Programmation répartie, optimisation par agent mobile

Salah El Falou

▶ **To cite this version:**

Salah El Falou. Programmation répartie, optimisation par agent mobile. Autre [cs.OH]. Université de Caen, 2006. Français. NNT : . tel-00123168

HAL Id: tel-00123168
https://theses.hal.science/tel-00123168

Submitted on 8 Jan 2007

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

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

UNIVERSITÉ de CAEN/BASSE-NORMANDIE
U.F.R. SCIENCES
ÉCOLE DOCTORALE SIMEM

THÈSE

présentée par

Salah EL FALOU

et soutenue

le 29 Novembre 2006

en vue de l'obtention du

DOCTORAT de l'UNIVERSITÉ de CAEN

spécialité informatique

(Arrêté du 07 août 2006)

Programmation répartie, optimisation par agent mobile

MEMBRES du JURY

Anne Boyer	Habilitée à diriger des recherches	Université Nancy 2	(rapportrice)
Hervé Guyennet	Professeur	Université de Franche-Comté	(rapporteur)
Franck Morvan	Maître de conférences	Université Paul Sabatier, Toulouse 3	(examinateur)
Abdel-Illah Mouaddib	Professeur	Université de Caen	(examinateur)
François Bourdon	Professeur	Université de Caen	(directeur)

Remerciements

Tout d'abord, j'adresse mes plus vifs remerciements à mon directeur de thèse François Bourdon, professeur à l'université de Caen, qui a encadré ce travail de recherche. Je lui suis reconnaissant pour m'avoir accordé sa confiance, et m'avoir laissé une liberté dans mes recherches tout en sachant les recentrer quand il le fallait. J'ai apprécié ses qualités humaines, son dynamisme et ses précieux conseils.

J'adresse mes sincères remerciements à Madame Anne Boyer, maître de conférences à l'université Nancy 2, ainsi qu'à Monsieur Hervé Guyennet, professeur à l'université de Franche-Comté, pour l'intérêt qu'ils ont porté à mon travail en acceptant d'être rapporteurs de cette thèse.

Je tiens à remercier Monsieur Abdel-Illah Mouaddib, professeur à l'université de Caen et Monsieur Franck Morvan, maître de conférences à l'université Paul Sabatier Toulouse, pour m'avoir fait l'honneur de participer à mon jury. J'ai particulièrement apprécié les discussions enrichissantes avec A.I. Mouaddib tant du point de vue scientifique que sur le plan humain.

Je remercie également tous les membres de l'équipe MAD pour leur gentillesse et leur bonne humeur. Les réunions de mardi auxquelles je participais étaient très bénéfiques pour l'orientation de mon travail.

Mes remerciements s'adressent aux chercheurs et aux membres du personnel du laboratoire GREYC pour leur accueil et leur aide.

Je remercie également le CNRS libanais qui m'a permis de mener ma thèse dans des conditions financières favorables.

Je remercie vivement Bassam et Maya pour le temps qu'ils ont accordé pour lire et relire ma thèse. Leurs nombreuses remarques ont mené ce document à sa version finale.

Ce long travail fastidieux ponctué par des hauts et des bas a été illuminé par la présence de mes amis. Les repas au RU et les pauses cafés que je partageais avec Bassam, Dang, Hicham, Hossam, Jin, Nadia m'ont permis d'échapper au stress induit par la thèse.

Un grand merci à mes parents, mes sœurs et frères qui ont su m'entourer pendant mon cursus doctoral. Leur encouragement, leur soutien et leur amour ne m'ont jamais fait défaut. Qu'ils trouvent ici l'expression de ma plus profonde gratitude.

Une tendre pensée à mes nièces et neveux qui ont embelli mon existence par leur innocence.

Table des matières

Introduction....1

Chapitre 1

Les Agents mobiles....5

1 Technologie d'agents mobiles....5

2 Communication sur le réseau Web....5

2.1 Client/serveur....6

2.2 Événement asynchrone....6

2.3 Mémoire virtuelle partagée (MVP)....7

2.4 Communication par agent mobile....7

3 Concepts de base des agents mobiles....7

3.1 Introduction....7

3.1.1 Évaluation distante....7

3.1.2 Code à la demande....8

3.1.3 Agents mobiles....8

3.2 Définition d'un agent mobile....9

3.3 Environnement d'exécution d'agents mobiles....9

3.4 Services requis pour l'exécution d'agents mobiles....10

3.4.1 Structure d'un agent mobile....10

3.4.2 Création d'agents mobiles....10

3.4.3 Migration d'un agent....11

3.4.4 Service de nommage....12

3.4.5 Service de localisation....12

3.4.6 Communications entre les agents....13

3.4.7 Exécution d'un agent....13

3.4.8 Sécurité....14

3.4.9 Tolérance aux pannes....15

3.4.10 Traçabilité....16

3.4.11 Cycle de vie et contrôle de l'agent....16

4 Plate-forme et Standardisation....17

5 Conclusion.....19

Chapitre 2

Agents mobiles et applications réparties : un tour d'horizon.....21

1 Introduction.....21

2 Le commerce électronique.....22

2.1 Le M-Commerce.....22

2.2 Apports des agents mobiles.....23

3 La recherche d'information sur le Web.....23

4 L'administration du réseau.....24

5 Le calcul distribué et les agents mobiles.....25

6 Les applications distribuées adaptables (cache réseau).....26

7 Les agents mobiles et l'informatique nomade.....26

8 Adaptation d'un flux vidéo en fonction de la machine utilisée.....27

9 Le code mobile et les cartes à puce.....28

10 La gestion de réseaux actifs.....29

11 Conclusion.....29

Chapitre 3

Choix d'une plate-forme d'expérimentation.....31

1 Introduction.....31

2 Moorea.....32

2.1 Architecture de la plate-forme.....32

2.2 Les spécifications techniques.....33

2.2.1 La communication.....33

2.2.2 Agent réactif et référence d'agent.....33

2.2.3 L'exécution synchrone et les instants.....33

2.2.4 Exemple d'un programme Rhum.....34

2.3 Programmation sur Moorea.....34

2.3.1 Programmation d'un agent avec l'API Moorea.....34

2.3.2 Démarche à suivre lors du développement d'une application Moorea...35

3 Grasshopper.....36

3.1 Architecture de la plate-forme....36
3.2 Les spécifications techniques....37
3.2.1 La communication....37
3.2.2 Persistance....37
3.3 Programmation dans Grasshopper....38
3.3.1 Migration et structuration d'un agent....38
4 JavAct....39
4.1 Architecture de la plate-forme....39
4.2 Les spécifications techniques....40
4.2.1 La communication....40
4.3 Programmation avec JavAct....40
4.3.1 Migration et structuration d'un agent....41
5 Comparaison et choix de la plate-forme....41
5.1 Communication entre deux agents....41
5.2 Migration d'un agent mobile....43
6 JavAct sur le Web....44
7 Conclusion....46
Chapitre 4
Agent mobile et calcul réparti....47
1 Introduction....47
2 Répartition de charge....48
2.1 Placement statique versus dynamique....48
2.1.1 Placement statique....49
2.1.2 Placement dynamique....49
3 Collecte d'informations....50
4 Prise de décision....51
5 Pourquoi utiliser les agents mobiles ?....51
6 Agentification et distribution d'un algorithme....52
6.1 Limitation de la distribution....52
6.2 La communication entre les agents : push vs pull....53

7 Équilibrage de charge itératif par agent mobile....54
7.1 Architecture....55
7.2 Agent moniteur et prise de décision....56
7.3 Agent mobile collecteur....57
7.4 Réseau hétérogène et normalisation....58
8 Conclusion....59
Chapitre 5
Une première expérience : application de la mobilité à l'équilibrage de charge....61
1 Introduction....61
2 La colonne à distiller....62
3 Modèle dynamique de la colonne à distiller....64
4 Résolution par l'algorithme d'Euler....65
5 Agentification de la colonne à distiller....66
6 Étude du comportement de la colonne....67
6.1 La plate-forme oRis....67
6.2 Résultats de la simulation....70
7 Distribution de la colonne à distiller....70
7.1 Placement statique....71
7.1.1 Réduction de la communication....71
7.1.2 Charge équitable....72
7.1.3 Résultats....72
7.2 Gains liés à la distribution statique....73
8 Placement dynamique et équilibrage de charge....73
8.1 Résultats....74
9 Conclusion....76
Chapitre 6
Déplacement intelligent des agents mobiles....77
1 Introduction....77
2 Communication entre deux machines....78

2.1 Approche client/serveur....78
2.2 Approche agent mobile....79
2.3 Comparaison entre les deux modèles....79
2.4 Gestion d'un parc informatique (application et résultats)....80
3 Communication avec n machines....82
3.1 Client/serveur....83
3.2 Agent mobile....83
3.3 Comparaison entre les deux approches et résultats des tests....84
4 La communication hybride....85
5 Recherche de la politique optimale : cas des interactions certaines....87
5.1 Construction du graphe et calcul des coûts....87
5.2 Algorithme....88
6 Recherche de la politique optimale : cas des interactions incertaines....89
6.1 Apprentissage sur les interactions : cas de la recherche d'informations sur le Web....90
6.2 Méthode statistique pour le choix de la politique de l'agent....91
6.3 Discussion et limites de la méthode....92
7 Processus décisionnel de Markov....93
7.1 Définition d'un MDP....93
7.2 Politique de l'agent....93
7.3 Politique optimale....94
7.4 L'algorithme Policy Iteration....96
7.4.1 Complexité de l'algorithme....97
7.5 L'algorithme Value Iteration....97
7.5.1 Complexité de l'algorithme....98
7.6 Problématique de la mise en oeuvre....98
8 Modélisation du déplacement de l'agent mobile sous forme d'un MDP....98
8.1 Choix des actions....98
8.2 Fonction de récompense....99
8.3 Construction du graphe d'états....99
8.4 Résolution du MDP....100

9 Application.....100
9.1 Présentation de l'application.....100
9.2 Résultats.....102
10 Migration intelligente du code de l'agent.....103
11 Décharge intermédiaire de l'agent.....105
11.1 MDP et décharge de l'agent mobile.....107
11.2 Résultats.....107
12 Conclusion.....109
Conclusion et Perspectives.....111
Conclusion et Apports.....111
Perspectives.....113
Recherche d'information.....113
Équilibrage de charge.....114

Table des figures

Figure 1.1 : Évaluation distante....8
Figure 1.2 : Code à la demande....8
Figure 1.3 : Agent mobile....8
Figure 1.4 : Migration d'un agent mobile....11
Figure 1.5 : Infrastructure du MAF et ses interfaces CORBA associées....18
Figure 2.1 : Adaptation de flux multimédia par agents mobiles....28
Figure 3.1 : Architecture Moorea....32
Figure 3.2 : Instant logique dans Moorea....33
Figure 3.3 : Exemple d'un programme Rhum....34
Figure 3.4 : Structure hiérarchique des composantes dans Grasshopper....37
Figure 3.5 : Structure de la méthode live()....39
Figure 3.6 : Architecture des agents mobiles adaptables....40
Figure 3.7 : Communication entre deux machines....42
Figure 3.8 : Influence de la taille du message sur le temps de communication....43
Figure 3.9 : Migration d'un agent mobile....43
Figure 3.10 : Influence de la taille de l'agent sur le temps de le migration....44
Figure 3.11 : JavAct et agence applet....45
Figure 3.12 : Migration d'un agent entre deux agences applet....46
Figure 4.1 : Modèle de communication....54
Figure 4.2 : Architecture du système....55
Figure 5.1 : La colonne à distiller....63
Figure 5.2 : Bouilleur....63
Figure 5.3 : Condenseur....64
Figure 5.4 : Échange entre plateaux....66
Figure 5.5 : Évolution de la colonne....68
Figure 5.6 : Évolution de la concentration des plateaux....68
Figure 5.7 : Influence du paramètre Zf sur la colonne....69
Figure 5.8 : Évolution de la concentration des plateaux....70
Figure 5.9 : Placement statique et réduction de la communication distante....72

Figure 5.10 : Placement statique et équilibrage de charge.....72

Figure 5.11 : Effet du nombre de machines sur le temps de la stabilité.....73

Figure 5.12 : Déplacement des agents.....74

Figure 5.13 : Gain obtenue par un placement dynamique.....75

Figure 5.14 : Évolution des seuils.....75

Figure 6.1 : Communication entre deux machines.....78

Figure 6.2 : Comparaison entre les deux approches « client serveur » et « agent mobile » dans le cas de la communication entre deux machines.....82

Figure 6.3 : Communication entre n machines.....83

Figure 6.4 : Comparaison entre les deux approches client serveur et agent mobile dans le cas de 5 machines.....84

Figure 6.5 : Seuil N avec modification du nombre de machines à visiter.....85

Figure 6.6 : Communication hybride.....86

Figure 6.7 : Recherche de plus court chemin.....88

Figure 6.8 : Recherche d'information sur le Web.....90

Figure 6.9 : Action skip.....99

Figure 6.10 : Graphe d'états du système.....100

Figure 6.11 : Recherche de restaurants dans une ville.....101

Figure 6.12 : Communication hybride.....103

Figure 6.13 : Migration avec un code de taille zéro.....105

Figure 6.14 : Influence de la quantité d'informations transportées sur la performance de la communication par agent mobile.....106

Figure 6.15 : Trois types d'actions.....107

Figure 6.16 : MDP avec envoie des données au client en cours de déplacement.....108

Introduction

L'évolution des réseaux à grande échelle a permis la naissance d'un grand nombre de nouvelles applications qui se développent autour de ce type de réseau [Arcangeli02] : commerce électronique, recherche d'information sur le web, plate-forme pour calcul réparti... Ces applications sont formées par des entités réparties et la bonne fonctionnalité nécessite la communication et les échanges entre ces entités. Le modèle client/serveur où les échanges se font par des appels distants à travers le réseau est le modèle le plus utilisé. Dans ce modèle, seul le client représente une application au sens propre du terme et le rôle du serveur est de répondre aux demandes des clients. Le serveur construit ses réponses indépendamment du client, ainsi une partie des données envoyées est inutile augmentant ainsi le trafic sur le réseau. De plus ce modèle exige une connexion permanente entre le client et le serveur, ce qui n'est pas le cas des terminaux mobiles qui sont exposés à la perte de la connexion. Le concept d'agent mobile apparaît dans ce contecte comme une solution facilitant la mise en œuvre d'applications dynamiquement adaptables, et il offre un cadre générique pour le développement des applications réparties sur des réseaux de grande taille. L'envoi du code sur le serveur permet d'adapter les services distants aux exigences du client et de ne lui envoyer que les informations utiles.

Un agent mobile [Arcangeli02] est une entité logicielle qui se déplace d'un site à un autre en cours d'exécution pour accéder à des données ou à des ressources distantes. Il se déplace avec son code son état d'exécution et ses données propres. La décision de migration peut se faire à l'initiative de l'agent lui même, de manière autonome ; la mobilité est ainsi contrôlée par l'application et non par le système d'exécution. Le but du déplacement est généralement d'accéder localement à des données ou à des ressources initialement distantes, d'effectuer le traitement en local et de ne déplacer que les données utiles. Nous distinguons deux rôles essentiels pour un agent mobile : les échanges de données et l'agent mobile calculateur. Dans une application donnée, un agent peut jouer l'un des deux rôles ou les deux en même temps.

Le but de l'utilisation d'un agent de communication est la réduction du trafic, ainsi en envoyant l'agent là où les tâches se font, les messages échangés deviennent locaux et libèrent d'autant la charge du réseau. Un agent mobile, en se déplaçant sur le réseau, utilisera les ressources des machines visitées et de ce fait il profitera de leur puissance de calcul. Le client aura donc la possibilité de décharger sa machine en déléguant l'exécution des tâches aux autres machines du réseau. L'agent mobile est ainsi utilisé comme entité de calcul.

Dans ce travail, nous étudions l'utilisation de la technologie d'« agents mobiles » dans le domaine de la recherche d'information sur le web. Dans ce cadre, un agent

mobile est créé par un client qui lui indique la tâche à effectuer. Cet agent se déplace entre les différentes machines de l'application, sur chaque machine il effectue des échanges et filtre les informations collectées. Ce filtrage permet de réduire la quantité d'informations transportées avec l'agent et par conséquent le trafic sur le réseau. L'agent transporte avec lui les données demandées par son client et les informations qui lui seront utiles dans le reste de son itinéraire. Dans le cas de la recherche d'information sur le web, la taille de l'agent mobile ajoute une surcharge lors de la communication. Cette taille est souvent compensée par le fait que l'agent effectue des interactions locales. Nous montrons que l'utilité de l'utilisation d'un agent mobile pour les échanges de données est liée à la quantité d'informations échangées. Ainsi, l'approche « agents mobiles » est bien adaptée dans le cas de fortes interactions entre le client et les serveurs. Dans le cas d'une faible interaction, cette approche devient moins efficace, car elle est pénalisée par la taille de l'agent.

La réalisation d'une tâche dans une application répartie nécessite des interactions fortes avec certains serveurs et faibles avec d'autres. Notre contribution consiste à proposer un modèle de communication hybride qui va utiliser les deux modèles d'échange. Ainsi la communication est déléguée à un agent intelligent mobile qui aura la possibilité de communiquer soit par envoi de messages distants, soit par migration. Le point fort du modèle proposé est que l'agent va s'adapter à son environnement afin de réduire le trafic réseau. Le choix de la politique suivie par l'agent dépend des caractéristiques du réseau de communication, de la taille de l'agent et des interactions entre les différents sites du réseau. Pour le choix de la politique, nous avons proposé des algorithmes dans le cas d'un environnement certain ou incertain. Dans un environnement certain, le déplacement de l'agent est modélisé sous forme d'un graphe, le choix de la politique de l'agent revient à la recherche du plus court chemin dans ce graphe. Dans le cas d'un environnement incertain, nous avons proposé deux algorithmes pour le choix de la politique de l'agent : un premier alors basé sur une méthode statistique et un deuxième qui est basé sur les processus décisionnels de Markov (MDP). Ce dernier permet d'obtenir la politique optimale de l'agent. La mise en œuvre sur des exemples pratiques, des algorithmes proposés, montre la réduction du délais d'attente et du trafic réseau.

L'évolution rapide des capacités de traitement des stations de travail, et l'amélioration de la qualité du réseau reliant ces stations, ont motivé l'utilisation du réseau Internet comme support pour les calculs distribués. Une bonne utilisation de cette gigantesque plate-forme permet ainsi d'augmenter la puissance potentielle des ordinateurs. La recherche d'un maximum de puissance dans les systèmes distribués nécessitent une répartition et un équilibrage de charge efficaces. Ceci est d'autant plus vrai lorsque l'on se trouve dans un contexte hétérogène. Travailler avec un grand nombre de machines perd de l'intérêt si le système est mal équilibré, car une partie de la puissance totale disponible est inutilisée. La bonne répartition d'une application nécessite des connaissances préalables sur son évolution permettant une modélisation assez fine de l'application. Dans la plupart des cas, cette connaissance est très coûteuse voire impossible.

Nous adoptons une approche itérative où l'équilibrage de charge se fait de proche en proche, les machines les moins chargées « aidant » les machines trop chargées. Nous proposons une architecture pour l'équilibrage de charge dynamique basée sur la technologie des « agents mobiles ». Dans cette architecture, la prise de décision s'effectue en fonction de la charge locale de la machine ce qui permet de réduire les échanges d'information. L'algorithme de prise de décision utilise deux seuils

dynamiques et lorsque les seuils sont dépassés un appel d'offre est lancé afin de faire la migration des tâches. Dans le but d'avoir une vision globale sur le système, nous proposons un agent mobile collecteur qui va explorer le réseau afin de construire une vision sur la charge moyenne. Cette charge moyenne sera transportée aux agents de décision afin d'ajuster les seuils. La mise en œuvre des algorithmes proposés sur un exemple réel montre l'efficacité de l'architecture proposée et le rôle joué par l'agent collecteur.

Cette thèse est constituée de six chapitres répartis comme suit. Dans le premier chapitre, nous présentons les différents modèles utilisés pour la répartition d'une application sur le réseau Internet et nous nous focalisons sur une approche de communication basée sur la technologie d'agents mobiles. En effet, un agent mobile est exécuté dans un environnement d'exécution. Son bon fonctionnement nécessite l'existence d'un certain nombre de services fournis par l'environnement. Ainsi, dans ce chapitre nous détaillons les principaux services qu'un environnement d'exécution devrait fournir à ces agents. Nous évoquons également la création et de la migration d'un agent, de la communication et de l'échange des messages entre les agents mobiles mais aussi la sécurité de l'agent et de celle de son site d'accueil. Avec la multitude des plates-formes d'exécution, des efforts dans le domaine de la standardisation ont été déployés. Nous présentons le standard MASIF qui définit les fonctionnalités minimales pour permettre l'interopérabilité d'agents mobiles.

Dans le chapitre 2, nous illustrons l'intérêt d'utiliser les agents mobiles pour des applications réelles. Ces applications montrent l'efficacité de la bonne utilisation de cette technologie pour améliorer la fonctionnalité des applications réparties. Dans les applications que nous avons choisies, les agents mobiles assurent les fonctionnalités distinctes de communication ou de calcul. Nous allons approfondir ces deux aspects en proposant des algorithmes améliorant les performances des applications grâce aux agents mobiles.

Pour s'exécuter, un agent mobile nécessite une plate-forme dotée de services pour son déplacement, sa localisation et les communications avec les autres agents. Cette plate-forme va donc jouer un rôle très important dans la qualité des services offerts via les agents mobiles ; tout particulièrement au niveau de ses performances et de sa fiabilité. Pour cela, nous présentons dans le chapitre 3 les plates-formes étudiées au cours de ce travail. Nous comparons ces plates-formes sur la base de leurs possibilités en matière de migration et de communication entre les agents. Nous justifions ensuite notre choix de plate-forme utilisée pour effectuer nos tests et valider nos algorithmes. Nous avons ajouté une extension à cette plate-forme facilitant son utilisation. Dans ce cas, l'ajout d'une nouvelle machine au système se fait à partir du navigateur Web permettant son utilisation à plus grande échelle.

Dans le chapitre 4, nous étudions le problème de l'équilibrage de charge dans une application répartie. Nous présentons les algorithmes centralisés et les algorithmes décentralisés. Ensuite, nous présentons notre architecture pour l'équilibrage de charge que l'on qualifie de dynamique et décentralisée. Cette architecture est basée sur la technologie des « agents mobiles ». L'algorithme utilisé est un algorithme itératif où l'équilibrage se fait de proche en proche. L'architecture proposée est complètement décentralisée et la décision de migration s'effectue en fonction de la charge locale de la machine. Un agent collecteur mobile se déplace sur les différentes machines de l'application ; il construit une vision globale sur la charge moyenne du système. Cette vision globale est utilisée pour guider la prise de décision locale et pour définir les

seuils utilisés lors du processus de migration.

Dans le chapitre 5, nous proposons une mise en oeuvre concrète d'un algorithme d'équilibrage de charge dans une application réelle. L'application choisie représente une simulation de l'évolution d'une colonne à distiller. L'intérêt d'une telle simulation est de l'utiliser pour le contrôle d'une colonne réelle qui nécessite de très coûteux calculs. L'étude de cette application nous permet de souligner les différents problèmes pouvant intervenir lors de la modélisation et la mise en place d'une application réelle. Les différentes entités de l'application n'évoluent pas de la même manière : ainsi certaines machines sont plus chargées en moyenne que d'autres. L'application que nous avons choisie possède les bonnes caractéristiques pour tester notre algorithme d'équilibrage de charge dynamique. Nous allons ainsi montrer la facilité de mise en place de l'architecture proposée pour une application réelle.

Dans le chapitre 6, nous étudions les deux modèles de communication : « client/serveur » et « agents mobiles ». Une comparaison entre les deux modèles permet de montrer que la communication par agents mobiles est plus avantageuse dans le cas d'une forte interaction entre les entités communicantes, alors que le modèle client/serveur est pour sa part mieux adapté pour des faibles interactions. Afin de profiter des deux modes de communication nous proposons une communication hybride ; un agent intelligent mobile est créé afin d'effectuer la tâche demandée par le client. Cet agent utilise les deux modes de communication afin de réduire le délai d'attente du client. Le choix de la politique que l'agent suivra dépend du débit sur les liens, de la taille de l'agent et des interactions entre les différents sites du réseau. Nous proposons un algorithme qui permet de trouver la politique optimale dans les cas d'un environnement déterministe, c'est-à-dire où les interactions entre les différentes entités sont connues par l'agent. Dans le cas d'un environnement incertain, nous modélisons le déplacement de l'agent mobile sous la forme d'une chaîne de Markov qui permet de trouver la politique optimale pour son déplacement. Dans cette politique, un agent peut entreprendre deux types d'actions : faire une communication distante ou se déplacer vers la source d'information. En appliquant nos résultats à la recherche d'information sur le web, nous montrons que ce modèle réduit les délais d'attente et le trafic réseau. Lorsque la réalisation d'une tâche demandée par le client nécessite la visite d'un grand nombre de machines, la taille des données transportées par l'agent dégrade les propriétés liées à la migration. Nous étudions l'influence de cette taille des données de l'agent dans une application réelle ; cette étude conduit à ajouter une nouvelle action pour l'agent, consistant à envoyer les données utiles au client en cours de déplacement.

Les différents résultats obtenus au cours de ce travail nous permettent de conclure sur les contributions que nous avons apportées au domaine de la programmation répartie. Nous mettons en valeur les apports d'une bonne utilisation des « agents mobiles » dans le cadre des applications distribuées. Ensuite, nous signalons d'éventuelles pistes de recherche pour poursuivre ce travail.

Chapitre 1

Les Agents mobiles

1 Technologie d'agents mobiles

La plupart des applications sur le réseau Internet nécessitent l'interaction entre différentes entités à travers le réseau, afin d'échanger des données et de répartir les tâches. Aujourd'hui, le modèle « client/serveur » où les échanges se font par des appels distants à travers le réseau est le modèle le plus utilisé. Ce modèle présente l'inconvénient d'augmenter le trafic sur le réseau et il exige une connexion permanente entre le client et le serveur ce qui n'est pas le cas des terminaux mobiles qui sont exposés à la perte de la connexion. Dans ce chapitre, nous présentons les différents modèles utilisés pour la répartition d'une application sur le réseau Internet. Nous exposons les différents modes de communication que l'on puisse rencontrer et nous nous focaliserons sur une approche de communication basée sur la technologie d'agents mobiles. Ces agents sont des entités qui se déplacent d'une machine à l'autre sur le réseau, sans perdre leur code ni leur état. Ainsi, en envoyant les agents là où les tâches se font, les messages échangés deviennent locaux et libèrent d'autant la charge du réseau.

Un agent mobile est un programme qui s'exécute dans un environnement. Les services nécessaires à l'agent mobile, qui sont fournis par l'environnement, doivent exister avec de bonnes propriétés. Ainsi, dans ce chapitre nous détaillons les principaux services qu'un environnement d'exécution doit fournir à ces agents. Nous abordons la création de l'agent, sa migration entre les différentes machines, sa localisation, sa protection, la protection de son site d'accueil et la communication entre les agents. Nous traitons le cycle de vie d'un agent mobile depuis sa création jusqu'à sa suspension et les différents points d'entrées qui vont aider le programmeur dans le contrôle de l'agent. Dans un souci de compatibilité entre les environnements d'exécution des agents mobiles, des efforts sont faits dans le domaine de la standardisation afin de faciliter l'interopérabilité, ainsi serons amenés à parler de la standardisation d'intergiciels agent.

2 Communication sur le réseau Web

Internet regroupe des centaines de milliers de machines connectées par le réseau IP et qui permet la communication entre ces différentes machines. Ainsi le réseau IP représente un environnement réparti dans lequel des machines peuvent fonctionner en parallèle. Une bonne utilisation de ce réseau permet d'augmenter potentiellement

«indéfiniment» la puissance des ordinateurs. Les éléments qui constituent cet environnement, échangent (entre les machines) de l'information par transmission de messages à travers le réseau. Le choix du placement des éléments sur le réseau, leur rôle et la manière dont ils communiquent influent particulièrement sur les propriétés d'une application distribuée. Nous présentons ici les modèles de communication les plus connus pour le développement d'applications réparties.

2.1 Client/serveur

Dans le modèle client/serveur, un serveur représente un objet géré sur un site et offrant des services aux clients. Les fonctionnalités d'une application sont alors encapsulées dans des services et proposées/exécutées au sein de serveurs. Les serveurs sont ainsi vus comme des fournisseurs de services. On trouve soit : (i) des serveurs de données, dans lesquels des clients accèdent à des données localisées sur chaque serveur (SGBD, LDAP, etc...), (ii) soit des serveurs de calculs, dans lesquels des clients utilisent les ressources de chaque serveur afin d'exécuter une tâche précise.

Dans le modèle client/serveur, seul le code initialement installé sur le serveur pourra être exécuté sur ce dernier, et le rôle du serveur est de répondre aux demandes des processus clients. C'est le client, demandeur d'un service, qui établit une interaction avec un serveur. Nous pouvons envisager que l'exécution d'une tâche sur un serveur nécessite l'interaction avec d'autres serveurs (consultation d'une base de données). Dans ce cas un même processus peut être à la fois client et serveur. Dans le modèle client/serveur seul le client représente une application au sens propre du terme. Le serveur a pour fonction de répondre aux demandes des clients. Les réponses dépendent des requêtes formulées et non pas des applications clientes qui l'interrogent.

Le client envoie une requête au serveur. Cette requête décrit l'opération à exécuter (service demandé) ainsi que ses paramètres. On dit alors que le client invoque une opération sur le serveur. Le serveur exécute le service demandé par le client et revoie la réponse. Il s'agit d'une invocation synchrone. Dans ce type de communication le client qui émet une requête vers le serveur se bloque, en attente de la réponse du serveur.

2.2 Événement asynchrone

Ce modèle permet la communication entre plusieurs processus et ceci d'une façon indirecte. Les différents composants de l'application coopèrent par l'envoi et la réception de notification. Les processus communiquent entre eux par l'intermédiaire d'une base représentant un gestionnaire d'abonnement et de distribution d'événements. Cette base se charge d'aiguiller un événement vers ses abonnés. Dans ce modèle, il y a deux types de processus : les émetteurs qui envoient ou publient des événements dans la base, et les consommateurs qui s'abonnent à certaines catégories d'événements. Quand la base reçoit une notification d'événement, elle se charge de la distribuer à tous les composants qui ont déclaré leur intérêt pour recevoir ce message. Ainsi, ce tiers fait un découplage entre les sources et les consommateurs d'événements. L'émetteur d'une communication n'est pas obligé de spécifier la destination de ses messages et le destinataire ne connaît pas nécessairement l'origine du message. Ce type d'infrastructure [Leclercq04] permettant de communiquer à l'aide de messages s'appelle souvent MOM (Message Oriented Middleware).

2.3 Mémoire virtuelle partagée (MVP)

Les échanges se font par l'intermédiaire d'une mémoire partagée de grande taille qui sert d'espace de communication entre des processus sur des ordinateurs qui ne partagent pas physiquement leur mémoire. Son intérêt est de permettre l'utilisation d'un modèle de programmation qui a des avantages par rapport aux modèles basés sur l'échange de message. Les processus accèdent à la mémoire partagée par des lectures et des mises à jour sur ce qui leur semble être de la mémoire ordinaire à l'intérieur de leur espace d'adressage. Le système, tournant en tâche de fond, assure de manière transparente que les processus s'exécutant sur différents ordinateurs observent les mises à jour effectuées par d'autres processus.

Le grand avantage de la MVP [Bennett90] est d'épargner au programmeur la gestion de l'échange de messages quand il écrit une application qui en aurait besoin sans la MVP. On ne peut toutefois pas éviter complètement l'échange de message dans un système distribué, il faut bien sûr que le système sous-jacent de la MVP envoie les mises à jours aux différents processeurs, chacun ayant une copie locale des données partagées [Guyennet97]. Ces données doivent être mises à jour de façon régulière pour une question de performance mais aussi de validité des données utilisées.

2.4 Communication par agent mobile

Ce modèle se base sur le concept d'agent mobile. Il s'agit d'une entité autonome qui se déplace d'une machine à l'autre sur le réseau, sans perdre ni son code ni son état. Ainsi, lorsqu'un agent a besoin de communiquer avec un autre site, il utilise le réseau pour se déplacer. Après sa migration, les messages échangés deviennent locaux et libèrent d'autant la charge du réseau.

3 Concepts de base des agents mobiles

3.1 Introduction

Avant de présenter la technologie d'agents mobiles, nous allons la situer par rapport à une grande famille : la technologie du code mobile. Un processus informatique est constitué par une séquence d'instructions (code) qui s'exécute sur une machine et qui utilise les ressources de cette machine. Dans le contexte de la mobilité, il faut envisager la possibilité d'interrompre l'exécution d'un processus afin de le poursuivre sur une autre machine. Le processus est représenté, en plus de son code, par son état d'exécution. Par état d'exécution nous entendons la valeur du compteur ordinal, celle de la pile d'exécution et les différents registres du processeur. Selon le schéma d'exécution de ce code et la localisation des différentes entités du système (ressource, code et état d'exécution) nous pouvons distinguer les notions d'évaluation distante, de code à la demande et d'agents mobiles.

3.1.1 Évaluation distante

Dans une interaction par évaluation distante (voir Figure 1.1), un client envoie un code à un site distant. Le site récepteur utilise ses ressources pour exécuter le programme envoyé. Éventuellement, une interaction additionnelle délivre ensuite les résultats au client. Dans ce schéma, seul le code est transmis au serveur et l'exécution

du code se déroule uniquement sur ce dernier. Les interactions avec les imprimantes PostScript sont réalisées par ce modèle. Le code d'une requête SQL émis vers un serveur de base de données représente un autre exemple d'évaluation distante.

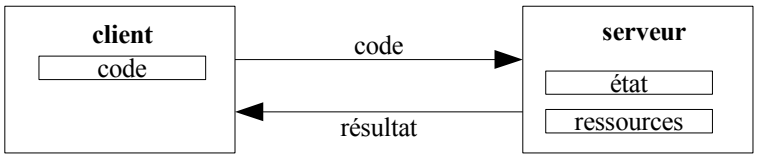

Figure 1.1 : Évaluation distante

3.1.2 Code à la demande

Dans ce schéma, le processus client interagit avec un site distant afin de récupérer un savoir faire qui sera exécuté sur la machine cliente. Ainsi, le client télécharge le code nécessaire à la réalisation d'un service. Le rôle du site distant est de fournir le code du service qui sera exécuté sur le site client (voir Figure 1.2). Les Applets Java reposent sur cette technologie de code mobile, il s'agit d'un programme chargé à partir d'une page Web pour être exécuté sur la machine du client.

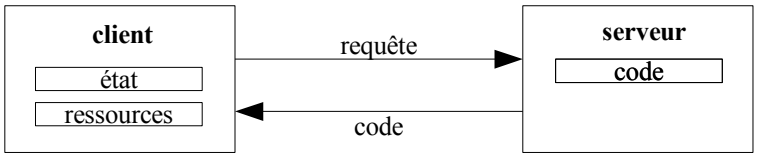

Figure 1.2 : Code à la demande

3.1.3 Agents mobiles

Par comparaison avec les deux schémas précédents, l'exécution du processus débute sur le site client. Dans la mesure où le client a besoin d'interagir avec le serveur, ce même processus (code, état d'exécution et données) se déplace à travers le réseau pour continuer son exécution et pour interagir localement avec les ressources du serveur (voir Figure 1.3). Après exécution, l'agent mobile retourne éventuellement vers son client afin de lui fournir les résultats de son exécution.

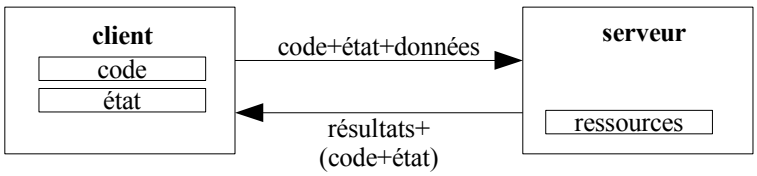

Figure 1.3 : Agent mobile

Dans ce schéma, le savoir-faire appartient au client, l'exécution du code est initiée côté client et continuée sur les différentes machines visitées.

3.2 Définition d'un agent mobile

Le terme agent est très répandu en informatique. La notion d'agent a donné lieu à de très nombreuses définitions, notamment dans le domaine de l'intelligence artificielle. Un agent [Beale94] est une entité logicielle (informatique ou électronique) plongée dans un environnement dans lequel elle est capable d'agir. Il possède un comportement autonome relié à ses observations, à sa connaissance et aux interactions qu'il entretient avec l'environnement et avec les autres agents. Nous définissons un agent mobile comme étant un agent répondant à cette définition et capable de se déplacer d'un site à un autre dans un réseau pour accomplir la tâche du client. De cette définition découlent les points essentiels suivants :

Comportement : Un agent est pourvu d'un programme qui lui dicte la tâche à accomplir. On dit qu'il agit par délégation pour le client. Un agent peut ainsi venir avec une compétence particulière sur une machine hôte.

Autonomie : Un agent agit indépendamment du client. Il décide lui-même là où il va se déplacer et ce qu'il doit y faire, en fonction du comportement qui lui a été donné. Cela implique que l'on ne peut pas toujours prévoir l'itinéraire des agents.

Mémoire : Un agent mobile dispose d'une capacité de mémorisation lui permettant de récolter des informations sur les sites visités. Les informations mémorisées seront livrées par l'agent après son retour au client.

Environnement : L'environnement dans laquelle l'agent évolue est constitué :

- par les machines qui vont accueillir l'agent lors de son exécution ; ainsi l'agent utilise les ressources (unité de calcul, mémoire, etc...) disponibles sur la machine afin d'accomplir la tâche demandée par le client,
- par les liens réseau que l'agent utilise pour se déplacer entre les différents sites,
- par les autres agents fixes et mobiles avec qui l'agent interagit afin de réaliser son but.

3.3 Environnement d'exécution d'agents mobiles

Un environnement d'exécution d'agents mobiles fournit une interface de programmation et un environnement d'exécution offrant des primitives pour créer, lancer et exécuter un agent mobile. Cet environnement est constitué d'un ensemble de programmes statiques, appelés places, s'exécutant sur les sites du système susceptibles d'accueillir des agents.

Les environnements d'exécution d'agents mobiles offrent plusieurs services de base permettant l'exécution d'un agent mobile sur un site. Ces services sont les suivants : création et migration d'un agent mobile, communication et échange de messages entre les agents mobiles, accès aux ressources locales par les agents mobiles et traçage des agents mobiles en cours d'exécution. Il faut ajouter à ces services, fournis par l'environnement d'exécution d'agents mobiles, des mécanismes mettant en oeuvre des notions de qualité de service requise par les applications comme par exemple la sécurité ou la tolérance aux pannes.

3.4 Services requis pour l'exécution d'agents mobiles

Dans cette section, nous allons décrire les fonctions nécessaires pour construire des applications réparties à agents mobiles. Après une présentation de la structure d'un agent mobile, nous allons présenter les services suivants :

- création d'un agent,
- transfert d'un agent,
- désignation d'un agent, c'est-à-dire offrir un nom globalement unique à un agent,
- localisation d'un agent,
- communication entre agents,
- exécution d'un agent,
- protection d'un agent.

Ces différents services sont détaillés dans les paragraphes suivants.

3.4.1 Structure d'un agent mobile

Un agent mobile est un programme pouvant se comporter d'une façon autonome au profit de son client. Chaque agent a son flot d'exécution pour pouvoir prendre l'initiative d'exécuter des tâches. Un agent doit avoir la capacité de migrer d'une machine à une autre sur le réseau. Pour parler de la migration d'un agent, il est important de connaître les éléments à transférer avec l'agent. Une instance d'agent mobile comporte trois parties [Tanenbaum92] :

- un code exécutable qui représente la suite d'instructions définissant le comportement statique de l'agent mobile,
- un contexte d'exécution qui reflète l'état d'exécution courant de l'agent mobile (valeurs des registres, pile d'exécution),
- les données ou les ressources utilisées par l'agent mobile ; ces ressources sont divisées en deux parties [Fuggetta98] :
  - les ressources transférables qui sont les valeurs des attributs de l'agent et qui lui donnent un état global,
  - les ressources non transférables qui constituent l'environnement d'exécution fourni par le système (par exemple les fichiers ouverts, les connexions sockets, les registres, etc.) et les matériels physiques utilisés par l'agent (imprimante, écran, etc.).

Dans le paragraphe §3.4.3, nous allons montrer comment l'agent migre avec son code, son contexte d'exécution et ses données.

3.4.2 Création d'agents mobiles

Dans la mesure où ces agents peuvent se déplacer, on peut se poser la question, lors de leur création, de l'endroit où l'on souhaite qu'ils démarrent leurs activités. Une fois

créé, l'agent peut être actif, c'est-à-dire que l'exécution de son code est déclenchée. La création/lancement d'un agent peut prendre plusieurs formes :

- création locale,
- création et exécution à la demande (*code on demand*),
- création et exécution à distance (*remote execution*).

La création de l'agent n'est pas directement liée à la mobilité, mais à des mécanismes sur lesquels s'appuie la mobilité. À sa création, un nom globalement unique (cf. §3.4.4) doit être attribué à l'agent ce qui va permettre de localiser l'agent et de communiquer avec lui.

3.4.3 Migration d'un agent

La migration permet le transfert d'un agent en cours d'exécution d'un site à un autre à travers le réseau.

Nous allons prendre le cas d'un agent qui désire migrer entre deux machines avec la possibilité de recevoir des messages pendant son déplacement (voir Figure 1.4). Cette migration comporte les étapes suivantes [Ismail99] :

1. la sérialisation du contexte de l'agent et la production d'un message incluant le contexte sérialisé de l'agent et son code ;
2. le processus de l'agent étant suspendu sur sa machine d'origine, toutes les communications avec d'autres agents sont donc suspendues et l'agent est détruit ;
3. l'envoi du contexte et du code de l'agent vers la machine de destination ;
4. l'état de l'agent migrant est restauré sur la machine de destination, un nouveau processus léger est créé pour la poursuite de son exécution ;
5. les communications en cours sont redirigées vers la machine de destination, l'agent n'étant pas encore activé, ces messages seront traités après sa réactivation ;
6. réactivation de l'agent sur la machine de destination, dès que la partie de son contexte nécessaire à son exécution est reçue ; la machine de destination s'occupe des liens dynamiques entre l'agent et son code. À partir de ce moment, la migration prend fin. Le processus sur la machine d'origine peut être définitivement effacé.

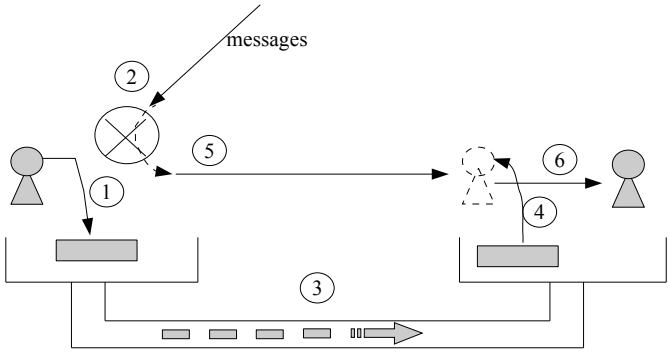

Figure 1.4 : Migration d'un agent mobile

Les différentes stratégies de migration d'agents dépendent de la manière dont sont traitées ces diverses étapes et les données transférées avec l'agent lors de sa migration. Deux types de migrations ont été proposées dans les plates-formes existantes :

1. La migration forte permet à un agent de se déplacer quelque soit l'état d'exécution dans lequel il se trouve. Dans ce type de migration l'agent se déplace avec son code, son contexte d'exécution et ses données. Dans ce cas, l'agent reprend son exécution après la migration exactement là où elle était avant son déplacement. La migration forte nécessite un mécanisme de capture instantanée de l'état d'exécution de l'agent. Elle peut être *proactive* ou *réactive*. Dans la migration proactive, la destination de l'agent est déterminée par l'agent lui-même. Dans la migration réactive, la migration de l'agent est dictée par une partie ayant une relation avec l'agent mobile.

2. La migration faible ne fait que transférer avec l'agent son code et ses données. Sur le site de destination, l'agent redémarre son exécution depuis le début en appelant la méthode qui représente le point d'entrée de l'exécution de l'agent, et le contexte d'exécution de l'agent est réinitialisé. Pour que l'agent se branche sur une instruction particulière de son code après sa migration, le programmeur doit inclure dans l'état de l'agent des moments privilégiés (explicites) dans le code de l'agent (point d'arrêt) pour pouvoir le relancer.

