomas R. Gross,** « extending a Best Effort Operating System to provide QoS Processor Management », In: Proceedings of the 9th International Workshop on Quality of Science, London, UK Springer-Verlag, pp. 92-106, 2001.

[DHT+98] **B. Dumant, F. Horn, F. Dang Tran, J-B. Stefani,** « Jonathan: An Open Distributed Processing Environment in Java », Middleware'98: IFIP International Conference on Distributed Systems Platforms and Open distributed Processing, Sept. 1998.

[DL02] **Pierre-Charles David et Thomas Ledoux**, « An Infrastructure for Adaptable Middleware », In: On the Move to Meaningful Internet Systems, DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002. Springer-Verlag, volume 2519 de Lecture Notes in Computer Sciences, pp. 773-790, 2002.

[DL03] **P.C. David et T. Ledoux**, « Towards a framework for self-Adaptive Component Based Applications », In: Proceedings of International Conference on Distributed Applications and Interoperable Systems 2003, the 4th IFIP WG6., DAIS 2003, volume 2893 of LNCS, Spring-Verlag, pp. 1-14, 2003.

[DR01] **Hans Domjan et Thomas R. Gross,** « Extending a best-Effort Operating System to provide QoS Processor management », In: Proceedings of the 9th International Workshop on Quality of Service, London, UK, Springer-Verlag, pp. 92-106, 2001.

[DUR] **E. H. Durfee,** « Blissful Ignorance: Knowing Just Enough to Coordinate Well », Proceedings on the First International Conference on Multi-Agents Systems (ICMAS), pp. 406-413, 1995.

[EJB] http://www.oracle.com/technetwork/java/javaee/ejb/index.html

[Fer95] **Jacques Ferber,** « Les systèmes multi-agents: vers une intelligence collective », Inter Editions, Collection Dunod Masson Ho, Editeur Dunode, ISBN 2-7296-0665-3,522 pages, 1995**.**

[FIP09] FIPA, http://www.fipa.org, accessed 22th may 2009.

[Fou00] **Bruno Fouquet,** « Gestion de la qualité de service : réseaux, serveurs et applications », Eyrolles, Collection solution d'entreprises, 1ère édition, ISBN-10 : 2212092407, ISBN-13 : 978-2212092400, 251 pages, 2000.

[FK98] **Svend Frolund et Svend Koistinen,** « Quality of Service Specification in distributed object systems », Distributed Systems Engineering Journal, Volume 5, N° 4, pp. 179-202, 1998.

[GDR05] **Jean-Philippe Georges, Thierry Divoux, Eric Rondeau,** « Confronting the performance of a switched Ethernet network with industrial constraints by using the network calculus », International Journal of Communication Systems, pp. 877-903, 2005.

[GDR+05] **Jean-Philippe Georges, Thierry Divoux et Eric Rondeau,** « Strict priority versus Weighted Fair Queuing in Switched Ethernet Networks for Time Critical Applications », In : International Parallel end Distributed Processing Symposium, Denver, CA IEEE Computer Society, pp. 141-141, 2005.

[GHR+04] **Sourav Ghosh, Jeffery P. Hansen, Ragunathan Rajkumar et John P. Lehoczky,** « Integrated Resource Management and Scheduling with Multi-Resource Constraints », In: Real-Time Systems Symposium, Lisbon, Portugal, IEEE Computer Society, pp. 12-22, 2004.

[GMM98] **Robert H. Guttman, Alexandros G. Moukas, and Pattie Maes,** « Agent-mediated Electronic Commerce: A Survey », Software Agents Group, MIT Media Laboratory, pp. 1-10, 1998.

[Hen97] **Schulzrinne Henning,** « RTP : Overview », en ligne sur http://www.cs.columbia.edu/~hgs/rtp/, 1997.

[HD99] **S. Hyacinth, T. N. Divine,** « A Perspective on Software Agent Research », The knowledge Engineering, Cambridge University Press, 99.

[HIM01] **R. Hino, K. Izuhara and T. Moriwaki,** « Message exchange method for decentralized scheduling », Assembly and Task Planning, Proceedings of the IEEE International Symposium, pp. 244-249, 2001.

