DOMS : une architecture de suivi de la Qualité de Service dans les Systèmes Répartis Objet

Nathanaël Cottin

Responsable : M. Wack
UTBM – DEA IAP Besançon

Février/Juillet 2000

Remerciements :

Je souhaiterais remercier l'ensemble des professeurs et toutes les personnes ayant contribué à ce projet ainsi qu'à l'élaboration du présent document.

Je remercie notamment les membres du groupe de recherche et particulièrement :

- Mon responsable de stage, M. Wack, pour m'avoir permis d'intégrer le groupe de recherche de l'Université de Technologie de Belfort – Montbéliard (UTBM)
- J. Gaber et O. Baala pour leurs précieux conseils concernant les grandes orientations à prendre et les écueils à éviter afin de mener le travail demandé à son terme ainsi que pour leur aide quant à la rédaction du rapport
- B. Mignot pour ses connaissances concernant les architectures et algorithmes parallèles.

Je remercie également chaleureusement ma famille et mes amis pour leur soutien.

Stage soutenu le Lundi 2 Octobre 2000 à l'UTBM de 9h30 à 10h30 en présence de :

- M. Gérard Bahi, professeur à l'IUT de Belfort
- M. Maxime Wack, responsable du DEA
- M. Jaafar Gaber
- Mme Oumaya Baala
- M. Bernard Mignot
- M. Fabrice Sibille
- Mlle Emilie Boillot.

Note attribuée : 16 / 20.

Mention : BIEN.

à René

Sommaire :

1 Introduction :

Les principaux acteurs du développement de *middlewares* ou environnements répartis objets tels que CORBA de l'OMG, DCOM de Microsoft et JAVA RMI de Sun, sont devenus incontournables. Ces logiciels intermédiaires proposent un ensemble d'outils chargés de simplifier le développement des applications réparties objet.

Cependant, les performances accrues des ordinateurs associées aux traitements parallèles et aux environnements répartis ne suffisent pas, à eux seuls, à combler les besoins de qualité de service (QoS) des applications réparties. Les critères de QoS retenus dans notre étude concernent uniquement les performances (temps de réponse) et la fiabilité de ces applications.

Nous nous intéressons particulièrement aux applications réparties orientées objet ou *applications réparties objet* dans lesquelles des objets communiquent entre eux par envoi de messages synchrones. Ces objets « peuvent s'exécuter sur des machines ayant des architectures matérielles et des systèmes d'exploitation hétérogènes et peuvent être écrits dans des langages hétérogènes » [Badidi 98].

Une solution permettant d'améliorer cette QoS est de proposer au développeur un outil prenant en charge la supervision du réseau et des machines ainsi que le suivi des objets de son application.

Une application répartie objet est plus performante et fiable lorsque les impacts du réseau de communication et de la disponibilité des machines sont réduits, c'est à dire lorsque :

- Les communications distantes (via le réseau) entre les objets sont optimisées : un appel local est généralement plus performant et fiable qu'un appel distant entre deux objets car le réseau n'intervient pas
- Le travail est réparti équitablement sur les machines : aucune machine n'est surchargée alors qu'une autre est oisive.

Nous proposons DOMS (Distributed Objects Management System), un environnement réparti objet qui fournit au développeur des outils de mesure des performances et de la fiabilité des applications réparties objet. Ce système, basé sur l'étude des relations dynamiques entre les objets, prend en compte l'évolution de la charge des machines et les temps de réponse des objets.

L'objectif est d'obtenir une répartition optimum des objets sur les machines du réseau de communication en permettant au développeur de déterminer les objets à déplacer ainsi que le moment et le lieu de migration.

Après une première partie consacrée à la définition des termes employés, nous présenterons les travaux menés par d'autres laboratoires et indiquerons ce en quoi DOMS se démarque des systèmes existants. La troisième partie spécifiera un modèle décrivant les relations entre les objets ainsi que les indices de mesure de la QoS (performances et fiabilité) des applications réparties objet. Le système DOMS et les protocoles sous-jacents seront ensuite présentés. Les annexes approfondiront enfin les notions introduites lors de la présentation de l'architecture DOMS.

2 Terminologie :

2.1 Système réparti objet :

Un système réparti objet (SRO) [Chatonnay 96] décrit les éléments nécessaires à l'exécution des applications réparties objet. Il est constitué :

- D'un ensemble de machines reliées entre elles via un réseau de communication
- D'un ou plusieurs réseaux de communication interconnectés
- D'un ensemble d'applications réparties objet (ARO) qui coopèrent
- D'un ou plusieurs environnements répartis objet permettant de déployer ces applications.

Ces quatre éléments forment un ensemble permettant de répondre aux demandes des utilisateurs. De chacun d'eux dépendent les performances et la fiabilité du système (mesures de sa QoS).

Nous considérerons par la suite l'ensemble des ARO d'un SRO comme une application unique.

2.2 Environnement réparti objet :

Un environnement (ou *middleware*) réparti objet (ERO) est une couche logicielle qui permet au développeur d'applications réparties objet de s'affranchir de la gestion [Geib 97] :

- Des mécanismes de création et destruction des objets sur les machines du réseau
- Des communications entre les objets (retrouver un objet sur le réseau, invoquer une méthode à distance, recevoir le résultat d'une invocation)
- De l'hétérogénéité des plates-formes sur lesquelles sont déployées les applications réparties.

Ainsi le développeur peut se consacrer entièrement à la conception fonctionnelle de son application.

Les environnements répartis les plus utilisés à l'heure actuelle sont CORBA de l'OMG [Orfali 95], JAVA RMI de Sun, ainsi que DCE et DCOM de Microsoft. Tous ces environnements ont en commun qu'ils dérivent du mécanisme d'appel de procédure distante (RPC – Remote Procedure Call) introduit par Sun [Daniel 00].

Visibroker (Inprise) et ORBacus (Object Oriented Concepts) sont deux exemples d'implémentation de l'environnement réparti CORBA pour les langages objets C++ et JAVA.

2.3 Application répartie objet :

Les applications réparties objet (ARO) sont des applications pouvant être déployées à grande échelle sur des sites éloignés géographiquement. Elles sont composées d'objets chargés d'effectuer des tâches particulières. Ils coopèrent afin de répondre aux demandes des utilisateurs.

Cette coopération s'effectue par un mécanisme d'envoi de messages entre les objets.

Le développement de telles applications se heurte à trois problèmes principaux [Geib 97] :

- La définition d'un protocole de communication entre les objets répartis basé sur le principe d'interface et hérité de la conception orientée objet : ce protocole doit fournir une infrastructure permettant aux objets de communiquer par le biais d'envoi de messages
- Le masquage de la gestion de l'hétérogénéité des plates-formes (types de machines, systèmes d'exploitation) au développeur
- L'interopérabilité entre les différents environnements de création et de déploiement des applications réparties. Ceci implique un développement commun de solutions non-propriétaires.

Nous dénombrons cinq ensembles principaux servant à définir une ARO dans son environnement :

- $M^t = \{M_1, M_2, ...\}, Card(M^t) = p^t \geq 1$: ensemble des *p* machines disponibles sur le réseau à l'instant *t*, c'est à dire pouvant être utilisées par l'application au cours de son exécution

- $N^t \subseteq M^t, Card(N^t) = q^t \geq 1$: ensemble des machines utilisées par l'ARO à l'instant *t*
- $O^t = \{O_1, O_2, ...\}, Card(O^t) = k^t > 1$: ensemble des objets composant l'ARO à l'instant *t*
- $A^t = \bigcup_m A_m^t, A_m^t = \{O_1, O_2, ...\} \subseteq O^t, Card(A_m^t) = k_m^t > 0, \forall m \in N^t$: ensemble des ensembles d'objets présents sur chaque machine *m* à l'instant *t*, vérifiant les propriétés $Card(A^t) = q^t$, $A_m^t \cap A_n^t = \varnothing, \forall (m,n) \in (N^t)^2, m \neq n$, $\sum_m k_m^t = k^t, \forall m \in N^t$ et $O^t = \bigcup_m \bigcup_i i, \forall m \in N^t, \forall i \in A_m^t$, qui traduisent le fait qu'un objet ne peut être localisé que sur une et une seule machine à un instant donné
- $L^t = \{L_{1,1}, L_{1,2}, ...\} \neq \varnothing$: ensemble des liens de communications entre les objets à l'instant *t*, sachant que $L_{i,j}$ indique un appel de l'objet *j* par l'objet *i*. *L* est la matrice d'incidence décrivant les invocations entre les objets [Adoud 00] qui servira de base à DOMS.

Une ARO est ainsi définie par le graphe $G = (A, L, M)$ ou plus généralement par $G' = (O, L)$.

2.4 Objet réparti :

L'approche orientée objet pour le développement d'applications a fait ses preuves [Meyer 88] [Wegner 90], notamment pour la conception d'applications réparties [Orfali 95]. Nous appellerons objet réparti tout objet composant une ARO, identifiable sur le réseau de communication lors de sa création. Chaque objet est unique, même si plusieurs objets peuvent être des instances d'une même classe (plusieurs copies d'un même objet).

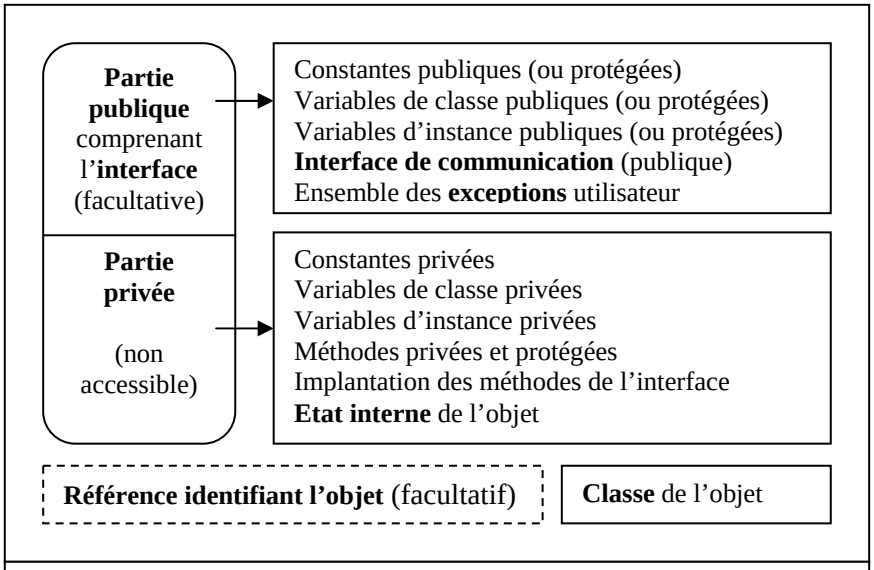

Figure 1 : composition d'un objet réparti

Un objet réparti est composé d'une partie publique et d'une partie privée (figure 1) :

- La partie publique décrit l'interface de communication de l'objet : l'interface de communication de l'objet est l'ensemble des signatures des méthodes publiques pouvant être appelées par d'autres objets. Une référence identifiant l'objet permet aux autres objets d'entrer en contact avec lui lorsque ce dernier dispose d'une interface de communication. Certains objets ne disposent cependant pas d'interface de communication. Ils ne peuvent être appelés. Un objet réparti est donc toujours identifiable, mais non systématiquement joignable par les autres objets répartis. Nous distinguerons ainsi par la suite plusieurs types d'objets répartis

- La partie privée est la partie cachée de l'objet qu'il est le seul à connaître : elle comprend le corps des méthodes de l'interface ainsi que les variables et méthodes privées utilisées par l'objet ; son état interne fait référence aux valeurs de ses variables publiques et privées ainsi qu'aux requêtes reçues et en attente de traitement.

La séparation entre l'interface (partie publique) et son implantation (partie privée) permet au développeur de modifier l'implantation interne d'un objet *o* sans pour autant devoir modifier les objets appelant *o*.

Les communications entre les objets s'effectuent par le biais d'appels distants des méthodes de l'interface de communication de chaque objet. Il est commun d'employer indifféremment les termes de « requête » et d'« invocation de méthode » ou, par extension, d'« invocation d'objet ».

Une invocation englobe (1) l'appel d'une méthode située sur un objet distant, puis (2) l'exécution locale de cette méthode chez l'objet distant et enfin (3) l'éventuel message de retour d'un résultat à l'objet appelant : $Invocation = Appel + Exécution \left[+ Résultat\_retourné\right]$.

On distingue ainsi l'objet appelant, à l'initiative de l'appel, de l'objet appelé, qui exécute localement la méthode demandée par l'objet appelant et lui retourne éventuellement un résultat.

L'invocation d'un objet peut être statique (ISO : invocation statique d'objet) ou dynamique (IDO : invocation dynamique d'objet) selon que la référence de l'objet appelé est connue ou non avant l'appel :

- L'invocation statique ISO ne permet pas à un objet appelant de changer d'objet appelé au cours de son exécution : il n'a pas d'autre choix que de contacter cet objet précis ; dans le cas où ce dernier a été détruit, l'objet appelant n'a aucun recours pour continuer son traitement
- Contrairement à l'ISO, l'invocation dynamique IDO d'un objet n'est pas figée. L'objet appelant ne contacte pas directement l'objet appelé mais demande la référence d'un objet de même classe *C* que l'objet devant être appelé. Un mécanisme inclus dans l'ERO, par exemple le service de nommage de CORBA [OMG 97a], est alors chargé de sélectionner la référence d'un objet de classe *C* parmi l'ensemble des objets enregistrés selon une politique donnée. En général, l'environnement réparti objet sous-jacent opère une sélection cyclique de l'objet par « round-robin ». La référence de l'objet sélectionné est alors envoyée à l'objet appelant qui peut alors contacter cet objet et invoquer ses méthodes à distance.

L'ISO et l'IDO ne doivent pas être confondues avec les mécanismes SII [Geib 97] et DII [Geib 97] d'invocation transparente et de découverte dynamique des méthodes de l'interface d'un objet.

2.5 Fiabilité des objets d'une application répartie :

L'utilisation du mécanisme d'invocation distante par envoi de messages utilisé par les ERO peut éventuellement déclencher trois types de problèmes (figure 2).

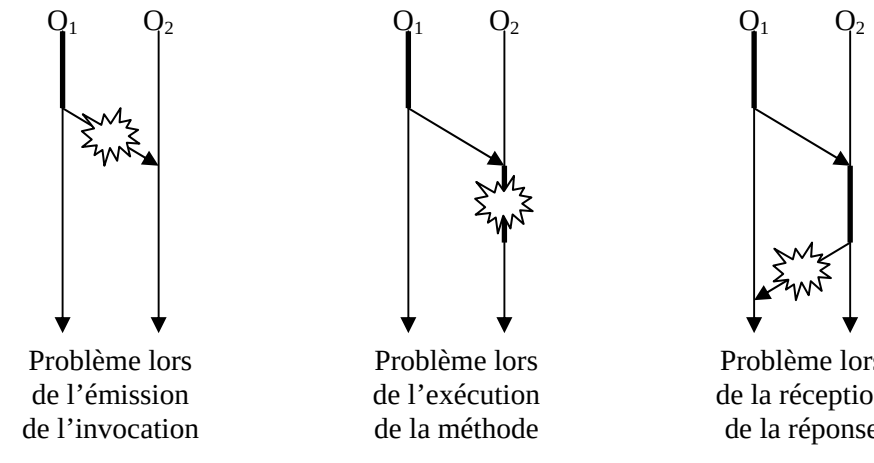

Figure 2 : problèmes rencontrés lors d'une invocation de méthode

Il apparaît que les premier et troisième problèmes sont liés à un dysfonctionnement du réseau.

Dans le premier cas, la distinction entre une erreur d'acheminement de la requête et une erreur causée par l'impossibilité de joindre un objet doit être faite. L'environnement réparti objet sous-jacent est souvent capable d'opérer cette distinction et de renseigner l'objet appelant sur l'origine de la panne.

2.6 Qualité de service des applications réparties objet :

La qualité de service (QoS) des ARO (facilité de développement, maintenance, déploiement à grande échelle ou « scalability », performances, répartition et vitesse de traitement des données manipulées, tolérance aux pannes, fiabilité et sécurité des communications sur le réseau) est un critère déterminant dans les choix stratégiques des solutions informatiques des entreprises.

Cette mesure exprime de manière générale un degré de satisfaction des requêtes entre les objets répartis sous des contraintes dynamiques ou statiques, selon qu'elles varient ou non au cours de l'exécution de l'application.

L'évaluation de cette QoS doit prendre en compte les QoS matérielle et logicielle dérivées de la description des systèmes répartis objet :

- La QoS matérielle concerne les machines (charge processeur et mémoire) et le réseau (temps de communication)
- La QoS logicielle concerne les objets de l'ARO en termes de nombre d'invocations et de délais de traitement.

2.7 Répartition de charge statique et dynamique :

Pour améliorer les performances et la fiabilité des ARO, deux méthodes sont envisageables : la répartition de charge statique (RCS) et la répartition de charge dynamique (RCD).

La RCS répartit les objets lors du déploiement de l'ARO contrairement à la RCD. Deux approches de la répartition de charge dynamique (intervenant pendant l'exécution d'une ARO) peuvent être mises en œuvre, à savoir la répartition d'invocation (RI) et la répartition d'objet (RO) :

- La première (RI) oriente les appels vers les machines et objets les moins chargés (effectuant le moins de traitements). La duplication de certains objets s'avère nécessaire (figure 3). L'intérêt est de limiter les files d'attente des objets (ensemble des requêtes non encore traitées par les objets appelés) lors d'invocations concurrentes d'un objet
- La répartition d'objet (RO) ne duplique pas « à priori » les objets mais autorise un déplacement dynamique de ceux-ci vers les machines les moins chargées (figure 4). Les files d'attente ne sont pas gérées mais l'exécution d'une méthode sera réalisée sur l'une des machines les moins chargées du réseau.

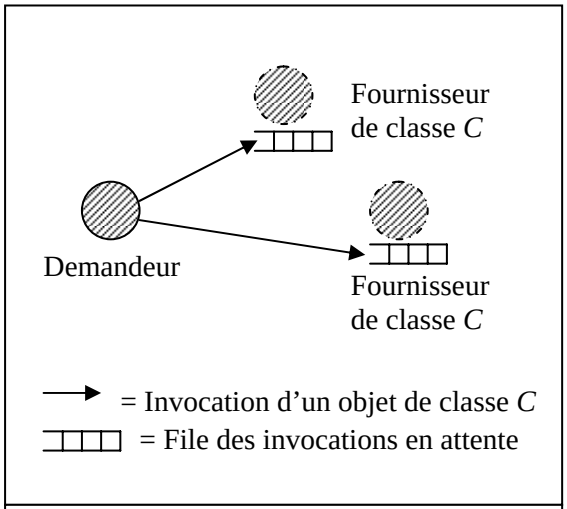

Figure 3 : répartition d'invocation (RI)

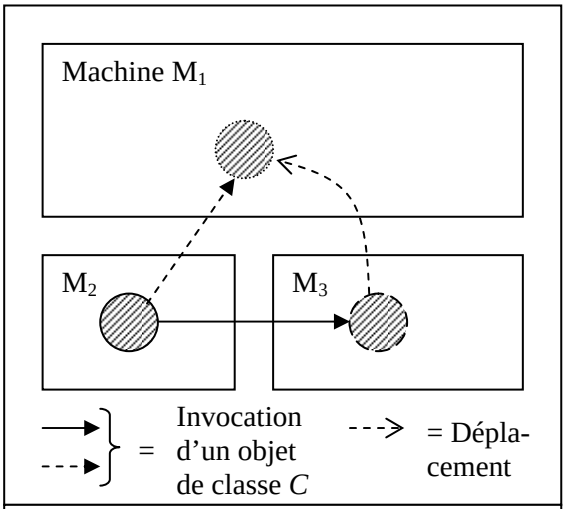

Figure 4 : répartition d'objet (RO)

2.7.1 Répartition d'invocation :

La répartition d'invocation à lieu lorsque plusieurs instances d'un même objet appelé sont disponibles.

Pour que ce mécanisme soit effectif, l'invocation d'une méthode ne doit pas faire appel à un objet précis (ISO : invocation statique d'objet) mais à un objet d'une classe donnée (IDO : invocation dynamique d'objet) afin que l'objet appelé soit sélectionné par l'ERO selon une politique à définir.

Le problème posé par cette méthode est de garantir la cohérence de l'état des instances des objets dupliqués. En particulier les variables de classe de ces objets doivent avoir des valeurs identiques à chaque instant pour que le résultat l'appel d'une méthode soit cohérent (qu'il ne dépende pas de l'objet effectivement appelé). Ceci implique le partage des variables entre les objets d'une même classe.

2.7.2 Répartition d'objet :

La répartition d'objet peut autoriser certains objets à migrer vers les machines les moins chargées du réseau. L'intérêt n'est plus dans ce cas de répartir les appels entre les objets mais de répartir les objets eux-mêmes sur les machines.

La référence identifiant l'objet peut ainsi varier au cours de l'exécution de l'ARO. L'objet doit alors être identifié par un autre mécanisme. Les appels statiques (ISO) et dynamiques (IDO) sont possibles dans le seul cas où chaque invocation effectue une connexion à l'objet préalable à chaque appel.

Si cette dernière méthode résout le problème de la conservation de l'intégrité des appels tout en évitant de surcharger les machines par duplication des objets, elle pose d'autres problèmes liés à l'identification, à la migration et à l'invocation des objets en cours de déplacement, auxquels DOMS apportera quelques éléments de réponse.

2.8 CORBA :

CORBA (Common Object Request Broker Architecture) est un environnement réparti objet non propriétaire créé en 1989 par l'OMG (Object Management Group), un consortium regroupant plusieurs grands comptes internationaux tels que Sun, 3Com, Unisys, Hewlett-Packard, Rational Software, Canon, Phillips, American Airlines ainsi que des universités (NASA, INRIA, LIFL).

L'idée directrice de l'OMG est que le design et l'implémentation de logiciels (*middlewares*) permettant la création d'applications réparties sans standards ni conventions ne mèneront à rien. L'OMG est donc un organisme de standardisation et non de développement. L'accent est mis sur l'interopérabilité de ces applications réparties qui doivent être capables de communiquer entre elles, les mécanismes tels que les sockets TCP/IP, RPC (Remote Procedure Call) et JAVA-RMI (Remote Method Invocation) ayant montré leurs limites [Geib 97] (soit de trop bas niveau : sockets, soit de granularité trop importante, c'est à dire ne permettant pas d'invoquer une méthode d'un objet dans un processus : RPC, soit trop spécialisé : JAVA-RMI, utilisé uniquement pour développer des applications JAVA).

Leurs recherches ont abouti à la mise au point de standards pour la conception d'applications réparties, indépendantes non seulement du langage de programmation des objets, mais également de l'hétérogénéité des machines utilisées.

Afin de permettre la communication entre les objets hétérogènes, l'OMG propose un bus réparti sur lequel sont connectés ces objets. Les objets dialoguent entre eux par l'intermédiaire de souches de communication qui leur masquent les communications à travers le réseau. Ces souches emballent et déballent les invocations dans les requêtes ensuite transportées par le bus. Les objets, interconnectés à travers le bus CORBA, n'ont plus à se préoccuper des aspects de la communication entre des sites hétérogènes, aspects qui sont entièrement gérés par le bus [Geib 97].

3 Travaux concurrents :

Ce chapitre présente un certain nombre de travaux dans le domaine de la surveillance des applications réparties issus de la littérature. Le dernier paragraphe est consacré à la comparaison des travaux mentionnés avec notre système. Il met en avant les contributions de notre environnement.

3.1 LYDIA :

3.1.1 Objectif :

Une approche de la répartition de charge dans les systèmes répartis non objet est implémentée dans le projet LYDIA [Gelenbe 96]. Les codes source des applications surveillées sont modifiés afin de permettre la mesure de leurs charges dynamiques. La répartition de charge est effectuée par un module central qui, pour chaque invocation, sélectionne le processus le plus approprié parmi ceux de même classe.

Le projet LYDIA approche le problème de la répartition de charges en proposant et en implémentant un système basé sur le modèle de G-Network, dérivé du modèle multiclasses afin de limiter le temps de réponse des processus composant l'application répartie surveillée.

Le système LYDIA est spécifique à la surveillance et la gestion d'applications réparties déployées sous environnement UNIX sur des machines homogènes (compatibles binairement).

3.1.2 Modèles multiclasses et G-Network :

3.1.2.1 Modèle multiclasses :

Introduit par [Gelenbe 91], le modèle multiclasses basé sur la théorie des files d'attentes à *N* serveurs. Ce modèle fait cohabiter *R* classes de clients « positifs » et *S* classes de clients « négatifs ». Les arrivées des clients sont supposées indépendantes et suivent une loi de Poisson. La distribution des temps de service est supposée exponentielle.

Les clients « positifs » sont les seuls à être servis. Ils attendent dans une file où le serveur pourra honorer leur demande. Plusieurs serveurs peuvent proposer le même service et chaque serveur propose en général plusieurs services. Lorsqu'un client « positif » a été servi, il peut :

- Changer de classe afin de demander un autre service (sur n'importe quel serveur pouvant répondre à sa demande)
- Changer de nature en devenant un client « négatif » : les clients « négatifs » agissent en perturbateurs en amenant les clients « positifs » à changer de serveur. Chaque client « négatif » de classe *m* s'attache à sélectionner un client « positif » de classe *c* différent
- Quitter le système.

Ces changements sont modélisés par des chaînes de Markov utilisant les notations suivantes :

- $P^+[i,j][k,l]$: probabilité qu'un client « positif » de classe *k* soit servi et quitte la queue *i* pour la queue *j* en tant que client « positif » de classe *l*
- $P^-[i,j][k,m]$: probabilité qu'un client « positif » de classe *k* soit servi et quitte la queue *i* pour la queue *j* en tant que client « négatif » de classe *m*
- $d[i,k]$: probabilité qu'un client « positif » de classe *k*, servi dans la queue *i*, quitte le système.

Les auteurs définissent la relation $\sum_{j=1}^{N}\sum_{l=1}^{R} P^+[i,j][k,l] + \sum_{j=1}^{N}\sum_{m=1}^{S} P^-[i,j][k,m] + d[i,k] = 1, \forall (i,k)$

qui traduit l'indépendance des trois différentes possibilités offertes aux clients « positifs ».

3.1.2.2 Modèle G-Network :

Le modèle G-Network transpose les notions et formules du modèle multiclasses pour les adapter aux processus répartis sur les machines d'un réseau.

Les clients « positifs » sont les processus composant une application répartie. L'activation d'un processus correspond à l'entrée d'un client « positif » dans le système et l'arrêt d'un processus à son départ du système.

Un module appelé « scheduler » simule l'arrivée des clients « négatifs » sur les machines du réseau en utilisant les formules du modèle multiclasses. Il analyse les données fournies par différents « load monitors » et transmet des demandes de migration d'une machine *i* à une machine *j* au module « migration daemon », sans mentionner le processus devant être déplacé.

Ce dernier transpose les formules de sélection d'un client « positif » en choisissant un processus à migrer puis effectue le déplacement effectif du processus choisi.

3.1.3 Architecture :

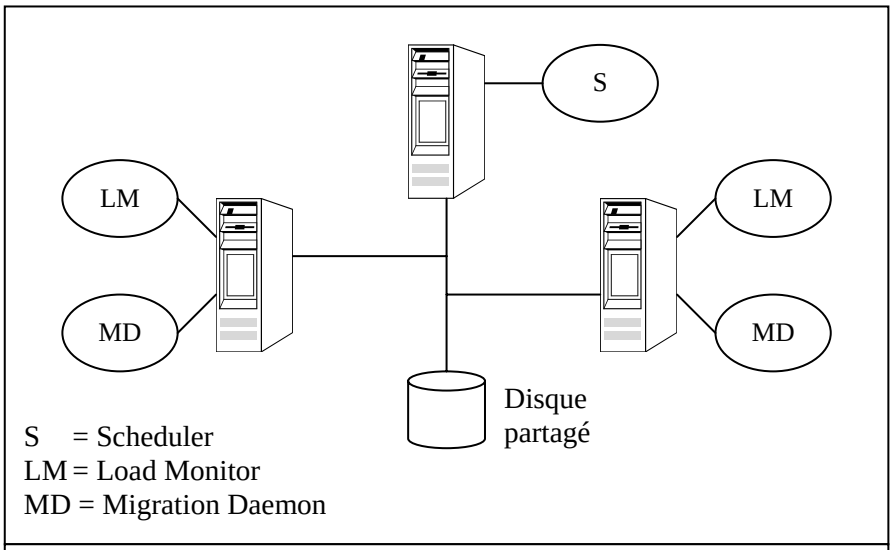

Figure 5 : architecture de LYDIA ; exemple avec trois machines

LYDIA est composé d'un « scheduler » central accédant aux données stockées sur un disque partagé ; en outre, un « load monitor » et un « migration daemon » sont exécutés sur chaque machine (figure 5).

3.1.4 Fonctionnement :

Le « scheduler » utilise les informations envoyées régulièrement par chaque « load monitor » pour déterminer si une machine est surchargée et dans ce cas vers quelle machine (plus disponible) la migration aura lieu. Le système actuel étant de type centralisé, il ne comporte qu'un seul « scheduler ».

Le module « load monitor » est exécuté sur chaque machine participant au transfert de charges. Il envoie au « scheduler » des informations concernant sa machine toutes les périodes *T*.

Ces informations sont :

- Load index : moyenne du nombre de processus actifs durant une période dans l'intervalle $[aT,(a+1)T[, \forall a \in N$
- f_{llu} : fréquence de mise à jour du « load index », exprimée en s^{-1}
- f_{glu} : fréquence des mises à jours globales par le « scheduler », exprimée en s^{-1}.

Le « migration daemon » prend en charge le déplacement des processus locaux.

Enfin, le disque partagé permet de sauvegarder l'état des processus UNIX devant être migrés.

3.1.5 Principe de migration de processus :

LYDIA suppose que chaque processus n'est composé que d'un unique programme. Migrer le processus consiste donc à migrer ce programme tout en conservant son état (dont l'ensemble des requêtes non encore traitées fait partie).

Un processus est décrit par les auteurs comme étant composé de :

- Un programme compilé et en cours d'exécution (il peut en règle générale contenir plusieurs programmes)
- Une pile contenant des données ainsi que la sauvegarde des registres
- Les données manipulées par le programme (ou les programmes dans le cas général)
- Le texte du (des) programme(s).