La migration forte, bien que beaucoup plus exigeante à implanter [Dillenseger02] que la migration faible [Grasshopper98], n'en est pas moins indispensable pour toutes les applications pour lesquelles les notions de fiabilité et de tolérance aux pannes sont primordiales.

3.4.4 Service de nommage

Dans un système à agents mobiles, les agents ont besoin de communiquer entre eux, ce qui nécessite de les nommer. Le nom donné à l'agent doit être globalement unique. Dans la plupart des systèmes à agents mobiles ce nom est construit à partir du nom de la machine (ou adresse IP) où l'agent s'exécute, d'un numéro de port et d'un identificateur localement unique.

3.4.5 Service de localisation

Afin de communiquer avec un agent mobile, il est nécessaire de le retrouver. La mobilité de l'agent introduit des contraintes sur les communications qui n'étaient pas présentes avec les systèmes classiques où les objets ne pouvaient pas changer de lieu d'exécution. En particulier, il faut que des entités mobiles puissent communiquer indépendamment de leur localisation et de leur mobilité [Alouf02]. Le point le plus important concerne la fiabilité, il faut pouvoir assurer les communications à tout moment avec les objets mobiles. Pour ces raisons, un système à agents mobiles doit offrir un service de localisation des agents à travers un serveur de noms. Ce serveur de noms contient la localisation courante de l'agent ou bien suffisamment d'informations pour le localiser.

Les principaux schémas de localisation sont les suivantes [Milojicicc98] :

- **mise à jour sur le site d'origine** : le serveur de noms situé sur la machine d'origine des agents est mis à jour à chaque migration d'un agent. Ceci entraîne

un nombre de communications avec le serveur d'origine égal au nombre de migrations de l'agent ($N_{com}=N_M$),

- **enregistrement** : les agents enregistrent leur mouvement auprès d'un serveur de noms défini et centralisé qui n'est pas situé sur la machine d'origine. Le nombre de communications est égal au nombre de mouvements de l'agent augmenté d'une communication de la machine d'origine vers le serveur de noms ($N_{com}=N_M+1$),
- **recherche** : le serveur de noms cherche la localisation d'un agent selon un itinéraire bien défini. Cela suppose que l'itinéraire d'un agent doit être connu à l'avance. Le nombre de communications peut varier de *1* jusqu'au nombre maximum de l'itinéraire ($1 \leq N_{com} \leq max\ (N_M)$),
- **poursuite** : suivre un lien de poursuite permet de localiser un agent. Le nombre de communications est au maximum égal au nombre de noeuds visités par l'agent ($1 \leq N_{com} \leq max\ (N_M)$).

3.4.6 Communications entre les agents

La communication entre deux entités peut se faire de deux façons ; la plus intuitive consiste à avoir un mécanisme permettant une communication directe entre les objets, ce qui correspond à la communication synchrone ou asynchrone entre les deux entités. Une deuxième manière de la faire est d'avoir des mécanismes de communication indirecte. Dans ce cas un objet voulant communiquer envoie son message à un objet tiers qui se charge de le faire suivre au destinataire.

Dans un système à agents mobiles, il faut différencier deux types de communication selon que la communication intervient entre deux agents ou entre un agent et un groupe d'agents. Les moyens de communication entre les agents sont multiples, nous pouvons citer la communication par message, session, tableau blanc, rendez-vous, par événement distribué ou local.

3.4.7 Exécution d'un agent

Un système à agents mobiles doit offrir la possibilité d'exécuter un agent sur les machines qui vont accueillir ce dernier, ainsi l'agent utilise les ressources disponibles sur la machine afin d'accomplir la tâche demandée par le client. Sur la machine d'accueil, les agents mobiles peuvent être amenés à effectuer des entrées/sorties, accéder à l'interface utilisateur, au système de gestion de fichiers, aux applications externes et à l'interface réseau. Les machines qui forment notre système d'exécution n'auront pas toutes le même type de système d'exploitation. Par conséquent, maintenir l'indépendance vis-à-vis du système d'exploitation d'un environnement d'agents mobiles tout en autorisant en même temps l'accès aux ressources est un problème important à surmonter dans les environnements d'exécution d'agents mobiles. C'est pour cette raison que la plupart des environnements d'agents mobiles utilisent le langage Java comme langage d'implémentation du code mobile [Gomoluch01].

Java est un langage orienté objet développé par Sun et présenté officiellement en 1995. Les applications écrites en Java [Java] sont compilées en *bytecode* et exécutées sur une machine virtuelle, la Java Virtuel Machine (JVM). Le succès du langage Java

se traduit, entre autre, par le fait qu'on peut raisonnablement considérer que la machine virtuelle Java constitue aujourd'hui un environnement d'exécution "universel", car disponible sur toutes les machines d'un système réparti. Cette propriété en fait une plate-forme de développement de choix pour les applications mobiles ou distribuées car le programmeur n'a pas à gérer différents langages ou différentes versions d'un programme suivant les systèmes sur lequel il va s'exécuter. En plus de cette propriété liée à la portabilité, Java propose toute une série de services qui permettent de construire des applications distribuées mobiles. On trouve notamment :

- la sérialisation/désérialisation d'un objet afin de le transmettre à travers le réseau,
- la communication entre deux objets situés sur des machines distantes (RMI, socket, etc...),
- le chargement dynamique de code à partir d'un site distant.

L'utilisation de Java permet de résoudre le problème lié à l'exécution de l'agent sur les différentes machines du système répartie. Mais avec la diversité des plates-formes disponibles, nous pouvons nous interroger sur la migration entre les différentes plates-formes. Chaque plate-forme possède sa propre façon pour définir le comportement des agents, ce qui pose un problème de compatibilité lors de la migration d'un agent entre les différentes plates-formes. Pour surmonter ce problème, des travaux comme dans [Groot04] proposent un langage générique, pour la description du comportement de l'agent, basé sur le langage XML. Une projection de cette description permet à une plate-forme d'agents mobiles de reconstruire l'agent avec son propre langage.

L'exécution d'un agent mobile sur une machine hôte pose le problème du piratage des sites visités. Ce problème, non spécifique aux agents mobiles, est symétrique pour ces derniers. En effet, un agent mobile est exposé au problème du piratage par les logiciels qui fonctionnent sur les sites qu'il visite. Dans la section suivante, nous allons détailler les problèmes liés à la sécurité dans un environnement d'agents mobiles.

3.4.8 Sécurité

Garantir la sécurité dans les environnements d'agents mobiles est important du fait de la nature même du modèle d'exécution des agents et du fait des interactions des agents mobiles avec plusieurs systèmes et ressources. Un système à agents mobiles doit offrir des mécanismes permettant une exécution sécurisée des agents au sein du système. Trois types de problèmes de sécurité ont été identifiés pour les environnements d'exécution d'agents mobiles :

- **La sécurité du site d'accueil contre un agent**. Au cours de son exécution, un agent mobile peut avoir accès aux ressources du site sur lequel il est situé. Un agent malveillant peut donc profiter des accès aux ressources locales pour propager des virus [Chess97], des worms ou des chevaux de Troie. Il peut masquer sa véritable identité et lancer une tâche lourde ce qui va entraîner un déni de service [Bellavista01]. La machine qui va accueillir l'agent mobile doit être amenée à détecter le mauvais déroulement de l'exécution de l'agent. Dans [Galtier01], les auteurs présentent une méthode qui permet à l'agent de définir ses besoins d'une façon indépendante de la machine sur lequel il s'exécute. Le site d'accueil peut détecter les abus commis par l'agent permettant ainsi de le

contrôler. Une approche classique pour protéger les sites des agents malveillants est de limiter les accès des agents aux ressources locales du site [Hantz06]. Dans cette approche, le comportement de l'agent est écrit sous la forme d'un langage interprété. L'agent est exécuté dans un environnement qui à son tour s'exécute au dessus du site. Le contrôle de l'agent est ainsi fait dans l'environnement d'exécution [Carvalho04]. Une seconde approche consiste à ne laisser s'exécuter que les agents authentifiés [Ametller04], une signature est attribuée à l'agent permettant au site d'accueil de l'authentifier avant son exécution.

- **La sécurité d'un agent contre le site d'accueil**. La protection des agents mobiles contre des sites hostiles est spécifique au domaine. Les agents sont à protéger à deux niveaux : la protection du code de l'agent (changement du comportement) et la modification de l'état de l'agent. Plusieurs approches ont été proposées : l'approche organisationnelle permet, seulement aux sites dignes de confiance, de gérer des systèmes d'agents mobiles ; l'approche de détection de manipulation offre des mécanismes fondés sur le traçage permettant de détecter les manipulations de données effectuées sur un agent [Diaz01] ; enfin l'approche de protection par boîte noire. La cryptographie mobile est une étape vers l'approche de protection par boîte noire. La spécification de l'agent est convertie en code exécutable et en un ensemble de données encryptées [Claessens03]. Le cryptage de données empêche un éventuel attaquant de lire ou de modifier les données. Dans [Benachenhou05], les auteurs présentent une nouvelle approche qui consiste à exécuter un agent clone sur un site sûr afin de surveiller l'exécution du vrai agent. Cette solution engendre une augmentation dans le trafic réseau (échange entre l'agent et son clone).
- **La sécurité d'un agent contre un autre agent**. L'agent doit être protégé contre une éventuelle attaque par un autre agent [Hohlfeld02] situé sur le site hôte ou sur un autre site. Un agent doit avoir sa propre politique de sécurité qui peut être soit assurée par l'agent lui même, soit par le site d'accueil.

3.4.9 Tolérance aux pannes

Le modèle d'exécution de l'agent mobile implique son interaction avec plusieurs sites ce qui expose l'agent à une éventualité de disparation à cause de la défaillance ou de la déconnexion soudaine et imprévue d'un site sur lequel il s'exécute. La disparition d'un agent entraîne un dysfonctionnement de l'application basée sur ce dernier. Une application distribuée sûre doit pouvoir continuer de fonctionner en cas de défaillance d'une partie du système. Pour certains types d'applications, il est essentiel que les environnements d'exécution d'agents mobiles offrent des mécanismes de tolérance aux fautes.

Dans une architecture de services, les défaillances de sites peuvent conduire à un comportement défaillant d'un service et donc le rendre inutilisable par le client. Plusieurs types de défaillances sont à considérer :

- Une défaillance par arrêt (crash, panne) quand un serveur ne rend pas de résultats suite à des invocations répétées.
- Une défaillance par omission quand un serveur omet de répondre à son client.
- Une défaillance temporelle quand la réponse du serveur est fonctionnellement correcte mais n'est pas arrivée dans un intervalle de temps donné.

- Une défaillance de valeur quand le serveur rend des résultats incorrects.

Pour une application distribuée basée sur le concept d'agents mobiles, la migration des agents engendre d'autres types de défaillance :

- Un agent peut disparaître après avoir visité plusieurs sites. Si aucune précaution n'est prise, les résultats de son exécution sur ces sites peuvent être perdus [Shao05].
- Il est important de détecter la disparition d'un agent pour en informer la place qui l'a lancé.
- Un problème d'atomicité pour l'exécution globale d'un agent qui se déroule successivement sur plusieurs sites. Ce qui implique qu'il faut garantir que l'agent reprend son exécution exactement là où elle était avant son dépalcement et ceci avec le même contexte d'éxecution. C'est au programmeur de garantir cette atomicité dans une migration faible (voir chapitre 3 : §3.3.1).

Pour faire face à la défaillance, les environnements d'agents mobiles offrent un mécanisme de point de reprise [Grasshopper98]. Les points de reprise sont conservés sur disque, support supposé fiable. Ils peuvent ainsi être utilisés ultérieurement pour restaurer l'agent en cas de défaillance. Cependant, lors de la restauration de l'agent il faut veiller à ne pas avoir deux agents actifs en même temps (problème rencontré en cas de défaillance lié à la perte de la connexion avec le site d'accueil de l'agent par exemple).

3.4.10 Traçabilité

Une fois lancé, l'agent mobile devient une entité autonome qui s'exécute d'une façon asynchrone. Pour la place qui l'a lancé, il est utile de disposer d'informations sur l'exécution de l'agent, sur son état et sur la place sur laquelle il se situe. Le mécanisme de traçabilité devient un moyen pour suivre et contrôler les déplacements des agents. Les agents mobiles peuvent disposer de toute l'autonomie leur permettant de choisir leurs itinéraires (espace) et les moments (temps) pour les explorer. Par exemple, il peut être judicieux pour un agent de changer l'ordre des sites qu'il doit visiter à cause d'une indisponibilité passagère d'un lien réseau ; il peut aussi décider de ne pas remettre en question l'ordre de son itinéraire, mais simplement d'attendre un peu que la perturbation du lien réseau s'estompe. On peut aussi avoir des agents dont les missions sont soit exploratoires (robots sur Internet), soit font référence à des buts ou à des besoins, charge aux agents de trouver les bons interlocuteurs pour mener à bien leurs tâches. En d'autres termes, même si les agents peuvent être programmés pour donner des informations sur ce qu'ils font, il peut être utile de disposer de mécanismes de traçabilité qui leur sont externes, pour garder un aperçu de ce qu'ils font.

3.4.11 Cycle de vie et contrôle de l'agent

Un environnement d'exécution pour agents mobiles doit offrir à l'utilisateur la possibilité de contrôler les activités de l'agent. Depuis sa création jusqu'à sa terminaison, un agent peut passer par plusieurs étapes (cycle de vie d'un agent). Un langage de programmation d'agents mobiles doit offrir au programmeur des points d'entrée qui vont lui permettre de contrôler l'activité de son agent. Un agent passe par une partie ou la totalité des étapes suivantes :

1. **Création et initialisation**. Lors de sa création, l'agent peut être initialisé avec des informations nécessaires à son exécution telles qu'un itinéraire, des préférences de son utilisateur, etc. Par ailleurs, un agent est initialisé par le système avec des informations nécessaires à son interaction avec son environnement, comme par exemple l'identité de son utilisateur.

2. **Migration**. L'agent se déplace entre les machines du système. Le but de cette migration est souvent motivé par une coopération en local avec des agents fixes ou d'autres agents mobiles s'exécutant sur le même site. Avant de reprendre ses activités, l'agent aura besoin des informations sur le nouveau site.

3. **Activation et désactivation**. Dans une application basée sur les agents mobiles, un agent se désactive en suspendant son exécution afin d'être sauvegardé sur un support non volatile. L'agent mobile est donc gelé. L'activation est l'opération inverse par laquelle l'agent est restauré afin de continuer son exécution.

4. **Terminaison**. Une fois que sa tâche est réalisée, l'agent se termine et son processus d'exécution est tué. Avant sa terminaison l'agent a besoin de livrer un bilan à son utilisateur.

Souvent, un environnement d'agent mobile offre aux développeurs des méthodes relatives au cycle de vie d'un agent. Ces méthodes sont appelées des « call-back ». Un agent doit être informé chaque fois qu'un événement du cycle de vie commence, réussi ou échoue (voir Table 1.1). Cela a pour but non seulement de réagir aux événements en question mais aussi d'être capable de refuser une création, une migration ou une réinitialisation après un déplacement.

Activité de l'agent	Call-backs
Création	afterBirth
Désactivation	beforeSuspend
Activation	afterResume
Migration	beforeMove, afterMove, afterMoveFailed
Terminaison	beforeDeath

Table 1.1 : Call-back dans un environnement d'agents mobiles

4 Plate-forme et Standardisation

La plupart des plates-formes pour agents mobiles ont introduit un ensemble de concepts similaires qui ont servi d'apport dans la standardisation des intergiciels agents. Alors que la FIPA[1] [FIPA] s'intéresse plus particulièrement à l'interopérabilité des agents intelligents (les efforts sont placés au niveau du langage, des protocoles et des infrastructures de communication), l'OMG s'intéresse à l'interopérabilité des agents mobiles à travers sa spécification appelée MAF [MAF], dont le premier RFP a été lancé en novembre 1995. Mars 1998 marque l'adoption finale des spécifications de la mobilité dans le standard de l'OMG, regroupées dans le « Mobile Agent System Interoperability Facility » (MASIF).

1 Foundationf or Intelligent Physical Agents.

L'idée de départ de MASIF était de définir les fonctionnalités minimales pour permettre l'interopérabilité d'agents mobiles. En particulier MASIF propose un profil par agent dans lequel sont précisées les exigences de l'agent vis-à-vis de la structure d'accueil lors de ses déplacements. Le langage de programmation de l'agent devient un paramètre de son profil. Une structure d'accueil peut accueillir tous les agents dont les profils sont compatibles avec ses possibilités. MASIF a été fait pour assurer dans un premier temps l'interopérabilité entre des agents écrits dans un même langage. MASIF ne standardise pas les opérations locales aux agents comme l'interprétation du code, sa sérialisation ou encore son exécution. L'interface de MASIF est placée au niveau de la structure d'accueil des agents mobiles, plutôt qu'au niveau des agents eux-mêmes. MASIF standardise la gestion des agents (création, suspension, reprise ou terminaison de son activité), le transfert des agents (sur des systèmes hétérogènes), le nommage des agents et des structures d'accueil (au niveau de la syntaxe et de la sémantique), le type des structures d'accueil (pour vérifier si une migration est compatible) et la syntaxe de leur localisation (pour les retrouver à travers le réseau).

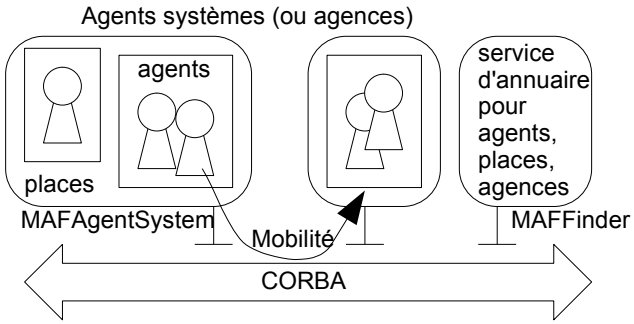

Figure 1.5 : Infrastructure du MAF et ses interfaces CORBA associées

L'infrastructure du MAF est basée sur les concepts suivants (voir *Figure 1.5*) :

- Les *agents* sont des programmes autonomes qui agissent pour le compte de personne ou d'organisation appelée *autorité*. Les agents sont exécutés dans des *places*, hébergées par des *agents systèmes*, que nous appelons également des *agences*. Pour faciliter la portabilité, la plupart des agents sont écrits avec des langages interprétés. Ils disposent d'une *thread* (fil d'exécution) leur conférant des capacités pro-actives (à leur propre initiative). L'état et le code d'un agent sont déplacés avec lui. Un agent est identifié de façon unique par son nom, lui-même composé du triplet {autorité (de qui il dépend), identifiant, type d'agent système/agence (son profil)}. Les agents mobiles se déplacent de place en place, entre des agences, à condition que leur profil (type d'agent système, langage et méthode de sérialisation) soit reconnu par l'agence destinatrice.

- Les agences qui représentent les environnements d'exécution sur un site. Une *agence* ou *agent système* assure la création, l'exécution, l'administration, le contrôle, le transport, la terminaison et les communications des agents. Elle dépend d'une autorité et est identifiée par un nom et une adresse. Plusieurs agences dépendant de la même autorité peuvent être regroupées au sein d'une région. Une machine peut contenir une ou plusieurs agences. Chaque agence définit le profil des agents qu'elle peut accueillir (par exemple une agence de type Aglet [Aglets], implique qu'elle est implantée par IBM, qu'elle utilise Java comme langage de programmation des agents et la sérialisation de Java pour la

mobilité du code et de l'état de l'agent).

- La région permet de regrouper au sein d'une même autorité des agences différentes de types également différents. Cela permet à plusieurs agences de représenter une même entité (personne ou organisation).
- La sérialisation permet de créer des agents à distance après avoir vérifié (agence destinatrice) que l'authentification du demandeur en a les droits. Un mécanisme de localisation du code de l'agent à migrer permet de reconstruire (désérialisation) l'agent sur l'agence destinatrice. Ce code peut être dans une agence ou dans un objet non CORBA comme un serveur Web.

Cette infrastructure est réalisée par deux interfaces CORBA. L'interface **MAFAgentSystem** doit être implantée par l'agent système (agence) pour gérer le cycle de vie des agents (création, suspension, reprise et terminaison d'activité), mais aussi pour la réception d'agents mobiles et le transfert des classes (code) des agents mobiles. La deuxième interface **MAFFinder** prend en charge l'enregistrement et la recherche des agents, des places et des agents systèmes (agence).

MASIF repose principalement sur les services CORBA suivants :

- La localisation d'agents afin de les contrôler et de permettre qu'ils se retrouvent pour interagir.
- L'administration afin de créer, détruire, suspendre ou copier les agents.
- Le transport pour transférer les agents suivant différents protocoles.
- La communication pour permettre les interactions entre une agence et les agents, indépendamment de leurs localisations.
- La sécurité afin de protéger les places des accès non autorisés, de signer les agents ou de chiffrer les données.
- La persistance pour sauvegarder périodiquement les états des agents.

5 Conclusion

Dans ce chapitre, nous avons présenté les différentes technologies utilisées pour la répartition et la communication entre les différentes entités d'une application répartie. Ainsi, nous avons situé la technologie d'agents mobiles par rapport aux différentes techniques présentées.

Un agent mobile est une entité autonome qui se déplace d'une machine à l'autre sur le réseau, sans perdre son code ni son état. C'est l'environnement d'exécution qui se charge d'assurer cette fonctionnalité. Il permet la création et la migration d'un agent, la communication et l'échange de messages entre les agents mobiles et il assure la sécurité de l'agent et de son site d'accueil.

Avec la multitude de plates-formes d'exécution, des efforts dans le domaine de la standardisation ont été déployés. Nous avons présenté le standard MASIF qui définit les fonctionnalités minimales pour permettre l'interopérabilité d'agents mobiles.

Dans le chapitre suivant, nous évoquons l'intérêt des agents mobiles à travers des applications réelles basées sur cette technologie. Ces applications vont insister sur

l'efficacité de cette technologie pour améliorer la fonctionnalité des applications réparties lorsqu'elle est bien employée. Ainsi, nous présentons les deux problématiques que nous avons essayées de résoudre au cours de la thèse, à savoir : l'utilisation des agents mobiles pour répartir la charge de calcul d'une application distibuée et leur utilisation pour diminuer le trafic sur le réseau.

Chapitre 2

Agents mobiles et applications réparties : un tour d'horizon

1 Introduction

Avec l'évolution du système distribuée nous avons vu l'apparition d'une nouvelle génération de terminaux qui sont inadaptés aux longues opérations, en particulier dans les réseaux où la connexion est facturée à la durée (réseau mobile). Un client qui désire chercher un billet d'avion depuis son terminal mobile aura besoin d'une connexion permanente entre le client et les différentes agences de voyages afin de choisir le meilleur prix. L'utilisation d'un agent mobile permet au client de créer un agent autonome et de charger ce dernier de la recherche de son billet. Le client n'a pas besoin de rester connecté longtemps après l'envoi de l'agent, à la nouvelle connexion le client récupère les résultats de la recherche. En plus lors de son déplacement l'agent se rapproche des données réparties [Bernard02] ce qui lui permet d'adapter ces données aux besoins de son client et de ne lui envoyer que les données utiles, ce qui permet d'éviter le transfert de données intermédiaires et de réduire le trafic réseau.

Dans ce chapitre, nous allons faire un tour d'horizon sur l'utilisation de la technologie d'agent mobile dans des applications réparties, trois catégories d'application sont presentées et ceci en fonction du rôle joué par l'agent mobile : délégation[1], filtrage[2] et calcul[3].

Nous allons montrer par la suite que dans certaines applications telles que : le commerce électronique, les cartes à puce, l'informatique nomade et la recherche d'information sur le web, l'agent agit par délégation pour son client.

Trois applications sont présentées dont lesquelles l'agent mobile va adpater les données au besoin de sont client : la première application concerne l'administration de réseaux [Sahai99], dans ce type d'application seules les informations significatives seront envoyées sur la station d'administration. La deuxième application est l'utilisation des agents mobiles pour la géstion de réseaux actifs. La toisième application represente

1 L'agent est crée par un client pour asurer une tâche précise pour ce dernier.

2 L'agent mobile effectue du traitement sur des informations collectés, les adapter ou les filtrer.

3 L'agent utilise les ressources disponibles sur la machine visitée, il se deplace sur le réseau en cherchant un endroit stratégique d'exécution.

l'adapation d'un flux vidéo au besoin de la machine cliente.

La troisième catégorie d'application montre l'utlisation de l'agent comme une entité de calcul, ce qui permet la distribution d'une application et son adpatation aux charges des machines. Dans cette catégorie, une seconde application est presentée, il s'agit de l'utilisation des agents mobiles pour la réalisation d'une poule de serveurs caches adaptable au besoin de clients.

Ainsi, dans un premier temps, nous allons montrer l'apport de la technologie d'agents mobiles pour l'amélioration de ces applications et présenter les problématiques qui en découlent dans un deuxième temps. Ces deux problématiques se présentent comme suit :

- Comment utiliser les agents mobiles pour répartir la charge de calcul d'une application distribuée ?
- Comment obtenir une politique optimale pour le déplacement d'un agent mobile ?

2 Le commerce électronique

Le commerce électronique peut se définir comme toute transaction dont les parties utilisent comme support de transmission un support électronique (par exemple le World Wide Web). Le commerce électronique est avant tout une nouvelle forme de commerce. Il occasionne un échange d'un produit ou d'un service contre de l'argent. Les acteurs qui vont participer à l'accomplissement d'une transaction électronique sont :

- un client souhaitant acquérir le produit ou le service. Ce client dispose d'un moyen d'accès aux sites de commerces électroniques ;
- un site de commerce électronique représentant le marchand. Ce site propose un catalogue de produits et de services ;
- les banques à la fois du client et du serveur entre lesquelles s'effectuent le transfert de l'argent ;
- un serveur de paiement sécurisé. Ce serveur a pour fonction d'authentifier les parties concernées, de vérifier le bon déroulement de la transaction entre le client et le marchand. Il doit protéger le client et le marchand contre les tentatives de fraudes ou de malhonnêtetés d'une des deux parties. Ce serveur doit ensuite ordonner le transfert de fond entre les deux banques.

Avec l'évolution du réseau des terminaux mobiles et la possibilité de les utiliser pour se connecter sur le réseau Internet, nous avons vu l'apparition d'une nouvelle forme de commerce électronique, le M-commerce.

2.1 Le M-Commerce

Le M-commerce ou mobile commerce est une forme d'e-commerce où une transaction peut s'effectuer à partir d'un PC mais également à partir d'un terminal mobile comme un téléphone mobile ou un assistant personnel (PDA). Dans ce type d'interaction, l'utilisateur peut être amené à gérer une transaction de commerce électronique [Carlier97, Stefano00] à partir de plusieurs terminaux. Imaginons un site de vente aux enchères qui propose des voyages à prix dégriffés [Sandholm00]. Un

client peut enchérir sur un voyage à partir de son ordinateur de bureau, puis être alerté sur son portable qu'un autre utilisateur a enchéri, ce qui lui permet de renchérir. Une fois arrivé chez lui, il pourra sur sa télévision visualiser son classement et continuer l'enchère. Dans ce type de transaction, le système doit s'adapter aux différents modes de connexion utilisés par le client ; l'utilisation d'un agent mobile permet de faciliter ces évolutions.

2.2 Apports des agents mobiles

Le développement de l'*ubiquitous computing* (*n* ordinateurs pour *1* personne) amène à penser que les agents mobiles vont devenir un outil essentiel dans le cadre du commerce électronique sur l'Internet.

En effet, les utilisateurs de ces services Web désirent continuer leurs transactions quelle que soit le lieu où ils se trouvent, et quelle que soit la machine qu'ils utilisent. Cela implique une forte hétérogénéité des machines disponibles pour exécuter une seule et même transaction commerciale (ordinateur de bureau, PDA, téléphone portable, SetTopBox de télévision interactive). Dans ce cadre, l'agent mobile va jouer le rôle de l'utilisateur, pour rassembler les informations, les adapter au terminal auquel elles sont destinées, et les transmettre pour permettre de poursuivre l'exécution de l'application.

3 La recherche d'information sur le Web

Avec le développement du réseau Internet, la plupart des fournisseurs de services ont développé leur interface Web. Cette interface leur permet de présenter leurs produits et de donner des informations (numéro de téléphone, plan d'accès etc.) sur la société. L'adoption de cette solution permet aux utilisateurs du réseau Internet d'avoir accès au monde à travers leur ordinateur à moindre coût (Internet est presque gratuit). Actuellement, le modèle « client/serveur », où les échanges se font par envoi de messages à travers le réseau, est le modèle le plus utilisé. Ce modèle possède l'inconvénient d'augmenter le trafic sur le réseau et exige une connexion permanente.

Prenons l'exemple de la recherche d'un billet d'avion sur le net [Charton03]. L'utilisateur interagit avec les différentes compagnies aériennes afin de leur demander le prix d'un billet respectant ses exigences. Les informations sur ses billets sont envoyées à travers le réseau sur la machine cliente. L'utilisateur effectue un tri sur les différentes offres afin de choisir le billet le moins cher. Une fois son tri effectué, les autres informations lui sont devenues inutiles. Dans le but de diminuer les échanges intermédiaires, nous pouvons envisager d'utiliser la technologie d'agents mobiles. Ainsi l'utilisateur crée un agent dont le but est de visiter les sites des principales compagnies aériennes et de lui ramèner le meilleur tarif. Ainsi en envoyant les agents sur les sites des compagnies, les messages échangés deviennent locaux et libèrent d'autant la charge du réseau. En plus de la taille de son code, l'agent mobile transporte avec lui les informations sur le billet le moins cher. Bien que dans la communication par agents mobiles la taille de l'agent ajoute une surcharge lors de la communication, cet inconvénient se transforme en avantage par le fait que l'agent effectue des interactions locales diminuant d'autant la quantité d'information échangée.

Dans ce travail (cf. chapitre 6), nous proposons un modèle pour le déplacement intelligent des agents mobiles. Ce modèle est utilisé pour améliorer la recherche

d'information sur le Web. Une solution hybride basée sur les technologies d'agents mobiles et de client/serveur est étudiée. Nous proposons plusieurs algorithmes pour la recherche de la politique de déplacement de l'agent sur le réseau. Le but de ces algorithmes est de minimiser l'utilisation du réseau et de réduire le délai d'attente du client.

4 L'administration du réseau

L'administration d'un parc informatique est nécessaire pour assurer une utilisation optimale des matériels mis à disposition des utilisateurs ainsi qu'une détection efficace des problèmes qui peuvent survenir. De nombreux outils d'administration [Netview, SMI] existent. Ils permettent d'observer l'état d'un réseau, de le maintenir et de détecter rapidement tout dysfonctionnement. Dans le terme général d'administration, on distingue (1) l'administration de l'infrastructure réseau d'une part, qui s'attache à la gestion plus particulière des matériels (switches, hubs, routeurs etc.) et des couches logicielles « bas-niveau » les contrôlant, et d'autre part (2) l'administration système qui s'intéresse à la maintenance des systèmes d'exploitations et des logiciels utilisateurs installés sur les diverses machines. Cette décomposition nous amène à distinguer deux types de pannes sur le réseau : les pannes réseaux liées aux incidents qui touchent le réseau (coupure d'une liaison, routeur en panne etc.) et les pannes systèmes liées aux incidents qui touchent les logiciels.

Un système d'administration de réseau fournit des mécanismes de surveillance et de contrôle d'un réseau. L'administration de réseaux implique la collecte et la visualisation de données caractérisant l'état du réseau, l'identification et la gestion des fautes, la configuration et la modification du comportement des éléments du réseau et la gestion de la sécurité. Un système d'administration de réseaux comporte une ou plusieurs stations d'administration de réseaux communiquant avec des agents d'administration situés dans les éléments contrôlés du réseau. Les stations d'administration et les agents d'administration utilisent un protocole pour communiquer. Le protocole standard utilisé est le protocole SNMP (Simple Network Management Protocol). Ce protocole, défini dans le RFC 1157 [Case90], repose sur un modèle d'interaction client/serveur pour l'échange des informations d'administration définies dans les MIB[1] entre les agents d'administration SNMP situés sur les éléments du réseau et une station d'administration.

Dans [Sahai98], l'auteur a travaillé sur la conception d'un gestionnaire mobile de réseaux permettant à un administrateur réseau d'accomplir ses tâches depuis un poste de travail quelconque connecté au système d'administration par un réseau quelconque. Selon le lieu où il se trouve, un administrateur peut utiliser différents types de poste de travail : un ordinateur de poche (PDA) en réunion, un ordinateur portable en déplacement ou une station de travail dans son bureau. Le débit et la qualité du réseau dont il dispose varie en fonction du support de communication utilisé lui imposant des contraintes comme la minimisation des ressources utilisées pour son fonctionnement, le fonctionnement en présence de déconnexions et la portabilité.

Afin de satisfaire ces contraintes et de minimiser le coût de communication, l'auteur propose un système fondé sur la technologie d'agents mobiles. Dans ce système [Sahai99], une tâche d'administration est confiée à un agent mobile qui l'exécute de façon autonome sur les éléments de réseaux concernés. Une fois l'agent initialisé et envoyé dans le réseau, le gestionnaire mobile de réseaux peut se déconnecter. Lors

1 Management Information Base

d'une reconnexion ultérieure, les agents mobiles précédemment envoyés sont rapatriés, rapportant avec eux les résultats de leur exécution.

L'utilisation de la technologie d'agents mobiles pour l'administration du réseau permet de réduire les coûts de communication sur un réseau à faible débit. L'exécution d'agents mobiles sur les éléments du réseau permet de répartir des analyses traditionnellement effectuées sur la plate-forme d'administration, ce qui offre l'avantage de diminuer la charge de cette dernière. Les agents mobiles peuvent aussi être utilisés pour réaliser de façon décentralisée des tâches d'administration impliquant plusieurs éléments de réseau. L'exécution de tâches répétitives comme par exemple l'installation ou la mise à jour de logiciels, peut être déléguée à des agents mobiles.

5 Le calcul distribué et les agents mobiles

Un programme parallèle peut être vu comme un ensemble de tâches réparties sur le réseau ; ces tâches utilisent le réseau pour communiquer et pour échanger des données. L'un des problèmes principaux de l'implantation d'un algorithme sur une architecture distribuée est celui de la répartition de charges sur les machines disponibles. Ainsi, le temps global d'exécution d'un algorithme parallèle peut être très significativement dégradé par une mauvaise utilisation des ressources (capacité de calcul et réseau de communication). La distribution des tâches sur les noeuds s'intéresse à ce problème en mettant en place un mécanisme permettant de décider du placement d'une tâche avant son exécution et de la migration de celle-ci au cours de l'exécution de l'application. Le placement des tâches peut s'effectuer à deux moments différents : soit au lancement de l'application (de manière statique), soit lors de l'exécution (de manière dynamique). Dans le cas du placement dynamique, les stratégies doivent disposer d'informations en provenance des différents noeuds (informations sur la charge de machines) pour ensuite prendre des décisions de localisation des tâches.

La distribution dynamique cherche à éviter qu'un noeud ne soit inactif alors que des tâches restent en attente sur d'autres noeuds. De manière plus forte, on cherche à ce que la charge de calcul soit la même sur l'ensemble du système. Il est en effet assez intuitif de penser que le système est utilisé au mieux lorsque chaque ressource a la même charge de travail. Le placement dynamique des tâches nécessite la migration de ces dernières entre les différentes machines du réseau. Cette migration devient opérationnelle grâce à l'utilisation de la technologie des agents mobiles. Dans une application distribuée, une tâche est représentée par un agent et la terminaison de celle ci correspond à la fin de la vie de l'agent. Les échanges d'informations entre les tâches correspondent à la communication entre les agents. Quant au déplacement d'une tâche, il correspond à la migration d'un agent à travers le réseau, il s'agit de ce fait, de représenter une tâche par un agent mobile. Dans le chapitre 4, nous allons présenter l'efficacité de l'utilisation des agents mobiles dans le domaine de la programmation repartie.

Dans la thèse de Lang [Lang99], on traite le problème posé d'une autre manière, l'objectif de la thèse était de trouver le meilleur endroit pour exécuter un processus. Il s'agit d'un comportement individualiste et le placement se fait à l'initiative des entités réparties. Un agent mobile sera chargé de trouver le bon endroit pour l'éxecution du processus. Le but ne consiste en aucun cas de faire l'équilibrage de charge du système. Mais l'auteur indique que si tous les processus appliquent le même comportement cela va permettre d'aboutir à une répartition globale.

6 Les applications distribuées adaptables (cache réseau)

Les agents mobiles permettent de développer des applications distribuées adaptables à leur contexte et aux activités de leurs clients. Au cours de notre stage de DEA nous avons développé une application de routage dynamique de requêtes [Falou03a] de type client/serveur. Dans cette application, les clients ne sont pas mobiles, en revanche les requêtes, échangées entre les clients et les serveurs, sont des agents mobiles dont le but est de trouver les serveurs adéquats aux demandes de leurs clients. De leur côté, les serveurs apparaissent, disparaissent, se déplacent ou peuvent être momentanément indisponibles. Lors de leur déplacement, les requêtes (agents mobiles) peuvent mémoriser la perception volatile du réseau qu'elles se font au fur et à mesure qu'elles l'explorent. Elles s'adaptent (dans le temps et dans l'espace) à son évolution. Pour optimiser les connexions (rencontres) entre les clients et les serveurs, via les agents requêtes, le réseau se reconfigure dynamiquement en fonction de l'usage macroscopique (en considérant des ensembles d'utilisateurs) qui en est fait. Pour cela le réseau gère dynamiquement une représentation logique des interconnections entre les différentes entités. Cette représentation permet ainsi l'adaptation du déplacement des requêtes en fonction de l'évolution de l'application.

Afin de mieux satisfaire les demandes des clients, nous avons proposé de faire migrer les serveurs. Ainsi, ces derniers se déplacent sur les différentes machines afin de se rapprocher de leurs clients. Le déplacement d'un serveur (logiciel) nécessite l'existence de certaines ressources sur la machine d'accueil, un mécanisme d'appel d'offre est lancé afin de trouver la machine prête à accueillir la surcharge dûe à l'arrivée d'un nouveau serveur.

Une application de gestion de serveurs de caches basée sur ce principe a été développée. Sur le réseau Internet, un cache se place entre le client et le serveur et rapproche les documents les plus demandés (sélectionnés par sa politique de remplacement) du ou des clients les demandant. Ceci implique la gestion de copies des documents. Le cache, en délivrant une copie du document d'origine, limite le trafic en amont du cache et diminue ainsi le temps d'attente du client. Le placement de serveur cache sur le réseau permet ainsi de diminuer le trafic sur le réseau. Dans cette application les serveurs caches, coopèrent entre eux pour optimiser la duplication et la répartition des documents sur le réseau. En plus de la politique de sélection de documents, la possibilité de migration des serveurs caches permet de replacer les serveurs sur le réseau afin d'optimiser son utilisation. Dans cette application, nous avons montré que l'optimisation est double, puisqu'il s'agit de réduire les temps d'obtention des documents recherchés par les clients (approche client) et de limiter les temps de mise à jour des documents dans les caches (approche serveur) afin de limiter l'encombrement réseau.

7 Les agents mobiles et l'informatique nomade

Une application distribuée est constituée de sites variés interconnectés par des réseaux de divers types. En particulier, certains sites peuvent être mobiles (ordinateur portable, PDA, etc.) et peuvent être connectés au réseau Internet par l'intermédiaire d'une connexion sans file ou d'une ligne téléphonique à faible débit. Une application distribuée intégrant des terminaux mobiles et des réseaux sans fil ajoute des contraintes

inhabituelles dans les systèmes distribués traditionnels. Ainsi, les ordinateurs portables sont sujets à de fréquentes déconnexions volontaires (à l'initiative de l'utilisateur) ou involontaires (suite à une interruption temporaire de communication dans un réseau sans fil par exemple). De plus, les réseaux sans fil sont caractérisés par un faible débit, un coût de communication élevé et une faible qualité de service.

L'utilisation de la technologie d'agents mobiles permet de surmonter les problèmes liés à la déconnexion des sites [Chen98]. En effet, un terminal mobile crée un agent mobile et lui demande d'agir pour son compte. L'agent créé, après sa migration, peut s'exécuter dans le système même si le site mobile créateur fonctionne en mode déconnecté. Une fois le site du client connecté, il va contacter l'agent mobile afin de lui demander de revenir sur son site d'origine. Cependant, l'exécution d'un environnement d'agents mobiles dans le contexte d'un système d'informatique nomade pose plusieurs problèmes. Le premier est relatif à l'apparition et la disparition dynamique des places pouvant accueillir les agents. Cette situation se produit lorsqu'une place s'exécute sur un site mobile et que son moyen de communication est déconnecté. Un second problème se pose au niveau du nommage des agents mobiles du fait de la mobilité des sites. Celle-ci entraîne un changement d'adresse IP ou même un changement de nom de site. Dans ce contexte, se pose le problème de la localisation des agents mobiles de fait de la mobilisation des sites d'accueils [Bi05].

L'environnement *D'agent* [D'agent] offre un mécanisme de poste d'accueil pour permettre un fonctionnement en mode déconnecté. Un poste d'accueil sur le réseau fixe est affecté à chaque ordinateur portable. Le poste d'accueil conserve et envoie les agents pour le compte de l'ordinateur portable qui lui est associé. Dans ce cas un agent qui désire migrer sur un site déconnecté va migrer sur la station d'accueil associée à ce site. Quand un site mobile se reconnecte, il informe son site d'accueil de sa nouvelle adresse sur le réseau. Le poste d'accueil fait alors migrer les agents en attente vers le site mobile. Un des inconvénients de cette approche est que la communication entre les deux sites (site d'accueil et site mobile) engendre un trafic supplémentaire sur le réseau.

8 Adaptation d'un flux vidéo en fonction de la machine utilisée

Les dernières années ont vu l'émergence d'une multitude d'équipements mobiles permettant de se connecter à Internet (téléphone, assistants personnels etc.). Ces équipements sont très hétérogènes, en terme de capacité mémoire, de calcul et d'affichage. Leur connection à Internet peut se faire à travers des réseaux très différents, comme GPRS, Ethernet, Wi-Fi, des modems ou des cartes infrarouges. De plus, les performances de l'Internet sont très variables. Un serveur multimédia doit offrir des services aux différents types d'équipement ; en observant la grande évolution et la diversité des équipements disponibles sur le marché, il devient très difficile de faire des hypothèses sur l'environnement dans lequel une application répartie sera exécutée.

Le projet Odyssey [Noble97] propose une solution basée sur l'idée de mettre plusieurs versions du fichier multimédia sur le serveur, chacune de ces versions répondant à différentes contraintes de QoS. L'avantage est la simplicité de l'approche, mais elle manque de flexibilité, car le serveur ne peut gérer toutes les versions correspondant à tous les cas possibles. De plus, une nouvelle contrainte (par exemple un nouveau type de terminal) nécessite d'être prise en compte sur tous les serveurs.

L'article [Hagimont02] présente une adaptation dynamique de flux en fonction des

caractéristiques de l'environnement d'exécution avec l'utilisation de la technologie d'agents mobiles. L'architecture proposée dans ces travaux (voir Figure 2.1) est basée sur le modèle client/serveur. Un client joue une vidéo sur un terminal, le contenu de la vidéo est fourni par un serveur à travers une infrastructure de communication.

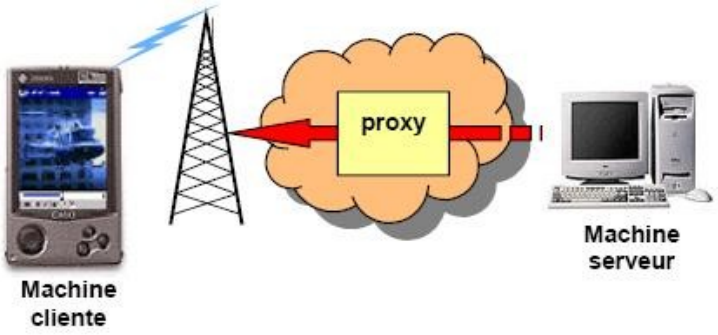

Figure 2.1 : Adaptation de flux multimédia par agents mobiles

Dans cette architecture, l'adaptation est instanciée sous la forme d'un agent mobile qui est déployé sur le site proxy (situé entre le deux machines) afin d'y effectuer un traitement sur la vidéo envoyée au client. Cette architecture basée sur les agents mobiles permet (1) de transférer la charge de calcul du client vers le proxy et (2) de diminuer la quantité de données transférées entre le proxy et le client. Autre que le flux vidéo, nous pouvons utiliser cette même architecture pour l'adaptation ou la compression des images ce qui permet de diminuer l'utilisation de la bande passante par les applications multimédia.