[HL95] **W. Hursch and C. V. Lopes,** « Separation of concerns », Technical Report NUCCS-95-03, Northeastern University, Boston, Massachusetts, 1995.

[HVD87] **Durfee E-H, Lesser V., Corkill D**, « Coherent Cooperation among Communicating Problem Solvers », IEEE Transactions On Computers, pp. 1275-1291, 1987.

[HWC99] **Jiandong Huang, Yih Wang, and Feng Cao,** « On Developing Distributed Middleware Services for QoS and Criticality-Based Resource Negotiation and Adaptation », Journal of Time-Critical Computing Systems, Volume 16, pp. 187-221, 1999.

[Hya96] **S. N. Hyacinth,** « software agents : An overview », Intelligent Systems Research Advanced Application and Technology Department, knowledge Engineering Review, Vol. 11, pp 1-40, 1996

[Int95] **International Organization for Standarization (ISO**), « QoS Basic Framework », CD TEXT ISO/IEC JTC1/SC21 N9309, 1995.

[Int99] **International Organization for Standarization (ISO/IEC),** « Quality of Service, Guide to Methods and Mechanisms », Rapport Technique ISO/IEC 13243, 1999.

[Iss97] **V. Issarny**, « Architectures logicielles pour les systèmes distributes », Habilitation à diriger les recherché, Université de Rennes 1, octobre 1997.

[Jun93] **Jung**, « Gestion de la Qualité de Service - Application au R.N.I.S. large bande », Thèse dedoctorat, École Nationale Supérieure des Télécommunications de Paris, France, 1993.

[KC03] **J. Keeney and V. Cahil**, « Chisel : a policy-driven, context-aware, dynamic adaptation framework », In : 4th International Workshop on Policies for Distributed Systems and Networks, pp. 3-14, 2003.

[KDS04] **B. Kalle, G. Daniel and N. T. Simin,** « Scale-up and performance studies of three agent platforms », IEEE International Conference On Performance, Computing and Communications, On pp.857-863, 2004.

[Kel03] **Terrance Kelly :** « Utility directed allocation », In : First Workshop on Algorithms and Architectures for Self-Managing Systems, San Diego, 2003.

[KGM06] **J. Kresimir, J. Gordon, K. Mario,** « A performance Analysis of Multi-agent Systems », published in the journal of International Transaction on Systems Sciences and Applications, Vol. 1, n. 4, 2006

[KLM+97] **G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes et al,** « Aspect-Oriented Programming » In : Proceedings of ECOOP, volume 1241 of LNCS, Springer-Verlag, June, 1997.

[KNS04] **M. Berna-Koes, I. Nourbakhsh, K. Sycara**, « Communication efficiency in multi-Agent Systems », Proceedings of the International Conference On Robotics and Automation, IEEE, Volume 3, pp. 2129-2134, 2004.

[KYO+99] **M. Kosuga, T. Yamazaki, N. Ogino, J. Matsuda**, « Adaptive QoS Management Using Layered Multi-agents System for Distributed Multimedia Application », Proceedings of the International Conference On Parallel Processing, pp.388-394, 1999.

[LES04] **R. Leszczyna,** « Evaluation of agent platforms », In : Performance, Computing, and Communications, IEEE International Conference, pp. 857-864, April 2004.

[LJJ06] **M. Luis, M. S. Jose, M. A. Juan,** « Performance Evaluation of Open-Source Multi-agent Platforms», AAMAS'06, ACM, pp. 1107-1109, 2006.

[LLR+99] **Chen LEE, John Lehoczky, Ragunathan Rajkumar et Dan Siewiorek**, « On quality of Service Optimization with Discrete QoS Options », In : RTAS'99 : Proceedings of the Fifth IEEE Real-Time technology and Applications Symposium, Washington, DC, USA. IEEE Computer Society, pages 276-286, 1999.

[LN00] **B. Li, K. Nahrstedt, QualProbles**, « Middleware QoS Profiling Services for Configuring Adaptive Applications », Middleware, LNCS 1795, pp. 256-286, 2000.