La migration des processus est prise en charge par le « migration daemon ». Ce dernier répond aux sollicitations du « scheduler » en choisissant un processus parmi l'ensemble des processus qu'il est possible de déplacer. Afin d'éviter les migrations multiples d'un même processus, le processus sélectionné ne devra pas avoir été déplacé par ce « migration daemon » au préalable.

Afin de permettre la migration d'un processus, chaque programme composant l'application répartie est lié lors de la compilation à une librairie « LB_LIB ». Une option autorise le compilateur à utiliser la fonction « LB_MAIN() » de la librairie à la place de « main() ».

Une fonction de cette librairie, « snapshot() », intégrée à l'exécutable du programme lors de la compilation, réagit au signal SIGTSP envoyé par le « migration daemon » situé sur la même machine que le programme à migrer (machine source). Cette dernière crée alors un fichier contenant :

- Le programme exécutable
- L'état courant du processus, comprenant les adresses de la pile de données et du texte du programme, le contenu des registres ainsi que le statut des fichiers ouverts par le programme.

Ce fichier est ensuite copié vers le disque partagé.

Le « migration daemon » situé sur la machine source envoie enfin une notification à son homologue situé sur la machine destination afin que celui-ci prenne connaissance du fichier créé.

3.1.6 Conclusion et perspectives :

LYDIA met l'accent sur la migration de processus dans les environnements UNIX uniquement. La surcharge induite par l'exécution des différents modules n'est pas prise en charge.

Le répertoire partagé est un organe vital car utilisé lors de chaque migration de processus.

Bien qu'étant opérationnel, le projet LYDIA souffre de quelques défauts que les auteurs vont tenter d'éliminer. Les principales améliorations prévues sont :

- De supprimer le disque partagé utilisé pour l'enregistrement et la consultation des données
- D'éviter le goulot d'étranglement du modèle centralisé (au niveau du « scheduler ») en s'appuyant sur un modèle totalement réparti
- De rendre LYDIA tolérant aux pannes, ce qui implique une révision du modèle de base
- De tenir compte de la charge du réseau ainsi que de la charge CPU de chaque machine afin d'adapter les fréquences d'envoi des données (entre les « load monitors » et le « scheduler ») de même que les fréquences de migration des processus (proportionnelles aux charges considérées).

3.2 Dome :

3.2.1 Objectif :

Le projet Dome [Arabe 95], Distributed Object Migration Environment, consiste en un ensemble de composants, d'opérations et de classes génériques (API) développées en C++ qui intègrent le *middleware* PVM (Parallel Virtual Machine). Ces classes permettent de développer de manière simple des programmes C++ :

- Automatiquement parallélisés par Dome
- Dont la charge sera dynamiquement répartie sur les machines d'un réseau
- Tolérants aux pannes via des mécanismes de *checkpointing* et redémarrage.

Le mécanisme de *checkpointing* a pour principal objectif de réduire le temps d'exécution d'une application en présence de fautes. L'insertion de points de contrôle permet de sauvegarder l'état du programme et autorise ainsi l'application à reprendre son exécution au dernier point de contrôle rencontré lorsqu'une faute survient. Ce mécanisme est comparable à la notion de transaction utilisée dans les SGBD.

3.2.2 Architecture :

L'utilisation de Dome suppose l'écriture de programmes spécifiques qui utilisent les classes et variables génériques de Dome.

Lorsqu'un tel programme, appelé « programme maître », est exécuté, l'environnement de Dome est créé. Son rôle est de répliquer le programme sur différentes machines avant d'exécuter chaque copie en parallèle, PVM prenant en charge les communications entre les copies et le programme maître.

Chaque copie effectue le même traitement sur une partie des variables du programme maître (les variables utilisées étant déclarées par Dome) : chaque variable est alors partitionnée entre les différentes copies du programme maître (figure 6). Les résultats sont collectés par le programme initial. La parallélisation du programme maître est ainsi quasi transparente au programmeur.

Par défaut, une seule copie du programme maître est exécutée sur chaque machine.

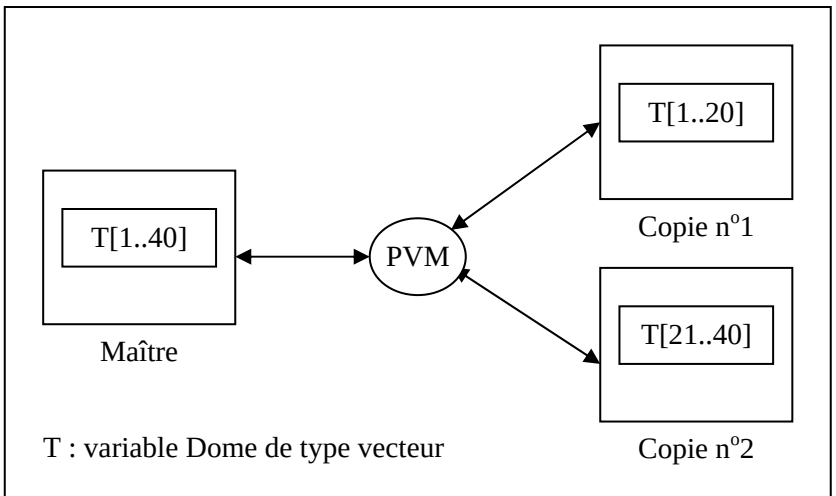

Figure 6 : architecture de Dome

3.2.3 Fonctionnement :

Répartir les charges entre les copies du programme maître revient à modifier le partitionnement des variables utilisées en tenant compte des charges interne et externe de chaque copie :

- Charge interne : charge liée à l'activité des divers modules composant le programme parallélisé sans tenir compte des machines et du réseau de communication ; la charge interne peut être mal répartie lorsqu'un module effectue un grand nombre de traitements en comparaison avec un autre module
- Charge externe : charge induite par l'exécution de modules qui partagent les ressources mémoire et processeur d'une même machine.

Pour ce faire, chaque opération usuelle (produit, somme, affectation, …) est surchargée (au sens C++) par Dome afin d'y intégrer une mesure de temps d'exécution. Les opérations utilisées dans le programme maître sont ainsi toutes des opérations surchargées.

La répartition de charge s'effectue de manière cyclique, dès lors qu'un certain nombre d'opérations *NB_OPERATIONS* a été effectué par les programmes, en quatre phases :

- Synchronisation des différentes copies et collecte des temps d'exécution par le maître
- Ordonnancement des copies selon la mesure croissante de leur temps d'exécution
- Repartitionnement des variables de chaque copie afin d'ajuster leurs prochains temps d'exécution
- Reprise de l'exécution du programme.

Un repartitionnement efficace des variables permet de minimiser non seulement les temps de calcul mais également le délai de synchronisation.

3.2.4 Conclusion et perspectives :

Dome a été porté sur plusieurs plates-formes UNIX. Il remplit les objectifs initiaux de facilité de programmation, répartition dynamique de charge, parallélisation automatique et tolérance aux pannes.

Cependant les variables génériques devant être utilisées sont restreintes, Dome ne comprenant que les variables de type vecteur. En particulier, les structures ne sont pas prises en compte à l'heure actuelle.

De plus, les points de contrôle doivent être manuellement intégrés au programme maître par le programmeur car le mécanisme de *checkpointing* n'est pas pris en compte par défaut.

Les travaux futurs concernent :

- La prise en considération des temps de communication entre les copies et le programme maître
- L'adaptation de l'algorithme de répartition de charge afin de modifier *NB_OPERATIONS* en fonction des temps moyens d'exécution collectés à chaque cycle
- L'ajout de nouveaux types de variables génériques et des opérations correspondantes
- L'introduction automatique de points de contrôle lors de la déclaration et de l'utilisation de chaque variable.

3.3 Hector :

3.3.1 Objectif :

L'environnement Hector [Russ 99] propose un système centralisé d'allocation automatique de tâches (processus) composant des programmes écrits en langage C, parallélisés et exécutés sur différentes machines d'un réseau.

Pour ce faire, Hector implémente des agents répartis chargés d'exécuter les tâches et de les surveiller. Les principes de *checkpointing* et de migration des tâches sont transparents à l'utilisateur et demandent des modifications ponctuelles du programme développé.

3.3.2 Architecture :

Hector est composé d'un « master allocator », organe central de prise de décision. Les décisions sont ensuite envoyées à des agents répartis appelés « slave allocators » via des communications socket

réalisées sous environnement UNIX. Ces communications sont événementielles : les messages envoyés ne sont pas bloquants et leur réception s'effectue à l'aide de signaux d'interruption.

Les agents effectuent des prélèvements périodiques concernant le système et les tâches actives. Ils envoient ensuite les informations recueillies au « master allocator ».

3.3.3 Fonctionnement :

Le fonctionnement du système Hector est décrit à trois niveaux.

3.3.3.1 Niveau des tâches :

Afin d'être surveillé, le programme est lié à un ensemble de trois librairies permettant de communiquer avec le système Hector (figure 7) :

- La librairie de migration fournit des fonctions nécessaires à la migration des tâches
- La librairie d'instrumentation permet à Hector de collecter les informations de charge des tâches (statuts des threads, temps d'exécution et de communication) : la collecte est rendue possible grâce à un système de fichiers particulier appelé « /proc » propre aux systèmes UNIX et décrit par [Stevens 92]. Chaque tâche est représentée par un descripteur de fichier qui permet de prendre connaissance de ses charges processeur et mémoire
- La librairie de gestion encapsule les communications entre les tâches.

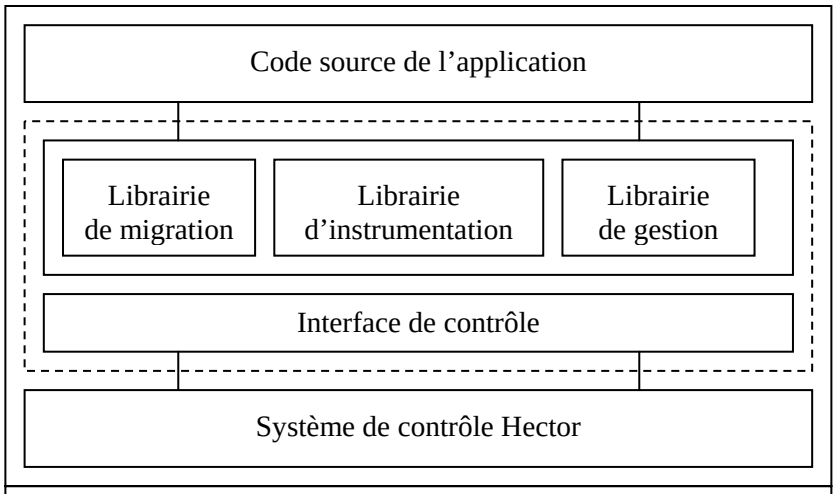

Figure 7 : intégration d'Hector dans l'application développée

3.3.3.2 Niveau des agents :

Les agents, ou « slave allocators », collectent les informations fournies par la librairie de gestion de chaque tâche locale et les envoient périodiquement par messages asynchrones au « master allocator ».

Les informations collectées par les agents concernent les tâches (via un mécanisme de mémoire partagée) et la charge des machines (via les appels système utilisant la fonction getkval() spécifiée par [Stevens 92] qui permet l'accès aux informations de charges processeur et mémoire du noyau UNIX).

Le second rôle des agents est de propager les commandes du « master allocator » aux tâches.

3.3.3.3 Niveau du « master allocator » :

Le « master allocator » collecte les informations envoyées par les agents et met à jour une base de données centralisée afin de prendre des décisions de réallocation des tâches.

Les commandes de migration sont générées par l'appel périodique d'une fonction d'optimisation qui analyse les informations stockées. En particulier, lorsqu'une nouvelle tâche est exécutée ou lorsqu'une

machine devient trop chargée ou, au contraire, trop peu chargée, des décisions de migration (commandes) sont prises.

L'algorithme d'optimisation se déroule en trois étapes :

- La première détermine la répartition théorique optimale des tâches sur chaque machine *m* à l'aide de la formule $Ideal_m = \left(\frac{Power_m}{\sum_{n=1}^{N} Power_n} \right) \times \sum_{p=1}^{P} Task_p$ où :
  - *m* désigne une machine parmi les *M* machines participant aux allocations de tâches
  - *N* est le nombre total de machines
  - *P* est le nombre de tâches
  - $Power_m$ désigne la puissance CPU ou mémoire de la machine *m*, selon les limitations de la machine
- Bien souvent $\sum_{m=1}^{M} Ideal_m < P$: il faut alors assigner les tâches restantes aux machines ayant le moins de tâches après la phase de répartition théorique optimale
- La troisième phase recherche une machine m_1 ayant un grand déficit de tâches et une autre machine m_2 ayant un petit déficit de tâches.

  Le déficit est calculé comme suit : $Deficit_m = Ideal_m - Effective_m$, $Effective_m$ étant le nombre de tâches assignées par les deux phases précédentes. Certaines tâches affectées à m_1 sont alors assignées à m_2 afin de maintenir la balance des charges des machines.

Lorsque toutes les décisions sont prises, chaque commande de migration est envoyée à l'agent correspondant afin qu'il la communique à son tour à la tâche cible.

Lorsque toutes les commandes ont été exécutées, le cycle de mise à jour, prise de décision et envoi des commandes peut alors reprendre.

3.3.4 Principe de migration de tâche :

La migration d'une tâche d'une machine (source) vers une autre machine (destination) est à l'initiative du « master allocator ». Elle est réalisée par la tâche elle-même à l'aide de la librairie de migration fournie par Hector.

Lorsqu'une commande de migration est reçue par une tâche, cette dernière [Robinson 96] :

- S'isole du système en stoppant les communications avec les autres tâches ; ceci est réalisé à l'aide de la librairie de gestion des communications
- Contacte son agent local afin qu'il demande à l'agent situé sur la machine destination de créer une tâche de même type
- Est contactée par la copie afin qu'elle lui envoie son état courant (segment de données, segment de pile et registres).

Enfin la copie remplace son état par celui donnée par la tâche initiale et rétablit les liens avec les tâches dont les communications avec la tâche initiale avaient été suspendues.

3.3.5 Conclusion et perspectives :

La charge engendrée par le système est faible comparé aux gains de la réorganisation des tâches sur les machines.

Ce système reste néanmoins spécialisé dans la répartition de programmes écrits en C et exécutés sous environnement UNIX.

Deux grands axes de recherches complémentaires sont proposés :

- Répartir totalement la base de donnée ainsi que la fonction d'optimisation chez les différents « slave allocators »
- Améliorer la fonction d'optimisation en :
  - Autorisant le groupement de tâches afin de raisonner en termes d'agrégat
  - Estimant le degré de complétion du programme afin de prédire l'arrêt des tâches.
  - Implémentant le modèle adaptatif étudié par [Lambert 98] qui permet de diagnostiquer les pertes de performances du programme et de prédire l'impact des décisions d'allocation des tâches.

3.4 Quartz :

3.4.1 Objectif :

Quartz [Siqueira 99] est un environnement générique chargé de vérifier qu'une application est conforme à certains critères de QoS prédéfinis, dépendants de l'application surveillée. Chaque critère est contrôlé par Quartz qui, si besoin est, supervise la réservation des ressources système et réseau lorsque les valeurs attribuées aux critères sont en deçà de valeurs critiques.

Quartz à pour objectif de répondre aux problèmes suivants :

- Les critères de QoS sont fortement dépendants des applications et sont définis à bas niveau
- L'amélioration de la QoS des ARO ne prend pas en compte à la fois les machines et le réseau
- Les systèmes de surveillances sont tributaires des systèmes d'exploitation des machines et ne sont ainsi aucunement portables
- Ces systèmes ignorent souvent la possibilité d'adaptation dynamique des ressources lorsque se posent des problèmes tels que le manque de ressources ou la reconfiguration du SRO.

Le domaine d'application de cette architecture est la surveillance des applications temps réel ou fortement contraintes (des applications manipulant du son et de la vidéo par exemple).

3.4.2 Architecture :

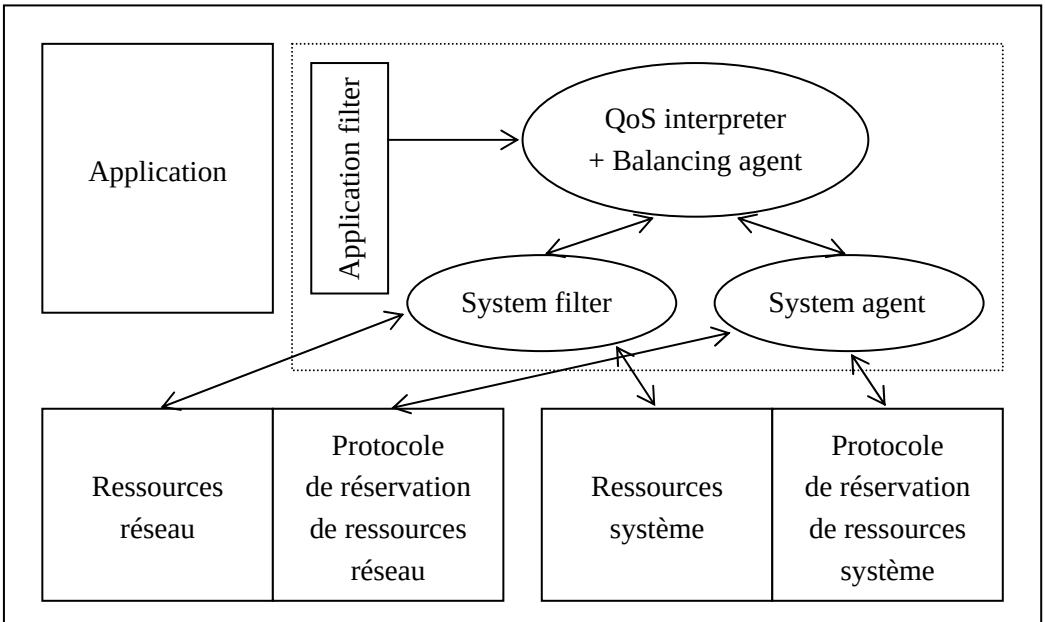

Figure 8 : architecture de Quartz

Quartz repose sur l'utilisation d'une multitude de composants prédéfinis ou construits a posteriori qui permettent d'adapter l'architecture en fonction de l'application surveillée ainsi que des protocoles de réservation de ressources, des réseaux et des systèmes d'exploitation utilisés.

Quartz se compose d'une partie centrale (ou noyau) constituée de deux modules (figure 8) :

- Le « QoS interpreter » permet de répartir les paramètres de QoS génériques en deux catégories selon qu'ils concernent le réseau ou le système. Ces paramètres seront ensuite transmis aux composants « System filter » et « System agent »
- Le « Balancing agent » est utilisé par le « Qos interpreter » pour fournir plus ou moins de paramètres de QoS aux « System filter » et « System agent » en fonction des ressources nécessaires au fonctionnement correct de l'application surveillée.

A ce noyau central s'ajoutent :

- Des filtres de transposition des critères de QoS exprimés au niveau de l'application en critères génériques compréhensibles par Quartz : ce sont les « Application filters »
- Des « System filters » qui sont soit des « network filters » soit des « system filters » selon qu'ils interagissent avec le réseau de communication ou la machine : chaque filtre système traduit les paramètres de QoS donnés par le noyau en paramètres compréhensibles par le réseau et le système d'exploitation sous-jacents (ATM, TCP/IP, RSVP, etc.)
- Des « System agents » qui prennent en charge la réservation des ressources réseau et système nécessaires à l'application surveillée, en accord avec les critères de QoS spécifiés. De plus, les agents ont un rôle de transmission des modifications réseau et système qui peuvent survenir au cours de l'exécution de l'application.

Le noyau et les différents filtres utilisés forment un « QoS agent ». Chaque « QoS agent » peut être considéré comme un réceptacle dans lequel se greffent les composants.

Cette conception modulaire permet d'adapter dynamiquement l'architecture en fonction des changements dans son environnement : changement de protocole de réservation de ressources et changement de système d'exploitation.

L'application surveillée peut être répartie sur plusieurs machines. Dans ce cas, Quartz utilise un « QoS agent » par machine.

3.4.3 Fonctionnement :

Quartz permet aux utilisateurs de spécifier leurs contraintes de QoS à haut niveau, en indiquant par exemple le nombre d'images par secondes et la résolution pour une vidéo ou un certain niveau de qualité pour de l'audio au lieu de spécifier la bande passante nécessaire.

Pour ce faire, Quartz utilise des composants chargés d'effectuer la traduction des critères spécifiques en critères génériques compréhensibles par le module de réservation de ressources (« QoS interpreter »).

Lorsque Quartz détecte que l'un des paramètres spécifiés ne répond pas aux contraintes de QoS (bande passante trop faible par exemple), il contacte soit l'application pour lui demander de se réorganiser (le processus de réorganisation est effectué par l'application et non par Quartz), soit le réseau et le système afin de réserver les ressources nécessaires.

Les paramètres de QoS génériques de Quartz sont organisés en deux groupes :

- Les paramètres relatifs à l'application surveillée
- Les paramètres système et réseau.

3.4.3.1 Paramètres application :

Chaque paramètre de QoS exprimé au niveau de l'application est traduit par un paramètre générique compréhensible par Quartz. Les paramètres génériques utilisés, relatifs à l'application, sont :

- Le nombre d'octets constituant un paquet manipulé par l'application surveillée
- La fréquence de production des paquets par l'application (en paquets par seconde)

- Le temps (en millisecondes) séparant la production de la consommation des paquets
- Le nombre d'erreurs (en bits par millions de bits)
- Le niveau de QoS garantie (au mieux, optimum, etc.)
- Le coût financier d'utilisation de l'application (en <monnaie> par seconde)
- Le niveau de sécurité (aucune, différents niveaux de cryptage des messages transmis).

3.4.3.2 Paramètres système et réseau :

Ces paramètres de QoS reflètent l'état de la machine et du réseau sous-jacents. Ils sont la traduction bas niveau des paramètres de haut niveau exprimés au niveau de l'application : par exemple, le niveau de qualité sonore ou la résolution vidéo, exprimés comme tels au niveau application (CD_QUALITY et 1024x768) sont traduits par Quartz en termes de bande passante.

Les paramètres génériques système et réseau retenus sont :

- Le degré de QoS garantie par Quartz : « au mieux », déterministe, etc.
- Le délai minimum et maximum de calcul sur la machine (en secondes)
- Le coût financier maximum d'utilisation de la machine (en <monnaie> par seconde)
- La bande passante réseau minimum et maximum disponible (en octets par seconde)
- La taille minimum et maximum des paquets réseau (en octets)
- Le temps de transmission réseau minimum et maximum (en secondes)
- Le taux d'erreurs maximum autorisé (en bits erronés par millions de bits)
- Le coût financier maximum d'utilisation du réseau (en <monnaie> par seconde)
- Le niveau de sécurité lors des transmissions réseau (aucune, différents niveaux de cryptage des messages transmis).

3.4.4 Conclusion et perspectives :

Une implémentation de Quartz sous l'environnement réparti CORBA Orbix 2.3 pour C++ d'Iona a été réalisée. Elle supporte le système d'exploitation Windows NT 4.0 ainsi que les protocoles réseau TCP/IP et UDP/IP. Cette implémentation a été utilisée avec succès avec diverses ARO multimédia nécessitant une synchronisation entre son et vidéo.

La flexibilité de Quartz lui permet de s'adapter à tout type d'application. Les améliorations de Quartz possibles concernent uniquement la mise à disposition de ce système sur différentes plates-formes ainsi que le support des applications temps réel et de plusieurs protocoles réseau (RSVP en particulier).

3.5 LoDACE :

3.5.1 Objectif :

Le système LoDACE [Badidi 98] [Badidi 99], Load Distribution Architecture for a Distributed Object Computing Environment, fournit une infrastructure CORBA permettant le partage de charge (répartition d'invocation ou « load sharing ») automatique entre des machines offrant à des objets clients un ensemble de services de même type proposés par des objets appelés serveurs.

Une ARO est considérée comme composée :

- Soit d'un unique serveur et de plusieurs clients (application client/serveur)
- Soit de plusieurs serveurs et plusieurs clients.

Un client peut obtenir un service donné en appelant un serveur proposant ce service. Chaque serveur peut proposer plusieurs services.

3.5.2 Architecture :

LoDACE est basé sur l'utilisation du service de médiation ou « trading service » [OMG 97a] [OMG 97b] [OMG RFP5] pour enregistrer les services proposés par les serveurs et sélectionner le serveur le plus apte à répondre à une demande de service d'un client (figure 9). Ce service fait correspondre à un service donné l'ensemble des serveurs qui le fournissent. La sélection d'un serveur parmi ceux proposant le même service demandé repose sur la surveillance de la charge des serveurs et des machines.

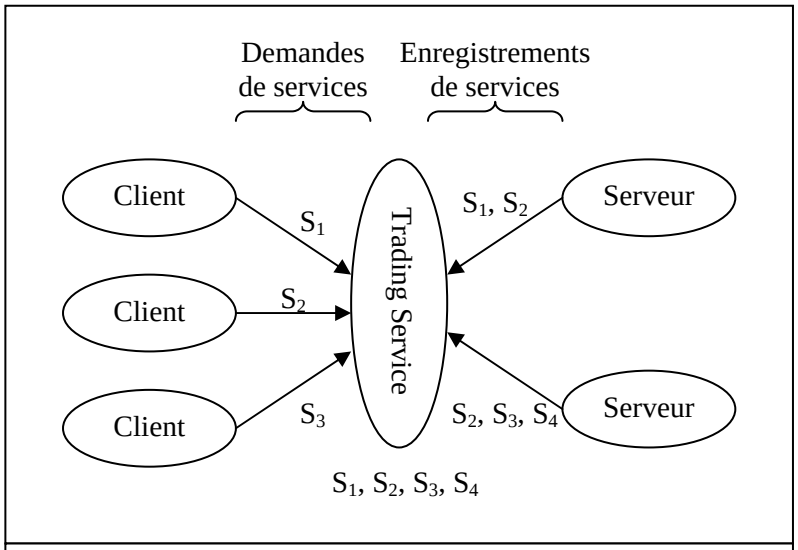

Figure 9 : architecture de LoDACE

Le mécanisme d'assignation d'un serveur à un client repose non seulement sur le service de médiation mais également sur quatre modules :

- Le « Binder » est une interface entre les clients et le service de médiation ; en effet, les clients ne doivent pas être directement en contact avec ce service puisqu'une sélection d'un serveur parmi ceux enregistrés doit être opérée par LoDACE
- Sur chaque machine, un « Host Load Monitor » (HLM) collecte les informations du système d'exploitation concernant la charge de la machine : utilisation du processeur, activité de pagination, entrées/sorties réalisées sur le disque, mémoire disponible, quantité de mémoire swap
- Présent sur chaque machine également, un « Load Monitor » (LM) recueille la charge de la machine collectée par le module HLM ainsi que la charge des serveurs (taux d'occupation du serveur et nombre de demandes par unité de temps)
- Un module central, le « Load Manager » (LMG), collecte les informations que lui fournissent périodiquement, ou à sa demande, les différents LM du système. Ces informations lui permettent de sélectionner le serveur le moins chargé fournissant le service demandé et de communiquer son adresse au module « Binder » afin qu'il la transmette au client. La sélection d'un serveur utilise plusieurs stratégies décrites ci-après.

3.5.3 Fonctionnement :

La gestion des serveurs par le module LMG fait appel à trois ensembles :

- $T = \{t_1, t_2, ...\}, Card(T) = m$: ensemble des services enregistrés par le service de médiation
- $S_{t_1} = \{S_1, S_2, ...\}, Card(S_{t_1}) = k$: ensemble des serveurs offrant le service t_i

- $PSL(S_{j_i}) = S_{t_i} - \{S_{j_i}\}$: ensemble des serveurs différents du serveur *j* offrant le service t_i.

Ces ensembles sont mis à jour périodiquement par chaque serveur afin de prendre en compte les ajouts et retraits de services lorsque des serveurs s'enregistrent ou se déconnectent auprès du service de médiation.

Ils sont utilisés par les objets serveurs et le « Binder » lors du mécanisme de partage de charge qui repose sur l'utilisation de quatre stratégies :

- *Placement initial* : lors d'une demande de service, le « Binder » retourne au client l'ensemble S_{t_i} des serveurs proposant le service t_i demandé. Cet ensemble est trié par charge croissante des serveurs. Le client se connecte alors au premier serveur qui accepte de répondre à sa demande de service en partant de la tête de la liste fournie par le « Binder »
- *Source initiative* : lorsqu'un serveur *i* surchargé (dont la charge est supérieure à un seuil maximum fixé au préalable) reçoit une demande d'un client, il choisit le premier serveur de son PSL, trié par charge croissante, qui accepte de déléguer la demande du client. Lorsqu'un serveur *j* répond favorablement, *i* remplace son adresse par celle de *j* dans le service de médiation. Les demandes futures seront alors adressées directement à j plutôt qu'à *i*
- *Receveur initiative* : un serveur *i* légèrement chargé (dont la charge est inférieure à un seuil minimum fixé au préalable) prend l'initiative de demander aux autres serveurs de son PSL, trié par ordre décroissant de charge des serveurs, s'ils acceptent de lui déléguer leurs demandes futures de service. Le premier serveur *j* qui répond favorablement remplace alors son adresse par celle de *i* au sein du service de médiation
- *Symétrique* : cette stratégie vise à combiner les stratégies *source initiative* et *receveur initiative* en fonction de la charge des serveurs.

3.5.4 Conclusion et perspectives :

Ce système permet de gérer des objets offrant un même service et répartis sur plusieurs machines. Il peut être considéré comme étant tolérant aux pannes lorsque la configuration initiale assure la disponibilité de chaque service sur au moins deux machines.

Un prototype utilisant l'environnement CORBA OrbixWeb de IONA Technologies et OrbixTrader est en cours de développement.

Cette architecture ne permet cependant pas de manipuler les objets répartis en effectuant des transferts de charge entre machines. Une extension prochaine de LoDACE sera d'intégrer un processus de migration et de réplication des serveurs sur les machines du réseau.

3.6 LDCE :

3.6.1 Objectif :

LDCE [Rackl 97], Load Distribution for CORBA Environments, propose un modèle permettant de simuler des stratégies de répartition de charge dans les applications CORBA.