9 Le code mobile et les cartes à puce

La carte à puce a été conçue comme un médiateur électronique de confiance entre l'utilisateur et les infrastructures matérielles qu'il utilise. Une carte à microprocesseur n'est pas autonome en terme d'énergie et ne possède pas d'horloge interne, l'ensemble lui étant fourni par le lecteur auquel elle est connectée. En tant que telle, la carte migre de système d'information en système d'information au gré de déplacement de son porteur. Lors de chaque nouvelle connexion, elle assure la continuité du service et la persistance des informations critiques exploitées, mais surtout elle sécurise le code qu'elle véhicule. De ce point de vue, une carte à puce est vue sous la forme d'un serveur.

Les nouvelles cartes à puce sont devenues des supports mobiles d'application (Java Card [SUN05], Smart Card for Windows [Rankl03] etc.). Ainsi, un programme placé dans une carte est amené à migrer et à s'exécuter sur différents terminaux d'accueil pour représenter le porteur de la carte. En ce sens, tout code encarté est un code mobile. Par ailleurs, les dernières générations de cartes permettant de charger de nouveaux programmes tout au long de leur cycle de vie, peuvent être elles-mêmes vues comme des structures d'accueil pour code mobile. Des travaux, comme [Grimaud02], montrent la réalisation d'une plate-forme d'accueil pour code mobile sur une carte à puce, il s'agit de la plate-forme CAMILLE [Grimaud00]. Le point essentiel de cette plate-forme est le traitement de la sécurité liée à l'utilisation du code mobile. La plate-forme CAMILLE utilise le langage FACADE [Grimaud99] qui permet la vérification de la sécurité du code chargé.

10 La gestion de réseaux actifs

Les réseaux actifs s'inscrivent dans une nouvelle approche des architectures de réseaux de télécommunication où des morceaux de programmes peuvent être téléchargés et exécutés dans les noeuds du réseau affectant ainsi leurs comportements [Wetheral99]. Cette approche sous-entend un changement de la philosophie du réseau de télécommunication, qui le ferait ressembler plus à un système réparti, et qui affecte non seulement les protocoles de réseau, les services et les applications, mais également les mécanismes et les processus de haut niveau. Un des processus affectés est celui du développement et du test de nouveaux protocoles. En exploitant les propriétés des réseaux actifs, le développement de nouveaux protocoles peut être assimilé à un développement plus simple de logiciel. Les implémentations de matériel coûteuses en temps et en argent peuvent être évitées, le logiciel pouvant être développé, partagé et testé par des chercheurs individuels. Au lieu de simulations souvent basées sur des hypothèses non réalistes, les tests peuvent être effectués moyennant des déploiements de réseaux d'essai et dans des conditions réelles. Des implémentations préliminaires du protocole, qui peuvent être modifiées facilement tout au long de l'évolution de celui-ci, peuvent être utilisées pour obtenir une rétroaction utile.

Deux approches sont explorées dans le domaine des réseaux actifs [Prabhavalkar03]. La première est l'approche des commutateurs programmables qui conservent le format des paquets. Elle repose sur un mécanisme de téléchargement des programmes qui modifient le comportement du commutateur. Le téléchargement des programmes peut se faire à l'initiative de l'administrateur système ou à l'initiative des terminaux finaux qui utilisent le réseau. La deuxième approche, appellée "capsule" [Schwartz00], va plus loin. En effet, dans cette approche les paquets passifs des architectures de réseaux actuels sont remplacés par des programmes miniatures actifs, appelés capsules, placés dans les paquets transmis et exécutés sur chaque noeud traversé par le paquet lors de son déplacement sur le réseau. Les capsules peuvent ainsi être considérées comme des agents mobiles légers qui encapsulent le programme à exécuter : elles comprennent des données et du code.

11 Conclusion

Dans ce chapitre, nous avons montré l'intérêt de l'utilisation de la technologie d'agents mobiles pour les applications réparties. Le déplacement de l'agent permet de se rapprocher de la source d'information et d'effectuer des interactions locales. Le client peut équiper l'agent d'un savoir faire qui va lui permettre d'effectuer le traitement des données sur le serveur et de ne renvoyer que des informations utiles à son client. En plus, l'utilisation des agents mobiles permet l'adaptation de l'application et des données transférées aux besoins et à la capacité de la machine de l'utilisateur. Cette technologie permet aussi l'adaptation de l'application aux besoins du réseau de connexion. Dans le cas d'un réseau non fiable ou d'un utilisateur mobile, l'agent peut jouer le rôle de son client en cas de non disponibilité de ce dernier. À la nouvelle connexion, l'agent lui envoie les informations sur l'avancement de la tâche et le client peut continuer son activité.

Les applications que nous avons présentées montrent l'efficacité de l'utilisation de cette technologie par rapport au modèle basé sur les invocations distantes. Une

meilleure gestion du déplacement de l'agent permet d'augmenter l'avantage de l'utilisation des agents mobiles par comparaison au modèle traditionnel. Cette question sera traitée dans le sixième chapitre de ce travail.

Lors de son exécution, un agent mobile utilise les ressources disponibles sur la machine d'accueil. Dans une application basée sur les agents mobiles, une bonne organisation des placements des agents permet de mieux équilibrer la charge de calcul. Ainsi, la bonne utilisation des machines disponibles va augmenter la puissance de l'application répartie. Cette question sera traitée dans le quatrième chapitre de ce travail.

Dans le prochain chapitre, nous allons présenter les trois plates-formes que nous avons eues l'occasion de les tester au cours de ce travail. Une comparaison basée sur des scénarios pratiques va nous permettre de sélectionner notre plate-forme de tests qui sera utilisée dans la validation de nos algorithmes.

Chapitre 3

Choix d'une plate-forme d'expérimentation

1 Introduction

Un agent mobile s'exécute sur une plate-forme qui lui offre des services pour son déplacement, sa localisation et la communication avec les autres agents. Ainsi, la performance et la fiabilité de l'environnement d'exécution vont être déterminant pour la qualité d'une application basée sur la technologie d'agents mobiles. Une centaine de plates-formes existent à nos jours [MAL] ; elles se différencient par le type de migration de l'agent (faible ou forte), par la sécurité et la protection de l'agent et du site d'accueil, par la fiabilité offerte lors de l'exécution (tolérance aux pannes), par la facilité de retrouver un agent mobile, par les facilités d'implémentations offertes aux programmeurs et par les services offerts pour la communication entre les agents. Dans ce travail, nous nous sommes intéressés à la migration de l'agent et à la communication entre les agents. Ainsi, ces deux points ont guidé le choix de la plate-forme que nous avons adopté pour l'implémentation de nos algorithmes.

Trois plates-formes significatives (Moorea, Grasshooper et JavaAct) ont été étudiées au cours de ce travail. L'étude de ces plates-formes a permis de couvrir les différents types des paltes-formes existantes. Les plates-formes (Moorea et Grasshooper) sont issues du milieu industriel et le troisième (JavaAct) a été developpé dans un laboratoire de recherche. L'un des motifs qui nous a poussé à étudier Moorea, est le fait de fournir une migration forte des agents, ce qui n'est pas le cas dans les deux autres autres paltes-formes. Le choix de Grasshooper est motivé par sa résistance aux pannes, il offre un service de persistance sur chacune de ses agences. Les deux premières paltes-formes sont issues du milieu industriel ce qui engendre l'impossibilité de modfication ou de specification. Cette difficulté nous a poussé a étudier JavaAct, il s'agit d'une plate-forme simple distribuée sous forme de logiciel libre sous licence LGPL. Il offre la possiblité de sa specification à nos besoin et facilite la validation des algorithmes proposés.

Dans ce chapitre, nous présentons les caractéristiques de chacune de ces plates-formes, c'est en fonction de ces caractéristiques et des besoins de l'application à développer que le choix de la plate-forme appropriée est fait. Les trois plates-formes sont comparées sur la base de la migration et de la communication entre les agents. L'influence de la taille de l'agent sur le temps de la migration est étudié. Pour la communication entre deux agents, nous allons étudier l'influence de la taille du

message sur le temps de la communication.

2 Moorea

Moorea (Mobile Object, Reactive Agents) est une plate-forme d'agents mobiles, développée par France Télécom R&D [Dillenseger02]. Cette plate-forme offre un modèle réactif spécifique d'exécution et de communication d'agents basé sur la programmation synchrone (de type langage ESTEREL [Berry05]) ; elle supporte ce modèle réactif dans un environnement distribué et l'utilise pour permettre la diffusion d'événements à travers le réseau et la transparence de la gestion de la mobilité des agents pour les programmeurs. De plus Moorea implante le standard MASIF de l'OMG.

L'une des particularités principales présentée par Moorea est que : la programmation d'agents est basée sur la définition d'un comportement réactif de l'agent dans le langage synchrone ad hoc *Rhum*. Moorea permet aussi la mobilité forte, à moindre coût, de l'état de l'exécution des agents. La mobilité forte s'exprime par le fait que Moorea prend en charge la sauvegarde de l'état de l'agent (pile d'exécution et données), quelque soit l'instant dans l'activité de l'agent, d'une demande (par le programmeur) de mobilité. Ceci facilite énormément la conception et la programmation des agents mobiles et permet la réalisation d'algorithmes très fins, basés sur la mobilité.

2.1 Architecture de la plate-forme

Moorea est une plate-forme d'agents Java mobiles et réactifs dont l'architecture (voir Figure 3.1) est basée sur :

- Un objet kernel réactif (*Junior*) et un langage réactif *Rhum* inspiré du langage synchrone *Esterel*, mais avec une sémantique légèrement modifiée.
- Un framework d'objets java mobiles (SMI implantant MASIF).
- Un ORB (object Request Brocker) très souple (Jonathan), offrant les deux API : Java-RMI et CORBA.
- Une transparence au niveau du support de la mobilité a été ajoutée aux couches Jonathan, Jérémie et Rhum.

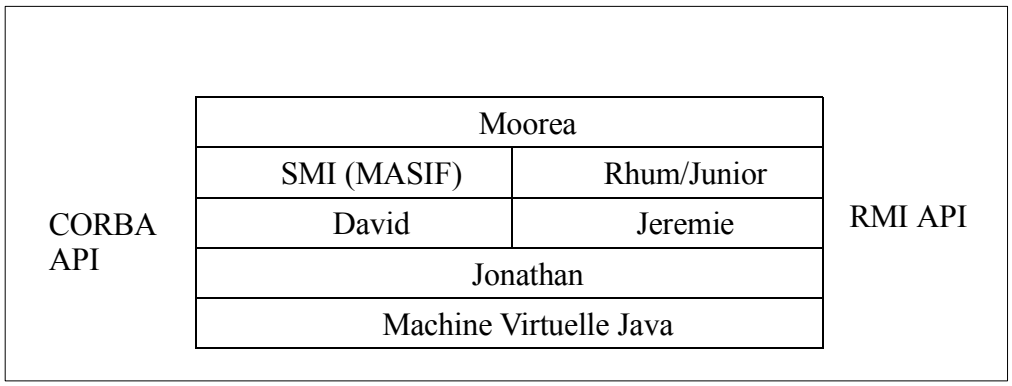

Figure 3.1 : Architecture Moorea

2.2 Les spécifications techniques

2.2.1 La communication

Les agents Moorea et les agences communiquent à travers un ORB, utilisant les services associés comme le service de *nommage*. Cette interopérabilité est assurée par la partie CORBA de Jonathan (David) ainsi que par les spécifications MASIF.

Le modèle réactif de Moorea réunit l'activité et la communication. Le caractère transparent de la mobilité se voit à travers la diffusion d'événements. Alors que les événements d'environnement restent purement locaux ; les événements pointés doivent toujours poursuivre les agents (concernés), même pendant la durée des déplacements.

2.2.2 Agent réactif et référence d'agent

Dans Moorea, l'activité d'un agent est totalement définie par son comportement réactif spécifié dans le langage de programmation *Rhum,* ainsi qu'avec l'objet java passif dont les méthodes sont appelées par le comportement réactif. Ce programme réactif est compilé en un objet java réactif incluant l'objet passif associé.

Un agent est désigné par sa référence réactive (Moorea Object Reference). Cette référence est utilisée pour gérer un agent ainsi que pour lui envoyer des événements.

2.2.3 L'exécution synchrone et les instants

Le modèle réactif de Moorea est inspiré de la programmation synchrone. Dans un tel modèle, le temps d'exécution est divisé en instants logiques (voir Figure 3.2). Ceux-ci définissent le cycle de vie d'un événement ainsi que les sémantiques de la réaction :

- Un événement est présent durant un instant si et seulement si il a été généré au cours du même instant.
- La réaction à un événement, i.e le déblocage d'une branche en attente de cet événement, est exécutée dans l'instant de sa génération.
- Quelque soit le nombre de fois qu'un événement ait été généré au cours d'un instant, il ne peut déclencher qu'une seule réaction par instant.

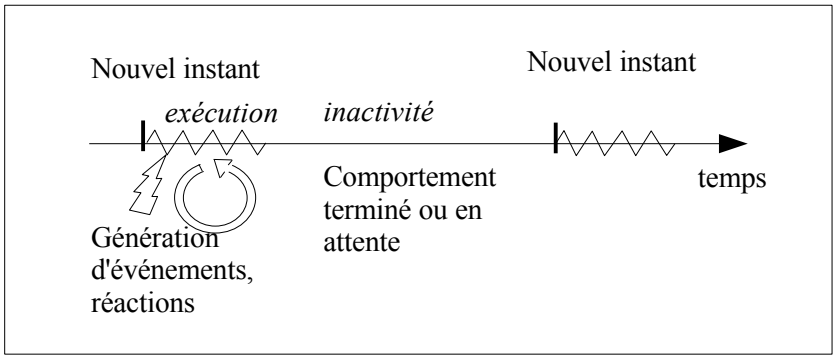

Figure 3.2 : Instant logique dans Moorea

Un instant se termine lorsque toutes les branches de tous les comportements sont soient terminées, soient en attente :

- d'une synchronisation avec une branche parallèle non terminée ;
- de la présence (ou de l'absence) d'un événement absent (respectivement, présent) dans l'instant en cours ;
- explicitement de l'instant suivant.

2.2.4 Exemple d'un programme Rhum

Un programme Rhum définit des exécutions d'actions en séquence ou en parallèle, la notion de boucles (itérations) et la génération et l'attente d'événement.

L'exemple de programme donné ci-dessous (voir Figure 3.3) définit deux branches parallèles, dont l'exécution provoque l'appel alternatif de la méthode *say-hello()*, puis *say-world()*, sur l'objet passif. Cet ordre est imposé par une génération et une attente des événements *hello* et *world*. On note que sans l'instruction *stop*, ce programme bouclerait au sein d'un même instant. A l'inverse, on effectue ici une boucle par instant. Le programme se termine à la fin du premier instant rencontré durant lequel l'événement *quit* est présent.

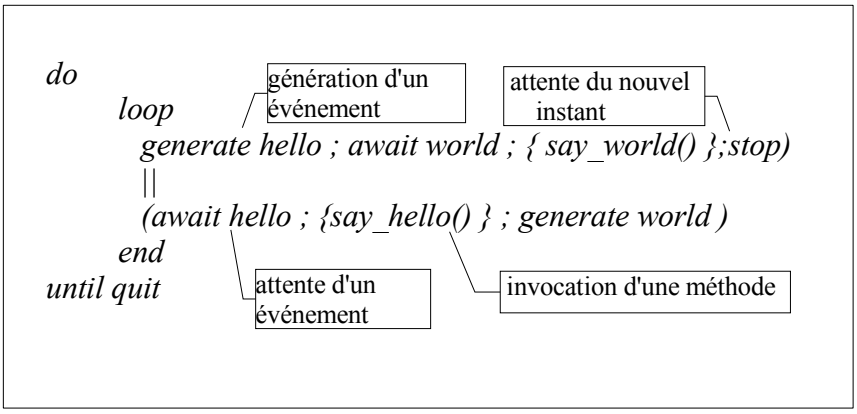

Figure 3.3 : Exemple d'un programme Rhum

2.3 Programmation sur Moorea

Dans cette partie, nous allons expliquer comment aborder la programmation sous Moorea, qui est une forme de programmation orientée *agent*.

2.3.1 Programmation d'un agent avec l'API Moorea

La programmation d'un agent à l'aide de l'API Moorea est relativement simple. Un agent Moorea est déterminé par quatre classes qui précisent les données d'initialisation de l'agent, son comportement passif, son comportement réactif ainsi que les événements auxquels il sera sensible. Mise à part la classe décrivant son comportement réactif et qui est écrite en langage Rhum, toutes les autres classes sont écrites en Java.

Il est préférable de se tenir à une certaine nomenclature afin de garder une cohérence dans le développement. Supposons que l'on souhaite programmer un agent que l'on nommera xxx, on définira les quatre fichiers sources de la manière suivante :

- **Dxxx.java** représente le fichier d'initialisation de l'agent lors de sa création. Cette classe doit dériver de la classe **DataAgent** de l'API.
- **Pxxx.java** représente le fichier définissant le comportement passif de l'agent ; il contient les méthodes que l'objet réactif appelle suite à la réception des événements. Cette classe doit dériver de la classe **PAgent** de l'API.
- **Ixxx.java** représente le fichier définissant les événements auxquels l'agent est sensible. C'est en réalité une interface qui doit dériver de l'interface par défaut **RIAgent** de l'API.
- **Rxxx.rhum** représente le fichier définissant le comportement réactif de l'agent. La syntaxe du langage Rhum bien que différente de celle du langage Java s'en approche par certains aspects. Ainsi, ce fichier source devra hériter de la classe **Pxxx.java** et implémenter l'interface **Ixxx.java** (si ces deux classes ont été redéfinies). Ce fichier se compile à l'aide du compilateur Rhum2Java qui génère un fichier source java contenant deux classes.

Une fois le comportement de l'agent défini à travers ces quatre classes, il reste à créer cet agent par le biais d'une agence. Une fois l'agent créé, son comportement est déterminé par le descriptif du fichier Rhum. On peut alors le faire migrer d'une agence à une autre. L'API Moorea fournit de nombreuses méthodes permettant la mobilité. De plus un système de *callback* permet d'intercepter les instants de son cycle de vie.

2.3.2 Démarche à suivre lors du développement d'une application Moorea

Une application Moorea est composée d'un ensemble d'agents Moorea s'exécutant à l'intérieur d'agences Moorea et effectuant des échanges inter-agents directs ou à travers l'environnement. Il faut d'abord analyser le problème que l'on cherche à résoudre, définir les agents et la communication inter-agents, et pour chacun des agents spécifier les points suivants :

- Que doit faire l'agent ?
- À quels événements doit-il réagir ?
- Comment doit il réagir à ces événements ?
- Quels événements peut-il générer ?
- À qui ces événements sont-ils envoyés ?

Une fois cette première analyse établie, la communication entre agents ainsi que leur comportement individuel sont définis. Reste alors à préciser les points suivants :

- Quelles données l'agent doit connaître et posséder ? Ces données représentent des données propres à l'agent, sur l'environnement et sur les autres agents.
- Quelles sont les modifications à apporter à ces données suite à l'évolution des agents et de l'environnement ?

3 Grasshopper

Grasshopper [Grasshopper98] est une plate-forme d'agents mobile développé par IKV++[1]. Elle permet la création d'applications multi-agents en fournissant un environnement d'exécution approprié pour eux. La plate-forme Grasshopper a été le premier système d'agents mobiles conforme au standard MASIF. Cette plate-forme est mise en oeuvre en Java et supporte la migration (mobilité faible) ainsi que les invocations distantes de manière transparente pour le programmeur. Plusieurs protocoles de communication entre les agents sont supportés (TCP/IP, Java RMI, CORBA IIOP, MAF IIOP, RMI SSL) et la plate-forme peut facilement être étendue en accédant à d'autres applications basées sur un même ORB. De même, Grasshopper supporte des mécanismes de sécurité pour les communications basées sur le protocole SSL. Finalement, Grasshopper est compatible avec les protocoles et les communications standards tels que KQML et FIPA.

3.1 Architecture de la plate-forme

La plate-forme est constituée de plusieurs entités. Nous pouvons distinguer des régions, des places, des agences, des régions d'enregistrement et différents types d'agents (voir Figure 3.4). Une région peut être découpée en plusieurs agences ; les agents sont regroupés en place. Les agences ainsi que les places sont associées à une région et elles sont enregistrées dans la région d'enregistrement correspondante. Un agent s'exécute dans une agence. Tous les agents associés à une agence seront automatiquement enregistrés dans la région d'enregistrement correspondante. Lorsqu'un agent se déplace, ces informations d'enregistrement sont actualisées automatiquement.

Les différents services offerts par la plate-forme sont les suivants :

- La communication : les interactions peuvent être exécutées par l'intermédiaire de CORBA IIOP, Java RMI ou par sockets.
- La gestion : ce service permet aux utilisateurs humains de surveiller et de contrôler les agents et les places.
- L'enregistrement : chaque agence doit être capable de tout savoir sur les agents et les places qui y sont hébergés.
- La sécurité : la sécurité interne protège les ressources des agences contre les accès, non-autorisés des agents, tandis que la sécurité externe protège les interactions entre les différentes entités.
- La persistance : ce service permet le stockage des agents et des places sur des supports persistants (système de fichiers). L'agent est restauré une fois son agence relancée.

1 Innovation Know-how Vision++

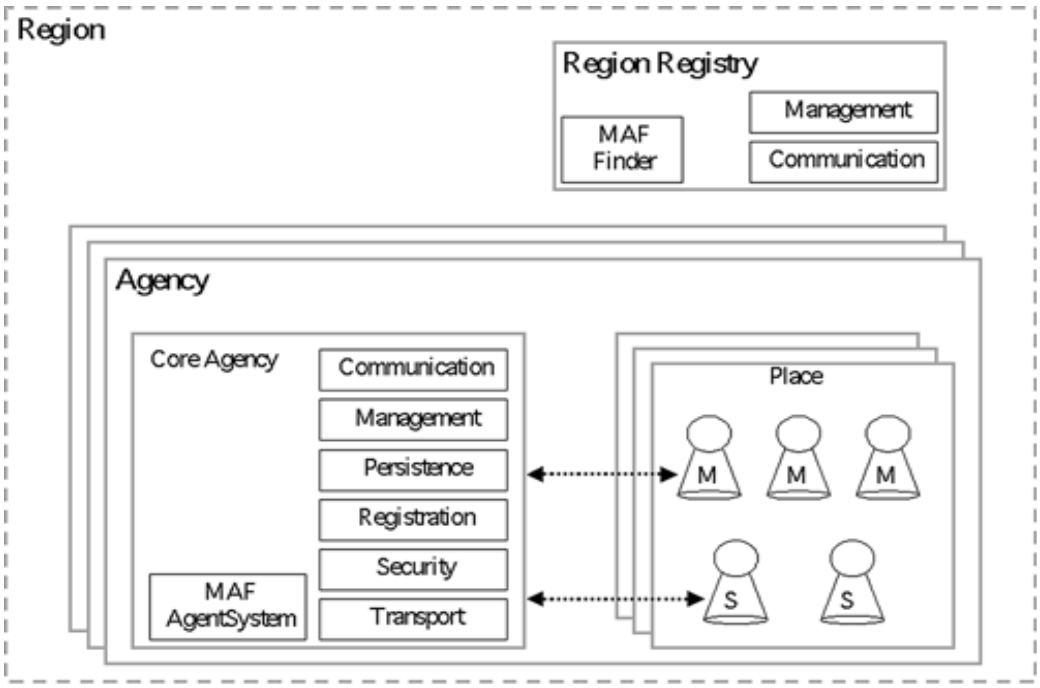

Figure 3.4 : Structure hiérarchique des composantes dans Grasshopper

3.2 Les spécifications techniques

3.2.1 La communication

Un agent peut invoquer des méthodes sur d'autres agents, dans ce cas l'agent n'a pas besoin de s'occuper de la localisation du service invoqué (transparence à la localisation). Ainsi, pour le programmeur, il n'y a pas de différence entre une invocation de méthode distante et une invocation locale au cours de laquelle l'agent mobile est co-localisé avec le service. Un agent est localisé dans une place qui assure son exécution. Les agents peuvent communiquer entre eux au sein d'une place ou avec l'extérieur en utilisant le service de communication de l'agence. Une place est incluse dans une agence qui permet le contrôle, le transport et l'enregistrement des agents.

3.2.2 Persistance

L'une des particularités de cette plate-forme est la résistance aux pannes ; ainsi Grasshopper permet d'utiliser un service de persistance sur chacune de ses agences (sauvegarde des places et agents). Les informations d'exécution des agents sont préservées afin qu'ils puissent continuer leurs tâches en cas d'arrêt imprévu du système. Les états des objets sont écrits dans un fichier local du site hôte. Une agence sauvegarde périodiquement (laps du temps déterminé par le programmeur) ses places et les agents qui s'y exécutent. En cas d'arrêt du système, l'agence est redéployée dans une autre agence et tous les agents enregistrés ainsi que les places y sont automatiquement relancés. Le fichier de sauvegarde est utilisé pour reconstruire les agents dynamiquement. L'instance de l'agent est recréée avec son état et le processus correspondant est relancé. L'agent redémarre son exécution avec les données enregistrées lors de la dernière sauvegarde qui précède son interruption. Ce service de

persistance réduit considérablement les performances temporelles du système d'exécution mais il permet d'augmenter sa fiabilité.

La plate-forme Grasshopper permet de geler un agent inactif ce qui va permettre de libérer les ressources qu'il utilise. Le rechargement de l'agent et sa ré-activation sont effectués automatiquement par le système lorsqu'une opération sur l'agent (envoi de message, exécution d'un service, ...) est invoquée.

3.3 Programmation dans Grasshopper

Les agents dans Grasshopper sont vus comme des services. Deux types d'agents sont à distinguer : mobiles et stationnaires. La classe agent peut être dérivée de la classe *StationnaryAgent* ou *MobileAgent*, ces deux classes héritent de la super-classe *Service*. Un agent persistant peut être dérivé de la classe *PersistentStationnaryAgent* ou *PersistentMobileAgent*

Pour le comportement individuel de l'agent, le programmeur est invité à définir la méthode *live()*, dont le but est de spécifier les tâches actuelles de l'agent.

On distingue deux sortes de méthodes qui représentent les interfaces essentielles entre les agents et leur environnement :

- Les méthodes qui permettent à l'agent d'accéder au noyau local de l'agence (exemple : *listMobileAgents( )*).
- Les méthodes qui permettent l'accès aux autres agents (exemple : *getState( )*).

3.3.1 Migration et structuration d'un agent

La mobilité dans Grasshopper est faible. Après sa migration, l'agent fait appel à la méthode *live()* pour continuer ses activités. Il commence toujours au début de cette méthode. Afin de permettre à un agent de continuer sa tâche, après une migration, le programmeur est invité à utiliser une partie des données de l'agent pour indiquer son évolution. La méthode *live()* de l'agent est découpée en plusieurs blocs d'exécution, chacune représente un ensemble d'opérations qui doivent être exécutées à un endroit donné. Après exécution d'un bloc, l'agent fait appel à la méthode *move()* afin de migrer sur le prochain site, une variable *state* est incrémentée de un pour indiquer que l'agent a franchi une étape dans son itinéraire. L'agent utilise le variable *state* pour connaître le bloc à exécuter après son arrivée sur le nouveau site (voir Figure 3.5).

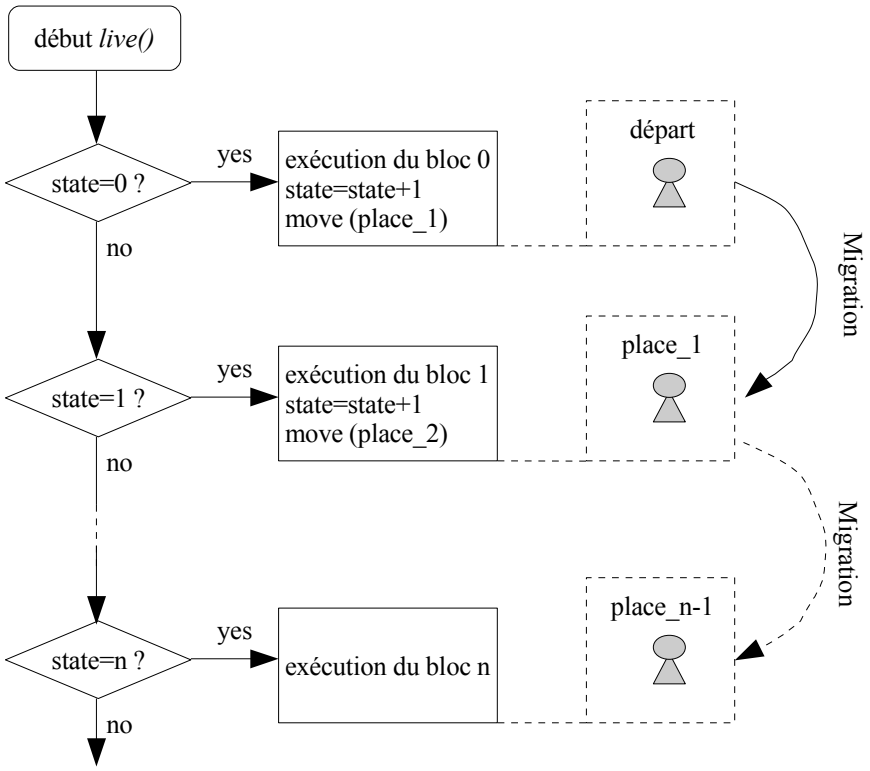

Figure 3.5 : Structure de la méthode live()

4 JavAct

JavAct [Arcangeli01] est une plate-forme Java pour la programmation d'applications concurrentes, réparties et mobiles, développée par l'équipe IAM (Ingénierie des Applications Mobiles) de l'IRIT (Institut de Recherche en Informatique de Toulouse, France) et distribuée sous forme de logiciel libre sous licence LGPL.

JavAct [Arcangeli04] fournit des mécanismes pour la programmation d'agents mobiles adaptables qui s'appuient sur les concepts d'acteur et d'implémentation ouverte.

La communication dans JavAct est asynchrone, elle repose sur le paquetage *java.rmi* (RMI : Remote Method Invocation) il s'agit d'une abstraction de la communication pour le programmeur. En plus de la communication, JavAct facilite la programmation multithreadée en offrant des mécanismes pour l'ordonnancement et la synchronisation de threads.

Enfin, JavAct repose sur la bibliothèque Java standard et son utilisation ne demande pas de modification au niveau de la machine virtuelle ce qui facilite sa portabilité sur le réseau Internet.

4.1 Architecture de la plate-forme

Un agent dans JavAct est défini [Leriche04] sous forme de micro-composants (voir Figure 3.6) ce qui lui permet de s'adapter dynamiquement aux besoins de l'environnement. Dans JavAct : adapter un agent revient à modifier un ou plusieurs de ses micro-composants.

Les principaux éléments de la plate-forme JavAct sont :

- Le système d'accueil : le rôle d'un système d'accueil est d'offrir les services nécessaires à la création et à l'hébergement d'agents. Un ensemble de sondes de bas niveau permettent de surveiller le système et d'informer les agents lorsqu'on détecte un changement.
- Le contrôleur : il joue le rôle de connecteur entre les micro-composants ; c'est le régulateur et le point de passage obligé de tout appel de méthode des composants.
- L'analyseur : l'analyseur implémente la politique d'adaptation des composants, il définit un ensemble de règles qui vont lui permettre de décider du remplacement d'un composant et ceci en fonction des événements liés aux variations du contexte d'exécution.

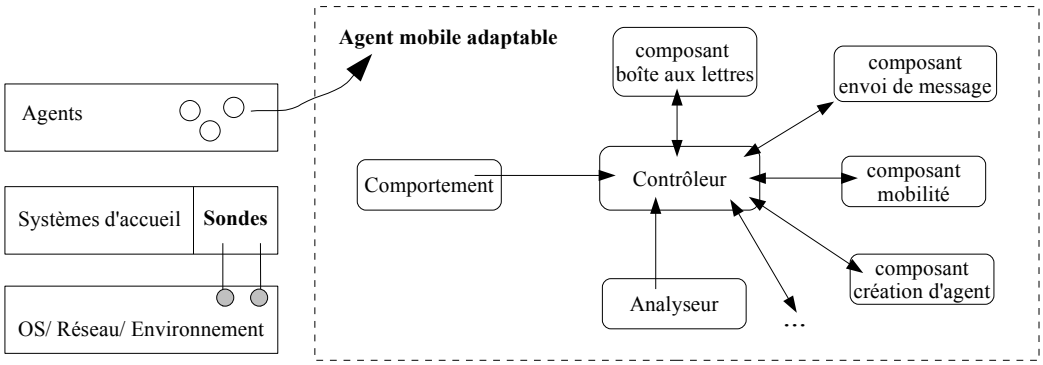

Figure 3.6 : Architecture des agents mobiles adaptables [Leriche04]

4.2 Les spécifications techniques

4.2.1 La communication

La communication entre agents étant supposée sûre, la mobilité pose le problème de l'acheminement des messages. Le protocole d'acheminement des messages est défini par : l'acteur d'origine qui se transforme lors du déplacement en un *proxy* du nouvel acteur (distant) ; il ne traite plus les messages qu'il reçoit mais les transmet. Ainsi, l'acteur reste connu à son adresse d'origine qui demeure valide, et l'envoi de messages reste possible à tout instant y compris pendant que l'acteur se déplace (les messages sont stockés au niveau de l'acteur d'origine, puis transmis quand le clone distant est opérationnel). D'autre part, les communications sortantes (émissions de messages) restent valides (pour cela, les références d'acteur doivent être visibles à travers le réseau). Le mécanisme de mobilité est donc transparent pour le système de communication. La multiplication des déplacements d'un acteur a alors pour effet de construire une chaîne de liens de poursuite à travers laquelle l'acteur mobile peut être localisé et les messages acheminés.

4.3 Programmation avec JavAct

Il convient de respecter un certain ordre dans les étapes de développement d'une application JavAct :

1. Spécification des profils des comportements et des acteurs (définition d'interfaces Java).
2. Génération automatique des classes pour les messages et des classes abstraites pour les comportements.
3. Implantation des comportements (codage des classes qui étendent les classes abstraites générées).

Pour effectuer la génération automatique, il suffit d'exécuter le script *javactgen* qui se chargera d'extraire toutes les informations nécessaires à la génération. Deux types de fichiers sont générés :

- les implantations de *QuasiBehavior* implantant le code des méthodes *become* ;
- les classes correspondant aux messages échangés entre les différents acteurs.

En fait, on peut spécifier deux sortes de méthodes auxquelles correspondent deux sortes de messages :

- les messages asynchrones classiques correspondant aux méthodes renvoyant *void* ;
- les messages avec retour correspondant aux méthodes qui renvoient une valeur.

Pour les messages *asynchrones* classiques, le préfixe *JAM* est ajouté devant le nom du message pour obtenir le nom du fichier java correspondant. Pour les messages *avec retour*, on ajoute le préfixe *JSM* ainsi que le nom du type de retour, pour distinguer les messages de même nom mais renvoyant des valeurs de types différents, et la méthode *getReply()* est engendrée avec le «bon» type de retour.

4.3.1 Migration et structuration d'un agent

La mobilité dans JavAct est faible. Après sa migration, l'agent fait appel à la méthode *run()*. Pour continuer son exécution, il se branche au tout début de cette méthode. Afin de permettre à un agent de continuer sa tâche, après une migration, le programmeur est invité à utiliser des techniques de programmation comme nous l'avons présenté pour la méthode *live()* de Grasshopper.

5 Comparaison et choix de la plate-forme

En plus de la migration de l'agent, une plate-forme d'agents mobiles offre des services pour la protection de l'agent et du site visité, pour la localisation de l'agent et pour sa communication avec les autres agents. Dans notre travail, nous nous sommes intéressé principalement à la migration de l'agent et aux communications entre les agents. Pour effectuer notre choix, les trois plates-formes sont comparées essentiellement sur ces deux aspects.

5.1 Communication entre deux agents

Dans une plate-forme d'agents mobiles, les agents doivent communiquer pour échanger des données. Pour mesurer le temps de communication, nous avons pris l'exemple de deux agents situés sur deux machines dans un réseau local ; les deux

machines sont connectées par un lien de 10Mbs. Le but est de mesurer l'influence de la taille du message sur le délai d'attente lors de la communication entre ces deux agents et ceci pour les trois plates-formes.

Nous avons choisi un scénario simple pour mesurer ce temps : un premier agent crée le message et l'envoie au second. Le temps de l'envoi se mesure par le temps écoulé entre l'envoi du message par le client et sa réception par le serveur. Les horloges des deux machines ne sont pas synchronisées et les résultats deviennent non-fiables.

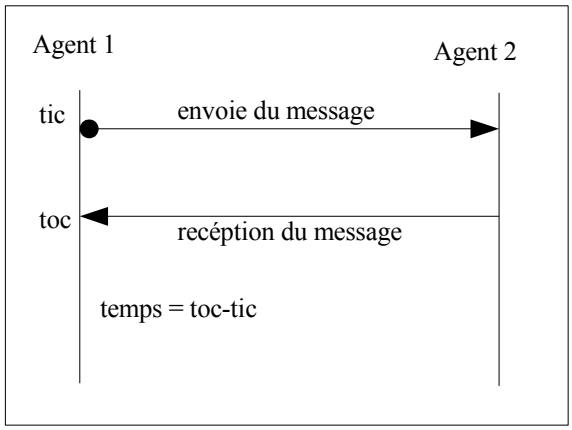

Figure 3.7 : Communication entre deux machines

Afin d'obtenir des résultats fiables, nous avons adopté un autre scénario pour effectuer la mesure. Dans ce scénario (voir Figure 3.7), à la réception du message, le deuxième agent le renvoie au client. Ainsi, le temps de l'envoi du message correspond au temps écoulé entre son envoi par le client et sa réception par ce dernier divisé par deux.

$$temps = \frac{toc - tic}{2}$$

Nous avons effectué les mesures pour les trois plates-formes en faisant varier la taille du message échangé. Les trois courbes (voir Figure 3.8) représentent le temps de communication. Sur ce schéma, l'axe des abscisses représente la taille du message en Mb et l'axe des ordonnées représente le temps d'attente en ms. Les trois courbes montrent que la communication dans Grasshopper est la plus lente et que la plate-forme JavAct est la meilleure. L'utilisation de JavAct permet d'avoir un temps de communication deux fois plus rapide que les deux autres plates-formes.

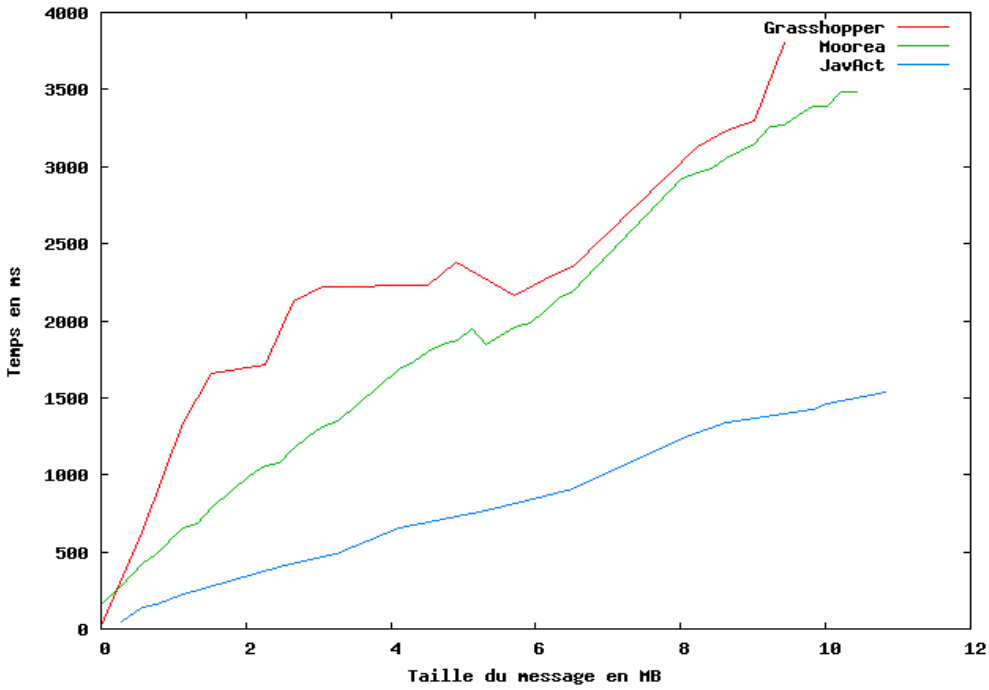

Figure 3.8 : Influence de la taille du message sur le temps de communication

5.2 Migration d'un agent mobile

Lors de sa migration, en plus du code qui décrit son comportement, l'agent transporte ses données. Afin de mesurer l'influence de la taille de l'agent sur son déplacement, nous avons pris l'exemple de la migration d'un agent entre deux machines dans un réseau local. Ici aussi, les deux machines sont connectées par un lien de 10Mbs.

Les deux machines ne sont pas synchronisées comme dans le paragraphe précédent. L'agent se trouvant sur la première machine migre (1) sur la seconde. Une fois arrivé sur la deuxième machine, il effectue une deuxième migration et retourne (2) sur la machine M1. Ainsi la même horloge est utilisée pour effectuer les mesures (voir Figure 3.9).

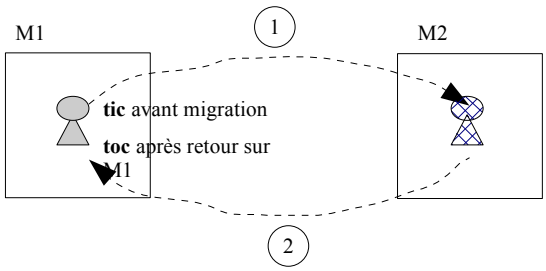

Figure 3.9 : Migration d'un agent mobile

Le temps de la migration d'un agent correspond au temps écoulé entre son envoi par la première machine et sa réception par cette dernière, divisé par deux :

$$temps = \frac{toc - tic}{2}$$

Nous avons effectué les mesures pour les trois plates-formes en faisant varier la taille des données transférées avec l'agent. Les trois courbes (voir Figure 3.10) représentent le temps de la migration de l'agent. Sur ce schéma, l'axe des abscisses représente la taille de l'agent en Mb et l'axe des ordonnées représente le temps de migration en ms. Ces courbes montrent que la migration dans Grasshopper est la plus lente et que la plate-forme JavAct est la meilleure. L'utilisation de JavAct permet d'avoir un temps de migration deux fois plus rapide que dans Moorea et quatre fois plus rapide que dans Grasshopper.

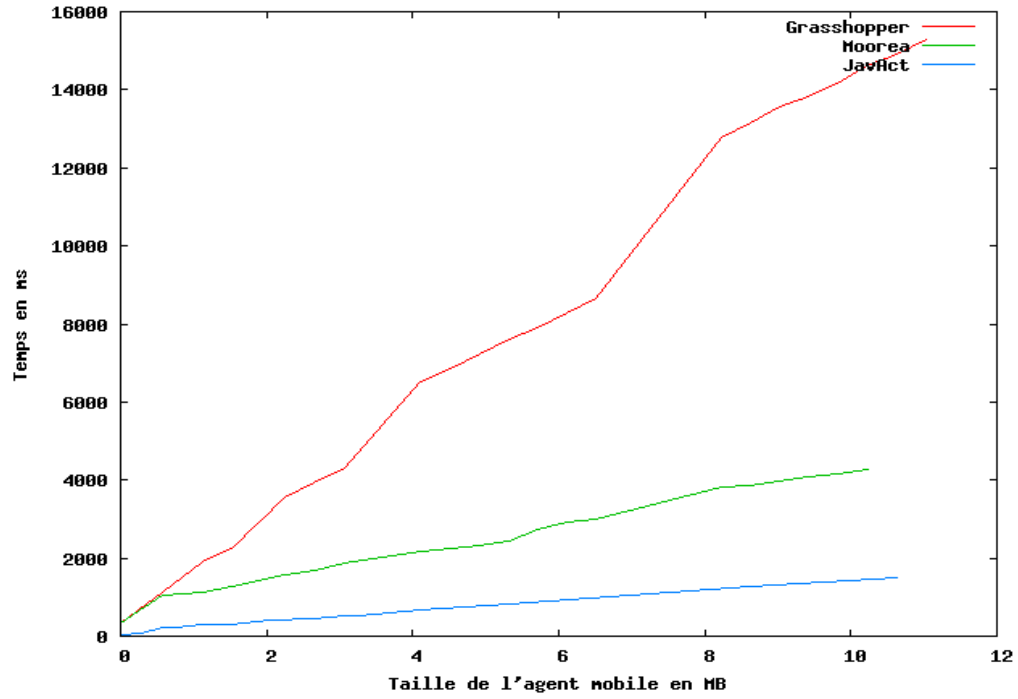

Figure 3.10 : Influence de la taille de l'agent sur le temps de le migration

6 JavAct sur le Web

Certaines applications nécessitent la participation d'un grande nombre de machines afin de réaliser la tâche demandée (exemple : recherche de la clé de chiffrement en un temps raisonnable [Distributed.net]). Le développement du réseau Internet qui regroupe un grand nombre des machines nous amène à les rassembler afin de profiter de leur puissance. Il s'agit du concept de *grid computing* (calculs en grille) qui met en réseau des ordinateurs afin de les faire travailler sur un même projet. On additionne ainsi, à moindre coût, leur puissance de calcul ou de stockage. Cette technique tire parti du temps de non utilisation des processeurs. Elle permet de résoudre des problèmes que l'utilisation d'un super-ordinateur ne pourrait pas traiter dans un temps réaliste.