[LN99] **Baochun Li et Klara Nahrstedt**, « Dynamic Reconfiguration for Complex Multimedia Applications » In : IEEE International Conference on Multimedia Computing and Systems, Florence, volume 1, pp. 165-170, 1999.

[Lou10] **Christine Louberry**, « Adaptation au contexte pour la gestion de la qualité de service », Thèse de doctorat, Université de Pau et des Pays de l'Adour, 205 pages, 2010.

[LS01] **José Lino Contreras, Jean-Louis Sourrouille**, « A Framework for QoS Management », In : TOOLS'01 : Proceeding of the 39 International Conference and Exhibition on technology of Object-Oriented Languages and Systems (TOOL39), Washington, DC, USA, IEEE Computer Society, pp. 183-193, 2001,

[LS98] **Chen Lee, et Dan Siewiorek**, « An approach for Quality of Service Management », Rapport technique CMU-CS-98-165, Computer Science Department, Carnegie Mellon University, 30 pages, 1998.

[LY97] **K. Lakshmann and R.Yavatkar,** « Integrated CPU and Network I/O QoS Management in an End system », IFIP 1997.

[MPT98] **Herman de Meer, Antonio Puliafito et Orazio Tomarchio**, « Management of QoS with software Agents » Cybernetics and Systems : An International Journal, pp. 499-523, 1998.

[MR95] **Wooldridge, M. & Jennings N. R.**, « Intelligent Agents : Theory and Practice », Knowledge Engineering Review, pp. 115-152, 1995.

[MRM05] **R. Maayan, S. Reid and V. Manuela**, « Decentralized communication strategies for coordinated multi-agent policies », Multi-Robot Systems, From Swarms to Intelligent Automata. Springer, Volume III, pp. 93-105, 2005;

[NQ96] **Klara Nahrstedt and Lintian Qian**, « Tuning System for Distributed Multimedia Applications », Technical Report, N° UIUCDCS-R-96-1958, UILU-ENG-96-1721, University of Illinois, Urbana, IL, 1996.

[NSN+97] **B. Noble, M. Satyanarayanan, D. Narayanan et al.**, « Agile Application-Aware Adaptation for Mobility », 16th ACM Symposium in Operating System Principles, 1997.

[Nwa96] **Hyacyn S. Nwana**, « Software Agents : An Overview », Knowledge Engineering Review, Volume 11, pp. 1-40, 1996

[OMG03] **OMG**, « UML Profile for Schedulability Performance and Time Specification », OMG documents.pts/02-03-2003, Mars 2003.

[OMG07] **OMG,** « Object Management Group, UML 2.1.1, Superstructure » document format/07-02-03, http://www.omg.org/docs/format/07-02-03, 2007

[PMC98] **Owezarski P., Diaz M., Chassot C**, « A Time-Efficient Architecture for Multimedia Applications' », IEEE Journal on Selected Areas in Communications, Volume 16, pp. 383-396, 1998.

[RFS02] **Douence R., Fradet P., Sudholt M.**, « A Framework for the detection and Resolution of Aspect Interactions » Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, volume 2487 of Lecture Notes in Computer Science, Springer-Verlag, pp. 173-188, 2002.

[RL98] **P-G. Raverdy, R. Lea : DART**, « A Distributed Adaptive Run-time. Work in progress session », Middleware'98: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, The Lake District, U.K., September 1998.

[RLL+97] **Ragunathan Rajkumar, Chen Lee, John P. Lehoczky et Daniel P. Siewiorek**, « A Resource Allocation Model for QoS Management » In : RTSS'97 : Proceedings of the 18th IEEE Real-Time Systems Symposium, IEEE Computer Society, pp. 298-307, 1997.

[RLS+97] **Braden R., Zhang L., Berson S., Herzog S. et Jamin S.**, « Resource Reservation Protocol (RSVP) », Version 1, functional specification, Request for comments RFC 2205, 1997.

[SA09] **M. T. Segarra, F. André,** « A Distributed Dynamic Adaptation Model for Component-Based Applications », Proceedings of the International Conference on Advanced Information Networking and Applications, IEEE, pp. 525-529, 2009.