Chaque ARO est considérée comme composée d'objets service et client. Un service est équivalent à un objet fournisseur et un client à un objet demandeur. Les machines utilisées pour le déploiement de l'ARO sont appelées serveurs. Un type de service est une fonctionnalité proposée par un service.

3.6.2 Architecture :

Le système LDCE repose sur l'utilisation du service de médiation, ou « Trading Service », de CORBA [OMG 97a] [OMG 97b]. Il se compose (figure 10) :

- D'un « trader » qui implémente le service de médiation. Ce « trader » propose un ensemble de types de services $T = \{T_1, T_2, ..., T_n\}$ associés à chaque serveur par la fonction $t(T_i) = \{S_x, ...\}$ (un type de service T_i est proposé par un ensemble de serveurs S_x,…)
- D'un ensemble de serveurs (ou machines) $S = \{S_1, S_2, ..., S_m\}$. Une fonction *p* associe à chaque serveur l'ensemble des types de services qu'il fournit : $p(S_i) = \{T_x, ...\}$, c'est à dire l'ensemble des objets fournisseurs enregistrés sur chaque machine. Une fonction *m* indique de plus l'ensemble des types de services autorisés : $m(S_i) = \{T_x, ...\}$. Un service proposant un type de service non autorisé sur un serveur *S* ne pourra être migré sur ce serveur
- D'un ensemble de clients : un client utilise un type de service proposé par un serveur
- D'un ensemble de fabriques d'objets exécutées sur chaque serveur. Chaque fabrique prend en charge la création d'un nouveau service sur son serveur local
- D'un superviseur chargé :
  - De collecter les informations de charge des serveurs
  - D'effectuer les déplacements des services sur les serveurs
- D'un évaluateur qui enregistre les informations en provenance des clients, des serveurs et du superviseur
- D'un contrôleur prenant en charge le processus de simulation.

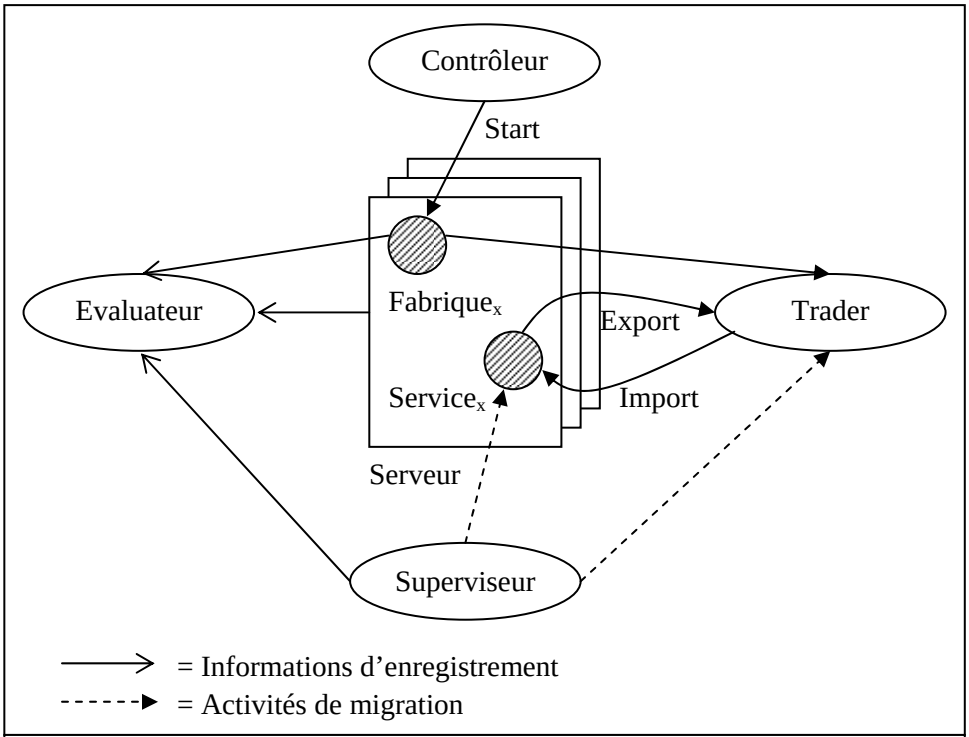

Figure 10 : architecture de LDCE

Les opérations « import » et « export » sont spécifiques au « trader » :

- L'opération « import » permet à un client de recevoir les références des services fournissant un type de service demandé
- L'opération « export » est utilisée par les services afin d'enregistrer auprès du « trader » les types de services qu'ils proposent.

3.6.3 Fonctionnement :

LDCE met en avant la division possible de chaque stratégie de répartition de charge (« source initiative », « receveur initiative » et « symétrique » décrites lors de présentation de l'architecture LoDACE) en quatre politiques :

- La politique d'information : indique au système de surveillance le moment opportun pour la collecte des informations qui peut être réalisée à la demande, périodiquement, ou lorsqu'un changement significatif d'état du SRO survient
- La politique de transfert : renseigne le système de surveillance sur le moment propice à la migration d'un objet, par exemple lorsqu'un indice de charge dépasse un seuil prédéfini
- La politique de sélection : lorsque la politique de transfert a été exécutée, cette politique a pour objectif de sélectionner un objet parmi l'ensemble des objets de l'ARO
- La politique de placement : permet de sélectionner la machine devant recevoir l'objet transféré.

Pour réaliser la simulation, un indice de charge des serveurs, appelé « load index », est utilisé. Il représente le pourcentage d'utilisation moyenne de chaque serveur durant les soixante dernières secondes selon la formule $load\ index = N/60$, N étant le nombre de secondes utilisées par le service pour réaliser ses traitements durant les dernières soixante secondes.. Cet indice sert au superviseur pour appliquer une stratégie de migration des services.

Cette stratégie, dont l'objectif est de déplacer des services situés sur les serveurs les serveurs les plus chargés vers les serveurs les moins chargés (capables d'accepter les types de services fournis par les services à déplacer) se décompose selon l'algorithme suivant :

- Les serveurs sont classés du plus au moins chargé
- Chaque serveur dont la charge dépasse un seuil fixé est contrôlé afin de déterminer son habilité à migrer l'un de ses services
- Les serveurs ayant répondu favorablement sélectionnent alors chacun le service le plus consommateur de ressources
- Chaque service sélectionné est alors déplacé vers les serveurs les moins chargés aptes à les recevoir. Les serveurs destinations sont ceux dont la charge est en deçà de la charge des serveurs source. Dans le cas contraire, la migration d'un service entraînerait la surcharge du serveur destination.

Cet algorithme, appelé « Cooling Algorithm », est exécuté périodiquement par le superviseur.

3.6.4 Principe de migration des services :

Les fabriques utilisent la politique « server-per-method » d'activation des objets créés : chaque création d'objet est réalisée dans un nouveau processus qui sera détruit dès lors que l'objet créé sera activé. Les fabriques sont contactées par le superviseur via des communications non bloquantes.

La migration d'un objet d'un serveur à un autre s'effectue en sept étapes :

- Le superviseur prend la décision de migrer un service quelconque depuis un serveur *src* sur un serveur *dest*
- Il contacte *dest* et lui demande de créer un nouveau service de même type que le service initial. Pour ce faire, *dest* utilise la fabrique correspondant au service à créer
- *dest* demande à *src* de lui fournir l'état courant du service. Dès lors, chaque client qui fera appel au service situé sur *src* recevra un message d'erreur
- *dest* met ensuite à jour l'état du service nouvellement créé à l'aide des informations fournies par *src*
- Il contacte alors le « trader » afin qu'il mette à jour la fonction *t*
- Lorsque l'enregistrement de la migration par le « trader » est complétée, *dest* informe *src* du succès de la migration

- *src* détruit alors l'ancien service. Si un client appelle l'ancien service après sa destruction, un message système lui sera renvoyé, stipulant que le service demandé n'existe pas. Le client importera alors à nouveau le service via le « trader » et recevra la nouvelle référence du service.

3.6.5 Conclusion et perspectives :

LDCE a pour principal objectif de simuler le comportement d'un système de surveillance d'une ARO. Chaque information manipulée est soit fixée soit déterminée à l'aide d'une formule mathématique :

- Le transfert de l'état des services lors de leur migration n'est pas pris en compte : l'état est modélisé par une chaîne de caractères constante. Tous les services ont ainsi un coût de migration fixe
- Chaque appel d'un service génère l'envoi d'une chaîne de caractères de taille fixe
- A chaque service est associé une constante qui indique sa consommation processeur.

La migration d'un service demande l'activation préalable de la fabrique correspondante sur le serveur destination afin de permettre la création d'un nouveau service de même type que le service devant être déplacé.

Les résultats des simulations mettent néanmoins en évidence la suprématie de la stratégie « source initiative » par rapport à la stratégie « receveur initiative » lorsque le SRO est chargé, l'inverse se vérifiant lorsque le SRO est peu chargé. L'utilisation d'une stratégie « symétrique » combine les performances des deux autres stratégies et semble le meilleur choix à adopter.

Les travaux concernent la mise en pratique des résultats obtenus en définissant un système réellement opérationnel.

3.7 Apports de DOMS :

Chacune des architectures présentées s'attache à un point particulier d'amélioration de la QoS des applications réparties et de ce fait peut être considérée comme étant représentative de quelques axes de recherches concernant la surveillance et la gestion des applications réparties.

Les architectures LYDIA et LoDACE s'intéressent toutes deux à la répartition des invocations (« load sharing ») en prenant ou non en compte la notion d'objet. Le processus de migration de tâche ou d'objet nécessaire à la répartition d'objet n'intervient pas.

Dome est un exemple d'architecture de partage de charge dans des applications conçues spécialement pour être supervisées par ce système.

Hector permet la migration de processus UNIX (« load balancing »). Cependant la migration est effectuée indépendamment de la nature de la tâche active dans chaque processus migré. La gestion de la nature des processus et des liens entre eux est négligée au profit de la seule charge des machines.

LDCE propose un point de vue intéressant concernant le principe de migration des objets répartis adopté lors des simulations. Le protocole de migration des objets répartis de DOMS en est dérivé.

DOMS est la spécification d'un système de surveillance et de gestion des applications réparties objet. Il surveille les applications réparties objets et les contrôle afin de les répartir « au mieux » sur les machines disponibles. Il prend en compte non seulement la charge des machines mais également le degré de corrélation entre les objets pour déterminer dynamiquement la répartition des objets sur les machines du réseau. Il utilise pour cela les principes complémentaires de répartition d'invocation et de répartition d'objet. Ces derniers seront traduits par des algorithmes spécifiques.

DOMS propose également un modèle de représentation des applications réparties objet et de calcul d'indices génériques pouvant servir de base à d'autres travaux concernant la surveillance et la gestion des applications réparties.

4 Le modèle abstrait DOMM :

4.1 Introduction :

L'objectif de DOMM (Distributed Objects Management Model) est de fournir un formalisme et des outils aptes à limiter (1) les temps de réponse des objets répartis tout en évitant de surcharger les machines du réseau et (2) les communications distantes entre les objets via le réseau.

La conception de tels outils repose sur l'étude des objets répartis et des relations existant entre eux. Il en résulte la définition d'indices permettant de mesurer la QoS des objets composant les ARO en termes de performances d'exécution et de fiabilité.

[Rackl 97] propose trois indices généraux, sans toutefois indiquer comment les calculer :

- La longueur de la file d'attente d'un objet, lorsque plusieurs requêtes sont en attente de traitement (note : selon la politique utilisée, une instance de l'objet peut être créée lors de chaque invocation de méthode ; dans ce cas, la file d'attente n'existe plus)
- Le degré de charges processeur et mémoire des machines pendant les *k* dernières secondes
- Le temps de réponse à une requête : le temps que met le résultat d'une invocation à parvenir à l'objet appelant.

Les recherches menées par Rackl et [Casavant 87] aboutissent à la conclusion qu'il vaut mieux utiliser peu d'indices simples et bien choisis plutôt que beaucoup d'indices complexes issus de combinaisons d'indices simples.

L'utilisation de peu d'indices permet de plus de limiter les temps de collecte, de transfert et de traitement des informations (calcul de la solution « au mieux » et non de la solution optimale).

Les deux premiers indices proposés (longueur de la file d'attente des objets et degrés de charges processeur et mémoire des machines) influencent fortement le dernier indice dans le cadre de notre étude car (1) plus la longueur de la file d'attente d'un objet est grande, plus la requête en queue de file mettra du temps à être traitée et (2) plus la machine est chargée moins le processeur accordera de temps à l'exécution des requêtes.

DOMM se limite donc au seul temps moyen global de réponse aux appels, à minimiser. Le calcul de cet indice suppose la connaissance d'autres indices tels que le degré de répartition des objets et les temps moyens d'exécution des méthodes distantes (la notion de file d'attente est dans ce cas sous-jacente).

4.2 Classement des objets répartis :

Les objets répartis peuvent être définis de différentes manières selon que l'on se place du point de vue utilisation (application) ou du point de vue gestion.

Deux grandes catégories en découlent :

- Point de vue application (figure 1) : vu du développeur, un objet est défini par ses parties publique (qui comprend généralement l'interface permettant de l'appeler) et privée (implémentation des méthodes de l'interface) ainsi que par sa référence ; le développeur s'intéresse aux fonctionnalités plutôt qu'aux problèmes de déploiement de son application sur les machines du réseau
- Point de vue gestion : lorsque l'on souhaite surveiller les objets d'une ARO, un objet est défini (1) par les ressources nécessaires à son bon fonctionnement et à son activité et (2) par les communications initiées avec d'autres objets. Les ressources utilisées par l'objet (charge induite) concernent à la fois la charge du processeur et la charge mémoire de la machine sur laquelle celui-ci réside. Son activité fait référence au nombre d'appels qu'il génère et qu'il reçoit. Les communications inter-machines et intra-machines doivent être prises en compte par le système car elles influencent fortement les temps de réponse des objets appelés.

Le modèle utilise le second point de vue pour définir les indices et seuils permettant de déterminer la localisation « au mieux » des objets sur les machines.

Que ce soit des points de vue application ou gestion, on distingue trois types d'objets, selon qu'ils invoquent ou non des méthodes distantes :

- Les objets demandeurs (ou « objets clients ») sont des objets qui ne sont jamais appelés car ils ne disposent d'aucune interface de communication permettant de le faire ; de tels objets disposent en général d'une IHM permettant le lien entre l'utilisateur et l'ARO. Ces objets n'ont pas de référence car ils ne sont jamais appelés par d'autres objets, contrairement aux deux autres types d'objets
- Les objets fournisseurs (ou « objets serveurs ») n'appellent aucun objet ; par contre les méthodes décrites dans leur interface de communication sont invoquées par d'autres objets
- Les objets délégateurs sont à la fois fournisseurs et demandeurs : ils sont fournisseurs en ce sens que les méthodes de leur interface peuvent être appelées et demandeurs car ils appellent eux-mêmes des objets qualifiés de « délégataires » qui fournissent une partie du travail qui est demandé aux objets délégateurs.

Les objets délégateurs se décomposent en trois sous-classes :

- Les objets réactifs invoquent les méthodes d'objets délégataires (fournisseurs ou délégateurs eux-mêmes) uniquement lorsqu'ils sont sollicités par des requêtes émanant d'objets demandeurs ou délégateurs afin de réaliser complètement le traitement qui leur est demandé : les appels aux objets délégataires sont réalisés au sein du corps des méthodes de l'interface de l'objet délégateur
- Les objets autonomes invoquent des méthodes d'objets délégataires indépendamment des méthodes de leur interface : ils fournissent un travail coopératif même s'ils ne sont pas appelés par d'autres objets
- Les objets mixtes sont à la fois réactifs et autonomes.

Ainsi, la définition d'une application répartie objet telle qu'énoncée précédemment, est à nuancer : l'ensemble O^t des objets composant une ARO à un instant t donné doit en effet comporter au moins deux éléments mais il doit de plus compter au moins un objet « fournisseur ».

Théorème1 : une application répartie objet comporte un objet fournisseur et soit un objet demandeur, soit un objet délégateur autonome.

Preuve :

- Supposons qu'une ARO ne comporte aucun objet fournisseur : dans ce cas l'application ne fournit aucun travail car aucun objet ne sera en mesure de terminer le travail demandé ; un objet de type fournisseur peut être assimilé à une condition d'arrêt du traitement demandé. Une invocation d'un objet ne se termine que lorsque :
  - Un objet fournisseur est appelé : dans ce cas, il exécute la méthode appelée
  - Un objet délégateur est appelé : la complétion de la méthode appelée ne sera effective que lorsque tous les objets fournisseurs ou délégateurs appelés en son sein auront eux-mêmes terminé leur exécution
- Supposons ensuite qu'une ARO ne comporte que des objets fournisseurs : de même, aucun traitement ne pourra être réalisé car aucun d'eux ne sera en mesure de l'initier. Un objet apte à invoquer un objet fournisseur doit donc être présent : ce peut être soit un objet demandeur soit un objet délégateur autonome.

Théorème2 : une application répartie objet constituée de k objets comporte l liens de communication tels que $0 < l \leq k(k-1)$.

Preuve :

- Une ARO comporte toujours au moins un objet fournisseur (théroème1) qui sera appelé par au moins un objet demandeur ou délégataire autonome : ainsi au moins un lien de communication peut être établi dans une ARO, donc $0 < l$
- Le nombre de liens dans un graphe complet de *k* sommets est de *k(k-1)* [Adoud 00]. Ainsi $l \leq k(k-1)$.

4.3 Modélisation des communications entre les objets répartis :

Il existe deux types primaires permettant de classifier les communications entre les objets répartis d'une ARO :

- L'invocation partielle, notée $\rightarrow$: un objet appelle une méthode distante d'un autre objet mais n'attend aucun retour ni acquittement. Dans ce cas, l'objet appelant ne dispose d'aucun moyen de contrôle de l'exécution de la requête, même si ce type de communication est fiabilisé par les environnements répartis
- L'invocation totale ou complète, notée $\Rightarrow$: un objet appelle une méthode distante d'un autre objet et attend un retour (ce peut être une valeur de retour ou simplement un acquittement lorsqu'aucun résultat n'est attendu)
- L'exclusion, notée $\circ-$, indique qu'un objet ne communique pas avec un autre objet.

Ces types primaires permettent de dériver quatre types secondaires qui décrivent les liens entre deux objets A et B :

- L'invocation unidirectionnelle partielle $A \rightarrow \circ B$, équivalente à $(A \rightarrow B) \wedge (B \circ - A)$: A appelle B partiellement et B exclut A
- L'invocation unidirectionnelle totale $A \Rightarrow \circ B$, équivalente à $(A \Rightarrow B) \wedge (B \circ - A)$: A appelle B totalement et B exclut A
- L'invocation bidirectionnelle, par exemple $(A \rightarrow B) \wedge (B \Rightarrow A)$: les objets A et B s'appellent mutuellement, que les invocations soient partielles ou complètes.

Par extension, une invocation est dite « mutuelle » ou « symétrique » lorsqu'elle est bidirectionnelle et que de plus la communication entre les deux objets est de même type primaire :

- Invocation mutuelle partielle : $A \leftrightarrow B$, équivalente à $(A \rightarrow B) \wedge (B \rightarrow A)$
- Invocation mutuelle totale : $A \Leftrightarrow B$, équivalente à $(A \Rightarrow B) \wedge (B \Rightarrow A)$.

Une exclusion est « mutuelle » ou « symétrique » lorsque deux objets A et B s'excluent l'un l'autre : $A \circ - \circ B$ ou $(A \circ - B) \wedge (B \circ - A)$. Il suit que tous les objets de type « demandeur » s'excluent mutuellement car ils ne sont jamais appelés (ils ne possèdent pas d'interface de communication).

Note : les signes décrivant les types d'invocations peuvent être écrits dans les deux sens, ainsi le lien « A invoque totalement B » peut se noter indifféremment $A \Rightarrow B$ ou $B \Leftarrow A$.

4.4 Modélisation du comportement de l'application répartie objet :

4.4.1 Ensembles manipulés :

En sus des ensembles *M*, *N*, *O* décrivant un système réparti objet (SRO), d'autres ensembles propres à chaque objet peuvent également être déclarés :

- I_{obj} : ensemble des méthodes composant l'interface de l'objet *obj*
- C_{obj} : ensemble des objets susceptibles d'être appelés par l'objet *obj* et qui détermine l'ensemble *L* par la relation $L = \bigcup_{i=1}^{k} L_{i,j}, \forall j \in C_i$; la relation $obj \notin C_{obj}$ est vérifiée pour tout objet *obj* de l'ARO surveillée

- $C_{obj}^{t} \subseteq C_{obj}$: ensemble des objets appelés par l'objet *obj* depuis l'instant de son dernier changement de machine jusqu'à l'instant *t*
- E_{obj} : ensemble des exceptions susceptibles d'être levées par l'objet *obj*
- $E_{obj}^{t} \subseteq E_{obj}$: ensemble des exceptions déjà levées par l'objet *obj* jusqu'à l'instant *t*.

Les caractéristiques des différents types d'objets sont alors définies comme suit ($\perp \equiv$ indéterminé) :

- Objets demandeurs : $demandeur_{obj} \Leftrightarrow \left(I_{obj} = \varnothing\right) \wedge \left(C_{obj} \neq \varnothing\right) \wedge \left(E_{obj} = \perp\right)$
- Objets fournisseurs : $fournisseur_{obj} \Leftrightarrow \left(I_{obj} \neq \varnothing\right) \wedge \left(C_{obj} = \varnothing\right) \wedge \left(E_{obj} = \perp\right)$
- Objets délégateurs : $délégateur_{obj} \Leftrightarrow \left(I_{obj} \neq \varnothing\right) \wedge \left(C_{obj} \neq \varnothing\right) \wedge \left(E_{obj} = \perp\right)$.

Ainsi, l'ensemble *O*, préalablement défini, est un ensemble d'ensembles $O_i = \left\{I_i, C_i, E_i\right\}$. supportant la propriété $\exists\left(O_1, O_2\right) \in O^2, O_1 \neq O_2, fournisseur_{O_1} \wedge \left(demandeur_{O_2} \vee délégateur_{O_2}\right)$.

Les ensembles *O* et C_i peuvent ne pas être finis lorsque l'application est composée d'un nombre infini d'objets créés dynamiquement.

Le tableau suivant récapitule, pour chaque type d'objet, les cas pour lesquels il est possible d'effectuer un déplacement.

	Réactif	**Autonome**	**IHM**	**Déplacement**
Demandeur	NON	OUI	OUI	NON
Fournisseur	OUI	NON	NON	OUI
Délégateur	NON	OUI	NON	OUI
	NON	OUI	OUI	NON
	OUI	NON	NON	OUI
	OUI	NON	OUI	NON

Un objet *obj* ne peut ainsi être déplacé que si $I_{obj} \neq \varnothing$ (objet non demandeur) et que de plus il ne dispose pas d'interface utilisateur (ce qui est le cas le plus fréquent).

Notre étude ne prend pas en compte les objets délégateurs ayant une IHM. Nous supposerons donc que seuls les objets de type demandeur ne peuvent être déplacés.

4.4.2 Graphes d'invocations :

Une ARO étant généralement composée de plus de deux objets, les liens qui les unissent peuvent être représentés par un graphe traduisant la matrice d'incidence *L* qui caractérise cette application.

La collaboration entre objets s'effectue par un mécanisme dérivé de l'appel de méthodes à distance basé sur l'envoi de messages : un objet A appelle une méthode *m()* située sur un objet B et peut selon le cas attendre ou non une réponse de B. B peut lui aussi appeler un ou plusieurs objets avant de terminer l'exécution de *m()*.

De même que l'on distingue plusieurs types d'objets répartis, il existe deux types de graphes d'invocations : le graphe fonctionnel et le graphe de gestion qui correspondent respectivement aux points de vue application et gestion précédemment utilisés pour classifier les objets.

4.4.2.1 Graphe fonctionnel :

Le graphe fonctionnel s'intéresse aux liens fonctionnels entre les objets (figure 11). Il traduit le graphe $G' = (O, L)$ caractéristique de l'ARO. On compte parmi ces liens l'appel d'au moins une méthode d'un objet par un autre objet, l'instanciation d'un ou plusieurs objets dans un autre objet, la consultation ou modification d'une variable de classe ou, dans une moindre mesure, le passage par valeur d'un objet en paramètre d'une méthode distante.

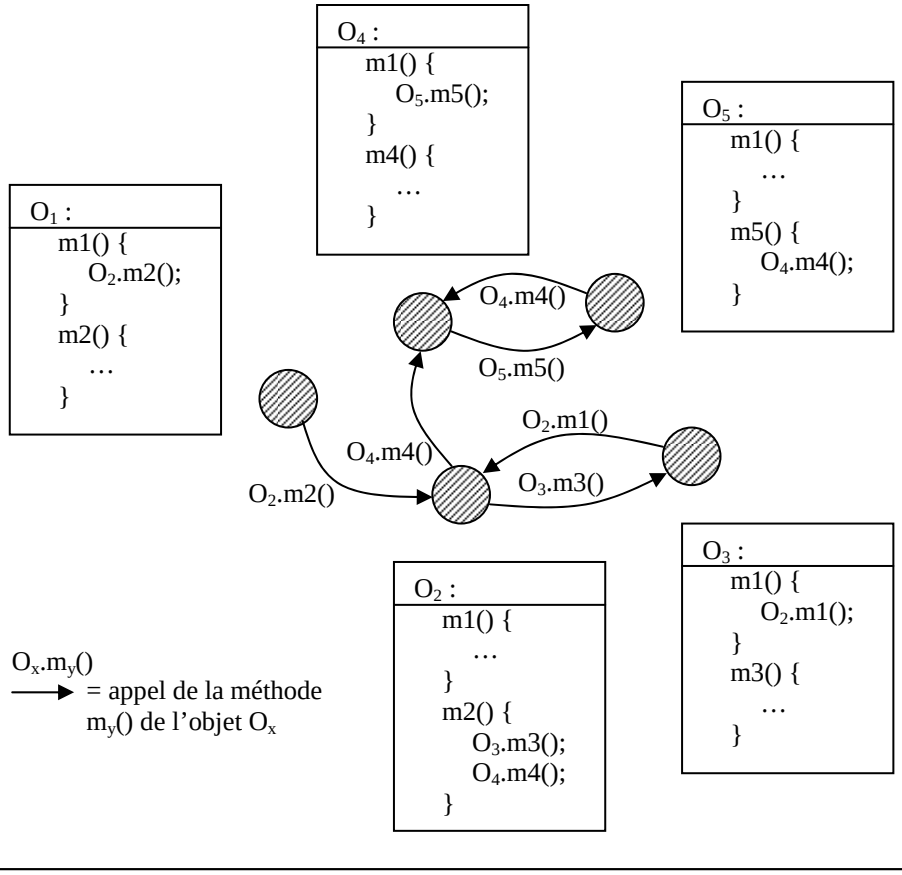

Figure 11 : exemple de graphe fonctionnel avec cinq objets

Ce graphe est cependant insuffisant pour permettre la surveillance de l'application car certaines données ne sont pas mentionnées, en particulier les informations relatives aux charges requises par les objets. Le recours à un autre graphe s'avère nécessaire.

4.4.2.2 Graphe de gestion :

Pour répondre aux besoins de surveillance et de gestion de l'ARO, le graphe fonctionnel *G'* est complété par $G = (A, L, M)$ afin d'y intégrer dynamiquement des informations concernant :

- Le degré de communications entre les objets répartis
- Le regroupement des objets sur les machines, qui permettra de distinguer les communications intra-machines et inter-machines : nous emploierons respectivement les termes d'« invocation locale » et d'« invocation distante », car bien qu'une invocation soit toujours distante lorsque l'on raisonne au niveau des objets eux-mêmes, elle peut être considérée comme locale lorsque les objets appelant et appelé sont situés sur la même machine et distante lorsque deux machines différentes les hébergent

- Les délais de communication lors des invocations.

Contrairement au graphe fonctionnel qui pouvait être obtenu de manière statique ou dynamique, le graphe obtenu est dynamique car les informations qu'il contient (les délais de communication et la localisation des objets sur les machines en particulier) sont soit inconnues lors de la phase initiale de déploiement de l'application, soit changent au cours de l'exécution de l'ARO.

Ce graphe, appelé graphe de gestion (figure 12), servira de base au procédé de répartition de objets sur les machines.

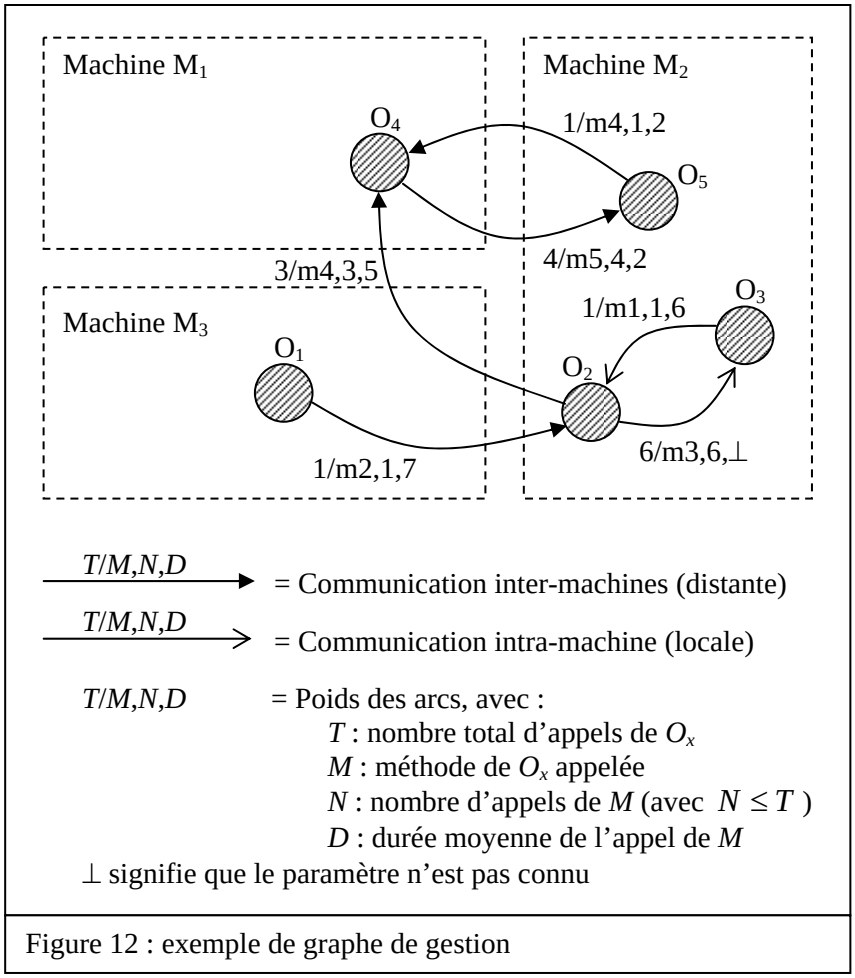

Figure 12 : exemple de graphe de gestion

Les arcs $L_{i,j}$ (i et j désignant respectivement l'objet appelant et l'objet appelé) sont pondérés par un ensemble de valeurs $T_{i,j}[/M_x, N_x, D_x]^+$ qui permettent d'évaluer le degré de corrélation entre les objets et leur degré d'appartenance à une machine (figure 8).