La majorité des utilisateurs utilisent le réseau Internet à travers leur navigateur pour accéder au Web. Dans le but de faciliter l'ajout de leur machine sur la grille de calculs,

nous avons développé une extension de la plate-forme JavAct qui permet son utilisation à travers un navigateur.

Dans JavAct une machine qui accueille un agent doit avoir un programme qui permet de le faire, en effet, il s'agit d'une agence. Créer une agence sur la machine permet donc de l'ajouter à la grille de calculs. Ainsi, nous avons développé une agence Applet qui est chargée à travers le navigateur (voir Figure 3.11). L'applet que nous avons développée est une applet signée ce qui permet d'avoir toutes les fonctionnalités offertes par une agence.

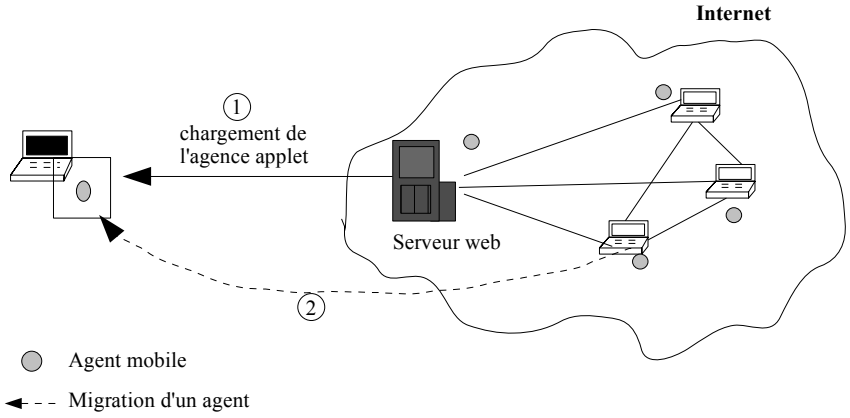

Figure 3.11 : JavAct et agence applet

Un utilisateur qui désire ajouter sa machine utilise son navigateur pour se connecter au serveur web. Il charge l'agence Applet (programme java) sur sa machine (1). L'activation de cette applet crée une agence, permettant à la machine d'accepter la migration d'un agent pour s'y exécuter (2). Le développement d'une application sous forme d'agents mobiles permet de profiter de la puissance de la machine ajoutée. Une machine qui lance l'applet est considérée comme un support à l'exécution des agents. L'utilisateur peut utiliser l'applet chargée pour lancer sa propre application, bénéficiant ainsi de la puissance des autres machines du réseau.

Pour mesurer l'influence de l'utilisation d'une agence applet sur la migration d'un agent mobile, nous avons testé cette migration entre deux machines avec une agence applet activée sur chaque machine. Sur le schéma (voir Figure 3.12), l'abscisse représente la taille de l'agent mobile et l'ordonnée représente le temps de la migration entre les deux machines. Nous pouvons remarquer que la migration entre deux agences normales est plus rapide que celle entre deux agences applets.

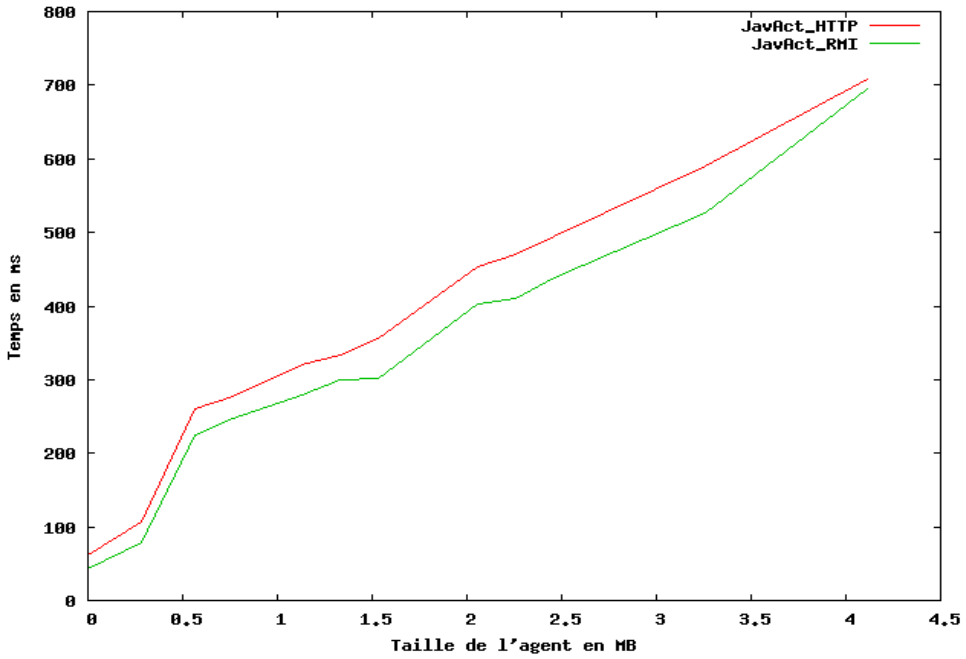

Figure 3.12 : Migration d'un agent entre deux agences applet

7 Conclusion

Dans ce chapitre nous avons présenté trois plates-formes représentatives des besoins de notre travail et que nous avons pu tester : Moorea, Grasshopper et JavAct. La plate-forme Moorea, à l'inverse des deux autres, offre la possibilité d'une migration forte des agents à tout moment, la rendant transparente pour le programmeur. Dans Grasshopper et JavAct le programmeur doit utiliser des techniques de marquage dans son code (cf. §3.3.1) afin de contrôler l'agent lors des migrations.

La particularité de la plate-forme Grasshopper est la sécurité des agents et la résistance aux pannes ; ainsi Grasshopper permet d'utiliser un service de persistance sur chacune de ses agences. Il faut signaler que la société IKV a arrêté le développement du projet Grasshopper.

Les deux points sur lesquels nous avons mis l'accent au cours de ce travail sont liés à la migration intelligente de l'agent et à la distribution d'une application par agents mobiles. JavAct est la plate-forme que nous avons choisie pour effectuer nos tests et valider nos algorithmes. Ce choix est motivé par le fait que la communication et la migration dans JavAct sont plus rapides par comparaison aux deux autres plates-formes. Afin de faciliter la distribution d'une application à travers le réseau Internet, nous avons développé une extension de JavAct. Elle permet à un utilisateur de charger une applet à travers son navigateur, l'applet chargée lui permet d'utiliser sa machine comme support pour l'exécution d'agents mobiles.

Chapitre 4

Agent mobile et calcul réparti

1 Introduction

Avec l'évolution rapide des capacités de traitement des stations de travail et l'amélioration de la qualité du réseau reliant ces stations, le calcul distribué a grandement évolué au cours de cette dernière décennie. Internet regroupe des centaines de milliers de machines connectées par le réseau IP et qui permet la communication entre ces différentes machines. Ainsi, le réseau IP représente un environnement réparti dans lequel des machines peuvent fonctionner en parallèle. Une machine connectée au réseau Internet est en état de repos à 95% de son temps [Ibrahim02], une bonne utilisation permet de profiter de sa puissance sans affecter ses propres travaux. Ainsi, la bonne utilisation de cette gigantesque plate-forme permet d'augmenter potentiellement "indéfiniment" la puissance des ordinateurs. Il faut noter que cette distribution n'est pas sans limites, chose que nous présentons avec la loi d'Amdahl [Amdahl67].

La manière standard de rendre parallèle un algorithme consiste à partitionner les données sur lesquelles il travaille et à affecter chaque sous-tâche à une machine particulière. Bien entendu, sauf dans le cas où les calculs d'une tâche ne dépendent pas des autres tâches, il faut mettre en oeuvre des communications entre les machines exécutant des tâches dépendantes. On pourra donc décomposer les algorithmes en étapes de calculs et en étapes de communications. L'objectif recherché dans la distribution d'un algorithme étant d'accélérer sa vitesse d'exécution, nous aurons constamment à vérifier deux exigences souvent contradictoires : minimiser la communication et équilibrer au mieux la charge de travail des machines disponibles.

Dans le domaine de l'équilibrage de charge, il existe deux grandes familles d'algorithmes : centralisés et décentralisées. Pour un algorithme centralisé, les informations sur la charge des différentes machines seront envoyées sur une machine centrale qui va se charger de la répartition et du déplacement des tâches. Pour un algorithme décentralisé, la décision se fait en local sur chaqu'une des machines ; celles-ci ne connaissent que les caractéristiques de leurs voisins directs. Les algorithmes utilisés sont itératifs et l'équilibrage de charge s'effectue de proche en proche dans le temps. Il s'agit d'un équilibrage de charge dynamique dans lequel le déplacement de charge se fait en cours de l'évolution de l'application.

Nous nous intéressons au problème d'équilibrage de charge dynamique et décentralisé. L'équilibrage de charge nécessite le déplacement des tâches entre les

différentes machines ; une tâche sera donc modélisée par un agent mobile et le déplacement d'une tâche correspond à la migration de l'agent. Nous allons ainsi montrer l'utilité de l'utilisation de cette technologie pour le calcul distribué.

2 Répartition de charge

Un programme parallèle peut être vu comme un ensemble de tâches réparties sur le réseau ; ces tâches vont utiliser le réseau pour communiquer et pour échanger des données. Dans la plupart des cas, les tâches ne peuvent pas s'exécuter d'une manière indépendante ; il existe un certain ordre à respecter lors de l'exécution des tâches. Concevoir un algorithme parallèle, c'est avant tout déterminer un ordonnancement des tâches. L'un des problèmes principaux lors de l'implantation d'un algorithme sur une architecture distribuée est celui de la répartition de ces tâches sur les machines disponibles. Une mauvaise distribution permet d'avoir une situation de non équilibre, avec des machines trop chargées et d'autres moins. Une tâche s'exécutant sur une machine chargée va prendre plus de temps ce qui va ralentir les autres tâches qui en dépendent et par conséquent allonger le temps global d'exécution. De manière générale, l'équilibrage de charge est utilisé pour améliorer une application distribuée et intervient aussi bien au niveau des processus [Hegarty97] qu'au niveau des données [Tonguz03]. L'enjeu de l'équilibrage de charge est d'obtenir des performances optimales pour le système auquel il est appliqué.

Le temps global d'exécution d'un algorithme parallèle peut être très significativement dégradé par une mauvaise utilisation des ressources (capacité de calcul et réseau de communication). La distribution des tâches sur les noeuds s'intéresse à ce problème en mettant en place un mécanisme permettant de décider du placement d'une tâche avant son exécution et de la migration de celle-ci au cours de l'exécution de l'application. Le placement des tâches peut s'effectuer à deux moments différents : soit au lancement de l'application (de manière statique [Tantawi85]), soit dynamiquement lors de l'exécution. Dans le cas du placement dynamique, les stratégies doivent disposer d'informations en provenance des différents noeuds (informations sur la charge de machines, etc...) pour ensuite prendre des décisions de localisation des tâches.

2.1 Placement statique versus dynamique

Le placement statique de tâches nécessite une connaissance préalable de différents paramètres associés aux tâches (temps de calcul et temps de communication) et n'utilise pas ces informations en provenance directe de l'exécution. La bonne connaissance de ces informations permet ainsi de faire une bonne modélisation du problème et de mieux placer les différentes tâches. De ce fait, les approches statiques ne sont pas toujours applicables aux systèmes distribués, ces derniers pouvant évoluer de façon imprévisible. A l'inverse, les stratégies purement dynamiques ne demandent aucun préalable à l'exécution. Les décisions qu'elles sont amenées à prendre n'utilisent que des informations obtenues en temps réel au cours de l'exécution. La reconfiguration dynamique s'utilise avantageusement lorsque le système permet la migration de tâches, ce qui offre plus de liberté au processus de distribution.

2.1.1 Placement statique

Cette solution n'est envisageable que lorsque l'on connaît à l'avance assez précisément le schéma d'exécution du programme ; ce qui est le cas dans de nombreux problèmes de calcul numérique. Un placement est une fonction (notée *loc* pour localisation) qui à une tâche t associe un processeur.

$$\forall t \in T,\ \exists p \in P, loc(t) = p$$

T représente l'ensemble des tâches à placer et *P* l'ensemble des processeurs.

Si *p* représente le nombre de processeurs et *n* le nombre de tâches, alors il existe p^n placements possibles, ce qui rend trop coûteux l'exploration de toutes les possibilités de placement.

De très nombreuses stratégies de placement ont été proposées dans la littérature [Kwok99]. Ainsi on distingue :

- Les algorithmes exacts [Shen85, Sinclair87] dont le principe repose sur une exploration de toutes les solutions possibles afin de choisir la solution optimale. Cette stratégie est très coûteuse en pratique.
- Les algorithmes itératifs [Legrand04], qui partent d'une solution complète que l'on améliore par un processus de réorganisation élémentaire de tâches. Le nouveau placement sera accepté si l'on obtient une amélioration en temps de calcul.

Il est possible de considérer des solutions intermédiaires entre ces deux stratégies. Ainsi, un algorithme exact sera utilisé pour trouver une disposition de démarrage ; cette disposition sera ensuite utilisée dans un algorithme itératif afin de l'améliorer.

Dans un placement statique, même si l'on trouve la solution avec un algorithme exact, cette dernière n'est pas optimale, au sens d'une utilisation efficace des ressources disponibles. Des processeurs peuvent rester en attente de la terminaison d'une tâche, d'où notre orientation vers une solution dynamique.

2.1.2 Placement dynamique

La distribution dynamique cherche à éviter qu'un noeud ne soit inactif alors que des tâches restent en attente sur d'autres noeuds [Cybenko89]. Soit de manière plus forte, on cherche à ce que la charge de calcul soit la même sur l'ensemble du système. Il est en effet, assez intuitif de penser que le système est utilisé au mieux lorsque chaque ressource a la même charge de travail. L'un des problèmes que l'on rencontre lors d'un placement dynamique est lié à la collecte des informations concernant la charge des noeuds. Ces données sont utilisées pour la prise de décision lors du déplacement des tâches [Georgousopoulos03]. Cette collecte d'informations sur la charge des noeuds va pénaliser les performances de l'algorithme d'équilibrage de charge. Les meilleures stratégies de placement dynamique des tâches sont le plus souvent celles qui utilisent la meilleure stratégie de collecte d'informations.

Le processus de placement dynamique est composé de deux étapes bien distinctes : une étape d'évaluation et de collecte d'informations sur le système et une étape de prise de décision. La première étape consiste en l'obtention de l'état de charge des noeuds et

en une estimation d'un état global du système. Cette étape peut être absente dans le cas d'une prise de décision en fonction de l'état local du noeud ; c'est la solution que nous allons proposer dans le reste du chapitre. La deuxième étape consiste à utiliser cet état global du système afin de faire migrer les tâches sur les noeuds les moins chargés.

3 Collecte d'informations

La collecte d'informations est un processus qu'il ne faut pas négliger aussi bien en coût de communication (quantité d'informations nécessaires pour la prise de décision) que dans la validité dans le temps de ces informations (les informations collectées sont-elles encore pertinentes au moment où l'estimation est faite ?). Les meilleures stratégies de placement dynamique des tâches sont le plus souvent celles qui utilisent la meilleure stratégie de collecte d'informations : des collectes trop fréquentes surchargent le système de communication, alors que des collectes trop espacées donnent des estimations peu fiables.

Lors de la collecte d'informations, la question suivante s'impose : à quel moment les noeuds envoient-ils leurs états ? Deux stratégies s'opposent [Martin-flatin99] : celles où le noeud prend la décision d'envoi en fonction de son état et celles qui privilégient le rôle des collecteurs où les noeuds ne faisant que répondre à des demandes. Dans les deux cas, l'envoi peut se faire d'une manière périodique ou bien lors de l'apparition d'un état particulier. On peut toujours envisager une solution intermédiaire : ainsi des demandes périodiques, avec une faible fréquence, sont faites sur les différents noeuds ; parallèlement lorsqu'un noeud détecte un changement brutal de sa charge, il envoie une notification pour indiquer le changement.

La collecte d'informations pose également la question de savoir vers qui les données seront-elles envoyées ? Deux stratégies sont envisageables : une collecte centralisée ou une collecte décentralisée. Bien que la collecte décentralisée nécessite moins de communications, la solution centralisée permet d'obtenir une estimation globale facilitant d'autant la prise de décision. Une solution intermédiaire consiste à adopter une stratégie hybride. Ceci permet de diminuer le surcoût lié à la communication tout en conservant une vision globale sur le système.

L'utilisation d'un agent mobile est envisageable pour cette collecte. En effet, comme les travaux dans [Cao03], qui se situent dans le domaine de l'aiguillage de charge entre des serveurs Web distribués. Le but est de choisir le serveur qui va satisfaire la demande du client. Des agents mobiles vont ainsi se déplacer entre les différents serveurs afin de collecter des informations sur leurs activités. Ces informations vont ainsi guider le système lors de la distribution de la charge. L'agent mobile est utilisé pour la collecte des informations sur la charge des machines. Il permet donc de diminuer le trafic sur le réseau. D'autres travaux, comme dans [Padiou05, Charpentier05], ont utilisé les agents mobiles pour l'équilibrage de charge dans un réseau dynamique ; sur ce réseau les noeuds peuvent apparaître et disparaître et les liens entre les machines peuvent changer dans le temps. Un agent mobile sera chargé de faire l'équilibrage de charge, il se déplace aléatoirement entre les sites. Il collecte ainsi des informations sur la charge de chaque machine afin de calculer une charge moyenne (globale) pour toutes les machines. Une fois la vision globale construite, l'agent joue le rôle d'un répartiteur. Sur chaque machine visitée, l'agent calcule la charge de la machine, il la compare avec la charge moyenne et il ordonne le déplacement de certaines tâches dans le cas d'une surcharge. Dans ces travaux, le

déplacement se fait lui aussi d'une façon aléatoire entre les machines. Afin d'augmenter la performance du système, plusieurs agents mobiles peuvent être utilisés et des collaborations entre ces agents sont envisagées.

Au niveau de la collecte des données, une architecture hybride est proposé dans ce chapitre. Dans cette architecture, la prise de décision se fait en fonction de la charge locale de la machine. En plus, un agent mobile collecteur sera utilisé pour construire une vision globale du système, contrairement aux travaux de [Padiou05] où c'est l'agent collecteur qui ordonne le déplacement d'une façon aléatoire. Dans notre architecture, le rôle de l'agent collecteur sera d'informer le « décideur » local sur la charge globale du système.

4 Prise de décision

Après la collecte d'informations sur l'état du système, le processus de décision intervient afin d'effectuer le déplacement de tâches sur les noeuds ayant une faible charge. Il s'agit d'une double décision : la première consiste à choisir les tâches à déplacer et la deuxième concerne le choix de la localisation de ces tâches. La prise de décision peut se faire d'une façon centralisée ; dans ce cas le processus de déplacement utilise les informations collectées afin de choisir la réorganisation des tâches. Au contraire, la solution décentralisée utilise peu d'informations et dans ce cas le noeud chargé demande à ses voisins d'accepter la migration d'une tâche, ceux-ci pouvant accepter ou refuser la nouvelle tâche. Afin de ne pas se confronter à un système instable dans lequel les tâches font des allers-retours inutiles entre les noeuds, des seuils sont à définir pour la prise de décision ; dans ce cas on peut envisager des seuils dynamiques [Xu90] en fonction de l'évolution du système. Dans notre architecture, l'algorithme de prise de décision utilise des seuils dynamiques ; ces seuils seront modifiés en fonction de l'évolution de la charge locale de la machine ainsi que de la charge globale fournie par l'agent collecteur.

5 Pourquoi utiliser les agents mobiles ?

Le placement dynamique de tâches nécessite la migration de ces tâches entre les différentes machines du réseau. Cette migration devient opérationnelle grâce à l'utilisation de la technologie des agents mobiles. Dans notre approche, une tâche sera représentée par un agent. Ainsi, la terminaison d'une tâche correspond à la fin de la vie de l'agent. Les échanges d'informations entre les tâches correspondent à la communication entre les agents. Quant au déplacement d'une tâche, il correspond à la migration d'un agent à travers le réseau. Il s'agit ainsi de représenter une tâche par un agent mobile.

Lorsque l'on parle de la technologie d'agents mobiles, deux types de migration sont à considérer : (1) la migration forte qui permet à un agent de se déplacer quelque soit l'état d'exécution et de communication avec l'extérieur dans lequel il se trouve et de reprendre son exécution après la migration exactement là où elle en était avant ; (2) la migration faible ne fait que transférer avec l'agent son code et ses données. Elle nécessite des moments privilégiés dans le code de l'agent pour pouvoir être lancée (point d'arrêt) ; le programmeur doit donc explicitement préserver dans les données les informations d'état permettant la poursuite de l'exécution au point d'avancement atteint. Bien que la migration forte facilite la tâche du programmeur, nous avons opté (cf. chapitre 3) pour l'utilisation d'une plate-forme (JavAct) qui offre une migration faible

afin d'augmenter la performance de l'application distribuée ; dans cette approche la migration des agents n'est pas totalement transparente pour le programmeur.

La technologie d'agents mobiles a été largement utilisée dans le domaine de la programmation distribuée [Montresor02, Gomoluch01, Satoh04, Evripidou01]. Dans la suite de ce chapitre, nous proposons une architecture qui va permettre l'équilibrage de charge dans une application distribuée basée sur la technologie d'agents mobiles. Les agents mobiles seront utilisés à deux niveaux : agent de calcul et agent collecteur. Les agents de calcul représentent les tâches de l'application et ils auront la possibilité de se déplacer entre les machines. Un agent mobile collecteur sera utilisé pour visiter les différentes machines de l'application afin de construire une vision sur la charge globale du système. Avant de représenter l'architecture que nous avons proposée pour l'équilibrage de charge ainsi que les différents algorithmes utilisés, nous allons présenter le problème de l'agentification d'un algorithme séquentiel et les limites posées lors de la distribution.

6 Agentification et distribution d'un algorithme

Dans cette section, nous nous focalisons sur les algorithmes séquentiels par nature, où le calcul d'une étape du programme dépend fortement des calculs déjà effectués. Nous illustrons ici l'importance du choix de la granularité des calculs sur les performances de l'algorithme, une tâche sera considérée comme un agent. Dans ce cas là, on parle de l'agentification du code [Bernon00].

Dans [Authie94], l'auteur montre l'impact de la granularité sur les résultats obtenus lors de la distribution d'un algorithme de traitement d'image. D'une façon intuitive, le but recherché lors de la distribution d'un algorithme est d'utiliser (dans la limite du possible) au maximum les machines disponibles et ce d'une manière équitable. Cette hypothèse est valable si l'on néglige le surcoût lié à la communication ; en effet le coût dû à la communication peut dans certains cas détériorer le gain obtenu par la répartition. Dans ces travaux, l'auteur présente un algorithme dans lequel les processus ont tous, à chaque étape, le même volume de calcul à effectuer ; les expérimentations de cette version très bien équilibrée ont montré un facteur d'accélération de 0.5 avec 32 processeurs ! Ces résultats étonnants (32 processeurs vont moins vite qu'un seul) montrent que le surcoût dû à la communication peut détériorer les performances de l'algorithme.

L'auteur propose une autre solution afin de diminuer le coût global des communications en utilisant de plus longs messages. Bien que le nouvel algorithme présenté n'aboutisse pas à une charge équitable sur tous les processeurs, l'auteur obtient une accélération de 27 avec 32 processeurs, au lieu d'un ralentissement de 2 qu'il avait obtenu dans la première proposition. Ces travaux montrent qu'il ne faut pas négliger le surcoût lié à la communication lors de l'étude d'un algorithme distribué.

6.1 Limitation de la distribution

Les algorithmes parallèles sont-ils susceptibles d'exploiter les machines parallèles au maximum de leur puissance théorique ? Idéalement, on souhaite bien sûr aller p fois plus vite, avec l'utilisation de p machines. En contradiction avec ce but, Amdahl a énoncé dès 1967 un principe connu sous le nom de *Loi d'Amdahl* limitant fortement l'accélération, et donc l'extensibilité de nombreux algorithmes : soit un algorithme

s'exécutant en séquentiel en un temps T_1, décomposé en $f_s\ T_1 + f_p\ T_1$ avec f_s la fraction intrinsèquement séquentielle du programme, f_p la fraction parallélisée et $f_s+f_p=1$. Si l'algorithme est parfaitement parallèle, son exécution sur p processeurs prendra au mieux un temps de :

$$T_p = f_s T_1 + \frac{f_p T_1}{p}$$

Le facteur $S_p = T_1 / T_p$ d'accélération est alors égal à :

$$S_p = \frac{1}{f_s + \frac{(1-f_s)}{p}} \leq \frac{1}{f_s}$$

Autrement dit, même si seulement 1% du code est intrinsèquement séquentiel, l'accélération ne pourra jamais dépasser 100, rendant inutile l'utilisation de milliers de machines.

Contrairement aux bornes théoriques de la loi d'Amdahl, qui ne voient pas un avenir prometteur dans le calcul parallèle, Gustafson [Gustafson88] a défini une autre version de cette loi.

Soit T_p le temps d'exécution d'un programme parallèle sur p processeur et f_p la fraction du code parallélisable, alors le temps d'exécution sur un seul processeur est donné par :

$$T_1 = f_s T_p + p\ f_p T_p$$

Donc l'accélération S_p sera égale à :

$$S_p = f_s + (1-f_s)\ p$$

A partir de la loi de Gustafson, nous retrouvons une autre notion : l'extensibilité (scalability). Cette notion traduit le comportement d'un algorithme parallèle en fonction du nombre de processeurs. Nous pouvons donc soit augmenter le nombre de processeurs tout en préservant la même taille du problème, soit augmenter la taille du problème en fonction du nombre de processeurs. Avec cette notion, un système distribué ne sert pas seulement à résoudre plus rapidement un problème donné mais également à résoudre des problèmes de taille importante.

6.2 La communication entre les agents : push vs pull

Lors de la décomposition d'un algorithme sous forme d'agents, ces derniers auront besoin d'échanger des données afin d'accomplir la tâche demandée. Cette communication va avoir une grande influence sur la qualité de l'application ce qui nous invite à adopter un modèle de communication qui minimise la communication. Nous avons proposé de modéliser les tâches sous forme d'agents qui auront donc la possibilité de mémoriser des informations sur leurs voisins (agents avec qui ils vont communiquer). L'agent va donc modéliser son environnement. Comme nous allons le voir, une bonne modélisation permet de réduire les communications entre agents. On peut envisager deux modes de communication (voir Figure 4.1) entre agents :

Le mode pull : si l'agent a besoin de connaître des informations sur ses voisins, il leur envoie un message de demande d'informations ; à la réception de cette demande, l'agent voisin envoie les données demandées. L'inconvénient de cette méthode est que l'agent est obligé de demander systématiquement les informations à ses voisins, même si elles n'ont pas changées depuis la dernière demande, puisque l'agent ne connaît pas l'évolution des autres agents. En plus, l'obtention d'une information nécessite l'envoi de deux messages (demande/réponse) augmentant d'autant le trafic sur le réseau.

Le mode push : dans ce cas, l'agent va construire une représentation de l'état de ses voisins ; on parle d'une modélisation de leur état. Lorsqu'un agent *A* aura besoin d'une information sur un autre agent *B*, il va s'abonner auprès de lui. L'agent *B* sera donc invité à lui communiquer toutes modifications sur son état. L'agent *A* utilisera la dernière représentation qu'il a de l'agent *B*.

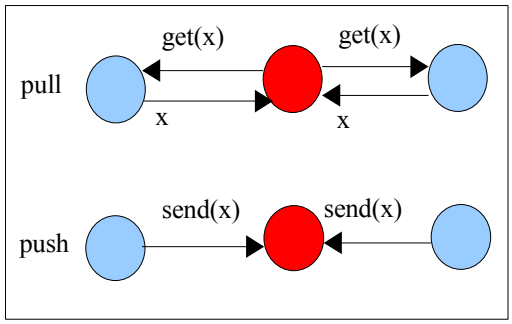

Figure 4.1 : Modèle de communication

Bien que la méthode push exige de l'agent une modélisation de ses voisins et que la méthode pull paraisse plus simple, nous avons choisi de communiquer par la méthode push. Cette méthode permet, dans le cas d'un système réparti sur plusieurs machines, de réduire les échanges entre les agents et donc de diminuer le trafic réseau. Un modèle hybride [Budau02] est envisageable, permettant à l'agent de basculer entre le deux modes de communication en fonction des caractéristiques de l'application.

7 Équilibrage de charge itératif par agent mobile

Notre algorithme se situe dans une grande famille d'algorithme, il s'agit des algorithmes dynamiques dont le but est d'obtenir la solution optimale d'une façon itérative [Cybenko89]. Nous avons opté pour le choix de ce type d'algorithmes du fait que les systèmes que l'on étudie sont imprévisibles et qu'une modélisation assez fine de l'évolution du système sera trop coûteuse voire presque impossible. Le système lui même évolue dans un monde incertain et les machines qui font tourner l'application que l'on cherche à optimiser peuvent avoir plusieurs applications qui fonctionnent en parallèle diminuant d'autant leurs performances.

Pour la collecte d'informations, l'architecture que nous proposons est une architecture hybride. Dans cette architecture, la prise de décision se fait en fonction de la charge locale de la machine. Elle est hybride parce qu'elle contient un agent collecteur qui se déplace sur les différentes machines afin de construire une vision globale du système. Arrivé sur une machine, l'agent collecteur l'informe de la charge

globale du système. Dans la section suivante, nous allons présenter l'architecture de notre système qui contient trois types d'agents.

7.1 Architecture

L'architecture (voir Figure 4.2) que nous proposons pour l'équilibrage de charge à partir d'un algorithme distribué, comporte trois types d'agents :

1. **Agent Moniteur (AM)** : un seul agent stationnaire responsable de la surveillance de l'activité de la machine où il se trouve. C'est lui qui s'occupe de la détection d'un surcharge de machine, il va ainsi proposer une migration de certaines tâches vers les machines voisines. Le voisinage est défini de façon à minimiser le trafic sur le réseau. L'agent moniteur possède deux seuils dynamiques qui vont lui permettre de déclencher le processus de déplacement que nous allons détailler par la suite.

2. **Agent d'Exécution Mobile (AEM)** : comme nous l'avons déjà mentionné, dans une application distribuée, les tâches sont modélisées par un agent mobile ; il va donc y avoir plusieurs agents AEM par machine. Ces agents migrent entre les machines dans le but d'équilibrer la charge du système. C'est l'agent moniteur (AM) qui ordonne la migration d'un agent AEM et ce n'est, en aucun cas, à l'agent AEM de décider de cette migration. En plus de la migration entre les machines, un agent AEM aura la possibilité de communiquer avec son environnement et avec les autres agents AEMs.

3. **Agent Collecteur Mobile (ACM)** : en général, un seul agent collecteur mobile est nécessaire par application mais il peut y avoir plusieurs dans le cas d'une application distribuée sur un grand réseau. Les agents moniteurs ont une vision locale du réseau et la prise de décision se fait en fonction d'une connaissance locale. En général, la construction d'une vision globale nécessite des échanges avec une machine centrale ce qui augmente le trafic du réseau. À l'inverse, l'agent collecteur mobile effectue des échanges en local et permet ainsi la réduction du trafic. Le rôle de l'agent collecteur sera détaillé plus tard dans ce chapitre.

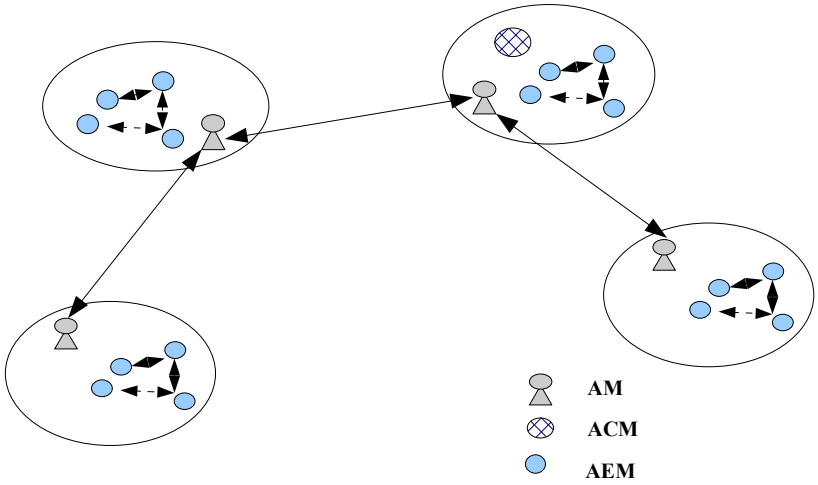

Figure 4.2 : Architecture du système

7.2 Agent moniteur et prise de décision

Comme nous l'avons cité auparavant, nous partons du principe que le système est bien équilibré, dans le cas contraire, c'est l'agent collecteur ACM qui l'aide pour attendre cet équilibre. Le but de notre algorithme est de ramener le système à son état d'équilibre après une perturbation. C'est le rôle de l'agent moniteur d'assurer cette tâche. L'agent moniteur possède deux seuils dynamiques s_{min} et s_{max} qui guident le déclenchement du processus de déplacement. Les deux seuils dépendent de la charge de la machine ainsi que de la charge moyenne engendrée par l'exécution d'un agent AEM. On définit par $\bar{C}$ la charge moyenne d'une machine et par c_{ag} la charge moyenne engendrée par l'exécution de l'agent sur une machine du système. La méthode qui permet de calculer c_{ag} est détaillée plus tard. Les deux seuils sont définis par :

$$s_{min} = \bar{C} - c_{ag}$$
$$s_{max} = \bar{C} + c_{ag}$$

Tant que la charge de la machine évolue entre les deux seuils, le processus de migration ne pourra être déclenché. Supposons, que la charge dépasse le seuil s_{min}, la machine est donc moins chargée et elle a la possibilité d'accepter une partie de la charge de ses voisins, ce qui déclenche le processus de migration. Dans le cas où la charge dépasse le seuil s_{max}, la machine étant donc surchargée, c'est le processus de migration qui permet de la décharger.

Le processus de migration des agents se fait via un contrat CNP (Contract Net Protocol) [Smith80]. Une étude détaillée des limites de ce protocole peut être trouvée dans [Aknine01] ainsi que la présentation des diverses extensions proposées dans la littérature. Dans ce protocole, l'agent AM lance un appel d'offre à tous les agents (AM) voisins. Les agents intéressés lui envoient une proposition en terme de leur charge et de leur disponibilité. L'agent moniteur évalue les propositions et choisit la meilleure offre. Il envoie un message de confirmation à l'AM choisi et le processus de migration est déclenché. Une fois le processus de migration terminé, l'agent AM ajuste les deux seuils en fonction de la nouvelle charge. Lorsque les voisins envoient des réponses négatives à l'appel d'offre, ces réponses impliquent que les machines voisines ont la même charge que la machine « demandeur » (les machines évoluent de la même manière). L'agent AM est alors invité à ajuster les deux seuils s_{min} et s_{max} et à adopter sa charge comme un nouvel état d'équilibre.

Le comportement de l'agent moniteur est décrit dans l'algorithme suivant :

Algorithme 4.1

Tant que le système est actif
  calculer la charge $\bar{C}$ *de la machine*
  Si $((\bar{C} \le s_{min}) \| (\bar{C} \ge s_{max}))$ **Alors**
    lancer un appel d'offre avec les voisins
    Attente *de la réception des propositions*
    choix de la meilleur proposition
    sélectionner les agents AEM à faire migrer
    ordonner la migration des agents
    actualiser les seuils s_{min} *et* s_{max}

fin Si
Fin Tant que

Nous pouvons remarquer que le déclenchement du processus de migration se fait localement sur chaque machine et ceci en fonction de sa charge. Si cette charge ne change pas, le processus n'est jamais déclenché. Ceci nous ramène à la situation suivante : Une machine qui ne possède pas la même charge que les autres, avec une évolution de sa charge locale ne dépassant pas les deux seuils. Dans ce cas, le processus de migration ne sera jamais déclenché et le système jamais équilibré. Afin de ne pas se retrouver dans une telle situation, l'agent mobile collecteur construit une vision globale du système. Cet agent se déplace entre les différentes machines. Dès qu'il arrive sur la machine non équilibrée, il informe l'agent moniteur sur la charge globale. Ainsi, ce dernier modifie les deux seuils, ce qui déclenche le processus de migration.

7.3 Agent mobile collecteur

Le rôle de l'agent mobile collecteur est d'assurer un équilibrage de la charge globale du système. Cet agent mobile se déplace entre les différentes machines de l'application. Dès qu'il arrive sur une machine, il mesure sa charge. La charge de la machine est comparée à la charge globale du système. Si l'agent collecteur détecte une différence entre les deux charges, il informe l'agent moniteur de la nouvelle charge globale (voir Algorithme 4.2). Lors de son déplacement, l'agent mobile construit une vision globale sur la charge moyenne des machines du réseau. Le choix de la construction d'une vision globale du système par l'utilisation d'un agent mobile est adopté afin de minimiser le trafic sur le réseau. L'utilisation d'un agent mobile pour la réduction du trafic est détaillée dans le chapitre 6.

La charge moyenne du système est définie par le nombre total des agents actifs sur le nombre total des machines de l'application. Le système peut évoluer assez vite et l'agent collecteur n'ayant pas la possibilité de connaître cette valeur précisément. Il va la calculer d'une façon approximative. L'agent collecteur se déplace entre les différentes machines et la valeur moyenne est calculée de la manière suivante :

- L'agent mobile compte le nombre de machines visitées, soit *nb* ce nombre. Après chaque migration l'agent incrémente *nb* de *+1* même s'il arrive sur une machine déjà visitée, car entre la première et la seconde visite d'une machine donnée, sa charge évolue et la valeur moyenne est ajustée en fonction de la nouvelle charge.
- À l'arrivée sur une machine, il calcule la charge *C* de la machine et ajuste la charge moyenne globale du système C_{moy} :

$$C_{moy} = \frac{C_{moy} \cdot nb + C}{nb + 1}$$

Pour le déplacement de l'agent collecteur nous pouvons envisager deux modes de déplacement : un déplacement aléatoire ou un déplacement cyclique. Dans un déplacement aléatoire, l'itinéraire que l'agent suivra n'est pas fixé à l'avance et l'agent choisit le lieu de sa migration d'une façon aléatoire ; le mode aléatoire est bien adapté dans le cas d'un réseau dynamique. Dans le mode cyclique l'agent doit connaître les

machines de l'application et il les visite les unes après les autres d'une façon cyclique. Dans ce cas, l'itinéraire est connu par l'agent collecteur.

Algorithme 4.2

Initialisation : $nb = 0$; $C_{moy} = 0$
Tant que le système est actif
  migrer vers une nouvelle machine
    $C_{moy} = (C_{moy} \cdot nb + C) / (nb + 1)$ *// C : charge de la nouvelle machine*
  $nb = nb+1$
  Si *(nb > K)* **Alors** *// l'agent aura visité plus que K machines*
    Si $(|C - C_{moy}| \geq \Delta)$ **Alors** *// la différence entre les deux charges*
    informer l'agent moniteur de la nouvelle charge C_{moy}
    Fin si
  Fin si
Fin tant que

Arrivé sur une machine, l'agent collecteur compare la charge C de la machine avec la charge moyenne C_{moy} du système ; si une différence entre les deux charges est détectée, il informe l'agent moniteur qui pourra dans ce cas déclencher le processus de migration afin d'ajuster le système. Afin de construire une charge globale assez proche de la réalité, l'agent collecteur doit observer un certain temps (K machines visitées) avant de passer à l'action (informer les agents moniteurs). Pour un système qui n'évolue pas trop vite, nous pouvons dire que plus le nombre de machines à visiter est grand plus la vision sur la charge globale du système est proche de la réalité. Il faut rajouter que l'agent collecteur doit être le plus rapide possible ; ce qui nous invite à réduire sa taille. Pour ce faire, il faut réduire la taille de son code (comportement de l'agent) ainsi que la quantité d'informations transportées avec lui. Nous avons choisi de ne transporter que deux valeurs avec l'agent collecteur : le nombre de machines visitées et la charge moyenne du système.

Dans notre architecture, nous avons proposé d'utiliser un seul agent collecteur mais nous pouvons toujours envisager de multiplier le nombre de ces agents et ceci en fonction de la taille du réseau (nombre de machine participante à l'application). Dans ce cas, il faut prévoir un mécanisme de coopération entre les différents agents collecteurs afin de construire une vision globale sur la charge du système.

Dans un réseau dynamique où les liaisons peuvent apparaître et disparaître, un agent collecteur peut se perdre. Dans ce type de réseau, il faut envisager un mécanisme de recréation de l'agent collecteur. Cette recréation est déléguée à un agent moniteur fiable qui sera chargé de détecter l'absence de l'agent collecteur et qui va créer un nouvel agent sur le réseau.

7.4 Réseau hétérogène et normalisation

Nous travaillons dans un environnement ouvert et hétérogène et les machines utilisées par l'application n'ont pas toutes la même capacité de calcul, ce qui pose une difficulté lors de l'utilisation du protocole CNP. Galtier [Galtier00] a travaillé sur ce

problème dans le domaine des agents mobiles. Elle a proposé une méthode pour modéliser les besoins d'une application active en temps de processeur sous une forme qui pourra être interprétée par n'importe quelle machine d'un réseau hétérogène. Un noeud de référence est choisi et l'agent y sera exécuté plusieurs fois afin de définir ses besoins en temps de calcul. Dans un réseau hétérogène un programme de calibration [CPU_bench] est exécuté sur les différentes machines ce qui permet de les comparer à la machine de référence. Le système de calibration permet ainsi d'exprimer les besoins d'un agent et ceci pour les différentes machines du système.

8 Conclusion

Le réseau Internet forme une gigantesque plate-forme qui contient un grand nombre d'unités de calculs et de stockages et sa bonne utilisation permet d'augmenter la puissance des ordinateurs. Ce chapitre aborde le problème de la distribution d'un algorithme sur des stations de travail reliées par un réseau de connexion (IP ou autre). Une application distribuée est formée par des entités réparties sur les machines du réseau, ces entités sont souvent dépendantes les unes des autres ce qui nécessite la communication entre elles. Une application distribuée est donc décomposée en étapes de calculs et en étapes de communications. Le but ultime de la distribution d'un algorithme est d'accélérer sa vitesse d'exécution ; nous aurons constamment à vérifier deux exigences souvent contradictoires : minimiser les communications et équilibrer au mieux la charge de travail des machines disponibles.

Dans ce chapitre, nous avons proposé une architecture qui permet de simplifier l'équilibrage de charge dans une application distribuée. Dans cette architecture, les différentes entités de l'application sont considérées comme des agents mobiles. L'équilibrage de charge est dynamique et l'algorithme utilisé est un algorithme itératif où l'équilibrage se fait de proche en proche. L'architecture proposée est complètement décentralisée et la décision de migration s'effectue en fonction de la charge locale de la machine. Un agent collecteur mobile se déplace sur les différentes machines de l'application et il construit une vision globale sur la charge moyenne du système. Cette vision globale est utilisée pour guider la prise de décision locale et pour définir les seuils utilisés lors du processus de migration.

