Université de Montréal

Architecture et services pour la distribution de charge dans les systèmes distribués objet

par

Elarbi Badidi

Département d'Informatique et de Recherche Opérationnelle

Faculté des Arts et des Sciences

Thèse présentée à la Faculté des études supérieures
en vue de l'obtention du grade de
Philosophie Doctor (Ph.D.)
en informatique

Mai, 2000

Université de Montréal

Faculté des études supérieures

Cette thèse intitulée :

Architecture et services pour la distribution de charge dans les systèmes distribués objet

présentée par

Elarbi Badidi

a été évaluée par un jury composé des personnes suivantes :

SAHRAOUI, HOUARI ABDELKRIM	Président-rapporteur
KELLER, RUDOLF	Directeur de recherche
KROPF, PETER	Codirecteur
LUSTMAN, FRANÇOIS	Membre du jury
SCHIPER, ANDRÉ	Examinateur externe
HOULE, JEAN-LOUIS	Représentant du doyen de la FES

Thèse acceptée le : 26 juillet 2000

SOMMAIRE

Cette thèse étudie le problème de la distribution de charge dans les systèmes distribués à base d'objets. Les techniques de distribution de charge au niveau processus sont assez bien maîtrisées dans les systèmes distribués classiques homogènes. Cependant, peu de travaux de recherche se sont intéressés à cette problématique dans les systèmes distribués à base d'objets, dont l'objet est souvent considéré comme l'unité de distribution, en tenant compte des spécificités de ces systèmes. L'objectif de ce travail est de définir des approches de distribution de charge adaptées à ces environnements caractérisés par l'hétérogénéité des machines et des systèmes d'exploitation et la structuration des applications en objets.

Le placement d'objets et la migration d'objets sont les deux approches principalement utilisées dans certains systèmes distribués à base d'objets pour mettre en oeuvre la distribution de charge au niveau objet. Ceci est réalisé en déplaçant les objets des machines surchargées à celles ayant plus de ressources disponibles. Ces deux approches s'appuient sur des concepts propriétaires et spécifiques de ces systèmes qui sont souvent homogènes. Les systèmes distribués objet récents sont capables d'être déployés dans des environnements hétérogènes. L'utilisation de la migration d'objets dans ces systèmes est encore difficile à réaliser bien qu'il y ait des travaux dans ce sens.

Les systèmes distribués objet fonctionnent conformément au modèle client/serveur. L'objet appelant est nommé client et l'objet appelé est nommé serveur. Nous considérons dans cette thèse une autre approche pour la mise en oeuvre de la distribution de charge dans ces systèmes. Cette approche est l'assignation d'invocations de méthodes (requêtes) qui consiste à assigner, conformément à une certaine stratégie, l'exécution d'une invocation de méthode à un serveur objet approprié capable d'exécuter cette méthode. Les systèmes fonctionnant suivant le modèle client/serveur, tels que les serveurs Web et les serveurs ftp, utilisent essentiellement l'assignation de requêtes pour effectuer la distribution de charge. L'assignation de requêtes est réalisée suivant deux approches : une approche orientée client basée sur les choix volontaires des clients et une approche centralisée basée sur l'utilisation

d'un répartiteur. Nous avons introduit une autre approche distribuée qui est basée sur la coopération entre les serveurs objets.

Afin d'évaluer ces approches d'assignation de requêtes dans le contexte des systèmes distribués à base d'objets, nous avons conçu une architecture appelée LoDACE (Load Distribution Architecture for Distributed Object Computing Environments) constituant une plate-forme pour la mise en oeuvre de ces trois approches avec diverses stratégies d'assignation. Cette architecture est basée sur l'organisation des objets serveurs dans des grappes logiques d'après les types de service offerts par ces serveurs, sur la découverte dynamique des serveurs, et sur la surveillance de la charge des serveurs et des machines du système. Un prototype de cette architecture a été réalisé dans un environnement CORBA avec le langage de programmation Java.

Les systèmes distribués tels que CORBA et DCOM n'offrent pas de service standard de distribution de charge bien que plusieurs autres services standards aient été spécifiés pour le nommage, la gestion des transactions, la gestion des événements, etc. La spécification CORBA décrit des principes pour la définition de nouveaux services. Ceci nous a amené à concevoir un service de distribution de charge, appelé LSS (Load Sharing Service), basé sur l'assignation d'invocations de méthodes, et qui est conforme à la philosophie des services CORBA. Un prototype du service LSS a été réalisé dans le même environnement que celui du prototype de LoDACE.

Les expérimentations conduites à l'aide des prototypes de LoDACE et de LSS nous ont permis d'évaluer et de comparer les performances de certaines stratégies d'assignation. La stratégie dynamique basée sur la sélection du serveur le moins chargé, parmi les serveurs qui offrent le service requis par un client, s'est avérée la plus performante en termes de temps de réponse moyen des requêtes des clients en comparaison avec les stratégies statiques (aléatoire, cyclique, et orientée client) qui ne tiennent pas compte de l'état des serveurs dans la sélection du serveur cible. De plus, la stratégie dynamique permet d'avoir une distribution uniforme de la charge des serveurs considérés dans l'expérimentation.

Table des matières

Liste des abréviations

BOA	Basic Object Adapter
CO	Client Oriented
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
DCE	Distributed Computing Environment
DCOM	Distributed Component Object Model
ESIOP	Environment Specific Inter ORB Protocol
GIOP	General Inter ORB Protocol
IDL	Interface Definition language
IETF	The Internet Engineering Task Force
IIOP	Internet Inter ORB Protocol
IP	Internet Protocol
IPC	Inter Process Communication
ISO	International Organization for standardization
ITU	International Telecommunication Union
LL	Least-Loaded
LoDACE	Load Distribution Architecture in Distributed Object Computing Environments
LRU	Last Recently Used
LSS	Load Sharing Service
ODP	Open Distributed Processing
OLE	Object Linking and Embedding
OMA	Object Management Architecture
OMG	Object Management Group
ORB	Object Request Broker

ORPC	Object Remote Procedure Call
RD	Random
RFP	Request For Proposal
RLRU	Relative Last Recently Used
RR	Round Robin
SLP	Service Location Protocol
TINA-C	Telecommunications Information Networking Architecture Consortium

Liste des figures

Chapitre 5

Chapitre 6

Chapitre 7

Annexe 1

Annexe 2

Liste des tableaux

Remerciements

Par la présente occasion, je tiens à remercier mon Directeur de Recherche le Professeur Rudolf K. Keller et mon Codirecteur le Professeur Peter G. Kropf pour leur aide, leurs conseils fructueux, et pour la confiance qu'ils m'ont témoignée tout au long de ce travail.

Je remercie Messieurs André Schiper, Houari Sahraoui et François Lustsman qui ont bien accepté de composer mon jury de thèse et de juger ce travail.

Je remercie Vincent Van Dongen, chercheur associé au CRIM (Centre de Recherche Informatique de Montréal), qui était mon codirecteur au début de ce travail de thèse.

Mes remerciements s'adressent aussi aux responsables et fonctionnaires de l'unité Systèmes Distribués et Télécommunications du Centre de Recherche Informatique de Montréal (CRIM) qui m'ont accordé toutes les facilités et l'environnement pour mener à bien ce travail.

Je remercie John McCann et Anne O'Shea de Iona technologies qui nous ont accordé la licence d'OrbixWeb et d'OrbixTrader que nous avons utilisé dans les expérimentations.

Je remercie tous les membres du laboratoire de génie logiciel "Gelo" du Département d'Informatique et de Recherche Opérationnelle de l'Université de Montréal, pour leur encouragement continuel.

Je remercie Souad, mes parents, mes frères, ainsi que mes amis pour leur soutien constant tout au long de ce travail.

Enfin, j'adresse mes remerciements à tous ceux qui ont contribué de près ou de loin à la bonne conduite de ce travail.

À mes parents : Sidi Mohammed et Lalla Fatima

Chapitre 1
Introduction

Ce chapitre présente sommairement le contexte de ce travail, la problématique et les motivations, les contributions apportées, et enfin l'organisation de ce rapport.

Contexte

Les services distribués d'aujourd'hui sont déployés dans de larges systèmes distribués, tel que l'Internet, et servent un grand nombre d'utilisateurs. Comme exemples de ces services, citons les services liés au commerce électronique et les services multimédia. Ces services distribués à grande échelle requièrent généralement un très haut degré de disponibilité et de fiabilité. Ainsi, plusieurs aspects doivent être pris en considération lors de la conception des applications sous-jacentes à ces services, à savoir la fiabilité, la disponibilité, la distribution de charge, la tolérance aux fautes, la sécurité, et la gestion de concurrence.

Les dix dernières années ont vu l'émergence d'environnements qui permettent le déploiement à grande échelle des services distribués. Ces environnements sont le résultat de la convergence de la technologie client/serveur et de la technologie orientée objet et sont appelés systèmes distribués objet. Les technologies les plus marquantes dans ce domaine sont les implémentations de CORBA (Common Object Request Broker Architecture) [OMG97b] définie par l'OMG (Object Management Group) et DCOM (Distributed Component Object Model) définie par Microsoft [Microsoft96]. Le traitement distribué orienté objet représente un paradigme qui permet aux objets d'être distribués dans un réseau hétérogène et d'interagir par échange de messages pour faire des demandes d'informations ou de services. Les objets assureront dynamiquement les rôles de clients et de serveurs dans toute interaction.

L'architecture CORBA représente un modèle standard pour la construction d'applications à objets distribués (répartis sur un réseau). Au centre de cette architecture, un routeur de

messages (ORB : Object Request Broker) permet à des objets clients d'envoyer des requêtes et de recevoir des réponses sans avoir à se préoccuper des détails techniques propres à l'infrastructure du réseau. L'ORB se charge de transmettre les requêtes aux objets concernés, où qu'ils se trouvent (dans le même processus, dans un autre, sur un autre noeud du réseau). Le bus CORBA (dont l'ORB est le composant central) permet d'assurer la transparence des invocations de méthodes. Les requêtes aux objets semblent toujours être locales. Une description plus détaillée de cette architecture est donnée à l'annexe 1.

DCOM est une extension du modèle objet COM (Component Object Model). Il construit une couche ORPC (Object Remote Procedure Call) au-dessus de la couche RPC (Remote Procedure Call) de DCE (Distributed Computing Environment) [Rosenberry92] pour offrir le support des objets distants. Un objet COM peut supporter plusieurs interfaces, chacune représentant une vue différente du comportement de l'objet. Une interface définit un ensemble de méthodes qui contribuent à offrir une même fonctionnalité. Un client COM interagit avec un objet COM au moyen d'un pointeur sur une de ses interfaces, en utilisant les invocations des méthodes de l'interface, comme si l'objet résidait dans le même espace d'adressage que le client.

Problématique et motivations

L'émergence des réseaux à haut débit et à large distance (ATM, réseaux tout optique, etc.) et d'architectures de référence de systèmes ouverts, telles que ODP [ISO96a] et CORBA, permet d'envisager la construction de systèmes distribués objet à grande échelle. Dans les environnements distribués à base de ces architectures, le problème de la disponibilité des données et de l'optimisation de l'utilisation des ressources de traitement et de communication est un aspect dont l'importance ne cesse de croître.

Les politiques de distribution de charge entre les processeurs d'un système ont pour objectif de répondre à cette problématique. De telles politiques sont aujourd'hui assez bien maîtrisées dans le contexte des systèmes distribués homogènes, à base de processus et s'exécutant sur un réseau local. Cependant, peu de travaux ont étudié la problématique de distribution de charge dans les systèmes distribués à base d'objets en tenant compte des spécificités de ces systèmes, soit la structuration des applications en objets distribués, la communication par invocation de

méthodes, et l'hétérogénéité des machines et des systèmes d'exploitation. De plus, l'architecture CORBA et le système DCOM n'offrent jusqu'à présent aucun service standard en matière de distribution de charge. Dans ce travail, nous essayons d'apporter des éléments de réponse à cette problématique par la proposition d'approches de distribution de charge adaptées à ces systèmes.

Les travaux existants considèrent essentiellement deux méthodes pour réaliser la distribution de charge dans les systèmes distribués objet :

- La migration d'objets -- Cette méthode consiste à déplacer les serveurs objet des machines surchargées vers des machines ayant plus de ressources disponibles.
- Le placement initial d'objets -- Cette méthode consiste à placer les objets d'une application distribuée dans les diverses machines du système de sorte que l'accès aux ressources matérielles et logicielles soit plus commode et plus efficace.

Les solutions proposées dans ces travaux restent des solutions propriétaires car elles se basent sur des concepts propres aux systèmes sous-jacents. Par conséquent, leur utilisation dans des systèmes hétérogènes n'est pas appropriée. La migration d'objets, plus particulièrement, est difficile à réaliser en raison de ses exigences de sauvegarde, de transfert, et de restauration de l'état de l'objet, de son contexte d'exécution, et de ses canaux de communication. De plus, la littérature de ces travaux ne décrit pas de façon précise l'efficacité de ces solutions dans la réalisation de la distribution de charge.

Nous avons retenu et appliqué une autre méthode qui est l'assignation de requêtes comme moyen efficace et peu onéreux pour réaliser la distribution de charge dans les systèmes distribués objet. Cette méthode consiste à assigner l'exécution d'une invocation de méthode à un serveur objet approprié capable d'exécuter cette méthode. Elle a fait ses preuves dans les architectures client/serveur et dans les systèmes transactionnels distribués. Elle est utilisée récemment par les serveurs Web. En outre, il est possible de l'utiliser dans des systèmes hétérogènes. Son utilisation dans les systèmes distribués objet reste encore très limitée.

Contributions

Les contributions majeures de ce travail résident aux niveaux de la proposition de trois approches d'assignation de requêtes, de la conception d'une architecture d'assignation de requêtes qui sert à mettre en œuvre ces trois approches, de la conception d'un service de distribution de charge conforme à la philosophie des services CORBA, et de l'expérimentation de ces approches au moyen de prototypes.

Approches d'assignation de requêtes

Nous proposons et évaluons trois approches, schématisées dans la figure 1.1, pour réaliser l'assignation de requêtes : orientée client, orientée répartiteur, et orientée serveur.

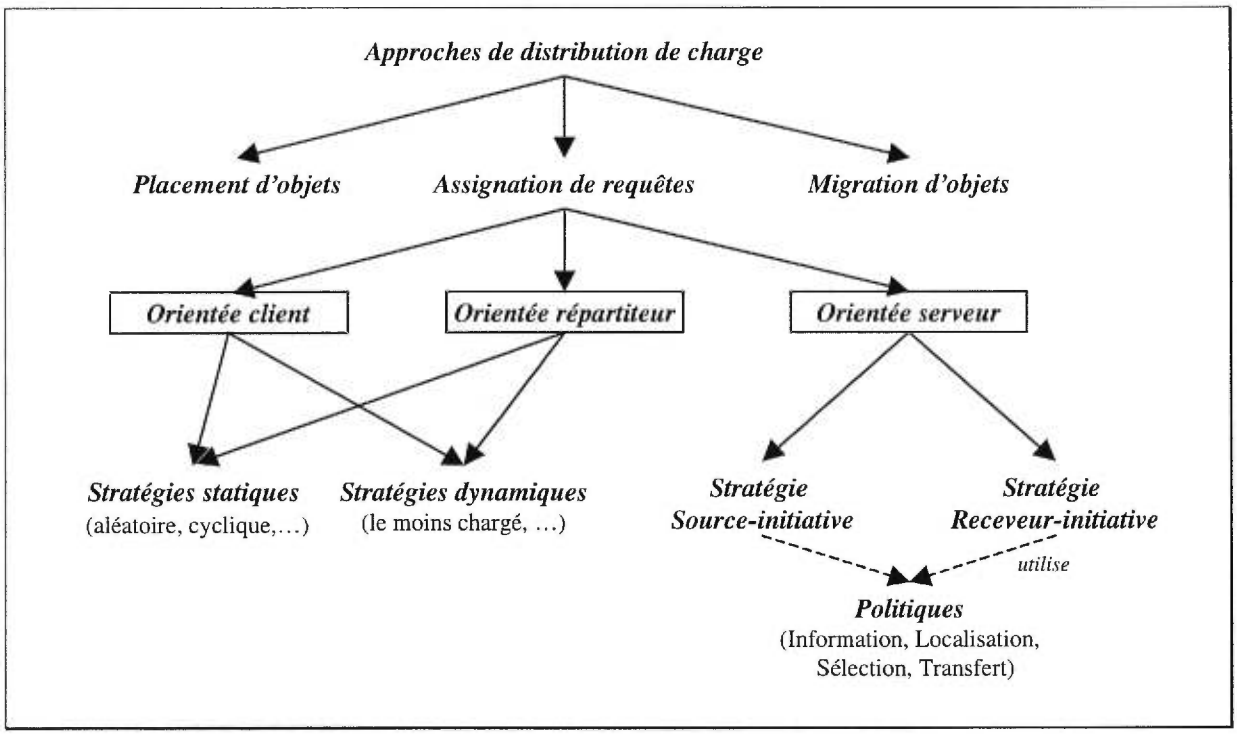


Figure 1.1 - Approches de distribution de charge dans les systèmes distribués objet

Les deux premières approches sont inspirées des méthodes de distribution de charge utilisées dans les architectures client/serveur, et sont utilisées dans peu de systèmes distribués objet. Les stratégies d'assignation de requêtes utilisées par ces approches peuvent être statiques ou dynamiques. Des exemples de stratégies sont cités à la section 1.3.4. La dernière approche

s'appuie sur des méthodes de distribution de charge utilisées dans les systèmes distribués classiques non orientés objet. À notre connaissance, elle n'a pas été utilisée auparavant dans les systèmes distribués objet. Elle est basée sur la coopération des serveurs objet par échange de leurs informations de charge et de requêtes. Nous proposons dans le cadre de cette approche des algorithmes pour la réalisation des stratégies source-initiative et receveur-initiative. Cette approche présente l'avantage d'être distribuée car les serveurs implémentent les politiques de distribution de charge (information, localisation, sélection, et transfert) et sont capables de prendre eux-même les décisions de transfert de requêtes vers d'autres serveurs contrairement aux deux premières approches.

Architecture de distribution de charge

Nous présentons la conception d'une architecture de distribution de la charge dans les systèmes distribués objet basés sur un modèle objet dans lequel les objets sont des entités passives constituées d'attributs représentant l'état de l'objet et d'un ensemble d'opérations sur ces attributs. Cette architecture, appelée LoDACE, est basée d'une part sur la surveillance de la charge des serveurs objet et sur le groupage des serveurs dans des grappes logiques selon les types de service offerts par les serveurs, et d'autre part sur la découverte dynamique des serveurs offrant les services requis par les utilisateurs. Elle utilise essentiellement la méthode d'assignation de requêtes aux différents serveurs, et ainsi les approches d'assignation mentionnées ci-haut peuvent être mises en oeuvre dans le cadre de cette architecture.

Service de distribution de charge dans l'environnement CORBA

Nous avons adopté une approche service pour le support de la distribution de charge dans les environnements CORBA. En effet, nous proposons un service de distribution de charge appelé LSS pouvant être utilisé par les applications pour répartir la charge entre les serveurs offrant un même type de service. Le service LSS spécifie un ensemble d'interfaces à utiliser par les clients pour accéder au service, et par les serveurs pour distribuer leurs charges avec d'autres serveurs offrant le même type de service. LSS ne définit pas une composante monolithique pouvant constituer un point susceptible d'être l'objet d'un goulot d'étranglement. Il est décomposé en plusieurs entités offrant des fonctionnalités diverses, telles que la surveillance de la charge des serveurs et la découverte dynamique des serveurs

sur la base des types de service offerts et des contraintes des clients. Ces composantes contribuent à la mise en œuvre de la distribution de charge entre les serveurs d'une même grappe. Cette organisation en plusieurs composantes promeut la modularité et la réutilisation.

Expérimentation et validation

Des prototypes de l'architecture LoDACE et du service LSS ont été réalisés dans l'environnement commercial OrbixWeb [Iona96] en utilisant le langage de programmation Java. Comme application de test, nous avons considéré la gestion de comptes bancaires. Ce service est offert par des serveurs qui implémentent une même interface. Les clients envoient des requêtes aux serveurs pour consulter la balance d'un compte ou pour faire des opérations de retrait et de dépôt. Cette application a servi dans les expérimentations conduites à l'aide des deux prototypes dans le but de tester et de comparer les performances des approches orientée client et orientée répartiteur discutées à la section 1.3.1. Les stratégies statiques considérées dans ces expérimentations sont : (1) la stratégie aléatoire (random – RD), (2) la stratégie cyclique (round-robin – RR), et (3) la stratégie orientée client (client-oriented – CO). Ces stratégies ne tiennent pas compte de la charge des serveurs dans le choix des serveurs cibles. La stratégie dynamique considérée est basée sur la sélection du serveur le moins chargé (least-loaded – LL).

La stratégie LL s'est avérée plus performante en termes de temps de réponse moyen en comparaison avec les stratégies RD, RR, et CO. Sous la charge la plus grande de l'expérimentation (35 clients émettant un total d'environ 200 requêtes avec chacune des stratégies considérées), le temps de réponse moyen avec la stratégie LL est 3,5 fois meilleur qu'avec la stratégie CO, 12,5 fois meilleur qu'avec la stratégie RD, et 13,3 fois meilleur qu'avec la stratégie RR. De plus, cette stratégie dynamique permet d'avoir une distribution presque uniforme de la charge des serveurs.

Organisation

Le chapitre 2 discute l'état de l'art en matière de distribution de charge dans les systèmes distribués. La première partie de ce chapitre présente une classification des approches utilisées dans les systèmes distribués classiques et les politiques qui composent un système de distribution de charge. Ensuite, elle décrit des exemples de systèmes opérationnels de

distribution de charge. La seconde partie de ce chapitre présente les approches utilisées dans les systèmes distribués à base d'objets ainsi que des exemples de systèmes existants utilisant ces approches.

Le chapitre 3 expose d'abord la problématique d'assignation de requêtes (invocations de méthodes) à des objets appropriés dans les systèmes distribués à base d'objets. Ensuite, il discute trois approches pour l'assignation de requêtes dans ces systèmes et présente une modélisation analytique de ce problème à l'aide d'un système de files d'attente avec de multiples classes de service.

Le chapitre 4 décrit l'architecture LoDACE que nous proposons pour réaliser la distribution de charge dans les environnements distribués à base d'objets. Cette architecture constitue un cadre d'application pour la mise en œuvre des approches d'assignation de requêtes discutées au chapitre 3. Ce chapitre présente également un prototype de l'architecture LoDACE qui a été réalisé dans l'environnement OrbixWeb.

Le chapitre 5 décrit LSS, un service de distribution de charge qui est conforme à la philosophie des services CORBA. De plus, les scénarios d'utilisation de ce service aussi bien par les objets clients que par les objets serveurs sont présentés.

Le chapitre 6 décrit les expérimentations que nous avons conduites à l'aide des prototypes de l'architecture LoDACE et du service LSS. Ensuite, il présente les résultats obtenus et une discussion de ces résultats.

Le chapitre 7 discute (1) les avantages et les inconvénients des approches d'assignation de requêtes présentées au chapitre 3, (2) les paramètres qui peuvent affecter les performances de ces approches, (3) les limitations des expérimentations décrites au chapitre 6, (4) l'impact des choix effectués lors de la conception et de l'expérimentation de LoDACE et de LSS, et (5) quelques propriétés désirables dans LoDACE et dans LSS pour offrir des fonctionnalités supplémentaires.

Le chapitre 8 conclut cette thèse en présentant un rappel des contributions et les perspectives futures de ce travail.

Chapitre 2

Distribution de charge dans les systèmes distribués

Ce chapitre présente la problématique de distribution de charge dans les systèmes distribués. La première partie décrit une classification des approches utilisées dans les systèmes distribués classiques, et présente les politiques essentielles qui composent généralement un système de distribution de charge. La seconde partie présente des exemples de systèmes distribués à base d'objets qui offrent le support pour la mise en oeuvre de la distribution de charge aux niveaux objet et opération en plus du niveau processus. La distribution de charge au niveau objet est généralement réalisée par le placement et par la migration d'objets. La distribution au niveau opération peut être réalisée par l'assignation d'invocations de méthodes à des objets appropriés. Le tableau 2.1 à la fin de ce chapitre récapitule pour chacun des systèmes présentés les approches utilisées et les grandes caractéristiques du modèle objet associé.

2.1. Introduction

Le progrès technologique réalisé dans la conception des processeurs a permis d'offrir aux utilisateurs des machines ayant des puissances de calcul considérables. Généralement, les utilisateurs utilisent seulement une fraction de la capacité de traitement de leurs machines. Ces machines se trouvent alors inutilisées pendant de longues durées. Des études ont montré que les stations de travail sont inutilisées jusqu'à 75% du temps [Mutka87,91]. Selon une étude du Los Alamos National Lab/USA [Genias], les stations de travail ne sont en service que pendant moins de 10% du temps. Néanmoins, il y a des périodes où un utilisateur désire exécuter des tâches qui demandent une puissance de calcul supérieure à la capacité effective

de sa machine. Il se voit alors obligé d'attendre pendant parfois de longues durées avant d'obtenir les ressources requises.

Donc, un problème important à résoudre est l'allocation des ressources CPU dans le but de répartir équitablement l'ensemble du travail sur les noeuds[1] du système. Contrairement au *partage de charge* (load sharing), qui consiste uniquement à répartir les processus[2] (ou tâches) entre les noeuds libres, *l'équilibrage de charge* (load balancing) consiste à équilibrer la charge entre l'ensemble des noeuds. Les deux termes, *équilibrage de charge* et *partage de charge*, sont souvent utilisés dans la littérature de façon interchangeable. Nous utilisons tout au long de cette thèse le terme *distribution de charge* pour designer aussi bien l'équilibrage de charge que le partage de charge. Les objectifs les plus usuels de la distribution de charge incluent la minimisation du temps de réponse moyen des tâches, la maximisation du débit moyen du système, la distribution équilibrée de la charge, la minimisation du temps d'inoccupation des processeurs, et l'augmentation de la fiabilité du système (tolérance aux fautes).

Avec l'avènement de la technologie objet, il est devenu plus facile de construire de grandes applications. En effet, le paradigme objet permet une réduction du temps de développement des logiciels, l'amélioration de la maintenance, et une meilleure réutilisation du code, en comparaison avec les techniques traditionnelles de programmation. En outre, avec le développement des systèmes d'exploitation distribués, il est devenu possible de construire des programmes formés d'un ensemble de modules pouvant s'exécuter concurremment sur un groupe de machines. L'intégration des deux concepts, paradigme objet et systèmes d'exploitation distribués, a permis d'avoir une nouvelle génération de systèmes appelés *systèmes distribués à base d'objets* [Chin91].

Ces dernières années, l'informatique distribuée à base d'objets s'est établie comme une base très pertinente pour le support des grands systèmes de calcul et de télécommunication comprenant des composantes matérielles et logicielles hétérogènes. En effet, des organismes internationaux comme l'ISO (International Organization for Standardization), l'OMG (Object Management Group), et TINA-C (Telecommunications Information Networking Architecture Consortium) ont défini des cadres d'application objet servant de base pour le développement

[1] Dans la suite de cette thèse, nous utilisons les termes *noeud* et *machine* pour designer une unité de traitement.

de systèmes distribués ouverts. CORBA et DCOM sont deux technologies déjà bien établies en matière de systèmes distribués objet. Ces deux technologies permettent aux objets d'opérer parmi des langages de programmation hétérogènes et des plates-formes hétérogènes.

Comme dans les systèmes distribués classiques, plusieurs aspects comme la performance, la disponibilité, la fiabilité, la tolérance aux fautes, la sécurité, et la distribution de charge doivent être considérés lors de la réalisation des applications distribuées à base d'objets [Schmidt95]. Dans cette thèse, nous nous concentrons plus particulièrement sur ce dernier aspect, soit le problème de la distribution de charge dans les systèmes distribués à base d'objets.

2.2. Distribution de charge dans les systèmes distribués classiques

Cette section décrit la problématique de distribution de charge dans les systèmes distribués classiques en présentant : une classification des approches utilisées, les politiques essentielles des systèmes de distribution de charge, un aperçu de travaux de recherche sur ce sujet, et des exemples d'implémentations de ces approches.

2.2.1. Classification des approches de distribution de charge

Les approches de distribution de charge présentées dans la littérature sont tellement nombreuses qu'il est impraticable de couvrir chacune d'elles dans tous ses détails. Néanmoins, il est possible de les classifier dans un petit nombre de catégories. La taxonomie qui est généralement utilisée est celle proposée par Casavant et Kuhl [Casavant88a]. Nous présentons dans cette section les quatre principales classifications suggérées par ces auteurs.

2.2.1.1 Approche centralisée et approche distribuée

Les approches de distribution de charge peuvent être classifiées en trois catégories, selon la manière dont l'information de charge est distribuée et dont les processus sont assignés aux noeuds [Shivaratri92].

Approche centralisée – Un noeud désigné comme coordinateur reçoit l'information de charge courante de tous les autres noeuds et l'assemble dans un vecteur de charge. Quand un noeud

[2] Les termes *processus* et *tâches* sont utilisés de manière interchangeable.

décide de transférer une tâche, il envoie une demande au coordinateur qui choisit alors un noeud cible, en utilisant le vecteur de charge, et informe le noeud source de ce choix. Cette approche réduit les frais généraux du système grâce à la centralisation de l'information de charge. Cependant, le coordinateur peut être l'objet d'un goulot d'étranglement. Une panne au niveau du coordinateur provoquera l'effondrement du système. Pour remédier à cette situation, des mécanismes de réplication sont souvent utilisés [Theimer89].

Approche distribuée – Le goulot d'étranglement, constaté lors de la collecte de l'information de charge dans l'approche précédente, peut être évité si l'on distribue la gestion de la prise de décision. À la différence de l'approche centralisée, chaque noeud construit de manière autonome son propre vecteur de charge en rassemblant l'information de charge des autres noeuds. Les décisions de placement sont faites localement en utilisant les vecteurs de charge locaux. Les approches distribuées peuvent accélérer de manière significative le processus de prise de décision, mais le coût des communications engendrées peut être élevé.

Approche mixte – Plusieurs systèmes de distribution de charge utilisent une approche mixte qui consiste à combiner les deux approches centralisées et distribuées pour profiter des avantages de chaque approche [Zhou88]. Un système peut, par exemple, adopter une politique d'information centralisée et une politique de localisation distribuée[3]. Dans ce cas, le coordinateur a besoin de distribuer périodiquement le vecteur de charge aux autres noeuds pour que chaque noeud puisse prendre localement ses décisions de placement.

Bien que certains auteurs affirment que les approches centralisées offrent de meilleurs résultats et sont évolutives [Theimer89, Ozden93, Zhou88], la majorité des auteurs reconnaît qu'une approche de distribution de charge doit être distribuée pour éviter des points de congestion et être ainsi évolutive [Kremien92, Shivaratri92].

2.2.1.2 Approche statique et approche dynamique

Une deuxième façon de classifier les approches de distribution de charge est suivant le degré d'information d'état des noeuds pris en compte par l'approche. On distingue alors trois catégories : statique, dynamique, et adaptative [Folliot93, Shivaratri92].

[3] Les politiques d'information et de localisation sont décrites à la section 2.2.2.

Approche statique – Les politiques de transfert et de localisation, définies à la section 2.2.2, sont basées seulement sur l'information concernant le comportement moyen du système. Les algorithmes statiques ne tiennent pas compte de l'état actuel du système pour mettre en oeuvre distribution de charge.

Approche dynamique – Contrairement à l'approche statique, les approches dynamiques utilisent l'information d'état du système lors des décisions de distribution de charge. Ainsi, elles ont le potentiel de surpasser les approches statiques en améliorant la qualité de leurs décisions. Ces approches améliorent la performance en exploitant les fluctuations à court terme dans l'état du système.

Approche adaptative – Les approches adaptatives de distribution de charge adaptent leurs activités aux changements intervenus dans l'état du système, en modifiant dynamiquement leurs paramètres voir même leurs algorithmes [Eager86b, Kremien92, Kumar89, Lin87, Mirchandaney89a,b]. Cette approche est capable d'offrir de meilleures performances lorsque l'état du système change fréquemment [Casavant88a, Zhou88, Shivaratri92].

2.2.1.3 Partage de charge et équilibrage de charge

La distribution de charge est souvent décrite dans la littérature aussi bien comme équilibrage de charge que partage de charge. Ces deux termes sont souvent utilisés de façon interchangeable, mais ont des définitions distinctes.

Partage de charge – En pratique, une distribution équilibrée de la charge du système n'est pas toujours l'objectif cherché. Ce qui est souvent désiré, c'est d'occuper toutes les machines lorsque des tâches sont en attente dans le système. Autrement dit, une tâche est placée à distance si la machine sur laquelle elle est destinée est surchargée et s'il existe dans le système une machine qui est légèrement chargée. Le placement et la migration des tâches ne sont envisagés que lorsque la charge locale dépasse un seuil admissible.

Équilibrage de charge – L'objectif de cette approche est de répartir équitablement la charge du système sur les machines du système. Autrement dit, elle essaie de garder la charge d'une machine aussi proche que possible de la charge moyenne du système. Le placement et la migration des processus sont envisagés chaque fois que les conditions globales du système changent. Il a été montré que l'équilibrage de charge a le potentiel d'offrir de meilleures

performances que le partage de charge si le coût de distribution de charge est ignoré [Krueger87].

2.2.1.4 Stratégies source-initiative et receveur-initiative

Dans un algorithme distribué de distribution de charge, chaque noeud du système est responsable de déterminer l'action à prendre. Le noeud doit localiser un noeud cible (receveur) auquel des processus peuvent être transférés, ou localiser un noeud source à partir duquel un processus peut migrer. L'état de chaque noeud peut être gardé dans un noeud central (coordinateur) ou diffusé à tous les noeuds du système.

Stratégie source-initiative – La stratégie *source-initiative* survient lorsqu'un noeud surchargé (source) cherche à transférer un processus local à un noeud cible (receveur) légèrement chargé. S'il existe un coordinateur ou si l'information de charge des noeuds est diffusée, alors le noeud source peut choisir un noeud cible en se basant sur l'information de charge diffusée ou obtenue auprès du coordinateur. Sinon, le noeud source doit envoyer des requêtes à plusieurs noeuds pour déterminer le noeud le plus vraisemblable pour l'exécution distante du processus. Deux conditions doivent être satisfaites dans ce cas pour que le transfert du processus soit effectif : la charge du noeud source doit dépasser le seuil courant, et un noeud cible doit être trouvé [Eager86a, Osser92, Dandamudi96a,b].

Stratégie receveur-initiative – La stratégie *receveur-initiative* est exécutée par un noeud légèrement chargé. Quand la charge d'un noeud est au-dessous du seuil minimal de charge, il demande à recevoir des processus à partir des noeuds surchargés. L'obtention du processus se fait par des méthodes similaires à celles utilisées dans les algorithmes source-initiative [Eager86a, Osser92, Dandamudi96a,b].

La différence entre les deux types de stratégie, source-initiative et receveur-initiative, réside dans le fait que dans le premier cas, les décisions de distribution de charge sont habituellement prises au moment de l'arrivée d'une tâche, alors que dans le second cas, les décisions sont prises lorsqu'une tâche se termine. La stratégie source-initiative produit de meilleurs temps de réponse quand la charge du système est basse, et la stratégie receveur-initiative donne de bons résultats quand la charge du système est haute [Eager86a]. Une stratégie hybride consiste à utiliser la stratégie source-initiative quand la charge du système

est basse ou moyenne, et la stratégie receveur-initiative quand elle devient excessive. Cette stratégie est souvent appelée stratégie *symétrique*.

2.2.2. Politiques de distribution de charge

Typiquement, un système de distribution de charge comprend quatre politiques : *la politique d'information*, *la politique de localisation*, *la politique de sélection*, et *la politique de transfert*.

2.2.2.1 Politique d'information

Cette politique est responsable de définir : (1) l'information d'état des noeuds qui doit être collectée, (2) l'instant pendant lequel cette information sera collectée, et (3) à partir de quels noeuds elle sera collectée. Elle est aussi responsable de la dissémination de l'information de charge de chaque nœud. La politique d'information peut être classifiée en trois types, bien que des versions hybrides de ces types puissent exister :

Politique conduite par la demande – Un noeud collecte l'information d'état des autres noeuds quand il veut faire le transfert d'une tâche. La collecte de l'information d'état est déclenchée par la politique de transfert. Les mécanismes utilisés pour la collecte de cette information sont l'exploration (probing) et les enchères. Avec la méthode d'exploration, un noeud désirant participer au transfert d'une tâche choisit un autre noeud et vérifie si celui-ci peut participer ou non à la redistribution de la charge de travail. Cette procédure est répétée jusqu'à ce qu'un noeud cible soit trouvé ou le nombre d'explorations effectuées atteigne un nombre maximal d'explorations défini auparavant. Avec la méthode des enchères, une demande d'offres est envoyée à un groupe de noeuds, et les offres sont reçues des noeuds désirant participer au transfert. Les offres sont alors évaluées pour choisir un noeud approprié [Casavant88b].

Politique périodique – Cette politique consiste à collecter l'information d'état périodiquement. Elle peut être centralisée ou distribuée. La politique de transfert peut décider le transfert des tâches en fonction de l'information collectée. L'image que chaque noeud a sur l'état du système peut ne pas correspondre à l'état réel du système en raison des délais des communications réseau et de la collecte périodique de l'information d'état. Cette politique

n'adapte pas généralement son taux d'activité à l'état du système. Par exemple, à des charges élevées, les avantages résultants de la distribution de charge sont minimes car tous les noeuds sont occupés. Cependant, la collecte périodique de l'information d'état ne fait qu'imposer une charge supplémentaire au système.

Politique conduite par le changement d'état – Avec cette politique, un noeud dissémine l'information concernant son état quand celui-ci change d'un certain degré. Cette politique diffère de la politique conduite par la demande dans le sens qu'elle dissémine l'information d'état d'un noeud plutôt que de faire la collecte de l'information des autres noeuds. Si elle est centralisée, alors l'information d'état est envoyée à un noeud central. Si elle est distribuée, elle est envoyée aux autres noeuds.

2.2.2.2 Politique de localisation

Cette politique est responsable de trouver pour un noeud donné un partenaire convenable (source ou receveur) une fois que la politique de transfert a décidé que ce noeud est source ou receveur. Dans une politique centralisée, pour trouver un partenaire convenable pour la distribution de charge, un nœud doit s'adresser au *coordinateur* qui collecte l'information sur le système (ceci est la tâche de la politique d'information). La politique de transfert utilise ces informations pour choisir des noeuds sources ou receveurs.

Une politique de localisation distribuée largement utilisée emploie *l'interrogation* (polling) pour trouver un noeud convenable. C'est-à-dire qu'un noeud interroge un autre noeud pour déterminer s'il est convenable ou non. Les noeuds peuvent être interrogés aussi bien en séquentiel qu'en parallèle (multicast). Une alternative à l'interrogation est la *diffusion* (broadcast), dans laquelle une requête est diffusée aux autres noeuds dans le but de rechercher un noeud disponible pour la distribution de charge.

Certaines politiques de localisation utilisent des modèles probabilistes au lieu de modèles basés sur l'état des noeuds. Ces modèles probabilistes distribuent les tâches selon un ensemble de règles pré-définies (vecteurs de probabilités). Des études ont montré que les politiques de localisation basées sur l'état du système offrent de meilleurs résultats que leurs contreparties probabilistes [Ryou93].

La politique de localisation peut prendre en considération certaines restrictions lors de la recherche d'un noeud partenaire. Ces restrictions peuvent inclure : les besoins en ressources, la précédence des tâches [Becker94], et l'affinité aux données. L'affinité aux données signifie que les tâches peuvent avoir besoin de certaines données qui doivent être extraites à partir de noeuds distants [Riedl96].

2.2.2.3 Politique de sélection

Une fois que la politique de transfert décide qu'un noeud est source (surchargé), alors la politique de sélection est responsable du choix des tâches à transférer. Nous distinguons quatre méthodes principales de sélection des tâches candidates au transfert :

(1) choisir une des tâches ayant contribué à ce que le noeud devienne surchargé.

(2) choisir n'importe quelle tâche – Toutes les tâches sont considérées comme éligibles pour la distribution de charge. Aucun filtrage des tâches n'est effectué.

(3) choisir une tâche raisonnable – Le filtrage des tâches est effectué pour éliminer celles qui ne conviennent pas au transfert. Ces tâches peuvent être rejetées de deux façons : 1) statiquement : par analyse des traces du système. 2) dynamiquement : un enregistrement dynamique du comportement des tâches peut être maintenu, et les tâches ayant mal répondu à l'exécution distante peuvent être évitées dans le futur.

(4) choisir une tâche appropriée – Le choix d'une tâche appropriée peut nécessiter une bonne connaissance sur la tâche aussi bien que sur les machines destinataires. Les systèmes utilisant la migration peuvent obtenir l'information sur le comportement de la tâche courante au moyen de la surveillance de chaque tâche lors de son exécution. La sélection des bons candidats peut être effectuée en se référant à ce comportement. Pour les systèmes effectuant le placement initial, les bons candidats peuvent être sélectionnés à l'aide de la prédiction et de la classification.

2.2.2.4 Politique de transfert

La politique de transfert détermine si un noeud est dans un état approprié pour participer à un transfert de tâche comme source ou comme receveur. La politique de transfert peut être basée sur des seuils ou être relative.

La politique de seuil décide qu'un noeud est source, si son indice de charge excède un seuil supérieur T_s, ou receveur, si son indice de charge est au-dessous d'un seuil inférieur T_i. Le choix de ces seuils est fondamental pour la performance de l'algorithme de distribution de charge [Shivaratri92]. Évidemment, les meilleures valeurs de ces seuils dépendent de la charge du système et du coût de transfert d'une tâche. Lorsque la charge est basse ou lorsque les coûts de transfert sont bas, les seuils doivent favoriser le transfert des tâches. Par contre, lorsque la charge est élevée ou lorsque les coûts de transfert sont élevés, l'exécution à distance doit être évitée. Eager et al. [Eager86b] affirment que le seuil optimal n'est pas très sensible à la charge du système. Pulidas et al. [Pulidas87] ont présenté des techniques qui adaptent dynamiquement et de manière efficace le seuil à la charge du système.

La politique de transfert relative considère la différence entre la charge d'un noeud et les charges des autres noeuds dans son domaine. Les noeuds sont considérés capables de participer à un transfert si leurs charges diffèrent d'une valeur supérieure à un certain seuil. Ils peuvent alors transférer un nombre fixe de tâches ou une fraction de la différence de charge [Shivaratri92, Casavant88b].

La politique de transfert peut être périodique ou déclenchée par événement. Cependant, les politiques proposées dans la littérature sont dans la grande majorité des cas des politiques déclenchées par événement (fin de traitement d'une tâche ou arrivée d'une tâche).

2.2.3. Aperçu des travaux de recherche

Dans cette section, nous présentons l'évolution des travaux de recherche en matière de distribution de charge dans les systèmes distribués classiques, en ce qui concerne plus particulièrement la seconde classification présentée à la section 2.2.1.2 et la caractérisation de la charge de travail. Ces travaux de recherche ont conduit à des implémentations que nous présentons à la section 2.2.4.

2.2.3.1 Approche statique

Les premières recherches en matière de distribution de charge dans les systèmes distribués étaient centrées sur les algorithmes statiques pour l'ordonnancement des tâches sur les machines du système suivant une distribution pré-calculée. Cela implique clairement moins

de frais généraux du processeur lors de l'exécution. Alors que des améliorations en performance ont été reportées, quelques inconvénients ont été cependant notés. Par exemple, les variations à court terme dans la charge de travail ne sont pas représentées et les déséquilibres de charge résultants limitaient les gains en performance. L'ordonnancement optimal des tâches dépend de la connaissance anticipée sur l'exécution des tâches et sur le temps d'arrivée des tâches qui n'est pas souvent disponible. Finalement, le calcul de l'ordonnancement optimal n'est pas trivial même quand toute l'information requise est disponible [Shirazi90, Lo88].

2.2.3.2 Approche dynamique

La recherche est passée ensuite à l'étude des algorithmes dynamiques dans lesquels les décisions d'ordonnancement sont basées sur les conditions de charge courante. Beaucoup de modélisations analytiques et de simulations ont été faites en utilisant les modèles de la théorie des files d'attente. Chaque noeud est modélisé comme un réseau de files d'attente. Le modèle M/M/1 est fréquemment utilisé en raison de sa simplicité[4].

Eager et al. [Eager86a] ont étudié deux algorithmes dynamiques dans lesquels les tâches sont transférées quand elles parviennent à une machine ayant une charge au-dessus d'un seuil donné. La tâche est envoyée ou bien à un noeud choisi de manière *aléatoire* ou au noeud *le moins chargé*. Ils ont conclu que des algorithmes relativement simples pourraient fournir des gains en performance intéressants, et que les algorithmes qui sont plus complexes fourniraient probablement un gain additionnel qui serait marginal, en raison des coûts inhérents très élevés.

La simulation a été aussi largement utilisée. Beaucoup de chercheurs ont eu tendance à comparer leurs algorithmes avec ceux des travaux largement connus comme ceux de Eager et al. [Eager86b] et ceux de Zhou et al. [Zhou88].

4 Le premier 'M' réfère à la loi de distribution exponentielle du temps entre les arrivées à la file, le deuxième 'M' indique que la distribution du temps de service est exponentielle, et le '1' indique que le réseau de file d'attente comprend un seul serveur.

2.2.3.3 Caractérisation de la charge de travail

Les distributions probabilistes ont été largement utilisées pour la modélisation des lois d'arrivée des tâches et des demandes de ressources. Souvent, les distributions du temps d'arrivée des tâches et des demandes de service sont supposées obéir à une distribution de Poisson. Les charges de travail sont généralement modélisées comme étant homogènes, et toutes les machines sont supposées recevoir des demandes de service intenses de manière similaire. Cependant, l'étude des charges de travail traitées par des systèmes réels laisse croire que les hypothèses précédentes pourraient être erronées [Cabrera86, Leland86, Zhou87, Svensson90, Harchol-Balter96].

L'utilisation des données de trace pour conduire des simulations, comme dans les études séparées de Zhou [Zhou88] et Leland [Leland86], est une approche plus fiable que l'approche probabiliste. Cependant, les données de trace d'un système ne peuvent pas être applicables à d'autres systèmes. D'autre part, les difficultés à obtenir les données de trace ont contribué à la faible adoption de cette approche.

L'exécution d'un algorithme de distribution de charge implique des coûts. Plus l'algorithme est complexe, plus le coût est élevé et plus grande est la performance qui doit être offerte pour compenser ces coûts. Eager et al. [Eager86b] affirment que des algorithmes excessivement complexes impliquent des coûts inacceptables. Leurs besoins en information d'état nécessitent un grand nombre de messages à échanger qui pourraient encombrer les noeuds et causer l'instabilité du système. Ils ont conclu que :

> [5] ''Nous concluons que la distribution de charge simple et adaptative est d'une valeur pratique considérable, et qu'il n'y a aucune preuve ferme que les coûts potentiels de collecte et d'utilisation de grandes quantités d'information d'état sont justifiées par des gains potentiels''.

Ils ont utilisé une distribution hyper-exponentielle dans une investigation sur les gains possibles de la distribution de charge avec migration de processus actifs [Eager88]. Ils ont conclu cette fois qu'il n'y a probablement aucune condition sous laquelle la migration

[5] Eager, Lazowska et Zahorjan

pourrait apporter des améliorations de performance supérieures à celles offertes par les approches de distribution de charge non migratoires. Sous certaines conditions assez extrêmes, la migration peut offrir des améliorations modestes en performance. Downey et Harchol-Balter ont conclu dans [Downey95] que les résultats précédents de Eager et al. ne s'appliquent pas à des systèmes réels, car ils sont basés sur l'analyse et la simulation avec des charges de travail synthétiques. Ils ont conclu dans [Harchol-Balter96] que :

> [6] ''L'amélioration en performance de la migration avec préemption par rapport au transfert sans préemption est typiquement entre 35% et 50%. Cette amélioration est un peu plus grande que celle prévue par des modèles analytiques précédents [Eager88]. La raison principale de cette non-conformité est que le modèle de charge de travail requis par l'analyse théorétique des files d'attente ne décrit pas des charges de travail réelles. En réalité, la variance des durées de vie des tâches est plus grande que celle des distributions exponentielles et hyper-exponentielles utilisées, et les temps entre les arrivées de tâches sont plus corrélés que les arrivées selon une loi de Poisson''.

2.2.4. Implémentations

De nombreuses implémentations de la distribution de charge ont été reportées dans la littérature. Les premiers systèmes [Hwang82, Bershad85, Hagmann86] étaient utilisés pour placer à distance des processus de longue durée sur des machines légèrement chargées. Un système un peu plus avancé est le système *Condor* [Litzkow88, Litzkow92, Condor]. Plus tard, certains systèmes ont fourni des facilités de migration de processus, souvent avec le support du placement des processus. Ces systèmes incluent *Charlotte* [Artsy86, Artsy89], *Sprite* [Douglis87, Douglis91, Ousterhout88], *V* [Theimer85], et le système *Mosix* [Barak89, Barak93, Barak85a, Barak85b]. Des systèmes commerciaux ont également fait apparition ces dernières années et incluent *LSF* [Zhou93, Platform94] de Platform Computing Corporation, *LoadLeveler* [IBM96] d'IBM Corporation, *Codine* [Genias] de Genias Software, etc. Dans ce qui suit nous présentons brièvement les systèmes Condor, LSF, et Mosix qui représentent des

[6] Harchol-Balter et Downey.

produits en matière de distribution de charge largement utilisés dans des environnements hétérogènes. La caractéristique commune de ces trois produits est qu'ils sont très orientés vers les environnements de traitement en grappes de machines (cluster computing environnements).

2.2.4.1 Condor

Condor est un système de distribution de charge qui permet d'exploiter le temps d'inoccupation des machines en exécutant des travaux du type batch sur des machines inoccupées. Les caractéristiques essentielles de Condor sont la localisation et l'allocation automatique des machines inoccupées, le checkpointing et la migration de processus. Toutes ces caractéristiques sont réalisées sans aucune modification au noyau Unix sous-jacent.

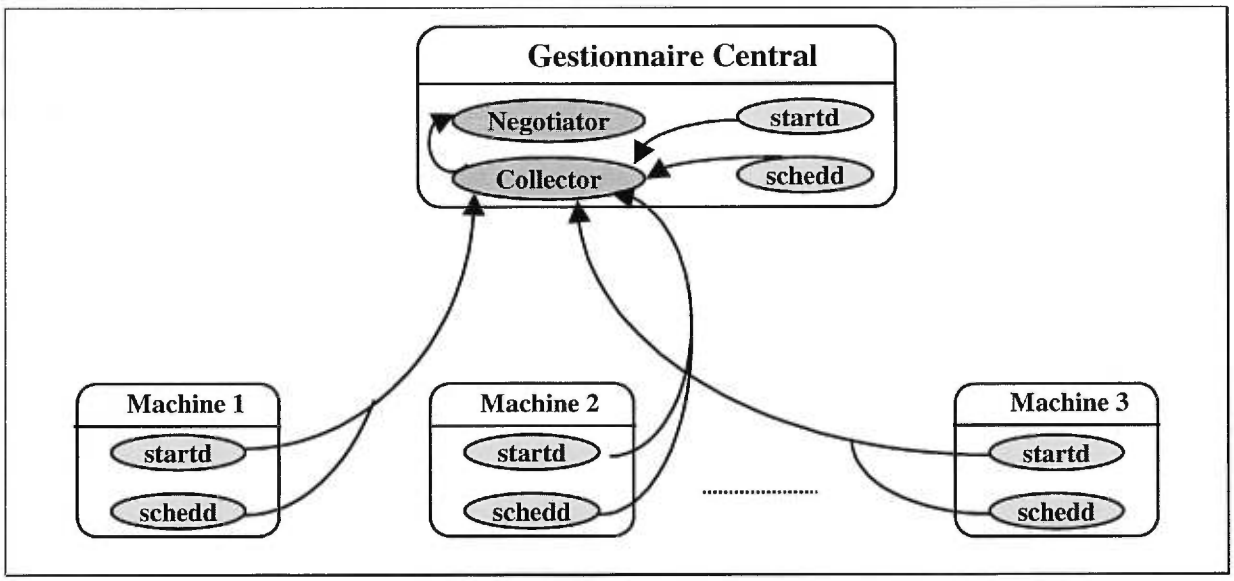


Figure 2.1 - Architecture d'un groupe Condor [Condor]

Il n'est pas nécessaire de modifier son code source pour s'exécuter sous Condor. Cependant, les programmes doivent être liés avec des librairies de Condor. L'algorithme d'ordonnancement centralisé de Condor est présenté dans [Mutka87], analysé dans [Litzkow88], et récapitulé dans [Nuttal94].