Cette pondération, permet de dériver des propriétés de communication qui seront utilisées pour mesurer la QoS du SRO surveillé :

- $T_{i,j} = \sum_x N_x, \forall N_x \in C_{i,j}, (i,j) \in O^2, i \neq j$: le nombre total d'appels d'un objet j par un objet i , indépendamment des machines sur lesquelles i et j résident, est égal à la somme du nombre d'appels par i_i des méthodes de j. De cet indice peut être déduite la fréquence des invocations entre objets
- $T^t_{i,j} \subseteq T_{i,j}$ indique le nombre d'appels d'un objet j par un objet i depuis que i réside sur sa machine jusqu'à un instant t donné

- $inside_i^t = \begin{cases} \sum_x N_x, \forall N_x \in C_{i,j}^t, i \in A_m^t, m \in N^t, \forall j \in C_i^t \quad si\ C_i^t \neq \varnothing \\ 0 \quad si\ C_i^t = \varnothing \end{cases}$: désigne le nombre d'invocations locales à l'initiative de l'objet O_i depuis qu'il réside sur sa machine, à un instant t donné
- $outside_i^t = \begin{cases} \sum_x N_x, \forall N_x \in C_{i,j}^t, i \in A_m^t, m \in N^t, \forall j \notin C_i^t, i \neq j \quad si\ C_i^t \neq \varnothing \\ 0 \quad si\ C_i^t = \varnothing \end{cases}$: nombre d'invocations distantes à l'initiative de l'objet O_i depuis qu'il réside sur sa machine, à un instant t donné
- $S_i^t = inside_i^t + outside_i^t$: nombre total d'invocations initiées par l'objet O_i depuis qu'il réside sur sa machine, à un instant t donné
- $membership_i^t = \begin{cases} \frac{inside_i^t}{S_i^t} \times 100 \in [0,100] \quad si\ S_i^t > 0 \\ 100 \quad si\ S_i^t = 0 \end{cases}$: cet indice, inspiré de l'indice g_{ijk} de gain de coopération introduit par [Adoud 00], exprime le degré d'appartenance de l'objet O_i à la machine m, à l'instant t, en fonction de ses relations locales et distantes, en sachant que les communications locales sont privilégiées par rapport aux communications distantes (lorsque $S_i^t > 0$, son degré d'appartenance est nul s'il n'opère que des communications distantes ; il est au contraire de 100% s'il n'effectue que des communications locales) et qu'un objet i n'ayant encore jamais communiqué depuis qu'il est sur la machine m a un degré d'appartenance à m de 100% ($S_i^t = 0$)
- $exclusion_m^t = \frac{\sum_i membership_i^t}{k_m^t} \in [0,100], \forall i \in A_m^t, m \in N^t$: exprime le degré d'exclusion de la machine m à un instant t donné. Plus le degré d'appartenance des objets à une machine est élevé, plus la machine sera exclue du SRO car indépendante
- $exclusion^t = \frac{\sum_m exclusion_m^t}{q^t} \in [0,100], \forall m \in N^t$: désigne l'indice d'exclusion de l'ensemble des machines utilisées par l'ARO à un instant t donné. L'objectif théorique est d'atteindre une exclusion de 100% pour toutes les machines utilisées à l'instant t.

4.5 Mesure de la QoS globale d'une application répartie objet :

4.5.1 Objectif :

Trois éléments entrent en compte lors du déploiement d'une ARO (rappel) :

- Les objets : ce sont les briques de base de l'application répartie. Ils sont répartis sur les machines du réseau
- Le réseau : il est utilisé par les objets pour communiquer (invocation de méthode)
- Les machines : de leurs performances et de leur charge dépendront en partie les performances de l'application en terme de temps de réponse (il convient de prendre également en compte le réseau).

Chacun de ces éléments doit se retrouver dans l'expression de QoS de l'ARO surveillée.

Les quatre critères de QoS retenus concernent, à l'instant *t* :

- La charge moyenne des machines ($load^t$) en fonction de la charge de chaque machine à cet instant
- Le temps moyen de réponse des invocations entre les objets qui composent l'ARO surveillée ($delay^t$) qui dépend du temps d'exécution des méthodes et des délais aller/retour de communication via le réseau
- La préférence des communications locales entre objets au détriment des communications distantes par la recherche de l'exclusion maximum de chaque machine du SRO ($exclusion^t$)
- La répartition des objets sur les machines ($distribution^t$) qui doit être la plus équitable possible afin d'éviter de migrer tous les objets non demandeurs de même classe sur une seule machine.

Ainsi, la mesure de QoS de l'ARO surveillée à l'instant *t*, notée $global^t$ et exprimée en pourcentage, est déterminée par l'expression suivante :

$global^t = \alpha \times (100 - load^t) + \beta \times delay^t + \gamma \times exclusion^t + \mu \times distribution^t \in [0,100]$, avec :

$load^t$: moyenne des pourcentages de charge des machines utilisées à l'instant *t*

$(\alpha, \beta, \gamma, \mu) \in [0,1]^4, \alpha + \beta + \gamma + \mu = 1$: poids associés aux quatre critères retenus

$delay^t$: pourcentage de réponses des invocations parvenues en-dessous d'un seuil de temps $delay_{\min}$ préalablement fixé

$exclusion^t$: degré d'exclusion des machines à l'instant *t*, tel que défini précédemment

$distribution^t$: pourcentage de répartition des objets dupliqués sur les machines du réseau.

L'objectif est d'améliorer cet indice dans le temps, à savoir $100 \geq global^t \geq global^{t'}, \forall t > t'$, ce qui implique l'amélioration d'au moins l'un des quatre critères pris en compte lors du calcul de $global^t$, à savoir :

$$\begin{cases} (distribution^t \geq distribution^{t'}) \vee (exclusion^t \geq exclusion^{t'}) \vee \\ (delay^t \geq delay^{t'}) \vee (load^t \leq load^{t'}), \forall t > t' \end{cases}.$$

4.5.2 Indices de charge liés aux machines :

4.5.2.1 Indices de charges statiques :

Les indices de charges statiques (ou à vide) des machines renseignent sur leur puissance lorsqu'elles ne sont pas utilisées :

- cpu_m donne la vitesse en MHz du processeur de la machine *m* (450 par exemple)
- mem_m indique le nombre de Mo de RAM disponibles sur la machine *m* (128 par exemple).

Ces indices permettent de comparer les charges des machines. En effet, une charge processeur de 10% sur une machine équipée d'un processeur à 450 MHz ne pourra être directement comparée à une charge identique sur une machine équipée d'un processeur à 133 MHz.

De même, un pourcentage d'occupation mémoire de 10% ne sera pas équivalent sur une machine disposant de 128 Mo et sur autre machine ne disposant que de 8 Mo.

Pour pouvoir comparer charge processeur et occupation mémoire entre deux machines, il est nécessaire d'introduire un référentiel commun calculé à l'aide de la formule suivante :

$loadref_m^t = \frac{cpu_m}{\max(cpu_{m'})} \times \frac{mem_m}{\max(mem_{m'})} \in]0,1], m \in N^t, \forall m' \in N^t$: coefficient de comparaison des charges des différentes machines utilisées à l'instant *t*.

4.5.2.2 Indices de charges dynamiques :

Le calcul du pourcentage de charge globale d'une machine *m* à l'instant *t* fait intervenir ses charges processeur (activité cpu) et mémoire :

$load_m^t = \alpha \times cpu_m^t + \beta \times mem_m^t \in [0,100], m \in M^t, (\alpha, \beta) \in [0,1]^2, \alpha + \beta = 1$, *t* entier, tel que :

α et β permettent de donner plus ou moins d'importance à la charge processeur ou mémoire

M^t désigne l'ensemble des machines prenant part à la répartition des objets (rappel)

cpu_m^t et mem_m^t renseignent sur les charges processeur et mémoire de la machine *m* à l'instant *t*, exprimées en pourcentage (ex : 60 pour 60%).

Comme mentionné précédemment, les charges globales de deux machines ayant des indices de charges statiques différents ne peuvent être comparées directement. Il est nécessaire de les mesurer à l'aide d'un référentiel commun pour déterminer la charge moyenne des machines du SRO.

Ainsi, le critère $load^t$ est défini par :

$$load^t = \frac{\sum_m \left(load_m^t \times loadref_m^t \right)}{q^t} \in [0,100], \forall m \in N^t .$$

Il semble de plus intéressant de prévoir la charge de chaque machine afin de mieux gérer les objets de l'application surveillée et d'anticiper sur l'état futur du SRO. Un nouvel indice, appelé gradient de charge instantanée et noté *gradient*, est utilisé à cette fin.

Le gradient de charge instantanée associé à la machine *m*, à l'instant *t* est défini comme suit :

$$gradient_m^t = \lim_{\substack{t' \to t \\ t' > t}} \frac{load_m^{t'} - load_m^t}{t' - t} .$$

Le gradient de charge correspond au vecteur unitaire tangent à la courbe de charge à un instant donné. La vitesse d'augmentation de charge est présumée proportionnelle à la valeur du gradient. Il suffit de connaître deux valeurs à des instants très proches pour calculer ce gradient.

De cette valeur instantanée du gradient de charge découle une valeur moyenne calculée à partir des *n* dernières valeurs du gradient instantané de la machine *m* :

$$agrad_{m,n}^t = \frac{\sum_t gradient_m^t}{n}, t \in E = \{t_1, ..., t_n = t\}, (t_i < t_j) \wedge (t_i \to t_j), \forall (t_i, t_j) \in E, i \neq j .$$

Ce gradient moyenne semble plus intéressant lors de la prévision de la charge future d'une machine car il tient compte de l'augmentation de charge sur un délai plus long que la valeur à un instant donné.

Par exemple, $agrad_{m,3}^t = \frac{gradient_m^{t_1} + gradient_m^{t_2} + gradient_m^t}{3}$ sachant que $t_1 < t_2 < t$, désigne la valeur moyenne du gradient de charge à l'instant *t* en considérant les trois dernières valeurs du gradient aux instants t_1, t_2 et *t*.

4.5.3 Indices de temps de réponse liés au réseau et aux objets répartis :

La mesure de l'indice $delay^t$, ou pourcentage de temps de réponses aux invocations en-dessous d'un seuil de temps $delay_{\max}$ préalablement fixé fait appel aux délais d'acheminement des messages via le réseau.

L'influence du réseau est sous-jacente (figure 13) car le temps de réponse d'une invocation fait intervenir à la fois le temps d'exécution de la méthode *m* appelée $(T_{exec})_m$ et les délais d'acheminement de l'appel $(T_{appel})_m$ et du retour du résultat $(T_{retour})_m$.

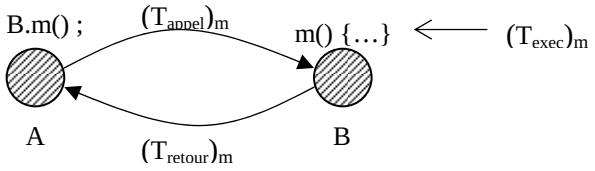

Figure 13 : mesure du délai lors d'un appel de méthode

L'indice $delay^t$ est déterminé dynamiquement en effectuant la moyenne des indices de temps de réponse $delay_{obj}^t$ de chaque objet *obj* à un instant *t* donné :

$$delay^t = \frac{\sum_i delay_i^t}{k^t} \in [0,100], \forall i \in O \text{, tel que :}$$

k^t désigne le nombre d'objets composant l'ARO surveillée à l'instant *t* (rappel)

$$delay_{obj}^t = \begin{cases} \dfrac{\sum_i inv_{obj,i}^t}{\sum_j inv_{obj,j}^t} \times 100 \in [0,100], \forall i \in C_{obj}^t, inv_{obj,i}^t < delay_{\max}, \forall j \in C_{obj}^t & si\ C_{obj}^t \neq \varnothing \\ 0 \quad si\ C_{obj}^t = \varnothing \end{cases}$$ est le

pourcentage d'invocations initiées par l'objet *obj* en-dessous du seuil $delay_{\max}$, avec :

$$inv_{obj,i}^t = \frac{\sum_m \sum_{j=1}^{N_m} \left(T_{exec} + T_{comm}\right)_m^j}{T_{obj,i}^t}, \forall m \in I_i \text{ , } \left(T_{comm}\right)_m^j = \left(T_{appel} + T_{retour}\right)_m^j$$ désigne le $j^{ième}$ appel de la méthode *m* de l'objet *i* et $T_{obj,i}^t$ le nombre total d'appels *m*.

4.5.4 Indices de répartition liés aux machines et aux objets :

Une répartition équitable des objets sur les machines du réseau est nécessaire afin d'éviter l'utilisation d'un nombre restreint de machines alors que beaucoup de machines sont disponibles. Cette répartition permet de plus de diminuer la charge globale des machines utilisées.

Le pourcentage de répartition globale des objets sur les machines utilisées à l'instant *t*, appelé $distribution^t$, est défini en fonction des degrés de répartition de l'ensemble des objets sur les machines (« nodes distribution » ou $ndist^t$) et des objets de même classe sur les machines (« objects distribution » ou $odist^t$) comme suit :

$distribution^t = \alpha \times ndist^t + \beta \times odist^t \in [0,100], (\alpha, \beta) \in [0,1], \alpha + \beta = 1$ tel que :

- $ndist^t = \frac{\sum_m ndist_m^t}{q^t} \in]0,100], \forall m \in N^t$ avec :

  $ndist_m^t = \begin{cases} \frac{\left|k_m^t - thnd^t\right|}{thnd^t} \times 100 \in]0,100] & si\ \left(k_m^t - thnd^t < 0\right) \vee \left(k_m^t - thnd^t > 1\right) \\ 100 \quad si\ 0 \le k_m^t - thnd^t \le 1 \end{cases}, m \in N^t$:

  $thnd^t = E\left(\frac{k^t}{p^t}\right)$ est le nombre par défaut d'objets que chaque machine disponible doit héberger, sachant que *E* est la fonction « partie entière » qui associe à chaque valeur réelle l'entier le plus proche par défaut $\left(x-1 < E(x) \le x, \forall x \in \Re\right) \wedge \left(E(y) = y, \forall y \in \Im\right)$. Ainsi, lorsque $Card(A_m^t) = thnd^t$, le pourcentage de répartition de la machine *m* est de 100% ; il diminue proportionnellement à la valeur $\left|thnd^t - k_m^t\right|$ lorsque $thnd^t > k_m^t$ ou $k_m^t > thnd^t$.

  L'indice de répartition d'une machine *m* à l'instant *t*, $ndist_m^t$, ne peut être nul car $thnd^t > \frac{1}{p^t}$ (en effet, $k^t > 1$ et $p^t \ge 1$) et $k_m^t > 0$ par définition (*m* est une machine utilisée par l'ARO à l'instant *t* : elle comporte donc au moins un objet à cet instant)

- $odist^t = \frac{\sum_C odist_C^t}{p^t} \in]0,100], \forall C \in Classes^t$ avec :

  $Classes^t$: ensemble des classes d'objets composant l'ARO à l'instant *t*

  $odist_C^t = \frac{Card\left(F_C^t\right)}{p^t} \times 100 \in]0,100], C \in Classes^t$ avec :

  $F_C^t = \bigcup_m m, \forall m \in M^t, C \in Classes^t, 0 \le N_{m,C}^t - thcd_C^t \le 1$: ensemble des machines pour lesquelles le nombre d'objets locaux de classe *C* à l'instant *t* convient, sachant que :

  $N_{m,C}^t - thcd_C^t$ est le nombre d'objets de classe *C* supplémentaires ou faisant défaut sur la machine *m* à l'instant *t*

  $N_{m,C}^t = Card\left(\bigcup_o o, \forall o \in A_m^t, Classe(o) = C\right), m \in M^t, C \in Classes^t$ est le nombre d'objets de classe *C* situés sur la machine *m* à l'instant *t*

  $thcd_C^t = E\left(\frac{N_C^t}{p^t}\right)$ est le nombre par défaut d'objets de classe *C* devant théoriquement se trouver sur chaque machine disponible à l'instant *t*,

$N_C^t = Card\left(\bigcup_o o, \forall o \in O^t, Classe(o) = C\right), C \in Classes$ désignant le nombre total d'objets de classe *C* enregistrés à l'instant *t*.

L'indice $odist_C^t$ vaut 100% uniquement lorsqu'autant d'objets de classe *C* sont enregistrés sur chaque machine disponible, avec une tolérance d'un objet supplémentaire par machine. L'équité est mesurée en fonction des machines ayant soit trop soit trop peu d'objets de même classe à l'instant *t*.

L'intérêt de scinder l'indice de répartition en deux permet de répartir les objets de manière précise. En effet, pour que cette répartition soit bénéfique il faut non seulement que chaque machine du réseau soit utilisée mais également que tous les objets instances d'une même classe ne se retrouvent pas sur une seule et même machine.

L'amélioration de l'indice $ndist^t$ limite la charge des machines en essayant de limiter le nombre d'objets sur chaque machine.

L'indice $odist^t$ permet de rendre :

- Le SRO plus tolérant aux pannes lorsque des objets de même classe sont répartis sur plusieurs machines : lorsqu'une machine tombe en panne, l'application pourra continuer à fonctionner si tous les objets qu'elle hébergait sont dupliqués sur au moins une autre machine
- Le mécanisme de répartition d'invocation du système de surveillance plus performant car il est plus aisé de privilégier les communications locales entre objets lorsque plusieurs instances d'un même objet sont exécutées sur des machines différentes.

4.5.5 Indices d'expression de la fiabilité d'un objet réparti :

DOMM considère un objet réparti comme une entité ayant un certain degré de fiabilité selon son activité. Trois indices de mesure de fiabilité sont utilisés afin de déterminer les objets les moins fiables, à déplacer en priorité :

- Indices de fiabilité interne à l'objet *obj* (intrinsèque à l'objet) :
  - $\Phi_{obj}^{inv} = \begin{cases} \frac{1}{N_{obj}^{inv}} \times 100 \in]0,100] & si\ N_{obj}^{inv} > 0 \\ 100 & si\ N_{obj}^{inv} = 0 \end{cases}$: fiabilité liée au nombre d'invocations des objets appelés par *obj*

    avec : N_{obj}^{inv} : nombre d'invocations demandées par l'objet *obj*
  - $\Phi_{obj}^{comm} = \begin{cases} \frac{1}{N_{obj}} \times 100 \in]0,100] & si\ N_{obj} > 0 \\ 100 & si\ N_{obj} = 0 \end{cases}$: fiabilité liée aux objets appelés par *obj*

    avec : $N_{obj} = Card(C_{obj})$: nombre d'objets différents appelés par *obj*
- Indice de fiabilité externe (dépendante de son environnement) :
  - $\Phi_{obj}^{call} = \begin{cases} \frac{N_{obj}^{exc}}{N_{obj}^{call}} \in [0,100] & si\ N_{obj}^{call} > 0 \\ 100 & si\ N_{obj}^{call} = 0 \end{cases}$: fiabilité liée aux appels reçus

    avec : N_{obj}^{exc} : nombre d'exceptions générées par l'exécution des appels

    N_{obj}^{call} : nombre d'appels reçus (envoyés par d'autres objets).

La fiabilité d'un objet est déterminée par la moyenne de ces trois indices pondérés (afin d'ajuster l'importance relative de la fiabilité interne vis à vis de la fiabilité externe de l'objet) :

$\Phi_{obj} = \alpha \times \Phi_{obj}^{inv} + \beta \times \Phi_{obj}^{comm} + \gamma \times \Phi_{obj}^{call} \in [0,100]$: pourcentage de fiabilité de l'objet *obj*, avec :

$\Phi_{obj}^{inv}, \Phi_{obj}^{comm}, \Phi_{obj}^{call}$: indices de fiabilités interne et externe de l'objet *obj*

$(\alpha, \beta, \gamma) \in [0,1]^3, \alpha + \beta + \gamma = 1$: poids associés aux différents indices de fiabilité.

4.6 Ordonnancement des objets et des machines :

L'indice de fiabilité Φ_{obj} permet d'ordonner totalement les objets selon leur fiabilité à un instant *t* donné. Cet ordre, noté $>_\Phi$ ou $=_\Phi$, exprime une notion de préférence stricte ou d'équivalence de fiabilité entre deux objets.

De même, l'indice $delay_{obj}^t$ introduit un ordre total concernant la charge des objets, noté $>_\ell$ ou $=_\ell$ selon que la préférence est stricte ou que les objets sont considérés équivalents.

Enfin, l'indice $load_m^t$ ordonne totalement les machines selon leur degré de charge.

Ces ordres donnent des indications au développeur souhaitant appliquer un algorithme de répartition des objets sur les machines.

L'ordre de charge permet de sélectionner les objets les plus consommateurs de ressources afin de les répartir sur des machines différentes (pour éviter les conflits d'exclusion entre objets : une exclusion se produit lorsque le placement de deux objets sur une même machine entraîne fatalement la surcharge de celle-ci) alors que l'ordre de fiabilité permet d'orienter les appels vers les objets les plus fiables en priorité, lorsque plusieurs objets d'une même classe sont présents.

4.7 Conclusion :

Ce modèle complète la spécification des applications réparties basées sur l'objet. Il décrit le fonctionnement de ces dernières et apporte des éléments de mesure de leur temps de réponse et de leur fiabilité. Le cadre d'utilisation de DOMM est générique, le système DOMS étant un exemple de mise en pratique des concepts introduits par le modèle.

5 Le système DOMS :

5.1 Présentation :

DOMS (Distributed Objects Management System) est un système de surveillance des ARO basé sur le modèle DOMM décrit précédemment. Ce dernier met l'accent sur les concepts de base de la gestion des objets d'une application répartie objet : il convient de les appliquer en offrant au développeur une interface (API) permettant l'accès aux indices et fonctions décrites par DOMM.

5.2 Hypothèses de mise en œuvre :

Certaines hypothèses doivent être posées pour que DOMS puisse jouer son rôle de surveillance :

- Le code source des objets composant l'application surveillée doit être accessible afin qu'une modification de code soit opérée par un composant de DOMS appelé « DOMSparser »
- Le réseau est fiable : tout message envoyé arrive fatalement à destination
- Les objets délégateurs ne disposent pas d'interface utilisateur (IHM) et peuvent donc être systématiquement déplacés : ainsi, seuls les objets demandeurs ne sont pas autorisés à migrer
- L'application surveillée peut être composée d'objets écrits dans des langages hétérogènes.

5.3 Principe de fonctionnement :

La surveillance et la gestion des objets d'une application répartie peut être décomposée en cinq phases (figure 14) définies par [Becker 92] formant un cycle de contrôle/commande des objets :

- Les machines et objets sont enregistrés et identifiés afin de pouvoir être gérés
- Le système de surveillance effectue alors des prélèvements sur les objets et machines répertoriés et collecte les informations qui en résultent
- L'état courant du SRO est alors évalué en fonction de certains seuils et indices prédéfinis
- A l'issue de cette évaluation, des décisions de répartition de charge (répartitions d'invocation ou d'objet) sont prises afin d'améliorer la QoS future de l'application
- Chaque décision prise entraîne alors une commande sur les objets : ce peut être une commande de migration d'une machine vers une autre (« load balancing » ou RO) ou d'orientation des appels vers d'autres objets (« load sharing » ou RI).

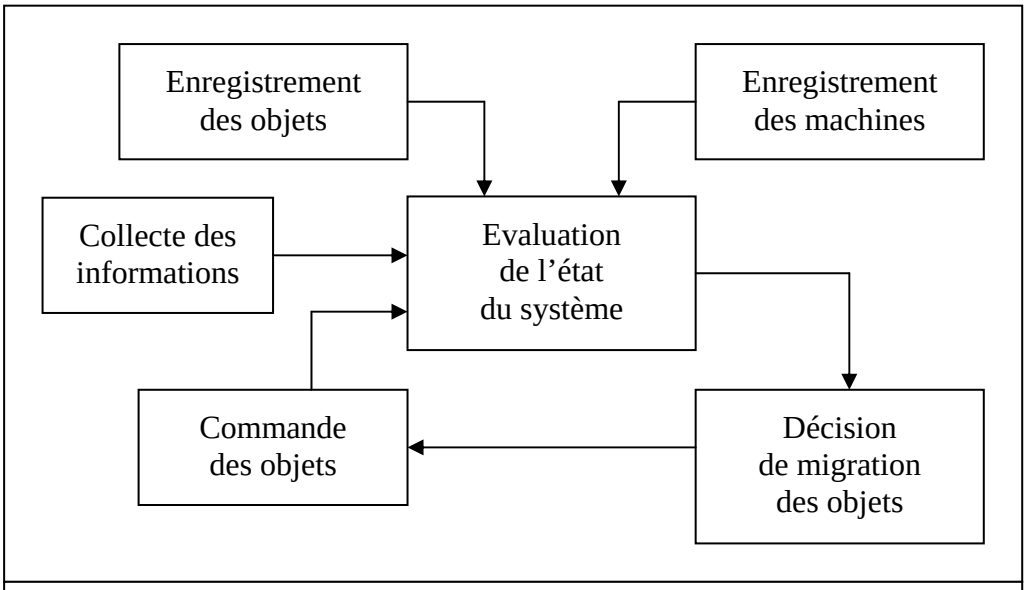

Figure 14 : cycle de contrôle et de commande des objets dans un SRO

Le système DOMS prend en charge les phases d'enregistrement des machines et des objets ainsi que la collecte des informations et propose une API permettant d'utiliser ces informations afin d'évaluer le SRO et de prendre des décisions de migration et d'orientation des appels des objets surveillés.

Les algorithmes de répartition d'objet et de répartition d'invocations automatiques devant être appliqués sont laissés à la discrétion du développeur. DOMS propose néanmoins deux algorithmes par défaut en guise d'exemples.

5.4 Architecture générale de DOMS :

DOMS fait le lien entre l'utilisateur, l'application répartie et les différentes machines participant à la gestion des objets (figure 15). L'utilisateur du système reçoit des informations et envoie des commandes manuelles à DOMS via une IHM.

DOMS dialogue avec trois acteurs de son environnement :

- Utilisateur : l'utilisateur consulte les informations collectées par DOMS, issues des deux autres acteurs, et envoie des commandes de correction ou de changement de stratégie à DOMS à l'aide d'une interface GUI (seules les commandes des objets sont actuellement prises en compte par cette version)
- Application répartie : DOMS collecte les informations (activité, fiabilité) liées aux objets de l'application répartie surveillée
- Machines et réseau : les informations concernant les machines et le réseau sont envoyées au système DOMS afin que ce dernier calcule les indices de DOMM.

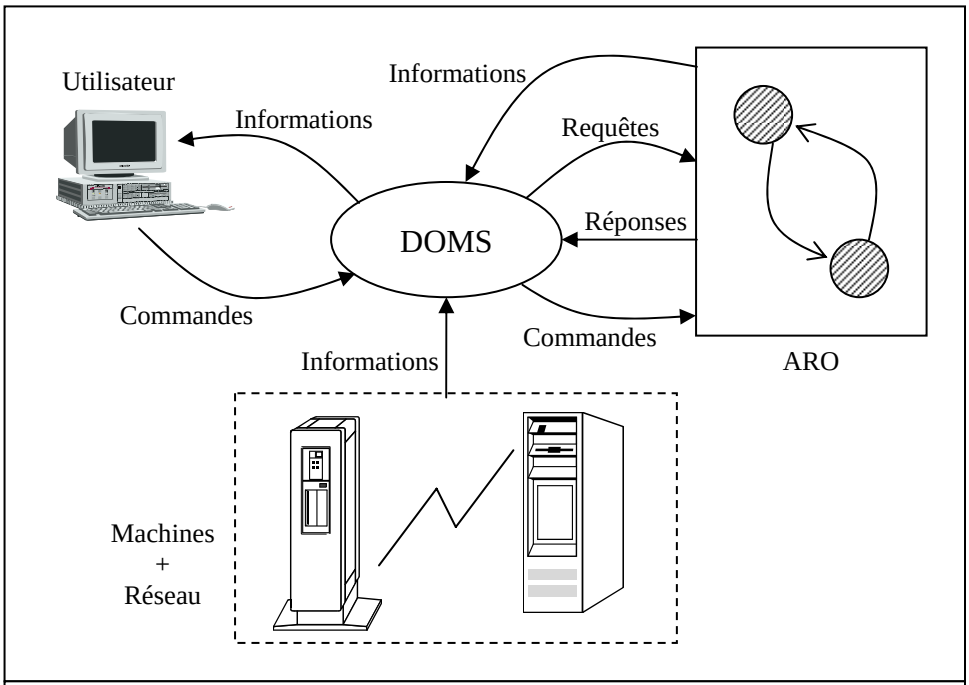

Figure 15 : le système DOMS dans son environnement, flots de données

Les messages envoyés et reçus par DOMS sont de quatre types :

- Informations : ce sont les données collectées qui concernent les objets de l'application surveillée, le réseau ainsi que les charges processeur et mémoires courantes des machines
- Requêtes : les requêtes, à l'initiative de DOMS, sont des demandes explicites d'informations ponctuelles (lorsqu'une information est manquante ou qu'elle est trop ancienne) concernant les objets de l'application, le réseau et les machines

- Commandes : envoyées aux objets par DOMS, les commandes sont des demandes de destruction ou de copie des objets vers d'autres machines (la gestion de la copie conservant l'intégrité des données des objets n'est pas actuellement prise en compte par le système) ou de modification des liens entre les objets (modification des références des objets appelés)
- Réponses : désignent les réponses des agents aux requêtes et commandes du manager.