Dans le chapitre suivant, nous allons présenter la distribution d'une application réelle pour mettre en évidence l'apport de notre architecture sur l'équilibrage de charge de cette application.

Chapitre 5

Une première expérience : application de la mobilité à l'équilibrage de charge

1 Introduction

Dans ce chapitre, nous nous intéressons à la mise en place concrète d'un algorithme d'équilibrage de charge dans une application réelle. L'application choisie représente une simulation de l'évolution d'une colonne à distiller. L'intérêt d'une telle simulation est de l'utiliser pour le contrôle d'une colonne réelle ce qui nécessite l'utilisation d'un système rapide. L'industrie d'aujourd'hui fait face à un obstacle majeur : il s'agit d'un manque d'instruments de mesures en ligne, ce manque est dû à des contraintes liées à la faisabilité des capteurs et au coût de leur fabrication.

En effet, il peut s'avérer très difficile de mesurer en ligne certaines variables et lorsque la mesure est techniquement réalisable, c'est la contrainte de coût qui intervient. Dans le domaine de l'industrie chimique, le procédé de distillation est l'une des opérations unitaires les plus intéressantes à considérer. Pour appliquer les techniques avancées de l'automatique (contrôle, supervision, diagnostic) à la colonne à distiller, il faut connaître les variables d'états (concentration, composition de l'alimentation, débits) du procédé. Il devient donc nécessaire de reconstruire ces variables d'une manière non coûteuse et ce à partir des connaissances disponibles, à savoir le modèle d'évolution et les mesures des entrées/sorties du procédé. Un observateur (capteur logiciel) sera utilisé pour estimer l'état de la colonne.

Un observateur est un système dynamique qui utilise l'entrée et la sortie du système observé comme des entrées et fournit comme sortie une estimation sur les états du système. Pour un système donné, la réalisation d'un observateur est liée à la propriété d'observabilité. Si un système est observable, alors on peut estimer ses états à partir des entrées et sorties mesurées et ceci en un temps fini. La distribution de l'algorithme permet d'augmenter la fiabilité de l'observateur en minimisant le temps de réponse.

Nous présenterons plus particulièrement dans ce chapitre, les différents problèmes pouvant intervenir lors de la modélisation et la mise en place d'une application réelle. Après une modélisation sous forme d'un SMA de la colonne à distiller, nous allons montrer le gain obtenu lors de la distribution de la résolution. Les différentes entités de l'application ne vont pas évoluer de la même manière, ainsi certaines machines vont avoir plus de charge que les autres ce qui prouve que l'application choisie est une bonne

candidate pour tester notre algorithme d'équilibrage de charge dynamique. Nous présentons les différents problèmes pouvant intervenir lors de la mise en place de l'équilibrage de charge.

2 La colonne à distiller

La colonne à distiller (voir Figure 5.1) est un dispositif physique composé d'une colonne à plateaux [Targui00]. Cette colonne est utilisée pour séparer un mélange de deux liquides. L'unité de distillation est donnée dans la figure suivante. Elle est constituée par n plateaux réels numérotés du bouilleur $i = 1$ vers le condenseur $i = n$. Le plateau d'alimentation correspond à $i = f$.

avec :

f : le plateau d'alimentation

F : le débit d'alimentation

Ni : la rétention molaire dans le plateau i

Li : le débit du liquide sortant du $i^{ème}$ plateau

Vi : le débit de la vapeur sortant du $i^{ème}$ plateau

D : le débit du distillat (sortie de la colonne)

Z_f : la composition de l'alimentation

Xi, Yi : les compositions « liquide et vapeur » dans le $i^{ème}$ plateau

Qb : la puissance de chauffe

$Qcond$: la puissance dégagée par le condenseur

α : la volatilité relative

i : l'indice du plateau

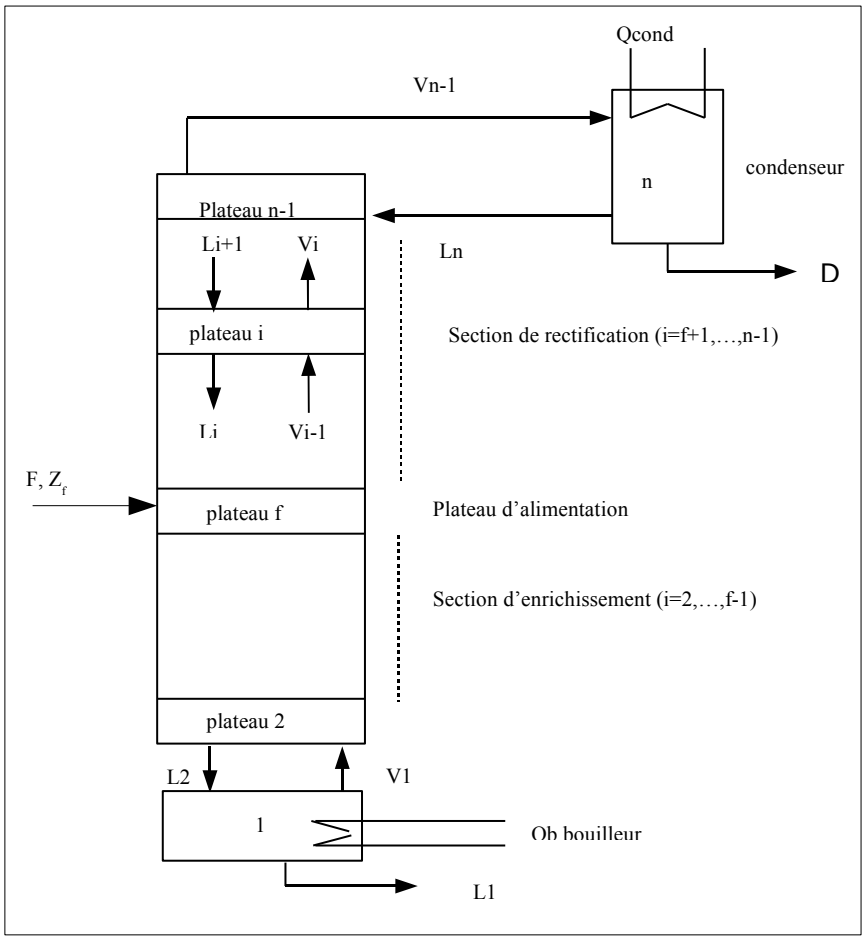

Figure 5.1 : La colonne à distiller

Bouilleur : le bouilleur (voir Figure 5.2) est la source principale de la vapeur dans la colonne à distiller. Le chauffage est assuré par une source de puissance Qb. Le soutirage est supposé par débordement.

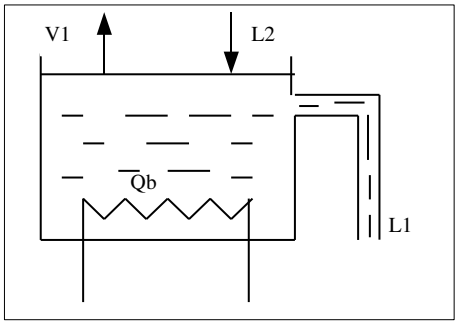

Figure 5.2 : Bouilleur

Condenseur : il permet de transformer toute la vapeur provenant de la colonne en un liquide, on parlera alors d'un condenseur total. Une partie du liquide condensé est injectée à l'intérieur de la colonne et on parlera alors du débit de reflux, l'autre partie s'échappe à l'extérieur, c'est le distillat.

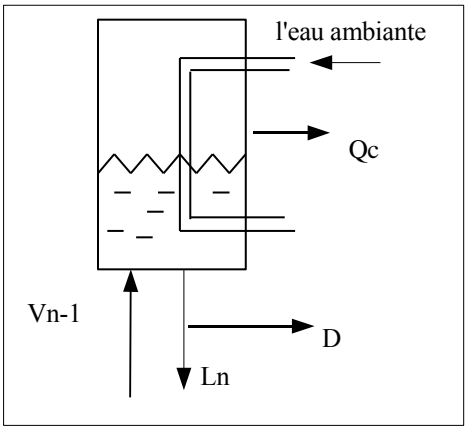

Figure 5.3 : Condenseur

Plateau réel : un plateau réel d'une colonne à distiller est constitué d'un barrage qui permet un séjour permanent du liquide et d'un ensemble de trous par lesquels la vapeur pénètre et traverse la colonne de liquide sous forme de bulles. Le liquide retenu dans un plateau s'appelle la rétention liquide. On peut aussi parler de rétention vapeur mais celle-ci est généralement négligeable.

3 Modèle dynamique de la colonne à distiller

Le modèle dynamique d'une colonne à distiller à plateaux est un ensemble d'équations différentielles déduites des bilans de matière et d'énergie sur chaque plateau, d'équations algébriques et d'un modèle thermodynamique pour prédire les propriétés physiques et thermodynamiques du mélange à distiller.

Dans cette section, nous rappelons le modèle le plus simple pour une colonne à distiller et qui a été retenu pour notre application. C'est le modèle de LEWIS dit à rétention constante.

Les hypothèses de LEWIS sont les suivantes :

- La phase liquide est une solution idéale.
- Les enthalpies molaires de vaporisation de chacune des constitutions sont égales à la chaleur latente de vaporisation.
- Les enthalpies molaires de vaporisation et les capacités calorifiques molaires ne varient pas avec la température.

Dans le cas d'une distillation de LEWIS, la relation la plus commode pour relier les concentrations dans la vapeur et le liquide à l'équilibre, utilise la volatilité relative, α, supposée constante. Elle est donnée comme suit :

$$y_i = \frac{\alpha x_i}{1+(\alpha-1)x_i} \quad , \alpha > 1$$

Les hypothèses de LEWIS nous permettent de déduire les équations différentielles suivantes :

$-\ Bouilleur\,(i=1)$

$$N_1\left(\frac{dX_1}{dt}\right)=(F+L)(X_2-X_1)+V(X_1-Y_1)$$

$-\ Section\ de\ rectification\ (i=2,\ldots f-1)$

$$N_i\left(\frac{dX_i}{dt}\right)=(F+L)(X_{i+1}-X_i)+V(Y_{i-1}-Y_i)$$

$-\ Plateau\ d'\ alimentation\ (i=f)$

$$N_f\left(\frac{dX_f}{dt}\right)=(F+L)(X_{i+1}-X_i)+V(Y_{i-1}-Y_i)+F(Z_f-X_f)$$

$-\ Zone\ d'\ enrichissement\ (i=f+1,\ n-1)$

$$N_i\left(\frac{dX_i}{dt}\right)=L(X_{i+1}-X_i)+V(Y_{i-1}-Y_i)$$

$-\ Condenseur\ (i=n)$

$$N_i\left(\frac{dX_i}{dt}\right)=V(Y_{i-1}-Y_i)$$

Ce système d'équations différentielles montre la dépendance entre les différents plateaux de la colonne. Nous constatons que pour calculer l'instant *k+1* d'un plateau, il nous faut sa concentration à l'instant *k*, ainsi que la concentration à l'instant *k* des deux plateaux au dessus et en dessous de ce dernier. Les équations différentielles montrent que dans la colonne à distiller, l'état d'un plateau dépend de l'état de ces deux voisins. Comme nous le verrons plus loin, cette propriété de localité est importante pour une approche SMA du problème.

4 Résolution par l'algorithme d'Euler

L'algorithme itératif d'Euler permet la résolution numérique de notre système d'équations différentielles. L'idée de base de l'algorithme d'Euler est d'approximer la courbe solution par sa tangente.

Les formules de récurrence ci-dessous permettent de construire la suite des couples $(x_i, y_i)_i$:

$$x_{i+1}=x_i+h$$
$$y_{i+1}=y_i+h\,f(x_i,y_i)$$

Il faut bien sûr ne pas oublier l'initialisation de l'algorithme par la condition initiale :

$$y_0=f(x_{0,}\,y_0)$$

D'autres algorithmes (Taylor, Runge-Kutta, etc ...) plus efficaces permettent de résoudre ce système, mais nous avons choisi l'algorithme d'Euler pour sa simplicité. Le choix de l'algorithme n'est pas très important pour notre étude, puisque l'essentiel consiste à présenter l'impact de la distribution et de la mobilité des agents sur le temps de calcul.

5 Agentification de la colonne à distiller

La communication dans une application répartie va trop influencer ces performances, afin de minimiser cette communication pour la colonne à distiller, nous avons décidé de considérer chaque plateau comme une tâche (agent). Ainsi, chaque plateau de la colonne représente un agent autonome, son rôle est de calculer la concentration du plateau qui le représente ; pour-cela, il lui faut la concentration de ses deux voisins (voir Figure 5.4).

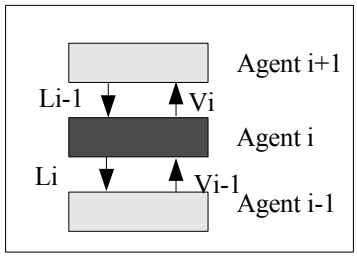

Figure 5.4 : Échange entre plateaux

Lorsque la concentration de ses deux voisins ne change pas, le plateau aura la possibilité de calculer sa concentration en tenant compte des anciennes concentrations. Pour se faire, un agent a besoin de modéliser les états des voisins et de la mémoriser. Une bonne modélisation permet de réduire la communication de l'agent. Ainsi deux modes de communications peuvent être adoptées :

Le mode pull : si l'agent a besoin de connaître les concentrations de ses voisins, il leur envoie un message leur demandant de communiquer leurs concentrations ; à la réception de la demande, l'agent voisin donne sa concentration. Une fois les deux réponses reçues, l'agent peut effectuer ses calculs. L'inconvénient de cette méthode c'est que l'agent est obligé de demander la concentration de ses voisins même lorsque cette concentration ne change pas dans le temps, ce qui ajoute un surcoût de calcul ; l'autre inconvénient du mode pull est le nombre de messages échangés entre les agents, introduisant une augmentation du trafic réseau dans le cas d'un système distribué (deux messages - demande/réponse - pour chaque concentration).

Le mode push : l'agent a la possibilité de mémoriser la concentration de ses deux voisins ainsi que leur état. On peut considérer deux cas possibles pour un agent : soit l'agent est stable et sa concentration ne change pas avec le temps, soit il est instable et sa concentration change avec l'évolution du système. Les voisins d'un agent stable peuvent utiliser la concentration déjà communiquée pour effectuer leurs calculs. Pour les informations à communiquer entre les agents, deux types de messages peuvent apparaître : si l'agent est stable, sa concentration n'a pas changé, il va notifier ses voisins de son nouvel état de stabilité. Dans le cas d'une nouvelle concentration du plateau, l'agent communique cette concentration à ses voisins.

Nous avons choisi d'utiliser le mode push pour la communication entre les agents car il permet de minimiser la quantité d'informations échangées. En plus, le mode push permet d'augmenter l'indépendance entre les agents, ainsi chaque agent a la possibilité d'évoluer seul ce qui permet de se rapprocher de la programmation asynchrone.

6 Étude du comportement de la colonne

6.1 La plate-forme oRis

Dans ce paragraphe nous examinons l'évolution des plateaux dans une colonne à travers une simulation en utilisant la plate-forme oRis [Harrouet02]. Il s'agit d'un environnement de simulation interactif : il est à la fois un langage de programmation par objets concurrents et un environnement d'exécution. Ses caractéristiques fournissent une plate-forme généraliste pour l'implémentation de systèmes multi-agents (SMA), plus particulièrement dédiée à la simulation. Il offre un langage orienté objets à typage fort et interprété, qui permet d'intervenir en cours de simulation pour observer le SMA. Il permet d'interagir avec les agents et de modifier leur comportement en temps réel. Il permet une représentation graphique (2D ou 3D) de ces agents, ainsi les agents dans oRis auront une représentation spatio-temporel. oRis offre une solution homogène pour les interactions, qu'elles soient implémentées par appel de méthode, lien réflexe, ou envoi de message (point-à-point ou par diffusion, avec traitement synchrone ou asynchrone). Le langage oRis a d'ailleurs de nombreuses similitudes avec les langages C++ et Java ce qui facilite son apprentissage et la réutilisation du code lors du passage à une plate-forme distribuée comme JavAct. Les détails de la plate-forme oRis se trouvent dans la thèse de Fabrice Harrouet [Harrouet00] ainsi que sur sa page personnelle [ORIS].

Dans nos travaux, nous avons utilisé la plate-forme oRis pour simuler la colonne à distiller afin de mieux comprendre l'évolution du système dans le temps. oRis permet de développer des agents autonomes (au niveau de leur comportement). Ces agents communiquent en utilisant des messages à travers la plate-forme. Ce modèle de communication par message permet de mieux comprendre les interactions entre les agents qui constituent notre système. Pour la colonne à distiller, nous avons envisagé deux types d'agents : les agents "plateau" et les agents "colonne".

5. Les agents "plateau" : un agent plateau représente un plateau de la colonne à distiller. Son rôle est d'échanger avec les autres agents (plateaux voisins) afin de présenter l'évolution du système dans le temps. Un agent plateau utilise les équations différentielles (conformes aux hypothèses de Lewis) pour calculer sa nouvelle concentration.

6. Les agents "colonne" : un tel agent représente la colonne ainsi que le changement des paramètres du système, comme par exemple le débit F d'alimentation ou sa composition Z_f. De cette façon l'utilisateur peut interagir avec cet agent afin de changer la composition de l'alimentation.

Après avoir défini les différents types d'agents, nous allons passer à l'implantation de nos agents dans oRis. Le temps dans oRis est découpé en cycles d'horloges logiques. Ainsi, à chaque cycle, un agent plateau calcule sa nouvelle concentration et diffuse cette concentration à ses voisins. Les outils de représentation graphique dans oRis permettent d'observer l'évolution du système. Un agent est représenté par un rectangle avec sa concentration du liquide en gris. Dans le cas d'un agent non-stable (sa concentration évolue) le plateau se colore en blanc. Lorsque l'agent se stabilise sa couleur devient noir.

La figure 5.5 présente une colonne formée de 12 plateaux avec l'alimentation sur le plateau n°5. Nous avons commencé la simulation avec une concentration xi=0.75

(i=1,..,12).

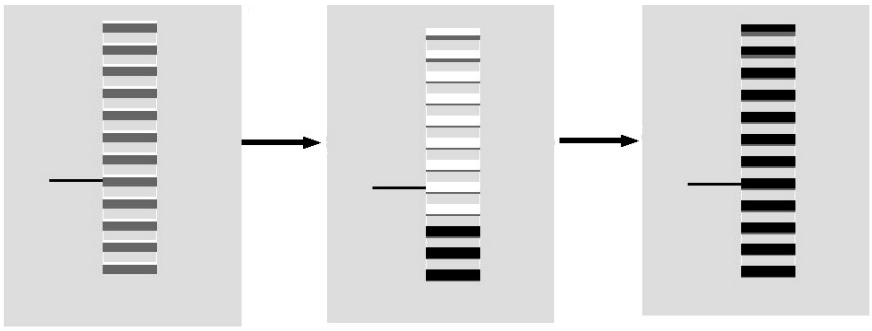

Figure 5.5 : Évolution de la colonne

Les courbes suivantes permettent de présenter l'évolution des concentrations en fonction du temps (voir Figure 5.6). Le temps représente le nombre de cycles dans oRis.

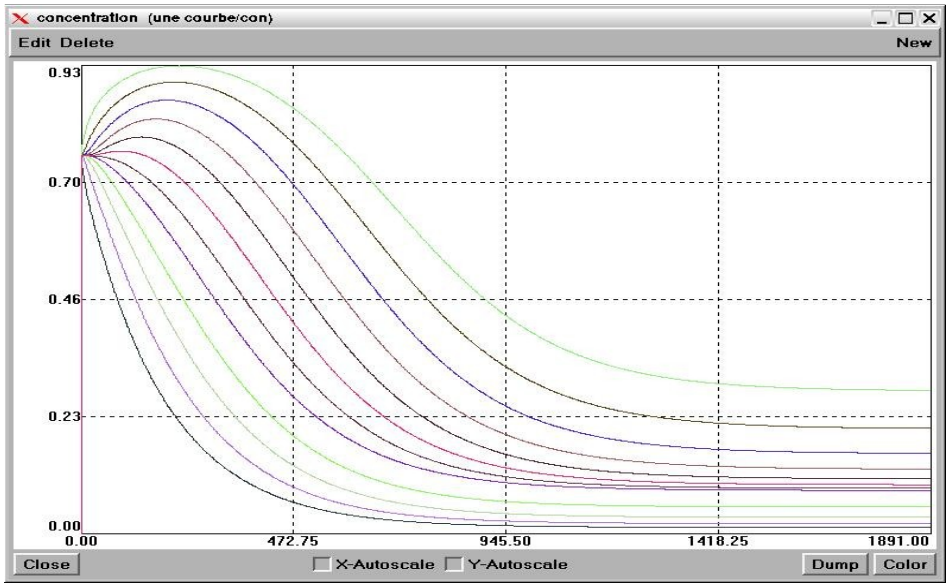

Figure 5.6 : Évolution de la concentration des plateaux

Une première observation sur l'évolution des concentrations permet de déduire que la stabilité des plateaux se propage de bas en haut, c'est-à-dire que le plateau bouilleur se stabilise en premier et c'est le plateau condenseur qui se stabilise en dernier. Cette propriété intéressante du système est obtenue pour une concentration initiale de 0,75. Afin de vérifier cette propriété, nous avons recommencé la simulation en changeant la concentration initiale des plateaux ; nous avons remarqué que c'est le même phénomène qui se reproduit. Cette propriété intrinsèque sera expliquée par le fait que la nouvelle concentration d'un plateau dépend de l'ancienne concentration ainsi que de celles de ses deux voisins. Ainsi, lorsque la concentration du bouilleur se stabilise, la

concentration de son voisin (2ème plateau) devient dépendante d'un seul plateau (3ème plateau). Ce qui explique qu'au bout d'un certain temps, la concentration du deuxième plateau se stabilise et ainsi de suite jusqu'au dernier plateau (condenseur). On dit que la stabilité se propage entre les plateaux de bas en haut.

Dans le système réel, la concentration Z_f change en fonction de la qualité du mélange que l'on cherche à séparer. Pour étudier l'influence de Z_f sur le système, nous avons ajouté la possibilité de changer cette concentration. On voit sur le schéma (voir Figure 5.7) l'influence de Z_f sur l'évolution du système.

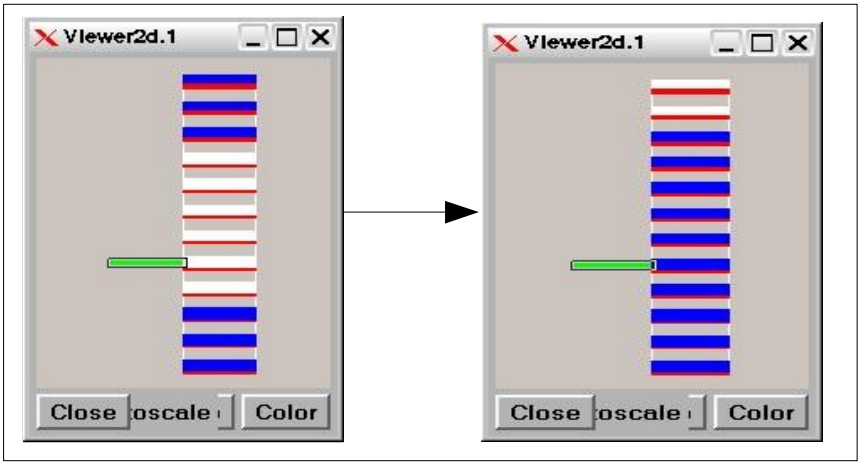

Figure 5.7 : Influence du paramètre Zf sur la colonne

Les courbes suivantes représentent l'évolution des concentrations dans le temps (voir Figure 5.8). Le temps est équivalent au nombre de cycles dans oRis.

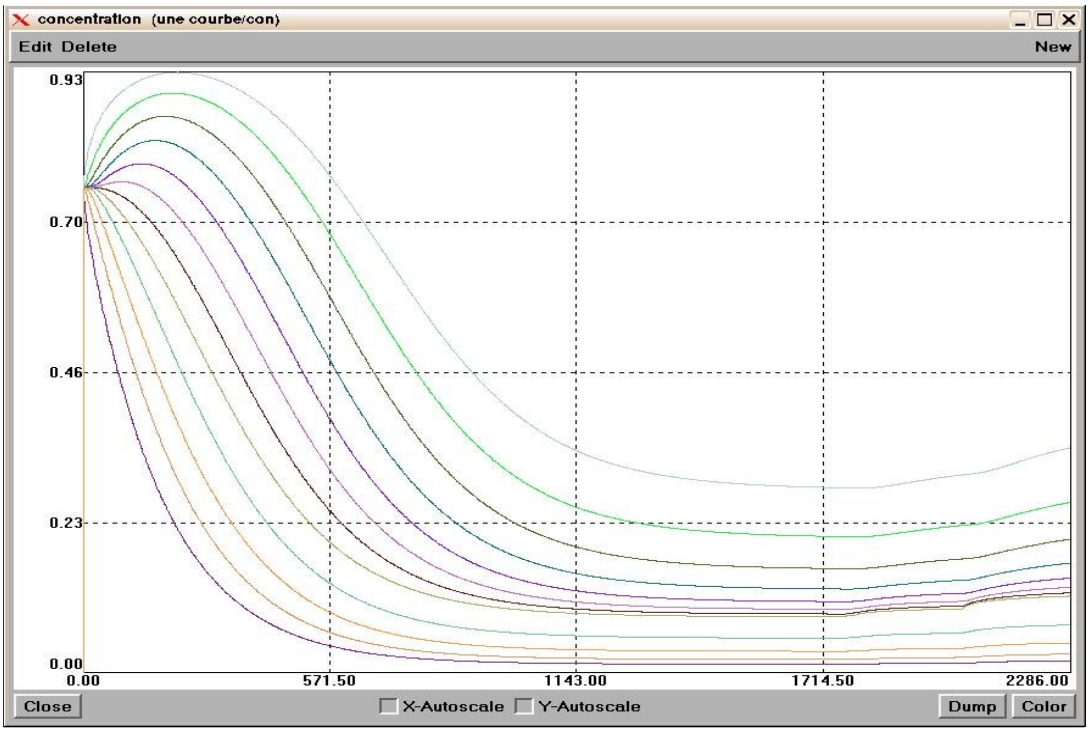

Figure 5.8 : Évolution de la concentration des plateaux

Le changement de Z_f entraîne des modifications de la concentration du plateau d'alimentation le rendant instable. Cette instabilité se propage de voisins en voisins à travers les plateaux. Certains plateaux sont beaucoup plus influencés par ce changement que les autres. En outre, le temps de retour à la stabilité n'est pas le même pour les différents plateaux.

6.2 Résultats de la simulation

Cette simulation permet de déduire des règles sur la stabilité des différentes tâches et sur les échanges entre elles. La première règle établit le fait que chaque plateau ne communique qu'avec ses deux voisins. Nous tenons compte de ce voisinage lors de la répartition de l'application, dans le but de réduire les communications distantes. Ainsi, dans la limite du possible, nous plaçons les plateaux voisins sur la même machine pour augmenter le nombre de communications locales.

Le comportement des plateaux, du point de vue de la stabilité, diverge d'un plateau à l'autre. Un agent stable n'a pas besoin de calculer sa concentration : il est en état de repos et il ne va donc pas consommer de ressources CPU. Une machine qui fait tourner cet agent sera moins chargée et aura donc la possibilité de décharger les autres machines de l'application. Lors de la distribution de la colonne, nous adoptons notre architecture d'équilibrage de charge. Ainsi le placement des agents sur les différentes machines se fera d'une façon dynamique.

7 Distribution de la colonne à distiller

Le temps d'exécution d'un programme sur une plate-forme distribuée est fonction du temps de calcul des tâches constitutives du programme, mais également du temps de communications entre les tâches. Ainsi, les performances d'un système dépendent

directement de l'équilibre entre ces deux grandeurs. Dans la réalité, les simulations étant utilisées pour le contrôle de la colonne à distiller, elles doivent absolument être rapides. Dans l'industrie, il existe des colonnes à distiller constituées de centaines de plateaux. Dans ce cas, l'utilisation d'un simulateur comme oRis pour le contrôle devient difficile à cause du nombre important de tâches à calculer sur la même machine. La distribution du problème sur plusieurs possesseurs permet de réduire ce temps de calcul.

Dans oRis, les plateaux sont des agents autonomes qui communiquent entre eux pour effectuer la résolution des équations différentielles. Les échanges entre les agents se font par envoi de messages locaux. Lors de la distribution effective du système, nous utiliserons les mêmes agents que dans la simulation oRis, à savoir les agents "plateau". La seule différence avec la simulation provient du fait que les agents se situent sur plusieurs machines et que les messages sont échangés à travers le réseau IP. Pour minimiser l'utilisation du réseau, nos agents interagissent par la méthode push (diffusion uniquement lorsque la concentration change). Les agents ne se stabilisent pas tous en même temps et certains d'entre eux deviennent inactifs. D'où le choix d'une plate-forme qui permet la mobilité des agents afin de mieux répartir dans le temps.

Dans la suite, nous examinons le mécanisme du placement statique qui oppose deux grandeurs antinomiques, à savoir minimiser la communication et optimiser l'équilibrage de charge, alors que le placement dynamique tire partie de ces deux grandeurs.

7.1 Placement statique

Une application répartie est d'autant plus performante, qu'elle utilise au mieux les ressources disponibles. L'objectif principal de la distribution étant de minimiser le temps d'exécution des processus, nous devons constamment vérifier deux exigences souvent contradictoires : minimiser la communication et équilibrer au mieux la charge de travail des machines disponibles. Les plateaux de la colonne à distiller peuvent fonctionner d'une manière autonome et la distribution nous paraît évidente dans ce cas. Ainsi, pour une colonne à *n* plateaux, le problème est divisé en *n* tâches (une tâche = un plateau) et chaque plateau est placé sur une machine. Si le nombre des machines est inférieur au nombre des plateaux, les plateaux seront placés d'une manière équitable (en fonction de la puissance de la machine) sur les différentes machines.

Deux placements statiques sont à envisager dans le cas de la colonne à distiller : le premier minimise la communication alors que le deuxième se base sur une charge équitable des différentes machines.

7.1.1 Réduction de la communication

Le but de cette stratégie est de réduire les communications distantes entre les agents. Chaque plateau communique avec ses deux voisins (au dessous et en dessus). Nous pouvons placer les plateaux d'une façon ordonnée (voir Figure 5.9). Dans ce cas la communication entre deux machines se réduit à la communication entre deux agents. Les plateaux qui se trouvent sur la même machine communiquent en local, alors que pour deux plateaux qui se trouvent sur deux machines distantes le réseau est utilisé pour la communication des messages.

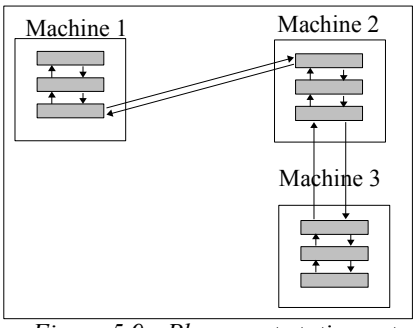

Figure 5.9 : Placement statique et réduction de la communication distante

7.1.2 Charge équitable

Nous avons remarqué que nos plateaux n'évoluent pas tous de la même manière. Ainsi, les plateaux qui se trouvent en bas de la colonne se stabilisent en premier et ils deviennent rapidement inactifs. Dans la première stratégie, la première machine, c'est-à-dire la machine qui contient les premiers plateaux devient inactive au bout d'un certain temps. Les machines ne fonctionnent pas de façon équitable. Ceci nous invite à proposer une nouvelle stratégie (voir Figure 5.10). Ainsi, le premier plateau est placé sur la première machine, le deuxième sur la deuxième et ainsi de suite de façon cyclique (modulo). Dans cette stratégie, le nombre de tâches par machine diminue avec l'évolution du système. Cette stratégie ne respecte pas le voisinage entre les plateaux et elle introduit plus de communications distantes entre les plateaux, augmentant d'autant la communication et l'encombrement du réseau.

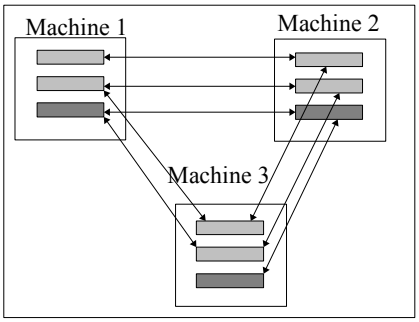

Figure 5.10 : Placement statique et équilibrage de charge

7.1.3 Résultats

Une première intuition permet de dire que la deuxième distribution paraît plus adéquate parce qu'elle permet d'avoir une charge presque équitable sur les différentes machines. La mise en place sur une plate-forme réelle, montre que cette stratégie est trop dégradée par la communication. Nous avons pris le cas d'une colonne à 12 plateaux placés sur trois machines. Nous avons obtenu une différence remarquable entre le temps d'exécution dans le cas d'un placement qui minimise le coût de communications (trois communications distantes) et un placement qui respecte une charge équitable (onze communications distantes).

Cette différence est de 47 % en faveur du placement qui respecte la réduction du

trafic. Ceci montre bien que la charge liée à la communication influence beaucoup le temps d'exécution de l'application.

7.2 Gains liés à la distribution statique

Afin de montrer le gain de la distribution de ce problème, nous prenons l'exemple d'une colonne à 8 plateaux distribués sur 1, 2, 4 et 8 machines. Ces machines sont connectées par un réseau local de 10Mbs. Nous obtenons les résultats suivants présentés dans la Figure 5.11.

La courbe montre que plus l'on distribue les plateaux plus l'on gagne en temps d'exécution de l'application. Dans ce type d'application, le surcoût engendré par la communication distante est nettement inférieur au gain obtenu en distribuant le système, sachant que le plateau représente l'élément de base pour la distribution.

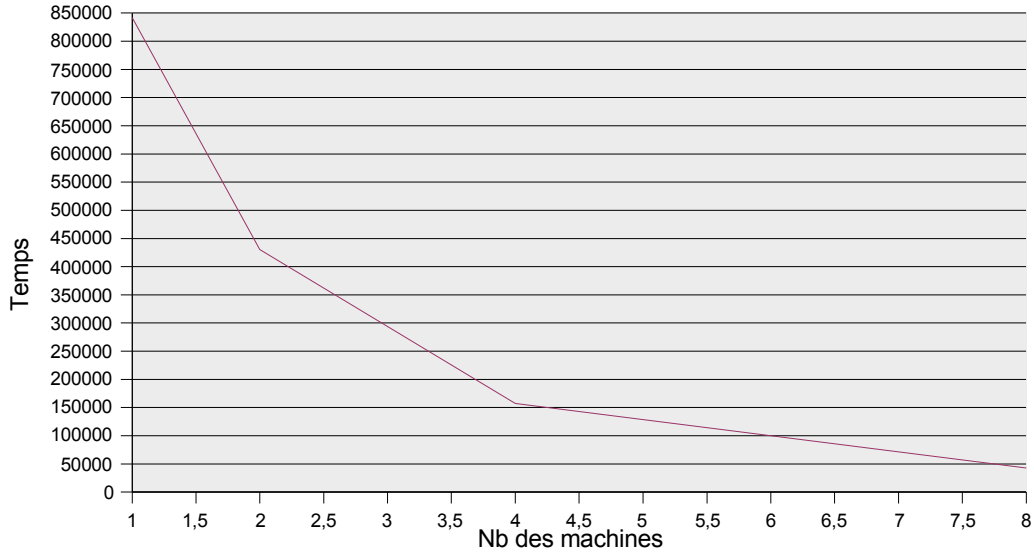

Figure 5.11 : Effet du nombre de machines sur le temps de la stabilité

Bien que la distribution de ce problème permette de diminuer son temps d'exécution, il ne faut pas oublier que les agents sont fortement couplés, ce qui veut dire que si une machine tombe en panne le système s'arrête. Ce problème se résout par l'utilisation de la technologie à agents mobiles, qui permet de migrer les agents d'une machine vers une autre avant que la panne n'intervienne.

8 Placement dynamique et équilibrage de charge

Un placement statique de la colonne à distiller a besoin d'une connaissance assez fine sur l'application distribuée. En plus, dans un placement statique nous sommes confrontés à deux exigences souvent contradictoires : minimiser la communication et équilibrer au mieux la charge de travail des machines disponibles. A l'inverse d'un placement statique, un placement dynamique permet de minimiser la communication distante tout en respectant une distribution équitable sur les différentes machines.

La plate-forme utilisée pour effectuer les tests est JavAct, et la mise en place du

processus d'équilibrage respecte l'architecture proposée dans le chapitre précédent. Un agent d'exécution (AEM) représente un plateau de la colonne. Le rôle de cet agent est de calculer la concentration du plateau et d'échanger des messages avec les autres agents afin de les informer de la nouvelle concentration. Sur chaque machine de l'application un agent moniteur (AM) sera placé, il va surveiller la charge de la machine et négocier la migration des agents vers les autres machines. Un seul agent collecteur (ACM) sera utilisé pour construire une représentation globale du système.

Au lancement de l'application, nous avons adopté une stratégie qui permet de minimiser la communication, les agents sont donc placés d'une façon ordonnée sur les différentes machines. Les machines utilisées pour la mise en place de l'application ont toutes les mêmes capacités (CPU, mémoire, etc...). Nous avons placé le même nombre d'agent sur chaque machine ; le système part donc d'un état d'équilibrage de charge. Les plateaux qui se situent en bas de la colonne vont donc se stabiliser avant les autres plateaux. Ainsi, en cours de fonctionnement, certaines machines vont être moins chargées et les agents AM qui se trouvent sur ces machines lancent une demande de migration, d'où une réorganisation du placement des plateaux.

Comme dans oRis, la stabilité se propage de bas en haut. Si l'on schématise les machines utilisées comme dans le schéma (voir Figure 5.12), nous pouvons dire que les machines vont se libérer de bas en haut. L'algorithme d'équilibrage de charge réagi à ce comportement de la colonne afin de mieux exploiter les ressources disponibles. Lors de la mise en marche de l'application, nous avons remarqué un flux de migration de haut en bas.

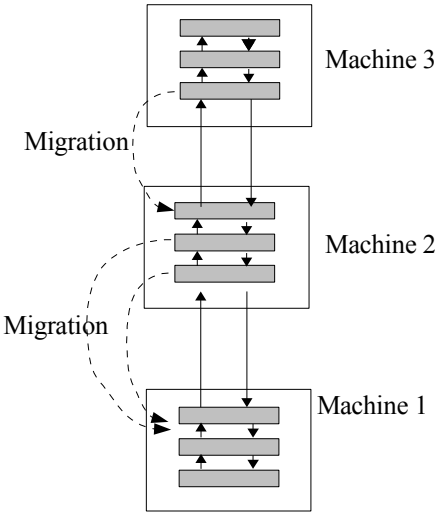

Figure 5.12 : Déplacement des agents

8.1 Résultats

Nous avons pris cinq machines identiques connectées par un réseau local de 10 Mbs. Un agent AM est installé sur chaque machine. Nous avons effectué les mesures en prenons de colonnes de différentes tailles (nombre des plateaux dans la colonne). Les résultats obtenus sont représentés sur le schéma (voir Figure 5.13).

L'axe des abscisses représente le nombre de plateaux de la colonne et l'axe des

ordonnées représente le pourcentage. La courbe représente le gain obtenu pour un placement dynamique par rapport à un placement statique. Si on prend l'exemple d'une colonne à 50 plateaux, le gain obtenu pour un algorithme dynamique par rapport à la version statique est de 245%.

Les seuils utilisés sont des seuils dynamiques ; avec l'évolution du système, ils vont s'adapter au changement de la charge de la machine qui accueille l'agent moniteur. Nous avons fait tourner l'application sur un réseau de 5 machines et une colonne formée de 50 plateaux. Avec l'évolution du système, les plateaux se stabilisent et le nombre de plateaux actifs diminue. La charge sur chaque machine diminue également ainsi que les seuils des agents moniteurs (voir Figure 5.14). Sur le schéma, l'axe des *x* représente l'évolution du système dans le temps et l'axe des *y* représente les seuils. Les agents plateaux, lorsqu'ils sont en état d'activité (plateau non stable), consomment tous la même quantité des ressources (mémoire et CPU). Nous avons pris la charge engendrée par un agent actif comme unité sur l'axe des *y*. Nous remarquons que les seuils s_{min} et s_{max} diminuent avec l'évolution du système, puisque la charge globale du sytème diminue.

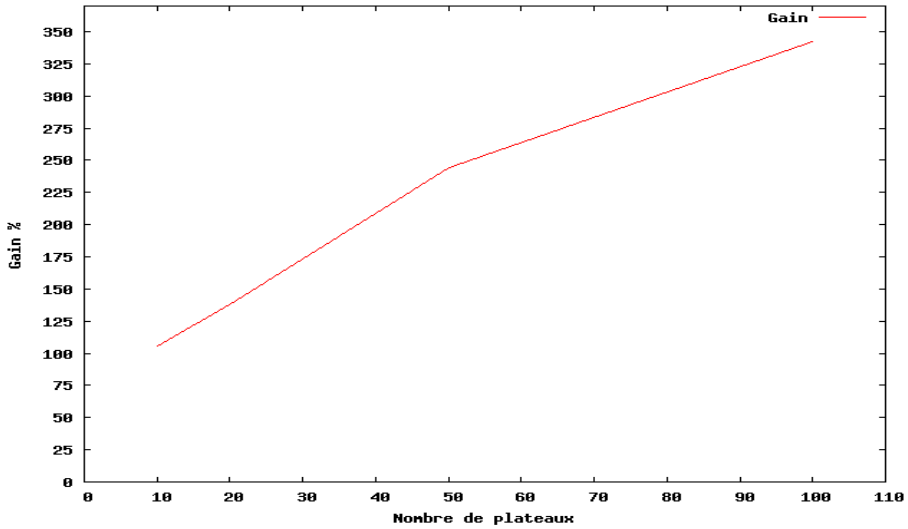

Figure 5.13 : Gain obtenue par un placement dynamique

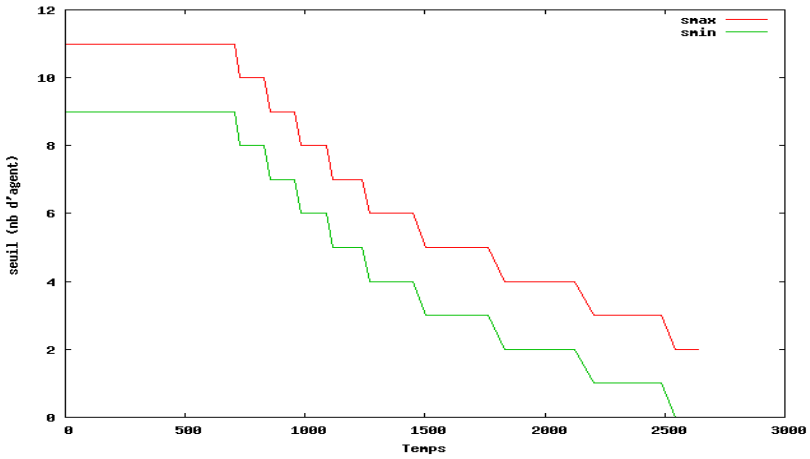

Figure 5.14 : Évolution des seuils

Le système démarre toujours avec une charge bien équilibrée sur les différentes machines. Lorsque la charge d'une machine change (plateau se stabilise) l'agent moniteur détecte ce changement et le processus de négociation est déclenché afin de réajuster le système. L'équilibrage de charge se fait de prêt en prêt et les machines vont avoir toutes la même charge. Le rôle joué par l'agent collecteur est minime.

Afin de montrer l'importance de l'utilisation de l'agent collecteur (ACM). Nous avons repris l'application avec le même nombre des machines (cinq machines). Nous avons considéré une colonne de 100 plateaux. Au lancement de l'application, les 100 plateaux sont placés sur la même machine. Nous avons pris un agent ACM qui se déplace d'une façon cyclique entre les différentes machines, nous pouvons toujours adopter un agent ACM aléatoire. Après quatre migrations, l'agent visitera les cinq machines et il aura un aperçu sur la charge globale du système. Lorsque l'agent ACM se retrouve sur la machine où les plateaux sont placés, il communique avec l'agent moniteur en lui indiquant que sa machine est trop chargée. L'agent moniteur déclenche le processus de migration et les plateaux se déplacent entre les machines afin d'attendre l'état d'équilibre. Nous avons remarqué qu'au bout de 15 négociations (échanges entre les agents moniteurs) le système arrive à un état d'équilibre, ce qui prouve l'importance de l'agent collecteur dans notre architecture.