Condor surveille l'activité de toutes les machines du réseau. Les machines déterminées comme étant inoccupées sont placées dans un groupe de ressources (resource pool). Les machines de ce groupe sont ensuite allouées pour l'exécution des tâches. Le groupe est une

entité dynamique dans laquelle les machines entrent lorsqu'elles deviennent inoccupées et qu'elles quittent lorsqu'elles deviennent occupées à nouveau. Condor fonctionne très bien comme un coordinateur des applications formées d'un seul processus de longue durée. La figure 2.1 schématise l'architecture d'un groupe Condor.

2.2.4.2 LSF

LSF est un système commercial de distribution de charge et de gestion de files d'attente pour les travaux du type batch dans un environnement Unix hétérogène [Zhou93, Platform94]. LSF gère le traitement des tâches en offrant aux utilisateurs une seule vue des ressources matérielles et logicielles indépendamment des machines du groupe (grappe ou cluster). LSF supporte les tâches du type batch, interactif, et parallèle, et les gère dans un groupe de machines en exploitant les machines et les serveurs inoccupés. Spécifiquement, LSF offre les services suivants : (1) ordonnancement et gestion des tâches batch dans des files d'attente, (2) distribution des tâches à travers le réseau, (3) distribution de la charge sur les machines du réseau, (4) accès transparent aux ressources hétérogènes du réseau, et (5) surveillance de la charge du réseau.

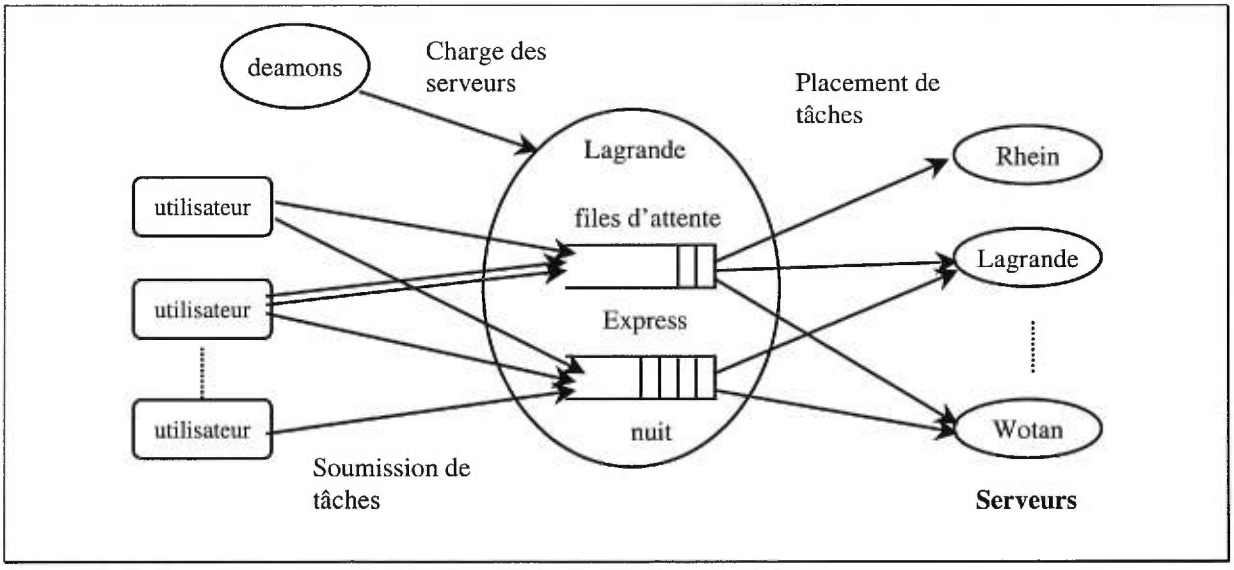


Figure 2.2 - Soumission des travaux batch dans LSF

LSF comporte un ensemble d'outils dont le but est de répartir la charge CPU sur les machines d'un groupe et ceci de manière transparente pour les utilisateurs. Le même environnement d'exécution (ID utilisateur, variables d'environnement, répertoire de travail, signaux,

paramètres du terminal, etc.) est maintenu lorsque les tâches sont envoyées pour exécution à distance. L'outil *lsbatch* permet la soumission des tâches en mode batch sur un groupe de machines via des files d'attente. *lsbatch* choisit la machine la moins chargée qui possède les ressources demandées. La figure 2.2 schématise le principe de fonctionnement de l'outil *lsbatch*.

2.2.4.3 Mosix

Mosix est un module logiciel pour le support du calcul de groupe (cluster computing). Mosix permet à n'importe quel groupe de machines de travailler de manière coopérative comme si elles constituaient des parties d'un même système. Avec Mosix, il n'y a aucun besoin de modifier les applications de l'utilisateur, de les lier avec des bibliothèques systèmes, ou même d'assigner des processus aux différents noeuds car Mosix le fait automatiquement.

Le noyau de Mosix comprend des algorithmes adaptatifs d'équilibrage de charge et de gestion de la mémoire qui répondent aux variations dans l'utilisation des ressources du groupe, pour maximiser la performance globale du système. Ces algorithmes utilisent la migration de processus avec préemption pour assigner et réassigner les processus parmi les noeuds, afin de profiter des meilleures ressources disponibles.

Comparaison

Les systèmes Condor et LSF présentent des caractéristiques comparables. Ils ont été portés à une variété de systèmes d'exploitation et ils implémentent la migration de processus au niveau de l'espace utilisateur. Le checkpointing utilisé dans Condor et dans LSF est particulièrement utile pour les processus de longue durée. Condor et LSF nécessitent l'édition des liens des programmes avec une librairie du système en question. Cependant, ils ne nécessitent aucune modification au noyau du système d'exploitation. Condor ne supporte pas les processus utilisant les signaux, les temporisateurs (timers), les librairies partagées, et les IPC (inter-process communication). La migration dans Condor implique la génération d'un fichier *core* du processus. Ce fichier est envoyé avec l'exécutable à la machine cible. Contrairement à Condor, LSF ne requiert pas la génération de ce fichier *core* et supporte les signaux.

Le système Mosix fait partie des systèmes d'exploitation semblables à Unix (Unix-like). Contrairement à Condor et à LSF, Mosix implémente la migration de processus au niveau du noyau. Il offre une seule image système en présentant au processus une vue uniforme du système de fichiers, des périphériques, et des facilités réseau indépendamment de la localisation actuelle du processus. De plus, il n'est pas nécessaire de faire l'édition des liens des programmes avec des librairies spécifiques pour pouvoir s'exécuter sous Mosix.

2.3. Distribution de charge dans les systèmes distribués à base d'objets

Le progrès réalisé dans le développement de systèmes d'exploitation distribués a permis de concevoir un environnement dans lequel les programmes composés d'un ensemble de modules peuvent s'exécuter concurremment dans un ensemble de machines. D'autre part, l'essor connu par la programmation orientée objet encourage une méthodologie de conception et de création de programmes comme une collection de composantes autonomes. L'amalgamation des deux concepts, systèmes distribués et paradigme objet, a permis d'avoir une nouvelle génération de systèmes appelés *systèmes distribués à base d'objets* tirant profit des avantages de chacun des deux concepts [Chin91]. L'élément commun de ces systèmes est le concept d'objet, ayant un état et des opérations sur cet état, souvent considéré comme l'unité de distribution contrairement aux systèmes classiques dans lesquels l'unité de distribution est le processus.

Dans ces systèmes, la distribution de charge peut toujours être réalisée au niveau processus car les objets sont encapsulés dans des processus pour pouvoir s'exécuter. Par conséquent, les approches et les politiques de distribution de charge, présentées aux sections 2.2.1 et 2.2.2, restent toujours applicables à ce niveau. Nous nous intéressons dans cette section à une autre forme de distribution de charge qui est celle au niveau objet qui a des potentialités d'offrir de bons résultats en tirant profit des spécificités de ces systèmes

2.3.1. Approches de distribution de charge

La classification, que nous avons présentée à la section 2.2.1, des approches de distribution de charge dans les systèmes distribués classiques reste aussi valable au niveau des systèmes distribués objet bien que cette fois l'unité de distribution soit l'objet ou l'invocation d'une méthode de l'objet. On distingue trois autres approches spécifiques aux systèmes distribués à

base d'objets : *le placement d'objets*, *la migration d'objets*, et *l'assignation d'invocations de méthodes*.

2.3.1.1 Placement d'objets

Le placement d'objet est un aspect très important dans les applications distribuées objet car il peut considérablement influencer le comportement des applications distribuées en matière de performances et de services qu'elles offrent à leurs utilisateurs. Un placement efficace des objets dans des environnements distribués à grande échelle peut être difficile à réaliser car le comportement des objets est souvent imprévisible et peut être influencé par plusieurs facteurs tels que : (1) l'hétérogénéité au niveau matériel et logiciel des systèmes sous-jacents, (2) les capacités des différentes ressources qui forment le système (vitesse des processeurs, mémoire, entrées/sorties, etc.), (3) la dynamique du système (disponibilité des ressources, interactions entre objets, etc.), et (4) l'hétérogénéité des modèles objets et des modèles d'exécution (structures des objets, granularité, persistance, groupage, composition, etc.). Ces facteurs sont souvent exprimés comme des contraintes dans les décisions de placement.

L'objectif du placement d'objet consiste à optimiser l'exécution des applications distribuées orientées objet en plaçant leurs différentes composantes dans les divers nœuds du système de sorte que l'accès aux ressources matérielles et logicielles soit plus facile et plus efficace. Ces applications sont formées d'un ensemble d'objets interagissant entre eux par invocation de méthodes de manière comparable aux interactions dans une approche client/serveur. Les deux objets client et serveur peuvent être dans un même site ou dans des sites distants. Ainsi, le placement des objets a un impact direct sur les performances des applications. Parmi les autres objectifs du placement d'objet on peut citer la minimisation des temps de réponse des requêtes clientes en offrant à chaque objet un environnement dans lequel il puisse s'exécuter dans les meilleures conditions, la minimisation du trafic réseau, et la distribution de la charge entre les différents nœuds d'un système.

Peu de travaux ont étudié le placement d'objets dans un but de distribution de charge entre les nœuds d'un système distribué. L'approche proposée dans [Chatonnay96] repose sur un modèle relationnel, décrivant les interactions entre objets, et sur la dérive des connaissances. Cette approche repose sur l'utilisation d'un gestionnaire de placement par site qui est chargé

de la gestion des grappes locales, du maintien de l'information de charge locale et des sites distants avec lesquels les grappes locales ont des relations, et finalement de la prise des décisions relatives à la réorganisation des objets. Le gestionnaire de placement effectue des traitements périodiques, tels que le calcul des débits et le traitement de la réorganisation des objets, et d'autres traitements tels que la création et la suppression de grappes.

2.3.1.2 Migration d'objets

Comme le placement, la migration d'objets est un des défis ouverts dans les systèmes distribués. La migration d'un objet peut influencer sa performance et le comportement des autres objets de l'application. La migration d'objet implique le déplacement d'un objet d'un noeud source à un noeud destinataire. L'objet déplacé devient local au noeud destinataire. Les objectifs de la migration sont étroitement liés au type d'applications utilisant la migration. Ces objectifs incluent :

L'accès à plus de puissance de calcul – C'est l'un des objectifs de la migration lorsqu'elle est utilisée pour permettre la distribution de charge. Un objet est déplacé d'un noeud surchargé à un autre noeud ayant suffisamment de ressources et de capacité de traitement. Nous avons vu à la section 2.2.4 des exemples de systèmes distribués à base de processus utilisant la migration de processus comme moyen de distribution de charge.

La continuité de service – Il peut être parfois nécessaire de migrer des objets à d'autres emplacements pour assurer la continuité de service en effectuant des tâches administratives comme l'arrêt d'une machine pour effectuer la mise à niveau du matériel et/ou du logiciel.

L'optimisation des coûts de communication – Les objets peuvent être déplacés à d'autres noeuds pour optimiser les coûts de communication entre objets distants et diminuer le nombre d'invocations distantes de méthodes. Les objets ayant une forte interaction entre eux sont normalement placés dans un même site (ou dans une même grappe) pour réduire les coûts d'interactions distantes. Généralement, cette optimisation des communications est réalisée en conjonction avec la distribution de la charge comme dans [Chatonney98].

La tolérance aux fautes – Les objets peuvent être répliqués dans de multiples nœuds. Quand un des nœuds contenant un objet tombe en panne, le système peut continuer à fonctionner

normalement en utilisant les objets situés dans d'autres nœuds. De plus, les objets peuvent être déplacés à d'autres nœuds lors de la dégradation d'un nœud en cours de panne.

2.3.1.2.1 Étapes de la migration

Comme nous allons voir à la section 2.3.2, les modèles objets associés aux systèmes distribués à base d'objets sont très variés. Par conséquent, la migration d'objet est plus ou moins complexe selon le type d'objets supportés par le système. Par exemple, la migration d'un objet actif implique la migration du processus qui lui est associé. Ceci introduit tous les concepts de la migration de processus qui impliquent le transfert du contexte d'exécution du processus ainsi que l'état du noyau (contenu des registres, fichiers ouverts, ID du processus, ID utilisateur, répertoire courant de travail, références aux processus fils et père, etc.). La migration des objets ayant un état nécessite la sauvegarde et le transfert de cet état. Les étapes de la migration d'objet peuvent être résumées comme suit :

1. ***Envoi d'une requête de migration à un nœud*** – Après négociation, la migration est acceptée.
2. ***Détachement d'un objet de son nœud source*** – L'exécution de l'objet est suspendue et il est déclaré dans un état de migration. Les canaux de communication de l'objet sont aussi temporairement suspendus.
3. ***Routage des communications*** – Les messages envoyés à l'objet en cours de migration sont collectés et lui sont délivrés après migration. L'objet ne peut pas invoquer d'autres objets à cette phase. Après migration, les canaux de communication sont rétablis et l'objet devient connu au monde externe.
4. ***Extraction de l'état de l'objet*** – L'état de l'objet est gardé localement au nœud source jusqu'à la fin de la migration. Si l'objet est actif, l'état du processus qui lui est associé est également extrait. Typiquement, cet état comprend le contenu de la mémoire, le contenu des registres, les fichiers ouverts, les canaux de communication, et tout le contexte du noyau.
5. ***Création d'une instance de l'objet dans le nœud destination*** – Cette instance n'est activée qu'après transfert de l'état de l'objet à partir du nœud source.
6. ***Transfert de l'état de l'objet vers la nouvelle instance***.

7. ***Activation de la nouvelle instance*** – Les messages reçus lors de la migration sont délivrés à la nouvelle instance de l'objet.

2.3.1.2.2 Problèmes à résoudre

Pour supporter la migration d'objet de manière efficace, un système doit offrir certaines fonctionnalités et résoudre certains problèmes :

L'importation et l'exportation de l'état de l'objet – Le système doit offrir des interfaces permettant d'exporter l'état d'un objet à partir d'un nœud source vers un nœud destination. Ces interfaces peuvent être offertes par le système d'exploitation sous-jacent ou par le langage de programmation.

Canaux de communication – Tous les canaux de communication d'un objet avec ses clients doivent être fermés avant la migration de l'objet et rouverts après sa migration. Tous les messages reçus lors de la migration doivent être traités. Ceci peut être accompli par une procuration locale (ou proxy) de l'objet au noeud source qui sauvegarde les messages reçus dans une pile et qui les achemine à la nouvelle instance créée une fois que les canaux de communication sont rétablis.

Localisation de l'objet – Le système peut opter de cacher ou non aux objets clients la nouvelle localisation de l'objet migré. Lorsqu'elle est cachée, le système doit acheminer tous les messages reçus à l'ancienne localisation de l'objet vers sa nouvelle localisation.

Nommage et accès à l'objet – L'objet déplacé doit être accessible par le même nom et par les mêmes mécanismes comme s'il n'y avait pas de migration. Il en est de même pour les ressources associées à l'objet telles que des fils d'activité, canaux de communication, fichiers et périphériques. En général, la transparence complète de nommage et d'accès à l'objet est difficile à réaliser.

Hétérogénéité – Avant la migration d'un objet entre différentes machines, il est nécessaire de convertir aussi bien la représentation des données que le code de l'objet dans un format compréhensible par la machine de destination. Si la migration s'effectue entre des machines ayant des architectures hétérogènes, il est essentiel de s'assurer que la définition de la classe correspondante de l'objet existe dans l'architecture de destination. Sinon, la migration de

l'objet impliquerait aussi la migration de sa classe et la compilation de la classe sur la machine destination.

2.3.1.2.3 Migration et distribution de charge

La migration de processus a été utilisée dans peu de travaux, et plus particulièrement dans certains systèmes d'exploitation distribués, tels que Mosix [Barak93], Sprite [Douglis87, Douglis91], et Condor [Litzkow88, Litzkow92], dans le but de réaliser la distribution de charge dans un environnement homogène. La majorité des travaux considèrent que le code de la tâche à migrer est disponible sous une forme binaire dans le noeud de destination. Nous avons vu à la section 2.2.3.3 les conclusions de Eager et al. [Eager88] à propos des gains minimes en performance des approches utilisant la migration de processus.

Au niveau des systèmes distribués à base d'objets présentés à la section 2.3.2, la migration d'objet est utilisée principalement dans certains systèmes dans le but d'optimiser les coûts de communication. La migration d'objet, lorsqu'elle est utilisée dans le but de réaliser la distribution de charge, comprend typiquement quatre politiques, que nous avons décrit à la section 2.2.2 : la politique d'information qui est responsable de déterminer quelle information d'état des nœuds doit être collectée, et où et quand cette information doit être collectée, la politique de localisation qui est responsable de la sélection d'un nœud de destination où un nœud source pourra transférer un objet, la politique de sélection qui est responsable de la sélection d'un objet candidat à la migration, une fois qu'un nœud de destination a été identifié, et la politique de transfert qui est responsable de l'établissement du niveau de charge à partir duquel la distribution de charge peut avoir lieu.

Peu de travaux ont reporté les performances de la distribution de charge par la migration d'objet. Jensen [Jensen96] a fait la comparaison des performances obtenues en considérant différentes granularités des objets candidats à la migration dans un environnement objet distribué COOLv2 [Jacquemot94]. Les résultats obtenus montrent que de meilleures performances sont obtenues avec des objets ayant une granularité grande.

2.3.1.3 Placement de requêtes

Le placement (ou assignation) de requêtes est un aspect important dans les environnements avec des objets répliqués et dans lesquels les requêtes des clients peuvent être servies par

plusieurs objets. Le but du placement de requêtes est d'assigner les requêtes aux serveurs pour améliorer la performance du système (débit et disponibilité des serveurs) et la qualité du service offert aux utilisateurs. L'assignation de requêtes est dans une certaine mesure semblable au placement des tâches sur les noeuds du système. Le placement statique affecte les requêtes d'un utilisateur à un ensemble d'objets sans tenir compte de l'état actuel du système. Par contre, dans le placement dynamique les décisions de placement sont prises en tenant compte de l'information dynamique sur l'état du système (charge des serveurs, charge réseau, etc.). Dans le chapitre 3, nous étudions un peu plus en détail l'assignation de requêtes dans les systèmes distribués à bases d'objets.

2.3.2. Implémentations

Dans cette section, nous présentons des exemples de systèmes distribués à base d'objets qui offrent des facilités pour la réalisation de la distribution de charge, en décrivant brièvement les modèles objets associés et les approches utilisées. Le tableau 2.1 à la fin de cette section récapitule pour chaque système les approches utilisées et les grandes caractéristiques du modèle objet associé. Ces caractéristiques sont principalement : *le degré d'activité de l'objet*, *la granularité de l'objet*, et *le groupage dynamique des objets*.

Le degré d'activité d'un objet caractérise le niveau de concurrence mis en oeuvre par l'objet. Un objet est dit *actif* lorsqu'il a son propre processus. Au contraire, un objet est dit *passif* lorsqu'il ne contient pas son propre processus. Il répond seulement aux demandes d'autres objets.

La granularité est la taille relative ou le niveau de détail qui caractérise un objet. Le volume de traitement effectué par un objet caractérise aussi sa granularité. On distingue les objets à granularité grande, à granularité moyenne, et à granularité fine. Les objets à granularité grande sont caractérisés par leurs grandes tailles, le grand nombre d'instructions qu'ils exécutent pour répondre aux invocations, et le peu d'interactions qu'ils ont avec les autres objets. Ils résident typiquement dans leurs propres espaces d'adressages [Chin91].

Le groupage d'objets permet un accès efficace aux objets en utilisant le concept de *grappe* (*cluster*) pour grouper des objets dans des objets à granularité grande (ou processus). La grappe est l'unité de transfert entre la mémoire permanente et la mémoire centrale. Elle est

assignée à un nœud à la fois. La grappe est l'unité de distribution et d'assignation dans plusieurs systèmes [Jensen96, Banatre95]. Le groupage dynamique signifie que chaque grappe contient un groupe d'objets qui peut varier dynamiquement par la création de nouveaux objets, par le ramasse miettes d'objets, et par la migration d'objets entre les groupes [Gourhant92].

2.3.2.1 Emerald

Emerald est un langage et un environnement de programmation pour le support des systèmes distribués [Ragendra91, Jul89, Jul93]. Le langage Emerald est un langage orienté objet conçu pour permettre la réalisation d'applications distribuées. Les objets locaux et distants sont manipulés et définis de manière similaire. Ils sont décrits en utilisant un modèle sémantique uniforme, et communiquent en utilisant des invocations d'objets. Un objet est identifié par un nom unique à travers le réseau. Il peut comporter plusieurs fils d'exécution synchronisés au moyen de moniteurs. Le langage Emerald fournit des mécanismes permettant le mouvement des objets, à granularité fine et grande, dans un système localement distribué. L'unité de la mobilité dans les systèmes précédents a typiquement été un processus (espace d'adressage). Emerald supporte la mobilité d'objets à grain fin et la migration de processus [Jul87]. Les primitives permettant la mobilité dans Emerald sont:

- Locate X - retourne le noeud où X réside
- Move X to Y – co-localise X avec Y.
- Fix X at Y - fixe l'objet à un noeud.
- Unfix X – Mettre X mobile après avoir été fixé

En plus des modes de passage des paramètres par référence et par valeur, Emerald fournit un autre mode de passage des paramètres appelé *appel par mouvement* (call by move). Dans ce mode, les objets référencés comme arguments peuvent être déplacés. Ce mode est susceptible d'être plus efficace que le mode dans lequel l'appelé fait des exécutions à distance sur des arguments. Les objets sont déplacés pour réduire le coût de transmission et le nombre d'invocations distantes. Du point de vue de l'utilisation CPU, il n'y a aucune mention concernant la distribution de charge dans la littérature au sujet d'Emerald.

2.3.2.2 Amadeus

Amadeus est un environnement distribué orienté objet qui offre le support des objets persistants et distribués [Gourhant92]. Un objet est une entité instanciée dans un contexte (espace d'adressage local). Les objets dans Amadeus sont passifs et sont manipulés par des processus distribués. Le modèle d'exécution dans Amadeus définit une unité à multiples fils d'exécution appelée une *tâche* (job). Les tâches se composent d'un ensemble d'activités qui sont des fils d'exécution distribués. Les programmes d'application sont écrits en utilisant le langage C** qui est une extension du langage C++. Amadeus utilise le groupage des objets dans des grappes (clusters). Une grappe représente l'unité du transfert entre la mémoire et le disque. Quand une tâche requiert un objet, elle charge toute la grappe contenant l'objet dans un contexte approprié. Amadeus supporte le groupage dynamique. C'est-à-dire que chaque grappe contient un groupe d'objets pouvant varier dynamiquement, par création de nouveaux objets dans la grappe, par ramasse-miettes dans la grappe, ou par migration d'objets entre les grappes.

La distribution de charge dans Amadeus a lieu à deux niveaux : lors de la création d'une activité et lors du placement des objets. Quand une activité est créée, la distribution de charge est activée pour lui assigner un noeud. Elle est également activée quand les grappes sont placées dans la mémoire. Les activités n'ont pas un noeud préféré qui leur est assigné et peuvent s'exécuter sur différents noeuds selon l'emplacement des grappes. Tangney et al. [Tangney92] décrivent quelques indications pour le placement des objets pouvant être utilisées par les programmeurs :

- une approche de distribution comprenant une politique *centralisée*, une politique *migratoire*, et une politique de *réplication*. Dans la politique centralisée, un objet est localisé dans une grappe à la fois et d'autres grappes ont des références sur cet objet. Dans la politique migratoire, l'objet est déplacé à la grappe où il est appelé en laissant une référence dans la grappe d'origine. Finalement, avec la politique de réplication, l'objet est répliqué dans chaque grappe au lieu de créer une référence sur lui. Ceci est trivial pour les objets en lecture seulement.
- une opération pour co-localiser les objets : co-locate(obj, obj_1, ..., obj_N) a pour effet de faire migrer les objets obj_1, obj_2, ..., obj_N à la même grappe que obj.

2.3.2.3 Guide-2

Le système d'exploitation *Guide-2* est un prototype de recherche conçu pour supporter le travail et le partage coopératifs d'objets persistants dans un environnement distribué [Balter91, Chevalier95, Jensen96]. Il est implémenté comme une machine virtuelle s'exécutant au-dessus de Mach 3.0. L'unité de base de calcul dans Guide-2 est une *tâche* (job), qui est un processus distribué (espace d'adressage) avec les ressources associées qui sont partagées entre des fils d'exécution séquentiels, analogues aux processus légers, appelés *activités*. Un espace d'adressage local est appelé un *contexte*. Guide-2 supporte un modèle basé sur les objets passifs qui sont accessibles par les activités définies dans le modèle d'exécution. Les objets sont groupés pour améliorer les performances et pour limiter le coût d'allocation des blocs de disque et de pages mémoire. Par défaut, les objets sont placés dans la même grappe que l'objet qui les a créés. Ceci augmente la probabilité de trouver les objets qui interagissent souvent dans la même grappe. La grappe est l'unité de placement et de distribution dans Guide-2, et donc aussi la granularité la plus fine de distribution de charge.

Le service de distribution de charge dans Guide-2 utilise le placement initial des objets. Il est implémenté à l'aide d'un agent local situé sur chacun des noeuds actifs du système. Ces agents locaux se composent d'une composante qui recueille l'information de charge locale et qui la distribue aux autres noeuds, et d'une composante de contrôle qui exécute les décisions de distribution de charge basées sur l'information disponible localement. Des indicateurs de charge sont envoyés à un centre global de l'information de charge, qui redistribue le vecteur global de charge à tous les noeuds. La composante de contrôle est responsable d'imposer trois politiques, à savoir la politique d'initiation (stratégie source-initiative), la politique de sélection (filtrage des grappes) et la politique de localisation (sélection round robin).

Jensen et al. [Jensen96] ont évalué les performances de la distribution de charge dans Guide-2 en considérant différentes granularités des objets. Les résultats obtenus montrent que la distribution de charge avec une granularité grande est meilleure qu'avec une granularité fine des objets.

2.3.2.4 Shadows

Le système *Shadows* offre les mécanismes de base pour l'implémentation des applications distribuées dans les environnements où les objets peuvent être mobiles et où les pannes sont occasionnelles [Caughey93]. Ces mécanismes sont le nommage, la localisation et l'invocation des objets, la persistance, et le ramasse-miettes. L'architecture Shadows est basée sur trois concepts : les serveurs objet, la migration d'objet, et l'invocation d'opération de manière transparente à la localisation de l'objet. Elle est aussi basée sur le modèle client/serveur. Les objets sont des entités qui se composent d'un état interne et d'un ensemble d'opérations par lesquelles cet état peut être consulté et modifié. Un objet est contrôlé à tout moment par un gestionnaire d'objets (un processus serveur avec son propre espace d'adressage). Les objets migrent entre les gestionnaires d'objets.

Les serveurs objets peuvent être instanciés n'importe où dans le réseau (placement initial guidé par le programmeur ou par le système). Les serveurs peuvent être créés automatiquement lorsque le système en détermine le besoin. La figure 2.3 montre le principe d'une invocation distante dans Shadows. La représentation locale du serveur au niveau du client est appelée *shadow*. Le rôle du shadow consiste à transférer les requêtes du client vers le serveur objet. Le client invoque en premier une certaine opération du shadow. Celui-ci effectue un appel RPC à l'objet réel du serveur en envoyant le détail de l'invocation du client. Le résultat est retourné au shadow et ensuite au client.

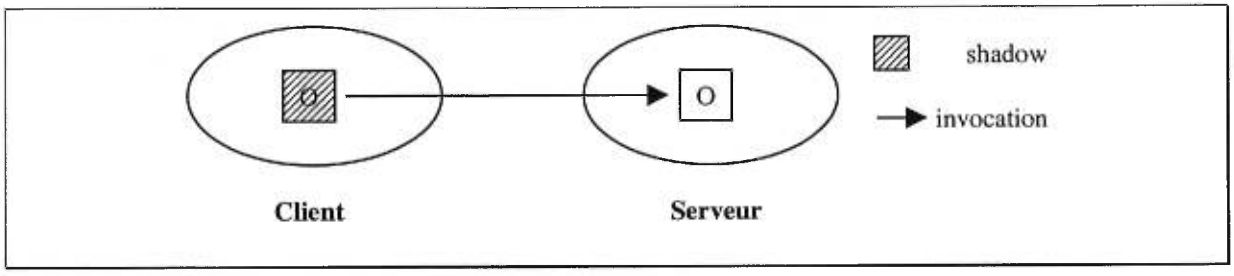


Figure 2.3 - Invocation distante dans Shadows [Caughey93]

La migration d'un objet est réalisée par la création d'une copie de l'objet au noeud destinataire et d'une référence à cette copie au noeud source. Cette référence remplace l'objet après la migration de l'objet. Shadows offre également des mécanismes pour rendre les gestionnaires d'objets mobiles et des mécanismes pour activer le processus de ramasse-

miettes. Caughey et al. [Caughey93] affirment que ces mécanismes peuvent être exploités pour l'implémentation de l'équilibrage de charge.

2.3.2.5 Isatis

Isatis est un environnement d'exécution distribué objet développé à l'IRISA en France [Banatre95]. Le développement d'Isatis a été motivé par le besoin d'avoir une exécution efficace d'applications concurrentes orientées objet par utilisation de la personnalisation. Il est défini en termes d'objets C++, et la personnalisation est réalisée par le mécanisme d'héritage. Les objets dans Isatis sont composés d'un état et d'un ensemble de méthodes qui offrent le seul moyen de manipuler cet état. Les objets sont passifs par défaut. Des objets actifs peuvent être implémentés en utilisant le mécanisme d'appel asynchrone de méthode. Différentes granularités des objets sont également supportées par Isatis. Les objets peuvent également être déclarés explicitement persistants. Le modèle d'exécution d'Isatis est basé sur la notion du délégué de protection (protection delegate) qui est un ensemble de domaines de protection (processus avec de multiples fils d'exécution). Les objets Isatis appartenant à la même application sont placés dans le même délégué de protection.

Le but de la distribution de charge dans Isatis est d'améliorer les performances de l'application. Ceci est atteint au moyen d'une stratégie d'équilibrage de charge par le placement initial des exécutions de méthodes selon la charge des machines et les caractéristiques des objets. Cette stratégie suppose un réseau homogène. Le placement initial des exécutions de méthodes essaie de minimiser le temps d'exécution de la méthode sur le noeud source (exécution locale) ou sur un noeud distant (exécution distante). L'algorithme de distribution de charge comprend deux étapes. La première étape consiste à déterminer un ensemble de noeuds éligibles pour l'exécution de la méthode invoquée. La seconde étape consiste à sélectionner de manière aléatoire un noeud parmi les noeuds éligibles.

2.3.2.6 Legion

Le projet *Legion* à l'Université de Virginia est une tentative de concevoir et de construire les services système qui donnent aux utilisateurs l'illusion d'avoir une seule machine virtuelle

dans un environnement distribué comprenant des groupes de postes de travail, des super ordinateurs, et des super ordinateurs parallèles [Lewis95].

Legion est réalisé comme une machine virtuelle en couches et utilise un contrôle distribué et extensible. Les objets de Legion ont un espace d'adressage, une classe, un nom, et un ensemble de facilités. Ce sont des objets indépendants qui contiennent des objets contenus. Les objets indépendants sont reconnus par le système et sont dans des espaces d'adressage disjoints. Les objets contenus, comme les entiers et les nombres complexes, sont des objets définis par le langage de programmation. Ils sont contenus dans l'espace d'adressage d'un objet indépendant. L'architecture du système est concernée seulement par les objets indépendants et leurs interactions par appel de méthodes. Chaque méthode a une liste formelle de paramètres typés et peut retourner une valeur typée. Chaque paramètre formel peut être ''in'' ou ''out''. Le modèle objet de Legion est décrit en détail dans [Lewis95].

La documentation de ce système ne fournit pas d'information au sujet de la distribution de charge dans Legion. Cependant, le placement d'objet est considéré comme un aspect important dans Legion car il peut considérablement influencer l'exécution d'un objet et sa performance. Le travail de Karpovich [Karpovich96] rentre dans ce cadre et consiste à développer une nouvelle approche pour supporter de manière adéquate le processus de placement dans un système distribué à grande échelle. L'approche proposée consiste à concevoir un cadre d'application pour le support d'une large gamme d'algorithmes de placement plutôt que la simple conception d'un nouvel algorithme.

2.3.2.7 CORBA

(Une description plus détaillée de cette architecture est donnée à l'annexe 1).

L'architecture *CORBA* définit un cadre pour le développement d'applications distribuées orientées objet qui communiquent entre elles comme si elles étaient écrites dans un seul langage de programmation et comme si elles s'exécutaient dans une même machine. CORBA définit une architecture standard pour le *courtier des requêtes objet* (Object Request Broker ou ORB). L'ORB est une composante logicielle qui permet le transfert des messages d'un programme à un objet distant en cachant au programmeur la complexité du réseau de communication sous-jacent. L'ORB permet de créer des objets standards dont les méthodes

peuvent être invoquées par des programmes clients situés n'importe où dans le réseau. L'architecture CORBA supporte plusieurs langages de programmation incluant Java, C++, Smalltalk, etc. Elle définit un ensemble de services standards [OMG97a] qui permettent aux objets des applications distribuées de communiquer d'une façon standard. Ces services incluent le service de nommage, le service de courtage, le service des transactions, le service de sécurité, le service d'événements, etc. Cependant, il n'y a aucun service standard pour la distribution de charge.

Peu de vendeurs d'ORBs, comme Expersoft, Inprise, et Chorus Systèmes affirment que leurs produits (CORBAplus, Visibroker, et COOLv2 respectivement) permettent la distribution dynamique de charge. Dans VisiBroker, elle peut être implémentée ou bien par un algorithme d'assignation de requêtes du type cyclique (round robin) ou bien par la migration d'objet. Dans CORBAplus, quatre algorithmes d'assignation de requêtes sont offerts au programmeur qui peut choisir comment répartir la charge parmi un ensemble de serveurs. Ces algorithmes sont *byQueueSize*, *byProcessLoad*, *byExplicitsetting* et *byRoundRobin*. Dans COOLv2, la migration d'objet est une alternative à l'invocation distante. Elle permet de migrer un objet serveur dans l'espace d'adressage d'un objet client pour procéder ensuite par appel local de méthodes. Cependant l'unité de migration n'est pas directement l'objet mais la grappe. Nous présentons une brève description de ces systèmes à la section 5.2.

Peu de travaux de recherche ont récemment examiné l'intégration de la distribution de charge dans les environnements CORBA. Nous décrivons brièvement deux de ces travaux : le projet LOCA [Rackl97, Schnekenburger97] et le projet LYDIA [Schiemann96a, Schiemann96b].

Projet LOCA

Le projet LOCA est un projet de recherche à l'Université de Munich qui a pour objectif d'évaluer différentes stratégies de distribution de charge pour les applications CORBA [Rackl97, Schnekenburger97]. Deux mécanismes pour réaliser la distribution de charge ont été étudiés par ce projet : l'assignation de requêtes et la migration d'objets. La figure 2.4 illustre ces deux mécanismes.

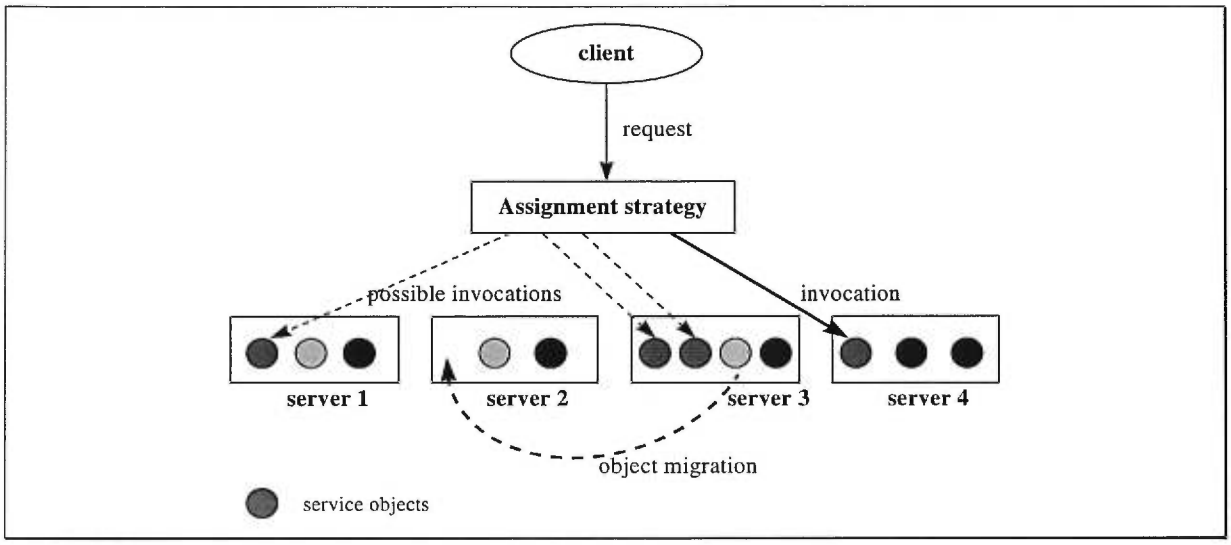


Figure 2.4 - Distribution de charge dans LOCA [Schnekenburger97]

L'assignation de requêtes est réalisée en utilisant le service de courtage pour implémenter les stratégies source-initiative et receveur-initiative. La migration proposée est basée sur un coordinateur central qui rassemble l'information de charge et qui est responsable de prendre les décisions de migration. Les résultats des expérimentations effectuées dans le cadre de ce projet ont montré que la stratégie symétrique est meilleure que les stratégies source-initiative et receveur-initiative plus particulièrement aux charges élevées du système, et que la performance de la stratégie de migration dépend de la structure et du comportement dynamique de l'application. À noter que dans les stratégies source-initiative et receveur-initiative utilisées dans ce projet, les auteurs identifient le client comme source et le serveur comme receveur, contrairement à la terminologie utilisée à la section 2.2.1.4 dans laquelle la source et le receveur sont deux noeuds qui exécutent les tâches des utilisateurs, et entre lesquels la distribution de charge est effectuée.

Projet LYDIA

Le projet LYDIA décrit comment la distribution de charge peut être intégrée dans les environnements IDL, comme CORBA [Schiemann96a, Schiemann96b]. La figure 2.5 illustre l'architecture de distribution de charge dans LYDIA.

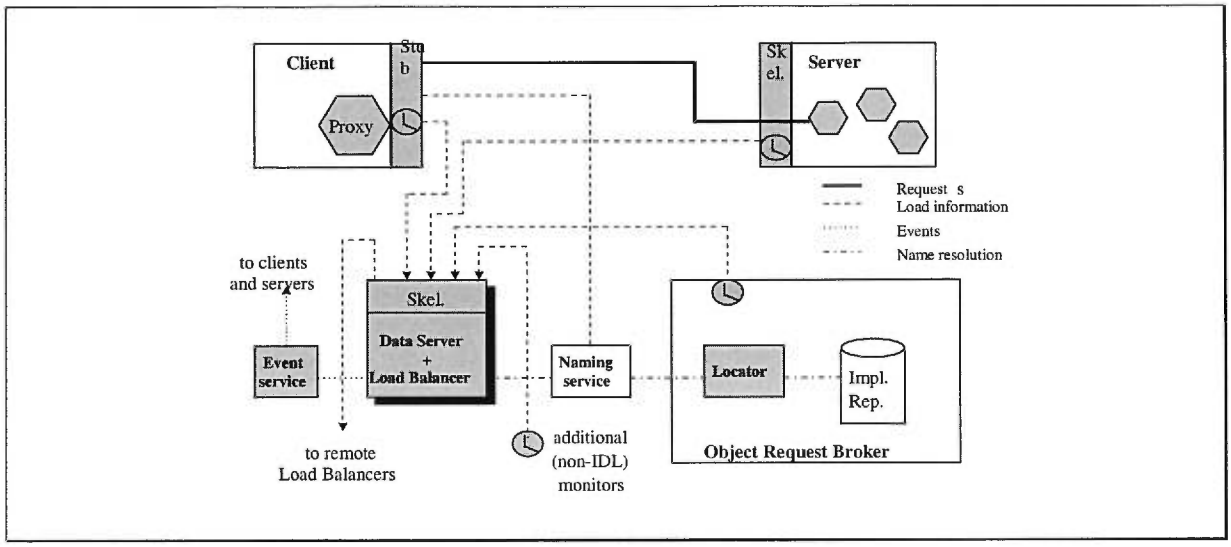


Figure 2.5 - Architecture d'un gestionnaire de charge dans LYDIA [Schiemann96b]

L'approche considérée dans LYDIA consiste à apporter des modifications au code source des talons client et serveur (stubs et skeletons) générés par le compilateur IDL de sorte que les programmes client et serveur soient instrumentés pour mesurer leur charge dynamique. L'information de charge est envoyée à un coordinateur central (load balancer) qui coopère avec le service de nommage pour le choix d'une référence objet appropriée parmi un ensemble d'alternatives.

2.3.2.8 DCOM

La distribution statique de charge dans DCOM consiste à assigner de manière permanente des utilisateurs à certains serveurs exécutant la même application. Cette technique est efficace avec la demande croissante des utilisateurs. Cependant, elle exige l'intervention d'un administrateur, et elle donne des résultats satisfaisants seulement dans le cas où la charge serait prévisible. DCOM n'adresse pas la distribution dynamique de charge. Cependant, il recommande d'utiliser un composant de référence, qui réside sur une machine donnée pour offrir ce service. Un client se connecte d'abord à ce composant, demandant une référence au service dont il a besoin (c'est semblable au service de courtage défini par l'OMG). Le composant de référence peut utiliser l'information de charge, la topologie du réseau entre client et serveur, et les statistiques concernant les demandes antérieures d'un utilisateur donné.

Synthèse

Le tableau 2.1 résume les approches de distribution de charge utilisées par les systèmes distribués à base d'objets que nous avons décrits dans cette section, ainsi que certaines caractéristiques des modèles objet associés.

Système	Approches de distribution de charge							Type d'objet	Groupage d'objet	Granularité	Langages Supportés
	Centralisée vs. distribuée	Statique vs. dynam.	Partage vs. équilib.	SI vs. RI	PO	MO	AR				
Emerald	distribuée	Dynam.	N/A	N/A	Non	Oui	Non	actif et passif	Non	Fine et grande	Emerald
Amadeus	Distribuée	Dynam.	Équilib.	N/A	Oui	Oui	Non	Passif	Oui (dynam.)	Fine	C**
Guide-2	Mixte	Dynam.	Partage	N/A	Oui	Oui	Non	Passif	Oui	Fine et grande	Guide C++
Shadows	N/A	N/A	N/A	N/A	Oui	Oui	Non	Passif	Non	Grande	C++
Legion	N/A	N/A	N/A	N/A	Oui	Non	Non	N/A	Non	Grande	C++
Isatis	Distribuée	Dynam.	Équilib.	N/A	Non	Oui	Oui	Passif	Non	Fine et grande	C++
ORBs CORBA											
VisiBroker	centralisée	statique	Partage	Non	Non	Oui	Oui	Passif	Non	Moyenne et grande	C++, Java
CorbaPlus	N/A	Statique et dynam.	Partage	Non	Non	Non	Oui	Passif	Oui	Moyenne et grande	C++, Java
COOLv2	N/A	N/A	Partage	Non	Non	Oui	N/A	Passif	Oui	Moyenne et grande	C++
Projets de recherche dans CORBA											
LOCA	centralisée	Dynam.	Partage	SI et RI	Non	Oui	Oui	Passif	Non	Moyenne et grande	C++
LYDIA	centralisée	Dynam	Équilib.	Non	Non	Non	Oui	Passif	Non	Moyenne et grande	N/A
Autres systèmes											
DCOM	N/A	N/A	N/A	N/A	N/A	non	non	Passif	Non	Moyenne et grande	Divers

Tableau 2.1 – Synthèse des méthodes de distribution de charge dans certains systèmes distribués objet

SI : source-initiative.
RI : receveur-initiative.
PO : placement d'objets.
MO : migration d'objets.
AR : assignation de requêtes.

2.4. Conclusion

Dans ce chapitre, nous avons présenté la problématique de distribution de charge dans les systèmes distribués. La première partie a exposé cette problématique dans le cadre des systèmes distribués classiques en présentant : quatre classifications des approches utilisées, les politiques qui composent un système de distribution de charge, un bref aperçu des travaux de recherche sur ce sujet, et des exemples de systèmes opérationnels de distribution de charge. La seconde partie a décrit cette problématique dans le cadre des systèmes distribués à base d'objets en présentant une classification des approches employées et des exemples de systèmes qui les utilisent. Ces approches sont le placement d'objets, la migration d'objets, et le placement d'invocations de méthodes.

L'utilisation limitée de la migration et du placement d'objets est due à beaucoup de facteurs dont : (1) l'hétérogénéité des modèles objet et des modèles d'exécution dans les systèmes distribués à base d'objets (objet passif contre objet actif, objet à granularité fine contre objet à granularité grosse, niveau de concurrence objet, etc.) [Bakker97], (2) l'hétérogénéité des langages de programmation supportés (C +, Smalltalk, Java, Emerald, etc.), (3) la nature dynamique des applications distribuées (les objets peuvent être créés ou supprimés en cours d'exécution) qui fait que la configuration des serveurs entre lesquels la charge doit être distribuée est dynamique, et (4) le support de certaines facilités spécifiques (groupage des objets, persistance, etc.) [Gourhant92].

Nous pensons que pour ces raisons le placement de requêtes est une solution acceptable et moins onéreuse que les deux premières approches, et qui peut être facilement mise en oeuvre dans des environnements distribués objets hétérogènes. Le reste de ce travail est consacré à l'étude du placement d'invocations de méthodes comme moyen efficace pour réaliser la distribution de charge dans ces environnements. Nous nous appuyons dans cette étude sur les travaux antérieurs réalisés au niveau processus et que nous essayons d'adapter au niveau objet. Les travaux les plus proches de notre approche sont les travaux LYDIA [Schiemann96a, Schiemann96b] et LOCA [Schnekenburger97, Rackl97] visant à intégrer la distribution de charge dans les environnements CORBA.

Chapitre 3

Assignation de requêtes dans un environnement distribué objet

Ce chapitre présente la problématique de l'assignation d'invocations de méthodes dans les systèmes distribués à base d'objets, propose une classification des approches d'assignation, et présente une modélisation analytique de l'approche centralisée à l'aide d'un système de files d'attente.

3.1. Introduction

Beaucoup de services distribués d'aujourd'hui, comme les services liés au commerce électronique et les services multimédias, exigent un haut degré de disponibilité. La panne d'un serveur peut coûter très cher aux entreprises. Ainsi, les serveurs sont souvent répliqués pour permettre un certain degré de tolérance aux fautes et afin d'offrir de meilleures performances en distribuant la charge parmi les serveurs. Avec l'avènement de systèmes tels que CORBA et DCOM, il est devenu possible de déployer des serveurs qui offrent un même service et des clients de ce service dans des environnements hétérogènes. Ces serveurs et ces clients peuvent être également écrits dans des langages différents. L'intégration des technologies en matière de systèmes distribués objet avec les technologies de l'Internet permet aux utilisateurs d'accéder facilement aux serveurs à travers un navigateur. Pour faire face à une demande croissante de services, il est nécessaire de mettre en œuvre dans ces systèmes des mécanismes de distribution de charge.

Dans ce chapitre, nous nous concentrons sur la distribution de charge au niveau opération car, comme nous l'avons discuté au chapitre précédent, la migration et le placement d'objets sont souvent difficiles à réaliser. Les aspects, présentés à la section 2.3.1.2.2, doivent être résolus pour pouvoir implémenter la migration d'objets avec succès.

Nous supposons un modèle objet avec des objets passifs interagissant par invocations de méthodes. Nous montrons qu'il est possible d'adapter les travaux antérieurs sur la distribution de charge au contexte des systèmes distribués à base d'objets en présentant trois approches permettant de réaliser l'assignation de requêtes :

1. l'approche orientée client basée sur les choix volontaires des clients,
2. l'approche orientée répartiteur basée sur l'utilisation d'un coordinateur central (ou répartiteur),
3. l'approche orientée serveur basée sur la coopération entre les serveurs.

Cette troisième approche présente l'avantage d'être distribuée car les serveurs implémentent les politiques de distribution de charge (information, localisation, sélection, et transfert) et sont capables de prendre eux même les décisions de transfert de requêtes vers d'autres serveurs contrairement aux deux premières approches. Dans le cadre de cette approche, nous discutons comment les stratégies du type source-initiative et receveur-initiative peuvent être adaptées au contexte des systèmes distribués à base d'objets.

La seconde approche se prête bien à la modélisation analytique. En effet, nous modélisons, à l'aide d'un système de files d'attente avec de multiples classes de service, un système composé d'une grappe de serveurs qui offrent un même type de service, d'un répartiteur, et d'un ensemble de clients de ce service. Les invocations d'une même méthode de l'interface d'un serveur correspondent à une même classe de service. Ce modèle analytique permet d'évaluer les performances moyennes de ce système en termes de temps de réponse et d'utilisation des serveurs.

3.2. Le problème d'assignation de requêtes

Cette section donne une formulation théorique du problème de l'assignation d'invocations de méthodes à des objets appropriés. Nous considérons un système distribué dans lequel les objets interagissent au moyen d'invocations de méthodes et dans lequel chaque objet implémente une interface spécifique comprenant un ensemble de méthodes. La distribution de charge au niveau opération consiste à assigner les requêtes (invocations de méthode) aux objets appropriés qui implémentent les méthodes appelées. Parmi les objectifs de cette

assignation on peut citer : la minimisation du temps de réponse des objets clients, l'augmentation de la disponibilité des serveurs, l'augmentation du débit (throughput) des serveurs, et la réduction de la congestion dans certains serveurs.

3.2.1. Définitions

On considère I l'interface implémentée par un objet O d'une application distribuée, et k une méthode de cette interface.

Nous définissons $INV_k = \{m_{k1}, m_{k2}, ..., m_{kn}\}$ comme étant l'ensemble des méthodes externes (implémentées par d'autres objets) que la méthode k invoque lors de son exécution. Cet ensemble est vide si aucune méthode externe n'est invoquée par k. Nous assumons que l'ensemble INV_k est connu à l'avance avant l'exécution de la méthode k. Nous assumons également que les méthodes dans INV_k sont invoquées d'une façon séquentielle comme dans l'exemple de la figure 3.1. Les objets mettant en oeuvre ces méthodes peuvent être locaux ou distants.