Afin de limiter les communications entre DOMS et son environnement, aucun envoi régulier de message ne sera utilisé par DOMS. Un message ne sera émis que lors d'un changement significatif du système (des machines, du réseau ou des objets) décrit dans le protocole de communication de DOMSP.

5.5 Architecture détaillée de DOMS :

DOMS se compose de deux unités distinctes : DOMSmanager et DOMSagent. Son architecture est issue du modèle centralisé, plus facile à mettre en œuvre qu'un système totalement réparti (figure 16).

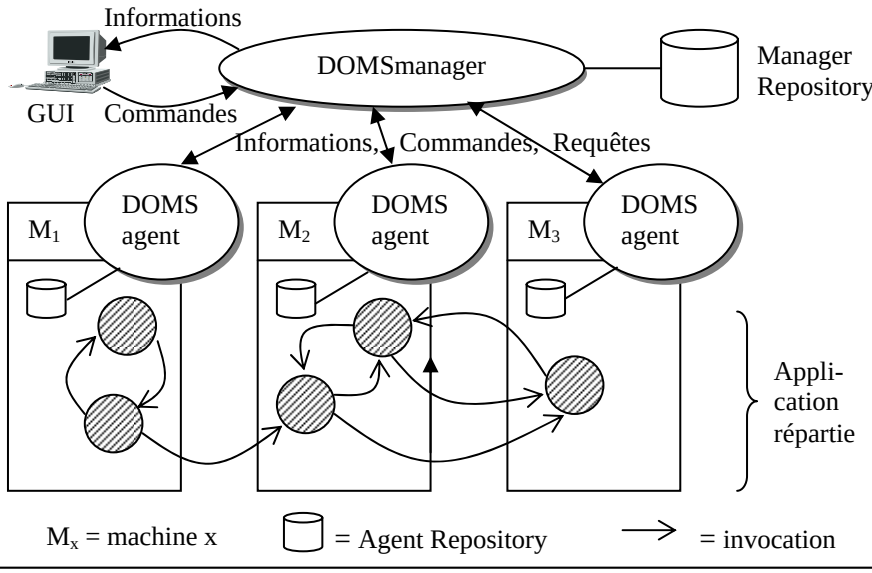

Figure 16 : architecture de DOMS

L'unité DOMSmanager a pour rôle de collecter les informations fournies par les DOMSagents, de les analyser, de demander les informations manquantes si besoin est, et d'envoyer en conséquence des commandes aux différents DOMSagents. Nous reviendrons sur la liste des commandes et messages par la suite.

L'unité complémentaire, DOMSagent, est exécutée sur chaque machine prenant part à la gestion de l'application répartie. Cet ensemble de machines comprend non seulement celles utilisées par l'application répartie mais également d'autres machines destinées à recevoir des objets déplacés par DOMS. Chaque agent, identifiant ainsi une machine unique, a pour mission de collecter certaines informations concernant sa machine et les objets sous sa responsabilité (objets locaux).

Les objets composants l'ARO sont administrables par DOMS après modification de leur code source selon un principe similaire à [Bouchi 00] : ils deviennent alors des DOMSobjects.

Chacun de ces trois composants hérite des ensembles décrits par le modèle DOMM. Cependant, ces ensembles ne sont pas transposés tels quels afin de tenir compte de certaines spécificités d'implémentation, en particulier le fait que chaque objet est une instance d'une classe.

5.5.1 Architecture de DOMSmanager :

5.5.1.1 Présentation :

DOMSmanager contrôle et commande directement les agents et indirectement les objets de l'application surveillée (par le biais des agents). Il est le cerveau du SRO surveillé. Il communique avec les différents DOMSagents afin :

- De collecter les informations liées aux machines et aux objets
- D'utiliser ces informations pour prendre des décisions de répartition de charge et envoyer les commandes résultantes aux agents qui les transmettront aux objets
- De modifier dynamiquement les liens entre les objets en modifiant la matrice d'incidence de l'ARO surveillée.

Il se compose de quatre modules exécutés en parallèle chargés de mettre à jour ou utiliser les informations de la base de données « Manager Repository » (figure 17). Ces informations lui sont communiquées par les DOMSagents.

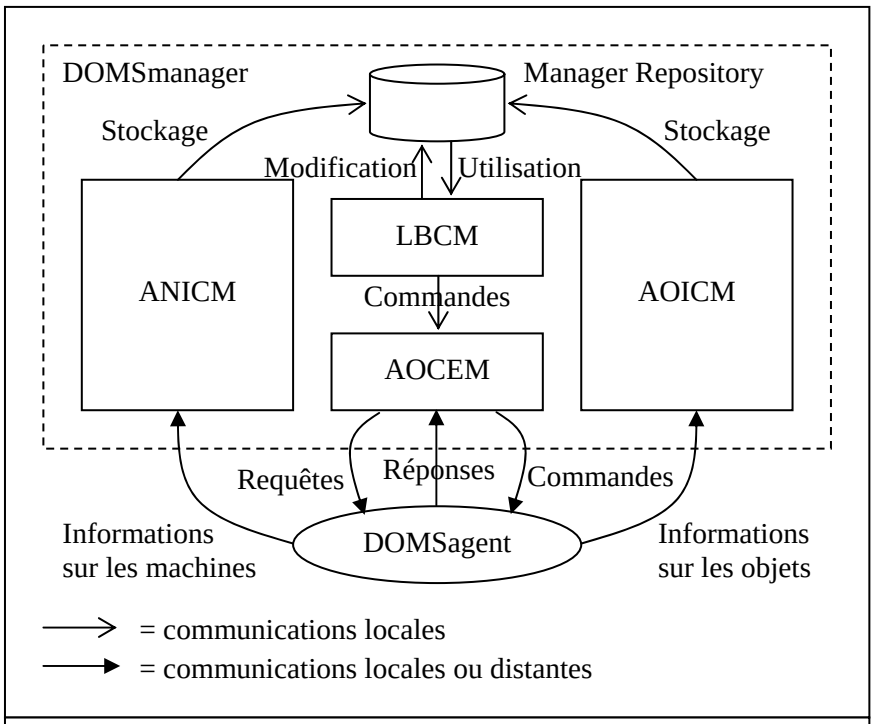

Figure 17 : composants de DOMSmanager

Les informations sont enregistrées dans une base de données qui contient la représentation de chaque ensemble décrivant l'ARO et son graphe de gestion. Cette base est décrite sous forme d'ensembles dont se servent les différents modules du manager.

5.5.1.2 Ensembles manipulés :

Le manager utilise les ensembles généraux qui caractérisent l'ARO surveillée ainsi que des ensembles dérivés du graphe de gestion.

Dans le tableau ci-après la mention N/A indique que ces ensembles n'ont pas été explicitement définis par DOMM mais s'avèrent utiles pour la spécification de DOMS.

DOMM	Transcription DOMSmanager	Signification
M^t	$RegisteredAgents = \{\{ID, R, C, M\},...\}$	Ensemble des machines disponibles. Pour chaque machine : identifiant *ID* et référence de l'agent *R*, puissances processeur (*C*) et mémoire (*M*) à vide
N^t	$UsedAgents = \{ID,...\}$	Ensemble des machines utilisées parmi les machines disponibles et identifiées par leur agent local *ID*
O^t	$RegisteredObjects = \{\{ID, C, R, N\},...\}$	Ensemble des DOMSobjects enregistrés par DOMS. Pour chaque objet : identifiant *ID*, classe *C*, référence *R* (éventuellement *null*) et nombre d'appels reçus *N*
L^t , $T_{i,j}^t$ et $T_{i,j}$	$TrustedObjects = \{\{ID_1, ID_2, N\},...\}$	Ensemble des liens de communications entre les objets. Chaque lien se compose : de l'identifiant de l'objet émetteur ID_1, de l'identifiant de l'objet récepteur ID_2 et du nombre d'appels effectués *N*
	$CalledObjects = \{\{ID, C, N, D, M, E\},...\}$	Ensemble des classes *C* des objets appelés par certains objets identifiés par *ID*, nombre d'appels *N* de chaque classe d'objets, durée moyenne des appels *D*, nombre d'appels *M* dont la durée est inférieure à $delay_{\max}$ et nombre d'exceptions *E* reçues par *ID*. Cet ensemble se distingue de *L* en ce sens qu'il n'indique pas les liens effectifs entre les objets mais plutôt les liens théoriques ; en effet, aucun objet particulier n'est mentionné)
E_{obj} , E_{obj}^t	$Exceptions = \{\{ID, E, N\},...\}$	Ensemble des exceptions *E* générées par un objet *ID*, *N* étant le nombre de fois que *E* a été générée
A^t	$LocalObjects = \{\{ID_{agent}, ID_{obj}\},...\}$	Cet ensemble permet de localiser les objets sur les machines en associant les objets aux agents tel que $ID_{agent} \in UsedAgents$
N/A	$Classes = \{\{C, M\},...\}$	Ensemble des classes *C* des objets enregistrés par le manager, *M* désignant la propriété de migration des objets de chaque classe
N/A	$LastMovedObjects = \{ID,...\}$	Ensemble des objets déplacés au cours de la dernière exécution de la répartition de charge

Certains des indices proposés par DOMM sont également manipulés sous forme d'ensembles par le manager :

Indice DOMM	Ensemble DOMSmanager	Signification
cpu_m^t, mem_m^t et $gradient_m^t$	$AgentsLoad = \{\{ID, C, M, G\},...\}$	Charges processeur (*C*) et mémoire (*M*) courantes et gradient de charge *G* des machines ayant des agents identifiés par *ID*
$inside_{obj}^t$	$Inside = \{\{ID, N\},...\}$	Nombre *N* de communications locales à l'initiative de l'objet identifié par *ID* depuis son dernier déplacement (ou sa création si l'objet n'a encore jamais été déplacé)
$outside_{obj}^t$	$Outside = \{\{ID, N\},...\}$	Nombre *N* de communications distantes de l'objet identifié par *ID* depuis son dernier déplacement (ou sa création si l'objet n'a encore jamais été déplacé)

Remarque : les indices cpu_m et mem_m sont intégrés à l'ensemble $RegisteredAgents$.

Ces ensembles permettent de calculer les valeurs de tous les indices mentionnés par DOMM.

5.5.1.3 Module AOICM (Agent Object Information Collection Module) :

Ce module prend en charge la collecte des informations concernant les objets surveillés. Celles-ci sont de cinq types :

- Enregistrement des objets lors de leur activation afin de leur assigner un identifiant
- Pour chaque objet *j* appelé par *obj* :
  - Nombre d'appels de *j* : $T_{obj,j}$
  - Temps moyen de communication : T_{comm}
- Listes des différents objets appelés : C_{obj} et C_{obj}^t
- Liste et nombre d'exceptions reçues pour chaque objet *obj* : E_{obj} et E_{obj}^t.

AOICM met à jour les ensembles $RegisteredObjects$, $CalledObjects$, $Exceptions$ et $Classes$ de la base « Manager Repository » lorsqu'une information lui parvient. Les informations contenues dans la base servent aux autres modules pour construire dynamiquement le graphe de gestion caractérisant l'application.

5.5.1.4 Module ANICM (Agent Node Information Collection Module) :

D'un fonctionnement similaire au module AOICM, ANICM collecte les informations concernant les machines, à savoir, pour chaque machine *m* :

- La vitesse processeur de *m* (ex : 133 MHz) : cpu_m
- La mémoire RAM dont dispose *m* (ex : 128 Mo) : mem_m
- La charge processeur courante : cpu_m^t
- La charge mémoire courante : mem_m^t

$\Big\}\ load_m^t$.

Ce module complète les données enregistrées par AOICM afin de fournir au module de répartition de charge des informations sur les machines en modifiant les ensembles *RegisteredAgents* et *AgentsLoad* . L'ensemble de ces informations constitue l'état du système.

5.5.1.5 Module LBCM (Load Balancing Command Module) :

Ce module est le cœur de DOMS puisqu'il est chargé de déterminer une configuration « au mieux » des objets sur les machines selon deux politiques définies par le développeur, l'une de répartition d'invocation, l'autre de répartition d'objet, à l'aide de l'API « DOMSmanagementAPI » fournie par DOMS. Les deux algorithmes fournis par le développeur, l'un de RI et l'autre de RO, sont ainsi utilisés afin de produire des commandes à destination du module AOCEM.

Ce module modifie les ensembles *LocalObjects* et *TrustedObjects* décrivant la répartition des objets sur les machines et la matrice d'incidence (liens fonctionnels) de l'ARO surveillée.

5.5.1.6 Module AOCEM (Agent Object Command Execution Module) :

AOCEM est l'exécutif de DOMS : les commandes de migration (ou de copie) envoyées par LBCM sont transmises aux DOMSagents concernés afin qu'ils les appliquent et rendent compte des résultats (délai de migration, succès ou échec de l'exécution de la commande, origine de l'échec – panne réseau, perte de communication avec un agent, etc.).

5.5.2 Architecture des DOMSagents :

5.5.2.1 Présentation :

Les agents servent de lien entre le manager et les objets de l'application surveillée (figure 18). Ils ont un rôle de collecte et de distribution des informations (concernant les objets, les machines et le réseau) ainsi que d'exécution des requêtes et commandes issues du manager.

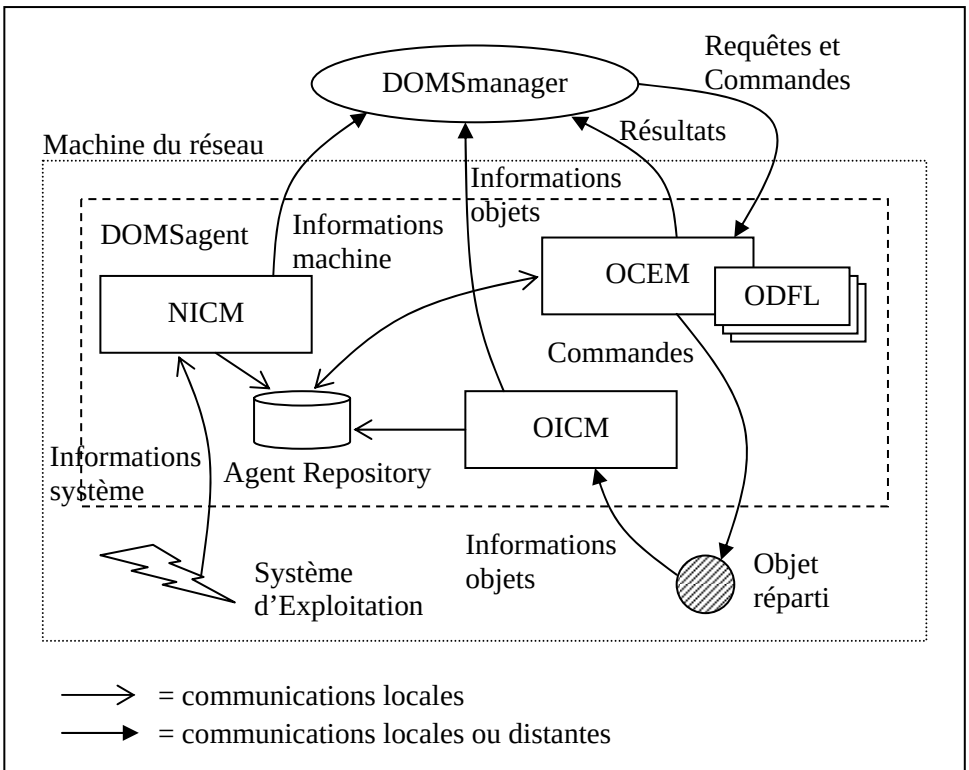

Figure 18 : composants de DOMSagent

5.5.2.2 Ensembles manipulés :

Chaque agent *A* utilise des ensembles dérivés des ensembles présentés par DOMM afin de répondre aux demandes des objets locaux et aux commandes du manager :

DOMM	Transcription DOMSagent	Signification
M^t	$Load_A = \{ID, C, M, P, Q\}$	Identifiant de l'agent, puissances processeur et mémoire (*C et M*) à vide et charges courantes (*P* et *Q*) de la machine sur laquelle est exécuté l'agent *A*
N/A	$LoadHistory_A = \{\{P, Q, D\},...\}$	Historique des charges processeur *P* et mémoire *Q* de la machine aux instants *D* : permet de calculer la valeur du gradient
A_A^t	$LocalObjects_A = \{\{ID, C, R, R', N\},...\}$	Ensemble des DOMSobjects enregistrés par DOMS. Pour chaque objet : identifiant *ID*, classe *C*, références *R* (éventuellement *null*) et *R'* (référence DOMSobject) et nombre d'appels reçus *N*
L^t , $T_{i,j}^t$ et $T_{i,j}$	$L_A = \{\{ID_1, ID_2, N\},...\}$	Ensemble des liens de communications à l'initiative des objets locaux. Chaque lien se compose : de l'identifiant de l'objet émetteur ID_1, de l'identifiant de l'objet récepteur ID_2 et du nombre d'appels effectués *N*
L^t	$TrustedObjects_A = \{\{ID, C, R, N, D\},...\}$	Ensemble des correspondances entre les objets classes *C* appelés par les objets locaux *ID* et les objets réellement appelés *R*. *N* est le nombre d'appels de l'objet *R* et *D* la durée moyenne des appels depuis la dernière migration de l'objet (ou depuis sa création si l'objet n'a encore jamais été déplacé). Cet ensemble permet d'attribuer à chaque objet local un objet pour chaque classe d'objets appelée

Ces ensembles apportent les connaissances locales de chaque agent qui seront transmises au manager.

Chaque agent est composé de trois modules concurrents prenant chacun en charge un rôle particulier et communicant avec, d'une part, les objets, et d'autre part, les modules du manager correspondants.

5.5.2.3 Module OICM (Object Information Collection Module) :

OICM est chargé :

- D'enregistrer les objets locaux surveillés en leur attribuant un identifiant unique
- De répondre aux demandes d'informations du manager concernant les objets (charge, état)
- De recueillir les informations envoyées par les objets locaux (objets distants appelés lors d'invocations de méthodes, délais de traitements des requêtes, etc.)
- De communiquer au manager les informations recueillies.

OICM est donc un module transitoire entre les objets et le module AOICM du manager.

5.5.2.4 Module NICM (Node Information Collection Module) :

Contrairement au module OICM, ce module ne communique pas avec les objets de l'application répartie surveillée. Il communique en revanche avec le module ANICM du manager, auquel il envoie des informations concernant la machine sur laquelle l'agent réside. Ces informations sont collectées à l'aide d'une sonde chargée de vérifier régulièrement la charge courante (processeur et mémoire) de la machine.

Les informations de charge machine ne sont cependant pas envoyées à intervalles de temps réguliers au DOMSmanager, mais lorsqu'un changement significatif d'état de la machine est décelé à l'aide des seuils $load_{\max}$ et $load_{\min}$ définis par DOMM.

5.5.2.5 Module OCEM (Object Command Execution Module) :

Associé au module AOCEM, ce module exécute les commandes du manager et informe AOCEM du bon déroulement de l'exécution de chaque commande reçue.

Ce module prend en particulier en charge le déplacement des objets (consulter les paragraphes suivants pour de plus amples détails). Il utilise un ensemble de modules appelés ODFL (Object Dynamic Factory Loaders), décrits en annexes, dont le rôle est de créer dynamiquement des objets en fonction de leur langage de programmation. Ces modules sont utilisés par l'agent lors du processus de migration des objets répartis : ainsi les objets surveillés peuvent être écrits dans des langages hétérogènes.

5.5.3 Les DOMSobjects :

5.5.3.1 Présentation :

Les DOMSobjects sont des objets répartis habilités à être instrumentés par DOMS. Pour ce faire, une phase de modification de leurs propriétés est nécessaire. Cette phase, réalisée avant la compilation du code, est l'œuvre du « DOMSparser » sommairement décrit en annexes. Ce dernier :

- Modifie l'implémentation des méthodes des objets non demandeurs afin d'y ajouter des calculs de temps de réponse des invocations (pour les objets demandeurs et délégateurs) et de temps de traitement des requêtes (pour les objets délégateurs et fournisseurs)
- Ajoute un objet délégateur autonome local, appelé *DOMSobjectSpec*, qui permet à chaque objet surveillé d'une part de conserver ses mesures de QoS ainsi que les données locales nécessaires à son instrumentation (ensembles dérivés du modèle DOMM) et d'autre part de répondre à son agent local via son interface DOMSobjectAPI (figure 19). Ainsi chaque objet surveillé, qu'il soit demandeur, fournisseur ou délégateur, contient un objet réparti ayant sa propre référence DOMS (référence *R'* stockée dans l'ensemble *LocalObjects* de l'agent local).

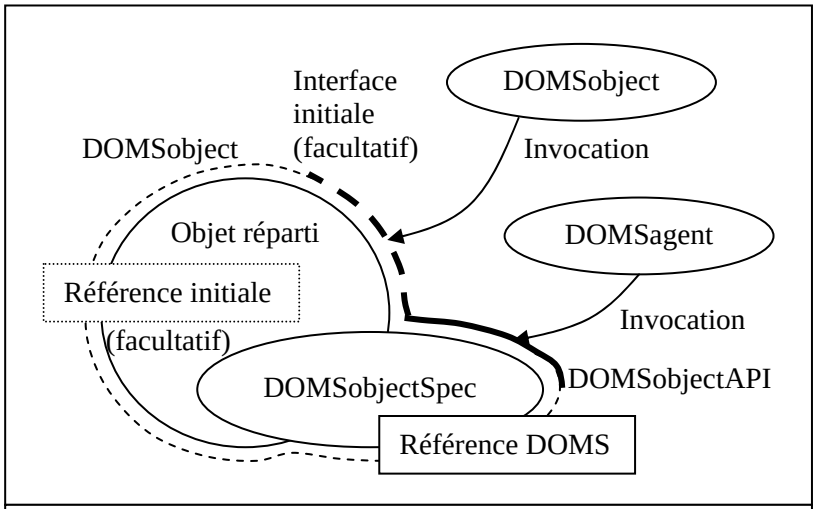

Figure 19 : composants d'un DOMSobject

5.5.3.2 Principe de fidélité :

DOMS met en œuvre le principe de *fidélité* des objets décrit ci-après :

- Un objet *B* de classe *C* *fidélise* un objet *A* lorsque *A* appelle *B* en priorité (dans le cas où plusieurs objets de classe *C* sont enregistrés par le système de surveillance)
- Inversement, un objet *A* *est fidèle à* un objet *B* de classe *C* lorsque *A* appelle *B* par défaut au moment où il cherche à contacter un objet de classe *C*
- L'objet *fidélisateur* (« trusted object »), désignant l'objet appelé, ne peut en aucun cas être de type demandeur car il reçoit des invocations
- Les objets *fidélisateurs* se fidélisent entre eux en priorité : si *B* fidélise *D* et *D* recherche un objet de classe *C* alors *D* appellera *B* en priorité. Ceci permet de simplifier la gestion de l'ARO surveillée car si *B* et *D* sont déplacés de telle sorte qu'ils soient sur la même machine, leurs communications locales en seront favorisées.

Ce principe est mis en œuvre car la migration de certains objets n'est bénéfique que si le système de surveillance contrôle les appels entre les objets, c'est à dire le graphe de l'application surveillée.

5.5.3.3 Ensemble manipulé :

La surveillance de chaque objet *obj* nécessite l'utilisation locale de l'ensemble L^t intégré à l'objet *DOMSobjectSpec* précité et dérivé du modèle DOMM :

DOMM	Transcription DOMSobject	Signification
L^t	$TrustedObjects_{obj} = \{\{C, R, N, T\},...\}$	Ensemble des objets de classe *C* et accessibles via leur référence *R*, *N* étant de nombre d'appels et *T* le délai moyen de réponse. Permet à chaque objet de connaître l'objet de classe *C* devant être appelé

5.5.3.4 Fonctionnement :

Les DOMSobjects s'enregistrent auprès de leur DOMSagent local lors de leur première activation en lui communiquant soit leur référence (pour les objets fournisseurs et délégataires) soit une référence vierge (pour les objets demandeurs). Ce dernier leur attribue alors un identifiant d'objet qui permettra de les connaître indépendamment de leur référence d'objet ou de leur classe d'appartenance. DOMS ne manipule ainsi les objets qu'au travers de cet identifiant. Un objet non identifié ne sera pas reconnu par DOMS.

Les objets enregistrés communiquent ensuite chaque information concernant leur état (invocation d'une méthode, envoi d'une exception, temps de réponse à une invocation, …) à leur DOMSagent local.

Une propriété de migration, appelée « MOVABLE », à valeur booléenne, caractérise chaque classe d'objets : par hypothèse seuls les objets demandeurs ne peuvent être déplacés. Cette propriété est donnée dès l'enregistrement des objets, les objets demandeurs se démarquant des autres objets par le fait qu'il ne disposent d'aucune référence permettant de les appeler.

5.6 Description du fonctionnement du module de répartition de charge :

5.6.1 Fonctionnement général :

Le module LBCM du manager a pour tâche initiale de charger dynamiquement les algorithmes de répartition d'objet dynamique (AROD) et de répartition d'invocation dynamique (ARID) qui implémentent les politiques respectives de RO et RI, implémentées par le développeur, devant être

utilisées par DOMSmanager. Les chemins d'accès à ces algorithmes sont passés en paramètre lors de l'exécution du manager :

$$\$> DOMSmanager\ AROD="chemin_{AROD}"\ ARID="chemin_{ARID}".$$

Lorsqu'un paramètre n'est pas indiqué, l'algorithme par défaut de DOMS correspondant, présenté en annexes, est appliqué. Lorsque l'un des algorithmes spécifiés n'a pu être trouvé, l'exécution du manager se termine.

Le second rôle imputé au module LBCM est de tenir le développeur au courant de l'évolution dynamique de son application à l'aide d'une interface de type GUI, connectée à DOMS, qui le renseigne sur la valeur de l'indice $global^t$ de QoS globale de son application.

5.6.2 Les algorithmes AROD et ARID :

5.6.2.1 Présentation :

L'AROD et l'ARID décrivent les actions que LBCM doit effectuer à un instant donné en fonction du graphe de gestion courant de l'application surveillée. Leur exécution permet de modifier dynamiquement ce graphe afin d'augmenter la valeur de l'indice $global^t$ en générant respectivement :

- Des commandes de migration des objets non demandeurs ; ces commandes entraînent des modifications temporaires des liens entre les objets
- De nouvelles connexions entre les objets en modifiant la matrice d'incidence *L* du graphe de l'ARO surveillée.

Ces deux algorithmes sont complémentaires, car l'AROD prend en compte l'environnement des objets, en particulier la charge des machines alors que l'ARID ne s'intéresse qu'aux liens fonctionnels entre les objets.

Leur fonctionnement général, sous forme de « boîte noire », est le suivant :

- Ils s'exécutent indépendamment l'un de l'autre de manière non continue : leur exécution est déterminée par LBCM en fonction des données enregistrées décrivant l'état courant du SRO
- Ils utilisent les variables décrivant le graphe de gestion de l'application à l'instant *t* et exécutent des traitements synchronisés sur ces variables
- Ils produisent des commandes de migrations d'objets et de changement des liens entre les objets (en modifiant les objets fidélisateurs appelés par défaut).

Leur conception est laissée à la discrétion du développeur. Le choix d'autoriser le développeur à intégrer ses propres algorithmes est motivé par trois constats :

- Il est très difficile de donner un algorithme général qui soit performant pour tout SRO
- Le développeur connaît l'environnement de déploiement de son application et peut ainsi en tenir compte dans ses algorithmes AROD et ARID
- Les performances des algorithmes dépendent fortement de la fréquence de duplication des objets et de la configuration initiale des objets sur les machines.

Le déterminisme de DOMS est fonction du déterminisme des algorithmes AROD et ARID utilisés.

5.6.2.2 Application des algorithmes AROD et ARID :

Ces algorithmes permettent au développeur de modifier dynamiquement les caractéristiques du graphe de gestion de son application. Aucune notion de temps n'est cependant prise en compte, à savoir :

- La durée d'exécution de chaque algorithme
- Les délais d'application des commandes générées
- La fréquence d'application de chaque algorithme.

Ces notions sont gérées par DOMS : les algorithmes se focalisent uniquement sur leur exécution et non sur le moment le plus opportun à leur exécution.

L'AROD et l'ARID sont exécutés indépendamment l'un de l'autre par LBCM. Il est donc nécessaire de prendre en considération leur exclusion mutuelle : en effet, leur utilisation aboutira à des incohérences telles que la migration d'un objet *o* par l'AROD alors que l'ARID indique à un objet *o'* qu'il peut joindre *o* en lui fournissant une adresse obsolète car en cours de modification par l'AROD.

De plus, l'exécution de chaque algorithme étant discrète, ils ne sont activés par DOMS que lorsqu'un changement significatif est détecté par le manager car leurs conditions d'activation sont différentes :

- L'AROD est normalement activé lors de chaque changement significatif de l'état de charge d'une machine. Cependant, afin de ne pas alourdir inutilement le SRO en appliquant trop souvent cet algorithme, une constante *AROD_DELAY* est définie. Elle exprime la durée minimum (en secondes) entre deux exécutions successives de l'algorithme. Il est possible que celui-ci ne soit pas appliqué pendant un laps de temps plus long que *AROD_DELAY* si aucun changement significatif de charge ne survient pendant ce délai. De plus, l'algorithme ne sera jamais activé si le SRO ne comporte qu'une seule machine.
- L'ARID est activé lors de chaque changement fonctionnel significatif. Il n'apparaît pas nécessaire d'introduire un délai équivalent à *AROD_DELAY* car le processus de modification des liens entre les objets est moins coûteux que la migration des objets (voir chapitre suivant). L'application systématique de cet algorithme permet de plus de réduire la charge des machines (en répartissant correctement les appels entre objets) et ainsi d'appliquer moins souvent l'AROD. Le protocole de migration des objets répartis fait appel à l'ARID afin de donner aux objets appelant les objets en cours de migration des références temporaires d'objets fidélisateurs de remplacement.

5.6.2.3 Changements de charge significatifs :

Le changement significatif de l'état de charge des machines est expliqué dans le chapitre traitant du protocole DOMSP. La charge de chaque machine est découpée en trois zones. Un changement significatif de charge n'est perçu que lorsque la charge passe d'une zone à une autre.