9 Conclusion

Ce chapitre représente concrètement la mise en place d'un algorithme d'équilibrage de charge d'une application répartie. L'application que nous avons choisie est la résolution d'un système d'équations différentielles qui permet de modéliser l'évolution de la concentration pour les plateaux d'une colonne à distiller. Après une présentation de l'application étudiée, nous avons passé à sa modélisation et à son agentification sous forme des agents mobiles. Une simulation avec la plate-forme oRis permet de bien comprendre l'évolution du système. Ainsi, nous avons remarqué que les différents agents ne vont pas évoluer de la même manière et que les plateaux ne se stabiliseront pas au même moment. La communication et les échanges entre les agents influencent la qualité d'une application distribuée. Nous avons étudié l'impact du coût de communication induit par la distribution de la colonne et ceci en considérant deux stratégies de placement : une qui favorise la communication et la deuxième qui favorise l'équilibrage de charge entre les machines.

Au regard des résultats obtenus, nous avons choisi d'introduire notre algorithme d'équilibrage de charge ainsi nous pouvons dire que l'équilibrage de charge apporte quasi toujours un avantage dans notre contexte d'exécution. Notre algorithme se base sur un équilibrage de charge itératif basé sur des informations locales aux machines. Pour une vue globale, nous avons proposé un agent collecteur, et le rôle de ce dernier a été mis en importance dans le cas d'un système non équilibré au lancement.

Plus généralement, nous pensons que ces résultats sont directement transposables dans toute application qui possède des caractéristiques de la colonne à distiller, c'est à dire des applications dans lesquelles les tâches ne prennent pas toutes le même temps d'exécution conduisant à une utilisation non équitable des ressources disponibles. Notre architecture pour le placement dynamique est facile à mettre en place par le programmeur, ce qui permet de mieux utiliser les ressources disponibles.

Chapitre 6

Déplacement intelligent des agents mobiles

1 Introduction

Les échanges des données et la répartition des tâches d'une application distribuée nécessitent l'interaction entre différentes entités à travers le réseau. Aujourd'hui, le modèle « client/serveur » où les échanges se font par envoi de messages à travers le réseau est le modèle le plus utilisé. Ce modèle possède l'inconvénient d'augmenter le trafic sur le réseau et exige une connexion permanente, ce qui n'est pas le cas des terminaux mobiles. Dans ce chapitre nous proposons une nouvelle approche pour la communication qui utilise les agents mobiles. Ainsi, en envoyant les agents là où les tâches se font, les messages échangés deviennent locaux et libèrent d'autant la charge du réseau. Bien que dans la communication par agent mobile la taille de l'agent ajoute une surcharge lors de la communication, cette taille est compensée par le fait que l'agent effectuant des interactions locales après son déplacement, diminue d'autant la quantité d'informations échangées.

La communication par agents mobiles permet un bon fonctionnement dans le cas où la communication n'est pas établie en permanence ; ainsi, le client se connecte pour créer un agent mobile, l'agent créé joue alors le rôle du client et lui transmettra en retour les résultats lors d'une prochaine connexion. La mobilité permet d'augmenter la fiabilité et le bon fonctionnement des systèmes distribués. Si une machine doit s'arrêter, l'agent qui s'y trouve peut changer de machine pour terminer la tâche en cours.

L'avantage de la communication par l'un des deux modèles, «client/serveur» ou « agent mobile », dépend de la tâche à effectuer (nombre des interactions distantes) et aussi du réseau (débit sur les liens). Dans ce chapitre, nous allons étudier les deux modes de communication et montrer l'influence de la taille de l'agent sur le choix de l'action à effectuer par ce dernier pour échanger de données. Nous avons choisit de valider nos résulats sur l'exemple de la gestion d'un parc informatique. Le but de l'application est d'assurer le bon fonctionnement des logiciels intallés, ainsi que la mises-à-jour des fichiers de configuration, nous montrerons l'intérêt de l'utilisation des agents mobiles dans ce type d'application.

Afin de mieux exploiter les deux modes de communication, nous avons proposé un modèle hybride qui permet de s'adapter au changement du réseau afin de diminuer le trafic. Un agent intelligent mobile est créé afin d'effectuer la tâche demandée par le client. Cet agent utilise les deux modes de communication afin de réduire le délai

d'attente du client. Le choix de la politique suivie par l'agent dépend du débit sur les liens, de la taille de l'agent et des interactions entre les différents sites de réseau. Lorsque le nombre de messages échangés est connu, le choix de la politique devient un simple problème analytique. Pour la plupart des applications, le nombre de messages échangés n'est pas connu à l'avance ; ainsi, nous allons présenter une méthode basée sur l'apprentissage afin d'établir un modèle probabiliste sur les échanges entre les différentes entités de l'application. Ce modèle probabiliste est utilisé afin de construire le graphe qui représente l'ensemble des états du système.

Le processus décisionnel de Markov est utilisé afin de calculer la politique optimale de déplacement de l'agent à travers les différents nœuds de l'application. Cette politique sera ainsi appliquée par l'agent mobile afin de réduire le délai d'attente du client et de réduire le trafic sur le réseau.

Nous avons appliqué notre modèle sur le problème de la recherche d'information sur le Web. Il s'agit des données qui se trouve sur plusieurs site à travers le réseau. Et le but de l'agent est de se dépalcer à travers le réseau afin de collectés des informations pour leurs clients. Les résultats obtenus montrent bien l'efficacité de l'utilisation du MDP [Bellman57] pour le choix de la politique de communication.

2 Communication entre deux machines

L'approche « agents mobiles » permet dans certains cas de diminuer le temps d'exécution total d'une application répartie par la réduction du temps de communication entre les différentes entités constituant l'application. Nous allons comparer la quantité d'informations échangées dans le cas des deux approches, client/serveur et agent mobile, afin d'en déduire le temps de communication. Nous nous plaçons dans le cas d'un échange entre deux machines M_i et M_j reliées par un lien de débit d_{ij} (voir Figure 6.1). Le temps de calcul nécessaire à produire la réponse, étant indépendant du modèle de communication utilisé, n'est pas pris en compte dans notre modélisation.

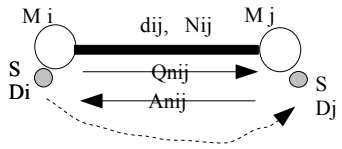

Figure 6.1 : Communication entre deux machines

2.1 Approche client/serveur

Dans le cas d'une seule requête échangée entre deux machines, le temps de réponse mesuré correspond au temps entre l'envoi de la requête par le client et la réception de la réponse envoyée par le serveur. Pour un échange qui nécessite l'envoi de plusieurs requêtes successives entre les deux machines, le temps mesuré est le temps écoulé entre l'envoi de la première requête et la réception du dernier paquet de la réponse. Nous notons par Q_{nij} la taille de la $n^{ème}$ requête envoyée. Le serveur répond avec un message de taille A_{nij}. La quantité d'informations échangées entre les deux machines M_i et M_j dans le cas d'un mécanisme « client/serveur » est donc :

$$D_{cs} = \sum_{n=1}^{N} (Q_{nij} + A_{nij})$$

Si l'on considère que les messages échangés (Question, Réponse) entre deux machines possèdent tous la même taille :

$$\begin{Bmatrix} Q_{nij} = Q_{ij} \\ A_{nij} = A_{ij} \end{Bmatrix} pour\ n = 0\ \grave{a}\ N$$

La quantité d'informations échangées entre les deux machines devient :

$$D_{cs} = N\,(Q_{ij} + A_{ij}) \qquad (1)$$

Les deux machines utilisent le lien M_i-M_j de débit d_{ij} pour leurs échanges. Le temps écoulé entre l'envoi de la première requête et la réception du dernier paquet de la réponse est :

$$T_{cs} = \frac{N(Q_{ij} + A_{ij})}{d_{ij}}$$

2.2 Approche agent mobile

Pour l'approche « agent mobile », le client crée l'agent sur la machine M_i et l'envoie vers M_j pour interroger le serveur en local ; à la fin de l'interaction l'agent mobile revient sur la machine M_i pour donner le bilan au client. Le temps de réponse mesuré correspond au temps écoulé entre la création de l'agent mobile par le client et la réception par ce dernier du bilan transféré avec l'agent mobile.

Supposons que *S* représente la taille de l'agent mobile. D_i (respectivement D_j) la taille de données lorsque l'agent se trouve sur la machine M_i (respectivement M_j). La quantité d'informations échangées entre les deux machines est donc :

$$D_{ma} = 2\ S + D_i + D_j \qquad (2)$$

Lors de son déplacement, l'agent utilise le lien M_i-M_j de débit d_{ij}. Le temps de la communication par agent mobile est :

$$T_{ma} = \frac{2\,S + D_i + D_j}{d_{ij}}$$

2.3 Comparaison entre les deux modèles

Les deux approches vont utiliser le même lien pour leurs échanges. Le débit d_{ij} du lien ne va donc pas influencer le choix du modèle ; le but est de choisir le modèle qui permet de minimiser la quantité d'informations échangées. Les deux formules (1) et (2)

vont nous aider à faire ce choix. La communication par « agent mobile » est d'autant plus avantageuse que la quantité d'information D_{ma} est inférieure à D_{cs}.

$$D_{ma} - D_{cs} < 0 \Rightarrow (2S + D_i + D_j) - N \cdot (Q_{ij} + A_{ij}) < 0$$

Cette formule nous permet de calculer un seuil qui représente le nombre des messages échangés entre les deux machines. À partir de ce seuil, l'interaction par la technologie agent mobile devient plus avantageuse :

$$N > \frac{2S + D_i + D_j}{Q_{ij} + A_{ij}}$$

L'approche « client/serveur » est plus rapide seulement dans le cas d'une faible interaction entre le client et le serveur, après quoi elle devient relativement coûteuse à l'usage. La décision de communiquer par l'un des deux modes dans le cas d'une communication entre deux machines ne dépend pas du débit du lien ; cette décision dépend seulement de la taille de l'agent mobile (code et données) et de celle des messages échangés (Q_{ij} et A_{ij}). Plus le nombre des messages échangés est grand, plus la communication par mode « agent mobile » devient plus avantageuse.

2.4 Gestion d'un parc informatique (application et résultats)

Nous allons prendre l'exemple de l'administration d'un parc informatique. Afin d'assurer le bon fonctionnement des logiciels installés, l'administrateur a besoin de mettre à jour les fichiers de configuration sur les différentes machines de son parc. Dans le but de diminuer la quantité d'informations échangées et d'améliorer l'utilisation de la bande passante, l'administrateur vérifie les fichiers et effectue le transfert, seulement, dans le cas d'une modification. Dans l'approche client/serveur, l'administrateur demande toujours un transfert de fichier sur sa machine, il fait une comparaison avec une version en local et ordonne le transfert d'un nouveau fichier en cas d'incohérence.

Afin de vérifier la cohérence entre deux fichiers, nous pouvons utiliser une méthode basée sur le calcul de la signature électronique. Il s'agit d'une fonction de hachage permettant d'obtenir un condensé (appelé *haché* ou en anglais *message digest*) d'un texte, c'est-à-dire une suite de caractères assez courte représentant le texte qu'il condense. La fonction de hachage doit être telle que, elle associe un et un seul haché à un texte en clair, cela signifie que la moindre modification du document entraîne la modification de son haché. D'autre part, il doit s'agir d'une fonction à sens unique (*one-way function*) afin qu'il soit impossible de retrouver le message original à partir du condensé. Les deux fichiers sont cohérents si et seulement si les deux hachés sont égaux. Ainsi, le haché représente en quelque sorte la signature électronique (*empreinte digitale*) du document.

Les algorithmes de hachage les plus utilisés actuellement sont :

- **MD5**[1] [MD5], développé par Rivest en 1991, crée une empreinte digitale de 128 bits à partir d'un texte de taille arbitraire en le traitant par blocs de 512 bits. Il est courant de voir des documents en téléchargement sur Internet accompagnés d'un

1 MD signifiant Message Digest

fichier MD5 ; il s'agit du condensé du document permettant de vérifier l'intégrité de ce dernier.

- **SHA**[1] [SHA], crée des empreintes d'une longueur de 160 bits. SHA-1 est une version améliorée de SHA datant de 1994 et produisant une empreinte de 160 bits à partir d'un message d'une longueur maximale de 2^{64} bits en le traitant par blocs de 512 bits.

Dans la plupart des cas, les fichiers de configuration d'un logiciel donné ne sont pas modifiés d'une façon régulière. Dans le modèle « client/serveur », la détection d'une modification nécessite l'envoi du fichier vers le client, ce qui surcharge le réseau, même lorsque le fichier n'a pas changé. Afin de diminuer la quantité d'informations échangées nous proposons d'utiliser la technologie d'agents mobiles pour ce type de problème. Ainsi, un agent mobile est créé sur le poste de l'administrateur. Cet agent possède une fonction lui permettant de calculer le haché d'un fichier. L'administrateur lui donne une liste qui contient les noms des fichiers à vérifier ainsi que leur haché. L'agent mobile se déplace sur les différentes machines du parc et y calcule les hachés des fichiers de configuration. Une comparaison entre le résultat de la fonction de hachage et le haché donné par son administrateur garantit l'intégrité du fichier de configuration du site visité. Une fois retourné sur la machine de l'administrateur, l'agent lui donne la liste des fichiers à envoyer. Seuls les fichiers modifiés sont envoyés à travers le réseau.

Pour comparer les deux approches, cette technique de hachage est utilisée avec le gestionnaire basé sur la technologie client/serveur. Le client qui se trouve sur la machine d'administration demande à chaque site distant (serveur administré) de lui envoyer les fichiers à vérifier ; l'algorithme de hachage est utilisé pour calculer le haché du document envoyé. Une différence entre le haché calculé et celui en local, provoquera un envoi du document sur le site distant. Les deux approches sont comparées en faisant varier le nombre des fichiers échangés entre le client et le serveur.

Nous avons pris deux machines équivalentes (mémoire et CPU) pour tester notre application. Ainsi, le temps que l'algorithme de hachage a besoin pour calculer le haché est le même pour les deux machines. Nous le négligeons lors de la comparaison entre les deux modèles. Dans le modèle « client/serveur », le haché est calculé sur la machine cliente, tandis que, dans l'approche « agent mobile », le haché est calculé par l'agent sur le serveur. Les deux machines sont reliées par un lien de débit 10 Mbs. Nous utilisons la plate-forme d'agent mobile JavAct pour effectuer nos tests. Les données réelles du problème sont :

- la taille du code de l'agent = 3694 octets (taille des fichiers *.class* en octets).
- la taille d'un fichier de configuration = 300 octets. Nous faisons l'hypothèse que les fichiers de configuration possèdent tous la même taille.

Nous nous plaçons dans le cas où les fichiers de configuration ne sont pas modifiés. Ce qui indique que l'administrateur n'aura pas besoin de les réenvoyer. Sur le schéma (voir Figure 6.2), l'axe des x représente le nombre de fichiers à vérifier, alors que l'axe des y représente le temps (en ms) écoulé entre l'envoi de la demande par l'administrateur et la réception de la réponse par ce dernier.

1 Secure Hash Algorithm

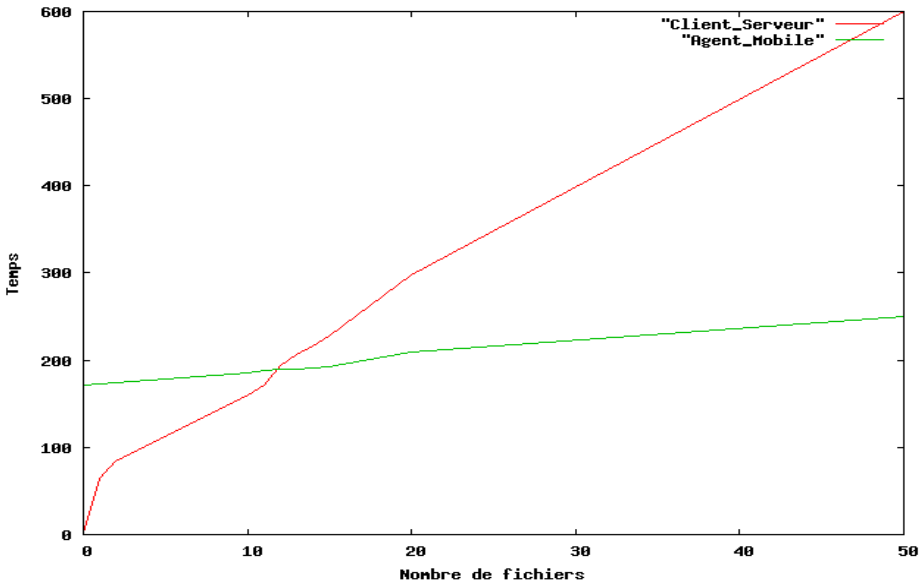

Figure 6.2 : Comparaison entre les deux approches « client serveur » et « agent mobile » dans le cas de la communication entre deux machines

La taille du code de l'agent mobile représente son comportement. Cette taille ne dépend en aucun cas du nombre des messages échangés entre le client et le serveur. Dans le cas de faibles interactions, le modèle à « agent mobile » est pénalisé par cette taille. C'est le modèle « client serveur » qui devient plus avantageux. Avec l'augmentation de la quantité d'informations échangées, la taille du code de l'agent n'a plus autant d'influence.

Dans notre application, l'approche « agent mobile » devient intéressante une fois que le nombre de fichiers échangés est supérieur à un seuil (N>12). Nous avons effectué ces mêmes tests en modifiant le débit du lien utilisé (56 Kbs, 512 Kbs et 5Mbs). Les résultats obtenus montrent que le seuil N ne dépend pas du débit, mais essentiellement de la taille du code de l'agent et de la quantité d'informations échangées entre les deux machines, ce qui confirme les résultats presentés dans la section précédente (cf. § 2.3).

3 Communication avec n machines

La plupart des applications réparties nécessitent des interactions entre plusieurs machines à travers le réseau. Nous allons considérer le cas d'une application qui nécessite la communication entre *n* machines connectées par le réseau TCP/IP (voir Figure 6.3). Dans le cas d'une communication « client/serveur », le client qui se trouve sur la première machine communique avec les (*n-1*) machines afin d'accomplir la tâche demandée. Ce modèle utilise les liens réseaux existant entre la première machine et les autres machines participantes à l'accomplissement de la tâche démandée. Dans le cas d'une communication par « agent mobile », un agent est créé sur la première machine, se déplace entre les différents sites et effectue des interactions locales. L'agent revient ensuite sur la première machine afin de lui communiquer le bilan.

Il faut signaler que les deux modèles ne vont pas utiliser les mêmes liens pour effectuer les interactions et accomplir la tâche demandée. Dans le modèle client/serveur, ce sont les liens (d_{12}, d_{13}, ... d_{1n}) qui relient la première machine aux

autres de l'application qui seront utilisées. Tandis que dans le modèle agent mobile ce sont les liens (d_{12}, d_{23}, ... d_{n1}) qui seront utilisés pour le déplacement de l'agent. Ceci nous invite à intégrer dans nos calculs les débits des différents liens.

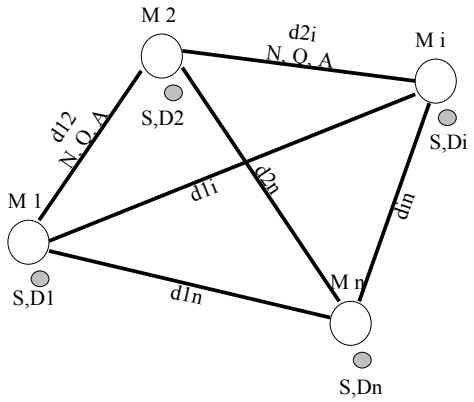

Figure 6.3 : Communication entre n machines

Avant de procéder à la comparaison des deux approches, il faut se rappeler que nous ne prenons pas en compte le coût de traitement d'une requête sur chaque serveur qui reste le même. Pour comparer les deux approches, les grandeurs suivantes sont à considérer lors de nos calculs :

- S est la taille du code de l'agent mobile.
- D_i est la taille des données de l'agent mobile après l'interaction avec la machine M_i.
- Q_{ij}, A_{ij} sont respectivement la taille d'une question et la taille de la réponse échangées entre M_i et M_j.
- N_{ij} est le nombre des messages échangés entre M_i et M_j.
- d_{ij} est le débit du lien M_i-M_j.

3.1 Client/serveur

Le client qui se trouve sur la machine M_1 communique avec les (n-1) machines (M_2, M_3, ... M_n). Les liens utilisés, pour les échanges de données, sont M_1M_2, M_1M_3 ... M_1M_n. Leurs débits respectifs sont d_{12}, d_{13}, ... d_{1n}. Le temps total de la communication vaut donc :

$$T_{cs} = \sum_{j=2}^{n} \frac{N_{1j}(Q_{1j} + A_{1j})}{d_{1j}}$$

3.2 Agent mobile

Un agent mobile est créé sur la machine M_1. Il se déplace sur la 2ème machine M_2 en utilisant le lien M_1M_2 de débit d_{12}. Une fois arrivé sur M_2, l'agent effectue des échanges locaux. La taille des données collectées par l'agent est D_2. Ce même agent se déplace ensuite sur la machine M_3 à travers le lien M_2M_3 de débit d_{23}. Ainsi de suite jusqu'à son

retour sur la 1ère machine M_1 afin de fournir les résultats au client. Le temps total de la communication est :

$$T_{ma} = \sum_{j=1}^{n-1} \left(\frac{S + D_i}{d_{i,i+1}} \right) + \frac{S + D_n}{d_{n1}}$$

3.3 Comparaison entre les deux approches et résultats des tests

La comparaison entre les deux approches montre que la communication par agent mobile est plus avantageuse lorsque le nombre des messages échangés est grand. Dans le cas de faible interaction, l'approche client/serveur devient plus avantageuse. Afin de montrer l'influence de la quantité d'informations échangées sur le choix du modèle de communication, nous avons pris l'application de l'administration d'un parc informatique présentée précédemment.

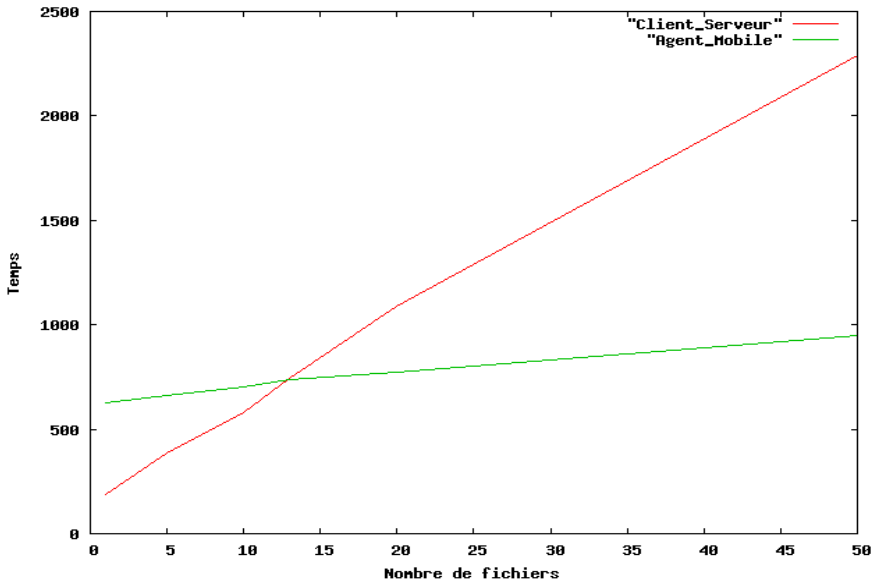

Figure 6.4 : Comparaison entre les deux approches client serveur et agent mobile dans le cas de 5 machines

Nous avons pris cinq machines identiques reliées par un réseau LAN de débits 10Mbs. Nous nous plaçons dans le cas où les fichiers de configuration ne sont pas modifiés. Ce qui indique que l'administrateur n'a pas besoin de réenvoyer ces fichiers. Sur le schéma ci-dessus (voir Figure 6.4), l'axe des abscisses représente le nombre de fichiers à vérifier, l'axe des ordonnées représente le temps (en ms) écoulé entre l'envoi de la demande par l'administrateur et la réception de la réponse par ce dernier. L'approche « agent mobile » devient intéressante une fois le nombre de fichiers échangés dépasse un certain seuil (N>13).

Nous avons fait tourner la même application en faisant varier le nombre de machines administrées par le gestionnaire du parc, afin de comprendre l'influence de ce

nombre sur le seuil. Il faut noter que le nombre de fichiers à vérifier par le gestionnaire est le même pour toutes les machines du parc. Les résultats obtenus sont représentés sur le schéma ci-dessous (voir Figure 6.5). L'axe des x représente le nombre de machines du parc. L'axe des y représente le seuil N, c'est-à-dire le nombre de fichiers à vérifier sur chaque machine. Ce seuil augmente avec l'augmentation du nombre de machines à visiter.

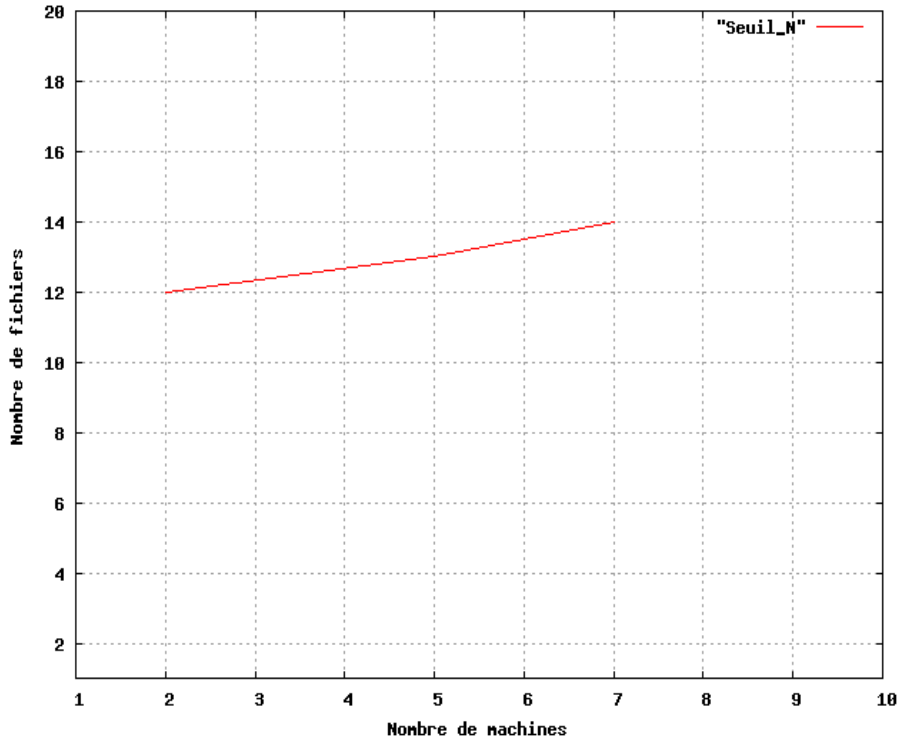

Figure 6.5 : Seuil N avec modification du nombre de machines à visiter

La comparaison entre les deux approches montre que le choix de la technologie de communication dépend de la quantité d'informations échangées entre les différents acteurs d'une application distribuée. Il dépend aussi de la taille de l'agent mobile et du débit des liens utilisés pour les échanges. Pour une faible interaction, l'approche client/serveur est plus avantageuse, tandis que l'approche agent mobile devient plus avantageuse pour une forte interaction. Dans une application réelle, nous pouvons rencontrer des situations qui nécessitent des interactions fortes dans certaines étapes et faibles dans d'autres. Pour les interactions fortes, la communication par agent mobile est plus avantageuse tandis que pour les interactions faibles la communication par mode client/serveur s'avère plus intéressante. Ceci nous invite à proposer un nouveau mode de communication : « la communication hybride » que nous détaillons dans la section suivante. Un agent est créé sur le 1er site, il est chargé d'interagir avec les différents sites de l'application. Afin d'accomplir la tâche demandée, cet agent peut alors choisir une politique mixte, interaction distante et migration.

4 La communication hybride

Dans [Strasser97], une comparaison entre les deux modes de communication est effectuée et un scénario de communication intégrant les deux modes est ensuite

proposé. Ce scénario montre que l'intégration des deux modèles permet de réduire le temps d'exécution de l'application. Ces résultats nous motivent à proposer et étudier un troisième mode de communication, à savoir la communication hybride. Dans la communication hybride, un agent mobile intelligent est créé sur le premier site et doit accomplir la tâche demandée par son client. Pour ce faire, l'agent a besoin d'interagir avec les *n* sites de l'application. Contrairement au modèle présenté dans le paragraphe précédent où l'agent effectuait des migrations entre les sites lui permettant une communication locale aux sites visités, nous proposons que l'agent désirant communiquer avec un site distant, puisse choisir de ne pas se déplacer sur ce site, mais d'effectuer des interactions distantes. Ainsi l'agent mobile n'est pas obligé de visiter tous les sites de l'application. Il choisit la communication distante lorsque la migration devient trop coûteuse.

L'agent mobile créé sur le 1er site peut choisir de communiquer par l'un des deux modes : interaction distante ou migration. Dans les deux cas, l'agent utilise le lien d_{12} comme support de communication (voir Figure 6.6). En seconde étape, l'agent interagit avec le 3ème site. Deux cas peuvent se présenter :

1. Si l'agent choisit (en 1ère étape) de communiquer par mode « client/serveur », alors il reste sur le 1er site. Dans ce cas, il utilise le lien d_{13} pour communiquer avec le 3ème site.

2. Si l'agent choisit (en 1ère étape) de migrer sur le 2ème site, c'est le lien d_{23} qui est utilisé pour communiquer avec le 3ème site.

L'utilisation des formules obtenues dans la section 2 de ce chapitre permet de calculer le temps que l'agent va prendre pour interagir avec les différents sites de l'application et ceci pour chacune des politiques possibles. Les différentes possibilités du déplacement de l'agent seront ainsi représentées sous forme d'un arbre binaire (voir Figure 6.6). Soit *n* le nombre de sites à visiter, le nombre des chemins possibles sera de 2^{n-1} chemins. Les deux approches « client/serveur » et « agent mobile » étudiées dans la section précédente figurent parmi les chemins possibles représentés sur ce graphe. L'approche client/serveur se trouve à gauche de l'arbre et l'approche agent mobile se trouve à droite de l'arbre.

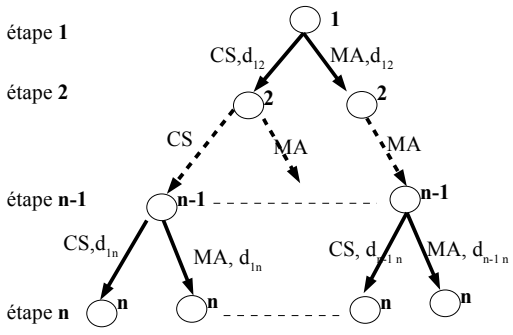

Figure 6.6 : Communication hybride

Pour une application donnée, nous considérons que la taille (S) du code de l'agent est constante et que les messages (Q_{ij}, A_{ij}) échangés dans le mode « client/serveur » dépendent de l'application. Ces messages sont aussi considérés comme constants. En plus de son code, l'agent transporte avec lui des informations représentant les résultats

de ses interactions avec les différents sites déjà visités. Soit D_i la taille des données cumulées après la visite du $i^{ème}$ site. Cette taille dépend de la taille des messages échangés (constante) ainsi que du nombre des messages échangés (N_i).

$$D_i = f(N_1, N_2, \ldots, N_i)$$

Le choix du mode de communication dépend des débits d_{ij} des liens utilisés et du nombre de messages échangés (N_{ij}) entre les différentes machines. Le débit est une information à récolter en permanence sur les différents liens utilisés par l'application. Il est calculé par l'utilisation du protocole ping [Lundgren01]. Ce débit est diffusé périodiquement aux différentes machines participantes à la prise de décision. Si pour une application donnée, le nombre de messages échangés est connu à l'avance, ce nombre permet de calculer le temps de communication pour tous les chemins du graphe. Une comparaison entre les différents résultats permet de choisir la communication la plus efficace.

5 Recherche de la politique optimale : cas des interactions certaines

Nous prenons le cas où les messages échangés entre les différents acteurs de l'application sont connus. L'incertitude sur les échanges entre les machines sera étudiée dans la section 6 de ce chapitre. Le but de l'algorithme est de retrouver la politique que l'agent va adopter afin d'effectuer la tâche demandée dans le plus court délai. Sur le graphe de la section précédente (voir Figure 6.6), un coût est attribué pour chaque arrête. Ce coût correspond au temps de communication.

La réalisation de la tâche nécessite la visite de toutes les machines utilisées par l'application. Ceci correspond à un chemin reliant la racine du graphe à l'une de ses feuilles. Nous allons prendre le chemin ayant le moindre coût.

5.1 Construction du graphe et calcul des coûts

La construction de notre arbre binaire nécessite la connaissance des coûts qui sont attribués à chaque arrête de l'arbre. Comme on l'a vu précédemment, le calcul du coût nécessite la connaissance de plusieurs paramètres :

1. les caractéristiques de l'agent mobile (taille, données, etc...),
2. la taille des messages échangés ainsi que leur nombre,
3. le débit des liens utilisés.

Pour une arrête donnée, le coût associé (temps de communication) correspond à la taille des données échangées divisée par le débit du lien utilisé.

$$coût = \frac{Data}{Debit}$$

Soit *n* le nombre des machines de l'application, le nombre d'arrêtes L sera de :

$$L = \frac{n(n-1)}{2}$$

Pour la quantité de données échangées, deux cas sont à considérer : la communication par migration ou la communication par interaction distante. Dans le cas de *n* machines, le nombre des possibilités sera de :

$$D = 2\ (n-1)$$

La valeur (coût) que l'on attribue aux différentes transitions (arrêtes) du graphe nécessite la connaissance de «*L.D*» grandeurs différentes. Le nombre des opérations (*L.D)* vaut:

$$L.D = n(n-1)^2$$

La complexité de l'algorithme qui calcule les différents coûts est donc en $O(n^3)$.

5.2 Algorithme

Afin de se rapprocher d'un problème connu en algorithmique, nous ajoutons un nouvel état dans notre arbre binaire. Il s'agit d'un état final T, relié à toutes les feuilles du graphe. On attribue un coût zéro aux arrêtes reliant les différentes feuilles à l'état T (voir Figure 6.7). Dans ce cas, la recherche de la politique que l'agent adoptera revient à la recherche du plus court chemin entre la racine et l'état T.

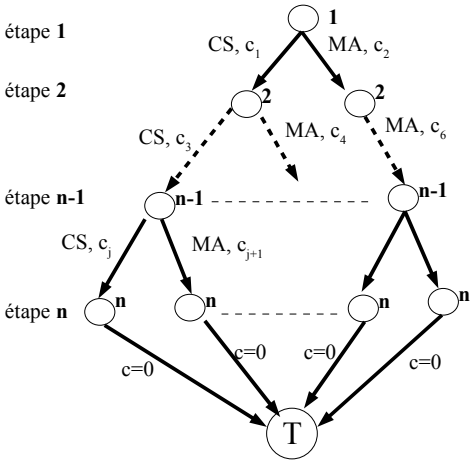

Figure 6.7 : Recherche de plus court chemin

Notre algorithme est basé sur l'algorithme de Dijkstra [Dijkstra59]. Cet algorithme permet de résoudre le problème du plus court chemin pour un graphe G(S, A) orienté et connexe dont le poids lié aux arcs est positif ou nul.

- S est l'ensemble des sommets de G.
- A est l'ensemble de arrêtes de G.

Algorithme 6.1

Données : G=(S,A) un graphe avec une valuation positive c des arrêtes,
s un sommet de S
Initialiser tous les sommets à « non marqué » ; Initialiser tous les labels L à « +∞ »
Marquer s;
L(s):=0 // Initialiser le label de s à 0

Tant Que il existe un sommet non marqué
Pour chaque sommet y non marqué
Calculer L(y) := min { L(x) + c(x,y) | x voisin marqué de y}
Fin Pour
Choisir le sommet y non marqué de plus petit label L
Marquer y
si *y ==T* ***fin*** *// modification de l'algorithme de Dijkstra*
Fin TantQue

L'algorithme de dijkstra se termine lorsque tous les sommets du graphe sont marqués. Notre but est de calculer le plus court chemin entre la racine et l'état T, alors l'algorithme s'arrêtera si l'état T est marqué. Nous avons ajouté une petite modification à l'algorithme de Dijkstra afin de vérifier cette condition. La connaissance de plus court chemin permettra de donner la politique à suivre par l'agent mobile. La complexité de l'algorithme de dijkstra est de $O(|S|.|A|)$.

En plus du poids qui donne le coût de communications entre deux machines, chaque arrête possède un attribut qui précise l'action à prendre. Deux actions sont envisageables (*MA* et *CS*). La connaissance du plus court chemin permet ainsi d'obtenir la politique de l'agent. Cette politique sera définie par :

$$\pi = (Ac_1, Ac_2, \ldots, Ac_n) \quad avec \quad Ac \in \{MA, CS\}$$

Ac_i représente l'action que l'agent prend pour communiquer avec la $i^{ème}$ machine.

Pour la plupart des applications, le nombre de messages échangés n'est pas connu à l'avance. L'agent ainsi évolue dans un monde incertain. Cette incertitude est modélisée par un modèle probabiliste qui permet de représenter les interactions entre les différentes entités de l'application. Dans la section suivante, nous présentons un modèle d'apprentissage simple qui permet de modéliser les interactions dans notre système.

6 Recherche de la politique optimale : cas des interactions incertaines

La connaissance des échanges entre les différentes entités permet de calculer le temps total dû à la communication. Ce temps correspond au coût que l'on utilise lors de la recherche de la meilleure politique de déplacement pour l'agent. Dans une application réelle, les interactions entre les différents acteurs ne sont pas connues, mais il existe des méthodes pour les modéliser et ceci en utilisant un modèle probabiliste. Nous parlons d'un agent mobile évoluant dans un monde incertain. Ainsi, nous proposons des algorithmes pour la prise de décision lors de l'évolution des agents mobiles dans un monde incertain. Les algorithmes proposés seront validés à partir

d'une application réelle.

6.1 Apprentissage sur les interactions : cas de la recherche d'informations sur le Web

L'étude théorique effectuée dans la partie 3 de ce chapitre montre que le choix de la politique adoptée par l'agent dépend du débit des liens utilisés. Il dépend également des interactions (*Ni*) entre les différents acteurs de l'application. Dans le cas de la recherche d'informations sur le Web, le client envoie une requête au système, lui spécifiant la tâche à réaliser. La réalisation de cette tâche nécessite l'interaction avec plusieurs serveurs du réseau Internet (voir Figure 6.8) afin de collecter les informations nécessaires. Les serveurs participant à l'accomplissement de la tâche peuvent être des serveurs Web, des serveurs de base de données, des annuaires LDAP, des serveurs de calcul, etc... Les interactions dans le système vont donc dépendre de la requête envoyée par le client.

Les clients se connectent sur le système et lancent leurs requêtes. Un agent mobile est créé ; il est chargé de réaliser la tâche demandée. L'objectif est de donner la réponse au client dans le plus court délai. Dans le cas où les interactions entre les différentes entités sont connues, le choix de la politique que l'agent va suivre représente un problème simple à résoudre (algorithme de Dijkstra). Dans la plupart des applications, ces informations sont difficiles à connaître et la solution consiste à modéliser le comportement de l'application sous forme d'un modèle probabiliste.

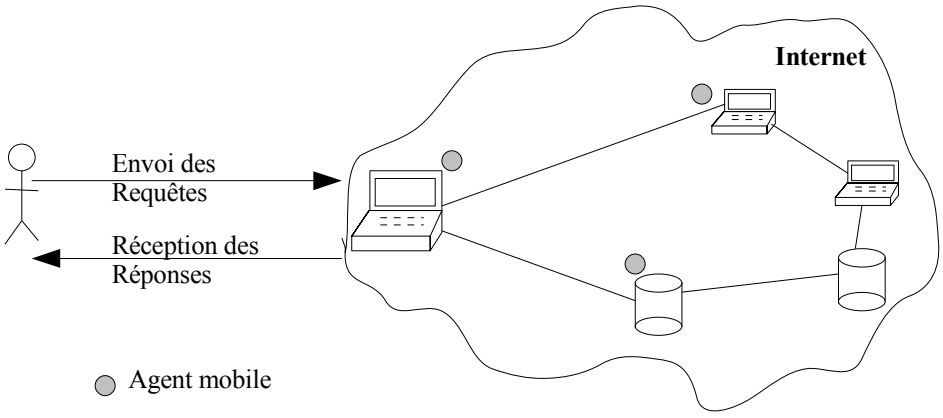

Figure 6.8 : Recherche d'information sur le Web

Un client se connecte à notre système et lance une requête qui spécifie la tâche à réaliser. La réalisation de cette tâche nécessite des interactions entre les différentes entités de l'application. Notre objectif est de modéliser ces interactions sous forme d'un modèle probabiliste. La représentation de ce modèle se fait sous forme d'un tableau (voir Table 6.1), dans lequel nous stockons les différents scénarios d'interaction que l'application peut effectuer. Un scénario d'interaction est représenté par un tuple dans le tableau (un tuple représente les différentes interactions *Ni* du scénario).

L'apprentissage sur le système est fait pendant un certain temps, appelé « période d'observation ». Pendant cette période nous allons retenir tous les scénarios que l'on puisse rencontrer pour l'application. Ces scénarios d'interaction n'ont pas tous les

mêmes pourcentages d'apparition. Un scénario s_i va se reproduire un certain nombre de fois pendant la période d'observation, soit m_i ce nombre. Ainsi, nous allons compter le nombre d'apparition de chaque scénario. Soit M le nombre total de demandes faites par le client pendant la période d'observation. La probabilité d'apparition d'un scénario s_i est donc :

$$p_i = \frac{m_i}{M}$$

Si	N1	N2	...	Nn		Prob
S1	1	2	...	1		p1
S2	3	3	...	2		p2
S3	2	5	...	10		p3
	...	...	...	...		...
Sm	15	10	...	1		pm

Table 6.1 : Résultat de la phase d'apprentissage.

Soit S l'ensemble des scénarios possibles de l'application, une fonction de probabilité P est défini sur l'ensemble S de la façon suivante :

$$\begin{aligned} P: \quad & S \rightarrow [0,1] \\ & s \rightarrow p \end{aligned}$$

La phase d'apprentissage permet de mieux comprendre le comportement du système. La durée de cette période dépend de l'application étudiée et le but recherché est de rapprocher le modèle probabiliste le plus possible de la réalité. Dans une application donnée, nous pouvons envisager une solution où l'observation se fait d'une façon permanente, ainsi lorsque le système détecte un grand changement dans le comportement des clients, un nouveau modèle est calculé pour la représentation du système. D'autres modèles d'apprentissage peuvent être utilisés pour modéliser notre système. Le vaste problème de la modélisation du système sort de l'objectif de ce travail. Après avoir proposé une modélisation du système, nous présentons le mécanisme d'utilisation du modèle pour trouver la politique qui minimise la communication des agents dans le système.

6.2 Méthode statistique pour le choix de la politique de l'agent

La phase d'apprentissage nous permet de mieux comprendre les interactions entre les différentes entités de l'application ; ainsi, les données stockées sont utilisées afin de choisir la politique que l'agent poursuit pour accomplir la tâche demandée. Dans le cas d'une application qui nécessite l'interaction entre n sites distants, pour échanger avec une machine de l'application, l'agent peut utiliser les deux modes de communication par agent mobile ou par client/serveur. Ainsi, la politique de l'agent est définie par :

$$\pi = (Ac_1, Ac_2, \ldots, Ac_{n-1}) \quad avec \quad Ac \in \{MA, CS\}$$

Ac_i représente l'action que l'agent va prendre pour communiquer avec la i^{ème}

machine.

Les données du problème sont :

- *G* est l'ensemble des politiques possibles, le cardinal de *G* vaut 2^{n-1} .
- *S* est l'ensemble des scénarios possibles. Une fonction de probabilité est définie sur cet ensemble.