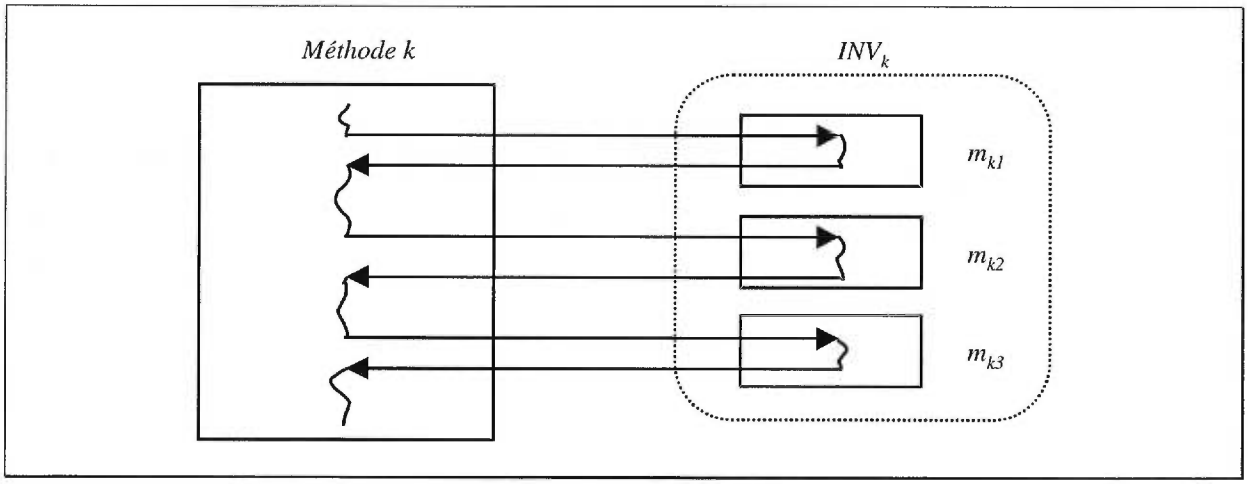


Figure 3.1 - Appel séquentiel de méthodes

Nous définissons Ω_I la collection d'objets répliqués qui implémentent l'interface I. Soient m une méthode dans INV_k (c'est-à-dire, k invoque m) et I_m l'interface d'un objet qui implémente la méthode m. L'objet O est alors un client des objets appartenant à Ω_{I_m}, et chaque objet dans Ω_{I_m} est un serveur de O.

Les objets d'une même application qui appartiennent à Ω_l forment une *grappe* (cluster) logique de serveurs objet. L'application peut être vue comme une collection de grappes dans lesquelles leurs membres interagissent mutuellement par invocation de méthodes[7]. Un objet peut se comporter comme client pour certaines opérations et serveur pour d'autres. Les objets d'une même grappe logique peuvent résider sur des machines distinctes du réseau et peuvent éventuellement être écrits dans des langages différents. Par conséquent, l'assignation d'invocations de méthodes à des serveurs appropriés est extrêmement importante pour atteindre de meilleures performances.

3.2.2. Formulation du problème d'assignation

Le problème d'assignation consiste à assigner la méthode appelée *m* à un objet approprié dans Ω_{l_m}. Ce problème est similaire à celui de l'assignation d'une tâche à une collection de processeurs[8]. Cependant, le problème d'assignation de requêtes devient extrêmement compliqué quand la méthode appelée invoque d'autres méthodes d'autres objets (locaux ou distants), et ces méthodes pourraient à leur tour appeler d'autres méthodes d'autres objets, et ainsi de suite. Le graphe de la figure 3.2 illustre cette situation. C'est un graphe acyclique orienté (DAG ou Direct Acyclic Graph) dont les nœuds sont les méthodes et les liaisons sont les relations d'invocation entre les méthodes. La méthode initiale *m*, appelée par l'objet client, invoque deux méthodes *n* et *o* dans cet ordre. La méthode *n* appelle trois méthodes *p*, *q* et *r* respectivement, et ainsi de suite. Nous dénotons ce graphe par G_m. L'indice dans G_m fait référence au nœud source du graphe. Chaque méthode dans G_m peut-être implémentée par une collection d'objets. Nous supposons que ces objets sont connus à l'avance ou peuvent être découverts dynamiquement au moyen d'un service de localisation (comme le service de nommage et le service de courtage décrit à l'annexe 2).

[7] Dans ce chapitre, nous considérons seulement les systèmes distribués à base d'objets de ce type. Dans d'autres modèles objets, les objets peuvent communiquer à travers une mémoire partagée.

[8] L'assignation optimale d'un ensemble de tâches à un ensemble de processeurs est un problème NP-complet, sauf dans peu de cas spéciaux [Ullman75].

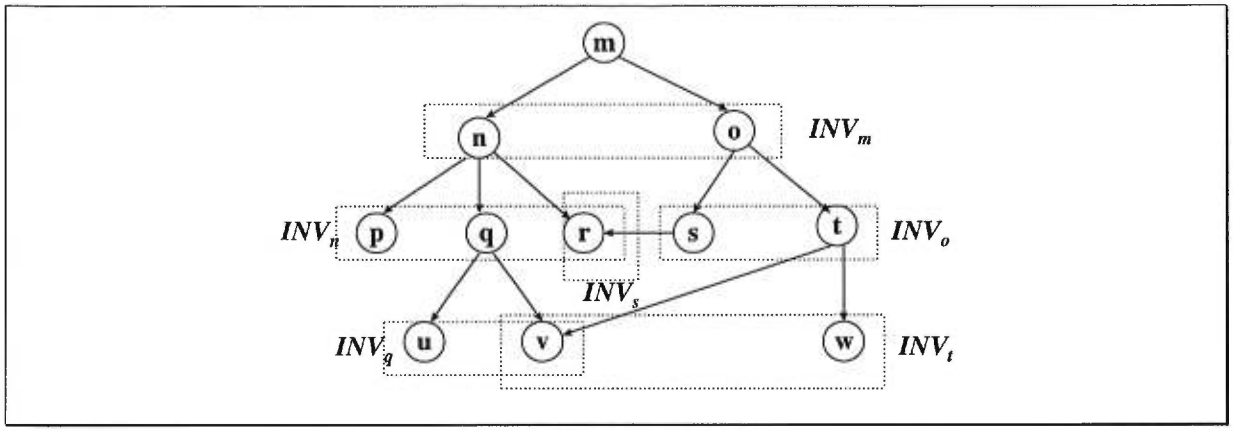


Figure 3.2 - Graphe G_m acyclique orienté d'appels de méthodes

Une assignation efficace de la méthode *m* exige l'assignation de toutes les méthodes de G_m à des objets appropriés. Nous suggérons à cet effet un algorithme récursif. L'assignation de chaque méthode dans INV_m conduira à l'assignation du sous-graphe dont le nœud source est cette méthode. La figure 3.3 décrit cet algorithme.

```
Procedure RequestAssignment(Gm:DAGType)
Begin
        If (Gm <> NIL) then
        Begin
              Assign source-node of Gm
              For each method j in INVsource-node
                   RequestAssignment(Gj)
              End For
        End
End
```

Figure 3.3 - Algorithme d'assignation récursif

Dans l'algorithme *RequestAssignment*, l'assignation du noeud source de G_m à un objet approprié se fait dans le but d'atteindre certains objectifs fixés au préalable. Un des objectifs du problème d'assignation est de minimiser le temps d'achèvement (completion time) de la méthode invoquée *m*. Du point de vue de l'objet client qui appelle *m*, le temps d'achèvement de *m* (CT_m) comprend les composantes suivantes :

- $T_{trans(m)}$: Le temps pour transférer la requête et ses paramètres au serveur.
- $T_{proc(m)}$: Le temps de traitement de la requête.

- $T_{res(m)}$: Le temps pour transférer les résultats à l'objet client.

$$CT_m = T_{trans(m)} + T_{proc(m)} + T_{res(m)} \quad (1)$$

$T_{trans(m)}$ et $T_{res(m)}$ dépendent de la topologie du réseau entre l'objet client et l'objet serveur à qui la méthode appelée m a été assignée, de la taille des paramètres, et du trafic réseau. $T_{proc(m)}$ peut être divisé en deux termes (équation 2). Le premier terme $T_{eff(m)}$ est le temps de traitement effectif de la méthode m par l'objet cible. Le deuxième terme correspond à la somme des temps d'achèvement de toutes les méthodes dans INV_m. $T_{eff(m)}$ dépend principalement de la vitesse du serveur cible (la vitesse CPU, la charge du serveur, la charge de la machine attribuable à d'autres processus, etc.).

$$T_{proc(m)} = T_{eff(m)} + \sum_{j \in INV_m} CT_j \quad (2)$$

Les équations (1) et (2) donnent :

$$T_{proc(m)} = T_{eff(m)} + \sum_{j \in INV_m} (T_{trans(j)} + T_{proc(j)} + T_{res(j)}) \quad (3)$$

Les équations (1) et (4) donnent :

$$CT_m = T_{trans(m)} + T_{eff(m)} + T_{res(m)} + \sum_{j \in INV_m} (T_{trans(j)} + T_{proc(j)} + T_{res(j)}) \quad (4)$$

Les équations (3) et (4) sont applicables d'une façon récursive à chaque méthode invoquée par m. L'assignation optimale correspond à l'assignation qui minimise l'équation (4). Puisque l'assignation de la méthode initiale produit des sous-problèmes correspondant à l'assignation de toutes les méthodes dans G_m, l'assignation optimale exige qu'à chaque exécution de l'algorithme *RequestAssignment* l'assignation soit optimale. Une solution théorique générale à ce problème est donc difficile à trouver vu le grand nombre de paramètres du problème. Cependant, il serait possible de trouver des solutions dans des cas spéciaux comme dans le problème d'assignation de tâches à un ensemble de processeurs. Le cas le plus simple de ce problème est celui dans lequel la méthode appelée n'invoque pas d'autres méthodes externes. Dans ce cas, INV_m est l'ensemble vide. L'équation (3) et l'équation (4) deviennent :

$$T_{proc(m)} = T_{eff(m)} \quad (5)$$

$$CT_m = T_{trans(m)} + T_{eff(m)} + T_{res(m)} \quad (6)$$

L'assignation optimale de m correspond à l'assignation qui minimise l'équation (6). En termes de coûts, le coût d'assignation d'une requête dans ce cas simple comprend le coût de communication, transmission des paramètres et des résultats, et le coût de traitement par le serveur choisi. En pratique, il est difficile de réaliser une assignation optimale car les coûts de communication et de traitement ne peuvent être estimés avec précision. L'évaluation de ces coûts peut même rendre la tâche plus difficile. Des méthodes d'assignation sous-optimales peuvent être suffisantes dans la majorité des cas.

3.3. Approches pratiques d'assignation de requêtes

Dans cette section, nous décrivons des approches pratiques pour l'assignation de requêtes à des serveurs objet dans un objectif de distribution de charge.

Le choix du meilleur serveur pour traiter une invocation de méthode dépend de plusieurs facteurs tels que la charge du serveur, la vitesse et la charge de la machine où réside le serveur, et la charge du réseau entre le client et le serveur. Ce choix peut être fait aussi bien par le client que par un serveur. Il peut être également fait par une composante externe au client et au serveur. Par conséquent, nous distinguons trois types d'approches pour l'assignation de requêtes : approche orientée client, approche orientée répartiteur, et approche orientée serveur.

3.3.1. Approche orientée client

Ce type de distribution de charge est basé sur les choix volontaires des objets client. Un objet client peut utiliser pour le choix du serveur cible différentes stratégies qui peuvent être statiques ou dynamiques. Les stratégies dynamiques utilisent l'information instantanée sur l'état du système lors de la prise de décision d'assignation. La figure 3.4 illustre la stratégie orientée client.

Les stratégies *aléatoire* et *cyclique* sont deux exemples de stratégies statiques qui sont facilement mises en oeuvre et qui peuvent donner des résultats acceptables en matière de performances. Elles n'exigent aucune connaissance sur l'état actuel des serveurs objet cibles. Avec la stratégie aléatoire tous les serveurs ont la même probabilité d'être choisis. Un de ces serveurs est choisi de manière aléatoire. La stratégie aléatoire est employée par beaucoup de

sites Internet pour permettre aux utilisateurs de télécharger des fichiers à partir d'une liste de serveurs ftp. Ainsi, la charge est distribuée parmi ces serveurs ftp en se basant sur les choix faits par les utilisateurs. Avec la stratégie cyclique, tous les services sont sélectionnés de manière cyclique. La stratégie de sélection du serveur *le moins chargé* est une stratégie dynamique qui exige l'échange de l'information d'état des serveurs périodiquement ou sur demande.

Dans cette approche orientée client, la surcharge des serveurs peut être évitée par la mise en place de quotas. Par exemple, aucun utilisateur ne peut envoyer plus de N requêtes pendant une période de temps T. Les serveurs peuvent aussi appliquer des seuils sur le nombre de requêtes que chaque serveur peut traiter pendant une période de temps T. Quand le seuil est atteint avant la fin de la période T, les requêtes suivantes sont rejetées jusqu'à ce que la charge du serveur tombe à une valeur inférieure à son seuil.

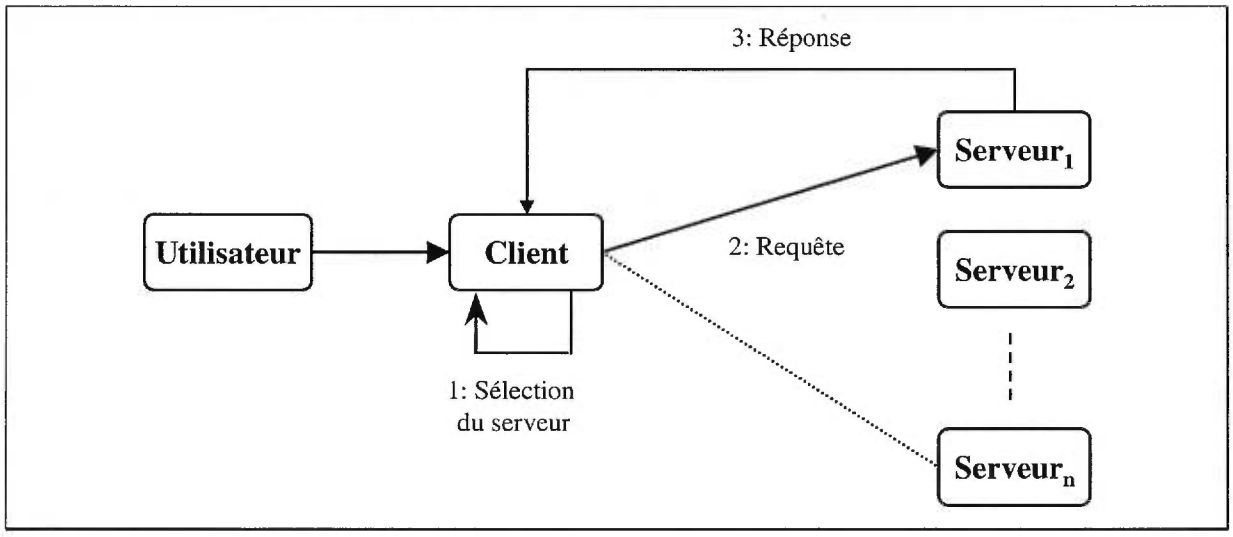


Figure 3.4 - Approche d'assignation orientée client

L'assignation orientée client peut aussi être effectuée par une tierce partie pouvant prendre les décisions d'assignation pour le compte du client. Certains systèmes distribués objet ont introduit le concept de *proxy* qui joue le rôle de représentation locale d'un serveur (pouvant être répliqué) au niveau du client. Un proxy intelligent, comme celui introduit dans OrbixWeb, peut être configuré par le client pour offrir des fonctionnalités telles que l'assignation de ses requêtes à un serveur moins chargé et la sauvegarde (caching) de

certaines informations du serveur localement au site du client. La figure 3.5 illustre l'assignation de requêtes par un proxy.

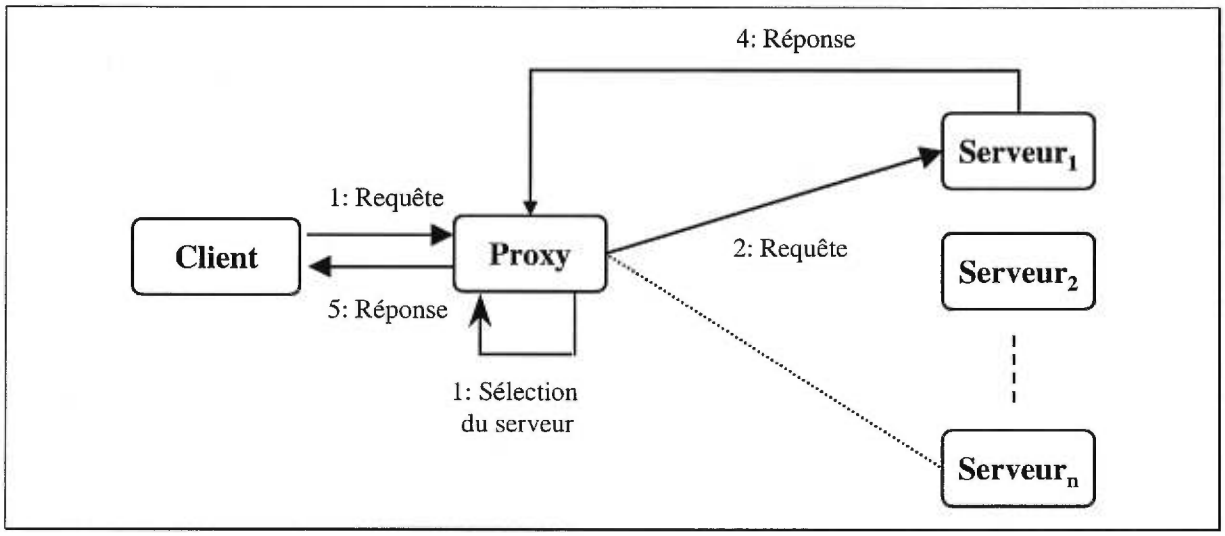


Figure 3.5 - Assignation de requêtes par un proxy

Le concept de *client intelligent* (smart client) a été introduit ces dernières années [Yoshikawa97]. L'idée derrière ce concept est la migration de certaines fonctionnalités du serveur à la machine cliente pour permettre de : (1) décharger le serveur et réduire la complexité de l'implémentation, (2) permettre aux clients d'utiliser de manière transparente de multiples serveurs sans connaissance des serveurs individuels, et (3) améliorer la distribution de charge et la transparence aux fautes.

3.3.2. Approche orientée répartiteur

La localisation des serveurs est souvent déterminée par des services standards tels que le service de nommage et le service de courtage [OMG97a, ISO96b]. Par utilisation d'un service de localisation, les objets clients sont capables de découvrir les objets serveurs pouvant traiter leurs requêtes. Cependant, ce service n'est normalement pas suffisant pour faire un bon choix car il ne dispose pas d'informations sur l'état actuel des serveurs. Dans plusieurs implémentations, cependant, un composant central appelé *répartiteur* (dispatcher) joue le rôle d'interface entre les clients et les serveurs. Avec ce modèle, les clients n'ont pas besoin de connaître les serveurs pour être en mesure de recevoir leurs services désirés. Les requêtes des clients sont acheminées à des serveurs appropriés par le répartiteur qui implémente certaines

stratégies d'ordonnancement et de distribution de charge et qui collecte l'information d'état des serveurs. La figure 3.6 illustre le principe de l'approche orienté répartiteur.

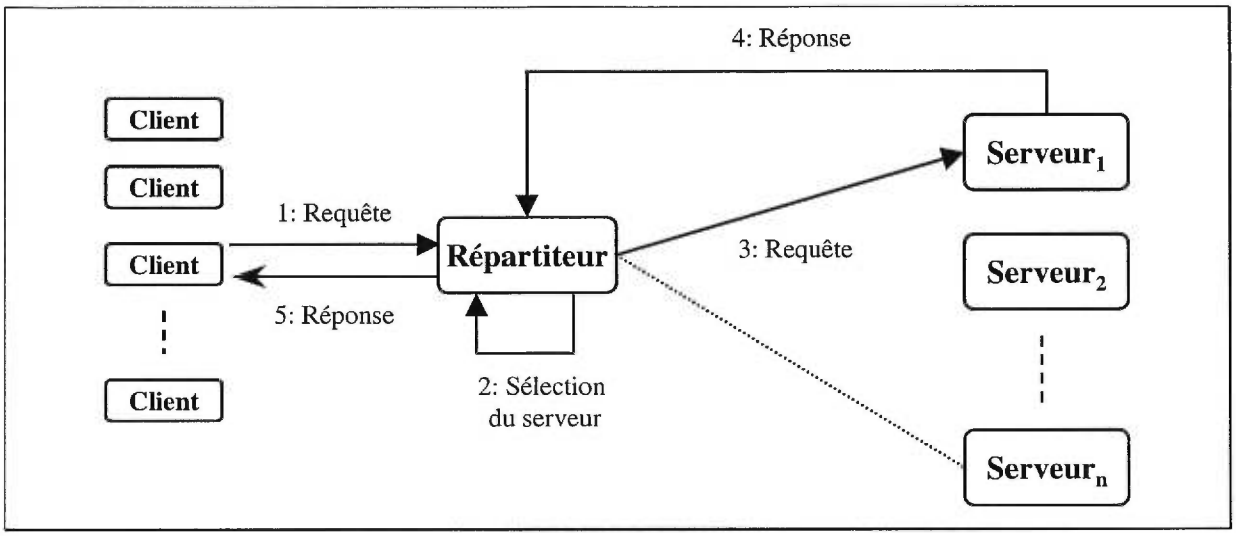


Figure 3.6 - Stratégie d'assignation par un répartiteur

Le répartiteur peut implémenter diverses stratégies d'ordonnancement pour effectuer l'assignation de requêtes des clients aux différents serveurs d'une grappe. Si l'opération invoquée n'invoque pas à son tour d'autres méthodes sur des objets (internes ou externes à la grappe), alors le résultat est immédiatement retourné au client. Sinon, le serveur devient client et la méthode invoquée est envoyée au répartiteur afin d'être traité par un serveur convenable d'une grappe appropriée. Nous illustrons ceci par une rétroaction dans la figure 3.7. Cette rétroaction schématise la récursivité dans l'algorithme *RequestAssignment* que nous avons présenté précédemment à la section 3.2.2.

Le modèle de la figure 3.6 avec une seule grappe a été utilisé dans les systèmes distribués classiques pour modéliser la distribution de charge et l'ordonnancement des tâches. Le système est organisé comme un ensemble de N files d'attente en parallèle qui représentent les ressources du système et un répartiteur central qui distribue les tâches entrantes entre les files d'attente [Bonomi90, Mni85, Eager86]. L'outil *lsbatch* de LSF [Platform94], par exemple, que nous avons présenté au chapitre 2, utilise ce modèle pour répartir les tâches entre les machines d'une grappe LSF. Toutes les tâches soumises à une même file d'attente partagent une même stratégie d'ordonnancement et de contrôle.

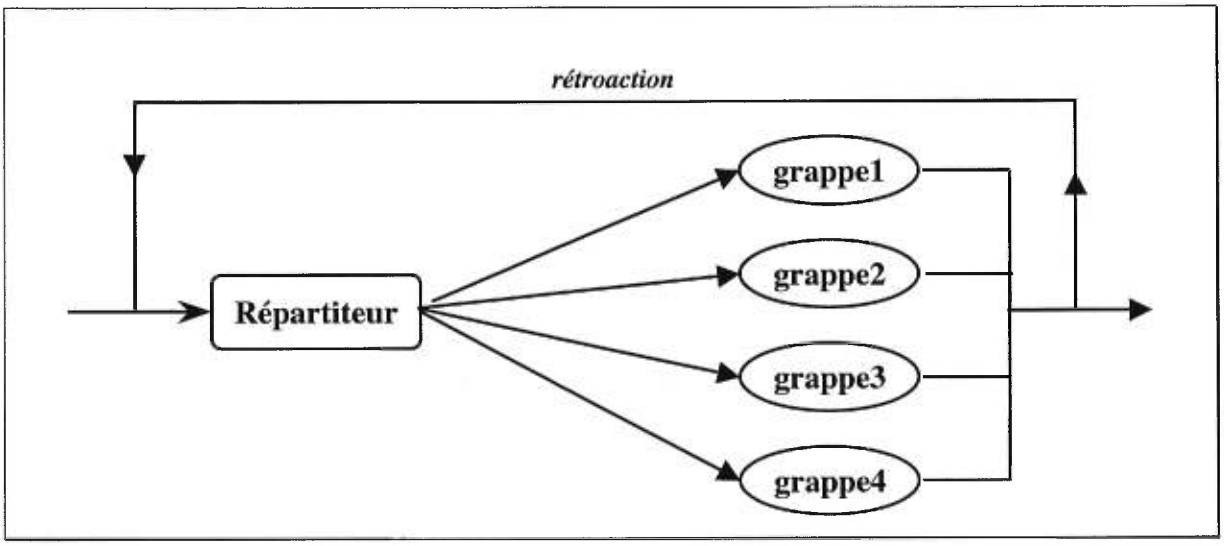


Figure 3.7 - Modèle d'un système avec de multiples grappes et un répartiteur

Ce modèle est utilisé de nos jours par les sites Internet qui sont très sollicités. Les requêtes qui arrivent à un site qui utilise ce modèle sont distribuées par un répartiteur parmi les serveurs Web répliqués du site. Plusieurs produits commerciaux tels que *F5 BIG/IP* [F5Networks], *IBM Network Dispatcher* [IBM], et *CISCO Local Director* [Cisco] assurent la fonction de répartiteur. Les stratégies utilisées par ces produits sont le plus souvent du type aléatoire et cyclique.

L'approche d'assignation avec un répartiteur est une approche centralisée qui peut être statique ou dynamique. Dans le cas d'une approche dynamique, les politiques présentées à la section 2.2.2, sont applicables. En termes de *la politique d'information*, le répartiteur reçoit l'information de charge de tous les serveurs périodiquement ou sur demande et l'assemble dans un vecteur de charge. En termes de *la politique de localisation*, quand le répartiteur reçoit une requête d'un objet client, il choisit un serveur cible en utilisant le vecteur de charge. En termes de *la politique de sélection*, une requête entrante au répartiteur est immédiatement considérée pour le traitement par un serveur approprié sélectionné par la politique de localisation. Finalement, en termes de *la politique de transfert*, une politique avec seuil de charge peut être appliquée pour décider de l'état d'un serveur (surchargé, moyennement chargé, légèrement chargé).

3.3.3. Approche orientée serveur

Cette approche suppose que les serveurs d'une même grappe logique se connaissent mutuellement et peuvent coopérer pour mettre en oeuvre la distribution de charge par échange de l'information de charge et de requêtes. Les stratégies source-initiative et receveur-initiative présentées à la section 2.2.1.4 peuvent être appliquées avec cette approche. La figure 3.8 schématise le principe de cette approche.

À notre connaissance, aucun travail n'a essayé d'adapter ces deux méthodes au contexte d'un environnement distribué à base d'objets. Comme exception, on pourrait évoquer le travail de Rackl et Schnekenburger [Rackl97, Schnekenburger97] pour l'implémentation de la distribution de charge dans un environnement CORBA. Cependant, dans ce travail, les auteurs comparent le client à un nœud source et le serveur à un nœud receveur. À notre avis, source et receveur doivent être deux entités de même nature et jouer le même rôle qui est celui de l'exécution des requêtes. Dans notre travail, nous comparons les serveurs aux nœuds d'un système distribué classique et les requêtes des clients aux tâches soumises par les utilisateurs.

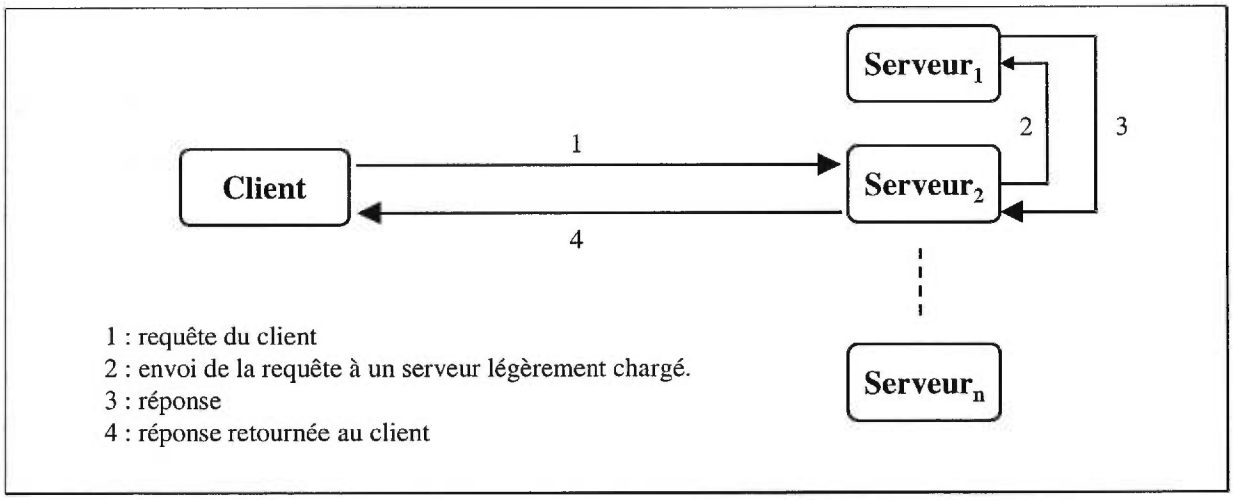


Figure 3.8 - Approche d'assignation orientée serveur

Pour appliquer les stratégies source-initiative et receveur-initiative dans le contexte de systèmes distribués à base d'objets, il doit y avoir coopération entre les serveurs d'une grappe logique qui implémentent une même interface par échange de l'information de charge et de requêtes. Dans la suite de cette section, nous décrivons comment ces deux stratégies peuvent être utilisées au niveau opération pour réaliser l'assignation de requêtes et donc la distribution

de charge. Nous nous basons dans la description de ces deux stratégies sur les politiques présentées à la section 2.2.2.

Avec la stratégie source-initiative, un serveur surchargé (source) initie l'activité de distribution de charge en essayant d'exécuter une requête *m* (invocation de méthode) dans un serveur légèrement chargé (receveur) appartenant à Ω_{Im}. Avec la stratégie receveur-initiative, un serveur légèrement chargé (receveur) et implémentant une interface *I* initie l'activité de distribution de charge en essayant d'obtenir une requête d'un serveur surchargé (source) appartenant à Ω_I. Les figures 3.9 et 3.10 schématisent deux exemples d'algorithmes qui réalisent respectivement les stratégies source-initiative et receveur-initiative.

3.3.3.1 Politique d'information

Cette politique est responsable de déterminer l'information d'état requise des autres serveurs ainsi que l'instant pendant lequel elle doit être collectée. Elle peut être aussi bien périodique que sur demande.

3.3.3.2 Politique de localisation

Différentes méthodes de localisation peuvent être considérées pour déterminer un serveur cible (receveur ou source selon la stratégie). Le serveur cible est choisi parmi les serveurs appartenant à Ω_{Im} (source-initiative) et à Ω_I (receveur-initiative). Nous supposons que chaque serveur connaît l'ensemble des serveurs appartenant à Ω_{Im} (respectivement Ω_I) qui implémentent la méthode *m* à transférer (respectivement l'interface *I*). Cet ensemble peut être connu à l'avance lors de l'activation du serveur ou découvert dynamiquement. Comme exemples de méthodes de localisation, citons les méthodes : *aléatoire*, *seuil, minimal,* et *le moins chargé* pour la stratégie source-initiative et les méthodes *aléatoire*, *seuil*, *maximal,* et *le plus chargé* pour la stratégie receveur-initiative.

Aléatoire -- Le serveur cible receveur (respectivement source) est choisi de manière aléatoire sans qu'il y ait échange d'information d'état entre les serveurs.

Seuil -- Un serveur receveur appartenant à Ω_{Im} (respectivement un serveur source appartenant à Ω_I) est choisi au hasard et interrogé pour déterminer si le transfert d'une requête

du serveur source provoque l'augmentation du taux d'utilisation du serveur receveur au-delà d'un seuil supérieur de charge T (respectivement la réduction de la charge du serveur source à une valeur inférieure à ce seuil T). Si ce n'est pas le cas, alors la requête est transférée au serveur receveur. Sinon, un autre serveur est choisi au hasard et est interrogé de la même façon. Le nombre maximal de serveurs à interroger est appelé la limite d'exploration L_p (polling limit). Si aucun serveur approprié n'est trouvé au bout de L_p interrogations, alors le serveur source doit exécuter la requête (respectivement le serveur receveur doit attendre pendant une certaine période de temps avant de recommencer la même procédure).

Minimal – Un sous-ensemble de Ω_{lm} est choisi au hasard, et le serveur ayant la charge minimale est choisi.

Le moins chargé – Cette méthode est similaire à la méthode *Seuil* sauf qu'elle n'est pas basée sur un seuil supérieur de charge. Le serveur source choisit au hasard un nombre fixe L_p de serveurs appartenant à Ω_{lm} et détermine le serveur le moins chargé parmi ces serveurs, soit à partir des informations de charge sur les autres serveurs dont il dispose, soit par interrogation successive des serveurs. Si le taux d'utilisation du serveur le moins chargé est supérieur à celui du serveur source alors la requête doit être traitée par le serveur source. Sinon, la requête est transférée au serveur le moins chargé.

Maximal – Un sous-ensemble de Ω_l est choisi au hasard, et le serveur ayant la charge maximale est choisi.

Le plus chargé -- Cette politique est similaire à la politique *Seuil* sauf qu'elle n'est pas basée sur un seuil supérieur de charge. Le serveur receveur choisit au hasard un nombre fixe L_p de serveurs appartenant à Ω_l, et détermine le serveur le plus chargé parmi ces serveurs soit à partir des informations sur les autres serveurs dont il dispose soit par interrogation successive des serveurs choisis. Si le taux d'utilisation du serveur le plus chargé est inférieur à celui du serveur receveur, alors les serveurs choisis sont légèrement chargés, et le serveur receveur attend pendant une certaine période de temps avant de recommencer la même procédure.

3.3.3.3 Politique de sélection

Seules les nouvelles requêtes reçues par le serveur source sont considérées pour le transfert. Le transfert d'une requête est réalisé en expédiant la requête du serveur source au serveur receveur choisi par la politique de localisation. Le résultat de la requête est retourné au serveur source et ensuite au client (voir figure 3.8).

3.3.3.4 Politique de transfert

La politique de transfert peut être une politique de seuil basée sur le taux d'utilisation du serveur (la charge). Un serveur est identifié comme *source* si une nouvelle requête de service fait passer le taux d'utilisation du serveur à une valeur supérieure à un seuil T. De la même façon un serveur s'identifie comme *receveur* approprié pour un transfert de requête si l'acceptation de la requête ne cause pas le dépassement du seuil T.

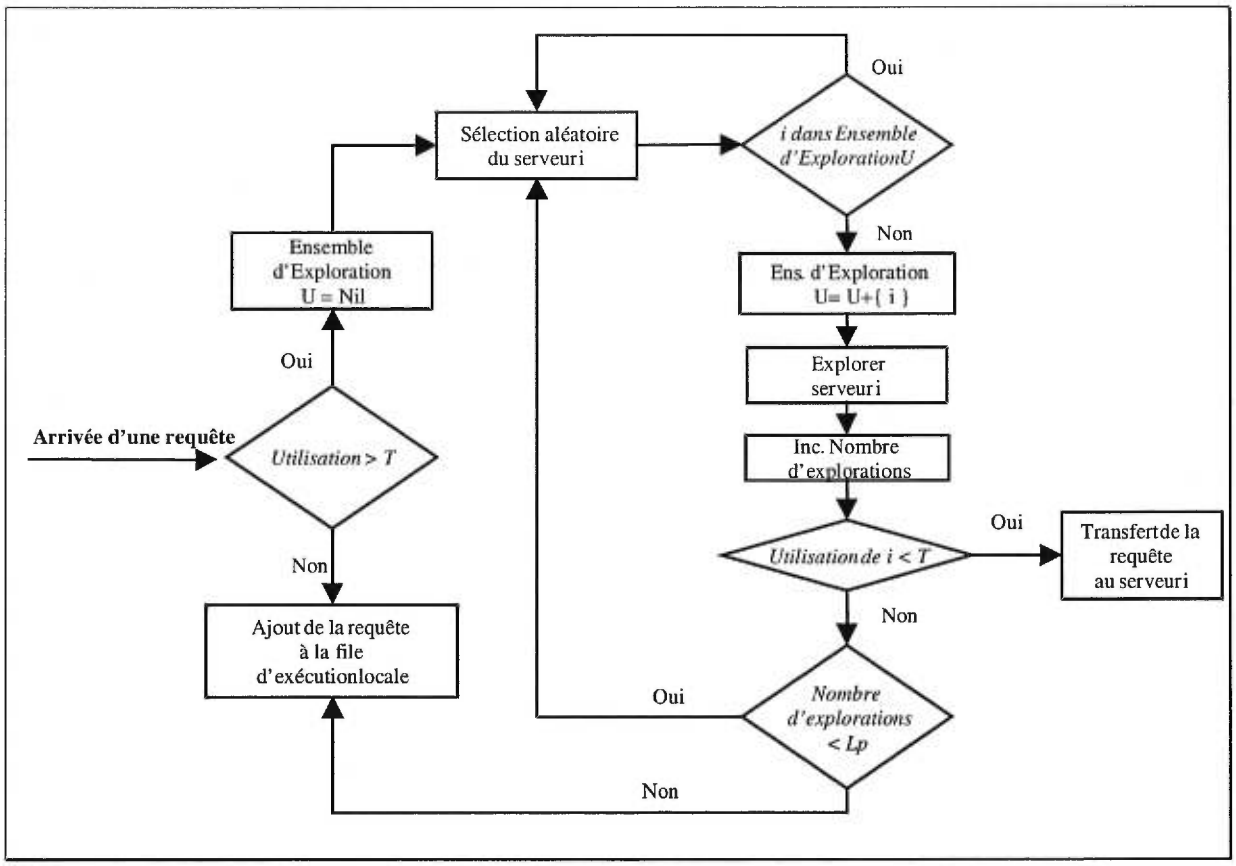


Figure 3.9 - Exemple d'un algorithme source-initiative

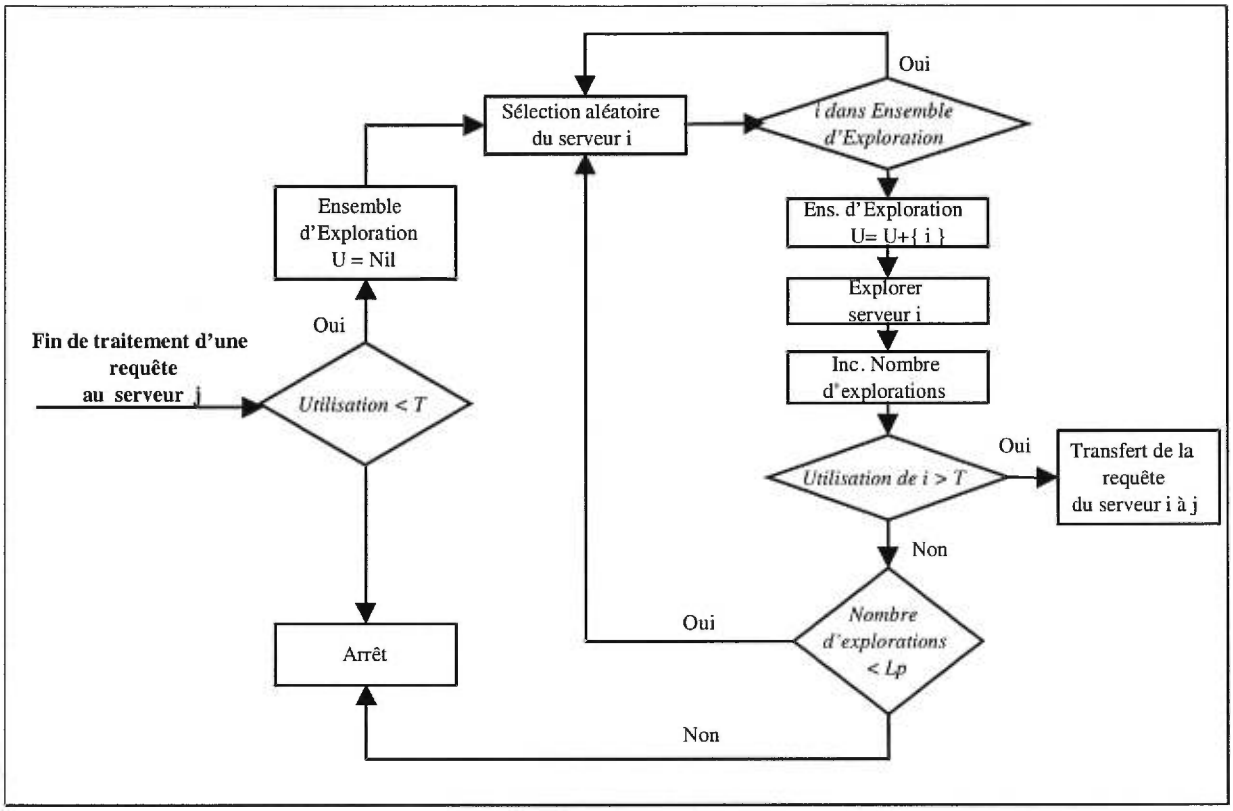


Figure 3.10 - Exemple d'un algorithme receveur-initiative

3.4. Modélisation analytique

Dans cette section, nous présentons un modèle analytique de l'approche orientée répartiteur présentée à la section 3.3.2. Le répartiteur dispose d'une file d'attente des requêtes qui sont en attente d'être acheminées vers l'un des serveurs du système. Les serveurs peuvent également disposer de leurs propres files d'attente. L'approche orientée client et l'approche orientée serveur ne se prêtent pas à la modélisation analytique avec des files d'attente car elles sont distribuées. Le but de cette modélisation est d'étudier le comportement moyen du système en évaluant ses performances moyennes. Le prototype de l'architecture LoDACE, décrite au chapitre 4, utilise l'approche orientée répartiteur. Les résultats des expérimentations faites avec ce prototype et décrites au chapitre 6 sont comparés aux résultats obtenus par cette modélisation analytique.

Dans un modèle de répartiteur avec de multiples grappes, chaque grappe est associée à une interface publiée par les objets de la grappe. En d'autres termes, les objets d'une même grappe implémentent la même interface. Nous modélisons chaque serveur comme une station

de service offrant de multiples classes de service. Les invocations d'une même méthode sont considérées comme des requêtes d'une même classe. Ainsi, chaque station de service a autant de classes de service que de méthodes publiques de l'interface du serveur associé. Le répartiteur assigne une nouvelle requête à une des stations de service en fonction de la classe de la requête. La figure 3.11 montre le modèle de file d'attente associé. Chaque serveur est modélisé comme un centre de service sans un flot dédié de requêtes. Nous supposons que toutes les nouvelles requêtes entrantes soient envoyées aux serveurs par l'intermédiaire du répartiteur.

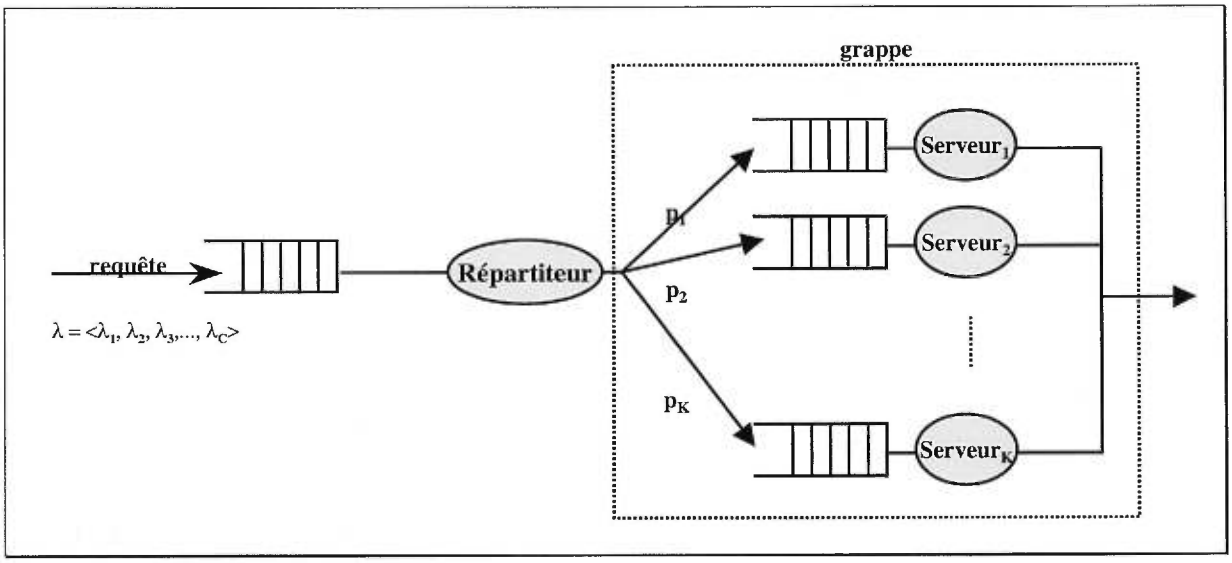


Figure 3.11 - Modèle de files d'attente avec de multiples classes de service

En outre, tous les serveurs du modèle appartiennent à la même grappe. Notons que le répartiteur peut manipuler des requêtes pour plusieurs interfaces où chaque interface est associée à une grappe logique de serveurs. Les serveurs d'une grappe peuvent appeler les méthodes d'autres grappes. Ceci est schématisé par la rétroaction dans la figure 3.7. Nous jugeons que la modélisation analytique d'un système avec de multiples grappes comme celui de la figure 3.7 est trop compliquée en raison de la rétroaction entre chacune des grappes et le répartiteur. Ainsi, nous nous limitons à la modélisation d'un système constitué d'une seule grappe, comme celui de la figure 3.11, à l'aide d'un modèle ouvert de files d'attente à multiples classes de service [Lazowska84].

3.4.1. Spécification du modèle analytique

I L'interface implémentée par les serveurs : $Serveur_1$, $Serveur_2$, ..., $Serveur_K$.

K Nombre de centres de service implémentant l'interface I. Nous supposons que ce nombre est fixe. En réalité ce nombre peut changer car les serveurs peuvent être créés ou supprimés dynamiquement.

C Nombre de méthodes offertes par l'interface I. Chaque méthode correspond à une classe de requêtes.

λ Taux d'arrivée des requêtes au système. Nous supposons que la distribution des arrivées suit la loi de Poisson avec la moyenne λ.

λ_j Taux d'arrivée des requêtes de la classe j. C'est aussi une distribution de Poisson.

$$\lambda = <\lambda_1, \lambda_2, \lambda_3, ..., \lambda_C> \qquad \text{et} \qquad \lambda = \sum_{j=1}^{C} \lambda_j$$

$D_{j,k}$ La demande de service (temps moyen de service requis par la requête pour être traitée) de la classe j au centre de service k.

D_k La demande de service de toutes les classes au centre de service k.

p_k La probabilité d'assignation de la requête au centre de service k. Nous supposons que cette probabilité est indépendante de la classe de service. La probabilité p_k dépend de la stratégie de sélection de serveur implémentée par le répartiteur. Le taux d'arrivée des requêtes au centre de service k est donné par :

$$p_k\lambda = < p_k \lambda_1, p_k \lambda_2, ..., p_k \lambda_C>$$

$P_k\lambda_j$: le taux d'arrivée des requêtes de classe j au centre de service k.

$$\sum_{i=1}^{K} p_i = 1$$

$R_{j,k}$ Le temps de réponse de la classe j au centre de service k.

R_j Le temps de réponse de la classe j à tous les centres de service.

$U_{j,k}$ Le taux d'utilisation de la classe j au centre de service k.

U_k Le taux d'utilisation de toutes les classes au centre de service k.

$Q_{j,k}$ La longueur de la file d'attente des requêtes de la classe j au centre de service k.

Q_j La longueur de la file d'attente des requêtes de la classe j à tous les centres de service.

Nous supposons que toutes les requêtes sont servies sur la base de premier arrivé premier servi (First Come First Served) et que chaque serveur est modélisé comme un centre de service *M/M/1* avec de multiples classes de service.

3.4.2. Résolution du modèle

La résolution de ce modèle donne les équations suivantes :

$$R_{j,k} = D_{j,k}/[1 - \sum_{i=1}^{C} U_{i,k}]$$

Le taux d'utilisation $U_{i,k}$ est donné par la formule de Little [Lazowska84]:

$$U_{i,k} = p_k \lambda_i D_{i,k}$$

Ceci conduit à : $$R_{j,k} = D_{j,k}/[1 - \sum_{i=1}^{C} p_k \lambda_i D_{i,k}]$$

Si nous considérons que toutes les classes de service ont le même taux d'arrivée, alors :

$$\lambda_j = \underset{1 \le j \le C}{\lambda/C}$$

$R_{j,k}$ devient : $$R_{j,k} = D_{j,k}/[1 - \frac{\lambda}{C} \sum_{i=1}^{C} p_k D_{i,k}]$$

$D_k = \sum_{i=1}^{C} D_{i,k}$ donc $R_{j,k} = D_{j,k}/[1 - \lambda p_k D_k/C]$

et $$R_j = \sum_{k=1}^{K} \frac{D_{j,k}}{1 - p_k \lambda D_k/C}$$

Le taux d'utilisation de la classe *j* au centre de service *k* est donné alors par :

$$U_{j,k} = p_k \lambda_j D_{j,k} = p_k \lambda D_{j,k}/C$$

et $$U_k = \sum_{j=1}^{C} \frac{p_k \lambda D_{j,k}}{C} = \frac{p_k \lambda}{C} \sum_{j=1}^{C} D_{j,k} = \frac{p_k \lambda}{C} D_k$$

La longueur de la file d'attente de la classe *j* au centre de service *k* est donnée par :

$$Q_{j,k} = p_k \lambda_j R_{j,k} = p_k \lambda R_{j,k}/C$$

et $$Q_j = \sum_{k=1}^{K} Q_{j,k} = \frac{\lambda}{C} \sum_{k=1}^{K} p_k R_{j,k}$$

Les paramètres R_j, U_k, et Q_j représentent les métriques de performance du système modélisé. Ils déterminent le comportement moyen du système. Nous analysons dans la section suivante

les valeurs prises par ces paramètres en fonction du taux d'arrivée des requêtes et de la distribution probabiliste { p_i }.

3.4.3. Interprétation des résultats

Dans la section précédente, nous avons paramétrisé les métriques de performance du système modélisé. Dans cette section, nous interprétons les valeurs prises par ses métriques en utilisant l'analyse asymptotique qui permet de déterminer les limites supérieures et inférieures du taux d'arrivée de requêtes dans le système et du temps de réponse moyen.

3.4.3.1 Taux d'arrivée

La limite supérieure du taux d'arrivée indique le taux maximal des arrivées des requêtes que le système peut traiter avec succès. Elle peut être calculée à partir des équations du taux d'utilisation :

$$U_{j,k}=p_k\lambda D_{j,k}/C \qquad \text{et} \qquad U_k=p_k D_k \lambda/C$$

Étant donné une distribution de probabilité $\{p_i\}$, un centre de service k peut supporter un taux croissant des arrivées tant qu'il a de la capacité inutilisée (taux d'utilisation < 1). Le centre de service devient saturé quand son utilisation atteint la valeur 1. Par conséquent, la limite de débit du centre de service k correspond au taux d'arrivée $\lambda_{sat,k}$ à partir duquel le centre de service se sature.

$$p_k\lambda D_k / C \leq 1$$

donc $$\lambda_{sat,k} = C/p_k D_k$$

La limite du taux d'arrivée du système est le plus petit taux d'arrivée λ_{sat} à partir duquel le système se sature et n'arrive pas à traiter les requêtes suivantes avec succès. Le système entier se sature lorsqu'un des centres de service se sature. Par conséquent, λ_{sat} est la plus petite valeur de l'ensemble des valeurs $\lambda_{sat,k}$:

$$\lambda_{sat} = min\ \{\lambda_{sat,k}\}\ pour\ 1 \leq k \leq K$$

Pour illustrer ceci, nous considérons un système dont les paramètres sont donnés au tableau 3.1. Les limites du débit de chaque centre de service et du système entier sont données au

tableau 3.2. Le taux d'utilisation U_k des quatre centres de service de l'exemple est représentée dans la figure 3.12. Le système se sature lorsque le taux d'arrivée des requêtes atteint la valeur 666.66 requêtes par seconde.