5.6.2.4 Changements fonctionnels significatifs :

Sont considérés comme changements fonctionnels significatifs :

- L'invocation d'une nouvelle classe d'objet par un objet donné
- La destruction d'un objet
- La création d'un objet de même classe qu'un objet non demandeur déjà enregistré
- L'appel d'un objet de classe *C* un nombre de fois multiple de la constante *NB_CALLS*.

5.6.2.5 Propriétés des AROD :

Tous les AROD développés possèdent certaines propriétés communes :

- La classe de l'AROD hérite de la classe « DOMSmanagementAPI » (présentée en annexes) afin d'accéder aux méthodes de gestion que propose DOMS et implémente l'interface « DOMSRunAROD » qui sera utilisée par le module LBCM via l'appel à la méthode virtuelle « AROD.run » implémentée par le développeur
- Ils recherchent l'augmentation de l'indice de QoS de l'ARO surveillée, noté $global^t$ à l'instant *t*, en améliorant les critères $distribution^t$, $exclusion^t$, $delay^t$ et $load^t$ qui le composent : $global^t = \alpha \times distribution^t + \beta \times exclusion^t + \gamma \times delay^t + \mu \times \left(100 - load^t\right)$
- Ils scrutent les objets qui composent l'ARO afin de déterminer les plus encombrants (à l'aide critères propres à chaque algorithme, basés sur les mesures des indices de DOMM)
- L'AROD est une fonction injective qui associe à chaque objet une machine hôte.

Ils doivent également obéir à trois règles essentielles :

- $Règle_1$: seuls les objets ayant leur propriété MOVABLE à valeur VRAI peuvent être déplacés ; les objets n'ayant encore pas reçu d'appel ne peuvent être migrés
- $Règle_2$: lorsque plusieurs objets d'une même classe sont enregistrés, ils ne peuvent pas être migrés sur une seule et même machine ; tous les objets d'une même classe ne peuvent se retrouver sur une même machine à un instant donné via une migration
- $Règle_3$: dans le cas où tous les objets d'une même classe se retrouvent sur une même machine, il faut migrer au moins l'un d'entre eux sur une autre machine afin de garantir une meilleure tolérance aux pannes (considérant que deux objets migrables de même classe ne s'appellent pas mutuellement).

5.6.2.6 Propriétés des ARID :

Les ARID permettent de modifier dynamiquement les liens d'appels entre les objets. Les modifications effectuées se traduisent formellement par les relations $\exists o \in O, \neg fournisseur_{obj}, \exists t > t', i \in C_o^{t'}, j \notin \in C_o^{t'}, i \notin C_o^t, j \in C_o^t, (i, j) \in Classe^2$ ou, en logique temporelle, $\exists o \in O, \neg fournisseur_o, \Diamond(((i \in C_o) \wedge (j \notin C_o)) \wedge \mathrm{O}((i \notin C_o) \wedge (j \in C_o)))$.

Tous les ARID développés possèdent certaines propriétés communes :

- La classe de l'ARID hérite de la classe « DOMSmanagementAPI » (présentée en annexes) afin d'accéder aux méthodes de gestion que propose DOMS et implémente l'interface « DOMSRunARID » qui sera utilisée par le module LBCM ; cette dernière comporte trois méthodes virtuelles « run », « runMultiple » et « selectReplacingTrustedObjects » devant être implémentées par le développeur (consulter l'exemple en annexes pour de plus amples détails)
- Recherchent l'augmentation de l'indice de QoS de l'ARO surveillée en modifiant dynamiquement les liens entre les objets tout en respectant le principe de fidélité des objets
- La méthode principale de l'ARID (« run ») est une fonction surjective qui associe à chaque objet une liste d'objets à appeler.

Afin de pouvoir changer dynamiquement d'objet appelé, il est nécessaire de ne pas avoir recours à l'ISO. Un objet n'appelle ainsi pas un objet précis mais demande l'appel à un objet de classe *C*. Le système DOMS prend alors en charge la sélection de l'objet le plus approprié en respectant le principe de *fidélité* des objets.

5.6.2.7 Exemples d'algorithmes :

DOMS propose en annexes, et ce dans un but pédagogique, deux algorithmes AROD et ARID simples afin de permettre au développeur d'appréhender la conception de tels algorithmes.

5.7 Conclusion :

Le prototype OMENS (Objects Manager, Environment and Network Supervisor), une implémentation de DOMS réalisée en JAVA et CORBA sous ORBacus 2.3 de OOC (Object Oriented Concepts), est actuellement en cours développement afin de démontrer la viabilité du projet et l'efficacité des protocoles retenus par DOMSP.

La volonté de concevoir une architecture modulaire permet d'enrichir OMENS au fur et à mesure de son développement.

DOMS a fait l'objet d'une publication à la conférence PDPTA'00 sur les systèmes parallèles et répartis qui s'est déroulée à Las Vegas du 26 au 29 Juin 2000 [Cottin 00].

6 Le protocole DOMSP :

6.1 Définition :

Le protocole DOMSP (DOMS Protocol) est un ensemble de règles qui définissent :

- La fréquence d'envoi des messages
- Le format et l'acheminement des messages entre DOMSmanager, les divers DOMSagents et les DOMSobjects
- Les processus d'invocation et de migration des DOMSobjects.

6.2 Problèmes généraux liés à l'exécution d'un système de surveillance :

Il est primordial de connaître les difficultés d'ordre général posées lors de la spécification d'un système de surveillance.

[Becker 92] fait état des principaux problèmes posés par l'exécution d'un système de surveillance :

- La surcharge due à la collecte, au transfert et au traitement des informations ainsi qu'à la réorganisation dynamique de l'application : il apparaît donc nécessaire de ne pas chercher une stratégie optimum mais plutôt définir des heuristiques qui permettront de se rapprocher de la solution optimum (solution satisfaisante)
- La brusque surcharge des machines faiblement chargées : ce problème apparaît lorsque plusieurs objets sont créés simultanément sur la machine par le système de surveillance
- La non mise à jour des informations : généralement, les informations dont dispose le système de surveillance sont obsolètes car l'application et son environnement ont évolué, le temps que les informations parviennent à l'organe décisionnel du système de surveillance. Bien qu'il soit impossible de connaître en temps réel l'état d'une application et de son environnement, il est toutefois possible d'actualiser les informations à intervalles réduits. En contrepartie, le système de surveillance surcharge les machines et le réseau. Il convient donc de définir une autre méthode, basée sur l'extrapolation des informations fournies
- Le comportement oscillatoire du système de surveillance : ce problème se pose lorsque le module décisionnel du système de surveillance est déterministe. Il est en effet possible qu'un ensemble de déplacements des objets de l'application surveillée aboutisse à la configuration actuelle du système. Le système de surveillance bouclera ainsi indéfiniment en effectuant toujours les mêmes déplacements
- L'instabilité de l'application surveillée due au système de surveillance : lorsque le système est très chargé, il se peut que d'une part certains objets soient en constant déplacement (et donc indisponibles) et d'autre part le fait de déplacer un objet induisant une charge supplémentaire des machines source et destination, le système devient de plus en plus chargé car plus la charge d'une machine augmente, plus le système de surveillance va chercher à la délester d'un nombre croissant d'objets.

Il est difficile de résoudre tous ces problèmes à la fois, chacun d'eux étant infiniment complexe. Une solution consiste à trouver un compromis permettant de collecter et traiter les informations recueillies tout en évitant de surcharger le réseau et les machines : le modèle DOMM est ici d'une grande utilité car il ne se contente que de peu d'informations et indices.

6.3 Hypothèses de mise en œuvre :

DOMS tolère l'emploi de plusieurs instances d'un même objet lors du déploiement de l'application (DOMS ne prend pas lui-même en charge la réplication des objets). Les répliques appartiennent à la même classe que l'objet initial.

6.4 Identification des DOMSagents :

Il est nécessaire d'identifier les agents afin que le manager :

- Sache de quel agent proviennent les messages qu'il reçoit
- Puisse envoyer des demandes d'information à un agent précis
- Puisse commander un objet précis de l'ARO surveillée.

Comme chaque DOMSagent s'exécute sur une machine du réseau, il est aisé de les identifier en leur attribuant le nom de la machine hôte. Cette identification permet de plus d'interdire l'exécution de plusieurs agents sur une même machine.

6.5 Identification des DOMSobjets :

Les objets répartis doivent être identifiés afin de pouvoir construire le graphe des invocations et savoir de quel objet sont issues les informations collectées.

Une première approche serait d'identifier les objets par leur référence. Cependant cette solution n'est pas adaptée ; en effet :

- Les objets demandeurs ne possèdent pas de référence
- La référence d'un objet non demandeur change lors de chaque déplacement.

La méthode adoptée par le protocole est de donner à chaque objet, lors de son référencement chez son DOMSagent local, un identifiant composé de l'identifiant de l'agent (unique) et d'un numéro. Ce numéro est incrémenté lors de chaque enregistrement d'un nouvel objet.

Ainsi, le premier objet créé sur la machine *HOST* portera l'identifiant *HOST1*, et ce qu'il soit de type demandeur, fournisseur ou délégateur. Il conservera cet identifiant tant qu'il sera actif.

6.6 Protocole de communication :

Le protocole de communication (DCP : DOMS Communication Protocol) d'une part entre les objets répartis composant l'ARO surveillée et les DOMSagents et d'autre part au sein de DOMS (entre les différents modules qui composent DOMS) a pour objectif de fournir des informations reflétant l'état courant du SRO tout en limitant le nombre de messages envoyés ainsi que la charge induite par l'exécution des modules de DOMS.

Les messages générés par DOMS sont à la fois synchrones (lors de l'appel d'un nouvel objet ou lors de la migration d'un objet) et asynchrones (lors de la transmission des messages d'information).

Les messages sont envoyés aux agents et au manager uniquement lors d'un changement significatif de l'état du SRO déterminé par plusieurs indices :

- De charge des machines (processeur et mémoire en cours d'utilisation) : changement significatif de charge
- D'activité des objets (nombre de communications entre les objets, appels de nouveaux objets, création et destruction d'objets) : changement significatif fonctionnel.

Les objets sont eux-mêmes sensibles uniquement à ces changements : ils ne transmettent leurs informations à leurs agents locaux uniquement lorsque cette étape est nécessaire.

Les agents ne sont ainsi informés que de manière sporadique et non de manière régulière ce qui réduit les communications d'une part entre les objets et leurs agents et d'autre part entre les agents et le manager.

De ce fait, les agents ont un rôle de médiation entre le manager et les objets surveillés.

6.6.1 Changement significatif de charge :

6.6.1.1 Seuils statiques d'expression des charges dynamiques :

Afin de rendre le système sensible uniquement à des changements significatifs de la charge des machines, DOMM définit pour chaque machine deux seuils de charge prédéfinis, $load_{max}$ et $load_{min}$, qui divisent la charge de la machine en trois zones distinctes (figure 20) :

- OVERLOADED : la machine est surchargée et ne peut en aucun cas recevoir un nouvel objet ; le système va chercher des machines hôtes qui permettront de décharger cette dernière de quelques objets
- LOADED : la machine commence à être chargée. Elle limite son accès au déplacement d'objets afin d'éviter de déplacer trop d'objets vers celle-ci, ce qui créerait une brusque surcharge
- AVAILABLE : la machine est disponible et peut héberger de nouveaux objets ; il faut néanmoins prendre garde à ne pas déplacer un nombre trop important d'objets vers celle-ci.

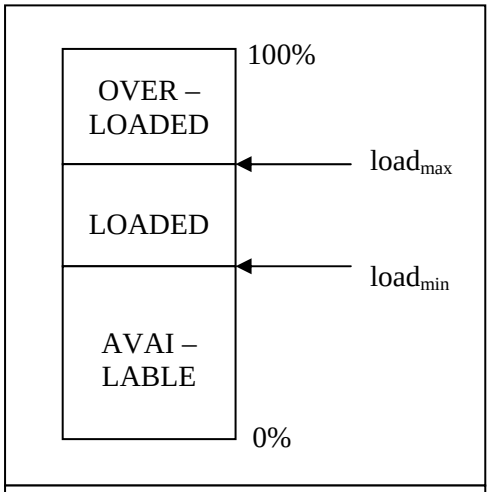

Figure 20 : trois états d'une machine

Les deux seuils de charge maximum et minimum, statiques et indépendants du processeur et de la mémoire des machines, s'expriment en pourcentage de charge ; par exemple :

- $load_{max} = 70$: seuil maximum de charge, limite entre les zones LOADED et OVERLOADED, fixé dans cet exemple à 70% de la charge maximum ($cpu_m = mem_m = 100\%$)
- $load_{min} = 45$: seuil minimum de charge, limite entre les zones AVAILABLE et LOADED, fixé arbitrairement à 45% de la charge maximum de 100%.

Seul le passage d'une zone à une autre est considéré comme changement significatif de la charge d'une machine : le système de répartition de charge ne prend connaissance que des changements de zone de charge, contrairement à des envois réguliers et systématiques de ces informations tels qu'effectués par d'autres systèmes.

Utiliser moins de zones supprime la notion de « zone tampon » (zone LOADED) : les machines passent brutalement de la zone « disponible » à la zone « surchargée ». En utiliser plus alourdit le système en générant beaucoup d'informations non pertinentes quant aux changements de charge des machines. L'utilisation de trois zones semble un bon compromis entre pertinence et nombre des informations générées.

6.6.1.2 Extension des seuils de charge des machines :

Il apparaît néanmoins que ces seuils peuvent s'avérer insuffisants, notamment lorsque la charge courante d'une machine oscille constamment d'une zone à une autre. Pour y remédier, deux constantes, Δ_{max} et Δ_{min}, sont introduites :

- $\Delta_{max} > 0$: définit un intervalle $I_{max} = \left[load_{max} - \frac{\Delta_{max}}{2}, load_{max} + \frac{\Delta_{max}}{2} \right]$
- $\Delta_{min} > 0$: définit un intervalle $I_{min} = \left[load_{min} - \frac{\Delta_{min}}{2}, load_{min} + \frac{\Delta_{min}}{2} \right]$

Les relations $100 - \frac{\Delta_{max}}{2} > load_{max}$, $load_{max} - \frac{\Delta_{max}}{2} > load_{min} + \frac{\Delta_{min}}{2}$ et $load_{min} > \frac{\Delta_{min}}{2} > 0$ doivent toujours être vérifiées afin de garantir que :

- Les trois zones de charge soient effectivement distinctes (sans chevauchement de zone lorsque l'on prend en compte les indices Δ_{max} et Δ_{min}) : $I_{max} \cap I_{min} = \varnothing$
- Les intervalles I_{max} et I_{min} sont à bornes strictement positives.

Les indices Δ_{max} et Δ_{min} sont utilisés comme suit (figure 21) :

- Lorsque la charge passe de la zone AVAILABLE à une zone plus chargée (charge montante), un changement significatif de l'état de la machine n'est perçu que si celle-ci dépasse $load_{min} + \frac{\Delta_{min}}{2}$ (passage dans la zone LOADED) ou $load_{max} + \frac{\Delta_{max}}{2}$ (passage dans la zone OVERLOADED sans passer par la zone LOADED)
- Lorsque la charge passe d'une zone autre que la zone AVAILABLE à une zone moins chargée (charge descendante), un changement significatif de l'état de la machine n'est perçu que si celle-ci est inférieure à $load_{max} - \frac{\Delta_{max}}{2}$ (passage de OVERLOADED dans la zone LOADED) ou à $load_{min} - \frac{\Delta_{min}}{2}$ (passage de OVERLOADED ou LOADED dans la zone AVAILABLE).

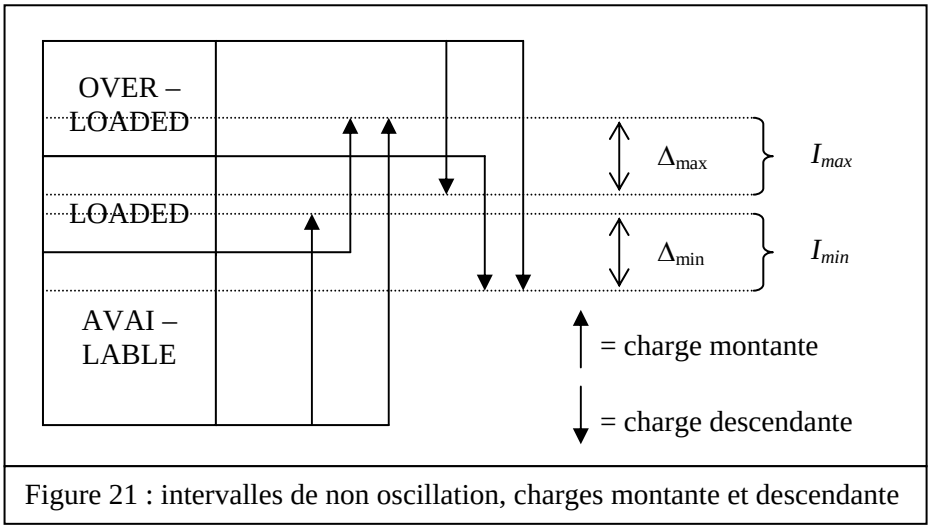

Figure 21 : intervalles de non oscillation, charges montante et descendante

6.6.2 Changement fonctionnel significatif :

L'exécution de l'ARID est conditionnée par chaque changement fonctionnel significatif. Chaque changement fonctionnel tel que le n^{ième} appel d'un objet par un autre ne peut être pris en compte par le système car il s'en suivrait une charge conséquente liée à de nombreux envois de messages entre d'une part les objets et leurs agents locaux et d'autre part les agents et le manager.

Ainsi, les informations fonctionnelles issues des objets ne sont pas transmises systématiquement au manager.

Elles le sont lorsqu'à un instant *t* un objet *obj* :

- Est activé : $\exists t > t', t \to t', \left(obj \notin O^{t'}\right) \wedge \left(obj \in O^{t}\right)$ ou $\Diamond\left(\left(obj \notin O\right) \wedge \mathrm{O}\left(obj \in O\right)\right)$
- Appelle un nouvel objet *o* : $\exists t > t', t \to t' \left(o \notin C_{obj}^{t'}\right) \wedge \left(o \in C_{obj}^{t}\right)$ ou, en logique temporelle, $\Diamond\left(\left(o \notin C_{obj}\right) \wedge \mathrm{O}\left(o \in C_{obj}\right)\right)$: la méthode « run » de l'API est alors exécutée
- A envoyé un multiple de *NB_CALLS* appels en direction d'un même objet *j* : $\exists t > t', t \to t', \exists \lambda \in \Im^{*}, \left(T_{obj,j} = \lambda \times NB\_CALLS - 1\right)^{t'} \wedge \left(T_{obj,j} = \lambda \times NB\_CALLS\right)^{t}$ ou $\Diamond\left(\exists \lambda \in \Im^{*}, \left(T_{obj,j} = \lambda \times NB\_CALLS - 1\right) \wedge \mathrm{O}\left(T_{obj,j} = \lambda \times NB\_CALLS\right)\right)$: la méthode « runMultiple » de l'API est exécutée
- Génère une nouvelle exception *e* : $\exists t > t', t \to t', \left(e \notin E_{obj}^{t'}\right) \wedge \left(e \in E_{obj}^{t}\right)$ ou, en logique temporelle, $\Diamond\left(\left(e \notin E_{obj}\right) \wedge \mathrm{O}\left(e \in E_{obj}\right)\right)$
- Génère un nombre d'exceptions multiple de *NB_EXC* exceptions : $\exists t > t', t \to t', \exists \lambda \in \Im^{*}, \left(N_{obj}^{exc} = \lambda \times NB\_EXC - 1\right)^{t'} \wedge \left(N_{obj}^{exc} = \lambda \times NB\_CALLS\right)^{t}$ ou $\Diamond\left(\exists \lambda \in \Im^{*}, \left(N_{obj}^{exc} = \lambda \times NB\_EXC - 1\right) \wedge \mathrm{O}\left(N_{obj}^{exc} = \lambda \times NB\_EXC\right)\right)$
- N'arrive pas à correspondre avec un objet (référence invalide) lorsque l'objet est en cours de déplacement ou lorsqu'il a été détruit indépendamment de DOMS
- Est détruit : $\exists t > t', t \to t' \left(obj \in O^{t'}\right) \wedge \left(obj \notin O^{t}\right)$ ou $\Diamond\left(\left(obj \in O\right) \wedge \mathrm{O}\left(obj \notin O\right)\right)$.

6.7 Protocole d'invocation dynamique des objets répartis :

6.7.1 Description sommaire :

DOMS se substitue aux mécanismes d'invocations statique et dynamique (ISO et IDO) originaux. Ce dernier utilise en effet son propre protocole d'invocation des objets répartis (DIP : DOMS Invocation Protocol) afin de tenir compte :

- De l'indisponibilité temporaire des objets en cours de migration
- Du fait que certains objets ne possèdent pas de référence fixe
- De l'obsolescence de certaines références d'objets.

Lorsqu'un objet *A* souhaite appeler un objet de classe *C*, le protocole retenu peut se résumer comme suit :

- *A* consulte sa table $CalledObjects_A$ afin de déterminer s'il est déjà entré en contact avec un objet de classe *C*
- Si oui, *A* rappelle cet objet : c'est la *fidélité des objets* (*A* est fidèle à l'objet appelé)
- Si non, *A* demande à son agent local de lui donner la référence d'un objet de classe *C* pour qu'il puisse l'appeler. *A* restera alors fidèle à cet objet autant que possible.

6.7.2 Surcharge induite :

La surcharge induite par ce protocole est définie dans les cas pire, meilleur et intermédiaire à l'aide de la description complète du protocole fournie en annexes. Pour effectuer les calculs nous considérerons que l'application surveillée est composée de *n* objets répartis dont *d* objets demandeurs et *f* objets fournisseurs $(d + f = n)$, tous de classes différentes (aucun objet n'est dupliqué). Chaque objet demandeur fait *m* fois appel au même objet fournisseur.

6.7.2.1 Cas pire :

Chaque objet demandeur n'a encore jamais appelé d'objet fournisseur et doit appeler tous les objets fournisseurs présents. A chaque appel, l'objet appelé est en cours de migration et l'agent local n'en a pas encore pris note.

Chaque appel va alors générer huit messages au préalable :

- Un message d'appel de l'objet fournisseur. Ce dernier ne répond pas à l'adresse demandée (génération d'une exception système)
- Un message pour demander à l'agent local la nouvelle référence de l'objet à appeler
- Deux messages avant que l'agent n'obtienne une référence vierge (car l'objet demandé est en cours de migration)
- Deux autres messages pour que l'agent puis l'objet prennent connaissance de la nouvelle référence de l'objet déplacé.

Ainsi, le nombre total de messages générés par DOMS afin que tous les objets demandeurs puissent communiquer avec tous les objets fournisseurs est $total = 8 \times d \times f \times m$. La surcharge induite par DOMS est en $O(n)$ dans le cas pire.

6.7.2.2 Cas meilleur :

Chaque objet demandeur fait appel à un objet fournisseur disponible. En cas de migration d'un objet fournisseur, chaque agent local a mis à jour ses données et transmis le changement à ses objets locaux concernés.

Avant de pouvoir invoquer un objet fournisseur, il suffit à chaque objet demandeur A de consulter sa table $CalledObjects_A$ afin de connaître la référence de l'objet à appeler.

La surcharge induite par ce protocole est alors nulle.

6.7.2.3 Cas intermédiaire :

Un objet fournisseur *B* est appelé par un objet demandeur *A* alors qu'il est en déplacement avec une probabilité $p_{A,B} < 1$. Soit p_B la probabilité que *A* appelle *B*. Lorsque ce cas se présente, la probabilité que l'agent local à *A* ne soit pas encore tenu au courant de la migration de *B* est notée c_B. Ainsi, le nombre total de messages générés par le système DOMS est $total = d \times \left(f \times m \times \overline{p}_B \times \left(8 \times \overline{p}_{A,B} + \left(1 - \overline{c}_B \right) \right) \right)$. La surcharge est alors en $O(n)$.

6.8 Protocole de migration des objets répartis :

Lorsque le système a pris des décisions de déplacement des objets répartis à l'aide de l'algorithme AROD, il faut lui fournir les moyens de les exécuter. Le protocole de migration propose un moyen dérivé des protocoles énoncés par [Rackl 97], [Schnekenburger 97] et [Pellegrini 00].

Ce protocole concerne uniquement les objets à valeur $MOVABLE = VRAI$, c'est à dire les objets non demandeurs.

6.8.1 Principes de migration des objets répartis :

Il existe trois principes de migration d'un objet d'une machine vers une autre :

- La sauvegarde de l'objet dans un fichier accessible depuis la machine destination qui sera ensuite chargée de recréer l'objet ; enfin destruction de l'objet initial
- L'envoi de l'objet sous forme de flux vers la machine destination : cette méthode est similaire à la sérialisation de l'objet
- La création d'un nouvel objet sur la machine destination auquel on transmet l'état de l'objet initial puis destruction de l'objet initial ; l'état d'un objet fait référence (1) aux valeurs attribuées à ses variables et (2) aux requêtes en attente de traitement (file d'attente de l'objet).

Le premier principe énoncé demande la création d'un répertoire partagé accessible depuis toutes les machines : c'est une solution invalide lors d'un déploiement à grande échelle de l'application.

Le second principe ne permet pas de garantir que l'objet pourra effectivement être recréé sur la machine destination : en effet, le flux transmis peut être erroné ou peut ne jamais arriver à destination.

Enfin, le troisième principe permet un meilleur contrôle de l'objet à déplacer ; le problème est de pouvoir transmettre l'état de l'objet initial (source) à l'objet nouvellement créé (destination).

Le déplacement d'un objet doit s'accompagner de la mise à jour de la matrice d'incidence *L* de l'ARO car la référence de l'objet n'est plus la même.

6.8.2 Approche du problème :

La migration d'un objet réparti doit faire face à cinq difficultés :

- La gestion des appels à destination de l'objet alors qu'il est en cours de déplacement
- La transmission de l'état de l'objet source à l'objet destination (l'état de l'objet et son contexte doivent être cohérents avant et après migration) :
  - Les valeurs courantes de ses variables
  - La liste des appels en attente
- La transmission de l'état des variables de surveillance locales à l'objet, utilisées par DOMS afin de conserver la trace des appels ainsi que les objets devant être appelés par défaut par l'objet après son déplacement
- La conservation du graphe fonctionnel après déplacement de l'objet : les objets doivent être tenus au courant du changement de référence de l'objet lors de son déplacement afin de pouvoir le contacter après migration
- La limitation du nombre de messages générés par DOMS lors du déplacement de l'objet et lors de la communication de la nouvelle référence aux autres objets.

6.8.3 Description sommaire :

Le protocole de migration des objets répartis retenu par DOMSP est dérivé de la dernière des trois possibilités proposées précédemment, à savoir la destruction de l'objet source suivi de la construction d'un nouvel objet sur la machine destination. Les objets source et destination doivent avoir le même identifiant. L'état de l'objet créé doit être identique à celui de l'objet détruit.

Le protocole adresse chaque problème comme suit (voir annexes pour une description détaillée) :

- Tous les agents sont prévenus de la migration de l'objet initial avant que celle-ci soit effectivement réalisée : ainsi les appels à cet objet sont différés jusqu'à ce que la nouvelle adresse de l'objet soit connue. Le cas exceptionnel où un objet fait appel à l'objet à déplacer avant que son agent local ait reçu la notification de migration de cet objet est également pris en compte par le protocole
- Les valeurs des variables externes (non liées au système de surveillance) et internes (utilisées par DOMS : objets fidélisateurs, etc.) sont transmises sous forme de flux à l'agent situé sur la machine destination ; elles seront passées en paramètres lors de la construction de l'objet destination par l'agent précité : les variables externes seront recopiées et les variables internes seront passées en paramètre au constructeur du composant *DOMSobjectSpec*

- Les appels reçus avant l'ordre de déplacement sont traités : aucun appel n'est en attente lorsque l'objet est déplacé ; les appels reçus après réception de la commande de migration renvoient une exception indiquant que l'objet appelé est temporairement indisponible
- Lorsque la nouvelle adresse de l'objet destination est connue de son agent local, ce dernier la transmet au manager qui la communique à son tour à tous les autres agents.

Les protocoles de migration et d'invocation sont fortement liés car lors de la migration d'un objet iA, celui-ci est d'abord détruit puis recréé : il n'existe plus pendant le laps de temps entre la destruction et la nouvelle création de *A*.

Le problème pouvant se poser lorsqu'un autre objet appelle *A* alors que *A* n'existe plus est résolu par le protocole d'invocation des objets (consulter le paragraphe traitant du protocole d'invocation des objets répartis) qui assigne temporairement un objet de remplacement à chaque objet en contact avec l'objet déplacé.

Le protocole retenu ne gère cependant pas les objets composés de plusieurs « threads ». Une prochaine version y intégrera les remarques formulées par [Fünfrocken 98] concernant la migration transparente des agents mobiles.

6.8.4 Surcharge induite :

La surcharge induite par le système de surveillance lors de la migration des objets répartis est définie pour les cas pire, meilleur et intermédiaire. Les hypothèses concernant l'ARO sont identiques à celles formulées précédemment : l'application surveillée est composée de *n* objets répartis dont *d* objets demandeurs et f objets fournisseurs $(f < n)$.

6.8.4.1 Cas pire :

Tous les objets demandeurs sont situés sur une machine différente et sont affectés par la migration d'un objet fournisseur (car tous appellent tous les objets fournisseurs).

$8 \times (d+1)$ messages sont générés par DOMS lors du déplacement d'un objet fournisseur *B* :

- *d* messages d'information envoyés par le manager à chaque agent afin de leur notifier que la décision de migrer l'objet *B* est prise
- Chaque agent indique à son objet demandeur local que *B* est temporairement indisponible
- Le manager envoie une commande de migration à l'agent qui héberge *B* : cet agent est noté *source*
- L'agent *source* détruit *B* (deux messages) et contacte l'agent destination (noté *dest*) pour que ce dernier crée une copie de *B*
- *dest* envoie au manager la nouvelle référence de *B*
- Le manager communique aux *d* agents la nouvelle référence de *B*
- Chaque agent la transmet aux objets demandeurs locaux : *d* messages

Ainsi, lorsque tous les agents migrent, le système génère $total = 8 \times (d+1) \times f$ messages (chaque communication est sécurisée et demande un acquittement). Le nombre de messages est en $O(n)$.

6.8.4.2 Cas meilleur :

Tous les objets demandeurs sont situés sur la même machine. Tous appellent tous les objets fournisseurs.