Comme on a vu dans la section 4 de ce chapitre, les différentes politiques que l'agent peut choisir sont représentées sous la forme d'un graphe binaire. Pour un scénario *s*, la connaissance de l'état du réseau (débit des liens) permet de calculer les différents coûts de communication sur les différentes arrêtes du graphe. Une politique π définit les actions que l'agent entreprend pour effectuer ces interactions. Cette politique indique le chemin à suivre sur l'arbre binaire. Ce chemin permet de calculer le coût total de l'application de la politique π dans le cas du scénario *s*. Soit $C(s,\pi)$ ce coût, ainsi pour chaque scénario s_i de l'ensemble *S*, nous calculons le coût associé et ceci pour chaque politique π_i de l'ensemble *G*.

Notre but est de choisir la politique qui minimise le délai d'attente du client. Pour ce faire, nous commençons par le calcul du coût moyen de l'application de cette politique en tenant compte de l'ensemble *S* des scénarios. Soit C_m la fonction qui représente ce coût moyen, cette fonction est définie comme suit :

$$C_m(\pi)=\sum_{s\in S} C(s,\pi).\,p(s)$$

L'agent va ainsi choisir la politique qui minimise le coût moyen de la communication, soit π^* cette politique :

$$\pi^*=\pi \;/\; C_m(\pi^*)=min_{\pi\in G}\, C_m(\pi)$$

Le choix de cette politique représente un temps supplémentaire à ajouter pour chaque requête du client. Nous envisagons d'effectuer ces calculs hors-ligne (off-line) et ceci pour chaque état du réseau, ce qui nous permet de définir la politique qui minimise le temps de réponse pour chaque état. Une fois le calcul de la politique fait, cette dernière est transmise à l'agent pour l'appliquer en fonction de son état et de l'état de son environnement.

6.3 Discussion et limites de la méthode

Les informations sur l'état du réseau sont à envoyer régulièrement sur le 1er site de l'application, c'est-à-dire le site qui reçoit la demande des clients et qui crée l'agent mobile. Un agent mobile est créé pour chaque nouvelle requête. Après sa création, la tâche à effectuer est donnée à l'agent ainsi que l'itinéraire à suivre (politique de l'agent). L'agent applique cette politique, sans faire d'ajustement, du début jusqu'à la fin de la tâche. Il faut noter que lorsque l'agent évolue dans l'application de la politique choisie, il a plus d'informations sur le scénario d'interactions et sur son environnement mais il n'a pas la possibilité de changer sa politique.

À la fin de ce chapitre, nous présentons une nouvelle méthode basée sur le processus décisionnel de Markov. Elle permet de choisir une politique en tenant compte des différents états du système. Cette nouvelle méthode prévoit une politique qui utilise les informations acquises au cours de l'avancement de l'agent dans son environnement.

7 Processus décisionnel de Markov

Le processus décisionnel de Markov, que nous appellerons MDP, est un modèle probabiliste qui aide à la prise de décisions séquentielles sous incertitude. Dans ce modèle, les récompenses et les probabilités de transition d'un état à un autre ne sont fonction que de l'état courant du système et de l'action courante, et non de la trajectoire passée. Les MDPs permettent, à partir d'un état initial, d'obtenir une politique d'actions dont l'exécution parviendra à réaliser au mieux les objectifs de l'agent.

7.1 Définition d'un MDP

Un processus décisionnel de Markov est un processus stochastique, c'est-à-dire qu'à partir d'un état *s*, une action *a* peut mener l'agent dans différents états *s′*. Chaque transition de *s* vers *s′* est caractérisée par une probabilité de transition. Cette probabilité doit satisfaire la propriété de Markov, c'est-à-dire que la probabilité de transition ne doit dépendre que de l'état *s* du système à l'instant *t* et de l'action effectuée.

La définition d'un MDP [Puterman94] pour la planification des actions d'un agent sous incertitude se représente par un 4-uplets (*S*, *A*, *Pr*, *R*) où :

- S est l'ensemble des états du système.
- A est l'ensemble des actions possibles pour l'agent.
- $Pr: S \times A \times S \to [0,1]$ est une distribution de probabilités, telle que $Pr(s, a, s')$ représente la probabilité de passer de l'état s à l'état s' en effectuant l'action $a (\in A)$.
- $R: S \to \mathbb{R}$ est une fonction de récompense permettant de calculer l'utilité espérée.

L'utilité espérée permet d'évaluer les politiques. La politique optimale est celle dont l'utilité espérée est maximum. Elle fournit à l'agent un comportement optimal, c'est-à-dire que dans chaque état, l'action choisie est celle susceptible d'augmenter, le plus, le gain de l'agent. Un MDP est dit à horizon fini si : (1) au moins un état terminal appartient à son espace d'état *S* et (2) le nombre maximal d'actions qui peuvent être consécutivement appliquées à partir de n'importe quel état jusqu'à un état terminal est fini. Dans le cas contraire, le MDP est à horizon infini.

7.2 Politique de l'agent

Nous appelons politique une fonction $\pi: S \to A,$ qui spécifie l'action à appliquer dans chaque état $s \in S.$ Formellement, $\pi(s)$ est l'action choisie à appliquer dans l'état *s*. Le gain espéré associé à une politique, noté $V^{\pi}(s),$ est tout simplement le gain espéré associé à l'action correspondante :

$$V^{\pi}(s)=\sum_{s'\in S} Pr(s,\pi(s),s')\cdot V(s') \tag{3}$$

À la lumière de la définition précédente, nous définissons la fonction du gain V pour chaque état non terminal $s\in S$. En effet, cette fonction est définie par la somme de deux valeurs. La première notée *r(s),* est appelée le gain immédiat qui représente la récompense à obtenir du fait que l'on soit dans l'état *s*. Le gain immédiat est spécifié selon le problème à traiter. La seconde valeur définissant la fonction du gain V, quant à elle, caractérise ce que l'on peut gagner à partir de l'état *s* par l'application d'une action. Elle dépend donc de la politique suivie dans l'état *s*. De façon formelle, pour une politique π, on a :

$$V(s)=r(s)+V^{\pi}(s) \tag{4}$$

Le gain associé à un état terminal *s* est égal au gain immédiat indépendamment de la politique suivie : $V(s)=r(s)$. Afin d'éviter la divergence dans le cas où le MDP est à horizon infini un facteur de pondération $0<\gamma<1$ peut être utilisé de la façon suivante :

$$V(s)=r(s)+\gamma\cdot V^{\pi}(s) \tag{5}$$

Nous avons donc :

$$V(s)=r(s)+\gamma\cdot\sum_{s'\in S} Pr(s,\pi(s),s')\cdot V(s') \tag{6}$$

L'équation précédente (6) est l'équation de Bellman qui définit la fonction du gain V sachant qu'une politique π est suivie. Nous pouvons définir une politique optimale pour un MDP en se fondant sur l'équation de Bellman. En fait, les fonctions du gain espéré $V^{\pi},\forall\pi$ définissent un ordre partiel ($\geq$) sur les différentes politiques. Une politique π est mieux qu'une autre π', $(\pi\geqslant\pi')$, si son gain espéré est supérieur, ou égal, à celui de π' pour tous les états. Formellement, $\pi\geqslant\pi' ssi\ \forall s\in S, V^{\pi}(s)\geqslant V^{\pi'}(s)$.

7.3 Politique optimale

Il existe au moins une politique qui soit meilleure que toutes les autres politiques. Cette politique est appelée politique optimale et notée π^{*}. Plus formellement, $\forall s\in S,\forall\pi, V^{\pi^{*}}(s)\geqslant V^{\pi'}(s)$, ce qui conduit à :

$$\forall s\in S, V^{\pi^{*}}(s)=\max_{\pi} V^{\pi'}(s)$$

La condition précédente peut être aussi exprimée en terme d'actions par :

$$\forall s \in S, V^{\pi^*}(s) = \max_{a \in A} \{ \sum_{s' \in S} Pr(s, a, s') \cdot V^*(s') \} \quad (7)$$

où, $V^*(s')$ est le gain optimal associé à l'état s' sachant que la politique optimale π^* est suivie.

Le gain $V^*(s')$ correspondant à la politique π^* peut être défini par l'équation :

$$V^*(s) = r(s) + \gamma \cdot V^{\pi^*}(s) \quad (8)$$

Selon les deux équations (7) et (8), nous avons :

$$V^*(s) = r(s) + \max_{a \in A} \{ \gamma \cdot \sum_{s' \in S} Pr(s, a, s') \cdot V^*(s') \} \quad (9)$$

et

$$\forall s \in S, V^{\pi^*}(s) = \max_{a \in A} \{ \sum_{s' \in S} Pr(s, a, s') \times [r(s') + \gamma \cdot V^{\pi^*}(s')] \} \quad (10)$$

Les deux équations (9) et (10) sont les équations d'optimalité de Bellman définissant la fonction du gain V^* et le gain espéré associé à une politique optimale π^*.

Par ailleurs, la résolution d'un MDP consiste à déterminer la politique optimale pour chaque état. En effet, la connaissance de toutes les valeurs $V^*(s), \forall s \in S$ permet de déterminer une politique optimale. En se fondant sur l'équation (7), une politique optimale peut être déterminée par :

$$\forall s \in S, \pi^*(s) = arg \max_{a \in A} \{ \sum_{s' \in S} Pr(s, a, s') \cdot V^*(s') \}$$

Le calcul des valeurs $V^*(s), \forall s \in S$ est faisable. En effet, l'équation d'optimalité de Bellman (9) est un système de $|S|$ équations à $|S|$ inconnues $V^*(s)$. En principe, il existe différentes méthodes mathématiques pour résoudre un tel système d'équations non linaires. De façon similaire, nous constatons que la connaissance de toutes les valeurs $V^{\pi^*}(s), \forall s \in S$ permet de déterminer une politique optimale comme suit :

$$\forall s \in S, \pi^*(s) = arg \max_{a \in A} \{ \sum_{s' \in S} Pr(s, a, s') \cdot [r(s') + \gamma \cdot V^{\pi^*}(s')] \}$$

L'équation d'optimalité de Bellman (10) est un système de $|S|$ équations à $|S|$ inconnues $V^{\pi^*}(s)$. Ce système peut être résolu par des méthodes mathématiques.

La résolution d'un système de $|S|$ équations à $|S|$ inconnues, pour un grand système, est très coûteuse et parfois infaisable informatiquement parlant. Dans [Sutton98], les auteurs proposent l'exemple d'un simple jeu appelé « Backgammon » qui peut être formalisé en un MDP fini avec un espace d'états de l'ordre de 10^{20} états. Les auteurs

affirmaient, en 1998, que la résolution de l'équation (9) aurait pu prendre des milliers d'années sur la machine la plus puissante connue. Ce coût considérable nous oblige à proposer des méthodes d'approximation qui peuvent conduire à des solutions qui se rapprochent de la solution optimale.

Dans la littérature, il existe plusieurs familles de méthodes pour résoudre un MDP fini. Des algorithmes utilisant les techniques de la programmation dynamique et le principe de Bellman permettent de calculer la politique optimale. Les deux algorithmes les plus connus sont : *Policy Iteration* [Howar60] et *Value Iteration* [Bellman57].

7.4 L'algorithme Policy Iteration

Cet algorithme est fondé sur le fait que la connaissance d'une politique π et des valeurs de gain $V(s), s \in S$ étant l'état courant correspondant à π, permette de trouver une politique π' meilleure que π. De plus, l'amélioration consécutive d'une politique conduit à la politique optimale. Le calcul d'une politique optimale dans cet algorithme se réalise en deux phases :

1. **L'évaluation** : cette phase évalue la politique π donnée en entrée. L'évaluation d'une politique n'est que le calcul des valeurs $V(s), s \in S$ sachant que la politique π est suivie dans l'état *s* et le sera dans les états futurs. L'algorithme utilise l'équation (6) pour calculer les valeurs $V(s)$. La condition d'arrêt est que la différence entre l'ancienne valeur de $V(s)$ et celle obtenue par l'itération actuelle est inférieure ou égale à une certaine valeur minimale ϵ.

2. **L'amélioration** : cette deuxième phase améliore la politique π trouvée dans la première phase sauf si celle-ci est déjà optimale.

Cet algorithme utilise le principe que la politique produite par la phase d'amélioration peut être à son tour améliorée, et ainsi de suite. Pour un MDP à horizon fini, le nombre de politiques est fini. L'algorithme converge vers la politique optimale au cours d'un nombre fini d'itérations. L'algorithme s'arrête lorsque la politique ne s'améliore pas, on dit que l'optimalité est atteinte.

Algorithme 6.2

Données : Un MDP fini, une politique initiale π, un seuil minimal positif ϵ
But : Calculer une politique optimale avec une marge d'erreur $\leqslant \epsilon$

Initialisation : $V(s), \forall s \in S$
//phase d'évaluation de la politique courante
Répéter
  politique_stable $\leftarrow$ *vrai*
  Répéter
    $\Delta \leftarrow 0$
  Pour tout $s \in S$ **Faire**
    $v \leftarrow V(s)$
    $V(s) \leftarrow r(s) + \gamma \cdot \sum_{s' \in S} Pr(s, \pi(s), s') \cdot V(s')$
    $\Delta \leftarrow max\{\Delta, |v - V(s)|\}$

Fin Pour
Jusqu'à $\Delta \leqslant \epsilon$
//phase d'amélioration
Pour tout $s \in S$ **Faire**
$b \leftarrow \pi(s)$
$\pi(s) \leftarrow arg \max_{a \in A} \{\sum_{s' \in S} Pr(s, a, s') \cdot V(s')\}$
Si $b \neq \pi(s)$ **Alors** *politique_stable* $\leftarrow$ *faux*
Fin Pour
Jusqu'à *politique_stable=faux*
Retourner (π)

7.4.1 Complexité de l'algorithme

Chaque itération de l'algorithme se décompose dans les étapes d'évaluation puis d'amélioration. La première étape nécessite dans le pire des cas $|S|^3$ opérations, tandis que la seconde s'effectue en $|A||S|^2$ opérations. Littman [Littman95] montre que l'algorithme est polynomial selon $|S|$, $|A|$, γ et B, où B est le nombre de bits nécessaires à la représentation des données du problème traité.

7.5 L'algorithme Value Iteration

Malgré sa convergence vers une solution optimale, l'algorithme *Policy Iteration* a un point faible. En effet, chaque itération comprend l'évaluation d'une politique, ce qui exige plusieurs passages sur tous les états. Une amélioration de cet algorithme nécessite l'interruption de la phase d'évaluation sans perdre la garantie de la convergence de l'algorithme. L'une des méthodes permettant de réaliser cette rupture consiste à évaluer une fois pour toute la valeur $V(s), \forall s \in S$. Cette méthode est appelée *Value Iteration.* Elle peut être exprimée par la combinaison de la phase d'amélioration et de la phase d'évaluation interrompue. Une valeur $V(s)$ est calculée selon l'équation de Bellman (9). On conserve l'itération uniquement pour la phase d'amélioration. Comme dans l'algorithme *Policy Iteration*, la condition d'arrêt maintenue est la suivante : la différence entre l'ancienne valeur de $V(s)$ et celle obtenue par l'itération actuelle doit être inférieure ou égale au seuil minimal ϵ.

Algorithme 6.3

Données : Un MDP fini, un seuil minimal positif ϵ
But : Calculer une politique optimale avec une marge d'erreur $\leqslant \epsilon$

Initialisation quelconque de $V(s)$, comme $V(s) = 0, \forall s \in S$
Répéter
$\Delta \leftarrow 0$
Pour tout $s \in S$ **Faire**
$v \leftarrow V(s)$
$V(s) \leftarrow r(s) + \gamma \cdot \sum_{s' \in S} Pr(s, \pi(s), s') \cdot V(s')$
$\Delta \leftarrow max\{\Delta, |v - V(s)|\}$

Fin Pour
Jusqu'à $\Delta \leqslant \epsilon$
Retourner $\pi^*(s) = arg \max_{a \in A} \{\sum_{s' \in S} Pr(s, a, s') \cdot V(s')\}$

7.5.1 Complexité de l'algorithme

L'efficacité de l'algorithme *Value Iteration* dépend de deux facteurs, à savoir : la complexité d'une itération et le nombre d'itérations nécessaires pour converger. Chaque itération consiste à calculer la valeur de transition entre les états ; pour chaque action, cela nécessite $|A||S|^2$ opérations. Le nombre d'itérations nécessaires est plus difficile à déterminer. Littman [Littman95] avance les mêmes résultats quant au caractère polynomial de l'algorithme.

7.6 Problématique de la mise en oeuvre

Mettre on oeuvre un processus décisionnel de Markov nécessite la création des états (arbre de décision) et la définition d'une fonction de récompense. La création des états se fait à partir de la discrétisation de l'environnement dans lequel l'agent évolue ainsi que des états internes de l'agent. Les transitions entre les états correspondent à une distribution de probabilités $Pr(s, a, s')$ basée sur les actions que peut réaliser chaque agent. La fonction de récompense est adaptée au problème posé.

8 Modélisation du déplacement de l'agent mobile sous forme d'un MDP

Le MDP permet de choisir la politique optimale d'un agent évoluant dans un monde incertain. Nous allons utiliser cette approche pour la communication par agent mobile sur le réseau Internet. Prenons le cas d'une application répartie, qui nécessite une interaction entre *n* sites sur le réseau. Afin d'accomplir la tâche demandée par son client, l'agent mobile a besoin d'interagir avec les *n* sites de l'application. L'interaction peut se faire de deux façons : par communication distante ou locale. Dans le 1er cas, l'agent utilise l'approche client/serveur pour des interactions distantes tandis que dans la communication locale, l'agent migre sur le site distant.

8.1 Choix des actions

Pour la modélisation du problème sous forme d'un MDP, l'ensemble *A* des actions possibles que l'agent peut prendre est limité à deux ($|A|=2$). Ces deux actions sont :

1. Action *CS* : communication par mode « client/serveur » (distante).
2. Action *MA* : communication par mode « agent mobile » (locale).

Ces deux actions indiquent que l'agent interagit avec les *n* sites du réseau. Dans une application réelle, nous pouvons rencontrer une situation où l'agent n'a pas besoin de communiquer avec un site M_i donné (voir Figure 6.9). Pour gérer cette situation, nous intégrons l'action « sauter un état » (*skip*). La fonction *skip* indique que le nombre de messages échangés entre l'agent et le site M_i est zéro. Dans ce cas, l'agent n'aura pas besoin de visiter ce site et reste sur le premier site M_{i-1}. L'action *skip* est équivalente à

la réalisation de l'action *CS* avec une interaction *N=0*. Cette action n'est donc pas considérée parmi les actions possibles, elle est substituée par l'action *CS*.

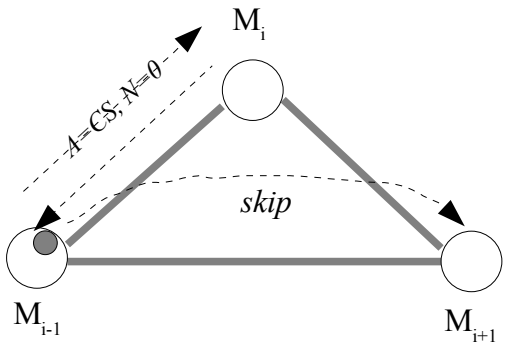

Figure 6.9 : Action skip

8.2 Fonction de récompense

Le but de l'utilisation de la technologie à agents mobiles est de réduire les délais d'attente du client en réduisant le temps de communication. La fonction de récompense dépend de ce temps.

Pour la communication entre deux machines, la connaissance du nombre de messages échangés N_i, de la taille de l'agent et du débit du lien utilisé permettent de calculer ce temps et ceci pour les deux modes de communication. Ce temps de communication représente la récompense que l'on attribue aux noeuds sur le graphe d'états. Le but du problème est de choisir une politique pour l'agent qui permet de minimiser le temps global de communication.

8.3 Construction du graphe d'états

Dans le problème du déplacement de l'agent, l'incertitude se traduit par la non connaissance de l'environnement. Le nombre des messages échangés entre les différents acteurs de l'application est inconnu. Une phase d'apprentissage sur les interactions dans le système permet de modéliser le comportement de l'application sous la forme d'un modèle probabiliste. Ce modèle probabiliste est utilisé pour la construction du graphe d'états de notre système. L'agent a besoin de visiter les n sites de l'application afin d'accomplir la tâche demandée.

À l'étape *i*, l'agent qui se trouve sur la $i^{ème}$ machine communique avec la $(i+1)^{ème}$ machine. Soit N_i le nombre de messages échangés à l'étape *i*, la connaissance de N_i et du débit sur le lien, permettent de calculer le temps de communications pour les deux actions *CS* et *MA*. Lorsque l'agent se trouve à l'étape *i*, il ne connaît pas le nombre de messages échangés à l'étape *(i+1)*. Ce nombre correspond au nombre de messages échangés entre la machine *(i+1)* et la machine *(i+2)*. Le nombre *N* de messages échangés est modélisé sous la forme d'un modèle probabiliste. Dans ce modèle, une probabilité p_j est attribuée à chaque valeur N_j. Soit *k* le nombre de possibilités que *N* peut prendre. Alors, pour chaque état de la $i^{ème}$ étape, il y a deux types d'actions (*CS* et *MA*), ce qui correspond à *2k* états dans la $(i+1)^{ème}$ étape lors de la construction du graphe (voir Figure 6.10).

Une fois le graphe d'états construit et la fonction de récompense attribuée à chaque état du système, le processus décisionnel de Markov permet de trouver la politique optimale de l'agent.

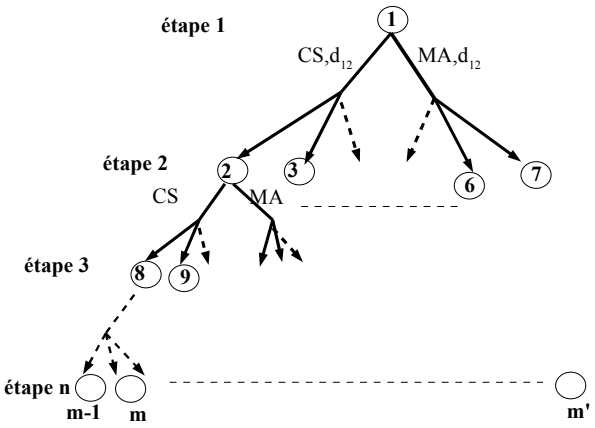

Figure 6.10 : Graphe d'états du système

8.4 Résolution du MDP

Le graphe d'états ne contenant pas des boucles (MDP à horizon fini), ceci nous permet de calculer la politique optimale avec un nombre d'itérations fini. Les algorithmes utilisés pour la résolution d'un MDP et la recherche de la politique optimale, cherchent à maximiser le gain. Dans le déplacement de l'agent mobile, le but est de minimiser le temps de calcul, ce qui nécessite une petite modification dans les deux algorithmes *Value Iteration et Policy Iteration.* Nous avons utilisé l'algorithme *Value Iteration* pour la recherche de la politique optimale. Cette modification consiste à remplacer la fonction *max* par la fonction *min*. L'algorithme converge très rapidement parce que le graphe d'états ne contient pas de boucles. Une fois l'utilité optimale connue, on peut aisément en déduire la politique optimale.

9 Application

9.1 Présentation de l'application

Une des applications les plus importantes dans le domaine des agents mobiles est la recherche d'informations sur le Web. Dans ces applications (recherche des hôtels, réservation d'un billet d'avion, etc...), des agents se déplacent sur différents sites pour rechercher des informations pour leurs clients.

Notre exemple représente une application répartie sur Internet dont le but est de chercher une liste de restaurants dans une ville donnée. Dans notre scénario, trois bases de données doivent être consultées pour avoir ces informations (voir Figure 6.11). La première recense des restaurants et permet d'en obtenir la liste dans une ville. La deuxième est un annuaire permettant d'obtenir des informations sur les restaurants (numéros de téléphone, adresses, etc...). Une fois les adresses obtenues, un troisième site est consulté afin d'obtenir le plan d'accès des restaurants. Ces trois bases de données sont gérées sur des sites différents par des administrations ou des entreprises

différentes (par exemple un office de tourisme, une compagnie de téléphone et un site de plan d'accès).

Les trois serveurs fournissent chacun une interface correspondant aux services qu'ils offrent. Pour le premier serveur, nous supposons que la requête du client consiste à donner le nom de la ville, le serveur lui retourne la liste des restaurants dans la ville. Pour le deuxième serveur, nous supposons que l'interface permet au client de donner le nom d'un seul restaurant, le serveur retournant le numéro de téléphone associé à ce nom. Pour le troisième site, nous supposons que l'interface permet au client de donner l'adresse d'un seul restaurant, le serveur retournant le plan d'accès du restaurant.

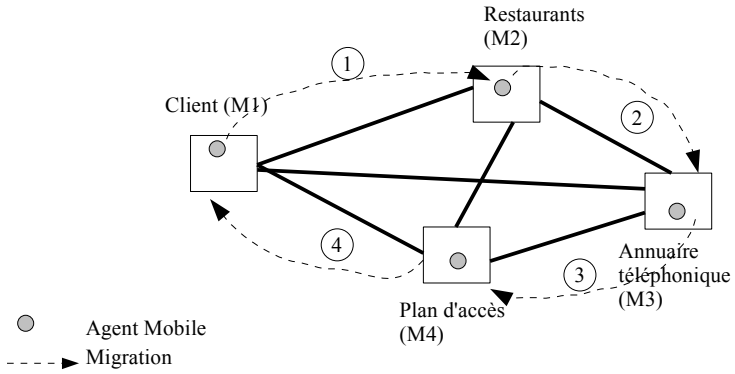

Figure 6.11 : Recherche de restaurants dans une ville

En utilisant le modèle « client/serveur » classique, le client doit faire un appel à distance pour interroger le premier serveur et récupérer la liste des restaurants dans une ville. À partir de cette liste, le client va ensuite, pour chaque restaurant, réaliser un appel à distance au second serveur afin de récupérer le numéro de téléphone et l'adresse du restaurant. Pour chaque restaurant, le client effectue ensuite un appel au troisième serveur afin d'obtenir le plan d'accès au restaurant.

Les trois serveurs M_2, M_3 et M_4 fournissent chacun une interface générique composée d'opérations élémentaires. Les clients peuvent ainsi utiliser le service selon leurs besoins. Cependant, cela se traduit par de nombreuses interactions entre le client et le serveur M_3 afin d'obtenir le numéro de téléphone et ceci pour chaque restaurant dans la liste. De même pour le client et le serveur M_4 afin d'obtenir le plan d'accès.

Une seconde solution, dans le cas de la recherche des numéros des téléphones par exemple, consiste à transmettre directement la liste des restaurants du serveur M_2 au serveur M_3. Ainsi, il faut installer un nouveau service sur les deux machines M_2 et M_3 afin de réaliser cette jointure, ce qui implique l'ajout d'un service pour chaque besoin spécifique d'un client. Une extension statique des serveurs (par les administrateurs des deux serveurs) n'est pas une solution réaliste dans la mesure où ces bases de données sont administrées séparément. De plus, le fait d'etendre un serveur pour chaque besoin spécifique d'un client ne nous semble pas pertinent. Nous proposons donc d'utiliser une spécialisation dynamique de serveurs par des agents mobiles.

Le client (voir Figure 6.11) crée un agent mobile contenant la tâche globale à réaliser. Cet agent se déplace (1) tout d'abord sur la machine M_2 et réalise localement la recherche de la liste des restaurants dans la ville. Cette liste est transférée avec l'agent mobile lors de son déplacement vers le serveur M_3 (2). Sur la machine M_3, l'agent

interagit localement afin d'obtenir le numéro de téléphone et l'adresse de chaque restaurant, ces informations sont transférées avec l'agent qui se déplace sur la machine M_4 (3). Sur la machine M_4, l'agent interagit localement afin d'obtenir le plan d'accès et ceci pour chaque restaurant. Dans l'objectif de minimiser la quantité d'informations transférées, l'agent mobile regroupe les plans d'accès sur une seule carte, pour des restaurants proches. Après son interaction avec la machine M_4, l'agent revient vers son client (4). Nous pouvons toujours envisager d'envoyer les informations collectées sans effectuer le déplacement de l'agent. Dans un souci de généralisation, nous ne traitons pas cette option.

Comme on le voit à travers cet exemple, l'utilisation d'agents mobiles permet d'étendre dynamiquement l'interface des serveurs. Les agents ont été utilisés ici pour permettre une redirection de requêtes entre des bases de données différentes.

On peut améliorer le dispositif en utilisant un agent mobile intelligent (solution hybride). Dans cette solution, cet agent, crée par le client, ne se déplace pas systématiquement sur un serveur ; il peut effectuer des interactions distantes avec ce dernier. Le choix de la politique que l'agent adoptera est basé sur l'étude théorique effectuée dans ce chapitre.

Dans le but de montrer l'efficacité de la solution basée sur le MDP, nous avons fait une comparaison entre cette solution et les autres présentées dans ce chapitre. Les différents résultats obtenus sont présentés ci-après.

9.2 Résultats

Notre application tourne dans un environnement dynamique et incertain. L'incertitude dépend de deux aspects : l'incertitude sur le débit des liens utilisés et l'incertitude sur les interactions entre les différents serveurs participant à l'application. La résolution par le MDP nécessite la modélisation des interactions dans le système, ce qui nécessite une phase d'apprentissage. Au début du lancement de l'application, une période d'apprentissage est observée dans le but de construire le modèle d'interaction. Pendant cette période le modèle est ajusté afin de se rapprocher du modèle réel. Une fois ce modèle construit, nous pouvons l'utiliser afin de calculer la politique de déplacement de l'agent.

Dans cette application, un client lance sa demande et attend la réponse envoyée par le système. Nous avons mesuré le temps écoulé entre cet envoi et la réception de la réponse. Nous avons obtenu les résultats présentés dans la Figure 6.12, où l'axe des x représente l'évolution du système dans le temps et l'axe des y représente le délai moyen d'attente du client (en ms). Cette évolution correspond au changement de débit sur les liens du réseau. Au cours de cette évolution, des clients consultent la machine M_1 afin de chercher des informations sur les restaurants dans une ville donnée.

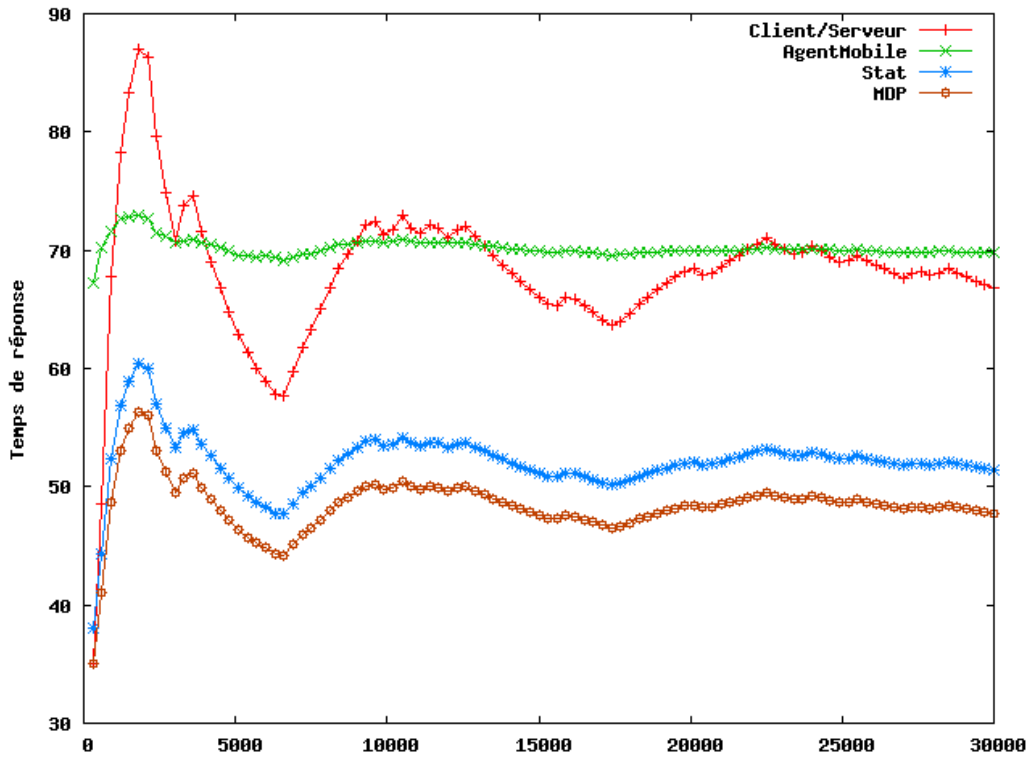

Figure 6.12 : Communication hybride

Les deux premières courbes (en haut de la Figure 6.12) représentent les temps moyens de communication respectivement par mode « agent mobile » et « client/serveur ». Nous constatons qu'avec l'évolution du système (changement des débits), l'avantage de l'un des deux modes par rapport à l'autre est alterné.

La troisième courbe représente le temps moyen de la réponse dans le cas d'une communication hybride utilisant la méthode statistique afin de choisir la politique de l'agent. Tandis que la quatrième courbe représente le temps moyen de la réponse dans le cas de la communication hybride, le choix de la politique de l'agent étant fait en utilisant le processus décisionnel de Markov. Nous constatons que l'utilisation d'un MDP permet de répondre au client plus rapidement tout en minimisant l'utilisation du réseau. Il faut signaler qu'avec l'augmentation de nombre de machines visitées, la solution basée sur le MDP devient plus avantageuse puisque c'est la seule solution qui prévoit une politique qui utilise les informations acquises au cours de l'avancement de l'agent dans son environnement.

La phase d'apprentissage n'est pas représentée sur ce schéma et les courbes sont tracées après cette période d'apprentissage. Dans la section suivante, nous proposons une amélioration de cette solution grâce à une migration intelligente du code de l'agent.

10 Migration intelligente du code de l'agent

La performance de la migration hybride dépend de la taille de l'agent qui se charge de la réalisation de la tâche demandée. Cette taille dépend du code de l'agent ainsi que de la taille des données transmises avec l'agent. Dans [Gavalas04], l'auteur indique

quelques recommandations permettant au programmeur de réduire la taille de l'agent. Il traite la question du transfert du code de l'agent. Ce transfert est nécessaire lorsque l'agent visite le site pour la première fois. En cas de visites multiples du même site, nous pouvons envisager de stocker le code de l'agent sur le site visité permettant ainsi son utilisation ultérieure. La technologie Java [Java] offre cette possibilité et *ClassLoader* permet de stocker les classes localement, pendant un certain temps, ce qui permet d'éviter le chargement multiple du code.

La compression des données transférées permet aussi de réduire la taille de l'agent. En plus, le programmeur est invité à respecter certaines consignes lors du développement de l'agent :

- opter pour l'utilisation de noms courts pour les variables et les classes plutôt que des noms longs,
- opter pour l'utilisation de structures simples pour les types des données : le type *int* plutôt qu'une instance de la classe *Integer,* un tableau plutôt qu'une instance de la classe *Vector,* etc...

Tracy [Braun05] est une plate-forme qui permet d'avoir plusieurs stratégies pour la migration de l'agent. Dans Tracy, un agent est défini sous forme de composants. Chaque composant représente un savoir-faire de l'agent. En plus de la migration classique, cette plate-forme permet à l'agent mobile de charger son code à la demande. Dans la plupart des applications, un agent donné n'a pas besoin de transmettre la totalité de son code sur tous les sites visités. L'agent charge ainsi le composant qui lui manque afin de réaliser la tâche demandée sur un site donné. Dans [Erfurth01], l'auteur utilise cette plate-forme pour effectuer ces tests. Il présente un classificateur qui permet de minimiser le coût de la migration de l'agent. Ce classificateur est intégré au niveau de ses agents afin d'ajouter de l'intelligence à la migration du code de l'agent.

Dans l'application que nous présentons, l'utilisation d'un simple *ClassLoader* permet de charger le code de l'agent au lancement de l'application. Ceci permet de négliger la taille du code de l'agent sauf pour la première demande faite par l'utilisateur. La Figure 6.13 représente les résultats obtenus avec et sans la migration intelligente du code de l'agent.

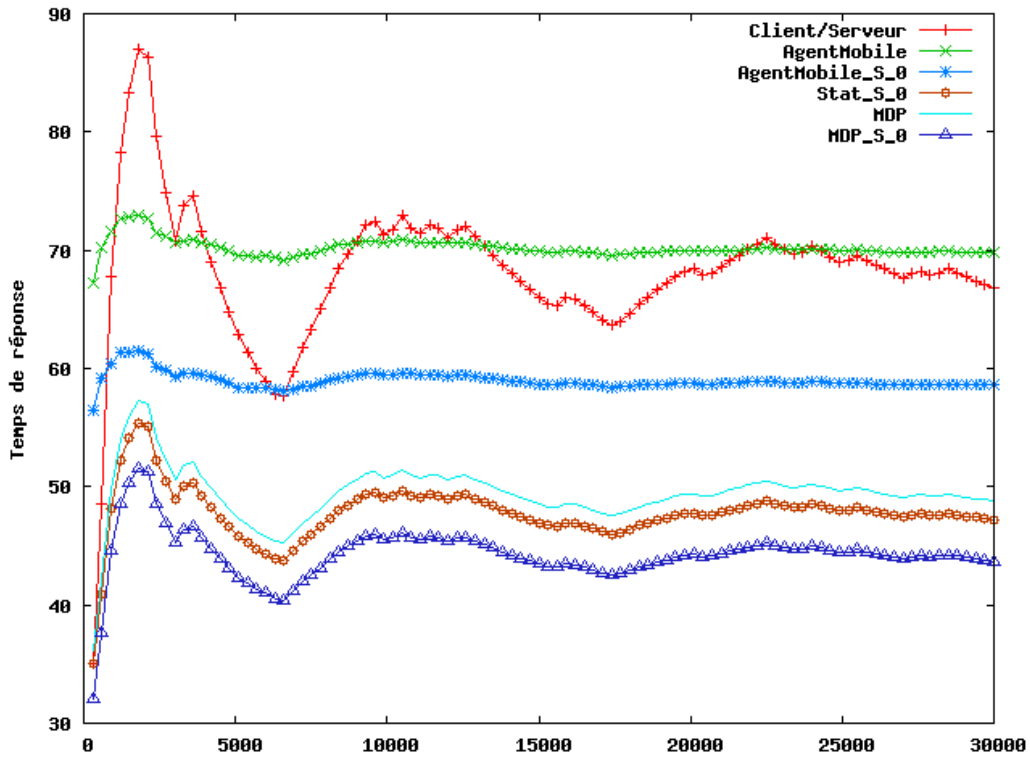

Figure 6.13 : Migration avec un code de taille zéro

À noter que l'influence de la taille du code de l'agent sur la solution basée sur le MDP n'est pas considérable. En effet dans l'approche hybride, la communication par mode client/serveur est choisie lorsque la taille de l'agent est grande par comparaison à la taille des données transmises. Ceci explique les résultats obtenus avec une plate-forme qui permet de faire une migration intelligente du code de l'agent.

11 Décharge intermédiaire de l'agent

Au cours de son déplacement sur le réseau, l'agent mobile collecte des données sur les différentes machines. Ces données sont divisées en deux catégories : des données intermédiaires et des données finales. Les données intermédiaires représentent les informations collectées par l'agent et qui lui seront utiles dans le reste de son itinéraire (les noms de restaurants). Les données finales représentent les informations demandées par le client mais qui ne seront pas utilisées pour les prochaines interactions (les numéros de téléphones). Pour ce dernier type de données, il est utile de s'interroger sur l'utilité de les transférer avec l'agent ou de les envoyer directement au client.

Avec l'augmentation du nombre des machines visitées, la quantité d'informations transportées avec l'agent augmentera sa taille, pénalisant d'autant l'utilisation de la technologie d'agents mobiles. Dans la section 3 de ce chapitre nous avons montré l'influence du nombre de messages échangées sur le choix de l'une des deux technologies. Ainsi, un seuil N est défini à partir duquel la taille de l'agent mobile est compensée par la quantité d'informations échangées. C'est à partir de ce seuil que nous aptons pour l'un des deux modes.

La taille de l'agent mobile augmente avec le nombre de machines visitées. Afin de montrer l'influence de ce nombre sur les performances de la technologie d'agents mobiles, nous reprenons l'exemple de la gestion d'un parc informatique. Dans cet exemple nous considérons que le nombre de fichiers à vérifier sur chaque machine est égal à 12 ($N = 12$). Ce nombre correspond au seuil à partir duquel la communication par agent mobile devient plus avantageuse (voir section 2.4). Les deux technologies sont comparées en faisant varier le nombre des machines visitées. Les deux courbes (voir Figure 6.14) représentent le temps de réponse (en ms) pour les deux technologies « agent mobile » et « client/serveur ». Pour un faible nombre de machines (<3), la technologie « agents mobiles » est toujours efficace, mais avec l'augmentation de ce nombre, la quantité d'informations transportées avec l'agent devient très grande, pénalisant ce mode.

La communication par « agent mobile » est plus efficace, par comparaison à la communication « client/serveur », lorsque le nombre de machines visitées par l'agent est compris entre une borne inférieure et une borne supérieure. Au delà de cette borne supérieure, la taille de l'agent mobile augmente de telle façon que la migration de l'agent devienne plus coûteuse. Afin de résoudre ce problème nous envisageons de décharger l'agent de certaines informations réduisant ainsi sa taille pour le reste de son itinéraire. L'agent envoie une partie des données collectées ; il s'agit des données finales qui ne sont pas utiles pour la suite de son itinéraire.

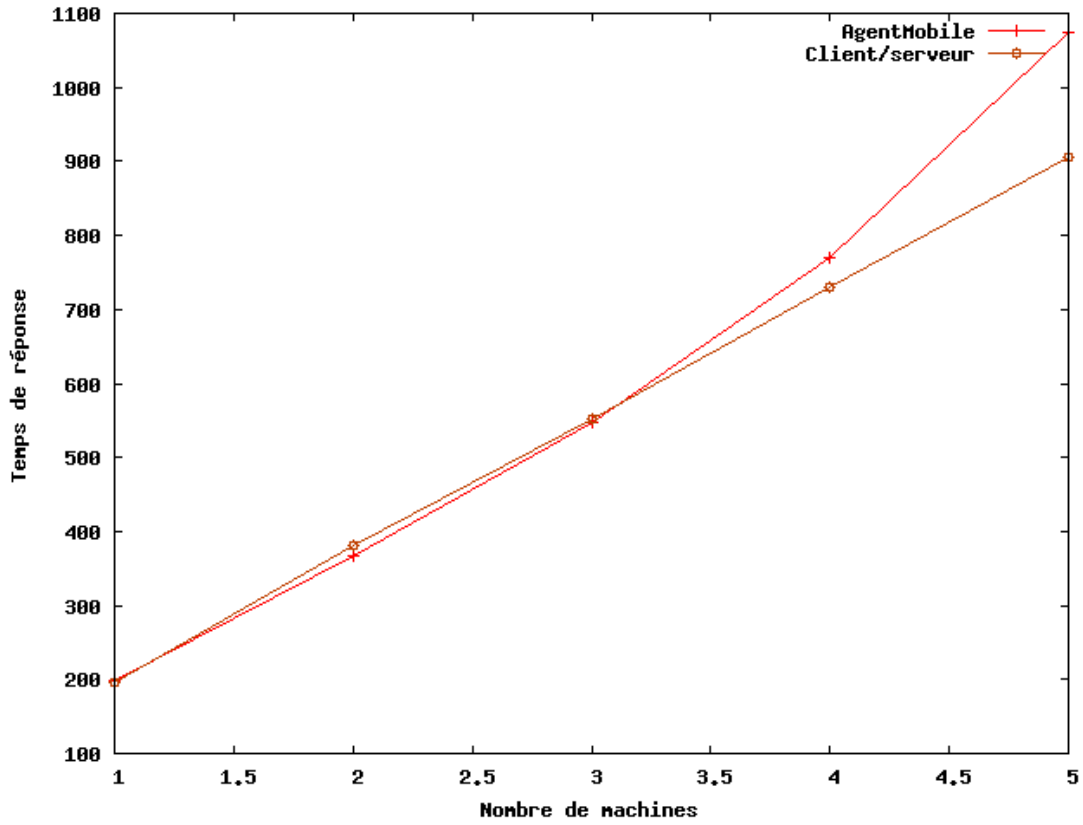

Figure 6.14 : Influence de la quantité d'informations transportées sur la performance de la communication par agent mobile

11.1 MDP et décharge de l'agent mobile

Dans la recherche de la politique optimale (§8.1), l'agent a deux types d'actions :

1. Action *CS* : communication par mode « client/serveur » (distantes).
2. Action *MA* : communication par mode « agent mobile » (locales).