Nombre de serveurs	4	Distribution probabiliste	$p_1 = p_2 = p_3 = p_4 = 0.25$
Nombre de classes de service	3	Demandes de service (ms)	Classe 1: $D_{1,1..4} = [5, 8, 4, 3]$ Classe 2: $D_{2,1..4} = [7, 6, 5, 2]$ Classe 3: $D_{2,1..4} = [5, 4, 5, 6]$

Tableau 3.1 - Paramètres du système exemple

Taux de saturation des centres de service (requêtes par seconde) (RPS)	$\lambda_{sat,1}$=705.88 $\lambda_{sat,2}$ = 666.66 $\lambda_{sat,3}$ = 857.14 $\lambda_{sat,4}$ = 1090.91
Taux de saturation du système (RPS)	λ_{sat} = 666.66

Tableau 3.2 - Limites du débit du système

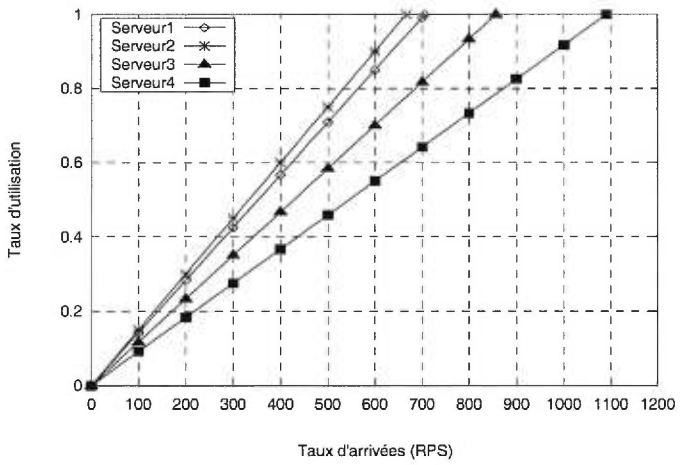


Figure 3.12 - taux d'utilisation du système

3.4.3.2 Temps de réponse

Les limites du temps de réponse représentent le plus grand et le plus petit temps de réponse rencontrés par les clients lorsque le taux d'arrivée des requêtes est λ. Pour simplifier l'analyse, nous considérons le cas où toutes les classes de service ont le même taux d'arrivée

des requêtes. C'est-à-dire que : $\lambda_j=\lambda/C$ pour $1\leq j \leq C$. Par conséquent, le temps de réponse $R_{j,k}$ est donné par:

$$R_{j,k} = D_{j,k} / [1-p_k D_k \lambda/C] = D_{j,k} / [1- \lambda/\lambda_{sat,k}]$$

La figure 3.13 schématise le temps de réponse $R_{1,1}$ comme une fonction du taux d'arrivée λ des requêtes. Quand ce taux est très faible alors le temps de réponse est presque égal à la demande de service $D_{j,k}$ car il n'y pas de requêtes dans la file d'attente du centre de service. Cependant, le temps de réponse augmente rapidement au fur et à mesure que le taux d'arrivée augmente et devient extrêmement grand lorsque le taux d'arrivée atteint le taux de saturation du centre de service $\lambda_{sat,k}$.

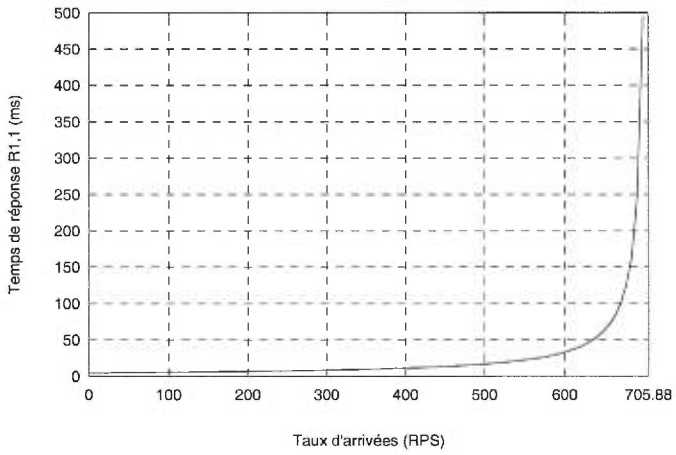


Figure 3.13 - Temps de réponse de la classe de service 1 au centre de service 1

3.4.3.3 Commentaires

L'analyse asymptotique du taux d'arrivée et du temps de réponse montre que tant que le taux de saturation du système n'est pas atteint, tous les centres de service peuvent exécuter des requêtes avec succès. Le taux de saturation d'un centre de service k dépend du nombre de classes de service C, de la probabilité de sélection de ce centre de service par le répartiteur, et de la demande de service de toutes les classes de service au centre de service en question. Ces facteurs doivent être choisis soigneusement afin d'avoir des valeurs plus grandes pour le taux de saturation. Si nous supposons que D_k est le même pour tous les centres de service, c'est-à-dire que D_k est égal à une valeur commune D pour tous les centres de service, alors la

distribution probabiliste { p_i } est le facteur principal pour avoir une grande valeur pour le taux de saturation.

3.5. Conclusion

Dans ce chapitre, nous avons présenté la problématique de distribution de charge au niveau opération (méthode) dans les systèmes distribués à base d'objets. Nous avons montré que le problème d'assignation de requêtes devient très compliqué lorsque les méthodes appelées invoquent à leur tour les méthodes d'autres objets locaux ou distants, constituant ainsi un graphe acyclique orienté d'invocations de méthodes. L'assignation optimale est difficile à réaliser vu la complexité du graphe associé à une méthode et en raison d'autres facteurs entrant en jeu, à savoir l'hétérogénéité au niveau matériel et logiciel des systèmes sous-jacent, l'hétérogénéité des modèles objets et des langages de programmation, etc.

Nous avons ensuite présenté trois approches pratiques pour réaliser l'assignation sous-optimale des requêtes. Ces approches sont : l'approche orientée client basée sur les choix des clients, l'approche orientée répartiteur basée sur l'utilisation d'un répartiteur central, et l'approche orientée serveur basée sur la coopération entre les serveurs. Dans le cas de l'approche orientée serveur, nous avons discuté comment adapter les stratégies source-initiative et receveur-initiative au contexte des systèmes distribués à base d'objets. L'architecture LoDACE décrite au chapitre 4 permet de mettre en oeuvre ces trois approches.

Finalement, nous avons présenté un modèle analytique de l'approche orientée répartiteur basé sur un modèle ouvert de files d'attente à multiples classes de service. Ce modèle permet d'évaluer les performances moyennes du système. L'analyse asymptotique des résultats obtenus montre que les serveurs du système peuvent exécuter avec succès les requêtes des clients tant que le taux d'arrivée des requêtes n'a pas atteint un taux de saturation au bout duquel les temps de réponse deviennent rapidement très grands.

Chapitre 4

LoDACE : architecture d'assignation de requêtes

Ce chapitre[9] présente LoDACE, une architecture pour la distribution de charge par l'assignation de requêtes (invocations de méthodes) dans un environnement distribué à base d'objets. Cette architecture permet de mettre en œuvre les approches d'assignation discutées au chapitre précédent.

4.1. Introduction

La distribution de charge dans les architectures client/serveur est souvent réalisée pour des types spécifiques de service et pour un nombre limité de serveurs. Comme exemples de tels systèmes, citons les serveurs de bases de données et les serveurs Web. Notre objectif est de concevoir dans un environnement distribué à base d'objets un cadre d'application générique qui peut : (1) supporter n'importe quel type de service, (2) être configuré facilement pour réaliser différentes approches de distribution de charge, (3) être déployé dans un environnement où les serveurs peuvent être créés ou supprimés dynamiquement, et (4) être évolutif (scalable) en supportant un grand nombre de clients et de serveurs et en s'adaptant à la taille du système. Ce dernier aspect est discuté en détail au chapitre 7.

Dans ce cadre d'application, les serveurs sont organisés dans des grappes logiques (ou *clusters*). Les serveurs d'une même grappe offrent le même type de service en implémentant une même interface. Le but est de permettre la distribution des requêtes entre les serveurs d'une même grappe selon une approche de distribution de charge donnée. Notons que des grappes distinctes peuvent utiliser des approches différentes. Par exemple, une grappe peut utiliser une approche orientée répartiteur basée sur la sélection aléatoire des serveurs. Une

[9] Les résultats de ce chapitre sont publiés dans [Badidi98a,b,c].

autre grappe peut utiliser l'approche orientée serveur basée sur la coopération entre les serveurs de la grappe par échange des informations de charge.

Plusieurs travaux de recherche s'intéressent à la communication de groupe dans les environnements distribués objet dans le but d'implémenter la tolérance aux fautes [Maffeis96, Felber96,97]. Avec cette approche, une requête d'un client est envoyée à toutes les copies d'un serveur répliqué qui forment le groupe. Le client peut alors recevoir une ou plusieurs réponses à sa requête. Contrairement à cette vision, la distribution de charge optimale requiert qu'une requête soit traitée par une seule instance de serveur qui est choisie par la politique de localisation. L'objectif de notre architecture LoDACE (Load Distribution Architecture for Distributed Object Computing Environments) s'inscrit dans cette optique [Badidi98a,b,c]. Sa mise en œuvre dans des environnements distribués objet comme CORBA et DCOM permettra de l'utiliser dans des applications distribuées dans lesquelles les clients et les serveurs peuvent être écrits dans des langages hétérogènes et déployés sur des plates-formes hétérogènes.

Le reste de ce chapitre est organisé comme suit : la section 4.2 présente un aperçu de l'architecture LoDACE en décrivant les composantes de cette architecture. La section 4.3 décrit la mise en œuvre de l'assignation de requêtes dans LoDACE. La section 4.4 décrit une implémentation d'un prototype de cette architecture dans un environnement CORBA. Finalement, la section 4.5 conclut ce chapitre.

4.2. Apercu de l'architecture LoDACE

La figure 4.1 schématise l'architecture LoDACE qui comporte les éléments suivants :

1. les objets clients et serveurs d'une application distribuée,
2. une couche middleware servant à acheminer les requêtes et les données entre l'ensemble des composantes de l'architecture, et
3. un ensemble de services pour réaliser l'assignation des requêtes des objets clients aux objets serveurs. Ces services sont : (1) *le service de découverte des serveurs* (Server Discovery Service), (2) *le service de surveillance de la charge* (Load Monitoring Service), et (3) *le service d'association* (Binding Service).

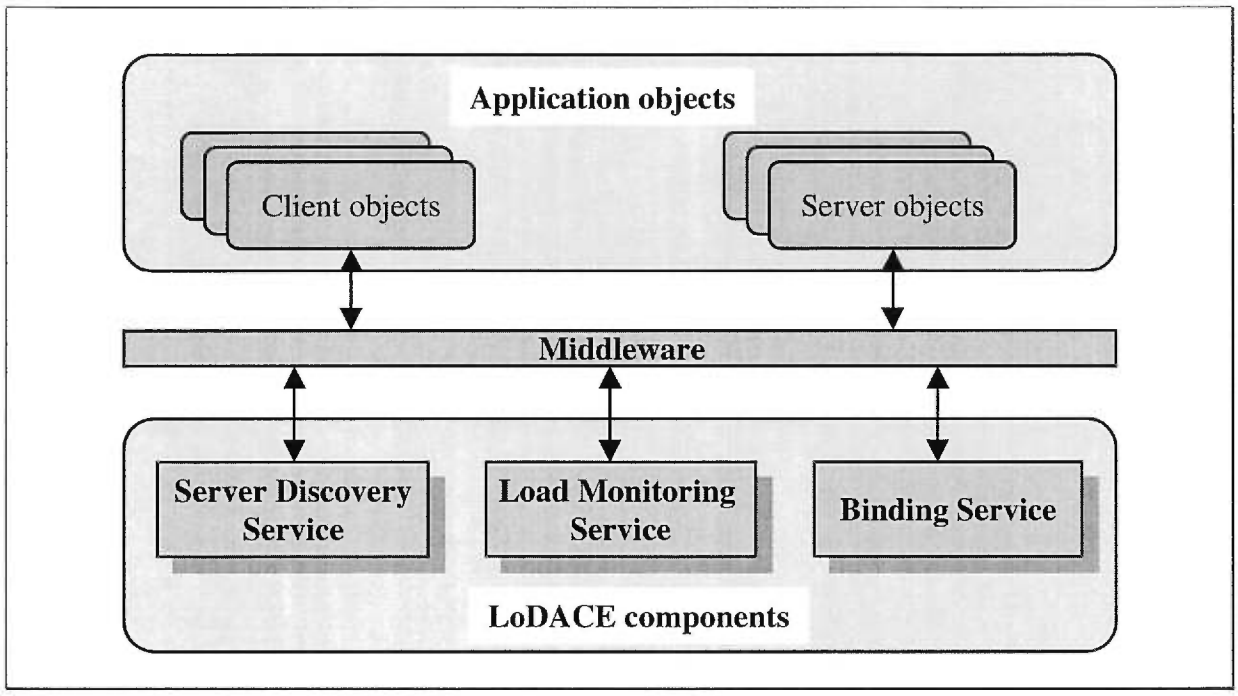


Figure 4.1 - Architecture LoDACE

LoDACE est une architecture à un haut niveau d'abstraction. Les services constituant cette architecture peuvent être implémentés de plusieurs façons en considérant des approches centralisées ou distribuées pour chacun de ces services. Nous présentons dans la suite de cette section une brève explication du rôle des objets du niveau application, ensuite nous expliquons la fonction de chaque service ainsi que leurs interactions.

4.2.1. Objets du niveau application

Les objets du niveau application sont de deux types : les objets serveurs et les objets clients. Les objets serveurs offrent divers types de service et implémentent diverses interfaces. Les objets clients utilisent les services offerts par les objets serveurs. Ces objets dépendent du domaine d'application considéré. Un objet serveur offrant un type de service donné peut se comporter comme client pour un autre type de service. Les objets qui offrent un même type de service forment une grappe logique de serveurs entre lesquels il sera possible de distribuer la charge.

Dans une application distribuée à base d'objets, les objets peuvent être créés ou supprimés dynamiquement. Par exemple, des serveurs peuvent être répliqués pour augmenter leur disponibilité et répondre à une demande croissante de leurs clients. Notons que le maintien de

la consistance entre les serveurs répliqués n'est pas l'objectif de ce travail de thèse. Plusieurs travaux de recherche étudient les techniques de réplication pour mettre en oeuvre la tolérance aux fautes dans les systèmes distribués [Guerraoui96]. Dans l'architecture LoDACE, les serveurs constituent les noeuds du système de distribution de charge. Contrairement à la configuration des noeuds dans les systèmes de distribution de charge classiques qui est généralement statique, la configuration des serveurs dans LoDACE est dynamique car les serveurs peuvent rejoindre ou quitter une grappe en cours d'exécution.

4.2.2. Service de découverte des serveurs

Vu que les environnements distribués objets sont dynamiques, la découverte dynamique des serveurs est essentielle pour l'implémentation de la distribution de charge dans ces environnements. Le service de nommage (Naming service) et le service de courtage (Trading service) sont deux services de base pour la localisation des objets dans un environnement distribué ouvert. Ces deux services ont été standardisés par l'ISO dans le cadre du modèle de référence ODP des systèmes ouverts et par l'OMG dans le cadre des services standards CORBA. Le service de courtage est plus approprié pour la découverte dynamique des serveurs car il est basé sur les concepts de type de service et d'offre de service. Nous présentons une brève description de ce service à la section 4.4.1 et une description plus détaillée à l'annexe 2.

4.2.3. Service de surveillance de la charge

La sélection des serveurs cible pour traiter les requêtes des objets clients sur la base de la charge actuelle des serveurs, requiert la mise en œuvre d'un service de surveillance de la charge qui est responsable d'évaluer la charge des serveurs et de communiquer l'information de charge à d'autres composantes de l'architecture. Ce service peut être implémenté d'une manière centralisée ou distribuée. Dans les deux cas, des aspects techniques doivent être considérés lors de la conception et de l'implémentation de ce service, à savoir : la définition d'un indicateur de charge et la dissémination (collecte et échange) de l'information de charge.

4.2.3.1 Indicateurs de charge

L'information de charge est typiquement représentée par un indicateur de charge qui est une mesure quantitative de la charge d'un serveur. Un indicateur de charge est défini comme une variable non négative qui prend la valeur zéro si la ressource est inoccupée et des valeurs positives dans le cas contraire [Ferrari87]. La mise à jour périodique de l'indicateur de charge est nécessaire quand la charge des serveurs change avec le temps. Cependant, le choix de la durée de cette période est très important. En effet, les mises à jour à des périodes courtes peuvent entraîner des coûts de communication excessifs, et les mises à jour sur des périodes longues peuvent rendre les indicateurs de charge obsolètes.

Une large variété d'indicateurs de charge est décrite dans la littérature [Devarankonda89, Ferrari87, Hac90, Kunz91, Svensson90]. Une liste partielle d'indicateurs de charge possibles inclut : longueur de la file d'attente des processus prêts à l'exécution, utilisation CPU, temps de réponse ou temps de traitement, etc. Comme exemples d'indicateurs de charge pouvant être considérés dans LoDACE, citons le taux d'utilisation d'un serveur (c.-à-d. taux d'occupation du serveur pendant une période de temps donnée) et le nombre de requêtes exécutées pendant une unité de temps par le serveur.

4.2.3.2 Dissémination de l'information de charge

La collecte de l'information de charge coûte souvent cher, et l'échange trop fréquent de cette information peut conduire à des coûts de communication prohibitifs. Des méthodes pour la collecte et l'échange de l'information de charge ont été décrites à la section 2.2.2.1. Rappelons que les deux méthodes utilisées dans la pratique sont l'exploration sur demande et la mise à jour périodique. Nous discutons brièvement ces deux méthodes dans la suite de cette section, et nous présentons ensuite quatre méthodes supplémentaires suggérées dans la littérature.

L'exploration sur demande consiste à interroger tous les serveurs potentiels afin d'obtenir la valeur de la charge courante de chaque serveur. Un vecteur de charge courante est alors constitué avant qu'une nouvelle requête ne soit assignée à l'un de ces serveurs. L'inconvénient de cette méthode est qu'une requête risque d'être retardée en attendant la constitution du vecteur de charge. D'autre part, une augmentation du nombre d'arrivées de requêtes dans le système entraîne une augmentation significative du trafic sur le réseau. La

mise à jour périodique est plus rapide que l'exploration sur demande quand l'information de charge est stockée localement. Le problème principal de cette méthode est que l'information de charge peut devenir obsolète entre deux mises à jour successives, et par conséquent, les assignations résultantes risquent d'être inadéquates. Ainsi, l'ajustement avec précision de la période de mise à jour est fondamental.

Quatre autres méthodes décrites dans la littérature sont : L'interrogation (polling), la diffusion (broadcast), la communication de groupe (multicast), et l'utilisation d'un coordinateur central.

Interrogation – Un message est envoyé à la fois à un seul serveur pour obtenir la valeur de sa charge courante. Si l'interrogation est faite sur demande, alors le résultat est une information de charge qui est à jour.

Diffusion – C'est une forme de communication unidirectionnelle dans laquelle tous les serveurs échangent l'information de charge par diffusion sur le réseau. Cette méthode présente certains inconvénients : (1) elle peut conduire à une augmentation inacceptable du trafic réseau, (2) tous les serveurs reçoivent l'information de charge qu'ils soient impliqués ou non dans la distribution de charge, et (3) il peut y avoir un délai de propagation grand dans le cas de grands réseaux.

Communication de groupe – C'est une forme de diffusion limitée aux membres d'un groupe donné. Seuls les membres du groupe reçoivent l'information de charge. Il peut y avoir aussi beaucoup de trafic réseau. La communication de groupe a été utilisée par Theimer et al. dans le système V [Theimer85].

Coordinateur central – L'utilisation d'un coordinateur central a été discutée à la section 2.2.1.1.

Dans l'architecture LoDACE, nous avons retenu les deux premières méthodes car elles sont faciles à réaliser et elles représentent les méthodes utilisées par la majorité des systèmes de distribution de charge. Le service de surveillance de charge du prototype de LoDACE, que nous décrivons à la section 4.4.2, utilise la méthode d'exploration sur demande.

4.2.4. Service d'association

Le service d'association (Binding service) représente l'interface externe de l'architecture LoDACE vis-à-vis des objets clients. Il permet aux clients d'avoir un accès transparent au service de découverte des serveurs et au service de surveillance de la charge. Comme pour les services décrits précédemment, le service d'association peut être implémenté de plusieurs façons. Par exemple, dans le cas d'un service centralisé, une seule composante appelée *Binder* offrirait ce service. Dans le cas d'un service distribué, plusieurs Binders distribués dans le système offriraient ce service. Le système peut être découpé en domaines, et un Binder serait associé à chaque domaine. Tous les clients d'un même domaine adresseraient leurs requêtes au Binder associé à leur domaine. La référence objet du Binder d'un domaine donné peut être obtenue auprès du service de nommage. Le Binder est similaire à un répartiteur capable de mettre en oeuvre plusieurs stratégies d'assignation de requêtes.

4.3. Assignation de requêtes dans LoDACE

Dans cette section, nous décrivons comment les approches présentées à la section 3.3 peuvent être mises en œuvre par l'architecture LoDACE. Dans toutes ces approches, l'assignation des requêtes des clients s'effectue principalement en trois phases :

1. ***Phase de découverte des serveurs*** – Les serveurs offrant le service requis sont identifiés.
2. ***Phase de sélection du serveur*** – Un serveur cible parmi les serveurs identifiés à la phase précédente est choisi suivant une stratégie donnée pouvant être statique ou dynamique.
3. ***Phase d'invocation de service*** – Cette phase correspond à l'assignation effective de la requête au serveur choisi.

Pour mesurer et comparer l'efficacité des algorithmes d'assignation de requêtes, un indicateur de performance doit être défini. Dans les architectures client/serveur, *le temps de réponse moyen* (average response time) par type de requête est l'indicateur de performance le plus communément utilisé. Il est aussi usuel de considérer la déviation standard du temps de réponse moyen qui permet de mesurer la variabilité du temps de réponse autour de sa moyenne. Du point de vue du système, *le débit* (throughput) du système, exprimé en nombre de requêtes traitées pendant une unité de temps, peut aussi être considéré comme indicateur

de performance. L'objectif est souvent de réduire le temps de réponse moyen et d'augmenter le débit du système. L'indicateur de performance dans LoDACE peut être exprimé aussi bien par le temps de réponse moyen que par le débit moyen du système. Nous considérons le temps de réponse moyen comme indicateur de performance dans l'expérimentation, décrite au chapitre 6, que nous avons conduite à l'aide du prototype de LoDACE réalisé dans un environnement CORBA.

4.3.1. Approche orientée client

Comme cette approche est basée sur les choix volontaires des clients, la sélection du serveur cible s'effectue par le client. Par conséquent, le client doit identifier les serveurs capables de le servir. Cette information dynamique est obtenue directement à partir du service de découverte des serveurs. Avec cette approche, le service d'association n'intervient pas dans la sélection du serveur cible. Chaque client établit sa propre stratégie pour l'assignation de requêtes. Les stratégies statiques ne nécessitent pas l'information d'état dynamique des serveurs. Cependant, si le client souhaite utiliser une stratégie dynamique telle que la sélection du serveur le moins chargé, il doit avoir le moyen d'obtenir l'information de charge courante des serveurs cibles. La figure 4.2 schématise trois approches dynamiques à l'assignation de requêtes.

Première approche (Figure 4.2 (a))

Le client a accès seulement au service de découverte des serveurs, et l'information de charge est considérée comme propriété dynamique des offres de service. Le client adresse une requête d'importation au service de découverte des serveurs pour identifier toutes les offres de service qui répondent à ces besoins (opération 1). Une fois qu'elles sont identifiées, le service de découverte des serveurs obtient l'information de charge des serveurs auprès du service de surveillance de la charge (opérations 2 et 3), et il retourne au client la liste des serveurs répondant à sa requête avec leurs informations de charge (opération 4). Le client effectue alors la sélection d'un serveur cible d'après ses propres stratégies. Par exemple, il peut choisir le serveur le moins chargé à cet instant pour lui soumettre ses requêtes de service. Ce modèle peut nécessiter des modifications au service de découverte des serveurs pour qu'il puisse

communiquer avec le service de surveillance de la charge afin d'obtenir l'information de charge des serveurs.

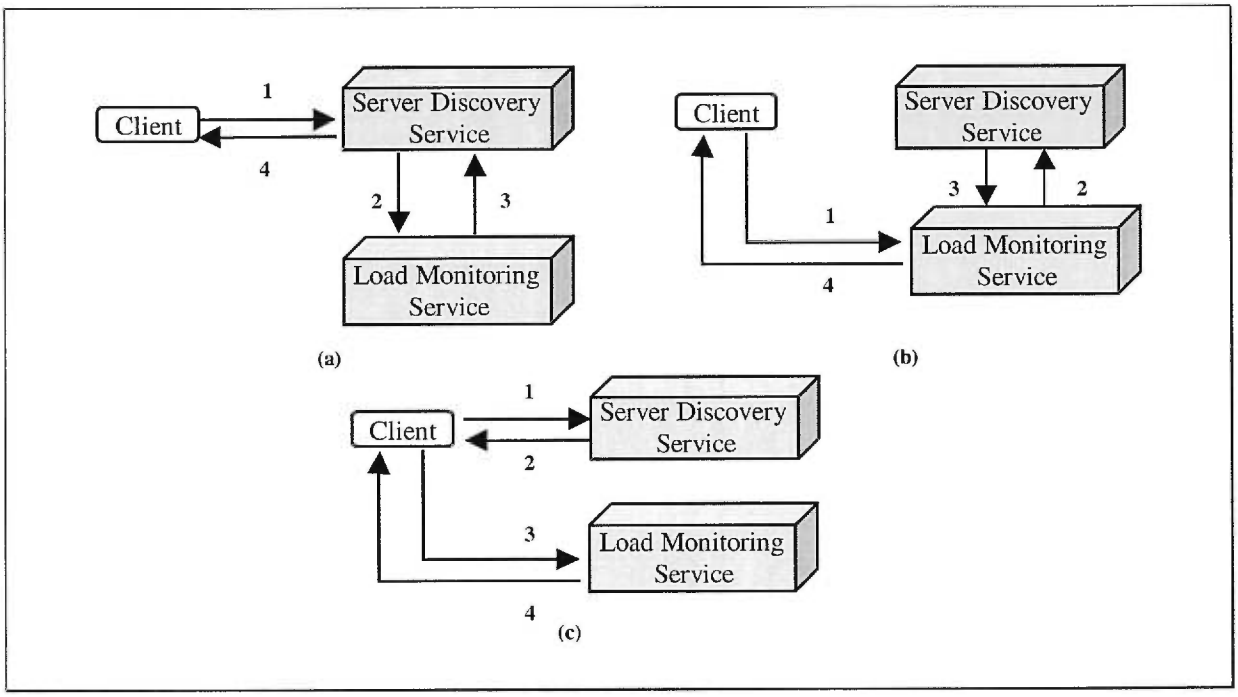


Figure 4.2 - Approche dynamique orientée client

Seconde approche (Figure 4.2 (b))

Le client a accès seulement au service de surveillance de la charge. Une requête du client est adressée à ce service (opération 1) qui l'achemine au service de découverte des serveurs pour obtenir la liste des offres de services qui satisfont la requête du client (opération 2). Une fois obtenue cette liste (opération 3), elle est retournée au client avec l'information de charge de chaque serveur figurant dans la liste (opération 4). Le client effectue alors la sélection d'un serveur cible selon ses propres stratégies. Ce modèle ne requiert pas des modifications au service de découverte des serveurs. Cependant, le service de surveillance de la charge doit être en mesure de recevoir les requêtes des clients et de les acheminer au service de découverte des serveurs.

Troisième approche (Figure 4.2 (c))

Le client a accès aussi bien au service de découverte des serveurs qu'au service de surveillance de la charge. Dans un premier temps, le client envoie une requête d'importation

au service de découverte des serveurs pour obtenir la liste des serveurs qui offrent le service requis (opérations 1 et 2). Dans un second temps, le client obtient l'information de charge des serveurs de la liste précédente auprès du service de surveillance de la charge (opérations 3 et 4). À partir de cet instant, le client effectue la sélection d'un serveur cible d'après ses propres stratégies. L'inconvénient de ce modèle réside dans le fait que l'obtention de l'information de charge n'est pas transparente au client. En outre, si le client ne consulte pas le service de surveillance de la charge pour obtenir l'information de charge des serveurs, il peut importer le service requis de n'importe quel serveur retourné par le service de découverte des serveurs. Ceci ne garantit pas d'avoir un partage efficace de la charge entre les serveurs.

4.3.2. Approche orientée répartiteur

Dans l'architecture LoDACE, le service d'association joue le rôle du répartiteur. Il permet d'utiliser diverses stratégies d'assignation qui peuvent être statiques ou dynamiques (tenant compte de l'état des serveurs) [Badidi98b]. Nous présentons dans cette section quatre exemples de stratégies : aléatoire, cyclique, enchère, et sélection du serveur le moins chargé. Les deux premières stratégies sont très utilisées dans les architectures client/serveur et dans les serveurs Web pour réaliser la distribution de charge. La troisième stratégie est décrite dans la littérature sur l'ordonnancement des tâches et l'allocation de ressources dans les systèmes distribués. La quatrième stratégie est utilisée dans la distribution dynamique de charge dans les systèmes distribués classiques.

La stratégie aléatoire (random – RD) : tous les serveurs ont la même probabilité d'être choisi. Un des serveurs est choisi de manière aléatoire. Cette stratégie est facile à mettre en oeuvre car elle ne tient pas compte de l'information d'état sur les serveurs.

La stratégie cyclique (round robin – RR) : tous les services sont sélectionnés de manière cyclique. Cette stratégie ne tient non plus compte de l'information d'état sur les serveurs.

La stratégie d'enchère (bidding -- BD) : cette stratégie est basée sur un modèle de marché avec des offres et des demandes. Elle consiste à activer une enchère par le répartiteur lors de l'arrivée d'une nouvelle requête d'un client. Les serveurs capables de servir cette requête soumettent alors leurs offres (bids) au répartiteur. Le prix d'une offre peut dépendre de plusieurs paramètres comme le taux de service, la localité des données, le coût des E/S, la

taille de la requête, etc. Le répartiteur choisit alors l'offre gagnante de l'enchère et assigne la requête au serveur correspondant. Les serveurs n'ont aucune connaissance des offres des autres serveurs et opèrent de manière autonome. La stratégie d'enchère a été utilisée pour l'allocation des ressources dans le système *Spawn* [Waldspurger92]

Stratégie de sélection du serveur le moins chargé (least-loaded – LL) : cette stratégie est dynamique car elle est basée sur l'information de charge des serveurs. Les étapes de cette stratégie, schématisées dans la figure 4.3, sont comme suit :

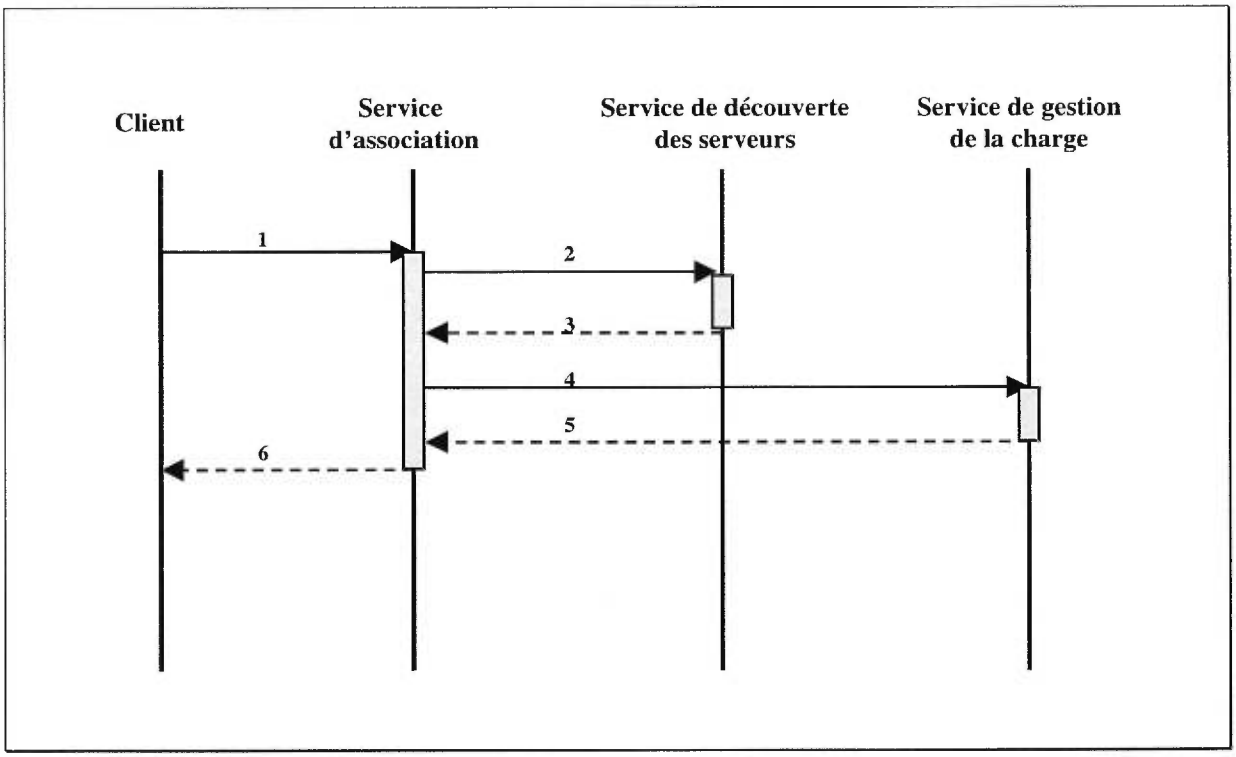


Figure 4.3 - Stratégie de sélection du serveur le moins chargé

1- Le client envoie sa requête, qui spécifie le type de service désiré, au service d'association.

2- Le service d'association achemine la requête du client au service de découverte des serveurs pour identifier les serveurs capables d'offrir le service requis par le client.

3- Le service de découverte des serveurs retourne au service d'association la liste des offres de service pour le type de service demandé. Chaque offre de service décrit les propriétés du service et l'adresse (référence objet) du serveur offrant le service requis.

4- Le service d'association envoie cette liste au service de surveillance de la charge afin d'obtenir l'information de charge de chacun des serveurs figurant dans la liste.

5- Le service de surveillance de charge retourne l'information de charge au service d'association. Cette information est déterminée suivant la politique d'information (périodique ou sur demande) implémentée par le service de surveillance de la charge.

6- Le service d'association, jouant le rôle de répartiteur, choisit alors le serveur le moins chargé. La référence de ce serveur est retournée au client comme résultat de sa requête. À partir de cet instant, le client peut invoquer son service requis à partir de ce serveur.

L'avantage de cette approche est que la sélection du serveur cible est transparente au client. Celui-ci a besoin de communiquer seulement avec le service d'association. Elle présente, cependant, un inconvénient. Si le client envoie ses requêtes ultérieures au même serveur, alors il n'est pas garanti que le serveur en question soit toujours le moins chargé. Pour éviter cette situation, le client doit implémenter un proxy intelligent qui intercepte les requêtes du client et obtient pour chaque requête l'adresse du serveur cible à travers les étapes 1-6 précédentes. Cette facilité est offerte par exemple par l'ORB commercial OrbixWeb [Iona96].

4.3.3. Approche orientée serveur

L'architecture LoDACE constitue une plate-forme pour la réalisation de l'approche orientée serveur que nous avons décrit en détail à la section 3.3.3. Les serveurs sont organisés dans des grappes suivant le type de service qu'ils offrent. Pour réaliser les stratégies source-initiative et receveur-initiative qui rentrent dans le cadre de cette approche, chaque serveur dispose d'une vue de la grappe à laquelle il appartient par consultation du service de découverte des serveurs. La distribution de charge entre les serveurs d'une même grappe est réalisée à travers la coopération entre ses serveurs par échange de leur information de charge. Le service d'association n'intervient pas dans la sélection du serveur cible car elle est faite par les politiques implémentées par les serveurs.

4.4. Mise en oeuvre de LoDACE dans un environnement CORBA

Nous avons réalisé un prototype de l'architecture LoDACE dans un environnement middleware comprenant l'ORB OrbixWeb et le service de courtage OrbixTrader [Iona97]. Le

langage de programmation utilisé est Java. Ce prototype est déployé sur un réseau local Ethernet avec deux postes de travail fonctionnant sous Solaris 2.5.

4.4.1. Service de courtage

Le service de courtage (Trading service), standardisé par l'ISO et par l'OMG, permet la découverte dynamique des serveurs sur la base de leurs types de service et des contraintes spécifiées par les clients. La figure 4.4 illustre le principe de fonctionnement du service de courtage. Pour plus de détails sur ce service voir l'annexe 2.

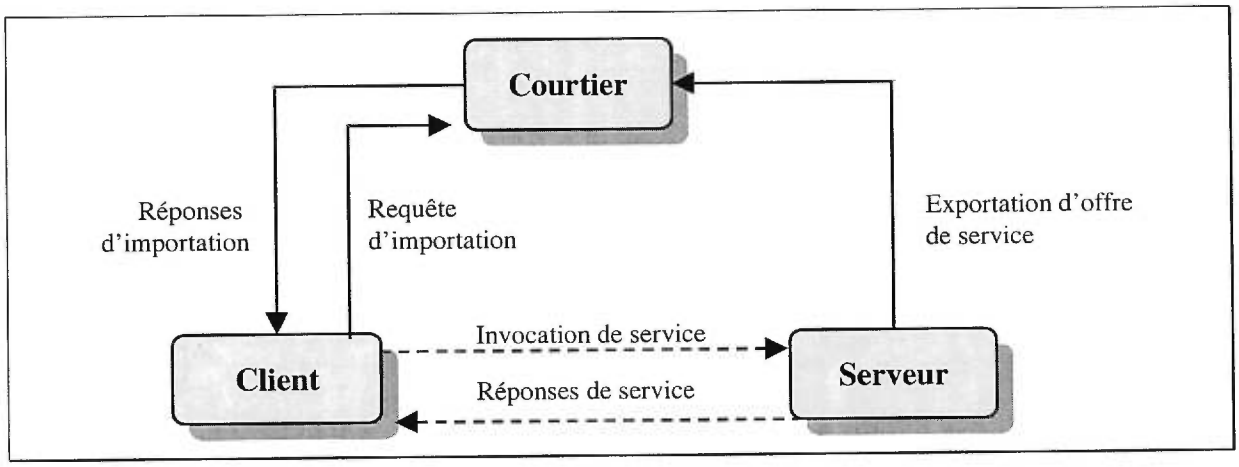


Figure 4.4 - Principe de fonctionnement du service de courtage

Le *courtier* (trader) maintient une base de données d'offres de service (c.-à-d. description des services offerts par les serveurs). Il permet aux serveurs d'enregistrer leurs offres de service (exportation) et aux clients d'obtenir les services requis (importation) sans connaissance préalable de la localisation des serveurs. Les clients formulent leurs requêtes d'importation en spécifiant leurs contraintes sur les propriétés des offres de service et leurs préférences en ce qui concerne la présentation des résultats. Le courtier maintient aussi une base de données des types de service.

4.4.2. Service de surveillance de la charge

Pour les besoins d'expérimentation des stratégies d'assignation de requêtes, nous avons adopté une approche centralisée pour la gestion de l'information de charge. La figure 4.5

schématise les composantes de ce service de surveillance de la charge, et la figure 4.6 décrit la spécification IDL correspondante.

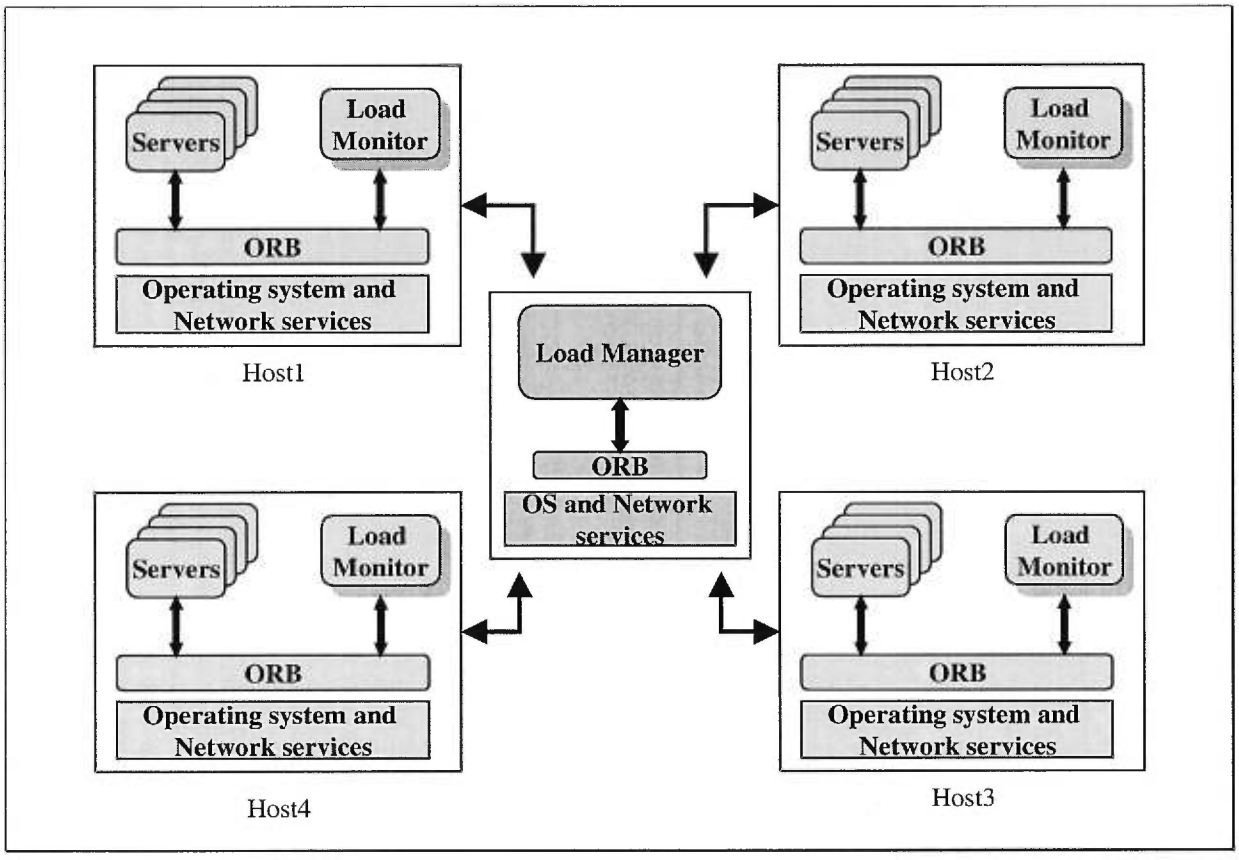


Figure 4.5 - Gestion de l'information de charge

```
module LoadMonitoring {
...
    interface LoadManager {
    ...
        readonly attribute LMTable lm_table;
        readonly attribute ServerTable server_table;
        readonly attribute LIB lib_table;

        void registerLoadMonitor(in string LoadMonitorName, in string
             HostName);
        Void  registerServer(in string ServerName, in string HostName);
        Void   sendCurrentLoad(in string ServerName, in string
               HostName, in double HostLoadValue, in double
               ServerLoadValue)
            raises ( IllegalLoadValue);
        short getLightlyLoaded(in CosTrading::OfferSeq offerSeq);
    };
```

```
interface LoadMonitor{
    ...
        readonly attribute ServerTable server_table;

        void registerServer(in string ServerName);

        short sendRequestStartTime(in string ServerName, in string
              CalledOperation,   in string TargetObject);
        void sendRequestEndTime(in short RequestNumber);
        void getLoadValue(in string ServerName, out double
              HostLoadValue, out double ServerLoadValue)
            raises (IllegalLoadValue);
    };
};
```

Figure 4.6 - Spécification IDL du service de surveillance de la charge

Un gestionnaire de charge (load manager) est responsable de la collecte de l'information de charge courante des serveurs et des machines dans lesquelles résident ces serveurs. La politique d'information implémentée est du type sur demande. La charge des serveurs et des machines est évaluée à l'aide de *moniteurs de charge* (load monitors) qui sont déployés dans chaque machine du système. Dans le cas d'un système à grande échelle, plusieurs domaines peuvent être considérés, et un gestionnaire de charge serait déployé par domaine. Ces gestionnaires de charge peuvent communiquer par échange de l'information de charge.

4.4.2.1 Évaluation de la charge d'un serveur

Nous avons choisi comme indicateur de charge *le taux d'utilisation* du serveur, c'est-à-dire le taux d'occupation du serveur pendant une période de temps fixée à l'avance. La durée d'occupation du serveur est calculée en enregistrant les temps d'arrivées des requêtes et les temps de terminaison de leur traitement par le serveur. Ceci est réalisé à l'aide d'un mécanisme de filtre à l'entrée et à la sortie du serveur. A chaque fois qu'une requête arrive au serveur, un signal *sendRequestStartTime* est émis au moniteur de charge local pour le notifier du début de traitement d'une nouvelle requête. Les paramètres de ce signal incluent le nom du serveur, l'objet invoqué, et l'opération sollicitée. Le moniteur de charge attribue un numéro à la requête (request_number) qui est renvoyé au serveur. De la même façon, lors de la fin de traitement de la requête, un signal *sendRequestEndTime* est émis au moniteur de charge pour le notifier de la fin du traitement de la requête. Ce signal comporte un seul paramètre qui correspond au numéro de la requête qui est attribué par le moniteur lors de l'envoi du signal *sendRequestStartTime*.

La figure 4.7 illustre le processus d'évaluation de la charge d'un serveur par utilisation d'un mécanisme de filtre. Le moniteur de charge dispose d'un tableau dans lequel sont enregistrées toutes les requêtes qui arrivent aux serveurs qui résident dans sa machine. Un serveur est identifié par un nom et doit s'enregistrer auprès du moniteur de charge pour que sa charge soit surveillée par ce dernier. Chaque ligne de ce tableau comporte le numéro de la requête, le nom du serveur sollicité, le temps d'arrivée de la requête (T_a), le temps de fin du traitement de la requête (T_f), l'objet cible et l'opération invoquée.

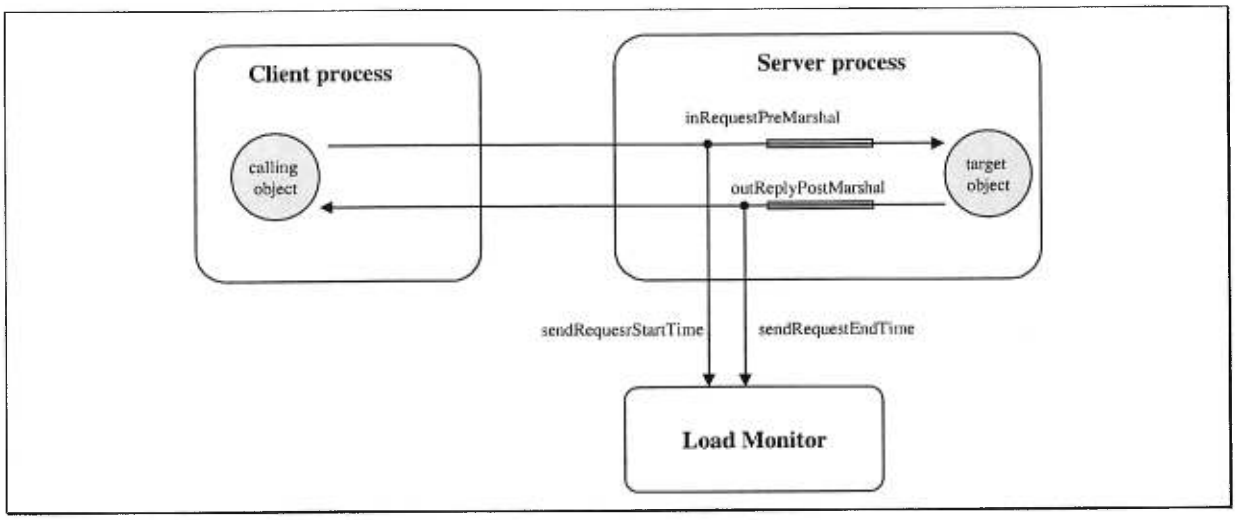


Figure 4.7 - Surveillance de la charge d'un serveur par un mécanisme de filtre

Le temps d'occupation d'un serveur donné pendant une période de temps, commençant à T_1 et se terminant à T_2, est calculé en considérant les requêtes adressées à ce serveur dont les temps d'arrivée (T_a) et de fin du traitement (T_f) répondent à l'une des quatre relations suivantes :

1. $(T_1 \leq T_a \leq T_2)$ et $(T_1 \leq T_f \leq T_2)$
2. $(T_1 \leq T_a \leq T_2)$ et $(T_f > T_2)$
3. $(T_a < T_1)$ et $(T_1 \leq T_f \leq T_2)$
4. $(T_a < T_1)$ et $(T_f > T_2)$

Seules les périodes de traitement qui sont à l'intérieur de la période considérée ($[T_1..T_2]$) sont prises en compte dans le calcul du temps d'occupation du serveur. Nous avons défini l'utilisation du serveur comme étant le rapport entre le temps d'occupation du serveur et la durée de la période de temps considérée.

$$\textit{Utilisation du serveur} = \textit{temps d'occupation}/(T_2 - T_1)$$

4.4.2.2 Évaluation de la charge d'une machine

Pour le système d'exploitation Unix, l'information nécessaire au calcul de la charge est localisée dans le fichier système /dev/kmem. L'accès à ce fichier est réservé au super-utilisateur (root). De plus, le format de ce fichier dépend de la version d'Unix utilisée (HP-Unix, AIX d'IBM, Solaris de Sun, etc.). Ceci rend l'exploitation directe de ce fichier difficile. Nous discutons dans la suite de cette section des commandes Unix qui permettent de déterminer la charge d'une machine, à partir des informations stockées dans le fichier

/dev/kmem, et qui peuvent être exécutées par n'importe quel processus. Ces commandes sont indépendantes de la version d'Unix.

La charge de la machine peut être basée sur la longueur moyenne de la file d'exécution telle qu'elle est retournée par les commandes Unix *uptime* et *w*. Cependant, cette méthode ne donne pas une image précise de la charge. Par exemple, un processus effectuant un calcul intensif peut générer une grande charge CPU tout en ayant une petite longueur moyenne de la file d'exécution. La charge peut aussi être basée sur l'information retournée par la commande *ps*. Cette information inclut le pourcentage de temps CPU de chaque processus ainsi que le temps CPU accumulé par chaque processus. Malheureusement, cette méthode ne donne pas le total de toutes les activités dans la machine. Une autre méthode consiste à utiliser la commande *iostat*. Le résultat de la commande *iostat* inclut le pourcentage de temps CPU de toutes les activités sur la machine qui est divisé en temps utilisateur, temps système et temps d'inoccupation. Les variations dans le format de sortie de la commande *iostat* d'un système à un autre empêchent l'utilisation de cette commande pour l'évaluation de la charge.