[SAW94] **B. Schilit, N. Adam, R. Want**, « Context-Aware Computing Applications », Proceedings of the Workshop on Mobile Computing Systems and Applications, IEEE Computer Society, pp. 85-90, 1994.

[SC00] **Frank Siqueira et Vinny Cahill,** « Quartz : A QoS Architecture for Open Systems », In : ICDCS'00 : Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS 2000), IEEE Computer Society pp. 197-204, 2000.

[SC02] **Jean-Louis Sourrouille, José Lino Contreras,** « Objects autonomes adaptable », In : Systems à composants adaptables et extensibles, Grenoble. 2002.

[Sch94] **D. Schmit,** « the Adaptive Communication Environment Software : an Object-Oriented Network Programming Toolkit for Developing Communication Software » 11th and 12th Sun Users Group Conference, pp. 1-25, 1994.

[Siq98] **F. Siqueira,** « The Design of a Generic QoS Architecture for Open Systems », Proceeding of the 19th IEEE Real Time Systems Symposium (RTSS'98), pp. 1-23 ,1998.

[SLR+03] **R. E. Schantz, J.P. Loyall, C. Rodrigues, D. C. Shimidt, Y. Krishnamurthy, and I. Pyarali,** « Flexible and Adaptive QoS Control For Distributed Real-time and Embedded Middleware », Proceedings of the ACM/IFIP/USENIX International Conference on Middleware, Springer-Verlag, pp. 374-393, 2003.

[Vie05] **Patrice Vienne**, « Systèmes autonomes pour la gestion de la qualité de service », Chapitre 3, Thèse de doctorat, Institut National des Sciences Appliquées (INSA) de Lyon, 159 pages, 2005.

[Vil02] **Olivier Villin,** « Gestion de la Qualité de Service de bout en bout dans les systèmes reparties : approche gestion de resources », Thèse de doctorat de l'Université d'Evry Val d'Essonne, page 17, Avril 2002.

[VKB+95] **A. Vogel, B. Kérhervé, G. von Bochmann, J. Gecsei,** « Distributed Multimedia and QoS : A survey », IEEE Multimedia, Volume 2, pp. 10-19, 1995.

[VD05] **S. Vadhiyar et J. Dongarra,** « Self Adaptation in Grid Computing Concurrency and Computation : Practice and Experiences », Special issue, Grid performance, pp. 235-257, 2005.

[VS03] **Patrice Vienne, Jean-Louis Sourrouille,** « DCBL A framework for Dynamic Control of Behavior base on learning », In : International Workshop on Intelligent Technologies for Software Engineering (WITSE03), ACM ESEC/FSE, pp. 44-47, 2003.

[VS04] **Patrice Vienne, Jean-Louis Sourrouille,** « A Framework for Decision-making based on learning in a QoS Management System », In : Advances in Intelligent Systems- Theory and Applications, Luxembourg, 2004.

[VS05] **Patrice Vienne, Jean-Louis Sourrouille,** « A middleware for Autonomic QoS Management based on learning », In : Software Engineering and Middleware : 4th International Workshop, Springer-Verlag, volume 3437 de lecture Notes in Computer Science, pp. 170-184, 2005.

[VSM05] **Patrice Vienne, Jean-Louis Sourrouille, Mathieu Maranzana,** « Modeling Distributed Applications for QoS Management », In : Software Engineering and Middleware : 4th International Workshop, Springer-Verlag, volume 3437 de lecture Notes in Computer Science, pp. 170-184, 2005.

[XXH00] **Chen Xiaomei, Lu Xichen, and Wang Huaimin,** « The design of QoS Management Framework Based on Corba A/V Stream Architecture », IEEE Computer Society, the Fourth International Conference/Exhibition Proceedings on High-Performance Computing in the Asia-Pacific Region, Volume 1, pp. 542-547, 2000.

[Yoa93] **Shoham Y.**, « Agent Oriented Programming » Artificial Intelligence, 1993.