Les résultats obtenus dans la section précédente montre l'influence de la taille des données transportées sur l'efficacité de l'utilisation de la technologie à « agent mobile ». Afin de diminuer la taille de l'agent mobile, nous considérons une troisième action que l'agent puisse appliquer ; il s'agit de le décharger (Action *DA*). L'agent va, dans ce cas, envoyer les données finales à son client, lui permettant de diminuer sa taille.

Lors de la construction de la politique de l'agent, trois types d'actions sont à considérer (voir Figure 6.15). Ainsi, à l'étape *i*, en plus des deux actions *MA* et *CS*, l'agent peut choisir de décharger les données qui ne seront pas utilisées dans la suite de son itinéraire, une fois ces données envoyées à l'utilisateur, l'agent a la possibilité de choisir entre l'une des deux actions (*MA* ou *CS*).

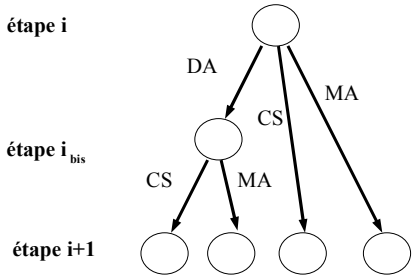

Figure 6.15 : Trois types d'actions

11.2 Résultats

Prenons l'exemple de la recherche de restaurants dans une ville donnée. Cet exemple permet de montrer l'influence de l'intégration de cette nouvelle action (décharger l'agent) lors de la modélisation (sous forme d'un MDP) du déplacement de l'agent mobile.

Dans cette application, l'agent interroge la deuxième machine afin d'obtenir la liste des restaurants. Cette information est transportée avec l'agent et lui sera utile dans le reste de son itinéraire. L'agent utilise cette liste pour ses interactions avec la troisième machine ; pour chaque restaurant, il donne le nom afin d'obtenir son numéro de téléphone ainsi que son adresse. Le numéro de téléphone est une information finale pour le client et ne sera pas utilisée pour la suite de ses interactions. En effet, sur la quatrième machine, l'obtention du plan d'accès d'un restaurant donné nécessite la connaissance de son nom et son adresse, et en aucun cas du numéro de téléphone du restaurant. La liste du numéro de téléphone ne sera donc pas utilisée pour la suite de l'itinéraire de l'agent et ce dernier aura donc la possibilité de l'envoyer à son client avant même de visiter la quatrième machine.

L'intégration de cette nouvelle action pour la recherche des restaurants dans une ville permet d'améliorer la performance de l'application et de minimiser le délais

d'attente du client (voir Figure 6.16).

Sur cette figure, l'axe des abscisses représente l'évolution du système dans le temps. Cette évolution correspond au changement de débit sur les liens du réseau. Au cours de cette évolution des clients consultent la machine M_1 afin de chercher des informations sur les restaurants dans une ville donnée. L'axe des ordonnées représente le délai moyen d'attente du client (en ms).

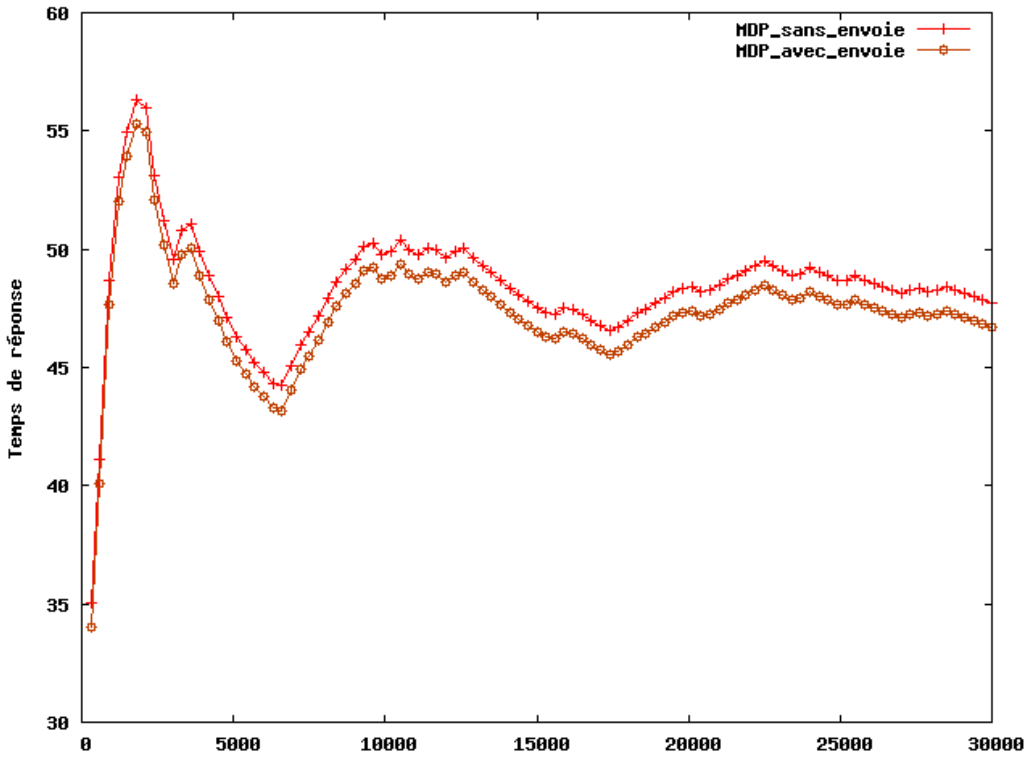

Figure 6.16 : MDP avec envoie des données au client en cours de déplacement

La première courbe en haut du schéma représente le délai d'attente (temps moyen) du client entre l'envoi de sa requête et la réception de la réponse dans le cas d'un agent mobile n'ayant pas la possibilité de se décharger. Pour la politique de l'agent, il aura la possibilité de choisir entre deux types d'actions seulement (*CS* ou *MA*).

La deuxième courbe représente le délai d'attente du client dans le cas d'un agent mobile avec la possibilité de se décharger. L'agent a donc la possibilité d'envoyer la liste des numéros de téléphone avant même d'interroger la quatrième machine. Concernant la politique adoptée, l'agent aura la possibilité de rajouter aux deux actions (*CS* et *MA*) déjà exisrantes, l'action *DA*.

Nous pouvons remarquer que l'intégration de cette nouvelle action lors de la modélisation sous forme d'un MDP permet d'améliorer les performances de l'application, surtout dans le cas d'une application qui nécessite la visite d'un grand nombre de machines. La politique optimale est obtenue en résolvant le MDP. Cette politique indique à l'agent comment se déplacer sur le réseau mais également le bon endroit où il se décharge des informations finales.

12 Conclusion

Dans ce chapitre, nous avons montré l'efficacité de l'utilisation de l'approche « agents mobiles » pour la programmation répartie dans un contexte de forte interaction entre les entités en jeu. Elle permet, en effet, de minimiser la bande passante utilisée et de réduire la durée de la connexion (le client a besoin d'avoir une seule connexion pour envoyer l'agent et pour recevoir les résultats). Elle permet aussi de réduire le temps de l'exécution de l'application répartie. Ce modèle permet aussi d'augmenter la fiabilité du système dans la mesure où les requêtes deviennent des agents autonomes.

La quantité d'informations échangées entre les différents acteurs, le type de l'application ainsi que les caractéristiques du réseau sont des paramètres déterminants pour le choix de la technologie à mettre en oeuvre. Nous avons montré que l'approche « client/serveur » est bien adaptée quand il s'agit d'une faible quantité d'informations sur une courte durée tandis que l'approche « agents mobiles » permet de gérer efficacement les interactions avec un élément du réseau pendant une longue durée. Dans ce cas, les échanges se font localement, ce qui permet de réduire le trafic sur le réseau. Nous avons montré que, dans certains cas, l'approche « agent mobile » paraît indispensable, par exemple dans le cas des machines mobiles qui ne sont pas connectées en permanence.

Afin de profiter des deux modèles, nous avons proposé de communiquer par un modèle hybride. Après la phase d'apprentissage et la construction d'un modèle probabiliste qui contient les différents scénarios que l'application adoptera, ce modèle probabiliste est utilisé pour la construction du graphe d'états du système. Le processus décisionnel de Markov nous permet de calculer la politique optimale de déplacement de l'agent. Cette politique est appliquée par l'agent mobile afin de réduire le délai d'attente du client et le trafic réseau. Comme nous l'avons illustré dans l'application sur la recherche des restaurants dans une ville, l'ajout de l'action qui consiste à décharger l'agent mobile en cours de son déplacement et d'envoyer les données utiles au client améliore la solution proposée.

Conclusion et Perspectives

Conclusion et Apports

Au cours de cette thèse, nous nous sommes intéressés à la technologie d'agents mobiles et à son utilisation dans le domaine de la programmation répartie. Cette technologie permet la construction d'applications flexibles et "dynamiquement" adaptables aux contraintes de l'environnement d'exécution ou de l'application elle-même. La mobilité permet d'adapter les services distants aux besoins des clients ; c'est une forme d'application répartie adaptable. La mobilité du code réduit la dépendance entre le programme et le site d'accueil et offre un premier degré de décentralisation des activités et une forte flexibilité pour le système [Arcangeli02].

La décision de migration peut se faire à l'initiative de l'agent lui-même, de manière autonome, elle est ainsi contrôlée par l'application et non par le système d'exécution. Le but du déplacement est généralement d'accéder localement à des données ou à des ressources initialement distantes, d'effectuer le traitement en local et de ne déplacer que les données utiles. Nous avons distingué deux rôles essentiels pour un agent mobile : les échanges de données et l'agent mobile calculateur. L'agent a été étudié sous ces deux angles ; des solutions pour son déplacement ont été apportées afin d'augmenter les performances d'une application répartie. Un agent mobile est souvent exécuté dans un environnement d'exécution qui lui offre des services pour son déplacement, sa localisation et la communication avec les autres agents. La performance et la fiabilité de l'environnement d'exécution influencent beaucoup sur la qualité d'une application basée sur la technologie à « agents mobiles ». Lors de l'expérimentation sur des applications réelles ; nous avons choisi une plate-forme qui ne pénalise pas nos algorithmes d'une manière considérable.

Les aspects communication et calcul ont été étudiés séparément. Dans un premier temps nous avons consideré l'agent comme une entité de calcul à part entière. Dans une application distribuée, la recherche d'un maximum de puissance passe nécessairement par une répartition et un équilibrage de charge efficace. Ceci est d'autant plus vrai lorsqu'on se trouve dans un contexte hétérogène. Nous avons étudié les différents algorithmes d'équilibrage de charge et nous avons présenté notre architecture basée sur la technologie à « agents mobiles ». L'équilibrage de charge se fait par un algorithme itératif et la bonne distribution se fait de proche en proche entre les machines. L'avantage de l'architecture proposée est qu'elle est complètement décentralisée et la prise de décision s'effectue en fonction de la charge locale de la machine ce qui permet de réduire les échanges d'informations nécessaire à la prise de décision. Un autre point fort dans cette architecure vient de l'utilisation d'un agent collecteur, le but de l'agent

collecteur est d'explorer le réseau afin de construire une vision globale sur la charge du système. Cet agent collecteur va communiquer ses connaissances aux différentes machines visitées afin de guider la prise de décision locale, la vision sur la charge globale construite par l'agent collecteur permet de définir les seuils dynamiques utilisés par le processus de migration. Il faut signaler que l'architecture proposé ne dépend pas des caractersistiques des problèmes étudiés ce qui la rend facile à intégrer dans une application réelle.

Pour valider notre architecture, nous avons appliqué l'algorithme d'équilibrage de charge sur un problème concret. Ceci nous a permis de détailler les différentes étapes nécessaires à la distribution d'une application et au déploiement d'un algorithme d'équilibrage de charge. Le problème choisit représente une simulation de l'évolution d'une colonne à distiller ; l'utilité d'une telle simulation est le contrôle d'une colonne réelle de grande taille nécessitant un système rapide. Cette application nous permet de faire emerger les différents problèmes rencontrés lors de la modélisation et la mise en place d'une application réelle. Les différentes entités de l'application n'évoluent pas de la même manière, ainsi certaines machines sont plus chargées que d'autres. Une telle caractéristique est indispensable pour tester notre algorithme d'équilibrage de charge dynamique.

Nous avons effectué des expérimentations en modifiant la taille du problème, c'est-à-dire en augmentant le nombre de plateaux de la colonne. Les résultats de ces expérimentations montrent que l'apport de l'équilibrage de charge est d'autant plus important que la taille du problème est grande. De manière plus générale, la mise en place de notre architecture sur une application réelle montre l'avantage de l'intégration d'un agent collecteur pour la construction d'une vision globale dans le cas d'un système non équilibré au lancement ou dans les cas où la charge supplémentaire (causée par les autres applications) des machines aura évolué. Ces résultats sont directement transposables dans d'autres applications qui possèdent certaines caractéristiques identifiées dans l'exemple de la colonne à distiller. Citons par exemple, le fait que les tâches ne prennent pas toutes le même temps d'exécution, induisant une utilisation non équitable des ressources disponibles.

Le second aspect des « agents mobiles » que nous avons étudié traite leur utilisation comme une entité de communication. Il s'agit ici de réduire le trafic sur le réseau et de diminuer la quantité d'informations échangées ; dans ce cas l'agent se déplace vers la source d'informations et effectue des échanges locaux, il s'agit d'une spécification au niveau du serveur. Cette dernière peut être rentable sous certaines conditions, en particulier, avec des débits faibles et des interactions fréquentes entre le client et le serveur.

Nous avons étudié l'utilisation de la technologie d'« agents mobiles » dans le domaine de la recherche d'informations sur le web. Un agent mobile est créé par le client qui lui indique la tâche à effectuer. Cet agent se déplace entre les différentes machines de l'application, sur chaque machine, il effectue des échanges et filtre les informations collectées. Ce filtrage permet de réduire la quantité d'informations transportées avec l'agent et par conséquent le trafic sur le réseau. L'agent transporte avec lui uniquement les données demandées par son client et les informations qui lui seront utiles dans le reste de son itinéraire. Dans le cas de la recherche d'information sur le web, la taille de l'agent mobile ajoute une surcharge lors de la communication. Cette taille est souvent compensée par le fait que l'agent effectue des interactions locales.

Afin de réduire l'influence de la taille de l'agent sur la solution proposée, nous avons choisit de ne pas le deplacer que lorsque c'est plus avantageux. Un modèle de communication hybride est proposé dans ce travail. L'avantage de ce modèle est que l'agent choisit entre les deux modes de communication afin de réduire le délai d'attente du client. La communication hybride permet à l'agent de s'adapter à son environnement afin de mieux servir le client, minimiser le delai d'attente. L'étude théorique de deux modèles de communication montre que le choix de la politique adoptée par l'agent dépend du débit sur les liens, de la taille de l'agent et des interactions entre les différents sites du réseau. Nous avons proposé un algorithme qui permet d'identifier la politique optimale dans le cas d'un environnement déterministe c'est-à-dire le cas où les interactions entre les différentes entités sont connues par l'agent.

Dans le cas d'un environnement incertain, le choix de la politique de l'agent est beaucoup plus complexe. Afin de résoudre ce problème, nous avons modélisé le déplacement de l'agent sous la forme d'une chaîne de Markov. Ceci nous a permis de trouver la politique optimale pour le déplacement de l'agent. Dans cette politique, un agent peut entreprendre deux types d'actions : une communication distante ou une migration vers la source d'information. Après avoir effectué des tests sur une application réelle, nous avons remarqué que lorsque la réalisation de la tâche demandée par l'agent nécessite la visite d'un grand nombre des machines, la taille des données transportées influence la qualité de la migration. Afin de minimiser l'influence de la taille de l'agent nous avons ajouté un troisème type d'action que l'agent peut effectuer. Cette nouvelle action consiste à décharger l'agent de données non utilisées, pendant son déplacement. Ainsi, le processus décisionnel de Markov permet de trouver le bon endroit pour décharger l'agent de ces informations. La mise en oeuvre sur le problème de la recherche d'information sur le web, montre que, le modèle proposé permet de réduire conjointement le délai d'attente et le trafic sur le réseau. Finalement, l'utilisation du processus décisionnel de Markov pour la recherche de la politique du déplacement de l'agent nous permet d'affirmer que cette politique est optimale.

Les différents résultats obtenus au cours de cette thèse nous permettent de conclure sur les contributions que nous apportons au domaine de la programmation répartie. Nous soulignons aussi les apports significatifs de la technologie d'« agents mobiles » dans l'amélioration des performances d'une application distribuée. Ensuite, nous présentons les différents axes d'étude qui s'ouvrent à la suite de ce travail. Les travaux présentés dans cette thèse ont fait l'objet de publications dans des conférences nationales et internationales [Falou03a, Bourdon03, Falou04a, Falou04b, Falou05a, Falou05b, Falou06].

Perspectives

Nous pensons que la technologie à « agents mobiles » jouera un rôle très important dans la conception des applications réparties. De nombreux axes peuvent être émergé à partir de résultats de recherche.

Recherche d'information

Nous pensons que l'intégration des deux dimensions (communication et calcul) dans une application donnée permet de mieux exploiter les agents mobiles. Par exemple, dans le domaine de la recherche d'informations sur le web, nous envisagons une application qui contient des serveurs dupliqués qui rendent tous les mêmes services.

Une amélioration de la performance nécessite que les différents serveurs soient sollicités de la même manière. Nous pensons qu'une extension de la solution basée sur le processus décisionnel de Markov avec intégration de la charge des différentes machines permet de guider les agents mobiles dans le choix de l'itinéraire et des serveurs à visiter. Cette politique permet de minimiser le délai d'attente d'un client et d'avoir un système bien équilibré.

Dans la recherche d'informations sur le web, un seul agent mobile est crée. Il est chargé de se déplacer pour réaliser la tâche demandée par le client. Nous pensons que l'utilisation de plusieurs agents mobiles pour la recherche d'informations permet d'améliorer la qualité de la solution proposée et de réduire le délai d'attente du client. Deux cas sont à envisager : 1) les agents se connaissent et peuvent communiquer entre eux afin de réaliser la tâches demandées ; on parlera d'agents coopératifs. 2) Les agents ne se connaissent pas et dans ce cas, chaque agent réalise sa tâche indépendamment des autres agents. Des travaux sur le DEC-MDP [Beynier05] permettent de trouver la politique optimale de plusieurs agents mobiles coopératifs.

La recherche de la politique optimale de l'agent nécessite la modélisation des interactions entre les différentes entités de l'application. La solution proposée nécessite l'observation d'une phase d'apprentissage afin de construire ce modèle. Nous pensons que cette phase mérite d'être mieux étudiée. Après la modélisation du système, la politique optimale de l'agent est calculée hors-ligne et elle sera appliquée par l'agent. Nous proposons d'intégrer un système d'ajustement pour le modèle d'interaction, dans ce cas, si l'on détecte un grand changement dans le comportement du système, la politique de déplacement de l'agent est recalculée.

Équilibrage de charge

Dans le domaine de l'équilibrage de charge, l'architecture que nous avons proposée intègre un agent collecteur pour la construction d'une vision globale du système. Le but de l'agent est de se déplacer sur les différentes machines afin de construire cette vision sur la charge globale. En arrivant sur une machine donnée, il informe son agent moniteur sur la charge assurée afin d'ajuster les seuils. Dans nos travaux, l'agent mobile collecteur se déplace d'une façon aléatoire. Nous pensons que cette question mérite d'être plus étudiée. Il faudrait intégrer un mécanisme qui permette à l'agent de construire cette vision à moindre coût en améliorant son déplacement. En plus, l'agent collecteur doit informer les agents moniteurs de la charge globale du système. Un mécanisme qui permet de se focaliser uniquement sur les machines les plus utilisées reste une question ouverte à étudier.

Un seul agent collecteur a été proposé ; l'augmentation du nombre d'agents collecteurs permettrait, dans certains cas, l'amélioration de l'équilibrage de charge. Dans ce cas, un mécanisme de collaboration entre les différents agents collecteurs est envisageable. Le nombre d'agents collecteurs à utiliser dépend de la taille du réseau et des caractéristiques de l'application étudiée. A notre avis, la détermination de ce nombre mérite d'être étudiée.

L'architecture que nous avons proposée ne dépend pas du réseau utilisé ; ce qui nous amène à avancer que son utilisation sur un réseau dynamique soit possible. Un réseau dynamique est un réseau dans lequel certains liens de communication peuvent être temporairement perdus ou surchargés [Bahi05a]. Nous pensons que les algorithmes asynchrones [Bahi05b] seront mieux adaptés dans ce type du réseau.

La prise de décision dépend des seuils dynamiques et la réactivité du système dépend de ces seuils. La définition des seuils nécessite la quantification de la charge engendrée par l'exécution d'un agent. Ces agents s'exécutent sur des machines hétérogènes et il est certain que ce point mérite d'être approfondie. Il faut prévoir un mécanisme qui permet la quantification du besoin d'un agent mobile en matière de charge machine ainsi que la quantification de la capacité et de la charge effective d'une machine.

Dans une application distribuée asynchrone, il faut détecter la terminaison de l'application. Différentes études [Bahi05b] proposent des algorithmes de détection de la convergence pour les contextes asynchrones. Notre architecture doit intégrer de tels algorithmes.

Malgré les différentes limites abordées, et les multiples perspectives ouvertes, nous espérons que ce travail aura convaincu le lecteur de l'utilité des mécanismes basés sur les agents mobiles pour la réalisation d'applications réparties sur Internet.

Bibliographie

[Aglets] IBM Corporation, Aglets Software Development Kit , http://www.trl.ibm.co.jp/aglets/

[Aknine01] Samir Aknine and Suzanne Pinson, Négociation multi-agent: analyse, modèles, expérimentation. *Intelligence Artificielle, RIA*, pp.173--217, 2001

[Alouf02] Sara Alouf and Fabrice Huet and Philippe Nain, Forwarders vs. Centralized Server: An Evaluation of Two Approaches for Locating Mobile Agents. *Proceedings of the 2002 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS-02)* pp.278--279, 2002

[Amdahl67] Amdahl, G. M., The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. *AFIPS Conf. Proc.*, pp.483--485, 1967

[Ametller04] Joan Ametller and Sergi Robles and Jose A.Ortega-Ruiz, Self-Protected Mobile Agents. *AAMAS* pp.362--367, 2004

[Arcangeli01] Jean-Paul Arcangeli and Christine Maurel and Frédéric Migeon, An API for high-level software engineering of distributed and mobile applications. *8th IEEE Workshop on Future Trends of Distributed Computing Systems , Bologna (It.)* pp.155--161, 2001

[Arcangeli02] Jean-Paul Arcangeli, Abdelkader Hameurlain, Guy Bernard, Jean-Francois Monin, eds., Agents et code mobiles. *Numéro thématique de la Revue des sciences et technologies de l'information, série Technique et science informatiques (RSTI-TSI),* Hermès Science Publications - Lavoisier, Paris, Vol 21 N°6, 2002

[Arcangeli04] Jean-Paul Arcangeli, Vincent Hennebert, Sébastien Leriche, Frédéric Migeon, Marc Pantel. JavAct 0.5.0 : principes, installation, utilisation et développement d'applications. Rapport de recherche, IRIT/2004-5-R, IRIT, février 2004

[Leriche04] Sébastien Leriche, Jean-Paul Arcangeli. Une architecture pour les agents mobiles adaptables. *Actes des Journées Composants JC'04, Lille*, mars 2004

[Authie94] Authie G., Ferreira A., Roch J.L, Villard G., Roman J., Roucairol C., Virot B., Algorithmes paralleles : analyse et conception, 1994

[Bahi05a] Jacques M. Bahi and Raphaël Couturier and Flavien Vernier, Synchronous distributed load balancing on dynamic networks. *J. Parallel Distrib. Comput*, pp.1397--1405, 2005

[Bahi05b] Jacques M. Bahi and Sylvain Contassot-Vivier and Raphaël Couturier and Flavien Vernier, A Decentralized Convergence Detection Algorithm for Asynchronous Parallel Iterative Algorithms. *IEEE Trans. Parallel Distrib. Syst*, pp.4--13, 2005

[Beale94] Russell Beale and Andrew Wood, Agent-Based Interaction. *BCS HCI* pp.239--245, 1994

[Bellavista01] Paolo Bellavista and Antonio Corradi and Cesare Stefanelli, How to Monitor and Control Resource Usage in Mobile Agent Systems , 2001

[Bellman57] Richard Bellman, Dynamic Programming, Princeton University Press, 1957

[Benachenhou05] Lotfi Benachenhou and Samuel Pierre, A New Protocol for Protecting a Mobile Agent Using a Reference Clone. *MATA*, pp.364--373, 2005

[Bennett90] John K. Bennett and John B. Carter and Willy Zwaenepoel, Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. *PPOPP*, pp.168--176, 1990

[Bernard 02] Guy Bernard, Leïla Ismail, Apport des agents mobiles à l'exécution répartie, *Revue des sciences et technologies de l'information, série Techniques et science informatiques*, vol. 21, n° 6, p. 771-796, 2002

[Bernon00] Carole Bernon and Marie-Pierre Gleizes and Pierre Glize, L'agentification du code en perspective. *4ième Ecole d'Informatique des Systèmes Parallèles et Répartis, ISYPAR 2000, Code Mobile , Toulouse*, pp.41--53, 2000

[Berry05] Gérard Berry, Esterel v7: From Verified Formal Specification to Efficient Industrial Designs. *Fundamental Approaches to Software Engineering, 8th International Conference, (FASE) 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK* pp.1, 2005

[Beynier05] Aurélie Beynier and Abdel-Illah Mouaddib, A polynomial algorithm for decentralized Markov decision processes with temporal constraints. *4rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005)* pp.963--969, 2005

[Bi05] Tra Goore Bi and Ibrahim Lokpo and Gérard Padiou, Localisation décentralisée et adaptative d'agents mobiles dans les réseaux dynamiques. *RenPar'16 : Rencontres francophones du Parallélisme, Le Croisic, France* pp.213--218, 2005

[Bourdon03] François Bourdon and Salah El Falou, The Interactional Semantics of Knowledge for Distributed Calculation in Large-Scale Open Systems. *The 7th World Multiconference on Systemics, Cybernetics and Informatics*, 2003

[Braun05] Braun, Peter and R. Rossak, Wilhelm, Mobile Agents-Basic Concept, Mobility Models, and the Tracy Toolkit, Morgan Kaufmann Publishers, 2005

[Budau02] Victor Budau and Guy Bernard, Synchronous/Asynchronous Switch for a Dynamic Choice of Communication Model in Distributed Systems. *ICPADS*, pp.97--102, 2002

[Cao03] Jiannong Cao and Yudong Sun and Xianbin Wang and Sajal K. Das, Scalable load balancing on distributed web servers using mobile agents. *J. Parallel Distrib.Comput.*, pp.996--1005, 2003

[Carlier97] David Carlier and Didier Donsez, Permanent Network Representation for Mobile User. *OPODIS* pp.263--268, 1997

[Carvalho04] Marco M. Carvalho and Thomas B. Cowin and Niranjan Suri and Maggie R. Breedy and Kenneth Ford, Using mobile agents as roaming security guards to test and improve security of hosts and networks. *SAC* pp.87--93, 2004

[Case90] Case, J.D. and M. Fedor and Schoffstall, M.L. and C. Davin, Simple Network Management Protocol (SNMP) RFC 1157, 1990

[Charpentier05] Michel Charpentier and Gérard Padiou and Philippe Quéinnec,

Cooperative Mobile Agents to Gather Global Information. *NCA*, pp.271--274, 2005

[Charton03] Romaric Charton and Anne Boyer and François Charpillet, Learning of Mediation Strategies for Heterogeneous Agents Cooperation. *ICTAI*, pp.330--337, 2003

[Chen98] Larry Chen, AgentOS: the agent-based distributed operating system for mobile networks. *Crossroads*, pp.12--14, 1998

[Chess97] D. Chess and C. Harrison and A. Kershenbaum, Mobile Agents: Are They a Good Idea?. *j-LECT-NOTES-COMP-SCI*, pp.25--48, 1997

[Claessens03] Claessens and Preneel and Vandewalle, (How) Can Mobile Agents Do Secure Electronic Transactions on Untrusted Hosts? A Survey of the Security Issues and the Current Solutions. *ACMTIT: ACM Transactions on Internet Technology*, 2003

[CPU_bench] Standard Performance Evaluation Corporation. pp., http://www.spec.org/

[Cybenko89] George Cybenko, Dynamic Load Balancing for Distributed Memory Multiprocessors. *J. Parallel Distrib. Comput*, pp.279--301, 1989

[D'agent] Dartmouth Agent, http://agent.cs.dartmouth.edu/

[Diaz01] Jesus Arturo Perez Diaz and Dario Alvarez Gutierrez and Igor Sobrado, A fast data protection technique for mobile agents against malicious hosts. *Electr. Notes Theor. Comput. Sci*, 2001

[Dijkstra59] E. W. Dijkstra, A note on two problems in connexion with graphs. *Numerische Mathematik*, pp.269--271, 1959

[Dillenseger02] Bruno Dillenseger and Anne-Marie Tagant and Laurent Hazard, Programming and Executing Telecommunication Service Logic with Moorea Reactive Mobile Agents. *Mobile Agents for Telecommunication Applications, 4th International Workshop, MATA 2002* pp.48--57, 2002

[Distributed.net] The distributed.net home page, http://www.distributed.net

[Erfurth01] C. Erfurth and P. Braun and W. Rossak,, Migration Intelligence for Mobile Agents, http://citeseer.ist.psu.edu/erfurth01migration.html, 2001

[Evripidou01] Paraskevas Evripidou and George Samaras and Christoforos Panayiotou and Evaggelia Pitoura, The PaCMAn Metacomputer: parallel computing with Java mobile agents. *Future Generation Comp. Syst*, pp.265--280, 2001

[Falou03a] Salah El Falou and François Bourdon and Bruno Dillenseger, Routing Information In Distributed Knowledge Environments with Reactive Mobile Agents.. *PDPTA* pp.1019--1025, 2003

[Falou04a] Salah El Falou and Wassim El Falou and François Bourdon, A Web-accessible mobile agents platform. *Information and Communication Technologies: From Theory to Applications(ICTTA04-IEEE)*, pp.499--500, 2004

[Falou04b] Salah El Falou and François Bourdon, Programmation répartie et agent mobile. *International Conference: Sciences of Electronic, Technologies of Information and Telecommunications, (SETIT04-IEEE)*, pp.400--405, 2004

[Falou05a] Salah El Falou and François Bourdon, Mobile Agent Migration: An Optimal

Policy. *Australian Conference on Artificial Intelligence*, pp.1204--1208, 2005

[Falou05b] Salah El Falou and François Bourdon, A MDP Solution for Mobile Agents Displacement. *ICTAI* pp.29--33, 2005

[Falou06] Salah El Falou et François Bourdon, Agent mobile et recherche d'information sur le Web : une solution basée sur le MDP. *Reconnaissance des Formes et Intelligence Artificielle (RFIA06)*, 2006

[FIPA] (FIPA) Abstract Architecture Specification statut: «Standard, version L», http://www.fipa.org/specs/fipa00001/2002

[Fuggetta98] Alfonso Fuggetta and Gian Pietro Picco and Giovanni Vigna, Understanding Code Mobility. *IEEE Transactions on Software Engineering*, pp.342--361, 1998

[Galtier00] Virginie Galtier and Kevin L. Mills and Yannick Carlinet and Stefan Leigh and Andrew Rukhin, Expressing meaningful processing requirements among heterogeneous nodes in an active network. *WOSP '00: Proceedings of the 2nd international workshop on Software and performance,* pp.20--28, 2000

[Galtier01] Virginie Galtier and Kevin L. Mills and Yannick Carlinet and Stephen F. Bush and Amit Kulkarni, Predicting and Controlling Resource Usage in a Heterogeneous Active Network. *Active Middleware Services*, pp.35--44, 2001

[Gavalas04] D. Gavalas,, Optimising Mobile Agent Migrations: An Experimental Approach,. *Proceedings of the 2nd International Working Conference on Performance Modelling and Evaluation of Heterogeneous Networks,* pp.43/1 - 43/9, 2004,

[Georgousopoulos03] Christos Georgousopoulos and Omer F. Rana, Combining State and Model-based Approaches for Mobile Agent Load Balancing. *SAC*, pp.878--885, 2003

[Gomoluch01] J. Gomoluch and M. Schroeder, Information agents on the move: A survey on loadbalancing with mobile agents, 2001

[Grasshopper98] M. Breugst and I. Busse and S. Covaci and T.Magedanz, Grasshopper -- A Mobile Agent Platform for IN Based Service Environments. *Proceedings of IEEE IN Workshop*, pp.279--290, 1998

[Grimaud00] Gilles Grimaud, Camille : un système d'exploitation ouvert pour carte à microprocesseur, 2000

[Grimaud02] G. Grimaud and S. Jean, Carte à Puce et Code Mobile. *Technique et Science Informatique*, pp.797--822, 2002

[Grimaud99] Gilles Grimaud and Jean-Louis Lanet and Jean-Jacques Vandewalle, FACADE: a typed intermediate language dedicated to smart cards. *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pp.476--493, 1999

[Groot04] David R. A. De Groot and Frances M. T. Brazier and Benno J. Overeinder, Cross-Platform Generative Agent Migration, 2004

[Gustafson88] John Gustafson, Reevaluating Amdahl's Law. *Communications of the*

ACM, pp.532--533, 1988

[Guyennet97] H. Guyennet and J-C. Lapayre and M. Tréhel, CAliF: une plate-forme de d'eveloppement de collecticiels utilisant la mémoire partagée distribuée. *Calculateurs paralleles*, pp.251-271, 1997

[Hagimont02] Daniel Hagimont and Nabil Layaïda, Adaptation d'une application multimédia par un code mobile. *Technique et Science Informatiques*, pp.877--897, 2002

[Hantz06] F. Hantz and H. Guyennet, A P2P Platform using sandboxing. *HPCS'06, Workshop on security and high Performance computing systems, In conjunction with ECMS 2006, 20th European Conf. on Modelling and Simulation, Bonn, Germany*, pp.736-739, 2006

[Harrouet00] Harrouet F., oRis : s'immerger par le langage pour le prototypage d'univers virtuels a base d'entites autonomes Universite de Bretagne Occidentale, Brest, France, 2000

[Harrouet02] F. Harrouet and Jacques Tisseau and Patrick Reignier and Pierre Chevaillier, oRis: un environnement de simulation interactive multi-agents. *Technique et Science Informatiques*, pp.499--524, 2002

[Hegarty97] David F. Hegarty and M. T. Kechadi, Topology preserving dynamic load balancing for parallel molecular simulations. *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pp.1--19, 1997

[Hohlfeld02] Matthew Hohlfeld and Aditya Ojha and Bennet Yee, Security in the Sanctuary System, http://historical.ncstrl.org/tr/ps/ucsd_cs/CS2002-0731.ps, 2002

[Howar60] Ronald A. Howard,, Dynamic Programming and Markov Processes, The MIT Press, 1960

[Ibrahim02] Ibrahim, M.A.M.and Lu Xinda, Utilization of cluster of PCs for the study of dynamic load balancing. *IEEE Conference on Computers, Communications, Control and Power Engineering (TENCON'02)* pp.335-337, 2002

[Ismail99] L. Ismail and D. Hagimont and J. Mossière, Evaluation of the Mobile Agents Technology: Comparison with the Client/Server Paradigm, 1999

[Java] Java Sun, http://java.sun.com

[Kwok99] Yu-Kwong Kwok and Ishfaq Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, pp.406--471, 1999

[Lang99] Christophe Lang, Répartition de charge dynamique à l'initiative des processus : étude, algorithmes et implémentations. Université de Franche-Comté, 1999

[Leclercq04] Matthieu Leclercq and Vivien Quéma and Jean-Bernard Stefani, DREAM: a component framework for the construction of resource-aware, reconfigurable MOMs. *Adaptive and Reflective Middleware*, pp.250--255, 2004

[Legrand04] Arnaud Legrand and Hélène Renard and Yves Robert and Frédéric Vivien, Mapping and Load-Balancing Iterative Computations. *IEEE Trans. Parallel Distrib. Syst*, pp.546--558, 2004

[Littman95] Michael L. Littman and Thomas L. Dean and Leslie Pack Kaelbling, On the complexity of solving {M}arkov decision problems. *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence (UAI--95)*, pp.394--402, 1995

[Lundgren01] Henrik Lundgren and David Lundberg and Johan Nielsen and Erik Nordstrom and Christian Tschudin, A Large-scale Testbed for Reproducible Ad hoc Protocol Evaluations, 2001

[MAF] (MAF) Mobile Agent Facility, http://www.omg.org/technology/documents/formal/mobile_agent_facility.html

[MAL] The Mobile Agent List, http://mole.informatik.uni-stuttgart.de/mal/mal.html

[Martin-flatin99] J. Martin-Flatin, Push vs. Pull in Web-Based Network Management, The 6th IFIP/IEEE International Symposium on Integrated Network Management (IM'99),pp.1-19, 1999

[MD5] Message Digest, http://www.ietf.org/rfc/rfc3174.txt

[Milojicicc98] Dejan S. Milojicicc and William LaForge and Deepika Chauhan, Mobile Objects and Agents (MOA). *Proceedings of the fourth Conference on Object-Oriented Technologies and Systems (COOTS)*, 1998

[Montresor02] Alberto Montresor and Hein Meling and Ozalp Babaoglu, Messor: Load-Balancing through a Swarm of Autonomous Agents. *AP2PC* pp.125--137, 2002

[Netview] IBM Tivoli NetView, http://www.tivoli.com

[Noble97] Brian D. Noble and M. Satyanarayanan and Dushyanth Narayanan and James Eric Tilton and Jason Flinn and Kevin R. Walker, Agile Application-Aware Adaptation for Mobility. *Sixteen ACM Symposium on Operating Systems Principles*, pp.276--287, 1997

[ORIS] oRis, http://www.enib.fr/~harrouet/oris.html

[Padiou05] Gérard Padiou, Michel Charpentier, Philippe Quéinnec, Collaborating Mobile Agents to gather Global Information in Dynamic Networks. Application to Load Balancing Example. *The 4th IEEE International Symposium on Network Computing and Applications (IEEE NCA05)*, pp.120--123, 2005

[Prabhavalkar03] Niraj Prabhavalkar and Manish Parashar, Controlling unresponsive connections in an active network architecture. *Int. J. Netw. Manag.*, pp.289--305, 2003

[Puterman94] Martin L. Puterman, Markov Decision Processes---Discrete Stochastic Dynamic Programming, John Wiley & Sons, Inc. , 1994

[Rankl03] W. (Wolfgang) Rankl and W. Effing, Smart Card Handbook, pub-WILEY, 2003

[Sahai98] Akhil Sahai and Christine Morin, Mobile Agents for Enabling Mobile User Aware Applications. *Proceedings of the 2nd International Conference on Autonomous Agents (Agents'98)*, pp.205--211, 1998

[Sahai99] Akhil Sahai and Christine Morin, Mobile Agents for Managing Networks: The MAGENTA Perspective. 1999

[Sandholm00] Tuomas Sandholm and Qianbo Huai, Nomad: Mobile Agent System for an Internet-Based Auction House. *IEEE Internet Computing*, pp.80--86, 2000

[Satoh04] Ichiro Satoh, Bio-inspired Deployment of Distributed Applications. *PRIMA,* pp.243-258, 2004

[Schwartz00] Beverly Schwartz and Alden W. Jackson and W. Timothy Strayer and Wenyi Zhou and R. Dennis Rockwell and Craig Partridge, Smart packets: applying active networks to network management. *ACM Trans. Comput. Syst.*, pp.67--88, 2000

[SHA] Secure Hash Algorithm, http://www.ietf.org/rfc/rfc1321.txt

[Shao05] Min-Hua Shao and Jianying Zhou, Protecting mobile-agent data collection against blocking attacks. *Computer Standards & Interfaces*, 2005

[Shen85] Chien-Chung Shen and Wen-Hsiang Tsai, A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion. *IEEE Transactions on Computers*, pp.197--203, 1985

[Sinclair87] J. B. Sinclair, Efficient Computation of Optimal Assignments for Distributed Tasks. *Journal of Parallel and Distributed Computing*, pp.342--362, 1987

[SMI] OpenMaster: Service management intelligence for IT and Telecom services, http://www.evidian.com/openmaster/index.htm

[Smith80] R. Smith, The contract net protocol: high-level communication and control in a distributed problem solver. *IEEE Trans Computers*, pp.1104-1113, 1980

[Stefano00] Antonella Di Stefano and Corrado Santoro, NetChaser: Agent Support for Personal Mobility. *IEEE Internet Computing*, pp.74--79, 2000

[Strasser97] Markus Strasser and Markus Schwehm,, A Performance Model for Mobile Agent Systems,. *PDPTA,* pp.1132--1140, 1997,

[SUN05] Java Card 2.2.2 Runtime Environment Specification, http://java.sun.com/products/javacard/2005

[Sutton98] R. S. Sutton and A. G. Barto,, Reinforcement Learning: An Introduction, MIT Press, 1998

[Tanenbaum92] A. S. Tanenbaum, Modern Operating Systems, Prentice-Hall .1992

[Tantawi85] Asser N. Tantawi and Don Towsley, Optimal static load balancing in distributed computer systems. *J. ACM*, pp.445--465, 1985

[Targui00] Boubekeur Targui, Modélisation et Observations des Colonnes à Distiller Université Claude Bernard - Lyon 1, 2000

[Tonguz03] Ozan K. Tonguz and Evsen Yanmaz, On the Theory of Dynamic Load Balancing. *IEEE Global Telecommunications Conference (GLOBECOM'03)*, pp.3626--3630, 2003

[Wetheral99] David Wetherall, Active network vision and reality: lessions from a capsule-based system. *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pp.64--79, 1999

[Xu90] Jian Xu and Kai Hwang, Heuristic methods for dynamic load balancing in a

message-passing supercomputer. *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pp.888--897, 1990

Résumé

Titre : Programmation répartie, optimisation par agent mobile

Pour bien fonctionner, une application répartie nécessite de communiquer et d'échanger des informations entre ces différentes entités. Les agents mobiles apparaissent dans ce contexte comme une solution prometteuse permettant la construction d'applications flexibles, adaptables aux contraintes de l'application et de l'environnement d'exécution. Dans cette thèse, la mobilité est étudiée sous deux angles. D'une part, l'envoi du code sur le serveur permet d'adapter les services distants aux exigences du client ce qui permet la réduction du trafic réseau. D'autre part, une machine surchargée peut déléguer l'exécution de certaines de ces tâches à une autre machine ce qui permet de gagner au niveau du temps d'exécution. Une architecture basée sur la technologie d'agents mobiles est proposée. Elle permet l'équilibrage de charge dans une application répartie. L'architecture proposée est décentralisée et l'équilibrage de charge se fait d'une façon dynamique. Un agent mobile collecteur est utilisé afin de construire une vision globale du système. Pour la réduction du trafic, nous proposons la communication par un agent intelligent hybride. L'agent utilise ainsi deux modes, client/serveur ou migration (échange locale), pour sa communication. Le processus décisionnel de Markov est utilisé pour trouver la politique optimale du déplacement de l'agent. Un travail d'expérimentation sur des problèmes concrets permet de valider les algorithmes proposés.

Mots-clés : Agents Mobiles, Intelligence Artificielle Répartie, Recherche sur Internet, Processus Décisionnel de Markov.

Abstract

Title: Distributed programming, optimization with mobile agent

A distributed application needs to communicate and to exchange information between various entities. The mobile agent appears in this context as a promising solution, allowing the construction of flexible applications, adaptable to the constraints of the application and to the execution environment. In this thesis, mobility is studied from two angles. First, when sending the mobile agent to a server, the distant services are adapted to the requirements of the clients such that it allows the reduction of the network traffic. In this case, we make the communication with a hybrid intelligent agent. This agent uses two modes: client/server, or migration (local exchanges), to carry out the client tasks. We use the Markov decision process to find the optimal policy for agent displacement. Secondly, mobile agents are studied as computation entites which have the capability to migrate from overloaded to underloaded machines. In this work, we propose an architecture based on mobile agents technology. This architecture is decentralized and the load balancing is done in a dynamic way. Decisions are made localy and a mobile agent is used to build a global vision of the system. Experimentation on concrete problems makes it possible to validate the proposed algorithms.

Key words: Mobile Agents, Distributed Artificial Intelligence, Internet Searching, Markov Decision Process.

Discipline : Informatique.

Groupe de Recherche en Informatique, Image, Automatique et Instrumentation de Caen.

CNRS UMR 6072