Finalement, la commande *vmstat* offre le pourcentage de temps CPU de tous les processus divisé en temps utilisateur, temps système, et temps d'inoccupation. Le temps utilisateur est le pourcentage d'utilisation CPU observé au niveau application durant un intervalle de temps donné. Le temps système est le pourcentage d'utilisation CPU par le noyau durant un intervalle de temps donné. Le temps d'inoccupation est le pourcentage de temps dans lequel le CPU est observé comme étant inoccupé durant un intervalle de temps donné. L'information retournée lors de la première invocation de *vmstat* est un résumé de l'information précédente depuis l'activation du système. La commande *vmstat* donne une représentation plus précise de la charge d'une machine que les commandes *uptime*, *w*, ou *ps*. Cependant, pour des raisons de simplicité et de facilité d'utilisation, nous avons utilisé dans l'expérimentation décrite au chapitre 6 la commande *uptime*.

4.4.3. Service d'association

Le service d'association est utilisé conformément à l'approche considérée pour l'assignation des requêtes de service. Avec l'approche orientée client et l'approche orientée serveur, il suffit d'utiliser directement le service de courtage pour identifier les serveurs éligibles. Le service d'association sera utilisé principalement dans une approche orientée répartiteur qui est

une approche centralisée. Dans ce cas, le service d'association est offert par des composantes *Binder* qui jouent le rôle d'interface entre les clients et les serveurs.

Pour rendre la découverte dynamique des serveurs transparente aux objets clients, le Binder hérite l'interface de consultation (*Lookup*) du service de courtage. Les requêtes d'importation des clients reçues par le Binder sont véhiculées par celui-ci au service de courtage pour déterminer les serveurs éligibles. Le Binder peut être spécialisé pour réaliser les stratégies décrites à la section 4.3.2. Dans le cas des politiques dynamiques basées sur la connaissance de la charge des serveurs lors de la sélection des serveurs cible, le Binder joue également le rôle d'interface avec le service de gestion de la charge. La figure 4.8 donne la spécification IDL du service d'association.

```
#include "CosTrading.idl"
#include <orb.idl>

module Binding {
        typedef string IORstring;
        typedef sequence <IORstring> IORTable;
        interface Lookup;
        interface Binder{
                readonly attribute Lookup look_if;
                IORTable bind();
        };
        interface Lookup:CosTrading::Lookup {
        }
};
```

Figure 4.8 - Spécification IDL du service d'association

4.4.4. Scénario d'utilisation

La figure 4.9 illustre un scénario d'utilisation du prototype en adoptant une approche orientée répartiteur avec la politique dynamique de sélection du serveur le moins chargé. On distingue trois phases dans le processus d'assignation de requêtes : une phase d'enregistrement, une phase d'interrogation, et une phase d'invocation.

4.4.4.1 Phase d'enregistrement

Cette phase correspond au déploiement des moniteurs de charge et des serveurs. Les moniteurs de charge doivent s'enregistrer auprès du gestionnaire de charge (opération 1). Les serveurs doivent également s'enregistrer auprès des moniteurs de charge des machines où ils

résident (opération 2) et auprès du gestionnaire de charge (opération 3), et doivent exporter leurs offres de service au courtier (opération 4). Chaque moniteur de charge maintient une liste des serveurs qui résident dans sa machine, et un numéro unique identifie chaque serveur. Plusieurs serveurs peuvent coexister dans la même machine.

Le gestionnaire de charge maintient trois listes : la liste des moniteurs de charge, la liste des serveurs, et la liste de l'information de charge. La première liste identifie tous les moniteurs de charge qui sont connus par le gestionnaire de charge. La seconde liste identifie tous les serveurs connus par les moniteurs de charge associés. La troisième liste garde les informations de charge, déterminées par les moniteurs de charge, pour chaque serveur.

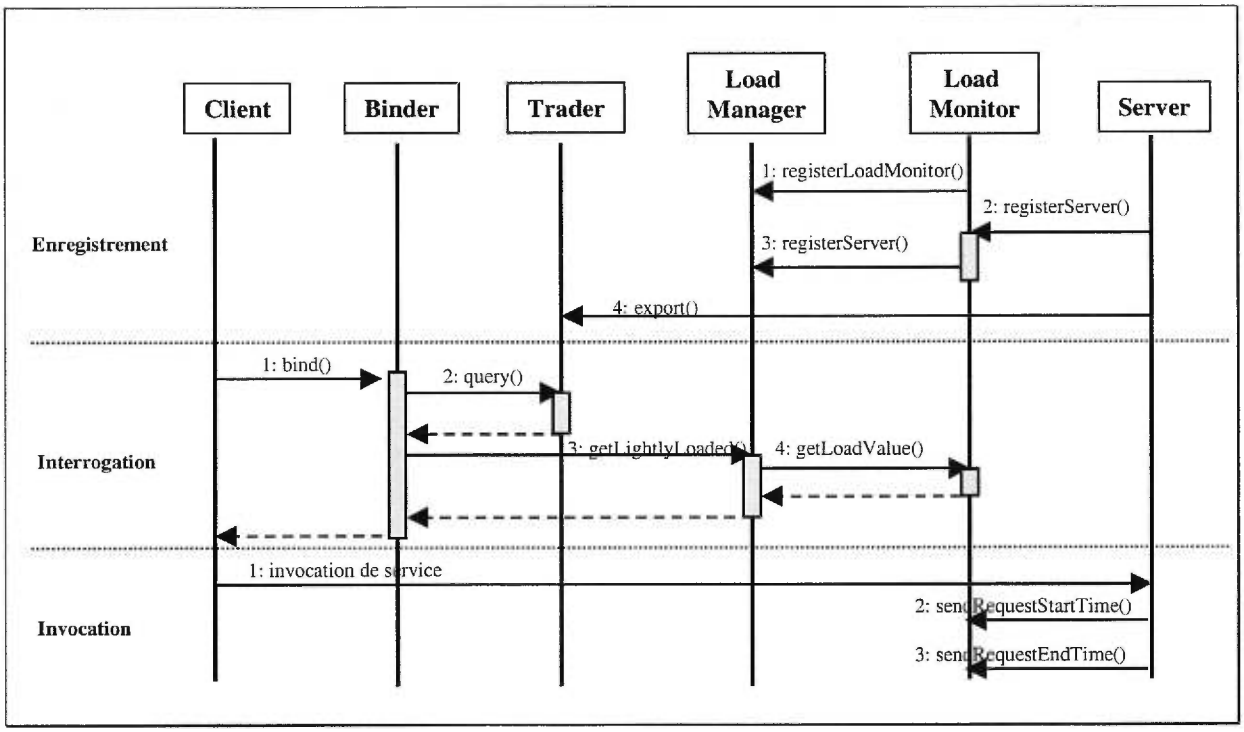


Figure 4.9 - Scénario d'utilisation du prototype

4.4.4.2 Phase d'interrogation

Dans cette phase, le client envoie sa demande d'importation au courtier en invoquant la méthode *bind()* du Binder (opération 1) qui l'achemine à l'interface *Lookup* du courtier (opération 2). Le Binder communique ensuite avec le gestionnaire de charge pour obtenir l'offre de service associée au serveur légèrement chargé parmi les offres de service retournées

par le courtier (opération 3). L'information de charge est obtenue en sollicitant les moniteurs de charge associés (opération 4).

La figure 4.10 montre une applet pour l'interface du Binder. La partie ''Query details'' de cette applet offre la possibilité de choisir un type de service à partir de la liste de types de service maintenus par le courtier dans son répertoire des types de service, et permet au client de spécifier ses contraintes, c'est-à-dire une expression sur les propriétés du type de service, et ses préférences, c'est-à-dire par exemple comment ordonner les offres résultantes.

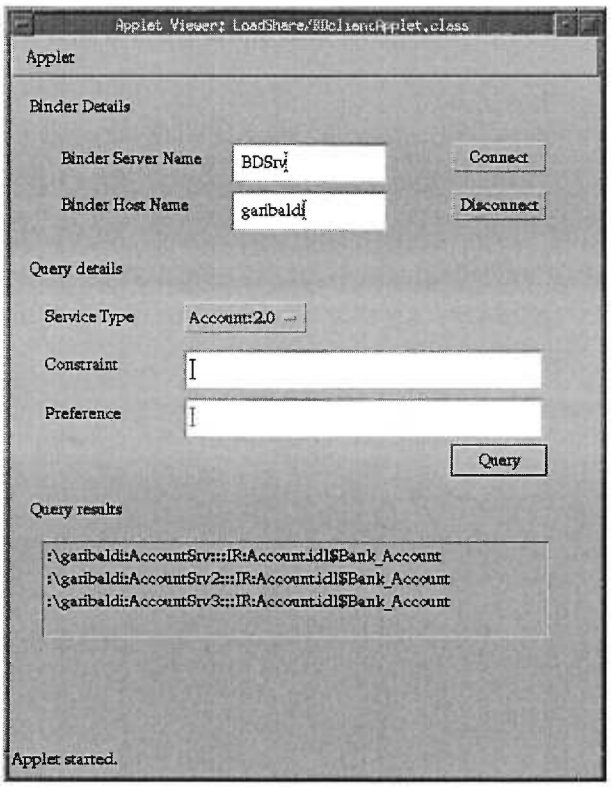


Figure 4.10 - Interface du Binder

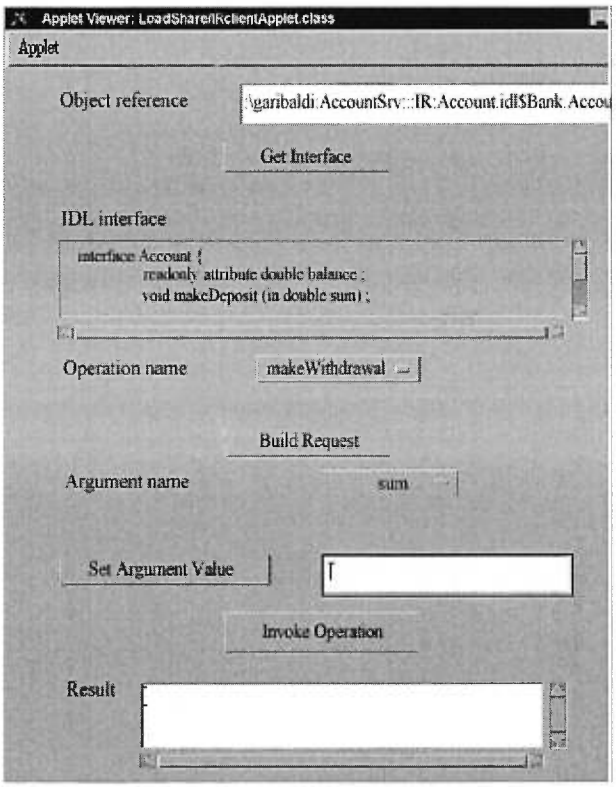


Figure 4.11 - Interface d'invocation dynamique

La liste des serveurs offrant le service requis est affichée et ordonnée selon la charge des serveurs dans la partie ''Query results'' de l'applet. Le premier élément de la liste correspond au serveur le moins chargé. En utilisant l'interface d'invocation dynamique (DII), le client peut découvrir l'interface du serveur choisi et établir des demandes de service. La figure 4.11 montre l'interface d'invocation dynamique pouvant être utilisée par les clients pour établir des demandes de service.

4.4.4.3 Phase d'invocation

Elle correspond à la phase d'invocation du service par le client (opération 1). Chaque requête de service donne naissance à des notifications envoyées par le serveur au moniteur de charge associé. L'arrivée de la requête de service est notifiée par l'opération 2. La fin de traitement de la requête est notifiée par l'opération 3. Ces deux opérations permettent au moniteur de charge d'évaluer la charge du serveur comme nous l'avons présenté à la section 4.4.2.1.

4.5. Conclusion

Nous avons présenté dans ce chapitre l'architecture LoDACE constituant une plate-forme pour la mise en oeuvre de la distribution de charge par l'assignation d'invocations de méthodes dans un système distribué à base d'objets. Cette architecture est composée principalement de trois services : le service d'association jouant le rôle d'interface avec les clients et permettant d'utiliser diverses stratégies d'assignation, le service de découverte des serveurs servant à localiser dynamiquement les serveurs capables d'offrir le service requis par le client, et le service de gestion de la charge servant à évaluer la charge des serveurs et des machines du système.

Nous avons montré comment les trois approches d'assignation, présentées au chapitre 3, peuvent être mises en oeuvre dans l'architecture LoDACE. Le prototype de LoDACE réalisé dans l'environnement OrbixWeb a servi à mener des expérimentations, décrites au chapitre 6, dans le but d'évaluer les performances des stratégies d'assignation de requêtes décrites à la section 4.3.2.

Chapitre 5

LSS : service de distribution de charge dans un environnement CORBA

Ce chapitre[10] présente le service LSS (Load Sharing Service) que nous avons conçu pour réaliser la distribution de charge entre les serveurs d'une application distribuée CORBA. Ce service est conforme aux principes de conception de services CORBA.

5.1. Introduction

La spécification de CORBA définit un ensemble de services standards comme le service de nommage, le service de gestion de cycle de vie, le service de gestion de la persistance, le service de gestion des transactions, le service de gestion des événements, le service de courtage, etc. Cependant, il n'y a aucun service standard pour la distribution de charge. Vu le besoin en la matière, plusieurs vendeurs d'ORBs ont adopté des solutions propriétaires qui utilisent des mécanismes propres à leurs ORBs. Ces solutions sont souvent basées sur des stratégies simples, comme l'assignation aléatoire ou cyclique de requêtes, qui ne tiennent pas compte de la charge réelle des serveurs.

Comme solution, nous avons décrit au chapitre 4 le système LoDACE qui est une architecture générique pour la réalisation de la distribution de charge entre les serveurs d'une application distribuée objet. Une réalisation de cette architecture dans CORBA a été décrite à la section 4.4. Cette implémentation a servi de base dans la conception d'un service de distribution de charge que nous appelons LSS (Load Sharing Service) [Badidi99]. Un objectif important de ce service est d'être portable en se conformant aux principes de conception des services CORBA et d'être indépendant des mécanismes propres à chaque ORB.

[10] Les résultats de ce chapitre sont publiés dans [Badidi99].

5.2. Distribution de charge dans CORBA

Dans cette section, nous décrivons brièvement les mécanismes utilisés par trois ORBs, *VisiBroker*, *CorbaPlus* et *BEA ObjectBroker*, pour réaliser la distribution de charge. Ces ORBs sont très représentatifs de l'ensemble des ORBs disponibles sur le marché.

5.2.1. VisiBroker

VisiBroker est un ORB commercial de la société Inprise [Inprise98]. La distribution de charge est mise en oeuvre dans VisiBroker par l'assignation cyclique (round robin) de requêtes et par la migration d'objets. Avec la stratégie d'assignation cyclique, *les Agents Intelligents* (smart agents) acheminent les requêtes des clients aux différents serveurs de manière cyclique. La migration d'objets est utilisée pour déplacer les objets des machines surchargées aux machines ayant plus de ressources ou de puissance de traitement. Elle est aussi utilisée pour assurer la disponibilité des objets quand une machine doit être arrêtée pour effectuer des opérations de maintenance du logiciel ou du matériel.

La migration d'objets sans état est transparente aux clients de ces objets. Si un client se connecte à une implémentation d'un objet ayant migré à une autre machine, alors *l'osagent* (ORB smart agent) détectera la perte de la connexion et connectera d'une manière transparente le client au nouvel objet créé sur la machine de destination. L'osagent est un service de répertoire dynamique et distribué qui offre des facilités aussi bien pour les applications clientes que pour les implémentations d'objets. Quand un client invoque la méthode *bind* pour se connecter à un objet serveur, l'osagent localise l'implémentation de l'objet serveur et établit une connexion entre le client et cette implémentation. Les implémentations d'objets enregistrent leurs objets auprès de l'osagent pour qu'ils puissent être localisés par les applications clientes. Quand un objet est supprimé, l'osagent l'enlève de sa liste d'objets disponibles. La migration d'objets avec état est aussi possible dans VisiBroker. Cependant, elle n'est pas transparente aux applications clientes qui sont connectées à l'objet avant que sa migration n'ait commencé.

5.2.2. CORBAplus

CORBAplus est un ORB commercial de la société Expersoft [Expersoft]. CORBAplus supporte la notion de référence objet ambiguë, c'est-à-dire qu'une seule référence objet est associée à un groupe dynamique d'objets. Les objets du groupe peuvent changer dans le temps. Un message (invocation de service) est délivré à un destinataire. Trois méthodes sont offertes au programmeur pour distribuer la charge parmi les serveurs objet d'un même groupe : *byQueueSize*, *byProcessLoad*, et *byExplicitSetting*. La distribution de charge par la méthode *byProcessLoad* est recommandée quand les serveurs s'exécutent sur des architectures de machine semblables. La méthode *byQueueSize* est recommandée quand les serveurs s'exécutent sur des architectures différentes. Finalement, la méthode *byExplicitSetting* est recommandée lorsqu'il y a le besoin d'implémenter des algorithmes spécifiques de distribution de charge.

5.2.3. BEA ObjectBroker

BEA ObjectBroker est un ORB commercial de la société BEA Systems [BEA]. BEA ObjectBroker permet la distribution de charge entre différentes implémentations d'une interface ou entre des instances multiples d'une implémentation par assignation cyclique des requêtes aux serveurs. En outre, il utilise le concept de *''SmartMaps''* qui permet le contrôle de la sélection du serveur qui va traiter chacune des requêtes émises par la définition de stratégies d'association (bind) paramétrables. Ces stratégies permettent, par exemple, de choisir le serveur le plus approprié au traitement de chacune des requêtes en utilisant des règles de routage spécifiques. Les règles utilisées peuvent employer comme paramètres les données du profil utilisateur ou les variables d'appel de la requête.

Toutes ces solutions restent des solutions propriétaires car elles dépendent en grande partie de mécanismes propres à chaque ORB. Ceci nous a incités à considérer la conception d'un service de distribution de charge qui est conforme à la philosophie des services CORBA, et qui peut être déployé dans tout ORB conforme à la spécification de l'OMG. Les clients doivent hériter de certaines classes pour pouvoir obtenir leurs services requis à partir des serveurs appropriés (par exemple, les moins chargés). Les serveurs doivent hériter d'autres

classes pour pouvoir exporter leurs offres de services et participer à la distribution de charge avec d'autres serveurs offrant le même type de service. D'autre part, aucun composant central n'est nécessaire pour recueillir l'information de charge et trouver les serveurs appropriés. Ceci permet d'avoir un service évolutif capable de supporter un grand nombre de clients et de serveurs.

5.3. Services objet CORBA

La spécification standard de CORBA décrit des directives pour la conception de services objet. Les services sont conçus pour être génériques et indépendants des applications utilisateurs. Ils sont généralement structurés comme des objets décrits en termes d'interfaces écrites dans le langage de description des interfaces (IDL). Par exemple, les interfaces du canal d'événement (channel event) acceptent les données de n'importe quel type. Les clients de ce service peuvent déterminer dynamiquement les types de ces données et les traiter convenablement. Ces objets peuvent être accessibles localement ou à distance, et peuvent être implémentés comme une librairie locale ou comme des serveurs distants.

Les services CORBA sont typiquement décomposés en plusieurs interfaces distinctes qui offrent des vues différentes pour divers types de clients du service. Par exemple, le service de courtage se compose des interfaces de consultation (*Lookup*), d'enregistrement (*Register*) et d'administration (*Admin*) des offres de service. Ces interfaces peuvent être utilisées par des clients ayant des rôles différents. Un client d'un service CORBA donné peut utiliser une référence objet distincte pour chaque objet interne du service. Ces objets internes du service coopèrent afin d'offrir le service complet.

Vu que les clients d'un service ont souvent des rôles différents, ils n'ont pas besoin d'avoir accès à toutes les fonctionnalités du service. Un simple client, par exemple, n'a pas besoin d'accéder aux interfaces d'administration du service. Par conséquent, l'héritage d'interface est utilisé comme moyen permettant aux divers clients d'utiliser seulement une partie du service.

5.4. Aperçu de LSS

Notre objectif est d'avoir un service générique capable de supporter divers types de serveurs. La distribution de charge dans ce contexte a un sens si elle est appliquée aux serveurs offrant

le même type de service à leurs clients. Le but de ce service est de procurer aux programmeurs la possibilité de répartir la charge entre plusieurs serveurs offrant le même type de service, et plus particulièrement entre les instances d'un serveur répliqué. Ainsi, les demandes de service d'un client seront assignées par exemple aux serveurs légèrement chargés. L'objectif est d'avoir des temps de réponse améliorés pour les clients et une plus grande disponibilité des serveurs.

5.4.1. Architecture de LSS

La figure 5.1 présente les composantes de base de LSS. Ces composantes communiquent à travers l'ORB, et chacune d'elles est spécifiée séparément.

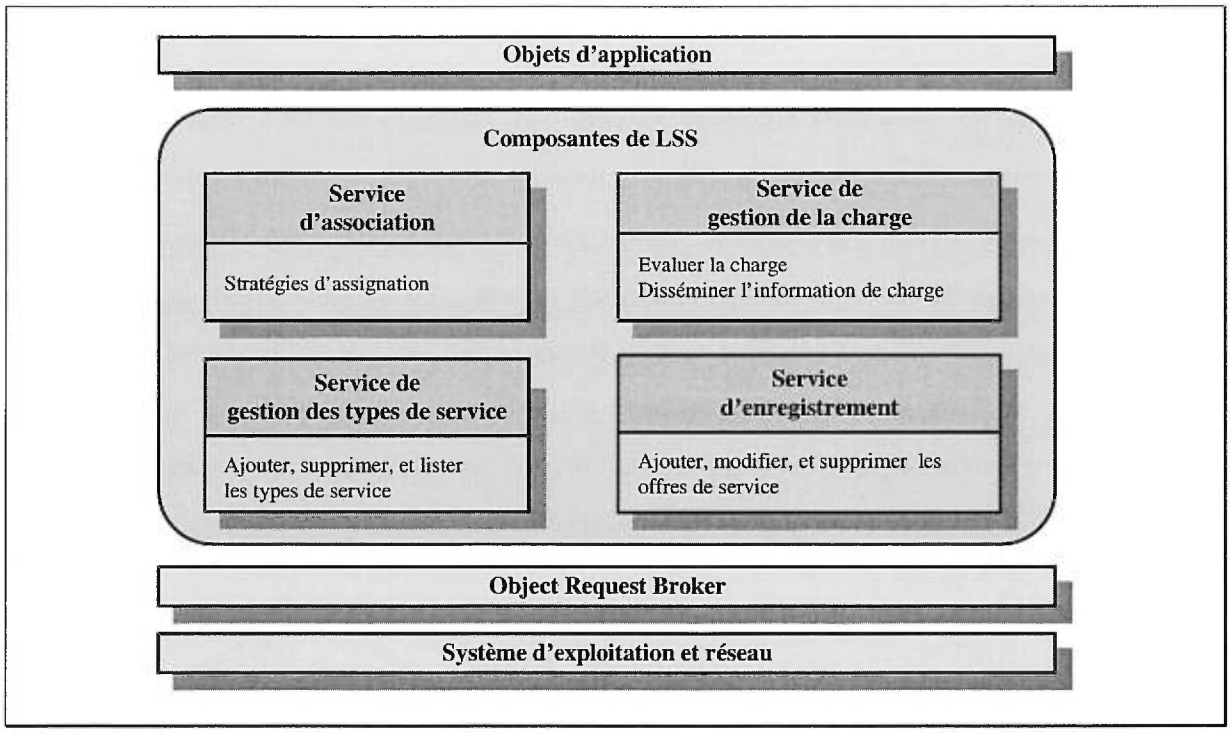


Figure 5.1 - Architecture du service LSS

Ces composantes sont :

1. *Le service de gestion des types de service (Service types management service)* : ce service permet de gérer (ajouter, modifier, supprimer et lister) les types de service qui sont offerts par les serveurs. Les serveurs offrant un même type de service forment une grappe logique.

2. *Le service d'enregistrement (Registration service)* : ce service permet d'enregistrer les offres de service des serveurs. Une offre de service décrit le type de service offert, l'interface où le service est offert, et un ensemble de propriétés décrivant le service.
3. *Le service d'association (Binding service)* : ce service met en oeuvre les stratégies d'assignation d'invocations de méthodes aux serveurs d'une même grappe.
4. *Le service de gestion de la charge (Load monitoring service)* : ce service permet d'évaluer la charge des serveurs et de disséminer l'information de charge.

Les interfaces des composantes de LSS sont décrites à la section suivante, et les interactions entre ces interfaces sont décrites à travers des scénarios d'utilisation à la section 5.6.

5.4.2. Structure du service

Nous distinguons dans le contexte de LSS les catégories d'objets suivantes : (1) les objets d'application clients, (2) les objets d'application serveurs, et (3) les objets internes du service LSS.

Les objets d'application clients sont des objets intéressés à obtenir un certain type de service. Ils ne connaissent pas les objets d'application fournissant leurs services désirés, ni leurs localisations. En utilisant le service LSS, ils pourront découvrir ces serveurs et envoyer leurs demandes de service à des serveurs appropriés.

Les objets d'application serveurs sont des objets offrant divers types de service et implémentant différentes interfaces décrites à l'aide du langage IDL. Ils dépendent du domaine d'application donné. Les serveurs offrant un même type de service forment une grappe d'objets pour lesquels la distribution de charge doit être effectuée. Pour participer au processus de distribution de charge, les serveurs doivent annoncer les types de service qu'ils offrent, et leurs charges doivent être surveillées. Ces serveurs doivent hériter de certaines interfaces génériques du service pour pouvoir répondre à ces exigences. Le service LSS présenté ici est basé sur l'utilisation du service de courtage de l'OMG pour la découverte dynamique des serveurs.

Les objets internes de LSS sont des objets qui permettent: (1) aux clients objet de découvrir les serveurs appropriés qui offrent leurs services requis, (2) aux serveurs d'exporter, de mettre à jour ou de retirer leurs offres de service, et (3) de surveiller la charge des serveurs.

La figure 5.2 montre le diagramme des classes du service LSS de distribution de charge. Ce diagramme schématise les interfaces du service et les relations entre elles. Ces interfaces offrent des vues différentes pour chaque catégorie d'utilisateurs du service LSS :

1. la vue du client, permettant de découvrir les serveurs appropriés,
2. la vue du serveur, permettant aux serveurs de prendre part au processus de distribution de charge, et
3. la vue du service, utilisée par les objets internes de LSS pour invoquer les serveurs d'application.

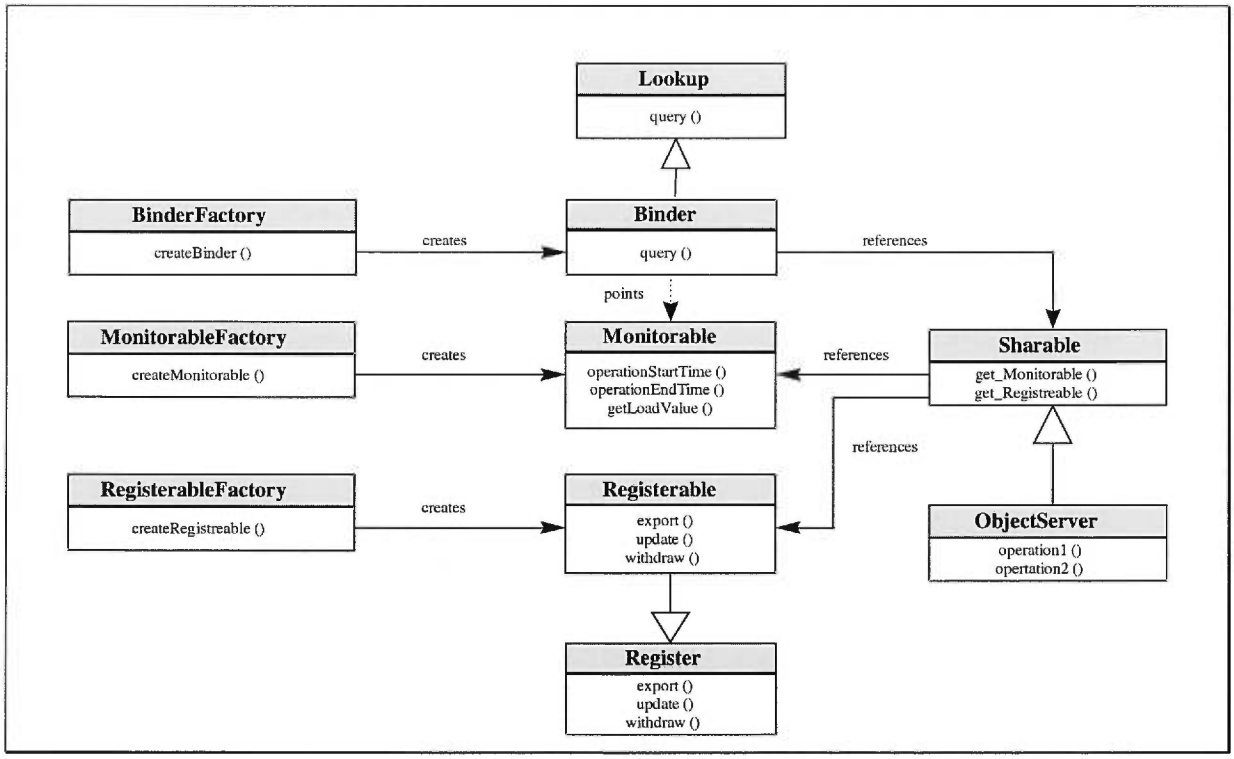


Figure 5.2 - Diagramme des classes du service LSS de distribution de charge

Le tableau 5.1 décrit les interfaces qui composent chacun des services internes de LSS.

Service	Interfaces
Service d'association	*Binder* et *BinderFactory*
Service de gestion de la charge	*Monitorable*, *MonitorableFactory*, et *Sharable*
Service d'enregistrement	*Registerable*, *RegisterableFactory*, et *Sharable*

Tableau 5.1 - Interfaces de chaque service

L'interface *Registerable* hérite de l'interface *Register* du service de courtage. L'interface *Sharable* permet aux serveurs d'avoir accès aussi bien au service de gestion de la charge qu'au service d'enregistrement. Le service de gestion des types de service n'est pas schématisé dans la figure 5.2 car nous supposons que cette gestion est faite par le service de courtage.

La figure 5.3 donne une vue simplifiée des composantes de LSS en présentant les interfaces utilisées par chaque objet du système. Une ligne solide avec un petit trait vertical est utilisée pour illustrer le fait que l'objet cible supporte l'interface nommée au-dessus de la ligne, et que les clients ayant une référence objet à cette interface peuvent appeler ses méthodes. La flèche pointe du client vers l'objet cible.

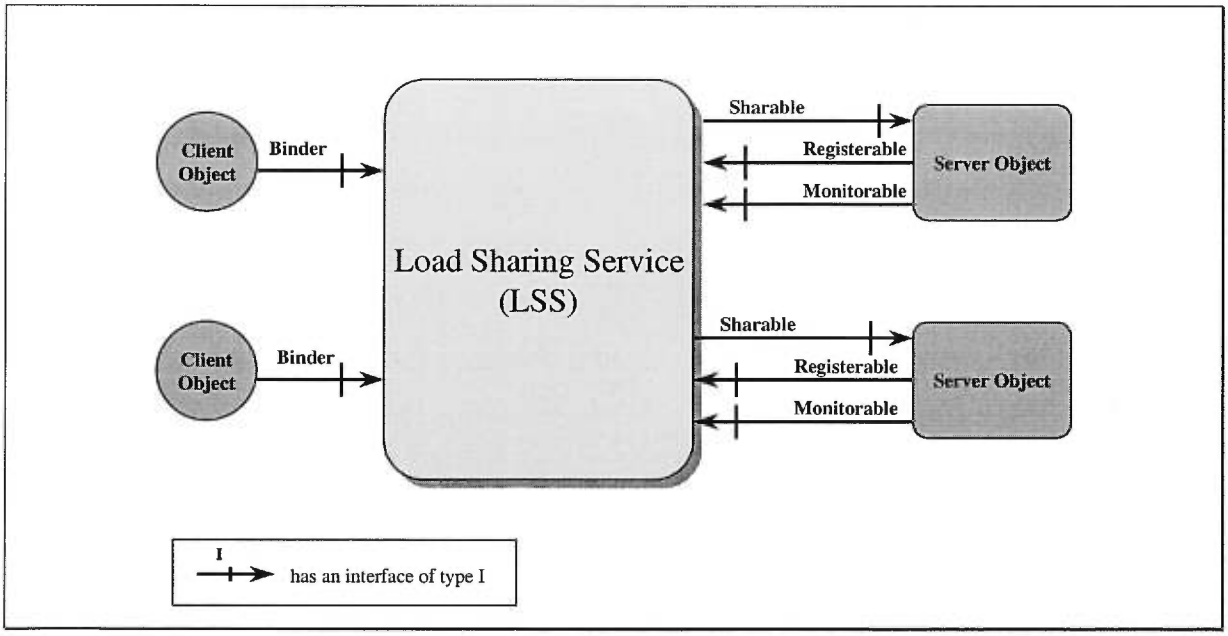


Figure 5.3 - Vue simplifiée des composantes de LSS

5.4.2.1 Vue du client

Cette vue permet à un client de dynamiquement découvrir les serveurs qui offrent son service requis, et auxquels ses requêtes peuvent être assignées. Une fois qu'un serveur cible est choisi, le client peut invoquer ses méthodes. Les serveurs sont découverts dynamiquement en utilisant l'interface Binder du service d'association dont la spécification IDL est présentée dans la figure 5.4. Un objet Binder est créé en appelant la méthode *createBinder()* d'un objet BinderFactory.

L'interface Binder hérite la méthode *query()* de l'interface *Lookup*[11] du service de courtage pour la découverte dynamique des serveurs. Les paramètres de cette méthode sont décrits dans la figure 5.4. Ils incluent le type de service requis par le client, des contraintes sur les propriétés du service, et d'autres paramètres. La méthode *query()* du Binder permet de choisir des serveurs appropriés, pour traiter les requêtes du client parmi la liste des serveurs retournés comme résultat de l'exécution de la méthode *query()* de l'interface Lookup. Ce choix s'effectue selon une certaine stratégie d'assignation d'invocations de méthodes. L'interface Binder peut être spécialisée pour mettre en oeuvre diverses stratégies d'assignation. Des exemples de telles stratégies sont présentés à la section 5.5.

```
//LSS.idl
module LoadSharing {
....
        interface Binder: CosTrading::Lookup {
        }

        interface BinderFactory {
            Binder createBinder();
        }
...
}
```

```
// CosTrading.idl
void query (
        in ServiceTypeName type, in Constraint constr,
        in Preference pref, in PolicySeq policies,
        in SpecifiedProps desired_props,
        in unsigned long how_many,
        out OfferSeq offers,
        out OfferIterator offer_itr,
        out PolicyNameSeq limits_applied
) raises (
        IllegalServiceType,   UnknownServiceType,
        IllegalConstraint,    IllegalPreference,
        IllegalPolicyName,    PolicyTypeMismatch,
        InvalidPolicyValue,   IllegalPropertyName,
        DuplicatePropertyName, DuplicatePolicyName
);
```

Figure 5.4 - Interfaces IDL du service d'association

[11] La spécification IDL de l'interface *Lookup* du service de courtage est décrite dans la spécification "CosTrading.idl" de l'OMG.

5.4.2.2 Vue du serveur

Cette vue permet à un serveur de prendre part au processus de distribution de charge. Ceci est réalisé en deux étapes. La première étape correspond à l'enregistrement du serveur auprès du service de courtage par l'exportation de son offre de service. La seconde étape correspond à la surveillance de la charge du serveur. La figure 5.5 présente les interfaces des composantes du service d'enregistrement et du service de gestion de la charge.

Pour annoncer son offre de service, un serveur doit créer un objet Registerable en appelant la méthode *createRegisterable()* d'un objet RegisterableFactory. L'interface Registerable hérite les méthodes suivantes de l'interface Register[12] du service de courtage : *export(), withdraw(),* et *update()*. Ces méthodes sont respectivement utilisées pour exporter, retirer, et mettre à jour une offre de service.

```
//LSS.idl
module LoadSharing {
    ....
    interface Monitorable{
            void operationStartTime(object targetObject, operation
            oper)
            void operationEndTime(object targetObject, operation
            oper)
            double getLoadValue()
    }

    interface MonitorableFactory {
            Monitorable createMonitorable();
    }

    interface Registerable: CosTrading::Register {
    }

    interface RegisterableFactory {
                Registerable createRegisterable();
    }
    ...
}
```

```
//CosTrading.idl

struct Property {
                PropertyName name;
                PropertyValue value;
};
typedef sequence<Property> PropertySeq;
struct Offer {
                Object reference;
                PropertySeq properties;
};

OfferId  export (in Object reference,in ServiceTypeName type,
                in PropertySeq properties
) raises ( InvalidObjectRef, IllegalServiceType,
        UnknownServiceType,InterfaceTypeMismatch,
        IllegalPropertyName, PropertyTypeMismatch,
        ReadonlyDynamicProperty,
        MissingMandatoryProperty,
        DuplicatePropertyName
);
```

Figure 5.5 - Interfaces IDL des services d'enregistrement et de gestion de la charge

La surveillance de la charge d'un serveur exige qu'il crée un objet Monitorable en appelant la méthode *createMonitorable()* d'un objet MonitorableFactory. Un objet Monitorable fournit trois méthodes : (1) l'opération *operationStartTime()* est utilisée par un serveur pour enregistrer l'arrivée d'une invocation sur un de ses objets, (2) l'opération *operationEndTime()*

[12] La spécification IDL de l'interface *Register* du service de courtage est décrite dans la spécification ''CosTrading.idl'' de l'OMG.

est utilisée par un serveur pour enregistrer la fin de traitement d'une requête, et (3) l'opération *getLoadValue()* est utilisée par un objet Binder pour obtenir la valeur de la charge du serveur qui est associé à un objet Monitorable. L'objet Monitorable maintient l'historique des demandes de service faites au serveur qui lui est associé.

5.4.2.3 Vue des objets du service

Cette vue est définie par l'interface Sharable qui doit être supportée par les serveurs. La figure 5.6 décrit la spécification IDL de l'interface Sharable. Cette interface définit deux attributs, qui sont de type Monitorable et Registerable respectivement. Ces deux attributs permettent d'avoir les références objet aux objets Monitorable et Registerable associés à un serveur donné. La référence de l'objet Monitorable est utilisée pour obtenir la valeur actuelle de la charge du serveur en question. La référence de l'objet Registerable est utilisée pour annoncer, modifier, ou supprimer l'offre de service du serveur.

```
//LSS.idl
module LoadSharing {
        ....
        interface Sharable {
                attribute readonly Monitorable monitorable ;
                attribute readonly Registerable registerable ;
        }
        ...
}
```

Figure 5.6 - Interface IDL Sharable

5.5. Stratégies d'assignation de requêtes

Comme dans l'architecture LoDACE, le Binder qui joue le rôle d'interface entre le client et les serveurs peut mettre en oeuvre diverses stratégies d'assignation de requêtes pouvant être aussi bien statiques que dynamiques tenant compte par exemple de la charge des serveurs.

Les stratégies d'assignation sont réalisées par des Binders spécialisés. Le Binder de base définit la méthode *query()* qui sert à découvrir dynamiquement les serveurs capables d'offrir le type de service requis par le client. En plus des stratégies décrites à la section 4.3.2, deux autres stratégies (*LRU* et *RLRU*) peuvent être utilisées.

Stratégie de sélection du serveur le moins récemment utilisé (*last recently used - LRU*)

Cette stratégie consiste à choisir parmi les serveurs éligibles le serveur qui est le moins récemment utilisé. Cette stratégie est dynamique, mais elle ne tient pas compte de la vraie charge imposée par chaque type de requête.

Stratégie LRU relative (*relative LRU – RLRU*)

Cette stratégie est une version modifiée de la stratégie précédente. Nous l'appelons LRU relative car elle tient compte du taux de service de chacun des serveurs. Nous supposons qu'un poids w_i est associé à chaque serveur S_i qui est proportionnel à son taux de service. Ce poids est considéré comme une propriété de l'offre de service de S_i. Plus le taux de service est grand, plus le poids associé au serveur est grand. Nous définissons le taux d'occupation du serveur S_i pendant une unité de temps comme étant le rapport entre le nombre de requêtes N_i ayant été traitées par S_i pendant cette unité de temps et le poids du serveur :

$$Taux\ d'occupation\ de\ S_i = N_i / w_i$$

Ainsi, le serveur choisi est celui qui présente le plus petit taux d'occupation.

Cette stratégie ne tient pas compte de la charge imposée par chaque type de requête. Ainsi, il est possible de l'améliorer en associant à chaque type de requête R_j un poids r_j proportionnel à la charge qu'elle impose au serveur. Cette charge peut être exprimée par exemple en termes de besoins en temps CPU de la requête ou en termes de nombres d'instructions exécutées lors du traitement de la requête. Nous définissons également $n_{i,j}$ comme étant le nombre de requêtes du type R_j ayant été traitées par le serveur S_i pendant la dernière unité de temps. Par conséquent, le taux d'occupation du serveur S_i devient :

$$Taux\ d'occupation\ de\ S_i = [n_{i,1}r_1 + n_{i,2}r_2 + \ldots + n_{i,m}r_m\ ] / w_i$$

5.6. Scénarios d'utilisation de LSS

Dans cette section, nous présentons deux scénarios pour la stratégie LL de sélection du serveur le moins chargé, un pour le côté client et l'autre pour le côté serveur, tout en montrant comment les interfaces du service de distribution de charge interagissent.

5.6.1. Scénario côté client

La figure 5.7 montre comment un client utilise le service de distribution de charge afin de trouver dynamiquement un serveur approprié qui fournit son service requis et qui est légèrement chargé.

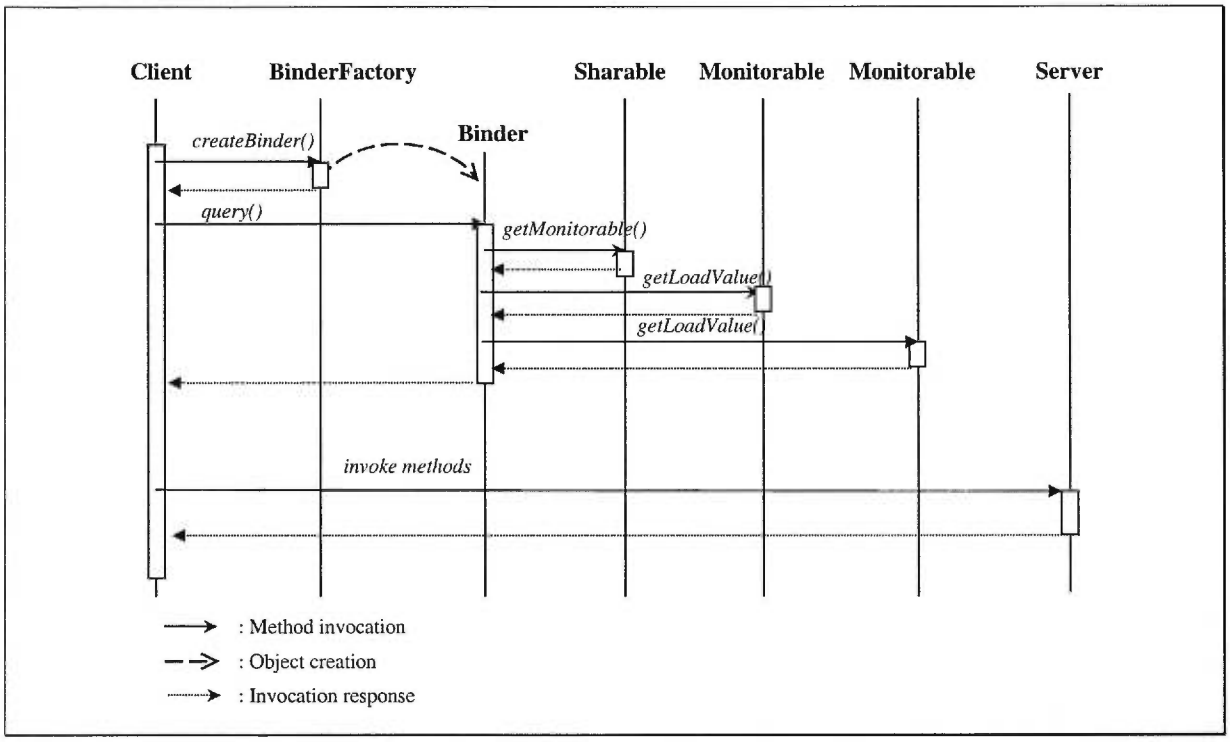


Figure 5.7 - Scénario d'utilisation - côté client

Les étapes sont comme suit :

1. Le client crée un objet Binder en appelant la méthode *createBinder()* d'une interface BinderFactory.
2. Le client formule sa demande en indiquant le type de service requis, ses contraintes et ses préférences, et puis, il appelle la méthode *query()* de l'objet Binder créé à l'étape 1. Le résultat est une liste de serveurs fournissant le service requis. Cette liste est triée suivant la charge actuelle des serveurs. Elle est obtenue en effectuant les étapes 3, 4, et 5.
3. La demande est envoyée à l'interface Lookup du service de courtage en appelant sa méthode *query()*.

4. Pour chaque offre de service de la liste obtenue à l'étape 3, le Binder appelle la méthode *get_monitorable()* du serveur correspondant afin d'obtenir la référence objet de l'objet Monitorable associé qui surveille sa charge. La référence objet de l'offre de service référence un objet Sharable (ou un serveur qui hérite de l'interface Sharable).

5. Le Binder appelle la méthode *getLoadValue()* sur les objets Monitorable obtenus à l'étape 4 afin d'obtenir la charge de chaque serveur offrant le service requis et découvert à l'étape 2.

6. Le client sélectionne le serveur le moins chargé et appelle ses méthodes pour obtenir le service s'il a un talon (stub) pour l'interface du serveur. Autrement, il doit utiliser le service d'invocation dynamique de CORBA.

5.6.2. Scénario côté serveur

La figure 5.8 montre comment un serveur annonce son offre de service et comment sa charge est surveillée. Un serveur doit hériter de l'interface Sharable comme il a été montré à la figure 5.2.

Les étapes sont comme suit :

1. L'objet Sharable (ou un serveur héritant de l'interface Sharable) crée un objet Monitorable, en appelant la méthode *createMonitorable()* d'un objet MonitorableFactory, afin de surveiller sa charge.

2. L'objet Sharable crée un objet Registerable, en appelant la méthode *createRegisterable()* d'un objet RegisterableFactory, pour permettre au serveur d'annoncer son offre de service.

3. Le serveur exporte son offre de service en appelant la méthode *export()* de l'objet Registerable créé à l'étape 2. Cette demande est envoyée à l'interface Register du service de courtage.

4. Une fois que les méthodes du serveur sont appelées par des clients, l'objet Monitorable associé, qui est créé à l'étape 1, est avisé de l'arrivée d'une nouvelle requête au serveur en appelant la méthode *operationStartTime()* et de la fin du traitement de la requête en appelant la méthode *operationEndTime()*. Ces deux exécutions permettent de calculer le temps d'occupation du serveur pendant une unité de temps, et par conséquent sa charge.

5. Si le serveur souhaite modifier son offre de service, il appelle la méthode *update()* de l'objet Registerable créé à l'étape 3.

6. Si le serveur souhaite retirer son offre de service, il appelle la méthode *withdraw()* de l'objet Registerable créé à l'étape 3.

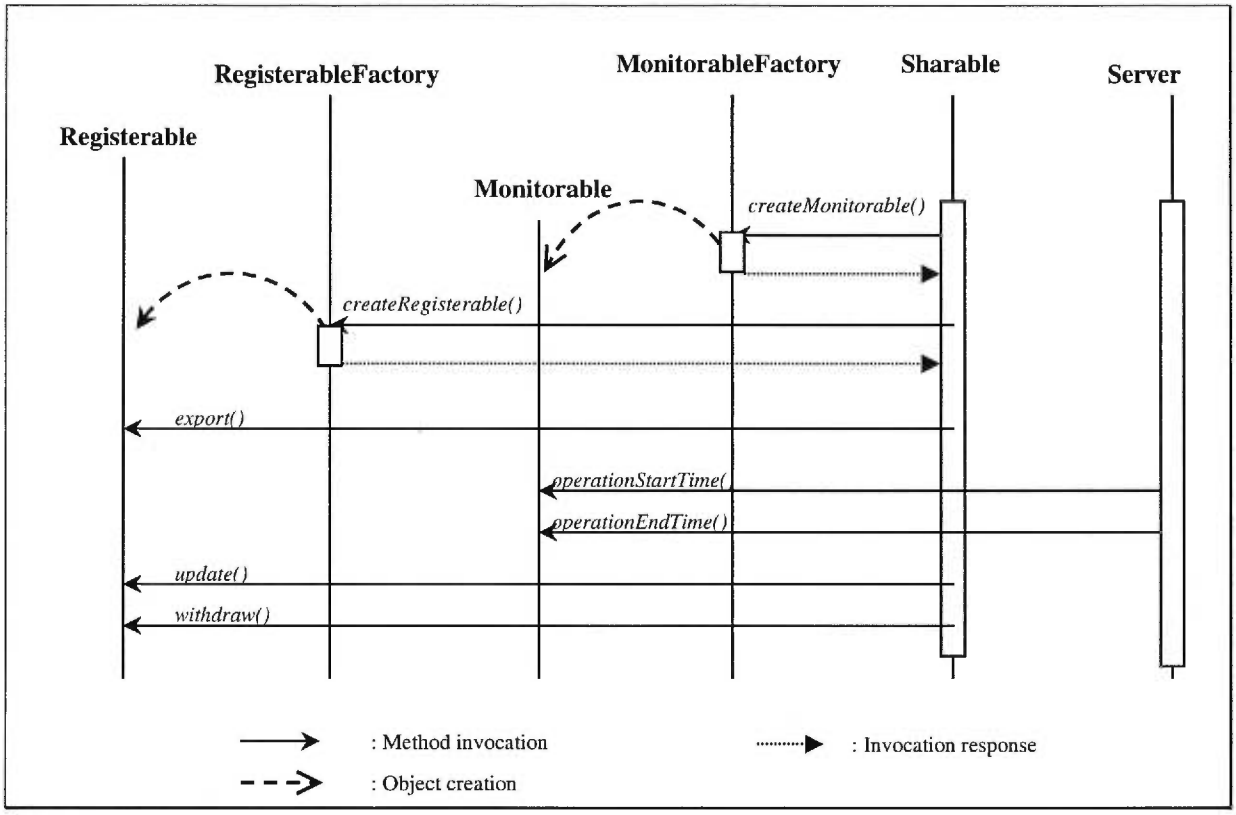


Figure 5.8 - Scénario d'utilisation - côté serveur

5.7. Implémentation

Un prototype du service LSS a été réalisé dans le même environnement que celui du prototype de l'architecture LoDACE décrit à la section 4.4.

Pour les besoins d'expérimentation de LSS que nous décrivons au chapitre 6, nous avons implémenté trois Binders correspondants respectivement aux stratégies : aléatoire (RD), cyclique (RR), et le moins chargé (LL). Comme dans LoDACE, nous avons choisi comme indicateur de charge le taux d'utilisation du serveur. Le taux d'utilisation du serveur est calculé suivant la procédure décrite à la section 4.4.2.1.

5.8. Conclusion

Ce chapitre a présenté une nouvelle approche pour la distribution de charge dans les systèmes distribués objet CORBA. Cette approche consiste à concevoir un service de distribution de charge conforme aux principes des services CORBA.

Le service LSS est structuré en quatre services internes : service d'association, service d'enregistrement, service de gestion de la charge, et service de gestion des types de service. Le service d'association joue le rôle d'interface entre les clients et les serveurs et réalise diverses stratégies d'assignation de requêtes. Le service d'enregistrement permet d'enregistrer les offres de service des serveurs afin d'être découverts dynamiquement par les objets client. Le service de gestion de la charge a pour rôle d'évaluer la charge des serveurs et de disséminer l'information de charge. Le service de gestion des types de service gère les types de services en effectuant les opérations d'ajout, de modification, de suppression, et d'édition.

Trois vues sont offertes par le service LSS : la vue des clients, la vue des serveurs objet, et la vue des objets internes du service. La vue du client offre au client le moyen de découvrir les serveurs appropriés. La vue du serveur offre au serveur le moyen d'annoncer son offre de service et de prendre part à la distribution de charge. La vue des objets internes du service permet l'accès aux serveurs et aux objets qui leur sont associés.