FOLIO ADMINISTRATIF

THESE SOUTENUE DEVANT L'INSTITUT NATIONAL DES SCIENCES APPLIQUEES DE LYON

NOM : BATOUMA DATE de SOUTENANCE : 30 septembre 2011
(avec précision du nom de jeune fille, le cas échéant)

Prénoms : NARKOY

TITRE : Un schéma d'emprunt de ressources pour l'adaptation du comportement d'applications distribuées

NATURE : Doctorat Numéro d'ordre : 05 ISAL

Ecole doctorale : Info-Maths

Spécialité : Informatique

Cote B.I.U. - Lyon : T 50/210/19 / et bis CLASSE :

RESUME :

L'objectif de cette thèse est de contrôler l'utilisation des ressources par les applications qui s'exécutent dans un environnement où la loi d'arrivée des applications est inconnue. Nous visons à maximiser cette utilisation pour optimiser la QoS.
Les systèmes d'exploitation utilisent très généralement une politique de « meilleur effort» (*Best-effort*) pour exécuter les applications. Tant que les ressources sont suffisantes, les applications s'exécutent normalement mais quand les ressources deviennent insuffisantes, des mécanismes de contrôle (*graceful degradation*) sont nécessaires pour continuer à fournir des services de qualité acceptable. Les architectures de la Qualité de Service (QoS) ont pour but de prendre en compte les propriétés non-fonctionnelles des applications (disponibilité des ressources, sécurité etc.) et de contrôler l'exécution des applications dans leur environnement.
Plusieurs architectures centralisées pour la gestion de la QoS d'applications distribuées ont été expérimentées dans notre équipe. Cette thèse a pour objectif de proposer une architecture décentralisée. Une première étude des architectures de gestion de la QoS a montré que les approches décentralisées ont un coût non négligeable en termes de messages échangés, même en se limitant dans chaque nœud à une vue partielle, des ressources disponibles dans le système.
La première partie de cette thèse propose un *middleware* totalement décentralisé pour contrôler l'utilisation des ressources des applications distribuées. Cette approche se fonde sur une planification approximative et un schéma d'emprunt de ressources afin d'améliorer la QoS globale du système. Via ce schéma d'emprunt, chaque nœud construit localement une vue partielle de la disponibilité des ressources dans le système. La connaissance locale de la disponibilité des ressources permet à chaque nœud de prendre des décisions et de planifier l'exécution des applications. Ainsi, des messages ne sont échangés que lorsqu'une information manque localement.
Pour un contrôle plus fin de l'exécution, la deuxième partie ajoute un support pour l'adaptation du comportement des applications. Le *middleware* utilise un modèle général des applications sous forme de graphe d'exécution décoré avec les besoins en ressources. Chaque application doit être conçue selon ce modèle pour être gérable par le *middleware* (approche intrusive). Dans ce graphe, les nœuds sont des points de décision et les arcs des actions à exécuter avec les besoins en ressources correspondants et une utilité quantifiant la perception du service rendu par cette action. Un chemin dans le graphe est une exécution possible de l'application avec une certaine utilité, et ce sont ces chemins qui fournissent les degrés de liberté dont le *middleware* a besoin pour adapter la consommation des ressources au contexte.
Les applications coopèrent avec le *middleware* dans le processus de gestion de la QoS lors de l'admission puis durant toute l'exécution. Le middleware est le chef d'orchestre et c'est lui qui pilote l'exécution des actions des applications (arcs dans le graphe d'exécution). Pour valider notre approche, un prototype à base d'agents a été réalisé à l'aide de JADE. Ce prototype simule un environnement avec plusieurs nœuds et plusieurs applications

et compare différents modes d'exécution, en particulier le *best effort* et notre solution, avec ou sans adaptation. Les résultats démontrent l'intérêt de notre approche.

MOTS-CLES : Qualité de Service, Gestion de la QoS, Gestion des Ressources, Schéma d'emprunt, planning, Architectures, Approche Décentralisée, Agent, Adaptation, Coordination de l'adaptation, Modèle d'Application, Système multi-agent, Applications distribuées.

Laboratoire (s) de recherche : DISP-LIESP

Directeur de thèse: Pr. Jean-Louis SOURROUILLE

Président de jury :

Composition du jury : Christian PERCEVOIS, Congeduc PHAM, Marc DALMAU, Denis TRYSTRAM, Jean-Louis SOURROUILLE.