Le système génère alors $4 \times (d+3)$ messages lors du déplacement d'un objet fournisseur :

- Un message d'information envoyés par le manager à l'agent sur lequel sont exécutés les objets demandeurs, noté *agent*, afin de lui indiquer que la décision de migrer l'objet *B* est prise
- *agent* envoie à chaque objet demandeur une notification de migration de l'objet *B* : *d* messages

- Le manager envoie une commande de migration à l'agent qui héberge *B* : cet agent est noté *source*
- L'agent *source* détruit *B* (deux messages) et contacte l'agent destination (noté *dest*) pour que ce dernier crée une copie de *B*
- *dest* envoie au manager la nouvelle référence de *B*
- Le manager communique à *agent* la nouvelle référence de *B* qui la transmet aux objets demandeurs (*d* messages).

Dans ce cas, le système génère un nombre total de messages $total = 4 \times (d + 3)$, soit $O(n)$.

6.8.4.3 Cas intermédiaire :

Les objets sont situés sur *q* machines $q \leq d$. Un objet demandeur *A* appelle un objet fournisseur *B* avec une probabilité p_B . Chaque agent à la même probabilité p_A d'héberger un objet demandeur. Le nombre de messages générés par le système de surveillance est de l'ordre de $total = f \times (8 \times (q \times p_A \times p_B) + 1)$ soit $O(n)$.

6.8.4.4 Conclusion :

La combinaison des protocoles d'invocation dynamique et de migration des objets répartis est de l'ordre de $O(n)$.

6.9 Conclusion :

Le protocole DOMSP régit les messages envoyés et reçus par DOMS ainsi que les mécanismes d'invocation et de migration des objets surveillés. Le principal objectif étant de minimiser la surcharge induite par le système de surveillance, ce protocole utilise des messages synchrones apériodiques : DOMS n'est sensible qu'aux changements significatifs liés aux machines et aux objets.

La cohésion de l'application surveillée est garantie par la forte imbrication entre les mécanismes d'invocation et de migration des objets employés.

Alors que la plupart des environnements répartis objet emploient des méthodes circulaires de sélection des objets (« round robin ») ou des méthodes aléatoires, DOMSP prône le principe de *fidélité* des objets, proche du comportement relationnel humain.

7 Conclusion et perspectives :

Dans ce travail, nous avons proposé un environnement de surveillance des applications réparties objet hétérogènes. Ce dernier fournit au développeur une interface de programmation (API) permettant l'accès à divers indices de mesure de la QoS de ces applications. Cette architecture est basée sur le suivi et l'analyse des relations dynamiques inter–objets sans introduire de surcoût significatif lors de son utilisation.

L'architecture proposée répond aux problèmes de sélectionner les objets à déplacer ainsi que de déterminer le moment et le lieu de migration des objets surveillés afin d'augmenter les performances et la fiabilité de l'application.

Elle ouvre de nombreuses perspectives :

- L'intégration des modules de cryptage et de compression/décompression des messages qui transitent par DOMS
- L'intégration de la gestion des « threads » dans le processus de migration des objets selon un principe similaire à celui exposé par [Fünfrocken 98]
- L'approfondissement de la spécification du module « DOMSparser » d'ajout automatique du code nécessaire à l'utilisation de DOMS dans les objets composant l'application surveillée
- L'utilisation de l'application modifiée par le « parser » avec ou sans DOMS (qui pourra ainsi être exécuté après le déploiement de l'application)
- L'extension de l'utilisation de DOMS par des applications utilisant le service de médiation défini par [OMG RFP5]
- La surveillance des applications utilisant des envois de messages asynchrones
- L'introduction dans l'API de méthodes permettant de dupliquer un objet tout en conservant l'intégrité des données entre les différentes copies. Ce mécanisme suppose la définition d'un algorithme de synchronisation des appels. L'objectif est de rendre l'ARO tolérante aux pannes
- La vérification formelle de la cohérence de l'architecture en la traduisant dans des langages compréhensibles par des théorèmes prouveurs (non–blocage, vivacité, …).

Pour valider notre architecture, nous avons développé un prototype baptisé OMENS (Objects Manager, Environment and Network Supervisor) en JAVA et CORBA. Nous l'avons utilisé pour surveiller une application CORBA réalisant un produit matriciel en parallèle. Les résultats obtenus ont montré l'intérêt de regrouper sur une même machine les objets fortement corrélés.

8 Annexes :

8.1 Définition des types de messages utilisés par DOMS :

8.1.1 Messages d'information :

Les messages d'information générés par DOMS sont regroupés dans le tableau suivant :

MsgID	Signification
NEW_OBJECT	Le manager est tenu au courant de la création d'un nouvel objet (en dehors de la migration d'un objet existant) par les agents à l'aide de ce message
OBJECT_DELETED	Lorsqu'un objet est détruit (sauf dans le cas d'une migration), il envoie ce message à son agent local qui le transmet ensuite au manager
NEW_OBJECT_CALL	Ce message est envoyé par un objet *o* à son DOMSagent local lorsqu'il souhaite initier une première communication avec une classe d'objets. L'objet *o* devient alors fidèle à l'objet correspondant à la référence que lui donne l'agent. L'agent utilise également ce message pour propager au manager la demande de communication de son objet local *o*. Le manager répond à l'agent en lui donnant une référence d'objet que *o* pourra utiliser à l'aide d'un message de type NEW_TRUSTED_OBJECT
INVOCATION_ERROR	Ce message est généré d'abord par un objet lorsqu'il ne parvient pas à contacter un objet de classe *C* donné, puis par son agent local afin que le manager lui communique soit une référence valide de l'objet soit une référence vierge (lorsque tous les objets de classe *C* sont indisponibles). Dans ce dernier cas, l'objet devra attendre l'arrivée d'un message de type NEW_TRUSTED_OBJECT
INVOCATION_NOTIFICATION	Ce message est envoyé par un objet lorsqu'il a pris contact un nombre de fois multiple de *NB_CALLS* avec un objet de classe *C* donnée. Son agent local propage alors le message au manager
MIGRATION_NOTIFICATION	Ce message informe les agents de la migration prochaine d'un objet afin qu'eux-mêmes avertissent leurs objets locaux. Chaque agent informe à son tour les objets locaux en contact avec l'objet devant prochainement migrer
MIGRATED_OBJECT	La migration terminée, l'agent destination en informe le manager à l'aide de ce message. Ce faisant, le manager renvoie, via les agents, un message NEW_TRUSTED_OBJECT aux objets concernés par la migration
NODE_CURRENT_LOAD	Par ce message, chaque agent informe le manager des charges processeur et mémoire courantes de sa machine ainsi que des valeurs moyenne et instantanée de son gradient de charge lorsqu'il passe d'une zone de charge à une autre

AGENT_ACTIVE	Ce message est envoyé régulièrement par les agents au manager afin que ce dernier soit tenu au courant de l'état des machines. Chaque agent transmet en même temps le pourcentage de charge de sa machine. Si un agent ne répond pas, le manager lui envoie la requête AGENT_STATE_REQ
INFO_NOT_AVAILABLE	Ce message est généré par un agent lorsqu'il ne connaît pas la réponse à une requête issue du manager
CLASS_NOT_FOUND	Lorsqu'un objet demande une communication avec un objet de classe *C* alors que cette classe n'est pas répertoriée chez le manager, ce dernier génère ce message et l'envoie à l'agent concerné qui le transmettra ensuite à l'objet. L'invocation sera interrompue. Une future version permettra de créer dynamiquement l'objet demandé
OBJECT_NOT_FOUND	Ce message est envoyé par le manager lorsqu'un objet dispose d'une référence incorrecte d'un objet auquel il est fidèle et qu'aucune autre référence n'est disponible (l'objet demandé était le seul de sa classe et a été détruit par un objet tiers ou lors du crash d'une machine)
OBJECT_CREATION_ERROR	Généré par les agents, ce message indique au manager que la création d'un objet de classe *C* donnée n'a pu être réalisée par échec du chargement dynamique de la classe demandée

8.1.2 Requêtes :

Les requêtes (à l'exception des messages LOAD_VALUE_REQ, DELAY_VALUE_REQ, EXCLUSION_VALUE_REQ, DISTRIBUTION_VALUE_REQ et GLOBAL_VALUE_REQ demandés par l'interface utilisateur GUI) sont des demandes à l'initiative du manager.

MsgID	Signification
AGENT_POWER_REQ	Lorsqu'un agent s'est enregistré en donnant des valeurs incorrectes concernant la puissance (processeur et mémoire) à vide de la machine qui l'héberge, le manager envoie ce message afin que l'agent lui retourne des valeurs correctes
AGENT_LOAD_REQ	Par cette requête, le manager demande à un agent sa charge courante (calculée en fonction des charges processeur et mémoire) ainsi que son gradient moyen de charge
OBJECTS_REQ	Le manager demande à un agent de lui renvoyer l'ensemble des identifiants de ses objets locaux
OBJECTS_CLASSES_REQ	Le manager demande à un agent de lui renvoyer l'ensemble des classes de ses objets locaux
CALLED_CLASSES_REQ	Le manager demande à un agent de lui renvoyer l'ensemble des classes des objets appelés par ses objets locaux depuis leur création
TRUSTED_OBJECT_REQ	Le manager demande à un agent l'objet fidélisateur de classe *C* d'un objet donné
TRUSTED_OBJECTS_REQ	Le manager demande à un agent l'ensemble des objets fidélisateurs d'un objet donné

AGENT_STATE_REQ	Le manager demande par ce message à un agent de lui répondre afin de savoir s'il n'a pas été détruit. Le manager considère qu'un agent n'est plus actif lorsque le système sous-jacent lui renvoie un message d'erreur de communication
CALL_OBJECT_REQ	Le manager demande à un agent d'appeler un objet donné afin de savoir si ce dernier existe toujours
OBJECT_STATE_REQ	Le manager demande à un agent l'état d'un objet à un instant donné indépendamment du processus de migration des objets
CLASSES_AVAILABILITY_REQ	Le manager demande à un agent de lui fournir l'ensemble des classes d'objets que ce dernier est susceptible de créer (en fonction de ses modules ODFL locaux)
LOAD_VALUE_REQ	Ce message est demandé par l'interface utilisateur (GUI) afin de connaître la valeur courante de l'indice $load^t$ de DOMM
DELAY_VALUE_REQ	Ce message est demandé par l'interface utilisateur (GUI) afin de connaître la valeur courante de l'indice $delay^t$ de DOMM
EXCLUSION_VALUE_REQ	Ce message est demandé par l'interface utilisateur (GUI) afin de connaître la valeur courante de l'indice $exclusion^t$ de DOMM
DISTRIBUTION_VALUE_REQ	Ce message est demandé par l'interface utilisateur (GUI) afin de connaître la valeur courante de l'indice $distribution^t$ de DOMM
GLOBAL_QOS_VALUE_REQ	Ce message est demandé par l'interface utilisateur (GUI) afin de connaître la valeur courante de l'indice $global^t$ de DOMM

8.1.3 Commandes :

Les commandes (à l'exception de C_MIGRATED_OBJECT et C_COPIED_OBJECT à l'initiative des agents) sont générées par le manager. Elles concernent les objets de l'ARO surveillée. Chaque commande est d'abord envoyée à un agent qui la transmet ensuite aux objets concernés.

MsgID	Signification
NEW_TRUSTED_OBJECT	Cette commande est envoyé par le manager aux agents ayant au moins un objet local en communication (ou ayant déjà communiqué) avec un objet donné *o*. En envoyant ce message, le manager demande aux objets en relation avec *o* de ne plus appeler cet objet mais un autre objet de même classe que *o*. Ce message est également utilisé par les agents lorsqu'ils souhaitent informer leurs objets locaux de la modification de référence
CREATE_NEW_OBJECT	Demande de création d'un nouvel objet de classe *C* donnée. Cet objet sera alors enregistré par DOMS qui lui attribuera un identifiant. Le manager peut auparavant demander l'état d'un autre objet de même classe (via OBJECT_STATE_REQ) afin de le transmettre à l'agent chargé de créer l'objet. Cette fonctionnalité n'est pas actuellement supportée par DOMS

MIGRATE_OBJECT	Cette commande est envoyée par le manager à un agent lorsque tous les agents concernés par la migration d'un objet donné ont reçu le message MIGRATION_NOTIFICATION. Ce faisant, l'agent contacté initie le processus de migration de l'objet sélectionné par le manager
DELETE_OBJECT	Ce message est envoyé par un agent à un objet local au cours du processus de migration de l'objet. En retour, l'objet détruit informe l'agent de son état courant. Ce message ne génère pas de message DELETED_OBJECT en retour
C_MIGRATED_OBJECT	Lors du processus de migration d'un objet, l'agent hébergeant l'objet initial demande à l'agent destination de créer une copie de l'objet en lui transmettant son identifiant et son état résultant du message DELETE_OBJECT
COPY_OBJECT	A l'initiative du manager, cette commande est destinée à un agent afin qu'il prenne en charge la copie de l'un de ses objets locaux chez un autre agent. La copie aura un identifiant différent de celui de l'objet original. Cette commande n'est pas supportée par la version 1.0 de DOMS
C_COPIED_OBJECT	Lors de la réception d'un message COPY_OBJECT, l'agent source contacté envoie ce message à l'agent destination en lui transmettant les paramètres nécessaires à la copie. Cette commande n'est pas supportée par cette version de DOMS

8.1.4 Synthèse :

Le diagramme suivant (figure 22) regroupe l'ensemble des types de messages manipulés par DOMS (à l'exception des requêtes issues du manager et de certains messages d'information tels que INFO_NOT_AVAILABLE ou CLASS_NOT_FOUND qui concernent les cas d'erreur).

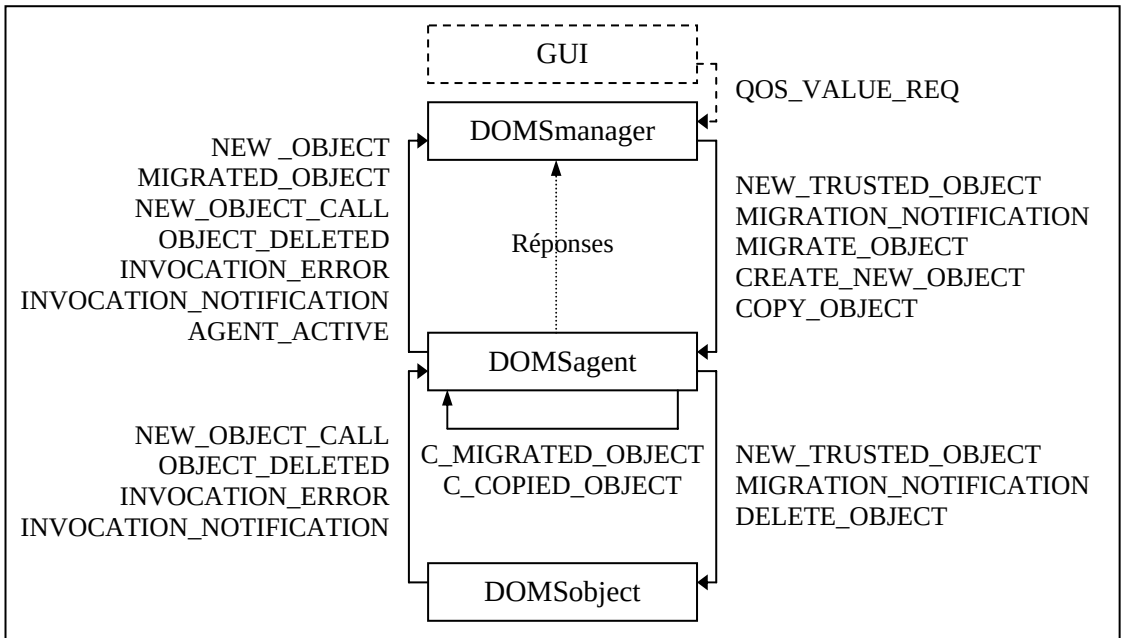

Figure 22 : principaux types de messages manipulés par DOMS

8.2 Format des messages internes :

On distingue deux ensembles de messages : les messages de communication entre les objets de l'ARO surveillée (externes au système) et les messages générés par DOMS (internes).

Les messages échangés entre les objets de l'ARO désignent les invocations de méthodes réalisées par les objets et formalisées par le graphe fonctionnel de l'ARO. Ces messages ne sont pas modifiés par DOMS et sont donc qualifiés d'« externes ». En particulier, le système de surveillance n'ajoute pas d'informations de surveillance (telles que le temps d'exécution de la méthode chez l'objet appelé) aux messages originaux : les signatures et types de résultats des méthodes des interfaces de communications originales des objets ne sont pas modifiées par DOMS.

Les informations de surveillance (messages internes) sont ainsi dissociées des messages de communication entre les objets de l'ARO.

8.2.1 Description :

Le format des messages internes est dérivé du format SNMP [Zeltserman 99]. Chaque message est ainsi composé de quatre parties (figure 23) :

- L'en-tête du message, avec :
  - SysID : identifiant du système pour lequel ce message est destiné (DOMS dans le cas présent) sous forme d'une chaîne de caractères
  - Version : chaîne de caractères indiquant le numéro de version du système, afin d'assurer la compatibilité du message avec les futures versions ou avec d'autres systèmes
  - MsgID : identifiant du type de message (type d'information fournie, type de commande à exécuter, type de requête)
- Les données, cryptées ou non, dont le format est propre à chaque type de message (par exemple, ce champ contient les identifiants de l'objet à déplacer et de l'agent destinataire lors d'une commande de migration).

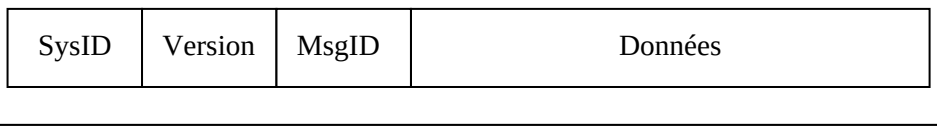

Figure 23 : format générique des messages manipulés par DOMS

8.2.2 Utilisation :

Dans le cadre de cette étude, les champs « SysID » et « Version » prennent respectivement les valeurs "DOMS" et "1.0".

Selon la valeur assignée au champ « MsgID », les données seront organisées différemment. Le tableau suivant en donne quelques exemples :

MsgID	Format du champ « Données »
NEW_OBJECT	• Identifiant de l'objet : *ID* • Classe de l'objet créé : *C*
NEW_OBJECT_CALL (envoyé par un objet)	• Classe de l'objet demandé : *C*

NEW_OBJECT_CALL (envoyé par un agent au manager)	• Identifiant de l'objet : *ID* • Classe de l'objet demandé : *C*
NEW_TRUSTED_OBJECT (envoyé par le manager)	• Identifiant de l'objet concerné : *ID* • Classe de l'objet demandé : *C* • Référence de l'objet fidélisateur : *R*
NEW_TRUSTED_OBJECT (envoyé par un agent à un objet)	• Classe de l'objet demandé : *C* • Référence de l'objet fidélisateur : *R*

MIGRATION_NOTIFICATION
(envoyé par le manager aux agents)
	• Ensemble des identifiants des objets fidélisés à l'objet *o* de classe *C* devant être déplacé : $\{\{ID, C, R\},...\}$ tel que :
	  • *ID* désigne l'objet fidèle à l'objet devant migrer
	  • *C* est la classe de l'objet devant migrer
	  • *R* est la référence temporaire d'un objet de même classe que *o*. Cette référence peut être nulle si *o* est le seul objet de classe *C*

MIGRATION_NOTIFICATION (envoyé par les agents aux objets)	• Ensemble $\{C, R\}$ indiquant à l'objet considéré que la référence temporaire de l'objet de classe *C* à contacter est *R*

8.3 Protocole de transmission de l'état d'un objet :

Lors du déplacement d'un objet, il est nécessaire de transmettre à l'objet destination l'état de l'objet source, c'est à dire l'ensemble des valeurs des variables qu'il manipule. Définir un protocole de transmission de l'état d'un objet s'avère primordial car l'objet destination doit être capable de comprendre le message reçu qui contient l'état de l'objet. Un tel protocole demande l'étude des types de données qui pourront être transmis d'un objet à l'autre. Les types de données sont classés en deux catégories : les types simples et les types complexes.

En supposant que l'état soit transmis sous forme de flux, ce dernier obéit à la grammaire suivante :

EtatFlux ::= NombreVariables + Descriptions

NombreVariables ::= Entier

Descriptions ::=
Descriptions ::= ':' + Desc

Desc ::= Nom + '{' + Structure + '}' + NombreEléments +
'[' ValeursElements + ']' + DescRec

Nom ::= Chaîne

Structure ::= Type + StructureRec
Structure ::= '{' + Structure + '}'

StructureRec ::=
StructureRec ::= ',' + Structure

Type ::= Entier | Réel | Booléen | Long | Chaîne | Caractère | Référence

NombreEléments ::= Entier

ValeursElements ::= ValeurElem + ValeurElemSuivant

ValeurElem ::= Valeur_attribuée_à_un_type + ValeurRec

ValeurRec ::=
ValeurRec ::= ';' + ValeursElem

ValeurElemSuivant ::=
ValeurElemSuivant ::= '|' + ValeurElem

DescRec ::=
DescRec ::= ':' + Desc

Ainsi, les flux EtatFlux = 0 et EtatFlux = $3;\text{var}_1\ \{Entier\}1[10]:\text{var}_2\ \{Entier, Réel\}2[5;2,5\,|\,8;1,2]$ traduisent respectivement :

- Un état vide : l'objet ne dispose pas de variables de classe
- La transmission de trois variables, var_1, var_2 et var_3, ayant les caractéristiques suivantes :
  - var_1 est un entier à valeur 10 : $var_1 = 10$
  - var_2 est un tableau de structures comportant un entier et un réel : $var_2[1].champ_1 = 5$, $var_2[1].champ_2 = 2,5$, $var_2[2].champ_1 = 8$ et $var_2[1].champ_2 = 1,2$.

Ce protocole permet ainsi de transmettre à la fois des types de données simples (entier, chaîne de caractères, booléen, etc.) et des structures complexes comportant plusieurs niveaux d'imbrication, comme dans l'exemple suivant :

EtatFlux = $1:\text{var}\{Entier, \{Caractère, Booléen\}\}2[1;'a';VRAI\,|\,2;'b';FAUX]$

Le flux désigne alors une variable appelée « var » qui se compose d'une structure comprenant un entier et une structure (imbriquée) composée d'un caractère et d'un booléen.

En l'état actuel, le système DOMS ne gère pas la transmission de l'état d'objets composés de plusieurs « threads ». En effet le processus de reconstruction des « threads » est complexe et nécessite une étude qui sera envisagée dans une future version. Ce problème est également posé lors de la sérialisation des objets en Java.

8.4 Protocole d'invocation des objets répartis :

Lorsqu'un objet *A* ayant un agent local *Agent* souhaite appeler un objet de classe *C*, il suit l'algorithme suivant :

Principal(*A*,*C*) :
  Si $(\{C,R\} \notin CalledObjects_A)$ alors
    // *A* appelle pour la première fois un objet de classe *C*
    $R \leftarrow$ *Agent*.newObjectCall(*A*,*C*)
    Si $(R = ERROR)$ alors
      // Aucun objet de classe *C* n'est répertorié
      Sortie
    Fsi
    $TrustedObjects_A = TrustedObjects_A \cup \{\{C,R\}\}$
  Sinon
    // *A* a déjà été en contact avec un objet de classe *C*
    // Il faut appliquer le principe de *fidélité* en

```
            // reconnectant A avec ce même objet si possible
            R ← getObjectReference(C)
        Fsi
        // Invocation de l'objet ayant la référence R
        res ← invoke(R)
        Tant que (res = ERROR) faire
            // Echec de l'invocation
            // Deux causes : l'objet a été détruit ou il est en cours de déplacement
            R' ← Agent.objectInvocationError(A,C,R)
            Si (R' = ERROR) alors
                // Il n'existe plus aucun objet de classe C qui soit enregistré
                Sortie
            Sinon
                // L'objet est en cours de migration, une référence de remplacement
                // est donnée
                TrustedObjects_A = {TrustedObjects_A − {{C, R}}} ∪ {{C, R'}}
                res ← invoke(R')
            Fsi
        Fait
    FinPrincipal

    Agent.newObjectCall(A,C) :
        R ← Manager.newObjectCall(A,C)
        Si (R = null) alors
            // Tous les objets de classe C sont en cours de migration
            // Il faut attendre que le manager envoie une référence destinée
            // à l'objet A
            R ← Manager.waitForReference(A,C)
        Fsi
        TrustedObjects_Agent = TrustedObjects_Agent ∪ {{A, C, R}}
        Retourner R
    FinAgent.newObjectCall

    Object.getObjectReference(C) :
        Faire
            Soit R, {C, R} ∈ TrustedObjects_A
            Si (R = null) alors
                // L'ARID n'a donc pas pu donner
                // d'objet de remplacement : il faut attendre
                // NEW_REFERENCE_DELAY millisecondes
                wait(NEW_REFERENCE_DELAY)
```

Fsi

Tant que $(R = null)$

Retourner *R*

FinGetObjectReference

Object.invoke(*R*) :

ID ← getObjectID(*R*)

InvokeObject(*R*)

Agent.objectInvokedNotification(*A*,*ID*)

FinObject.invoke

Agent.objectInvocationError (*A*,*C*,*R*) :

Si $(\{A,C,R\} \notin CalledObjects_{Agent})$ alors

// L'objet n'appelle visiblement pas une référence correcte

Retourner $R', \{A,C,R'\} \in CalledObjects_{Agent}$

Fsi

// L'objet a contacter est soit en cours de migration,

// soit a été détruit en dehors du processus de migration

// Soit *L* l'ensemble des références d'autres objets de classe *C*

// connues par l'agent local

$L = \bigcup R', \{X,C,R'\} \in CalledObjects_{Agent}, R' \neq null, R' \neq R$

Si $(L \neq \varnothing)$ alors

// L'agent connaît d'autres références d'objets

// que A pourra utiliser temporairement : *R'* par exemple

R' ← Choix(*L*)

$TrustedObjects_{Agent} = \{TrustedObjects_{Agent} - \{A,C,null\}\} \cup \{\{A,C,R'\}\}$

Retourner *R'*

Fsi

// $L = \varnothing$: l'agent ne connaît aucun autre objet de remplacement

// Il faut attendre que le manager connaisse la nouvelle

// référence de l'objet migré ou donne lui-même la référence

// d'un objet de remplacement

R' ← *Manager*.waitForReference(*A*,*C*)

$TrustedObjects_{Agent} = \{TrustedObjects_{Agent} - \{A,C,null\}\} \cup \{\{A,C,R'\}\}$

Retourner *R'*

FinAgent.objectInvocationError

Manager.newObjectCall(*A*,*C*) :

// *A* appelle un objet de classe *C* pour la première fois

// Cette communication est considérée par DOMS comme un changement

// fonctionnel significatif (ajout d'un lien entre deux objets dans la matrice

// d'incidence *L* de l'ARO)
// Tous les objets de classe *C* sont donc migrables
Si $(getClassMigrationAbility(C) = FAUX)$ alors
setClassMigrationAbility(C,VRAI)
Fsi
// Soit L_c l'ensemble des références des objets de classe *C* enregistrés
$L_C = \bigcup R, \{ID, C, R, N\} \in RegisteredObjects$
Si $(L_C \neq \varnothing)$ alors
// Sélectionner le « meilleur » objet *R'* selon l'algorithme ARID
// fourni par le développeur (ou l'algorithme par défaut) en
// tenant compte des objets en cours de migration
// L'exécution de l'algorithme modifie la matrice d'incidence *L*
// caractéristique de l'ARO surveillée
$R' \leftarrow$ *ARID*.run(*A*,L_C)
// *R' = null* si tous les objets de classe *C* sont en cours de migration
Retourner *R'*
Fsi
// Aucun objet enregistré n'est de classe *C*
// Le manager attend *REGISTRATION_DELAY* secondes
// avant de rechercher à nouveau les objets de classe *C* enregistrés
wait(*REGISTRATION_DELAY*)
$L'_C = \bigcup R', \{ID', C, R', 0\} \in RegisteredObjects$
Si $(L'_C \neq \varnothing)$ alors
// Sélectionner le « meilleur » objet selon l'algorithme ARID
// fourni par le développeur (ou l'algorithme par défaut) en
// tenant compte des objets en cours de migration
$R'' \leftarrow$ *ARID*.run(*A*, L'_C)
// R'' = *null* si tous les objets de classe *C* sont en cours de migration
Retourner R''
Sinon
Retourner *ERROR*
Fsi
FinManager.newObjectCall

Manager.waitForReference(*A*,*C*) :
Faire
$L_C = \bigcup R, \{C, R\} \in RegisteredObjects$
Tant que $(L_C = \varnothing)$
// Sélectionner le « meilleur » objet selon l'algorithme ARID
// fourni par le développeur (ou l'algorithme par défaut) en

// tenant compte des objets en cours de migration

$R' \leftarrow$ *ARID*.run(*A*,L_C)

Retourner *R'*

FinManager.waitForReference

Agent.objectInvokedNotification(*A*,*ID*)

// Incrémenter le nombre d'invocations de l'objet *ID* par *A*

A.objectCalled(*ID*).nbCalls ← *A*.objectCalled(*ID*).nbCalls + 1

FinAgent.objectInvokedNotification

8.5 Protocole de migration des objets répartis :

Le processus de migration (DMP : DOMS Migration Protocol) d'un objet *A* de classe *C* ayant *R* pour référence, de l'agent $Agent_1$ vers l'agent $Agent_2$, se décompose comme suit :

Principal(*A*,*C*) :

// Soit *E* l'ensemble des objets qui ont déjà appelé *A* au moins une fois et

// *F* l'ensemble des machines (ou agents) qui hébergent ces objets

$$E = \bigcup o, L_{o,A} \in L, \forall \{o, C', R, N\} \in RegisteredObjects$$

$$F = \bigcup m, \forall \{m, o\} \in LocalObjects, o \in E$$

// *T* est la liste des objets de même classe que *A*

// triée par ordre croissant du nombre de connexions *N* aux

// objets de *T*

$$T = \bigcup o, \{o, C, R, N\} \in RegisteredObjects, o \neq A$$

// Appel de la méthode de « load sharing » de l'algorithme ARID implémenté par

// le développeur : cette méthode attribue à chaque objet appelant *A* un objet de même