L'utilisation du service LSS est illustrée par deux scénarios, l'un pour le côté client et l'autre pour le côté serveur. Le prototype de ce service a servi dans l'expérimentation des stratégies d'assignation de requêtes présentées à la section 5.5. Cette expérimentation est décrite au chapitre 6.

Chapitre 6
Expérimentation et validation

Ce chapitre décrit les expérimentations que nous avons conduites avec le prototype de l'architecture LoDACE et le prototype du service LSS décrits respectivement aux chapitres 4 et 5.

6.1. Introduction

Le chapitre 4 a présenté l'architecture LoDACE constituant une plate-forme pour la mise en oeuvre de diverses stratégies d'assignation de requêtes dans le but d'effectuer la distribution de charge entre un ensemble de serveurs offrant un même type de service. Le chapitre 5 a présenté le service LSS de distribution de charge qui est conforme à la philosophie des services standards CORBA. Ce service est basé également sur l'assignation de requêtes aux serveurs appropriés.

Le présent chapitre décrit les expérimentations que nous avons effectuées avec les prototypes de LoDACE et de LSS. L'objectif de ces expérimentations est d'évaluer les performances des stratégies d'assignation décrites aux chapitres 4 et 5 et d'étudier l'évolution de la charge des serveurs en présence d'une demande de service croissante. Les stratégies comparées sont : (1) l'assignation aléatoire, (2) l'assignation cyclique, (3) l'assignation basée sur la sélection du serveur le moins chargé, et (4) l'assignation basée sur les choix volontaires des. L'indicateur de performance considéré est *le temps de réponse moyen*.

Une application distribuée comportant un certain nombre de serveurs, qui implémentent une même interface, et de clients est réalisée pour mener les expérimentations. Les requêtes des clients sont assignées aux serveurs suivant une des quatre stratégies d'assignation. La réalisation d'une série d'expérimentations requiert : (1) la génération de requêtes par les clients suivant une certaine distribution, (2) la collecte des données résultantes de

l'assignation de requêtes aux serveurs, telles que la charge des serveurs et le temps de réponse des requêtes émises, et (3) l'agrégation de ces données pour pouvoir tirer des conclusions.

6.2. Application de test

Comme application distribuée typique, nous avons considéré une application de gestion des comptes bancaire comportant un certain nombre de clients et de serveurs. Nous avons défini un simple type de service de manipulation d'un compte bancaire. La figure 6.1 donne la spécification IDL de ce service dont les opérations sont : le dépôt, le retrait, et la consultation de la balance.

```
// IDL
module Bank {
    interface Account    {
        readonly attribute double balance  ;

        void makeDeposit(in double sum);
        void makeWithdrawal(in double sum, out double newBalancce);
    };

    // Account Factory specification
    interface AccountFactory {
      Account createACserver(in string host);
    };
};
```

Figure 6.1 - Interface IDL d'un serveur de gestion de compte bancaire

Un client d'un serveur qui implémente l'interface *Account* pourra effectuer des opérations de retrait et de dépôt, et consulter la balance du compte. Nous avons réalisé un serveur et un client de l'interface *Account*, qui sont répliqués selon les besoins de l'expérimentation. Le serveur dispose d'un filtre de processus qui permet de capter les appels des méthodes des objets du serveur. Ceci permet d'évaluer la charge du serveur suivant la procédure que nous avons décrit à la section 4.4.2.1 concernant la surveillance de la charge des serveurs.

6.3. Plan d'expérimentation

Le plan d'expérimentation consiste à :

1. Définir une configuration des serveurs qui implémentent l'interface *Account*.

2. Définir des situations correspondantes à la charge de travail à laquelle seraient soumis les serveurs de la configuration considérée. Une situation correspond à un certain nombre de clients qui émettent des requêtes aux serveurs suivant une distribution donnée et de manière presque simultanée.
3. Expérimenter les stratégies d'assignation de requêtes *aléatoire* (random – RD) et *cyclique* (round robin – RR) qui ne sont basées sur aucune information d'état dynamique des serveurs.
4. Expérimenter la stratégie d'assignation de requêtes basée sur la sélection du serveur *le moins chargé* (least-loaded – LL).
5. Expérimenter l'approche d'assignation *orientée client* (client-oriented – CO) basée sur les choix volontaires des clients sans distribution explicite de la charge.

Rappelons que les prototypes de LoDACE et de LSS sont réalisés dans un environnement OrbixWeb en utilisant le langage Java et le service de courtage OrbixTrader. Ces deux prototypes sont déployés sur un réseau local Ethernet 10 Mbits/s avec deux postes de travail Sun Ultra-1 à 167 Mhz et 128 MB de mémoire chacun, et fonctionnant sous Solaris 2.5.

Notons que les résultats de l'expérimentation dépendent de la puissance des machines utilisées (vitesse des processeurs, mémoire disponible, etc.) et de la nature du réseau. En effet, comme nous l'avons expliqué à la section 3.2.2, le temps de réponse d'une requête comprend trois composantes : 1) le temps de transfert des paramètres de la requête ($T_{trans(m)}$), 2) le temps de traitement de la requête ($T_{proc(m)}$), et 3) le temps de transfert des résultats ($T_{res(m)}$). $T_{trans(m)}$ et $T_{res(m)}$ dépendent de la vitesse et de la charge du réseau sous-jacent. $T_{proc(m)}$ dépend de la puissance et de la charge des machines (attribuable à d'autres processus).

6.4. Expérimentation avec LoDACE

Dans cette section nous décrivons l'expérimentation que nous avons réalisée avec le prototype de l'architecture LoDACE décrit au chapitre 4.

6.4.1. Configuration

Le tableau 6.1 présente la configuration utilisée pour conduire une série d'expérimentations. Pour avoir une évaluation uniforme des quatre stratégies d'assignation de requêtes considérées, nous considérons que tous les clients envoient la même requête aux serveurs. Cette requête concerne la consultation de la balance du compte géré par le serveur. Cette requête est simple car elle ne fait que lire le contenu d'un attribut du serveur. Dans une application réelle, un serveur peut recevoir des requêtes différentes que nous avons assimilées au chapitre 3 à des classes de service. Nous supposons également que tous les clients génèrent des requêtes suivant une même distribution qui est ici une loi de Poisson.

Nombre de serveurs	4 (serveur1, serveur2, serveur3, et serveur4)
Situations considérées (en nombre de clients) numérotées de 1 à 7	5, 10, 15, 20, 25, 30, et 35.
Loi de génération des requêtes	Loi de Poisson de moyenne $\lambda = 5$
Type de requêtes	Consultation de la balance (*get_balance()*)
Indicateur de performance	Temps de réponse moyen
Indicateur de charge d'un serveur	Taux d'utilisation du serveur pendant une période de temps donnée.
Indicateur de charge d'une station de travail	Nombre moyen de tâches dans la file d'exécution de la CPU (retourné par la commande unix uptime)
Stratégies d'assignation considérées dans les expérimentations.	**RD** : aléatoire (random) **RR** : cyclique (round robin) **LL** : le moins chargé (least-loaded) **CO** : orientée client (client-oriented).
Durée d'envoi de requêtes par un client	1 minute

Tableau 6.1 - Configuration utilisée pour l'expérimentation de LoDACE

Nous avons adopté dans cette expérimentation l'approche orientée répartiteur pour l'assignation de requêtes dans laquelle le Binder joue le rôle de répartiteur. Des Binders spécialisés sont définis pour réaliser les stratégies d'assignation considérées. La figure 6.2 montre le processus de spécialisation du Binder. Le Binder de base hérite de l'interface *Lookup* du service de courtage. *RD_Binder*, *RR_Binder*, et *LL_Binder* héritent du Binder de

base et mettent en oeuvre respectivement l'assignation aléatoire, l'assignation cyclique, et l'assignation au serveur le moins chargé.

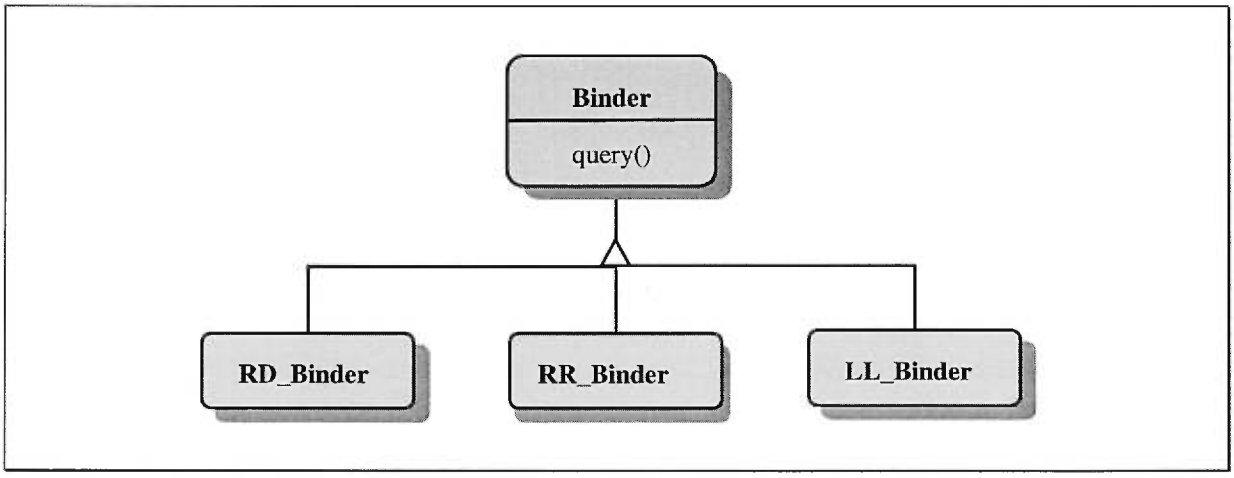


Figure 6.2 - Spécialisation du Binder

6.4.2. Génération de la charge de travail

La génération de la charge de travail est réalisée par l'activation d'un certain nombre de clients, correspondant à la situation en cours d'expérimentation. Ces clients génèrent les requêtes suivant une loi de Poisson de moyenne 5. L'activation des clients est réalisée à l'aide d'un script shell. Le nombre de clients à activer est donné comme paramètre de la ligne de commande du script shell. L'implémentation de la loi de Poisson dans le langage de programmation Java permet de déterminer le nombre d'événements qui vont se produire pendant la durée d'expérimentation du client (1 minute). Ce nombre d'événements correspond au nombre de requêtes que le client devra générer lors de l'expérimentation. Les requêtes sont générées par le client à chaque intervalle Δ défini par :

$$\Delta = \textit{unité de temps / nombre d'événements}$$

La figure 6.3 illustre la procédure de génération d'événements conformément à la loi de Poisson en utilisant le package *sdsu.algorithms.data* de la librairie Java de la SDSU (San Diego State University Java Library). La figure 6.4 représente l'évolution du nombre de requêtes générées en fonction du nombre de clients.

```
import sdsu.algorithms.data.*;
...
int ut = 60000; //durée d'expérimentation = 1min
int limbda = 5; //mean
Poisson p = null;
long nevents, delta;
try {
      P = new Poisson(limbda);
      nevents = (long) p.nextElement();
      delta = ut/nevents;
} catch (OutofBoundsException e) {
      System.out.println("out of bound exception "+ e.toString());
}
```

Figure 6.3 - Génération du nombre d'événements

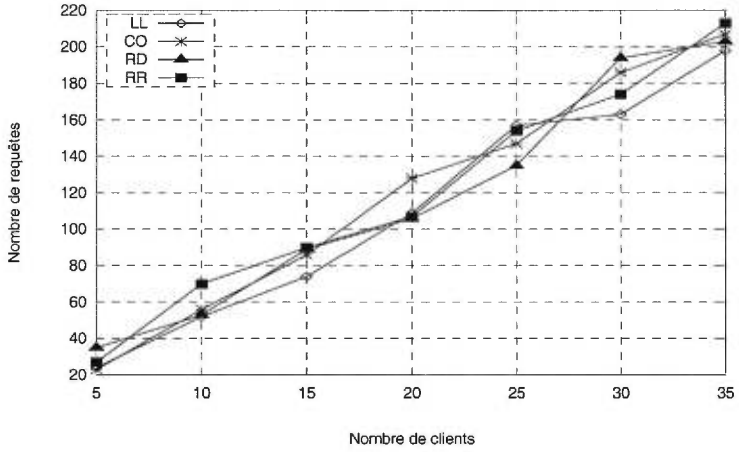


Figure 6.4 - Évolution du nombre total de requêtes générées

6.4.3. Collecte des résultats

L'état d'exécution de chaque client est sauvegardé dans un fichier. Ce fichier garde les informations concernant le nombre de requêtes émises (nombre d'événements générés par loi de Poisson), le serveur auquel chaque requête a été assignée, et le temps de réponse associé. La dernière ligne de ce fichier contient le temps de réponse moyen (TRM) de l'ensemble des requêtes émises par le client. Ce fichier permet de constituer un tableau qui récapitule pour chaque requête émise par le client le temps de réponse associé et le serveur auquel elle a été assignée. La figure 6.5 illustre un exemple de ce fichier.

```
Binder reference : IOR[type="IDL:Binding/Binder:1.0" IOPProfile[IIOP1.0 host=garibaldi port=1570
:\garibaldi:BDSrv:::IFR:Binding_Binder ]]
Number of Events = 4
delta = 15000
[ New IIOP Connection (garibaldi,IT_daemon, null,,pid=0) ]
[ New IIOP Connection (132.218.1.216,BDSrv, null,,pid=0) ]
Received 4 offers...
Host : garibaldi
Server : server4
[ New IIOP Connection (132.218.1.216,AccountSrv3, null,,pid=0) ]
Event 0 Elapsed time in Milli-seconds: 21

Received 4 offers...
Host : garibaldi
Server : server1
[ New IIOP Connection (132.218.1.216,AccountSrv, null,,pid=0) ]
Event 1 Elapsed time in Milli-seconds: 18
...
Average response time in milli-seconds: 19.0
```

Figure 6.5 - Exemple d'un fichier de sortie d'un client

Pour expérimenter différentes charges de travail nous avons considéré, pour chacune des stratégies d'assignation, sept situations qui correspondent respectivement à 5, 10, 15, 20, 25, 30, et 35 clients.

6.4.4. Résultats des expérimentations

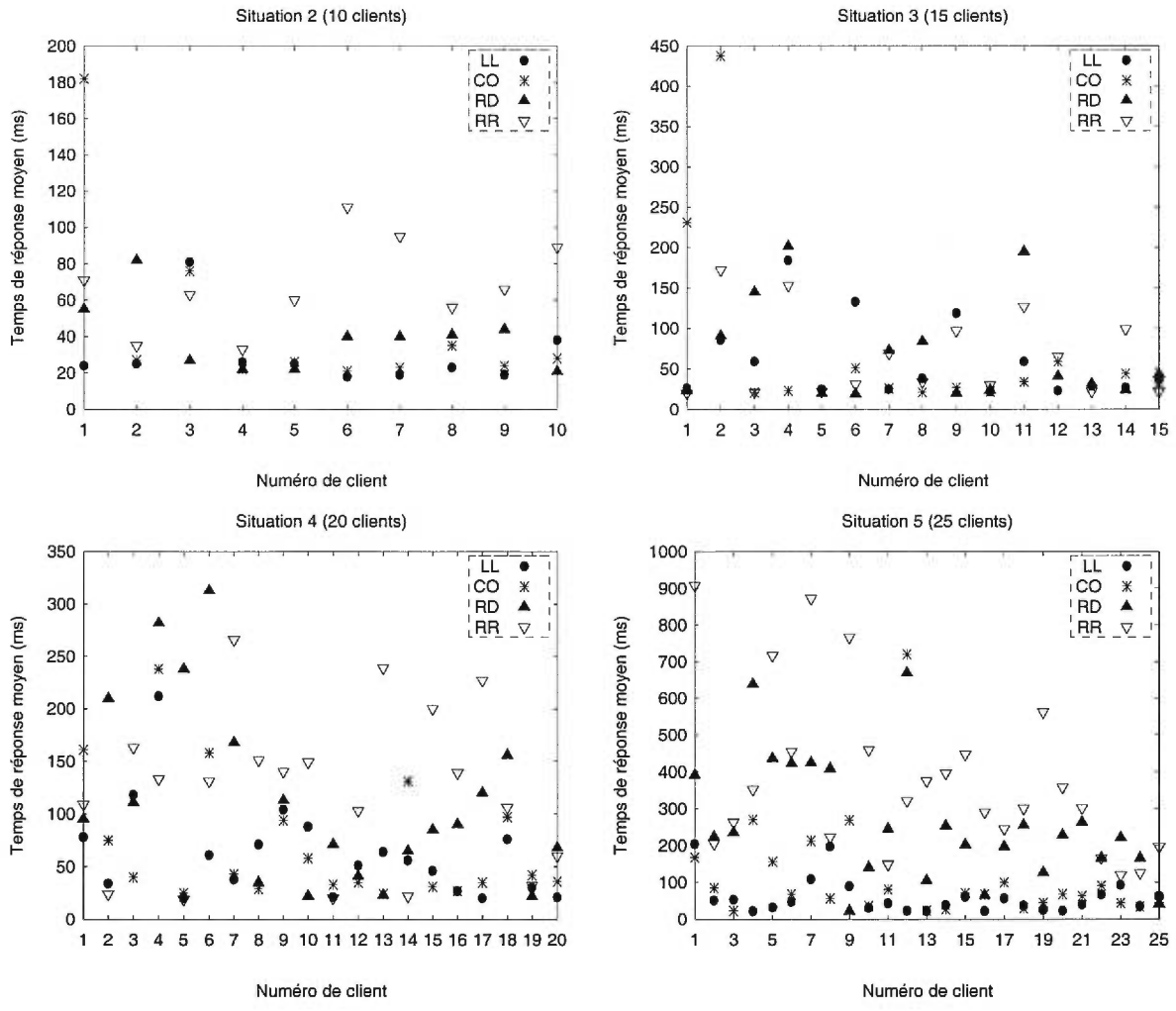
Pour chacune des situations et pour chacune des stratégies d'assignation considérées, le résultat de l'exécution de chaque client est enregistré dans un fichier. Vu le grand nombre de fichiers de sortie, nous avons procédé à une agrégation des résultats par situation comme dans le tableau 6.2 (situation 2 avec 10 clients) qui est schématisé graphiquement par la figure 6.6 (10 clients).

Notons que chaque client est identifié par un numéro. Le temps de réponse d'un client (TRM), exprimé en milli-secondes, correspond à la moyenne des temps de réponse de l'ensemble des requêtes émises par ce client pour la stratégie considérée. Étant donné que la génération de requêtes par chaque client suit une loi de Poisson, le nombre de requêtes générées par un même client n'est pas forcément le même pour chacune des stratégies considérées (RD, RR, LL, et CO).

Numéro client	RD		RR		LL		CO	
	Nbre. requêtes	TRM (ms)	Nbre. requêtes	TRM (ms)	Nbre. requêtes	TRM (ms)	Nbre. requêtes	TRM (ms)
1	4	55	10	71	7	24	4	182
2	6	82	4	35	4	25	7	27
3	4	27	6	63	7	81	2	76
4	2	22	7	33	7	26	7	22
5	4	22	8	60	4	25	9	26
6	10	40	6	111	3	18	4	21
7	4	40	8	95	5	19	10	23
8	8	41	9	56	6	23	5	35
9	5	44	6	66	3	19	5	24
10	6	21	6	89	6	38	3	28
Total et TRM	53	39,4	70	67,9	52	29,8	56	46,4

Tableau 6.2 - Résultats obtenus dans la situation 2 (10 clients)

La figure 6.6 montre le temps de réponse moyen des clients dans six situations et ceci pour chacune des quatre stratégies d'assignation considérées.



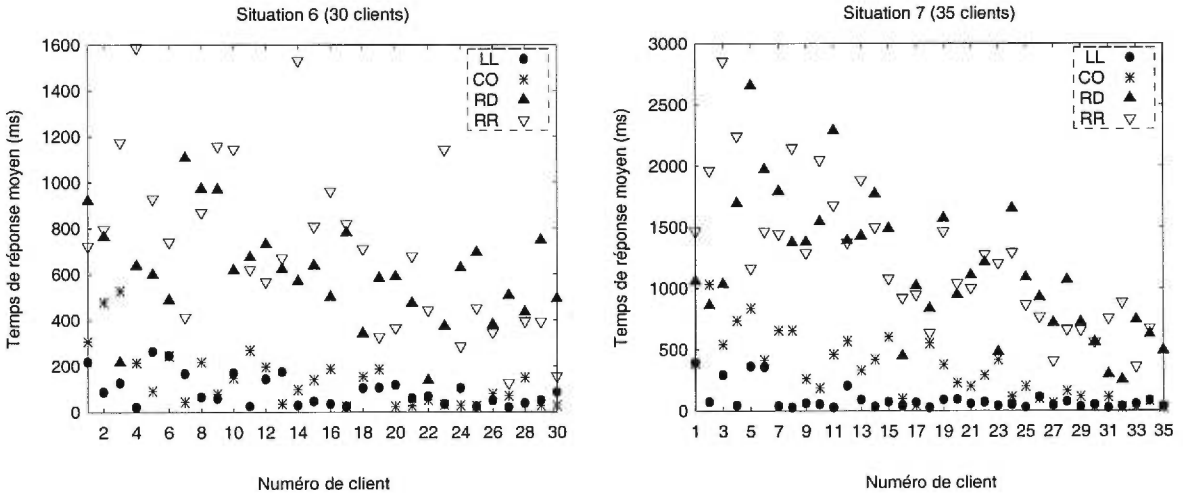


Figure 6.6 - Temps de réponse moyen de chacun des clients dans les différentes situations

L'ensemble des résultats obtenus dans l'expérimentation a été agrégé dans le tableau 6.3. Chaque ligne de ce tableau décrit le résultat global d'une situation donnée en termes de nombre total de requêtes émises, du temps de réponse moyen correspondant, et de la déviation standard par rapport à la moyenne, et ceci pour chacune des stratégies d'assignation. Les données de ce tableau sont représentées graphiquement par la figure 6.7 qui montre l'évolution du temps de réponse moyen en fonction du nombre de requêtes et par la figure 6.8 qui représente l'évolution de la déviation standard du temps de réponse moyen en fonction du nombre de clients.

Nbre. de clients	LL			RD			RR			CO		
	Nbr. de req.	TRM (ms)	Dév. Std.	Nbr. de req.	TRM (ms)	Dév. Std.	Nbr. de req.	TRM (ms)	Dév. Std.	Nbr. de req.	TRM (ms)	Dév. Std.
5	24	26.4	17,85	35	33.6	26,17	27	31.2	8,82	23	33.8	25,88
10	52	29.8	18,80	53	39.4	24,90	70	67.9	18,86	56	46.4	50,35
15	74	59.47	63,45	89	69.07	52,15	90	65.6	49,53	86	71.67	114,23
20	109	61.8	89,60	106	116.4	75,72	107	121.65	45,61	128	70.6	58,95
25	157	59.36	166,23	135	261.92	209,57	154	382.8	48,52	147	114.16	145,05
30	163	92.5	216,99	194	607.1	375,53	174	711.9	69,23	186	140.03	129,34
35	198	92.83	560,85	203	1157.57	573,13	213	1235.44	100,89	207	325.11	261,12

Tableau 6.3 - Résultats de chaque situation avec les quatre stratégies d'assignation

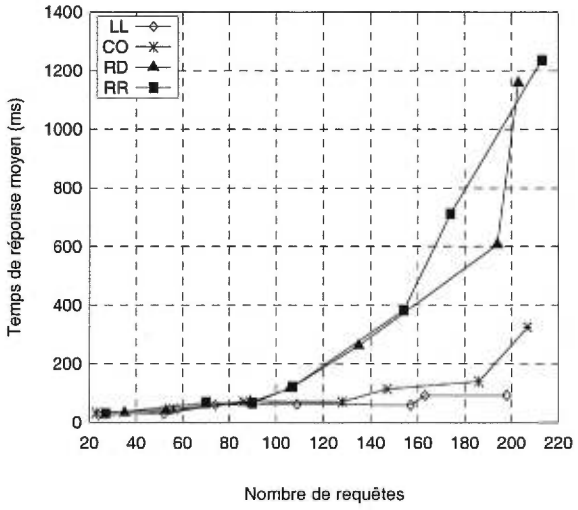


Figure 6.7 - Évolution du temps de réponse moyen

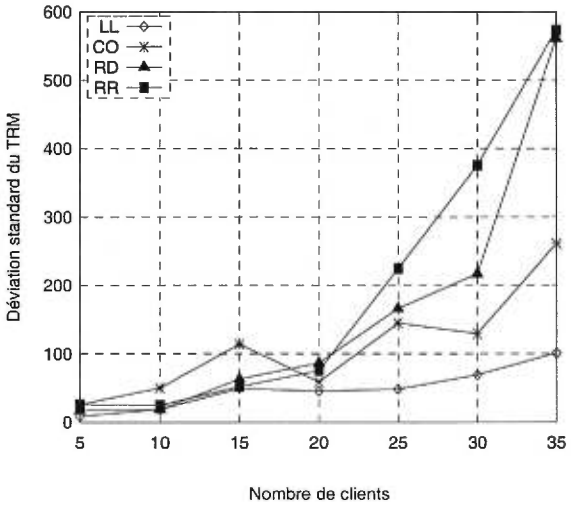
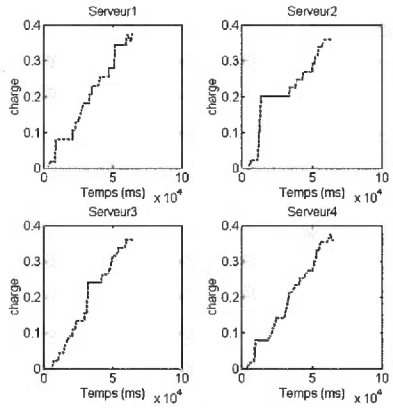
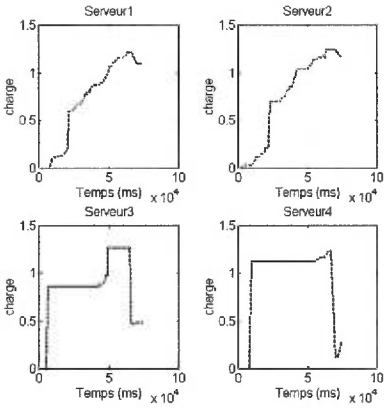


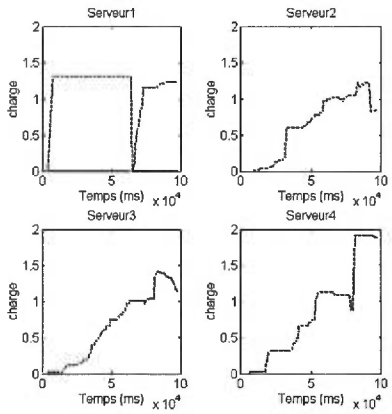
Figure 6.8 - Évolution de la déviation standard du temps de réponse moyen



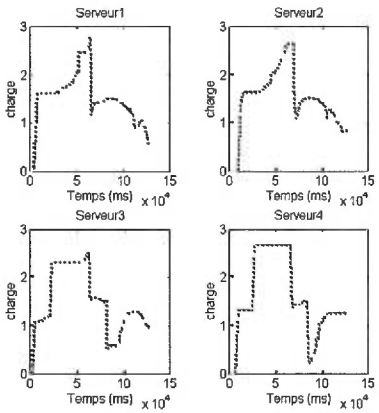
(10 clients)



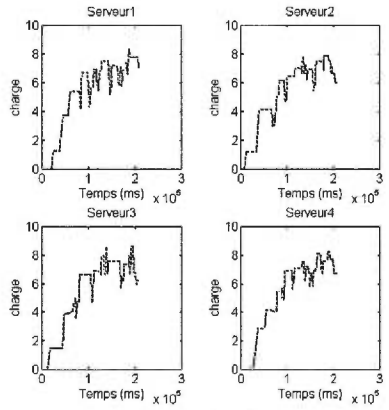
(15 clients)



(20 clients)



(25 clients)



(35 clients)

Figure 6.9 - Évolution de la charge des serveurs avec la stratégie LL

Les stratégies d'assignation aléatoire (RD) et cyclique (RR) ne requièrent aucune connaissance dynamique sur l'état des serveurs. Par contre, la stratégie LL requiert la connaissance de la charge des serveurs avant toute assignation. Nous avons enregistré la charge de chaque serveur tout au long de l'expérimentation avec cette stratégie. La figure 6.9 schématise l'évolution lors de l'expérimentation de la charge de chaque serveur pour les situations avec 10, 15, 20, 25, et 35 clients respectivement.

6.4.5. Description et interprétation des résultats

A partir des résultats obtenus faisons les observations suivantes :

- Le temps de réponse moyen des requêtes a tendance à augmenter avec le nombre de clients considéré dans chacune des situations.
- Les quatre stratégies offrent des performances comparables aux charges très basses.
- La figure 6.4 indique que le nombre de requêtes générées est proportionnel au nombre de clients avec un facteur de proportionnalité de l'ordre de 5.7 qui est presque égale à la moyenne de la distribution de Poisson utilisée pour la génération de requêtes.
- Les temps de réponse moyens obtenus avec les stratégies d'assignation RD (random) et RR (round robin) sont plus grands que ceux obtenus avec les stratégies LL (least-loaded) et CO (client-oriented). La figure 6.6 montre qu'avec les deux premières stratégies, il y a des grandes fluctuations du temps de réponse d'un client à un autre. Ceci veut dire que la déviation standard, qui mesure la variabilité autour de la moyenne, est très grande. Par

contre, avec LL et CO, les fluctuations sont moins importantes. Ceci est confirmé par la figure 6.8.

- La figure 6.7 montre qu'en termes de temps de réponse moyen, les stratégies CO et LL donnent de meilleures performances que les stratégies RR et RD et plus particulièrement aux charges moyennes et élevées. La stratégie LL est légèrement meilleure que la stratégie CO. Sous la charge la plus grande de l'expérimentation (35 clients émettant en moyenne 200 requêtes avec chacune des stratégies considérées), le temps de réponse moyen avec la stratégie LL est 3,5 fois meilleur qu'avec la stratégie CO, 12,5 fois meilleur qu'avec la stratégie RD, et 13,3 fois meilleur qu'avec la stratégie RR.
- La charge des serveurs schématisée dans la figure 6.9 évolue presque de la même façon dans chacune des situations considérées. En effet, dans les situations avec respectivement 10, 25, et 35 clients, la charge des quatre serveurs évolue de manière à peu près identique. Dans la situation avec 15 clients, la charge des serveurs 1 et 2 croît de la même façon avant de commencer à diminuer vers la fin de l'expérimentation. La charge des serveurs 3 et 4, cependant, croît brusquement au début de l'expérimentation pour atteindre rapidement les valeurs respectives d'environ 0,8 et 1,2. Elle reste constante à ces valeurs tant que la charge des serveurs 1 et 2 n'ait pas atteint ces valeurs. Pour avoir une explication à ce comportement, nous avons examiné les fichiers de sortie des clients dans cette situation. Nous avons constaté que le temps de réponse de certaines requêtes assignées aux serveurs 3 et 4 était très grand par rapport à la normale. C'est pourquoi la charge de chacun de ces serveurs reste constante pendant un certain temps. Les nouvelles requêtes sont assignées aux serveurs 1 et 2 tant que leur charge est inférieure à celle des serveurs 3 et 4. Dès que la charge des quatre serveurs atteint la même valeur, elle commence à diminuer. Dans la situation avec 20 clients, la charge des serveurs 2, 3, et 4 croît approximativement de la même façon. Cependant, la charge du serveur 1 évolue de manière différente aux autres serveurs. L'explication de ce comportement est la même que celle donnée pour la situation avec 15 clients.
- La charge moyenne de chacun des serveurs est donnée au tableau 6.4 et est schématisée graphiquement par la figure 6.10. Cette figure nous révèle que dans chacune des situations la charge moyenne est presque identique pour les quatre serveurs. La stratégie LL qui est

dynamique permet donc d'avoir une distribution presque uniforme de la charge des quatre serveurs de l'expérimentation. Ceci est l'objectif cherché par l'expérimentation de l'assignation de requêtes comme moyen de distribution de charge.

Nombre de clients	Charge moyenne			
	Serveur 1	Serveur 2	Serveur 3	Serveur 4
10	0.192	0.205	0.187	0.185
15	0.750	0.761	0.907	1.012
20	1.135	0.723	0.716	0.858
25	1.596	1.629	1.511	1.649
35	6.140	6.050	6.290	6.160

Tableau 6.4 - Charge moyenne des serveurs

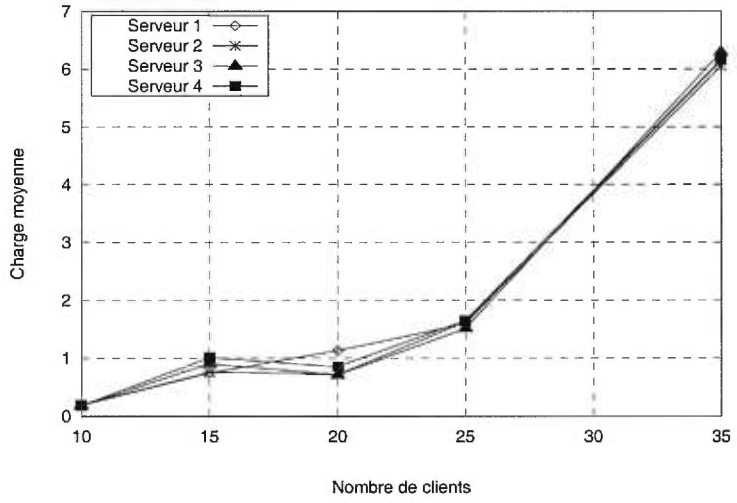


Figure 6.10 - Évolution de la charge moyenne des serveurs

- La figure 6.7 schématise la tendance du temps de réponse moyen en fonction du nombre de requêtes générées pour l'expérimentation de chacune des quatre stratégies. Avec RD et RR, il tend à croître très vite avec le nombre de requêtes. Par contre, il croît moins vite avec CO et LL. Les fichiers de sortie montrent qu'en raison de la nature probabiliste de RD et de la nature cyclique de RR, un serveur qui est déjà congestionné continue de recevoir des requêtes. Ces requêtes doivent donc attendre un certain temps avant d'être servies. Par contre, le temps de réponse moyen croît moins vite avec LL en raison de la distribution presque uniforme de la charge entre les serveurs. L'approche CO montre des performances

presque égales à celles offertes par LL aux charges basses et moyennes. Cependant, LL donne de meilleures performances que CO aux charges élevées. CO présente l'avantage de ne pas impliquer des coûts liés à la mesure de la charge et à la dissémination de l'information de charge. Pour expliquer ce comportement de CO, il est nécessaire de mener davantage d'expérimentations.

6.5. Expérimentation avec LSS

Dans cette section nous décrivons l'expérimentation que nous avons réalisée avec le prototype du service LSS. Nous avons adopté la même stratégie d'expérimentation et la même application de test que dans l'expérimentation réalisée avec le prototype de l'architecture LoDACE.

6.5.1. Configuration

Le tableau 6.5 présente la configuration d'expérimentation utilisée.

Nombre de serveurs	4 (serveur1, serveur2, serveur3, et serveur4)
Situations considérées (en nombre de clients) numérotées de 1 à 7	1, 2, 4, 6, 10, 14, et 20.
Loi de génération des requêtes	Loi de Poisson de moyenne $\lambda = 5$
Type de requêtes	Consultation de la balance (*get_balance()*)
Indicateur de performance	Temps de réponse moyen
Indicateur de charge d'un serveur	Taux d'utilisation du serveur pendant une période de temps donnée.
Indicateur de charge d'une station de travail	Nombre moyen de tâches dans la file d'attente d'exécution de la CPU.
Stratégies d'assignation considérées	***RD*** : aléatoire (random) ***RR*** : cyclique (round robin) ***LL*** : le moins chargé (least-loaded)
Durée d'envoi des requêtes par un client	1 minute

Tableau 6.5 - Configuration utilisée pour l'expérimentation de LSS

6.5.2. Résultats de l'expérimentation

La figure 6.11 représente graphiquement le temps de réponse moyen des requêtes émises par chaque client, identifié par un numéro, dans les différentes situations.

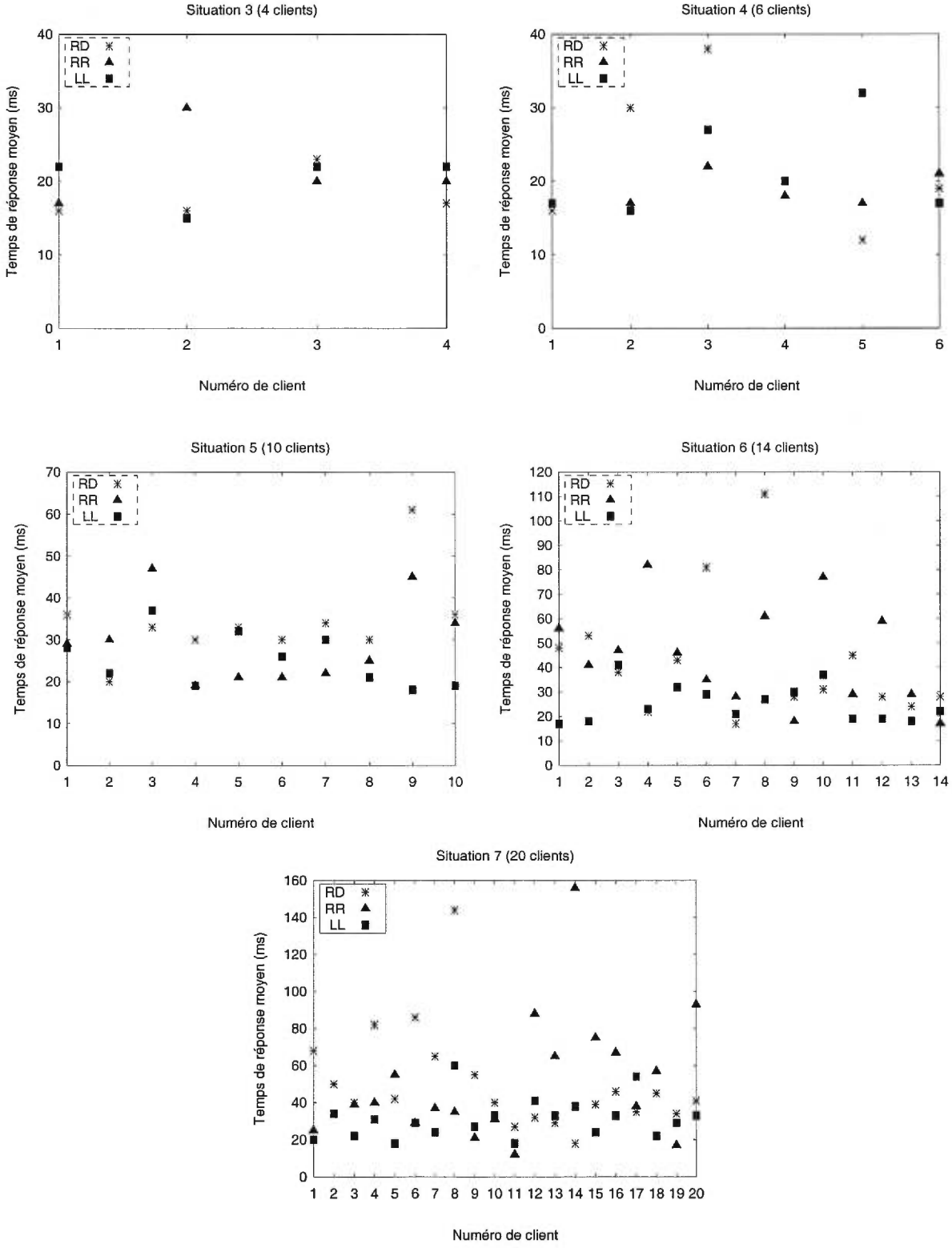


Figure 6.11 - Temps de réponse moyen de chacun des clients dans les différentes situations

Les résultats de l'expérimentation sont agrégés par situation et par stratégie d'assignation dans le tableau 6.6. Les données de ce tableau sont schématisées graphiquement par la figure 6.12 qui représente l'évolution du temps de réponse moyen en fonction du nombre de requêtes et par la figure 6.13 qui représente l'évolution de la déviation standard du temps de réponse moyen en fonction du nombre de clients.

Nbre. de clients	RD			RR			LL		
	Nbr. de req.	TRM (ms)	Dév. Std.	Nbr. de req.	TRM (ms)	Dév. Std.	Nbr. de req.	TRM (ms)	Dév. Std.
1	5	16		4	16		3	18	
2	10	15	0,00	9	15,5	0,71	15	18	1,41
4	28	18	3,37	22	21,75	5,68	27	20,25	3,50
6	34	26,17	12,34	29	24,50	13,63	30	21,50	6,53
10	53	34,30	10,45	50	29,30	9,99	73	25,20	6,44
14	86	42,64	25,60	89	44,64	20,38	87	25,21	7,63
20	122	50,9	50,90	105	59,05	59,05	131	31,15	31,15

Tableau 6.6 - Résultats de chaque situation avec les trois stratégies d'assignation

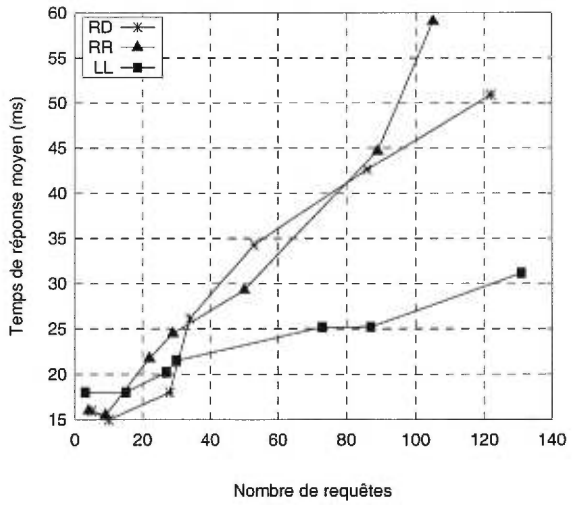


Figure 6.12 - Évolution du temps de réponse moyen

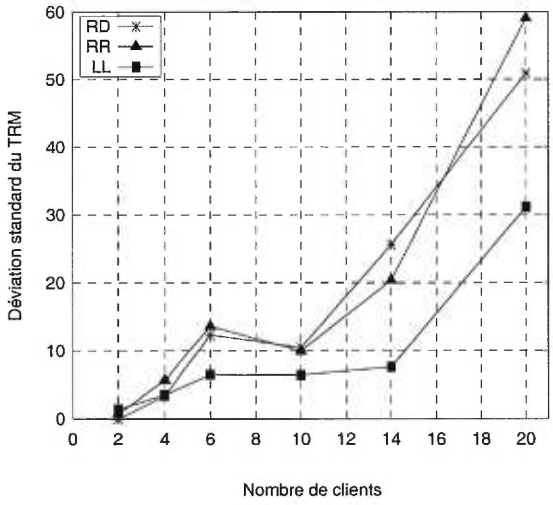
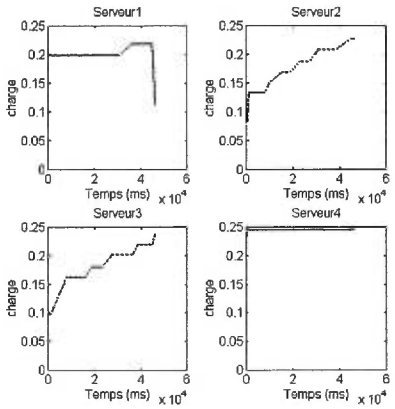
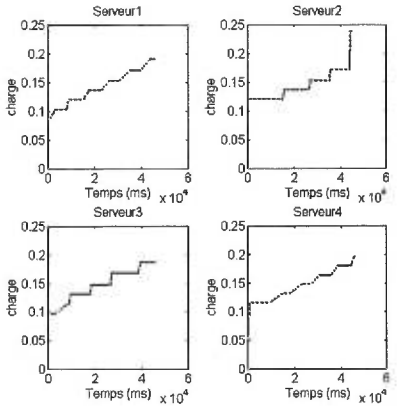


Figure 6.13 - Évolution de la déviation standard du temps de réponse moyen

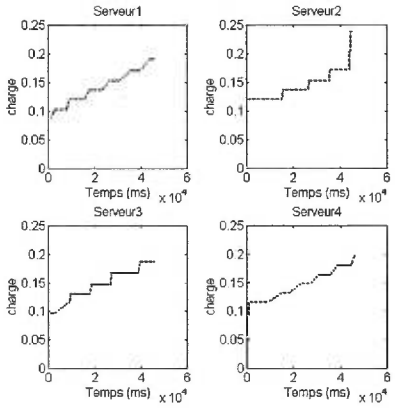
Comme avec le prototype de LoDACE, nous avons enregistré lors de l'expérimentation de la stratégie d'assignation LL, basée sur la sélection du serveur le moins chargé, la charge de chacun des serveurs dans les situations considérées. La figure 6.14 schématise l'évolution dans le temps de la charge des quatre serveurs dans chacune des situations.



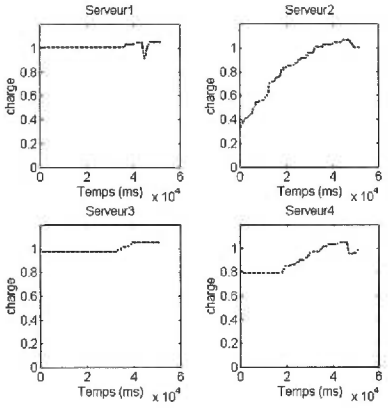
(2 clients)



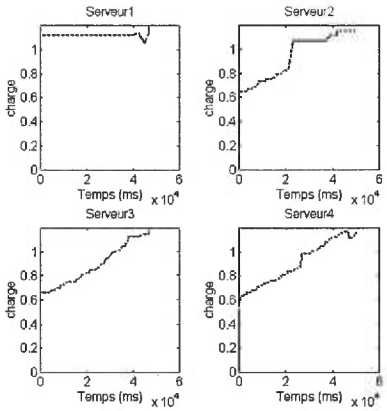
(4 clients)



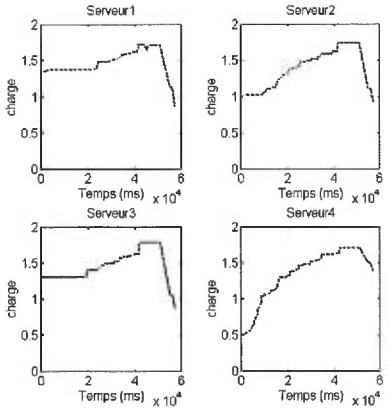
(6 clients)



(10 clients)



(14 clients)



(20 clients)

Figure 6.14 - Évolution de la charge des serveurs avec la stratégie *LL*

6.5.3. Description et interprétation des résultats

A partir des figures précédentes nous faisons les observations suivantes :

- Le temps de réponse moyen des requêtes a tendance à augmenter avec le nombre de clients considéré dans chacune des situations.
- Les figures 6.11 et 6.12 montrent que les temps de réponse moyens obtenus avec les stratégies d'assignation RD (random) et RR (round robin) sont plus grands que ceux obtenus avec la stratégie LL (least-loaded). D'autre part, la figure 6.13 montre qu'avec les stratégies RD et RR, la déviation standard du temps de réponse moyen est très grande. Par contre, elle est moins importante avec LL.
- La figure 6.14, qui schématise l'évolution de la charge des serveurs lors de l'expérimentation avec la stratégie LL, montre que la charge des quatre serveurs dans chacune des situations considérées évolue approximativement de manière similaire à quelques exceptions près. En effet, dans les situations avec respectivement 4 et 20 clients, la charge des quatre serveurs évolue de la même façon. Dans les autres situations, la charge de deux ou trois serveurs croît de façon analogue et celle des autres serveurs évolue différemment. L'explication de ce comportement est similaire à celle que nous avons donnée dans l'expérimentation avec LoDACE.
- Le tableau 6.7 donne la charge moyenne de chacun des serveurs dans chacune des situations considérées. Les données de ce tableau sont schématisées graphiquement par la figure 6.15. Cette figure révèle que dans chaque situation, la charge moyenne des quatre serveurs est presque identique. Ceci veut dire que la charge est distribuée uniformément entre les quatre serveurs et par conséquent l'objectif de distribution de la charge entre les serveurs est atteint.

Nombre de clients	Charge moyenne			
	Serveur 1	Serveur 2	Serveur 3	Serveur 4
2	0.196	0.176	0.179	0.245
4	0.140	0.149	0.143	0.140
6	0.371	0.364	0.430	0.355
10	1.008	0.815	0.992	0.897
14	1.123	0.937	0.907	0.901
20	1.476	1.383	1.460	1.353

Tableau 6.7 - Charge moyenne des serveurs

- La figure 6.12 montre bien la tendance du temps de réponse moyen en fonction du nombre de requêtes générées pour les quatre stratégies d'assignation. Avec RD et RR, il a tendance à croître très vite avec le nombre de requêtes. Par contre, il a tendance à croître moins vite avec LL. En raison de la nature probabiliste de RD et de la nature cyclique de RR, un serveur qui est déjà congestionné peut continuer à recevoir des requêtes de service. Ces requêtes doivent donc attendre un certain temps avant d'être servies. Avec la stratégie LL, les serveurs congestionnés ou plus chargés ne sont pas sélectionnés. Ceci explique la croissance moins rapide du temps de réponse avec cette stratégie et la distribution uniforme de la charge entre les serveurs.
- Aux charges basses, il n'est pas avantageux d'utiliser la stratégie LL car les deux stratégies RR et RD donnent de bons résultats à ce niveau de charge en comparaison avec LL et ne requièrent ni la collecte ni la dissémination de l'information de charge.

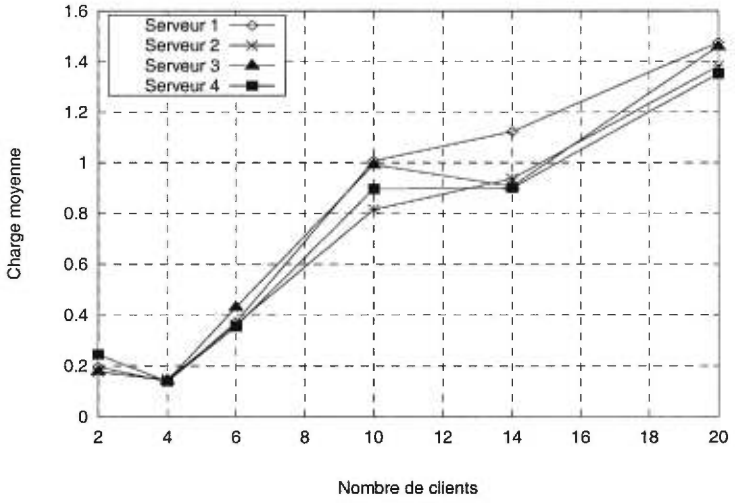


Figure 6.15 - Évolution de la charge moyenne des serveurs

6.6. Conclusion

Dans ce chapitre, nous avons présenté les expérimentations que nous avons conduites avec les prototypes de l'architecture LoDACE et du service LSS.

Les résultats de l'expérimentation avec le prototype de LoDACE révèlent que la stratégie LL offre de meilleures performances que les autres stratégies en termes de temps de réponse

moyen considéré comme indicateur de performance. De plus, la charge des serveurs évolue presque de la même façon avec cette stratégie. Ceci permet d'éviter la congestion des serveurs et d'augmenter leur disponibilité. Le temps de réponse moyen croît très vite avec RR et RD au fur et à mesure que le nombre de requêtes augmente. Cependant, aux charges basses et moyennes, la stratégie CO offre des performances comparables à celles offertes par LL. De plus, cette stratégie n'implique pas des coûts liés à la mesure de la charge des serveurs et à la dissémination de l'information de charge. Aux charges hautes, le temps de réponse moyen avec CO a tendance à croître plus vite qu'avec LL.

Les résultats obtenus avec le prototype de LSS sont presque similaires à ceux obtenus avec le prototype de LoDACE. En effet, la stratégie LL offre de meilleures performances en comparaison avec RD et RR en termes de temps de réponse moyen. De plus, avec LL la distribution de la charge des serveurs est presque uniforme. La comparaison des tableaux récapitulatifs des résultats obtenus avec LoDACE et LSS (tableaux 6.3 et 6.6) montre qu'avec la stratégie LL, par exemple, les performances en termes de temps de réponse obtenus avec LSS sont meilleures qu'avec LoDACE. Cependant, les expérimentations ont été conduites dans des conditions différentes bien que la configuration d'expérimentation soit la même. Par conséquent, il nous est difficile de confirmer que LSS est meilleur que LoDACE.

Bien que ces expérimentations présentent certaines limitations que nous discutons au chapitre 7, les résultats obtenus nous permettent de conclure que l'approche d'assignation de requêtes dans les systèmes distribués à base d'objets est un moyen efficace et peu onéreux pour accomplir la distribution de charge entre des serveurs offrant un même type de service. Les stratégies aléatoire et cyclique, largement utilisées dans beaucoup de systèmes, donnent des résultats acceptables lorsque la charge du système est faible ou moyenne. Cependant, lorsque la charge du système devient importante, des stratégies dynamiques comme LL s'imposent.

Chapitre 7
Discussions

Ce chapitre présente une discussion de l'approche de distribution de charge par l'assignation de requêtes, des expérimentations de cette approche, et des choix effectués lors de la conception et de l'expérimentation de LoDACE et de LSS.

7.1. Performances des stratégies d'assignation

Les expérimentations conduites à l'aide des prototypes de LoDACE et de LSS nous ont permis d'évaluer les performances en termes de temps de réponse moyen de quatre stratégies d'assignation de requêtes : (1) la stratégie aléatoire (*RD*), (2) la stratégie cyclique (*RR*), (3) la stratégie orientée client (*CO*), et (4) la stratégie de sélection du serveur le moins chargé (*LL*). Contrairement aux trois premières stratégies, la stratégie LL est dynamique car elle tient compte de la charge actuelle des serveurs dans la sélection du serveur cible pour traiter une nouvelle requête. La stratégie LL s'est avérée plus performante en termes de temps de réponse moyen en comparaison avec les trois autres stratégies qui ne tiennent pas compte de l'état des serveurs. Sous la charge la plus grande de l'expérimentation (35 clients émettant en moyenne 200 requêtes avec chacune des stratégies considérées), le temps de réponse moyen avec la stratégie LL est 3,5 fois meilleur qu'avec la stratégie CO, 12.5 fois meilleur qu'avec la stratégie RD, et 13.3 fois meilleur que la stratégie RR. De plus, la stratégie LL permet d'avoir une distribution presque uniforme de la charge des serveurs considérés dans l'expérimentation. La stratégie CO présente également des performances presque égales à celles de LL aux charges basses et moyennes.

Seule la charge des serveurs et des machines est prise en considération par la stratégie LL que nous avons expérimentée. Dans une application réelle, cependant, les performances des

stratégies dynamiques d'assignation de requêtes peuvent être influencées par d'autres facteurs tels que :

- le nombre de serveurs,
- le nombre de méthodes du serveur,
- le taux d'arrivée des requêtes,
- l'emplacement des serveurs et des clients, et
- la capacité des ressources matérielles des machines.

7.1.1. Nombre de serveurs

Le nombre de serveurs a un impact sur les performances du système. En effet, si la demande de service est grande et si le nombre de serveurs est limité, alors des requêtes peuvent avoir à attendre dans les files d'attentes des serveurs avant d'être servies. De plus, il y a risque de congestion de certains serveurs. D'où la nécessité de mettre en place des mécanismes de distribution de charge car il n'est pas souvent possible d'ajouter des serveurs pour faire face à une augmentation de la charge.

Dans certains systèmes distribués à base d'objets, un serveur peut supporter un ou de multiples fils d'exécution (threads). Un serveur avec de multiples fils d'exécution crée un nouveau fil d'exécution pour chaque nouvelle requête reçue par le serveur. L'inconvénient du modèle *fil d'exécution par requête* (thread-per-request) est qu'il consomme un grand nombre de ressources plus particulièrement lorsque plusieurs requêtes sont envoyées simultanément au serveur [Schmidt98]. En plus, le coût de ce modèle est extrêmement important lorsque les requêtes ont juste des durées de vie courtes. Une autre variation de ce modèle est celle d'un *groupe de fils d'exécution* (thread-pool) créés dans le serveur lors de son activation. Les requêtes des clients sont traitées concurremment jusqu'à ce que le nombre de requêtes simultanées excède le nombre de fils d'exécution dans le groupe. Les requêtes additionnelles sont mises dans une file d'attente si tous les fils d'exécution du groupe sont occupés.

Les stratégies d'assignation doivent donc tenir compte du niveau d'exécution parallèle (multi-threading) supporté par les serveurs afin de pouvoir affecter les requêtes aux serveurs ayant plus de ressources disponibles.

7.1.2. Nombre de méthodes du serveur

Les serveurs d'une même grappe sont des serveurs objet qui implémentent une même interface comportant un certain nombre d'attributs et de méthodes. Ces méthodes peuvent avoir des besoins différents en ressources systèmes (temps CPU, mémoire, entrées/sorties, etc.). Les paramètres et les valeurs de retour de ces méthodes peuvent avoir des tailles différentes. Ceci implique des coûts d'exécution et de transfert variés.

Comme nous l'avons présenté au chapitre 3, il peut y avoir des relations de dépendance entre les méthodes des objets d'une application distribuée. Par exemple, une méthode d'un objet donné peut appeler une méthode d'un autre objet. Par conséquent, les coûts d'exécution ne sont pas a priori prévisibles et l'assignation optimale des invocations de méthodes est difficile à réaliser.

7.1.3. Taux d'arrivée des requêtes

L'analyse du modèle analytique au chapitre 3 a montré que les performances du système en termes de temps de réponse et d'utilisation des serveurs est affectée par le taux total d'arrivée des requêtes au répartiteur. Le système se sature lorsque ce taux atteint une valeur appelée taux de saturation du système qui est égale à la plus petite valeur des taux de saturation des serveurs du système. Les clients peuvent avoir des taux de génération de requêtes différents. Ces requêtes peuvent être hétérogènes étant donné qu'un serveur objet implémente les méthodes de son interface. Chaque requête (invocation de méthode) impose une certaine charge sur le serveur et sur la machine du serveur. Dans la modélisation analytique de l'approche orientée répartiteur décrite au chapitre 3, chaque méthode de l'interface du serveur est assimilée à une classe de service ayant ses propres caractéristiques (taux d'arrivée des requêtes de cette classe, demande de service moyenne à un centre de service, etc.).

7.1.4. Emplacement des serveurs et des clients

L'emplacement des serveurs et des clients peut affecter les performances du système en raison des coûts de communication impliqués. D'où la nécessité de faire un placement initial adéquat des serveurs dans le système de sorte que ces derniers soient proches des clients potentiels. La migration d'objets permet de déplacer un serveur auprès des clients. Cependant,

les coûts de communication liés à ce déplacement peuvent être excessifs et sa réalisation dans un environnement hétérogène est extrêmement difficile.

7.1.5. Capacités des ressources matérielles et logicielles

Les capacités des ressources matérielles du système ont un impact direct sur les performances du système. En effet, les machines du système peuvent être hétérogènes au niveau des ressources matérielles (nombre de processeurs, vitesse des processeurs, quantité de mémoire disponible, taux de transfert de et vers la mémoire secondaire, etc.) et au niveau logiciel (systèmes d'exploitation et langages supportés). Ainsi, les stratégies d'assignation doivent tenir compte de cet aspect de sorte que les serveurs ayant plus de ressources soient plus sollicités que ceux ayant des ressources limitées.

7.2. Limitations des expérimentations

Les expérimentations réalisées à l'aide des prototypes de LoDACE et de LSS présentent les limitations suivantes :

- L'expérimentation a été conduite avec seulement quatre serveurs offrant un même type de service. Il serait alors intéressant de mener d'autres expérimentations afin d'évaluer les performances des stratégies d'assignation considérées avec différents nombres de serveurs et d'évaluer l'impact de ce facteur sur les performances globales du système.
- La génération de requêtes par les clients était supposée suivre une loi de Poisson de moyenne λ=5. Dans une application réelle, la génération de requêtes par les clients peut se faire à des taux différents et peut suivre des distributions différentes. Par conséquent, il est important d'expérimenter différents taux de génération de requêtes et d'autres distributions autres que la loi de Poisson afin d'évaluer l'impact de ce facteur sur les performances du système.
- Les requêtes utilisées dans l'expérimentation de LoDACE et de LSS sont simples et consistent seulement à lire le contenu d'une variable du serveur. Dans une application réelle, les requêtes peuvent être plus complexes en ayant des paramètres et des valeurs de retour de tailles différentes.

- Il n'y a pas de sauvegarde de l'état entre deux exécutions. Les applications client/serveur sont souvent orientées session. C'est-à-dire que l'état d'exécution est sauvegardé d'une exécution à une autre. Par conséquent, les requêtes d'une même session doivent être exécutées par un même serveur. Dans ce type d'applications, les stratégies RD, RR, CO, et LL doivent être alors utilisées uniquement lors de l'initiation d'une session et non pour l'assignation de chaque requête.
- La bande passante disponible entre le client et le serveur n'est pas prise en considération par les stratégies d'assignation expérimentées. Dans un système à grande échelle, les stratégies d'assignation doivent tenir compte de cet aspect afin d'éviter de choisir des serveurs dont les liens réseaux sont congestionnés. Pour cela, il faut disposer d'outils permettant d'évaluer la bande passante disponible à un instant donné entre clients et serveurs [Carter96].
- Les aspects de sécurité n'ont pas été pris en considération par cette étude. Ceci représente en fait une des limitations importantes car une requête d'un client ne peut s'exécuter que par un serveur auquel il a le droit d'y accéder. Donc, les stratégies d'assignation doivent tenir compte de cet aspect dans leur prise de décision.
- Les serveurs de l'expérimentation sont des serveurs répliqués sans état. Dans les applications réelles, les serveurs ont généralement un état. Par conséquent, il est nécessaire d'intégrer à l'architecture LoDACE un gestionnaire de la réplication qui permet de maintenir la consistance entre les serveurs répliqués.

7.3. Impact des choix

Dans cette section, nous discutons l'impact des choix que nous avons effectués lors de la conception et de l'expérimentation de l'architecture LoDACE et du service LSS de distribution de charge.

7.3.1. L'ORB

L'ORB utilisé dans les prototypes de LoDACE et de LSS est OrbixWeb. L'utilisation d'une plate-forme CORBA procure l'avantage de la facilité de développement et de déploiement des clients et des serveurs, ainsi que la possibilité d'utiliser des langages et des machines

hétérogènes. Le choix de l'ORB a un impact direct sur les performances escomptées. En effet, les ORBs présents sur le marché présentent des performances différentes et peuvent offrir des fonctionnalités différentes [Gokhale96, Gokhale97, AT&T]. Par exemple, le mécanisme de filtre que nous avons utilisé pour l'évaluation de la charge des serveurs est offert par peu d'ORBs dont OrbixWeb. Le mécanisme de smartMaps, présenté brièvement à la section 5.2.3, est aussi un mécanisme propre à BEA ObjectBroker pour la mise en oeuvre de diverses stratégies d'assignation.

7.3.2. Service d'association

Pour les besoins d'expérimentation, nous avons considéré dans le prototype de LoDACE un service d'association centralisé qui est offert par un seul Binder. Celui-ci reçoit toutes les requêtes des clients et les redirige au service de courtage OrbixTrader. Un goulot d'étranglement peut se produire au niveau du Binder si la demande de service est très grande. De plus, toute panne au niveau du Binder entraînerait l'effondrement du système. Pour éviter cette situation dans un cas réel, il est possible de déployer un Binder par machine du système, comme c'est illustré par la figure 7.1, de sorte que les clients d'une même machine adressent leurs requêtes au Binder installé sur cette machine ou, le cas échéant, à un Binder d'une machine voisine.

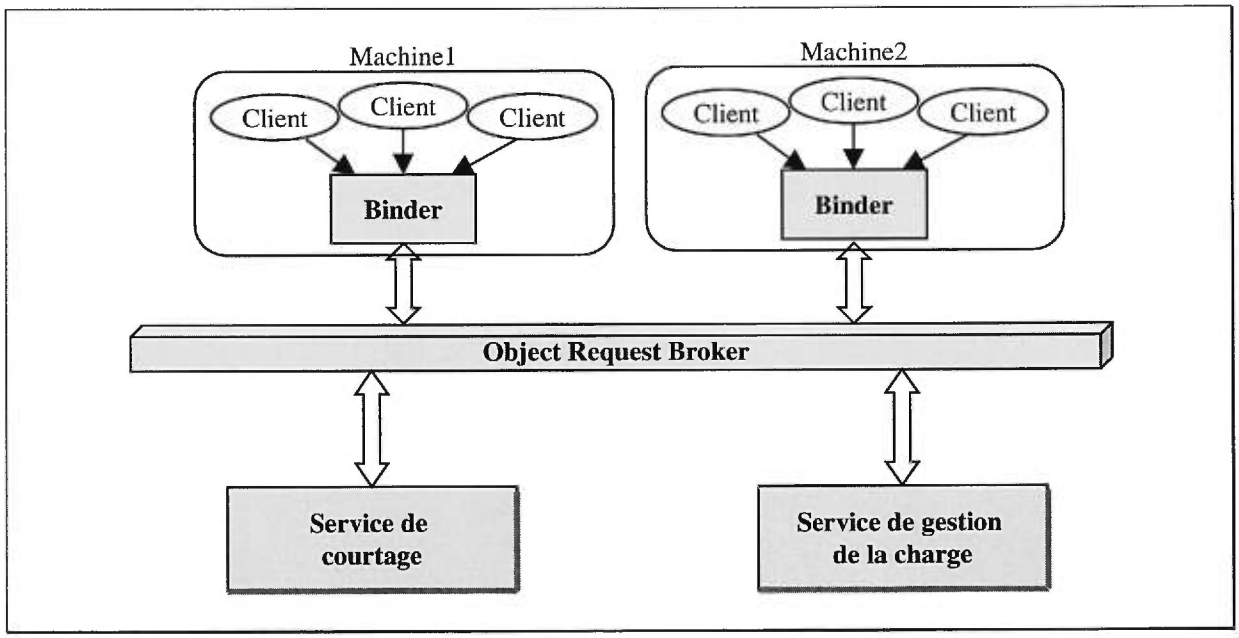


Figure 7.1 - Service d'association distribué

Les Binders peuvent fonctionner de manière autonome et chaque Binder peut également avoir sa propre stratégie d'assignation de requêtes. Par exemple, un Binder d'une machine donnée peut utiliser la stratégie aléatoire qui ne nécessite aucune information sur l'état des serveurs, et le Binder d'une autre machine peut utiliser la stratégie basée sur la sélection du serveur le moins chargé. Il est aussi possible d'avoir de multiples Binders par machine dans laquelle chaque Binder implémente une stratégie donnée. Les clients de cette machine auront par la suite le choix du Binder qui leur convient.

Au niveau du prototype de LSS, le service d'association est distribué. Chaque client crée son propre Binder par lequel passent toutes ses requêtes. Donc, le problème précédent ne se pose pas.

7.3.3. Service de courtage

Comme pour le service d'association, un seul courtier (trader) est considéré aussi bien dans le prototype de LoDACE que de celui de LSS pour permettre la découverte dynamique des serveurs. Un goulot d'étranglement peut ainsi se produire également au niveau de ce courtier si un grand nombre de requêtes lui est adressé. Dans le cas d'un grand système, il est nécessaire de partitionner le système en plusieurs domaines dans lesquels chaque domaine comporte un courtier qui reçoit les requêtes des clients de ce domaine. La norme ISO [ISO96b] prévoit la possibilité de former une fédération de courtiers afin de permettre à un client d'avoir accès à un espace plus large d'offres de service outre celles qui sont disponibles dans le domaine auquel il appartient.

Il est aussi possible de considérer d'autres services de localisation des serveurs. Le choix d'un service de localisation donné dépendra de la taille du système. Dans le cas d'un petit système, un simple fichier de configuration contenant les références des serveurs peut suffire. Cependant, dans le cas d'un système à grande échelle, il est nécessaire de considérer des services biens adaptés à la taille du système tel que SLP (Service Location Protocole) [IETF97] qui permet aux clients la découverte et la sélection des ressources et des services dans les réseaux IP (Internet Protocol). A ceci s'ajoute le problème de gestion des types de service pour pouvoir considérer des serveurs offrant divers types de service.

7.3.4. Gestion de la charge

Le prototype de LoDACE décrit à la section 4.4 utilise un service centralisé de gestion de la charge. Le gestionnaire de charge centralise l'information de charge, de tous les serveurs, qu'il reçoit des moniteurs de charge présents dans chaque machine du système. La politique d'information implémentée est une politique sur demande. Cette politique est intéressante lorsque le nombre de serveurs n'est pas grand. Dans le cas contraire, cette politique peut s'avérer très lourde car il faut interroger tous les serveurs d'une grappe avant de pouvoir faire le choix d'un serveur approprié (par exemple avec la stratégie *LL*). Dans ce cas, la politique périodique peut s'avérer intéressante bien que l'information de charge maintenue par le gestionnaire de charge risque d'être obsolète. D'où la nécessité d'ajuster la valeur de la période de collecte de l'information de charge.

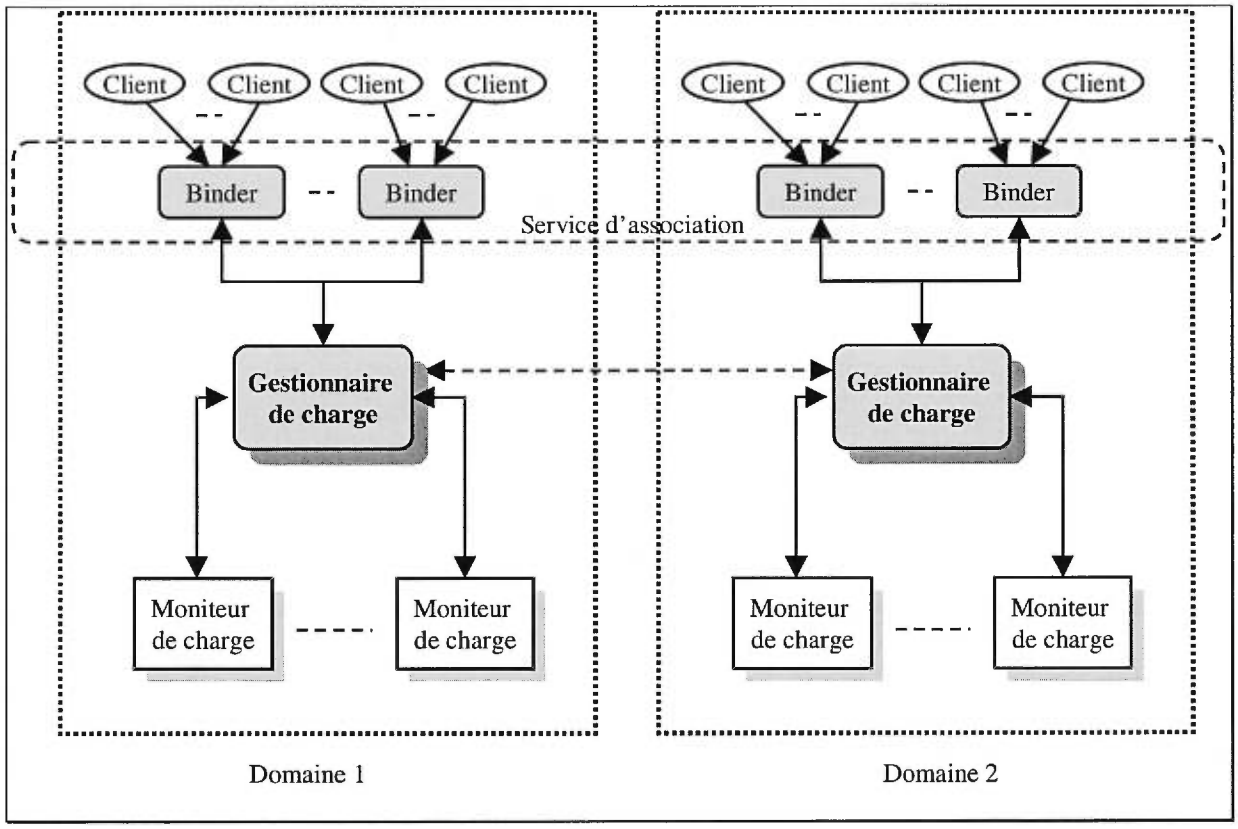


Figure 7.2 - Gestion de l'information de charge par de multiples gestionnaires.

Comme pour le service d'association, pour éviter que le gestionnaire de charge devienne un goulot d'étranglement, le système peut être partitionné en plusieurs domaines et un

gestionnaire de charge serait déployé par domaine comme dans la figure 7.2. Les gestionnaires de charge pourraient s'échanger l'information de charge comme c'est illustré par la figure 7.3.

Avec le prototype du service LSS, la gestion de la charge est distribuée. Dans le cas d'une stratégie d'assignation dynamique, chaque Binder collecte l'information de charge dont il a besoin sur demande. Le Binder n'a aucune connaissance des serveurs qui offrent tel ou tel type de service avant la réception du service de courtage de la liste d'offres de service qui correspondent au type de service requis par le client (voir scénario coté client à la section 5.6.1). Par conséquent, il est difficile d'adopter dans LSS une politique périodique pour la collecte de l'information de charge des serveurs.

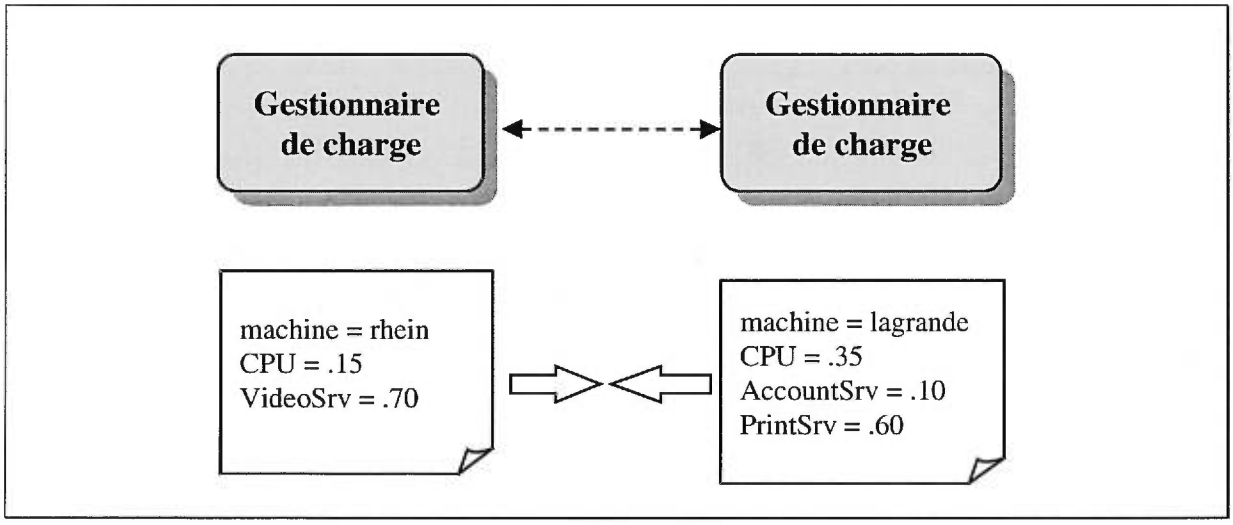


Figure 7.3 - Échange de l'information de charge entre gestionnaires

7.4. Autres aspects

Dans cette section, nous discutons quelques aspects qui doivent être considérés ultérieurement dans LoDACE et dans LSS afin de mieux répondre aux besoins des applications distribuées. Ces aspects sont essentiellement : le passage à l'échelle (scalability), la disponibilité et la performance, la tolérance aux fautes, la qualité de l'assignation, et l'optimisation de l'utilisation des ressources.

7.4.1. Passage à l'échelle

Le passage à l'échelle (scalability) est un aspect critique de toute application distribuée. Il représente sa capacité à croître avec le nombre d'utilisateurs et de services requis. L'architecture LoDACE et le service LSS sont extensibles (scalable) en termes de nombre de serveurs. En effet, des serveurs peuvent être ajoutés pour répondre à la demande croissante des clients. Cependant, comme nous l'avons signalé à la section 7.2 concernant les limitations des expérimentations, il est nécessaire d'intégrer des mécanismes de réplication pour maintenir la consistance entre les exemplaires d'un même serveur. Plusieurs serveurs peuvent coexister dans une même machine, et il n'y a aucune restriction au sujet de leurs types de service. Par conséquent, deux serveurs ou plus fournissant le même type de service peuvent être déployés sur la même machine. Dans le cas de grands systèmes, plusieurs domaines peuvent être considérés afin d'éviter qu'une des composantes de l'architecture devienne l'objet d'un goulot d'étranglement comme nous l'avons expliqué à la section 7.3.

7.4.2. Disponibilité et performance

Les résultats de l'expérimentation ont montré que la stratégie LL permet d'avoir une distribution presque uniforme de la charge des serveurs. Ceci augmente leur disponibilité et réduit le temps de réponse moyen des demandes des clients. L'analyse de la charge des serveurs permettra d'identifier les serveurs qui sont le plus congestionnés. Des instances additionnelles de ces serveurs peuvent être alors créées dans des machines appropriées du système par le gestionnaire de réplication afin de répartir la charge entre les serveurs déjà existants et ces instances nouvellement créées.

7.4.3. Tolérance aux fautes

La tolérance aux fautes est un aspect très important dans les applications qui exigent une très grande disponibilité des serveurs. Dans LoDACE et dans LSS, la panne d'un serveur n'empêchera pas le service requis d'être offert au client si d'autres serveurs du système offrent ce même type de service. Quand un client détecte qu'un serveur auquel sa requête a été assignée ne répond pas au bout d'un certain temps, il doit renvoyer sa requête au Binder pour qu'elle soit assignée à un autre serveur qui est disponible. Cette solution ne résout que partiellement le problème car l'état d'exécution du serveur qui est tombé en panne n'est pas

sauvegardé sur disque. Pour assurer une transparence vis à vis du client face à ce type de problème, des mécanismes plus sophistiqués de tolérance aux fautes doivent être considérés.

En plus de ces mesures, les services qui composent l'architecture LoDACE et le service LSS (services d'association, de courtage, et de gestion de la charge) ne doivent pas être des points centraux susceptibles de tomber en panne et de provoquer l'effondrement de tout le système. Les solutions que nous avons discutées à la section 7.3, pour éviter les goulots d'étranglement dans chacun de ces services, vont contribuer à la tolérance aux fautes dans le système.

7.4.4. Qualité de l'assignation

Les stratégies d'assignation de requêtes doivent tenir compte des facteurs discutés à la section 7.1 afin de pouvoir faire le meilleur choix possible du serveur cible pour une requête donnée.

7.4.5. Optimisation

L'optimisation de l'utilisation des ressources du système requiert que le trafic soit réparti entre les serveurs d'une même grappe sur la base d'informations de configuration de sorte que les serveurs ayant plus de ressources disponibles reçoivent plus de requêtes à traiter comme c'est le cas dans LSF [Platform94] et Condor [Condor, Litzkow88].

7.5. Conclusion

Ce chapitre a discuté les facteurs qui peuvent influencer les performances des stratégies d'assignation de requêtes. La prise en compte de ces facteurs permettrait de faire le meilleur choix possible du serveur cible.

Les expérimentations conduites à l'aide des prototypes de LoDACE et de LSS présentent certaines limitations qui sont dues d'une part aux choix effectués lors de la conception et de l'implémentation des prototypes de LoDACE et de LSS, et d'autre part à la configuration d'expérimentation utilisée. La prise en considération de ces limitations et des facteurs pouvant avoir un impact sur les performances des stratégies d'assignation permettrait d'apporter des améliorations à l'architecture LoDACE et au service LSS en vue d'offrir de nouvelles fonctionnalités et répondre à certaines exigences telles que la tolérance aux fautes, le passage à l'échelle (scalability), la performance et la disponibilité.

Chapitre 8
Conclusion

Dans ce travail, nous avons étudié les techniques utilisées pour mettre en oeuvre la distribution de charge dans les systèmes distribués à base d'objets en tenant compte des spécificités de ces systèmes, soit la structuration des applications en objets distribués, la communication par invocation de méthodes, et l'hétérogénéité des machines et des systèmes d'exploitation. Nous distinguons principalement trois techniques : le placement initial d'objets, la migration d'objets, et l'assignation de requêtes.

Les solutions basées sur les deux premières techniques sont des solutions propriétaires car elles s'appuient sur des concepts propres aux systèmes sous-jacents. Ces solutions ne peuvent pas être utilisées dans un système hétérogène. La migration d'objets, plus particulièrement, est difficile à réaliser car elle exige la sauvegarde, le transfert, et la restauration de l'état de l'objet, de son contexte d'exécution, et de ses canaux de communication.

Nous avons retenu dans ce travail la méthode de l'assignation de requêtes (invocations de méthodes) comme moyen efficace et peu onéreux pour réaliser la distribution de charge dans ces systèmes. Cette technique est utilisée principalement dans les architectures client/serveur.

Les principales contributions de ce travail, outre la synthèse des méthodes de distribution de charge dans les systèmes distribués objet, résident aux niveaux de la formulation théorique de l'assignation de requêtes dans ces systèmes, de la proposition de trois différentes approches d'assignation, de la conception d'une architecture d'assignation basée sur ces trois approches, de la conception d'un service de distribution de charge dans les environnements CORBA, et de l'expérimentation des ces approches à l'aide de prototypes. Ces contributions et les perspectives futures sont décrites dans les sections suivantes.

8.1. Contributions

Approches d'assignation

Nous avons proposé trois approches sous-optimales pour réaliser l'assignation de requêtes : l'approche orientée client, l'approche orientée répartiteur, et l'approche orientée serveur. Les deux premières approches sont inspirées des méthodes de distribution de charge utilisées par certains systèmes distribués conformes au modèle client/serveur. La dernière approche est inspirée des techniques utilisées dans les systèmes distribués classiques pour réaliser la distribution de charge. Elle est basée sur la coopération des serveurs objet par échange de leurs informations d'état. Nous avons proposé des algorithmes pour la réalisation des stratégies source-initiative et receveur-initiative qui rentrent dans le cadre de cette approche.

Comme nous l'avons discuté à la section 3.4, seule l'approche orientée répartiteur se prête bien à la modélisation analytique. En effet, nous avons modélisé un système composé d'une grappe de serveurs objets, d'un répartiteur, et d'un ensemble de clients à l'aide d'un système de files d'attente à multiples classes de service. Les invocations d'une même méthode de l'interface implémentée par les serveurs de la grappe correspondent à une même classe de service. Ce modèle permet d'évaluer analytiquement les performances moyennes de ce système en termes de temps de réponse et d'utilisation des serveurs.

Architecture d'assignation de requêtes

Afin de faciliter la mise en œuvre des trois approches précédentes, nous avons proposé une architecture d'assignation de requêtes appelée LoDACE. L'objectif de cette architecture est de : (1) supporter n'importe quel type de service, (2) être facilement configurable pour réaliser ces approches d'assignation, (3) être déployé dans un environnement où les serveurs peuvent être créés ou supprimés dynamiquement, et (4) être évolutif en supportant un grand nombre de clients et de serveurs et en s'adaptant à la taille du système. Cette architecture est basée sur l'organisation des serveurs objet dans des grappes logiques suivant les types de service offerts par ces serveurs. Elle comprend les services : de découverte dynamique des serveurs, de surveillance de la charge des serveurs et des machines, et d'association (interface entre clients et serveurs). Nous avons développé un prototype de LoDACE dans un

environnement OrbixWeb avec le langage de programmation Java. Ce prototype réalise principalement l'approche orientée répartiteur.

Service de distribution de charge dans les environnements CORBA

Étant donné l'absence d'un service standard pour la distribution de charge dans les environnements CORBA, nous avons proposé un service basé sur l'assignation d'invocations de méthodes pour offrir cette fonctionnalité. La conception de ce service, appelé LSS, est inspirée de l'architecture LoDACE. Nous avons réalisé un prototype de ce service dans un environnement OrbixWeb avec le langage de programmation Java.

Expérimentation et validation

Comme application de test, nous avons considéré la gestion de comptes bancaires qui comporte un certain nombre de serveurs et de clients. Elle a servi dans l'expérimentation de certaines stratégies d'assignation avec les prototypes de LoDACE et de LSS. Les stratégies testées sont : La stratégie dynamique basée sur la sélection du serveur le moins chargé (least-loaded - LL), et les stratégies statiques aléatoire (random - RD), cyclique (round robin - RR) et orientée client (client-oriented - CO). Le temps de réponse de chaque requête est mesuré ainsi que la charge des serveurs dans le cas de la stratégie LL.

La stratégie LL s'est avérée plus performante en termes de temps de réponse moyen en comparaison avec les stratégies RD, RR, et CO. De plus, la stratégie LL permet d'avoir une distribution presque uniforme de la charge des serveurs. Les résultats obtenus avec la stratégie RD sont très proches des résultats obtenus par la modélisation analytique d'une grappe de serveurs dans laquelle la distribution probabiliste considérée est du type aléatoire.

8.2. Perspectives

Les perspectives de ce travail portent sur plusieurs aspects autant sur le plan réalisation que sur le plan prospectif.

Extensions de LoDACE

LoDACE est une architecture ouverte et extensible. D'autres composantes peuvent être ajoutées à cette architecture pour offrir d'autres fonctionnalités ou pour combler les limitations discutées au chapitre 7. Par exemple, les composantes suivantes peuvent être ajoutées :

- *Gestionnaire de la réplication* : cette composante sera responsable du maintien de la consistance entre les serveurs d'un même cluster. D'autre part, elle sera responsable de la création, le cas échéant, de nouvelles instances d'un serveur lorsque toutes les instances de celui-ci sont surchargées.
- *Gestionnaire de la sécurité* : cette composante permettra d'ajouter des contraintes sur le choix des serveurs cibles pour le traitement des invocations de méthodes. Les stratégies d'assignation de requêtes seraient donc capables de tenir compte de l'aspect sécurité lors de leurs prises de décision.
- *Outils d'évaluation de la bande passante* : ces outils serviront à évaluer la bande passante disponible entre les clients et les serveurs. Par conséquent, les stratégies d'assignation de requêtes seraient capables d'ajuster leurs décisions en fonction des valeurs calculées. Ceci permettra d'éviter le choix des serveurs dont les liens réseau sont congestionnés.

Interactions de LSS avec d'autres services CORBA

Le service LSS est basé sur l'interaction avec le service de courtage pour la découverte dynamique des serveurs. Il serait intéressant d'étudier l'interaction avec d'autres services standards de CORBA comme les services de sécurité, de persistance, et de cycle de vie.

Expérimentation de l'approche orientée serveur

Dans ce travail, nous avons expérimenté principalement les deux approches orientée client et orientée répartiteur avec le prototype de LoDACE. Comme continuation de ce travail, l'approche orientée serveur est à expérimenter afin d'évaluer et de comparer ses performances avec celles des approches orientée client et orientée répartiteur.

Applications réelles

L'application de test ayant servi à conduire des expérimentations est une application simple. Il est alors souhaitable d'expérimenter avec une application réelle qui reflète la charge à laquelle pourraient être soumis les serveurs dans un cas concret. Par exemple, dans le cas du serveur Cisco 3410 [Cisco2000] de vidéo sur demande, cette charge est estimée à 15000 clients au maximum sur la base de 300 requêtes par minute. Les vidéos sont supposées avoir une durée de 60 minutes.

Support de langages différents

Les prototypes de LoDACE et de LSS ont été réalisés dans un environnement OrbixWeb avec le langage de programmation Java. Les serveurs et les clients de l'expérimentation ont été réalisés avec Java. Il serait intéressant de mener des expérimentations avec des applications réelles formées de serveurs ayant une même interface IDL mais réalisés dans des langages de programmation différents, comme C++ , Java, et Smaltalk.

Expérimentation avec d'autres ORBs

Les prototypes de LoDACE et de LSS sont réalisés dans un environnement OrbixWeb et s'appuient sur l'utilisation d'OrbixTrader. Afin de mieux évaluer cette architecture et ce service, il serait souhaitable de réaliser ces prototypes et de les expérimenter en utilisant d'autres ORBs et éventuellement d'autres implémentations du service de courtage. Comme nous l'avons vu à la section 7.3.1, ces ORBs différent au niveau de leurs performances et des fonctionnalités qu'ils offrent.

L'effort de portage à d'autres ORBs n'est pas négligeable. En effet, après traduction de la spécification IDL des prototypes vers un des langages supportés par l'ORB en question, il faut d'une part adapter les méthodes de chaque composante au nouvel environnement. Dans le cas d'un langage autre que Java il faut complètement réécrire ces méthodes. D'autre part, l'évaluation de la charge doit se faire en utilisant un mécanisme qui est similaire au mécanisme de filtre d'OrbixWeb.

Références

[Artsy86] Y. Artsy, H.Y. Chang, and R. Finkel. Processes migrate in Charlotte. Technical Report 655, Computer Sciences Department, University of Wisconsin-Madison, August 1986.

[Artsy89] Y. Artsy and R. Finkel. Designing a process migration facility: The Charlotte experience. *IEEE Computer*, 22(9), pp. 47-56, September 1989.

[AT&T] AT&T Laboratories Cambridge. OmniOrb2: Free High Performance CORBA2 ORB. http://www.uk.research.att.com/omniORB/omniORBPerformance.html. (dernière visite le : 10/5/2000).

[Badidi98a] E. Badidi, R.K. Keller, and V. Van Dongen. Vers une Architecture de Partage de Charge dans un Environnement CORBA. Actes des Rencontres francophones du parallélisme des architectures et des systèmes (RENPAR'10), Strasbourg, France, pp. 103--106, June 1998.

[Badidi98b] E. Badidi, R.K. Keller, P.G. Kropf, and V. Van Dongen. Dynamic Server Selection in Distributed Object Computing Systems. In Proc. of the Workshop on Distributed Computing on the WEB, Rostock, Germany, pp. 39-47, June 1998.

[Badidi98c] E. Badidi, R.K. Keller, P.G. Kropf, and V. Van Dongen. LoDACE: une architecture de partage de charge dans les systèmes distribués objet. Actes du Second Colloque International sur les NOuvelles TEchnologies de la REpartition (Notere'98), Montréal, Canada, pp. 281-296, Octobre 1998.

[Badidi99] E. Badidi, R.K. Keller, P.G. Kropf, and V. Van Dongen. The Design of a Trader-based CORBA Load Sharing Service. In Proc. of the Twelfth International Conference on Parallel and Distributed Computing Systems (PDCS'99), International Society for Computers and their Applications, Fort Lauderdale, Florida, pp.75-80, August 1999.

[Bakker97] A. Bakker, I. Kuz, and M. Van Steen. Towards a Taxonomy of Distributed Object Models. In Proc. of the Third Annual ASCI Conference, Heijen, The Netherlands, pp. 22-27, June 1997.

[Balter91] R. Balter, J. Bernadat, et al. Architecture and Implementation of Guide, an Object-Oriented Distributed System. *Computing Systems*, 4(1), pp. 31-67, April 1991.

[Banatre95] M. Banatre, Y. Belhamissi, V. Issarny, I. Puaut, and J.P. Routeau. Adaptive Placement of Method Executions within a Customized Distributed Object-Based Runtime System : Design, Implementation and Performance. In Proc. of the IEEE International Conference on Distributed Computing Systems, Vancouver, Canada, pp. 279-286, 1995.

[Barak85a] A. Barak and A. Litman. MOS: A multicomputer distributed operating system. *Software Practice and Experience*, 15(8), pp. 725-737, August 1985.

[Barak85b] A. Barak and A. Shiloh. A distributed load balancing policy for a multicomputer. *Software Practice and Experience*, 15(9), pp. 901-913, September 1985.

[Barak89] A. Barak. The evolution of the Mosix multi-computer UNIX system. Technical Report 89-17, The Hebrew University of Jerusalem, September 1989.

[Barak93] A. Barak, S. Guday, and R.G. Wheeler. *The MOSIX Distributed Operating System: Load Balancing for Unix*. Number 672 in Lecture Notes in Computer Science, Springer-Verlag, 1993.

[BEA] BEA System Inc. – Presentation de BEA ObjectBroker – http://www.beasys.fr/produits/objectbroker.htm. (dernière visite le : 10/05/2000).

[Bearman91] M. Bearman and K. Raymond. Federating Traders: an ODP adventure. In Proc. of the IFIP TC6/WG6.4 International Workshop on Open Distributed Processing, Berlin, Germany, North-Holland, pp. 125-141, October 1991.

[Bearman94] M. Bearman. ODP Trader. Open Distributed Processing, Eds. J. de Meer and B. Mahr and S. Storp, North-Holland, Vol. 2, pp. 19-33, 1994.

[Bearman97] M. Bearman, K. Duddy, K. Raymond and A. Vogel. Trader Down Under: Upside Down and Inside Out. *Journal of Theory and Practice of Object Systems* (TAPOS), Wiley, 3(1), pp. 15-30, 1997.

[Becker94] W. Becker, and G. Waldmann. Exploiting inter task dependencies for dynamic load balancing. In IEEE 3rd International Symposium on High Performance Distributed Computing, San Francisco, California, August 1994.

[Bershad85] B. Bershad. Load balancing with Maitre d'. Technical Report UCB/CSD 85/276, Computer Science Division, University of California, Berkeley, December 1985.

[Bonomi90] F. Bonomi and A. Kumar. Adaptive Optimal Load Balancing in a Nonhomogeneous Multiserver System with a Central Job Scheduler. *IEEE Transactions on Computers*, 39(10), pp.1232-1250, October 1990.

[Cabrera86] L. Cabrera. The influence of workload on load balancing strategies. In Summer USENIX Conference, Atlanta, Georgia, pp. 446-458, June 1986.

[Carter96] R.L. Carter and M.E. Crovella. Dynamic Server Selection Using Bandwidth Probing in Wide-Area Networks. Technical Report TR-96-007, Boston University Computer Science Department, March 1996.

[Casavant88a] T.L. Casavant and J.G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, 14(2), pp. 141-154, February 1988.

[Casavant88b] T.L. Casavant, and J.G. Kuhl. Effects of response and stability on scheduling in distributed computing systems. *IEEE Transactions on Software Engineering*, 14(11), pp. 1578-1588, November 1988.

[Caughey93] S.J. Caughey, G.D. Parrington, and S.K. Shrivastava. Shadows: A Flexible Support System for Objects in Distributed Systems. In Proc. of the Third International Workshop on Object Orientation in Operating Systems (IWOOOS-93), Ashville, North Carolina, December 1993. Also Available as Technical Report, ESPRIT Basic Research Project BROADCAST, Number BROADCAST#TR93-30, August 1993.

[Chatonnay96] P. Chatonnay, B. Herrmann, L. Philippe et F. Bourdon. Placement dynamique dans les systèmes répartis à objets. *Calculateur parallèle*, 8(1), pp. 11-30, 1996.

[Chatonnay98] P. Chatonnay. Gestion de l'allocation des ressources aux objets dans les systèmes répartis, une approche multicritère intégrant les communications. Thèse, Université de Franche-Comté, Janvier 1998.

[Chevalier95] P.Y. Chevalier, D. Hagimont, J. Mossière, and X. Rousset. Object Migration in the Guide System. BROADCAST TR No. 101, Bull-IMAG, 1995.

[Chin91] R.S. Chin and S.T. Chanson. Distributed Object-Based Programming Systems. *ACM Computing Surveys*, 23(1), pp. 91-124, 1991.

[Cisco] Cisco Systems, Inc. Cisco LocalDirector. http://www.cisco.com/univercd/cc/td/doc/pcat/ld.htm. (dernière visite le : 10/05/2000).

[Cisco2000] Cisco Systems, Inc. Cisco IP/TV 3400 Series Servers: Capacities, Performance Planning, and Controls. White paper, 2000. http://www.skystone.com/warp/public/cc/cisco/mkt/video/iptv/tech/ip34p_wp.pdf. (dernière visite le : 10/05/2000).

[Condor] Condor Team. Overview of the Condor High Throughput Computing System. http://www.cs.wisc.edu/condor/overview/. (dernière visite le : 10/05/2000).

[Dandamudi96a] S. Dandamudi and H. Hadavi. Performance Impact of I/O on Sender-Initiated and Receiver-Initiated Load Sharing Policies in Distributed Systems. School of Computer Science, Carleton University, Technical Report TR-96-23, 1996.

[Dandamudi96b] S. Dandamudi and M. Lo. Hierarchical Load Sharing Policies for Distributed Systems. Centre for Parallel and Distributed Computing, School of Computer Science, Carleton University, Technical Report SCS-96-1, January 1996.

[Devarakonda89] M.V. Devarakonda, and R.K. Iyer. Predictability of Process Resource Usage : A measurement-Based Study on UNIX. *IEEE Transactions on Software Engineering*, 15(12), pp. 1579-1586, December 1989.

[Douglis87] F. Douglis and J. Osterhout. Process migration in the Sprite operating system. In Proc. of the 7th IEEE International Conference Distributed Computing Systems, West Berlin, West Germany, pp. 18-25, September 1987.

[Douglis91] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software Practice and Experience*, 21(8), pp. 757-785, August 1991.

[Downey95] A.B. Downey and M. Harchol-Balter. A note on the Limited Performance Benefits of Migrating Active Processes for Load Sharing. Computer Science Division (EECS), University of California at Berkeley, Report No. UCB/CSD-95-888, November 1995.

[Eager86a] D. Eager, E. Lazowska, and J. Zahorjan. A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing. *Performance Evaluation*, 6(1), pp. 53-68, May 1986.

[Eager86b] D. Eager, E. Lazowska, and J. Zahorian. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, 12(5), pp. 662-674, May 1986.

[Eager88] D. Eager, E. Lazowska, and J. Zahorjan. The Limited Performance Benefits of Migrating Active Processes for Load Sharing. In Proc. of ACM SIGMETRICS, pp. 63-72, 1988.

[ExperSoft] ExperSoft Coporation. CorbaPlus: The Expersoft Difference: Per Invocation Load Balancing. http://www.expersoft.com/Products/TechAdv/Load_balance.htm. (dernière visite le : 10/5/2000).

[F5Networks] F5 Networks Inc. BIG/IP : Local high-availability, intelligent load balancing. http://www.f5.com/bigip/index.html. (dernière visite le : 10/5/2000).

[Felber96] P. Felber, B. Garbinato, and R. Guerraoui. The Design of a CORBA Group Communication Service. In Proc. of the 15th Symposium on Reliable Distributed Systems (SRDS-15), Niagara-on-the-Lake, Canada, pp. 150-159, October 1996.

[Felber97] P. Felber, R. Guerraoui, and A. Schiper. Replicating Objects using the CORBA Event Service. In Proc. of the 6th IEEE Computer Society Workshop on Future Trends in Distributed Computing Systems (FTDCS-6), Tunis, Tunisia, pp. 14-19, October 1997.

[Ferrari87] D. Ferrari, and S. Zhou. An Empirical Investigation of Load Indices for Load Balancing Applications. In Proc. of the Performance '87, 12th International Symposium on Computer Performance Modeling, Measurement and Evaluation, North-Holland, Amsterdam, pp. 515-528, 1987.

[Folliot93] B. Folliot. Méthodes et outils de partage de charge pour la conception et la mise en oeuvre d'applications dans les systèmes répartis hétérogènes. Thèse de doctorat de l'Université Pierre et Marie Curie, Avril 1993.

[Genias] Genias software. Codine Technical Description.
http://www.genias.de/products/codine/tech_desc.html. (dernière visite le : 10/05/2000).
http://www.genias.de/products/codine/overview.html. (dernière visite le : 10/05/2000).

[Gokhale96] A. Gokhale and D.C. Schmidt. Measuring the Performance of Communication Middleware on High-Speed Networks. In Proc. of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, ACM SIGCOMM Computer Communication Review, 26(4), pp. 306-317, August 1996.

[Gokhale97] A. Gokhale and D.C. Schmidt. Evaluating Latency and Scalability of CORBA Over High-Speed ATM Networks. In Proc. of the International Conference on Distributed Computing Systems '97, IEEE, Baltimore, Maryland, pp. 401-410, May 1997.

[Gourhant92] Y. Gourhant, S. Louboutin, V. Cahill, A. Condon, G. Starovic, and B. Tangney. Dynamic Clustering in an Object-Oriented Distributed System. In Proc. of the Objects in Large Distributed Applications Workshop {OLDA-II}, Ottawa, Canada, October 1992.

[Guerraoui96] R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. In Reliable Software Technologies - Ada-Europe'96, LNCS 1088, Springer-Verlag, pp. 38-57, June 1996.

[Hac90] A. Hac, and T.J. Johnson. Sensitivity Study of the Load Balancing Algorithm in a Distributed System. *Journal of Parallel and Distributed Computing*. Vol. 10, pp. 85-89, 1990.

[Hagmann86] R. Hagmann. Process server: Sharing processing power in a workstation environment. In Proc. of the 6th International Conference on Distributed Computing Systems, Cambridge, Massachusetts, pp. 260-267, May 1986.

[Harchol-Balter96] M. Harchol-Balter and A.B. Downey. Exploiting process lifetime distributions for dynamic load balancing. In Proc. of ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, Philadelphia, Pennsylvania, pp. 13-24, May 1996.

[Hwang82] K. Hwang, W. Croft, G. Goble, B. Wah, F. Briggs, W. Simmons, and C. Coates. A UNIX-based local computer network with load balancing. *IEEE Computer*, 15(4), pp. 55-66, 1982.

[IBM] IBM Inc. IBM secureWay Network dispatcher. http://www.software.ibm.com/network/dispatcher. (dernière visite le : 10/05/2000).

[IBM96] IBM Corporation. *Using and Administering LoadLeveler-- Release 3.0*. Document Number SC23-3989-00. August, 1996.

[IETF97] IETF, Network Working Group. Service Location Protocole. Request for Comments: 2165, June 1997.

[Inprise98] Inprise Corporation. *VisiBroker for Java version 3.3 Programmer's Guide*. 1998.

[Iona96] Iona Technologies Ltd. *OrbixWeb programmer guide*. 1996.

[Iona97] Iona Technologies PLC. *OrbixTrader Programmer's Guide and Reference*. August 1997.

[ISO96a] ITU/ISO . Reference Model of Open Distributed Processing -Part 1 : Overview. ISO/IEC 10746-1, ITU-T Rec. X901, 1996.

[ISO96b] ITU/ISO. ODP Trading Function Part 1: Specification. ISO/IEC 2nd DIS 13235-1, ITU-T Draft Rec. X950-1, 1996.

[Jacquemot94] C. Jacquemot, F. Herrmann, P.S. Jensen, P. Gautron, J. Mukerji, H.G. Baumgarten, H. Hartlage. COOL: The CHORUS CORBA Compliant Framework. In Proc. of Spring COMPCON 94, San Francisco, California, February 1994.

[Jensen96] C.D. Jensen. Fine-Grained Load Distribution in Object Based Systems, An Experiment with Grain Sizes in Guide-2. *Calculateurs Parallèles*, 8(1), 1996.

[Jul87] E. Jul, H. Levy, N. Hutchinson and A. Black. Fine-grained mobility in the Emerald system. In Proc. of the Eleventh ACM Symposium on Operating System Principles, pp. 105-106, November 1987.

[Jul89] E. Jul. Object Mobility in a Distributed Object-Oriented System. Ph.D. thesis, University of Washington, 1989.

[Jul93] E. Jul. Separation of Distributed Objects. In Proc. of the ECOOP'93 Workshop on object-based Distributed Programming. Eds. Rachid Guerraoui, Oscal Nierstrasz, and Michel Riveill, Germany, pp. 47-54, July 1993.

[Karpovich96] J.F. Karpovich. Support for Object Placement in Wide Area Heterogeneous Distributed Systems. Technical Report CS-96-03, University of Virginia, 1996.

[Kremien92] O. Kremien, and J. Kramer. Methodical analysis of adaptive load sharing algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 3(6), pp. 747-760, November 1992.

[Krueger87] P. Krueger and M. Livny. The Diverse Objectives of Distributed Scheduling Policies. In Proc. of the 7th International Conference on Distributed Computing Systems, IEEE, pp.242-249, September 1987.

[Kumar89] A. Kumar. Adaptive Load Control of the Central Processor in a Distributed System with a Star Topology. *IEEE Transactions on Computers*, 38(11), pp. 1502-1512, November 1989.

[Kunz91] T. Kunz. The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme. *IEEE Transactions on Software Engineering,* 17(7), pp. 725-730, July 1991.

[Kutvonen95] L. Kutvonen. Achieving Interoperability through ODP Trading Function. The Second International Symposium on Autonomous Decentralized systems (ISADS'95). Arizona, Publications of the IEEE Computer Society, pp. 63-69, April 1995.

[Lazowska84] E.D. Lazowska, J. Zahorjan, G.S. Graham, K.C. Sevcik. *Quantitative System Performance.* Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[Leland86] W.E. Leland and T.J. Ott. Load-balancing heuristics and process behavior. ACM Sigmetrics, Vol. 14, pp. 54-69, May 1986.

[Lewis95] M.J. Lewis, A. Grimshaw. The Core Legion Object Model. In Proc. of the Fifth IEEE International Symposium on High Performance Distributed Computing, IEEE Computer Society Press, Los Alamitos, California, August 1996. Also Available as Technical Report CS-95-35, August 1995.

[Lin87] F.C.H Lin, and R.M. Keller. The Gradient Model Load Blancing Method. *IEEE Transactions on Software Engineering*, 13(1), pp. 32-38, January 1987.

[Litzkow88] M.J. Litzkow, M. Livny, and M.W. Mutka. Condor : a hunter of idle workstations. In Proc. of the 8th International Conference on Distributed Computing Systems, IEEE, San Jose, California, pp.104-111, June 1988.

[Litzkow92] M.J. Litzkow and M. Solomon. Supporting checkpointing and process migration outside the UNIX kernel. In USENIX Winter Conference, San Francisco, California, pp. 283-290, January 1992.

[Lo88] V.M. Lo. Heuristic Algorithms for Task Assignment in Distributed Systems. *IEEE Transactions on Computers*, 37(11), pp. 1384-1397, November 1988.

[Maffeis96] S. Maffeis. The Object Group design Pattern. In Proc. of the 1996 Usenix Conference on Object-Oriented Technologies, Toronto, Canada. Usenix, June 1996. Also Available as Technical Report, Cornell University, Computer Science, Number TR96-1570, February 1996.

[Microsoft96] Microsoft Corporation. Microsoft Windows NT Server, DCOM Technical Overview. White Paper, 1996.

[Mirchandaney89a] R. Mirchandaney, D. Towsley, and J.A. Stankovic. Adaptive Load Sharing in Heterogeneous Systems. In Proc. of the 9th IEEE International Conference on Distributed Computing Systems, IEEE CS Press, Los Alamitos, California, pp. 298-306, 1989.

[Mirchandaney89b] R. Mirchandaney, D. Towsley, and J.A. Stankovic. Analysis of the effects of Delays on Load Sharing. *IEEE Transactions on Computers*, 38(11), pp. 1513-1525, November 1989.

[Mni85] L. Mni and K. Hwang. Optimal Load balancing in a Multiple Processor System with many classes. *IEEE Transactions on Software Engineering*, 11(5), pp. 491-496, 1985.

[Muller96] S. Muller, K. Muller-Jones, W. Lamersdorf, T. Tu. Global Trader Cooperation in Open Service Markets. In Proc. of the International Workshop on Trends in Distributed Systems, Aachen, Springer, October 1996.

[Mutka87] M.W. Mutka and M. Livny. Scheduling remote processing capacity in a workstation-processor bank computing system. In Proc. of the 7th International Conference on Distributed Computing Systems, Berlin, pp. 2-9, September 1987.

[Mutka91] M. W. Mutka and M. Livny. The available capacity of a privately owned workstation environment. *Performance Evaluation*, 12, pp. 269-284, 1991.

[Nuttall94] M. Nuttall. Survey of systems providing process or object migration. Technical Report DoC 94/10, Imperial College, London, May 1994.

[OMG96] Object Management Group. RFP5 Submission Trading Object Service. OMG Document orbos/960506, OMG, Framingham, Massachusetts, 1996.

[OMG97a] Object Management Group. CORBAservices: Common Object Services Specification. Updated version, July 1997.

[OMG97b] Object Management Group. The Common Object Request Broker: Architecture and Specification. Revision 2.1, August 1997.

[OMG97c] Object Management Group. A discussion of the Object Management Architecture. January 1997.

[OMG98] Object Management Group. The Common Object Request Broker: Architecture and Specification. 2.2 edition, February 1998.

[Orfali95] R. Orfali and D. Harkey. Client/Server with Distributed Objects. Byte Magazine, pp. 151-164, April 1995.

[Orfali96] R. Orfali, D. Harkey and J. Edwards. *The Essential Distributed Objects Survival Guide.* John Wiley & Sons Inc., 1996.

[Osser92] W. Osser. Automatic Process Selection for Load Balancing. Master thesis, University of California, Santa Cruz, June 1992.

[Ousterhout88] J.K. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2), pp. 23-36, February 1988.

[Ozden93] B. Ozden, A. Goldberg, and A. Silberschatz. Scalable and non-intrusive load-sharing in owner-based distributed systems. In 5th IEEE Symposium on Parallel and Distributed Processing, Dallas, Texas, pp. 690-701, December 1993.

[Platform94] Platform Computing Corporation. LSF: Load Sharing Facility Administrator's Guide. Technical Report, December 1994.

[Pulidas87] S. Pulidas, D. Towsley, and J.A. Stankovic. Design of efficient parameter estimators for decentralized load balancing policies. Technical Report, University of Massachusetts, Amherst, Computer Science, Number UM-CS-1987-079, August 1987.

[Rackl97] G. Rackl. Load Distribution for CORBA Environments. Diploma Thesis, Technische Universität München, 1997.

[Rajendra91] K.R. Rajendra, E.D. Tempero, H.M. Levy, A.P. Black, N.C. Hutchinson, and E. Jul. Emerald: A General-Purpose Programming Language. *Software Practice and Experience* 21(1), pp. 91-118, 1991.

[Riedl96] R. Riedl, and L. Richter. Classification of load distribution algorithms. In 4th EuroMicro Workshop on Parallel and Distributed Processing. Braga, Portugal, IEEE Computer Society Press, pp. 404-413, January 1996.

[Rosenberry92] W. Rosenberry, D. Kenney, and G. Fisher. *Understanding DCE*. O'Reilly & Associates, 1992.

[Ryou93] J.C. Ryou, and J.Y. Juang. An efficient load balancing algorithm in distributed computing systems. In 5th IEEE Symposium on Parallel and Distributed Processing. Dallas, Texas, IEEE Computer Society Press, pp. 233-240, December 1993.

[Schiemann96a] B. Schiemann. Requirement for an Interface Definition Language Environment for Load Balancing (Second Draft). ESPRIT III P8144, LYDIA/WP.4/T.4.1/D5, June 1996.

[Schiemann96b] B. Schiemann. Specification of IDL Mechanisms for Load Balancing (First Draft). ESPRIT III P8144, LYDIA/WP.4/T.4.2/D6, July 1996.

[Schmidt95] D.C. Schmidt and S. Vinoski. Object Interconnections, Introduction to Distributed Object Computing (Column 1). *C++ Report Magazine*, SIGS, 7(1), January 1995.

[Schmidt98] D.C. Schmidt. Evaluating Architectures for Multithreaded Object Request Brokers. *Communications of the ACM*, 41(10), pp. 54-60, October 1998.

[Schnekenburger97] T. Schnekenburger and G. Rackl. Implementing Dynamic Load Distribution Strategies with Orbix. In Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97), Las Vegas, Nevada, Volume II, CSREA, ISBN 0-9648666-6-8, pp. 996-1005, 1997.

[Shirazi90] B. Shirazi, M. Wang, and G. Pathak. Analysis and Evaluation of Heuristic Methods for Static Task Scheduling. *Journal of Parallel and Distributed Computing*, Vol. 10, pp. 222-232, 1990.

[Shivaratri92] N. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *IEEE Computer*, 25(12), pp. 33-44, December 1992.

[Svensson90] A. Svensson. History, an Intelligent Load Sharing Filter. In Proc. of the 11th IEEE International Conference on Distributed Computing Systems, IEEE CS Press, Los Alamitos, California, pp. 546-553, 1990.

[Tangney92] B. Tangney and A. Condon. Some Issues in Load Balancing in Amadeus. In Proc. of the ECOOP'92 Workshop on Load balancing in Object Oriented Systems, 1992.

[Theimer85] M. Theimer, K.A. Lantz, and D.R. Cheriton. Preemptable remote execution facilities for the V-System. In Proc. of the 10th ACM Symposium on Operating System Principles, pp. 2-12, December 1985.

[Theimer89] M. Theimer, and K. Lantz. Finding idle machines in a workstation based distributed system. *IEEE Transactions on Software Engineering*, 15(11), pp. 1444-1458, November 1989.

[Ullman75] J. Ullman. NP-Complete Scheduling Problems. *Journal of Computer and System Sciences*, Vol. 10, pp. 384-393, 1975.

[Vinoski97] S. Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 35(2), pp. 51-55, February 1997.

[Vinoski98] S. Vinoski. New Features for CORBA 3.0. *Communications of the ACM*, 41(10), pp. 44-52, October 1998.

[Waldspurger92] C.A. Waldspurger, T. Hogg, B.A. Huberman, J.O Kephart, and W.S. Stornetta. Spawn : A Distributed Computational Economy. *IEEE Transactions on Software Engineering*, 18(2), pp. 103-117, February 1992.

[Yoshikawa97] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D.Culler. Using Smart Clients to Build Scalable Services. In Proc. of the 1997 USENIX Annual Technical Conference, Anaheim, California, 22(2) , pp. 105-117, April 1997.

[Zhou87] S. Zhou. Performance studies of dynamic load balancing in distributed systems. Technical Report UCB/CSD 87/376, Computer Science Division (EECS), University of California, Berkeley, California 94720, October 1987.

[Zhou88] S. Zhou. A trace driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering*, 14(9), pp. 1327-1341, September 1988.

[Zhou93] S. Zhou, X. Zheng, J. Wang, and P. Delisle. Utopia: a load sharing facility for large, heterogeneous distributed computer systems. *Software Practice and Experience*, 23(12), pp. 1305-1336, December 1993.

Annexe 1
L'architecture CORBA

Cette annexe donne une brève description du standard CORBA de l'*Object Management Group (OMG)*. La spécification de ce standard a évolué de la version CORBA 1.0 à CORBA 2.0, à CORBA 2.1, CORBA 2.2, à CORBA 2.3, et présentement à CORBA 3.0. Nous décrivons dans cette annexe plus particulièrement la version 2.1 pour laquelle il y avait des ORBs lorsque nous avons entamé ce travail de thèse. Les versions ultérieures restent compatibles avec cette version tout en ajoutant d'autres fonctionnalités. Plus d'informations sur ce standard est disponible dans [OMG97a,OMG97b,OMG98, Vinoski98].

A1.1 L'Object Management Group (OMG)

L'Object Management Group (OMG) est une organisation internationale composée de plus de 750 membres, comprenant des manufacturiers de matériel informatique, des éditeurs de logiciels, des institutions universitaires, et des particuliers. L'OMG a été fondé en 1989 pour promouvoir, d'une part, la pratique de la technologie objet dans le développement du logiciel et, d'autre part, pour résoudre les problèmes apparaissant dans le développement de grands logiciels. Il a pour mission d'établir des spécifications de gestion objet et des directives pour l'industrie dans ce domaine.

Étant donné que les applications informatiques sont de plus en plus complexes, les approches traditionnelles utilisées dans la production de logiciel ne sont plus suffisantes pour développer des programmes de manière efficace. La complexité de plusieurs applications est aussi due à l'utilisation accrue de systèmes très hétérogènes, i.e., des systèmes produits par différents éditeurs et s'exécutant au-dessus de systèmes d'exploitation différents.

Pour résoudre ces problèmes, l'OMG propose un cadre pour le développement du logiciel basé sur deux paradigmes : le paradigme de programmation orientée objet et le calcul distribué. La technologie objet a prouvé ses capacités à produire de grands logiciels. En effet, elle représente une technique efficace permettant d'améliorer la réutilisation, la portabilité, et l'interopérabilité entre les produits logiciels. D'autre part, le calcul distribué représente un élément important dans le développement des systèmes d'information futurs. Ainsi, depuis sa création l'OMG a concentré ses efforts à la définition d'un cadre de développement d'applications distribuées objet. Pour cela, il a établi une architecture nommée *Object Management Architecture (OMA)* qui définit l'infrastructure conceptuelle sur laquelle sont basées toutes les spécifications qu'il a produites jusqu'à présent et les spécifications futures.

A1.2 L'architecture OMA

L'architecture OMA définit un modèle général pour les systèmes distribués orientés objet [OMG97c]. Elle spécifie les directives techniques générales qui doivent être suivies par chaque composante du système. Cette architecture est composée d'un *modèle objet* commun à toutes les composantes et d'un *modèle de référence*.

A1.2.1 Modèle objet

Le modèle objet OMG introduit les concepts de base caractérisant les objets d'un système et permettant d'avoir une sémantique commune à tous les objets. Ce modèle définit les objets pour représenter des entités ayant un état et un comportement. Le comportement d'un objet est implémenté au moyen d'opérations qui lui sont appliquées. D'autre part, chaque objet est une instance d'un type donné. Des relations d'héritage peuvent être définies entre les types [Vinoski97].

Un aspect important du modèle objet est son indépendance de toute implémentation. Ainsi, les types des objets sont définis en termes de types d'interfaces à l'aide d'un *langage de définition d'interface* (Interface Definition Language -IDL-). Ces définitions décrivent les opérations offertes par un type donné d'objet indépendamment des aspects d'implémentation. Les clients envoient des requêtes aux objets pour recevoir un certain service offert par ces objets. L'implémentation et la localisation de chaque objet sont cachées au client. Ceci permet l'interopérabilité entre les composantes s'exécutant dans différentes plates-formes. Les objets

peuvent être réalisés dans différents langages de programmation tels que C, C++, Smalltalk, Ada, et Java.

A1.2.2 Modèle de référence

Le modèle de référence OMA identifie et classifie les différents types d'objets constituant un système distribué objet. La figure A1.1 schématise les composantes intervenantes dans ce modèle. Ces composantes sont : *le bus objet* (Object Request Broker Architecture -ORB-), *les services objets* (Object Services), *les utilitaires communs* (common facilities), *les interfaces de domaine* (domain interfaces), et *les interfaces d'applications* (application interfaces).

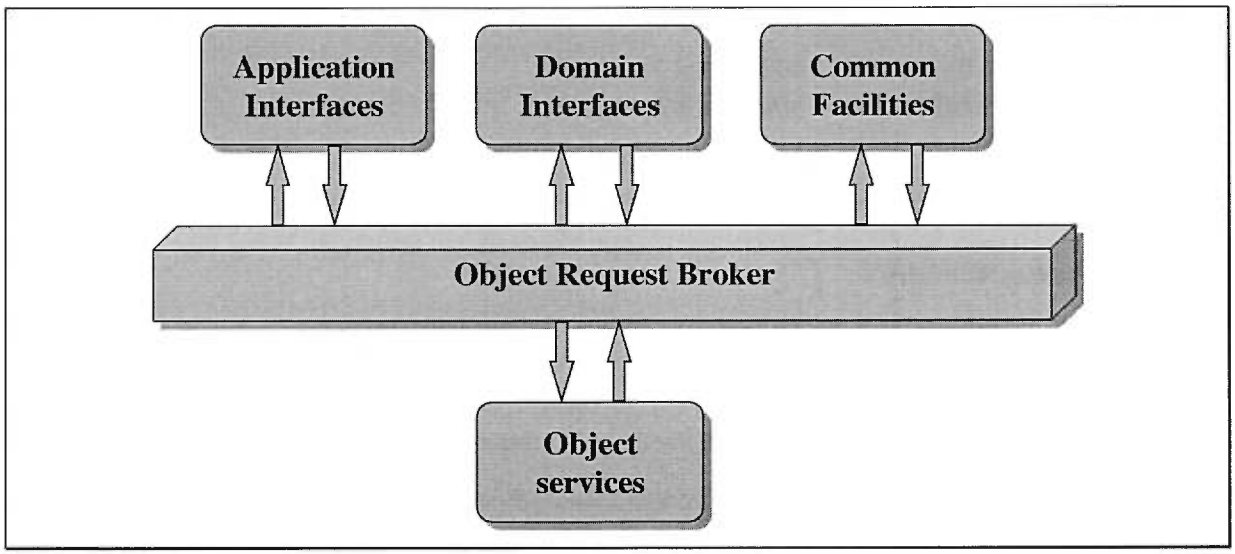


Figure A1.1 - Architecture OMA

A1.2.2.1 Bus objet (Object Request Broker (ORB))

Le bus objet est la composante la plus importante du modèle. Sa fonction est de faciliter la communication entre les objets d'un système distribué. Elle permet aux clients d'émettre des requêtes aux objets pour obtenir un certain service. Le client envoie des requêtes de façon indépendante de la localisation, du langage, et de la plate-forme de l'objet serveur. Par conséquent, l'ORB est la première composante permettant l'interopérabilité et la portabilité des applications.

A1.2.2.2 Services objet (Object services)

Ce sont des interfaces indépendantes du domaine d'application qui sont utilisées par plusieurs programmes distribués. Par exemple, un service permettant la découverte d'autres services disponibles est presque toujours nécessaire indépendamment du domaine d'application. Deux exemples de services objet qui accomplissent ce rôle sont : *le service de nommage* (naming service) qui permet aux clients de trouver des objets en se basant sur leurs noms, et *le service de courtage* qui permet au client de trouver des objets en se basant sur leurs propriétés. D'autres services pour la gestion de la sécurité, des transactions, et d'événements ont été spécifiés et standardisés par l'OMG.

A1.2.2.3 Utilitaires communs (Common facilities)

Les utilitaires communs sont des collections de composantes, définies en IDL, qui offrent des services directement aux applications de l'utilisateur final. Ils définissent des règles permettant aux composantes des usagers de collaborer efficacement [OMG97c,Orfali96]. Comme exemples, citons :

- Les services d'édition similaires à ceux offerts par OpenDoc[13] et OLE2[14].
- Les services de gestion de l'information tels que le stockage des documents composés et les mécanismes d'échange de données similaires à ceux offerts par OpenDoc et OLE2.
- Les services de gestion des systèmes définissant des interfaces permettant de gérer, instrumenter, configurer, installer, opérer, et réparer les composantes distribuées orientées objet.

A1.2.2.4 Interfaces de domaines (Domain Interfaces)

Ces interfaces ont le même rôle que les services objets et les utilitaires communs, mais sont orientées vers des domaines d'application spécifiques. Par exemple, l'OMG a émis des RFPs (Request for Proposal) dans le domaine des télécommunications, dans le domaine médical, et dans le domaine financier.

---

[13] OpenDoc est le standard du CIL (*Component Integration Laboratory*) pour les documents composés.
[14] OLE2 est le standard de *Microsoft* en matière de documents composés.

A1.2.2.5 Interfaces d'applications (Application Interfaces)

Ce sont des interfaces développées spécifiquement pour une application donnée. Ces interfaces ne sont pas standardisées par OMG puisqu'elles sont spécifiques aux applications.

A1.3 Common Object Request Broker Architecture (CORBA)

CORBA est l'une des premières spécifications de OMG. Elle détaille les interfaces et les caractéristiques de l'ORB. La figure A1.2 illustre les composantes de CORBA introduites dans la spécification CORBA.

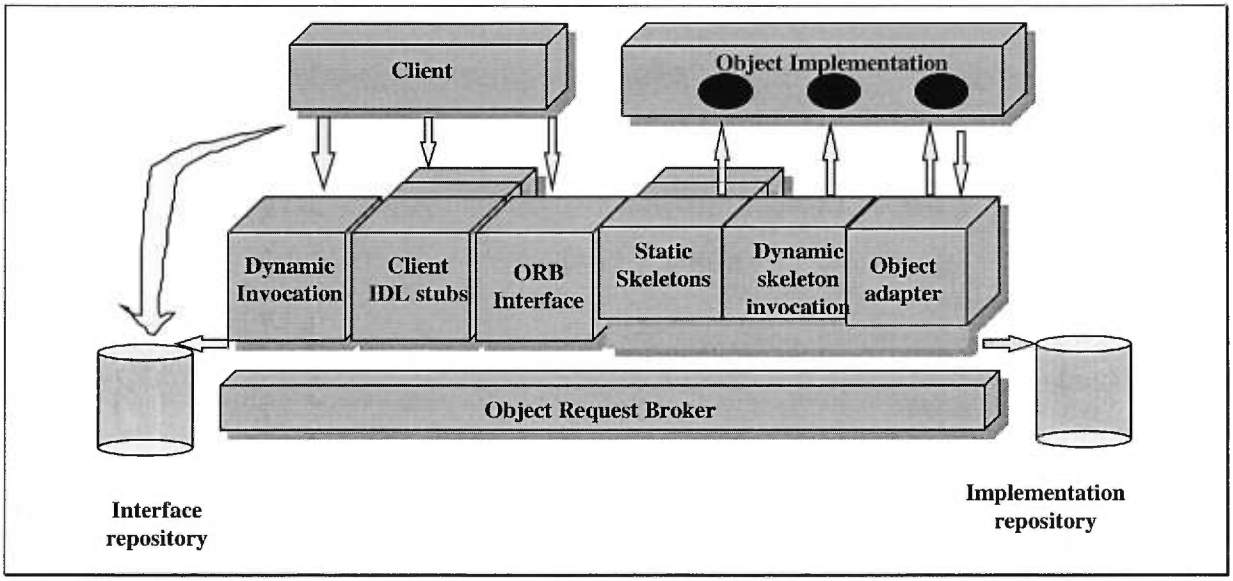


Figure A1.2 - Architecture CORBA

A1.3.1 Bus objet (ORB)

L'ORB (Object Request Broker) constitue le noyau de l'architecture CORBA. Il définit le support grâce auquel un client peut invoquer les services d'un objet dans un environnement distribué hétérogène. Le but de l'ORB est de fournir un certain nombre de services au client, pour faciliter l'invocation d'une opération sur un objet en faisant abstraction de la distance et de la localisation, mais aussi de l'implémentation de l'objet serveur.

Pour invoquer un service, le client n'a besoin de connaître que le nom de l'objet serveur, son type et l'opération à invoquer. Les problèmes liés à la localisation de l'objet, à sa disponibilité, et à la gestion d'autres services comme la sécurité ou l'atomicité, sont gérés par l'ORB. Ceci

constitue un progrès important sur les mécanismes existants en matière d'appel de services distants comme les RPCs.

Le client n'a pas également à se préoccuper de l'état de l'objet serveur. En effet, quand un client adresse une requête à un objet serveur, il n'a pas besoin de savoir si l'objet serveur est actif et prêt à recevoir des requêtes ou non. L'ORB démarre l'objet serveur si nécessaire avant de lui délivrer la requête.

A1.3.2 Langage de définition d'interfaces (IDL)

Pour décrire les interfaces des objets CORBA, l'OMG a défini le langage IDL qui est fortement inspiré de C++ dans sa syntaxe. L'interface fournit au client les spécifications des opérations invoquables sur l'objet serveur. La spécification des interfaces est la première étape de l'implémentation d'une application objet distribuée. L'IDL doit permettre, dans le processus de développement, une mise en correspondance rapide entre le modèle objet de l'application et son implémentation dans le langage cible. Le langage IDL a été défini à partir des propositions de la norme ANSI de C++, et enrichi de quelques caractéristiques propres aux mécanismes d'invocations des objets distribués. L'IDL ne permet de spécifier que les interfaces des objets. C'est pourquoi il n'inclut aucune structure algorithmique et ne permet pas la définition de variables.

Un exemple d'interface définie au moyen de l'IDL est :

```
interface compte-bancaire {
        attribute readonly double balance;
        void dépôt(double montant);
        void retrait(double montant);
        double balance();
};
```

Une caractéristique importante de l'IDL de OMG est son indépendance de tout langage. Comme c'est un langage déclaratif, et non un langage de programmation, les interfaces sont définies indépendamment des implémentations. Ceci permet aux objets d'être construits en utilisant différents langages de programmation et de communiquer les uns avec les autres. Les interfaces indépendantes de tout langage de programmation sont importantes dans les

systèmes hétérogènes, car tous les langages de programmation ne sont pas supportés ou disponibles sur toutes les plates-formes.

L'IDL offre un ensemble de types qui sont similaires à ceux trouvés dans plusieurs langages de programmation. Il offre des types de base tels que *long*, *double* et *boolean*, et des types construits tels que *struct*, *union*, *sequence* et *string*. Les types sont utilisés pour préciser les types des paramètres et des valeurs de retour des opérations. Comme vu dans l'exemple ci-dessus, les opérations sont employées dans des interfaces pour préciser les services offerts par les objets qui supportent ce type particulier d'interface.

A1.3.3 Traduction de l'IDL vers d'autres langages

Comme indiqué ci-dessus, l'IDL est seulement un langage déclaratif, et non un langage complet de programmation. Ainsi, il n'offre pas de caractéristiques comme les structures de contrôle et il n'est pas utilisé directement pour réaliser des applications distribuées. En effet, les traductions de langage (language mapping) déterminent comment les caractéristiques de l'IDL sont converties en concepts d'un langage de programmation. L'OMG a standardisé jusqu'à présent les transformations de l'IDL vers les langages C, C++, Smalltalk, Ada 95, Java, et COBOL.

A1.3.4 Répertoire d'interfaces (Interface Repository)

Le répertoire d'interfaces (IR) représente la composante de l'architecture OMA qui permet le stockage des définitions IDL des interfaces des objets, des méthodes qu'ils supportent et des paramètres qu'ils nécessitent (signatures des méthodes). Il constitue une base de données des interfaces d'objets. L'IR fournit l'information nécessaire à l'émission de requêtes utilisant l'Interface d'Invocation Dynamique (Dynamic Invocation Interface) de la figure A1.2.

A1.3.5 Talons clients et talons serveurs (client IDL stubs and Static skeletons)

L'étape qui suit la définition des interfaces en IDL est la génération des modules d'interaction avec l'ORB correspondant aux interfaces spécifiées. En effet, la spécification IDL est purement abstraite et n'intègre aucun élément propre aux communications client-serveur via

l'ORB. Ces éléments sont intégralement générés par la compilation des interfaces dans le langage cible. À cet effet, les solutions de développement CORBA intègrent systématiquement un compilateur IDL permettant de générer les fichiers nécessaires à l'implémentation du client et du serveur et qui sont les stubs et les skeletons de la figure A1.2.

Les talons serveur (skeletons) permettent le traitement des requêtes provenant des talons client (stubs) ou de l'interface d'invocation dynamique. L'adaptateur d'objets (Object Adapter) délègue aux talons serveur l'invocation des méthodes de l'objet, le décodage des requêtes CORBA, le codage des résultats de la requête, et le choix du bon "code" pour traiter la requête.

A1.3.6 Interface ORB (ORB Interface)

Cette interface est directement accessible à la fois par le client et par l'objet serveur. Elle regroupe un ensemble d'opérations et de fonctionnalités de l'ORB qui sont communes à tous les objets, telles que la conversion des références objet en chaînes de caractères ou encore la copie d'objet. Elle ne permet pas à proprement parler d'invoquer des services. Elle s'occupe de la gestion des objets dans le système.

A1.3.7 Adaptateur objet (Object Adapter)

Dans un environnement hétérogène, les implémentations des objets serveurs sont fortement liées au système hôte. La diversité de cet environnement oblige à définir une interface entre le noyau de l'ORB et l'implémentation. Cette interface est appelée Adaptateur Objet (Basic Object Adapter ou BOA). À travers l'adaptateur objet, l'implémentation de l'objet peut accéder aux services offerts par l'ORB.

La première invocation d'un objet se traduit au niveau de l'ORB et de l'adaptateur objet par :

1. L'activation de l'implémentation de l'objet serveur, c'est-à-dire le lancement du programme qui correspond à l'implémentation de l'objet.
2. L'implémentation activée s'enregistre auprès de l'adaptateur objet en signalant sa disposition à traiter ou non la requête.
3. L'objet serveur est ensuite activé par l'adaptateur objet.

4. L'adaptateur objet invoque enfin la méthode correspondante à la requête.

A1.4 Interopérabilité

Avec l'apparition des techniques d'Internet dans l'entreprise à travers ce qu'il est convenu d'appeler l'Intranet, l'OMG a dû introduire rapidement les nouvelles spécifications de la version 2.0 de CORBA visant à définir les techniques d'interopérabilité entre différents ORB. CORBA fait l'objet de nombreuses implémentations sur de nombreux systèmes. Ces implémentations ne sont pas directement compatibles et interopérables. Avec *GIOP* (General Inter ORB Protocol), l'OMG pose les règles d'interactions entre ORBs qui permettent par exemple à un client CORBA d'invoquer un objet serveur sur une autre plate-forme propriétaire différente de celle du client. Les possibilités offertes par l'interopérabilité élargissent le champ d'application de CORBA en permettant à tout client d'invoquer tout objet quels que soient sa localisation, son implémentation, et le système hôte. Dans le cas de l'Internet, un protocole spécifique appelé *IIOP* (Internet Inter ORB Protocol) a été introduit par la spécification CORBA pour permettre l'interopérabilité entre ORBs. Par extension, l'interopérabilité doit prendre en considération les composantes Microsoft COM et fournir les possibilités d'interactions avec l'environnement distribué DCOM. Plus généralement, l'interopérabilité est destinée à étendre l'invocation des objets serveurs d'un ORB vers d'autres ORBs, voire d'autres systèmes, en élargissant la portée de ces objets.

A1.4.1 GIOP (General Inter ORB Protocol)

GIOP définit une représentation standard des données dans les communications entre ORBs, ainsi qu'un ensemble limité de messages possibles entre les ORBs. Il est conçu pour s'adapter à tout type de protocole orienté connexion. Il se veut fiable, simple, portable, dans un souci de performance. GIOP peut être représenté par plusieurs protocoles propriétaires optimisés pour des besoins particuliers, chacun de ces protocoles pouvant être dans un deuxième temps interopérable à travers des ponts spécifiques (bridges).

Pour illustrer ceci, prenons l'exemple d'une compagnie mondiale disposant de systèmes CORBA hétérogènes. Au sein de chacune des unités de cette compagnie peuvent fonctionner plusieurs ORBs interopérables via un protocole IOP -Inter ORB Protocol- optimisé pour l'environnement et les traitements locaux à l'unité. Un tel protocole est appelé *ESIOP*

(Environnement Specific Inter ORB Protocol) qui s'appuie sur les RPC de DCE (Distributed Computing Environnement) [Rosenberry92]. Grâce à IIOP, les différentes unités deviennent interopérables.

A1.4.2 IIOP (Internet Inter ORB Protocol)

L'utilisation croissante de l'Internet à travers le monde et le succès grandissant de l'Intranet assurent à IIOP un avenir prometteur. De nombreux systèmes d'information existants sont appelés à migrer ou à intégrer les techniques du Web sous peine de devenir très rapidement obsolètes. À ces changements correspondent inévitablement des évolutions vers les technologies objets et vers l'architecture CORBA. De plus, le protocole HTTP montre ses limites pour une utilisation professionnelle. HTTP est en effet un protocole orienté document destiné au transport de pages HTML sur l'Internet. Ses carences notamment en termes de sécurité le condamnent inévitablement à se limiter à un rôle de transporteur d'hypertexte. Dans ce contexte, IIOP est traditionnellement décrit comme le successeur d'HTTP pour le transport de données. Déjà certains efforts sont faits pour intégrer ce protocole dans les navigateurs Web, si bien qu'à terme, tout client Web pourra invoquer tout type d'objet CORBA à travers le réseau.

Contrairement à HTTP, IIOP fonctionne en mode connecté, un mode dont ne peuvent se passer les applications de types transactionnelles, elles-mêmes appelées à se développer considérablement au-dessus de l'Internet.

Annexe 2
Le Service de courtage

Cette annexe décrit le service de courtage utilisé dans l'architecture LoDACE et dans le service LSS, pour permettre la découverte dynamique des serveurs.

A2.1 Aperçu du service de courtage

L'état actuel des systèmes distribués se caractérise par une très grande hétérogénéité au niveau des architectures matérielles et logicielles, et des environnements réseau. Pour pouvoir utiliser les divers services d'un système distribué, les utilisateurs ont besoin d'être au courant des fournisseurs potentiels de services et d'être capables d'accéder aux services offerts. De plus, le changement assez fréquent dans les localisations et les versions des services dans les grands systèmes distribués fait que l'association tardive (late binding), lors de l'exécution, entre les utilisateurs de service et les fournisseurs de services est une caractéristique fort souhaitable. Pour permettre l'association tardive, il faut disposer de mécanismes permettant de localiser et d'accéder dynamiquement aux services [Bearman97]. L'association tardive a l'avantage de permettre l'évolution flexible du système. En outre, elle aide aussi à améliorer la disponibilité et la tolérance aux fautes dans le système, car les objets peuvent dynamiquement sélectionner le fournisseur de service avec lequel ils souhaitent communiquer [Kutvonen96].

Le service de courtage (trading service) est un service permettant de dynamiquement découvrir les services offerts. Dans un système ouvert distribué, l'objet qui réalise le service de courtage est appelé *courtier* (trader). Un courtier est un mécanisme pour annoncer et découvrir des services. Il communique avec les serveurs (ou fournisseurs de service), avec les clients (utilisateurs de services), et avec d'autres courtiers [Bearman94,ISO9b,OMG96]. Dans le cas des réseaux à base du protocole IP, le Service Location Protocol (SLP) standardisé par le IETF est l'équivalent du service de courtage.

La figure A2.1 illustre le principe de fonctionnement de ce service de courtage. Quand un serveur désire annoncer son service, il enregistre son offre de service auprès du courtier. Une offre de service contient un type de service, un identificateur de l'interface du service, les valeurs des propriétés du service et les propriétés de l'offre de service. Pour un service d'impression par exemple, on peut avoir comme propriétés de service : le coût, la vitesse, et la qualité d'impression. Des exemples de propriétés d'offre de service sont le temps d'expiration et le propriétaire de l'offre de service. Le processus d'annonce d'un service est appelé *exportation*, et l'objet qui enregistre le service avec le courtier est appelé *exportateur*. Les offres de service sont enregistrées par le courtier dans sa base de données. Un exportateur peut retirer ou remplacer son offre de service.

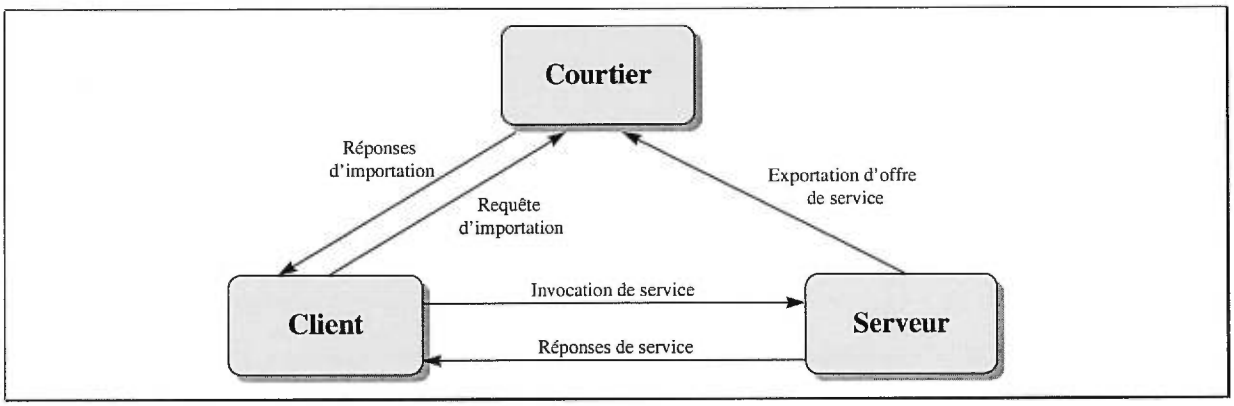


Figure A2.1 - Mécanisme de courtage

Quand un client requiert un service, il émet une requête de service au courtier pour trouver un serveur convenable. Une requête de service exprime les caractéristiques requises par le client en spécifiant le type de service désiré et des contraintes sur les propriétés de service et sur les propriétés de l'offre de service. Le processus de demande d'information sur un service convenable est appelé *importation*, et l'objet qui demande cette information est appelé *importateur*. Sur réception d'une requête de service, le courtier recherche dans sa base de données les offres de service répondant à la requête de service. La liste des offres de service trouvées est retournée au client. Ce dernier choisit un élément de cette liste et utilise l'identificateur de l'interface de service pour accéder au serveur.

Les courtiers peuvent être interconnectés pour partager leurs offres de service en formant une fédération de courtiers [ISO96b, Muller96, OMG96, Bearman91].

A2.2 Standardisation du service de courtage

La fonction de courtage est actuellement standardisée dans le cadre de la norme ODP [ISO96a], qui est le résultat d'un effort conjoint entre l'ISO et l'ITU. Cette norme est appelée *la Fonction de courtage ODP* (ODP Trading Function) [ISO96b].

En 1996, l'OMG a adopté une spécification pour le service de courtage dans le cadre des services objets de l'architecture CORBA2.0. Ce service est appelé *CORBA Object Trading Service* [OMG96]. Le travail sur le service de courtage dans ODP a atteint le statut de norme préliminaire DIS (draft international standard) quand l'OMG a reçu des réponses concernant les propositions de services objet RFP5 [OMG96]. Les soumissionnaires ont été principalement des compagnies ayant été impliquées dans le standard d'ODP. Ceci a permis la convergence des deux standards de courtage. La sémantique sous-jacente commune des deux efforts améliore beaucoup les perspectives d'interconnexion future des plates-formes des deux environnements.

A2.3 Fonction de courtage ODP

La norme ODP a pour objectif de permettre l'utilisation transparente des services sur des architectures matérielles et logicielles hétérogènes. La fonction de courtage ODP offre les moyens d'exporter un service et les moyens de découvrir ou d'importer les services ayant été offerts.

La fonction de courtage ODP constitue la spécification d'origine à partir de laquelle toutes les autres spécifications et implémentations produites par la suite ont été dérivées. La norme ODP définit la spécification de la fonction de courtage à partir de trois points de vue : *entreprise*, *information* et *traitement* (computational).

A2.3.1 Point de vue entreprise du courtier

Du point de vue entreprise, un courtier est un membre d'une communauté établie pour les besoins de courtage. Cette communauté est constituée de membres ayant des rôles tels que : courtier, exportateur, importateur, et administrateur de courtier. Les activités de courtage, exportation et importation des services, sont gouvernées par la politique de courtage du courtier, des importateurs, et des exportateurs. Les politiques du courtier sont les règles qui

déterminent et qui guident le comportement du courtier. Elles comprennent des politiques générales, telles que l'arbitrage entre les parties impliquées en cas de conflit et la rémunération liée aux activités de courtage, des politiques d'acceptation des offres de services, et des politiques d'acceptation des requêtes de service.

Chaque importateur peut aussi avoir sa propre politique d'importation qui décrit ses attentes d'une importation de service. Par exemple, une politique de portée (scoping policy) dans une opération d'importation peut restreindre l'ensemble des offres de services considérées par le courtier dans une importation de service. Par conséquent, le processus d'assortiment (matching) effectué par le courtier lors de l'importation est gouverné par la politique du courtier et de l'importateur. Chaque exportateur peut aussi avoir sa propre politique d'exportation qui décrit les attentes de l'exportateur d'une exportation de service. Cependant, la fonction de courtage ODP n'a pas donné de spécification pour la politique d'exportation.

La norme décrit aussi le point de vue entreprise de l'interconnexion de courtiers. Une fédération de courtiers est une communauté de courtiers, chacun avec des importateurs et des exportateurs. Les politiques d'interconnexion des courtiers sont des règles qui déterminent et qui guident le comportement d'un courtier en termes de propagation des requêtes de service dans la fédération des courtiers.

A2.3.2 Point de vue information du courtier

La spécification information de la fonction de courtage identifie les objets d'information, les règles, et les contraintes de manipulation de l'information par le service de courtage.

A2.3.2.1 Service, type de service, et type de propriété

Un *service* est une fonction offerte par un objet à une interface de traitement. C'est un ensemble de fonctionnalités disponibles via l'interface de l'objet. Un service peut être une opération atomique (i.e. write), une séquence d'opérations (i.e., open, write, close), ou un ensemble d'opérations (i.e., plusieurs write à effectuer dans un ordre aléatoire, offrir/consommer un flot d'informations, etc.).

Chaque service est une instance d'un *type de service*. Les instances d'un même type de service peuvent différer dans certains aspects non fonctionnels (tels que le coût d'utilisation du service). Ces caractéristiques additionnelles d'un service sont appelées *propriétés de service*.

Chaque propriété de service est une instance d'un *type de propriété de service* qui est constitué d'un nom qui identifie la propriété, d'un type qui détermine les valeurs permises de la propriété, et d'un mode qui spécifie si la propriété est *obligatoire*, *optionnelle*, *en lecture seule*, ou *modifiable*.

Un exportateur spécifie le type de service qu'il annonce et un importateur spécifie le type de service qu'il désire importer.

Un exemple de type de service est le service d'impression :

```
service PrintService {
                    readonly   property   string          PrinterName;
        mandatory   readonly   property   integer         PrinterType;
        mandatory   readonly   property   <resolution>    PrinterResolution;
                    readonly   property   integer         CostPerPage;
};
```

A2.3.2.2 Offres de service

Une offre de service contient l'information décrivant un service qui est échangé pour être utilisé par d'autres objets à une interface de traitement. Elle contient le type de service offert, l'interface à laquelle une association peut être établie pour obtenir le service, et les valeurs des propriétés du service.

Les propriétés de service sont exprimées comme des paires "nom propriété - valeur". Une offre de service doit avoir une valeur valable pour chacune des propriétés obligatoires du type de service précisé. En outre, les propriétés modifiables peuvent contenir, au lieu d'une valeur de propriété, une interface à laquelle la valeur réelle de la propriété peut être récupérée lors de l'assortiment. Ces propriétés sont appelées *propriétés dynamiques*. L'aptitude à supporter des propriétés dynamiques est optionnelle.

Le courtier attribue un identificateur unique à chaque offre de service enregistrée dans son espace des offres. Un exemple d'offre de service pour le service d'impression est le suivant :

```
interface reference
properties
    PrinterName          "hp3252"
    PrinterType          "laser"
    PrinterResolution    (720,720)
     CostPerPage         5c
```

A2.3.3 Point de vue traitement du courtier

Dans le point de vue traitement (computational), les interfaces d'un courtier avec son environnement sont visibles. Ces interfaces sont groupées en deux catégories :

- ***interfaces fonctionnelles*** - Les cinq interfaces fonctionnelles du courtier sont : *Lookup*, *Register*, *Proxy*, *Link*, et *Admin*. En outre, deux interfaces auxiliaires : *OfferIterator* et *OfferIdIterator* sont aussi spécifiées. Dans ce point de vue, les politiques des clients sont exprimées sous forme de paramètres des opérations, et les politiques du courtier sont exprimées comme des attributs du courtier.
- ***interfaces abstraites*** - La norme précise les interfaces abstraites : *SupportAttributes*, *ImportAttributes*, *LinkAttributes*, et *TraderComponents*. Les trois premières interfaces sont utilisées par les clients afin d'obtenir les valeurs des attributs correspondants du courtier. L'interface *TraderComponents* contient cinq attributs à lecture seule pour les cinq interfaces fonctionnelles, et est héritée par chacune des interfaces fonctionnelles, de sorte que la connaissance de n'importe quelle interface fonctionnelle soit suffisante à obtenir les valeurs des autres interfaces.

Les concepts introduits dans cette section et la spécification des interfaces précédentes sont décrits plus en détail dans [ISO96b].

A2.4 Le service de courtage de l'OMG

A2.4.1 Interfaces du courtier

Dans cette section, nous donnons un aperçu de la spécification des principales interfaces du courtier qui sont utilisées par les exportateurs pour annoncer leurs offres de services, et par les importateurs pour découvrir les offres de services répondant à leurs besoins.

A2.4.1.1 Interface Lookup

Cette interface est utilisée par les clients et par les courtiers pour découvrir et importer des services. Elle définit une seule opération appelée *query()*, qui requiert la spécification du type de service à importer et des contraintes sur les propriétés du service. Elle retourne une liste d'offres de services répondant au critère de recherche.

Le tableau suivant résume les opérateurs utilisés dans le langage de spécification des contraintes :

and or not	Opérateurs logiques.
== != < <= > >=	Opérateurs de comparaison pour les types simples : texte, numérique, et booléen.
+ - * /	Opérateurs arithmétiques pour les numériques.
<string1> ~ <string2>	string1 est une sous chaîne de string2.
<element> in <sequence>	Test d'appartenance à une séquence.
exist <property>	property est présente dans l'offre de service.

Un importateur désirant importer le service d'impression défini précédemment peut spécifier ses contraintes concernant les propriétés de ce type de service en utilisant les opérateurs du tableau ci-dessus. Un exemple de contraintes est comme suit :

"PrinterType = laser and CostPerPage < 5c"

A2.4.1.2 Interface Register

Cette interface offre aux exportateurs de services les opérations suivantes :

export	Annoncer une offre de service au courtier qui retourne son identificateur.
withdraw	Supprimer une offre de service du courtier.
describe	Retourner les propriétés d'une offre de service bien identifiée.
modify	Modifier les valeurs des propriétés d'une offre de service qui ne sont pas en lecture seule.

A2.4.1.3 Interface Link

Cette interface permet à un courtier d'utiliser les services d'un autre courtier. Les opérations offertes par cette interface sont :

add_link	Ajouter aussi bien le nom que la référence objet de l'interface Lookup du courtier cible. Une politique pour effectuer des recherches distantes à travers ce lien peut être spécifiée.
remove_link	Supprimer un lien dans le courtier.
modify_link	Changer les informations associées à un lien.
describe_link	Obtenir l'information concernant un lien
list_links	Obtenir une liste de tous les liens du courtier.

A2.4.1.4 Interface Proxy

Cette interface permet au courtier de déterminer, lors du processus d'assortiment, la référence objet de l'interface où le service est offert. Les opérations définies dans cette interface sont : *export_proxy()* , *withdraw_proxy()* , et *describe_proxy()* pour respectivement l'exportation, la suppression , et l'obtention de la description d'une offre proxy.

A2.4.2 Interfaces d'administration du courtier

La spécification d'OMG du service de courtage offre deux interfaces d'administration : l'interface *ServiceTypeRepository* et l'interface *Admin*.

A2.4.2.1 Interface ServiceTypeRepository

Elle permet la création et la gestion des types de services. Les opérations offertes par cette interface sont :

add_type	Créer un nouveau type de service. Le nom du type de service,

	son interface et ses propriétés doivent être donnés en paramètres.
remove_type	Supprimer un type de service.
list_types	Lister les noms de tous les types de service du répertoire des types.
describe_type	Obtenir la description d'un type de service particulier.
fully_describe_type	Obtenir la description d'un type et toutes ses interfaces parentes.
mask_type	Arrêter l'annonce d'un type de service.
unmask_type	Amener un type masqué à vie à nouveau.

A2.4.2.2 Interface Admin

Cette interface contient un grand nombre d'opérations permettant d'établir les politiques du courtier. Les utilisateurs ordinaires du courtier peuvent demander les attributs des autres interfaces afin de déterminer les politiques actuelles du courtier, mais n'auront jamais à utiliser l'interface Admin. Certains courtiers n'offrent pas cette interface car toutes les politiques sont déterminées par l'implémentation.

A2.4.3 Un scénario de courtier

Le scénario de la figure A2.2 montre comment toutes les interfaces du courtier fonctionnent ensemble.

1. **Le serveur crée un nouveau type de service** en invoquant l'opération *add_type()* de l'interface ServiceTypeRepository du courtier.
2. **Le serveur annonce son service** en invoquant l'opération *export()* de l'interface Register du courtier. Il passe le nom du type de service, la référence objet qui servira les requêtes des clients, et les valeurs des paramètres.
3. **Le client obtient une liste des types de service** en invoquant *list_types()* de l'interface ServiceTypeRepository.
4. **Le client obtient une description du type** en invoquant *describe_type()* de l'interface ServiceTypeRepository.
5. **Le client émet une requête de service** en invoquant l'opération *query()* de l'interface Lookup du courtier pour obtenir la liste des objets pouvant offrir le service requis. Le

client doit passer le type de service et les paramètres qu'il désire qu'ils soient retournés. Il peut rétrécir la requête en spécifiant ses contraintes et ses politiques. Il peut aussi ordonner les résultats en spécifiant ses préférences.

6. **Le client invoque le service**. Le client choisit un objet qui peut fournir le service et invoque ses méthodes pour obtenir le service. Le client peut invoquer la requête à travers des invocations statiques CORBA s'il a un talon (stub) de cette interface. Sinon, il doit invoquer dynamiquement l'interface en utilisant l'invocation dynamique d'interface de CORBA.

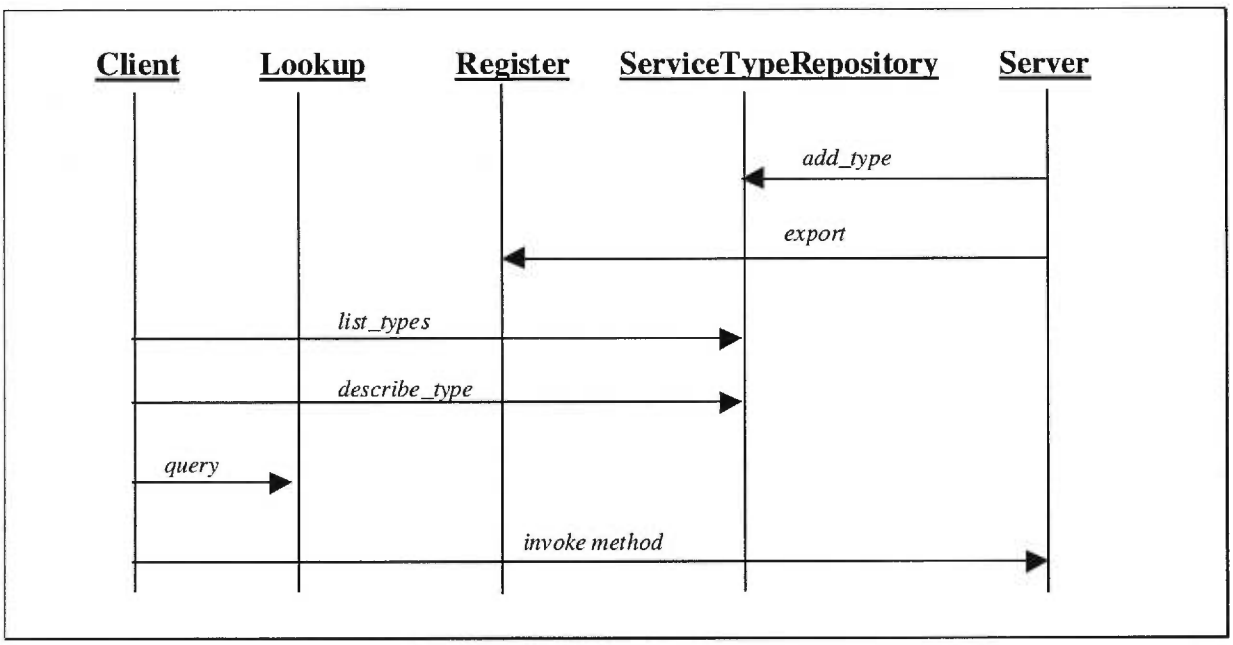


Figure A2.2 - Exemple d'utilisation des interfaces du courtier