// classe que *A* ou *null* si *A* est le seul objet de sa classe et modifie ainsi la matrice

// d'incidence *L* de l'ARO surveillée

// Le résultat est *Q*, ensemble des correspondances entre les anciennes références de *A*

// et les références des objets temporaires devant être appelés pendant la

// migration de *A*. *Q* est de la forme $\{\{o, C, R\}, ...\}$: le nouvel objet de classe *C* appelé par

// l'objet *o* a pour référence *R*

Q ← ARID.selectReplacingTrustedObjects(*A*,*E*)

// Il reste à transmettre les références des objets de remplacement à chaque agent afin

// qu'il en informe ses objets locaux

Pour tout $m \in F$ faire

// $Q_m \subseteq Q$ désigne l'ensemble des correspondances pour les objets de l'agent *m*

Si $(Q \neq \varnothing)$ alors

$$Q_m = \bigcup \{o, C, R\}, \forall \{m, o\} \in LocalObjects, o \in E, \{o, C, R\} \in Q$$

Sinon

$$Q_m = \varnothing$$

Fsi

m.sendCalledObjectMigrationNotification(Q_m)

Fait
// La migration effective de *A* commence lorsque tous les agents ont
// pris connaissance de la notification de migration
Agent$_1$.sendMigrationCommand(*Agent*$_2$,*A*)
Faire
// Attendre de connaître la nouvelle référence *R'* de *A*
Tant que $(\{A, C, R', N\} \notin RegisteredObjects)$
// Prendre en compte les nouveaux objets souhaitant communiquer
// avec un objet de classe *C* en mettant à jour les ensembles *E* et *F*
// Le nouvel ensemble *E* désigne l'ensemble des objets souhaitant ou
// ayant déjà opéré une communication avec un objet de classe *C*
$E = \bigcup o, L_{o,j} \in L, \forall(\{o, C', R, N\}, \{j, C, R', N'\}) \in RegisteredObjects^2$
$F = \bigcup m, \forall\{m, o\} \in LocalObjects, o \in E$
$L_C = \bigcup R, \{C, R\} \in RegisteredObjects$
// Sélectionner le « meilleur » objet selon l'algorithme ARID
// fourni par le développeur (ou l'algorithme par défaut) en
// tenant compte des objets en cours de migration pour chaque
// objet concerné par la migration de *A*
// Q désigne l'ensemble des correspondances entre les objets appelants
// et les nouvelles références attribuées par l'algorithme ARID
$Q = \varnothing$
Pour tout $o \in E$ faire
$\{L, R'\} \leftarrow$ *ARID*.run(*o*, L_C ,*L*)
// *R'* est la nouvelle référence que doit appeler *o* lorsqu'il souhaite
// contacter un objet de classe *C*. Il se peut que cette référence soit
// identique à la référence précédente (l'objet appelé reste inchangé)
$Q = Q \cup \{\{o, C, R'\}\}$
Fait
Pour tout $m \in F$ faire
// Soit $Q_m \subseteq Q$ l'ensemble des correspondances pour les objets situés
// sous la responsabilité de l'agent *m*
$Q_m = \bigcup \{o, C, R\}, \forall\{m, o\} \in LocalObjects, \{o, C, R\} \in Q$
// Envoyer à l'agent l'ensemble des correspondances qui le concernent
m.newCalledObjectReferences(Q_m)
FinPrincipal

Agent.sendCalledObjectMigrationNotification(Q_m)
Pour tout $o, \{o, C, R\} \in Q_m$ faire
// Remplacer les anciennes références par les références
// temporaires données par le manager

$$CalledObjects_{Agent} = \left\{CalledObjects_{Agent} - \{o, C, R'\}\right\} \cup \{o, C, R\}$$

// Transmettre les nouvelles références aux objets

o.newCalledObjectReference(*C*,*R*)

Fait

FinAgent.sendCalledObjectMigrationNotification

Agent.sendMigrationCommand(*Agent₂*,*A*)

// Processus de migration proprement dit

// *S* désigne l'état courant de *A* et *S'* l'état de ses variables DOMS

{S,S'} ← *A*.destroy()

Agent₂.createMigrationObject(*A.ID*,*A.getClass*,*S*,*S'*)

FinAgent.sendMigrationCommand

Agent.newCalledObjectReferences(Q_m) :

Pour tout $o \in Q_m$ faire

// Remplacer les anciennes références par les références

// temporaires données par le manager

$$TrustedObjects_{Agent} = \left\{TrustedObjects_{Agent} - \{o, A, R'\}\right\} \cup \{o, A, R\}$$

// Transmettre les nouvelles références aux objets

o.newCalledObjectReference(*A*,*R*)

Fait

FinAgent.newCalledObjectReferences

Object.newCalledObjectReference(*A*,*R*) :

$$TrustedObjects_{Object} = \left\{TrustedObjects_{Object} - \{A, R'\}\right\} \cup \{A, R\}$$

FinObject.newCalledObjectReference

Object.destroy() :

S ← *Object*.getState()

Object.delete()

Retourner *S*

FinObject.destroy

Agent.createMigrationObject(*ID*,*Class*,*S*,*S'*)

// Créer une copie de l'objet *ID*

// en passant l'état *S* au constructeur de la classe :

// ces opérations sont réalisées par le module ODFL adéquat.

// *S'* est l'ensemble $TrustedObjects_{ID}$ de l'objet, qui sera utilisé

// lors de la construction du composant *DOMSobjectSpec*

ODFL.createMigrationObject(*ID*,*Class*,*S*,*S'*)

FinAgent.createMigrationObject

8.6 Interface DOMSobjectAPI :

L'API « DOMSobjectAPI » rassemble les méthodes de surveillance et de gestion propres aux objets.

Cette API est utilisée par les DOMSagents lorsque le manager souhaite :

- Obtenir des informations concernant les objets (classe d'appartenance, référence de l'objet, état de l'objet, nombre d'appels reçus et envoyés, etc.)
- Commander les objets (destruction de l'objet, modification des objets appelés, etc.).

Pour chaque méthode présentée sont indiqués :

- La signature de la méthode : lorsqu'une méthode ne renvoie aucun résultat, le mot clé *procédure* est employé ; dans le cas contraire, le type du résultat retourné est indiqué
- Les nom et type de ses arguments : cette partie est facultative
- La signification de chaque argument (facultatif)
- La signification de la valeur retournée (dans le cas où la méthode n'est pas une procédure)
- Une description du rôle et de l'utilisation de la méthode.

La représentation tabulaire suivante est employée :

[Type =] [Procédure] Nom_méthode ([Type Argument$_1$, Type Argument$_2$, …, Argument$_n$])	
. Argument$_1$. Argument$_2$. … . Argument$_n$	. Description Argument$_1$. Description Argument$_2$. … . Description Argument$_n$
Description du résultat attendu lorsqu'aucune erreur ne se produit	
Rôle de la méthode et utilisation	
. Erreur$_1$. Erreur$_2$. … . Erreur$_m$	. Description Erreur$_1$. Description Erreur$_2$. … . Description Erreur$_m$

Quelques conventions de notation sont appliquées à ce tableau :

- Une ligne comportant la mention « N/A » signifie qu'elle n'est pas applicable à la méthode décrite
- Le type des arguments et du résultat retourné peut être simple (Entier, Réel, Chaîne, Caractère, Booléen, etc.) ou complexe (Ensemble{Type$_1$ [,Type$_2$,…,Type$_n$]} pour un ensemble, Liste{Type$_1$ [,Type$_2$,…,Type$_n$]} pour un ensemble ordonné)
- {Type$_1$ [,Type$_2$,…,Type$_n$]} est la notation retenue pour les données comportant *n* champs (structures de données complexes)
- Valeur$_1$/…/Valeur$_n$ représente une énumération de *n* valeurs possibles
- Une astérisque en fin de nom de méthode indique qu'elle n'est pas supportée par cette version du système mais le sera dans une future version.

L'API permettant l'instrumentation des objets est la suivante :

Liste{Chaîne} = destroy ()	
N/A	N/A
Etat de l'objet	
Etat de l'objet conforme à la grammaire du protocole de transmission de l'état des objets	
N/A	N/A

Entier = getInvocationsNumber ()	
N/A	N/A
Nombre d'invocations (≥ 0)	
Cette méthode renvoie le nombre d'appels à l'initiative de l'objet depuis sa création	
N/A	N/A

Entier = getInvocationsNumberOnNode* ()	
N/A	N/A
Nombre d'invocations (≥ 0)	
Cette méthode renvoie le nombre d'appels à l'initiative de l'objet depuis son dernier déplacement (ou depuis sa création s'il n'a jamais été déplacé)	
N/A	N/A

Réel = getAverageCPUOnNode* ()	
N/A	N/A
Pourcentage	
Consommation processeur moyenne de l'objet depuis son dernier déplacement (ou depuis sa création s'il n'a jamais été déplacé)	
N/A	N/A

Réel = getAverageMemoryOnNode* ()	
N/A	N/A
Pourcentage	
Consommation mémoire moyenne de l'objet depuis son dernier déplacement (ou depuis sa création s'il n'a jamais été déplacé)	
N/A	N/A

Ensemble{Chaîne,Type,Entier,Liste{Valeurs}} = getState ()	
N/A	N/A
Etat de l'objet	
Etat courant de l'objet	

N/A	N/A

Booléen = getAvailability ()	
N/A	N/A
VRAI ou FAUX	
Retourne VRAI si l'objet est disponible, FAUX sinon	
N/A	N/A

Chaîne = getClass ()	
N/A	N/A
Chaîne de caractères	
Retourne la chaîne de caractères représentant la classe de l'objet	
N/A	N/A

8.7 Interface DOMSmanagementAPI :

L'API « DOMSmanagementAPI » rassemble les méthodes de surveillance et de gestion du SRO. Celles-ci sont organisées en trois groupes :

- Les méthodes de surveillance des machines
- Les méthodes de surveillance du réseau
- Les méthodes de surveillance et de gestion des objets.

La représentation tabulaire est employée est identique à celle présentée lors de la définition de l'API « DOMSobjectAPI ».

8.7.1 Méthodes de l'interface utilisateur :

Réel = getGlobalQoS (Réel alpha, Réel beta, Réel gamma, Réel mu, Réel maxDelay)	
. alpha	. Poids associé à l'indice de charge
. beta	. Poids associé à l'indice de temps de réponse
. gamma	. Poids associé à l'indice d'exclusion
. mu	. Poids associé à l'indice de répartition
. maxDelay	. Délai maximum, exprimé en millisecondes, utilisé par *getGlobalDelay*
Pourcentage	
Pourcentage indiquant la mesure de QoS globale de l'application surveillée. Les paramètres permettent de pondérer les critères de QoS retenus par DOMM	
. INVALID_MAXDELAY	. Le délai maximum doit être strictement positif
. DELAY_NOT_AVAILABLE	. Le manager n'a pas encore reçu de résultat d'invocation en termes de temps de réponse
. WEIGHT_OUT_OF_RANGE	. L'un des poids n'est pas compris dans l'intervalle [0,1]
. WEIGHT_SUM_ERROR	. La somme des poids n'est pas égale à 1

Réel = getGlobalLoad (Réel alpha, Réel beta)	
. alpha . beta	. Poids associé à l'indice de charge processeur des machines . Poids associé à l'indice de charge mémoire des machines
N/A	N/A
Pourcentage	
Pourcentage indiquant la mesure de charge globale de l'application surveillée. Les paramètres permettent de pondérer les critères de charges processeur et mémoire retenus par DOMM	
. WEIGHT_OUT_OF_RANGE . WEIGHT_SUM_ERROR	. L'un des poids n'est pas compris dans l'intervalle [0,1] . La somme des poids n'est pas égale à 1

Réel = getLoadRef (Chaîne agentID)	
. agentID	. Identifiant de l'agent
Valeur réelle	
Donne la valeur du référentiel de la machine identifiée par l'agent donné permettant de comparer les charges des machines entre elles (chaque machine dispose de son propre référentiel)	
. AGENT_NOT_FOUND	. L'identifiant de l'agent destination est incorrect car non encore répertorié par le manager

Réel = getGlobalDelay (Réel maxDelay)	
. maxDelay	. Délai maximum autorisé (en millisecondes) pour répondre aux critères de QoS en termes de temps de réponse
N/A	N/A
Pourcentage	
Renseigne sur le pourcentage d'invocations (depuis l'exécution du manager) dont le temps de réponse est inférieur ou égal au paramètre indiqué	
. INVALID_MAXDELAY . DELAY_NOT_AVAILABLE	. Le délai maximum doit être strictement positif . Le manager n'a pas encore reçu de résultat d'invocation en termes de temps de réponse

Réel = getGlobalDistribution (Réel alpha, Réel beta)	
. alpha . beta	. Poids associé à l'indice de répartition de l'ensemble des objets sur les machines (méthode *getNodesDistribution*) . Poids associé à l'indice de répartition des objets de même classe sur les machines (méthode *getObjectsDistribution*)
Pourcentage	
Pourcentage indiquant la mesure de répartition globale des objets de l'application surveillée. Les paramètres permettent de pondérer les critères de QoS retenus par DOMM	

. WEIGHT_OUT_OF_RANGE	. L'un des poids n'est pas compris dans l'intervalle [0,1]
. WEIGHT_SUM_ERROR	. La somme des poids n'est pas égale à 1

Réel = getNodesDistribution ()	
N/A	N/A
Pourcentage	
Pourcentage de répartition de l'ensemble des objets sur les machines disponibles	
N/A	N/A

Réel = getObjectsDistribution (Chaîne class)	
. class	. Classe des objets dont on souhaite connaître le degré de répartition
Pourcentage	
Pourcentage de répartition des objets de même classe sur les machines disponibles	
. CLASS_NOT_FOUND	. Le paramètre de classe ne correspond pas à une classe connue par le manager $(class \notin Classes)$

Entier = getTheoricalNodesDistribution ()	
N/A	N/A
Nombre d'objets (≥ 0)	
Nombre d'objets devant théoriquement être hébergés par chaque machine (à un objet près par excès)	
N/A	N/A

Entier = getTheoricalObjectsDistribution ()	
N/A	N/A
Nombre d'objets (≥ 0)	
Nombre d'objets de même classe devant théoriquement être hébergés par chaque machine (à un objet près par excès)	
N/A	N/A

Réel = getGlobalExclusion ()	
. agentID	. Identifiant de l'agent
Pourcentage	
Degré d'exclusion de l'ensemble des machines	
. NO_AGENT_REGISTERED	. Aucun agent n'est enregistré par le manager

8.7.2 Méthodes de surveillance des machines :

Ensemble{Chaîne} = getAgentsIDSet ()	
N/A	N/A
Ensemble d'identifiants ou ensemble vide	
Retourne l'ensemble des identifiants des agents (qui sont les noms de leur machines locales)	
N/A	N/A

Entier = getAgentsNumber ()	
N/A	N/A
Nombre d'agents (≥ 0)	
Renvoie le nombre d'agents identifiés et enregistrés auprès du manager. On a la relation suivante : $getAgentsNumber = Card(getAgentsIDSet)$	
N/A	N/A

Réel = getAgentLoadGradient (Chaîne agentID)	
. agentID	. Identifiant de l'agent
Valeur du gradient de charge unitaire	
Permet de prendre connaissance de la dernière valeur connue par le manager du gradient de charge instantanée de la machine sous la responsabilité de l'agent identifié en paramètre	
. AGENT_NOT_FOUND . GRADIENT_NOT_AVAILABLE	. L'identifiant de l'agent destination est incorrect car non encore répertorié par le manager . Le gradient n'est pas connu du manager car l'agent ne lui a encore pas communiqué de valeur

Réel = getAgentAverageLoadGradient (Chaîne agentID, Entier n)	
. agentID . n	. Identifiant de l'agent . Nombre de valeurs instantanées devant être prises en compte pour le calcul de la moyenne (plus cette valeur augmente, plus le calcul fait appel à des valeurs anciennes du gradient de charge instantanée) : $n > 1$
Valeur moyenne	
Permet de prendre connaissance de la moyenne des n derniers gradients de charge instantanée unitaires de la machine sous la responsabilité de l'agent identifié en paramètre	
. AGENT_NOT_FOUND . GRADIENT_NOT_AVAILABLE . INTERVAL_ERROR	. L'identifiant de l'agent destination est incorrect car non encore répertorié par le manager . L'utilisateur demande un calcul qui fait appel à des valeurs trop anciennes ou inconnues (par exemple lorsque l'agent vient d'être exécuté sur la machine) . Le nombre *n* doit être strictement supérieur à 1

AVAILABLE/LOADED/OVERLOADED = getAgentLoadState (Chaîne agentID)	
. agentID	. Identifiant de l'agent
Etat de charge	
Retourne le dernier état de charge connu par le manager de la machine identifiée par son agent	
. AGENT_NOT_FOUND . STATE_NOT_AVAILABLE	. L'identifiant de l'agent destination est incorrect car non encore répertorié par le manager . L'état courant n'est pas connu du manager car l'agent ne lui a encore pas communiqué de valeur

Entier = getCPU (Chaîne agentID)	
. agentID	. Identifiant de l'agent
Valeur entière, exprimée en MHz	
Retourne la puissance processeur à vide de la machine identifiée par l'agent	
. AGENT_NOT_FOUND	. L'identifiant de l'agent destination est incorrect car non encore répertorié par le manager

Entier = getMemory (Chaîne agentID)	
. agentID	. Identifiant de l'agent
Valeur entière, exprimée en Mo	
Retourne la puissance mémoire à vide de la machine identifiée par l'agent	
. AGENT_NOT_FOUND	. L'identifiant de l'agent destination est incorrect car non encore répertorié par le manager

Réel = getCurrentCPU (Chaîne agentID)	
. agentID	. Identifiant de l'agent
Pourcentage	
Retourne le pourcentage courant d'utilisation du processeur de la machine identifiée par l'agent	
. AGENT_NOT_FOUND	. L'identifiant de l'agent destination est incorrect car non encore répertorié par le manager

Réel = getCurrentMemory (Chaîne agentID)	
. agentID	. Identifiant de l'agent
Pourcentage	
Retourne le pourcentage courant d'occupation mémoire sur la machine identifiée par l'agent	
. AGENT_NOT_FOUND	. L'identifiant de l'agent destination est incorrect car non encore répertorié par le manager

Réel = getAgentLoad (Chaîne agentID, Réel alpha, Réel beta)	
. agentID . alpha . beta	. Identifiant de l'agent . Poids associé à l'indice de charge processeur . Poids associé à l'indice de charge mémoire
Pourcentage	
Pourcentage indiquant la mesure de charge courante de la machine identifiée par l'agent indiqué	
. AGENT_NOT_FOUND . LOAD_NOT_AVAILABLE . WEIGHT_OUT_OF_RANGE . WEIGHT_SUM_ERROR	. L'identifiant de l'agent destination est incorrect car non encore répertorié par le manager . La charge courante n'est pas connue du manager car l'agent ne lui a encore pas communiqué de valeur . L'un des poids n'est pas compris dans l'intervalle [0,1] . La somme des poids n'est pas égale à 1

Entier = getAgentRegisteredObjectsClassNumber (Chaîne agentID, Chaîne class)	
. agentID . class	. Identifiant de l'agent . Classe des objets locaux
Nombre d'objets (≥ 0)	
Nombre d'objets de classe « class » situés sur la machine identifiée par son agent local donné	
. AGENT_NOT_FOUND . CLASS_NOT_FOUND	. L'identifiant de l'agent destination est incorrect car non encore répertorié par le manager . Le paramètre de classe ne correspond pas à une classe connue par le manager $(class \notin Classes)$

Entier = getAgentRegisteredObjectsNumber (Chaîne agentID)	
. agentID	. Identifiant de l'agent
Nombre d'objets (≥ 0)	
Nombre d'objets situés sur la machine identifiée par son agent local donné	
. AGENT_NOT_FOUND	. L'identifiant de l'agent destination est incorrect car non encore répertorié par le manager

Réel = getAgentExclusion (Chaîne agentID)	
. agentID	. Identifiant de l'agent
Pourcentage	
Degré d'exclusion de la machine identifiée par son agent par rapport à l'ensemble des machines. Cette méthode retourne la valeur 100 (exclusion maximale) lorsqu'aucun objet n'y est enregistré	
. AGENT_NOT_FOUND	. L'identifiant de l'agent destination est incorrect car non encore répertorié par le manager

Entier = getAgentUnreachableNumber* (Chaîne agentID)	
. agentID	. Identifiant de l'agent
Valeur entière	
Nombre de fois où l'agent n'a pas pu être contacté par le manager. Ce nombre s'apparente au nombre de crashs de la machine sur laquelle réside l'agent	
. AGENT_NOT_FOUND	. L'identifiant de l'agent destination est incorrect car non encore répertorié par le manager

8.7.3 Méthodes de surveillance du réseau :

Réel = getCommunicationDelay* (Chaîne agentID$_1$, Chaîne agentID$_2$, Entier size, Réel timeOut)	
. agentID$_1$	. Identifiant de l'agent situé sur la machine à l'initiative de la communication
. agentID$_2$	. Identifiant de l'agent situé sur la machine à la réception de la communication
. size	. Taille (en octets) du paquet à transmettre
. timeOut	. Délai (en millisecondes) d'attente de la réponse
Temps de communication en millisecondes	
Retourne le temps de communication aller et retour (en millisecondes) entre deux machines. Cette méthode crée puis envoie un paquet de « size » octets de la machine agentID$_1$ vers la machine agentID$_2$ qui le lui renvoie aussitôt	
. SOURCE_AGENT_NOT_FOUND	. L'agent émetteur n'est pas enregistré par le manager
. DESTINATION_AGENT_NOT_FOUND	. L'agent récepteur n'est pas connu du manager
. SOURCE_AGENT_NOT_REACHABLE	. La communication avec la machine source n'a pas pu être établie
. DESTINATION_AGENT_NOT_REACHABLE	. La communication entre la machine source et la machine destination n'a pas pu être établie
. MESSAGE_TIMEDOUT	. La réponse n'est pas parvenue à la machine émettrice avant « timeOut » millisecondes

8.7.4 Méthodes de surveillance et de gestion des objets :

Booléen = objectExists (Chaîne objectID)	
. objectID	. Identifiant de l'objet donné lors de sa création par un DOMSagent
Valeur *VRAI* ou *FAUX*	
Retourne *VRAI* si l'identifiant correspond à un objet enregistré, *FAUX* sinon	
N/A	N/A

Booléen = objectIsMigrating (Chaîne objectID)	
. objectID	. Identifiant de l'objet donné lors de sa création par un DOMSagent
Valeur *VRAI* ou *FAUX*	

Retourne *VRAI* si l'objet considéré est en cours de migration, *FAUX* sinon	
N/A	N/A

Chaîne = getObjectReference (Chaîne objectID)	
. objectID	. Identifiant de l'objet donné lors de sa création par un DOMSagent
Référence de l'objet ou *null*	
Renvoie la référence de l'objet donné ou *null* si l'objet est de type demandeur	
. OBJECT_NOT_FOUND . OBJECT_NOT_AVAILABLE . NO_REF_AVAILABLE	. L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS . L'objet est en cours de déplacement, il n'a donc temporairement pas de référence . L'objet ne dispose pas d'une référence car il est de type demandeur

Chaîne = getObjectResponsibleAgentID (Chaîne objectID)	
. objectID	. Identifiant de l'objet donné lors de sa création par un DOMSagent
Identifiant de DOMSagent	
Permet de prendre connaissance de l'identifiant de l'agent situé sur la même machine que l'objet identifié par objectID (agent local)	
. OBJECT_NOT_FOUND . OBJECT_NOT_AVAILABLE	. L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS . L'objet est en cours de déplacement, il n'a donc temporairement pas d'agent local

Ensemble{Chaîne} = getObjectsClassesSet ()	
N/A	N/A
Ensemble de classes ou ensemble vide	
Retourne l'ensemble des classes d'objets enregistrées à l'instant courant	
N/A	N/A

Ensemble{Chaîne} = getMigrableObjectsClassesSet ()	
N/A	N/A
Ensemble de classes ou ensemble vide	
Retourne l'ensemble des classes d'objets migrables enregistrées à l'instant courant	
N/A	N/A

Chaîne = getObjectClass (Chaîne objectID)	
. objectID	. Identifiant de l'objet donné lors de sa création par un DOMSagent
Classe de l'objet	
Retourne la chaîne de caractères indiquant la classe d'appartenance de l'objet indiqué	
. OBJECT_NOT_FOUND	. L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS

Ensemble{Chaîne} = getObjectsFromClassSet (Chaîne class)	
. class	. Classe des objets demandés
Ensemble d'identifiants d'objets	
Retourne l'ensemble des objets de classe « class » enregistrés dans le système	
. CLASS_NOT_FOUND	. Le paramètre de classe ne correspond pas à une classe connue par le manager $(class \notin Classes)$

Entier = getObjectsFromClassNumber (Chaîne class)	
. class	. Classe des objets demandés
Nombre d'objets (≥ 0)	
Retourne le nombre d'objets de classe « class » enregistrés dans le système	
. CLASS_NOT_FOUND	. Le paramètre de classe ne correspond pas à une classe connue par le manager $(class \notin Classes)$

Entier = getObjectsNumber ()	
N/A	N/A
Nombre d'objets (≥ 0)	
Le nombre d'objets retourné tient compte des objets en cours de migration. Lorsqu'aucun objet n'est enregistré par le manager (ce qui peut arriver lorsque le message d'enregistrement n'est pas encore parvenu au manager), la méthode retourne 0	
N/A	N/A

Entier = getMovableObjectsNumber ()	
N/A	N/A
Nombre d'objets (≥ 0) ayant la propriété MOVABLE à VRAI	
Le nombre d'objets retourné tient compte des objets en cours de migration. Lorsqu'aucun objet n'est enregistré par le manager (ce qui peut arriver lorsque le message d'enregistrement n'est pas encore parvenu au manager), la méthode retourne 0	
N/A	N/A

Entier = getObjectRecvInvNumberOnNode* (Chaîne objectID)	
. objectID	. Identifiant de l'objet donné lors de sa création par un DOMSagent
Nombre d'invocations (≥ 0)	
Le résultat est le nombre d'invocations reçues par l'objet concerné depuis sa dernière migration (ou depuis sa création s'il n'a encore jamais été déplacé)	
. OBJECT_NOT_FOUND	. L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS

Entier = getObjectRecvInvNumber (Chaîne objectID)	
. objectID	. Identifiant de l'objet donné lors de sa création par un DOMSagent
Nombre d'invocations (≥ 0)	
Le résultat est le nombre d'invocations reçues par l'objet concerné depuis sa création. Ce nombre est approché à la constante *NB_CALLS* près (la valeur attribuée à cette constante est donnée par le développeur)	
. OBJECT_NOT_FOUND	. L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS

Liste{Chaîne} = getCalledObjectsIDListOnNode* (Chaîne objectID)	
. objectID	. Identifiant de l'objet donné lors de sa création par un DOMSagent
Liste d'identifiants ou liste vide	
Cette méthode retourne la liste des identifiants des objets appelés par l'objet identifié en paramètre depuis sa dernière migration (ou depuis sa création s'il n'a encore jamais été déplacé). Cette liste, triée par ordre d'appel, fournit l'historique des appels de l'objet depuis sa création. Ainsi, un même identifiant peut s'y retrouver à plusieurs reprises. La liste peut être vide s'il n'a appelé aucun objet (c'est en particulier le cas des objets fournisseurs)	
. OBJECT_NOT_FOUND	. L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS

Liste{Chaîne} = getCalledObjectsIDList (Chaîne objectID)	
. objectID	. Identifiant de l'objet donné lors de sa création par un DOMSagent
Liste d'identifiants ou liste vide	
Cette méthode retourne la liste des identifiants des objets appelés par l'objet identifié en paramètre depuis sa création. Cette liste, triée par ordre d'appel, fournit l'historique des appels de l'objet depuis sa création. Ainsi, un même identifiant peut s'y retrouver à plusieurs reprises. La liste peut être vide s'il n'a appelé aucun objet (c'est en particulier le cas des objets fournisseurs)	
. OBJECT_NOT_FOUND	. L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS

Ensemble{Chaîne} = getObjectsIDSet ()	
N/A	N/A
Ensemble d'identifiants ou ensemble vide	

Donne l'ensemble des identifiants des objets enregistrés. Son cardinal est donné par la méthode « getObjectsNumber ». L'ensemble peut être vide lorsqu'aucun objet n'est enregistré par le manager
N/A | N/A

Ensemble{Chaîne} = getMovableObjectsIDSet ()
N/A | N/A
Ensemble d'identifiants ou ensemble vide
Donne l'ensemble des identifiants des objets enregistrés ayant la propriété MOVABLE à VRAI. Son cardinal est donné par la méthode « getMovableObjectsNumber ». L'ensemble peut être vide lorsqu'aucun objet migrable n'est enregistré par le manager
N/A | N/A

Entier = getCreatedObjectsNumber ()
N/A | N/A
Nombre d'objets (≥ 0)
Retourne le nombre d'objets créés et enregistrés depuis le lancement du manager (ou 0 si $getObjectsNumber = 0$). Le résultat de cette méthode peut être différent de celui donnée par « getObjectsNumber » car certains objets ont pu être détruit. Par contre, la relation $getObjectsNumber \leq getCreatedObjectsNumber$ est vérifiée
N/A | N/A

Entier = getDeletedObjectsNumber ()
N/A | N/A
Nombre d'objets (≥ 0)
Retourne le nombre d'objets détruits par le développeur depuis le lancement du manager (ou 0 si aucun objet n'a été détruit). La relation $getDeletedObjectsNumber \leq getCreatedObjectsNumber$ est toujours vérifiée et de plus on a $getObjectsNumber = getCreatedObjectsNumber - getDeletedObjectsNumber$
N/A | N/A

Procédure migrateObject (objectID, agentID)	
. objectID . agentID	. Identifiant de l'objet donné lors de sa création par un DOMSagent . Identifiant de l'agent destination
N/A	
A la base du processus de répartition de charge, cette méthode permet de déplacer dynamiquement un objet donné vers une machine destination identifiée par son agent tout en conservant l'état de l'objet. La migration entraîne une modification de la matrice d'incidence de l'ARO car une référence temporaire est donnée aux objets en contact avec l'objet en cours de migration. Le processus de migration est annulé en cas d'erreur	

. OBJECT_NOT_FOUND . AGENT_NOT_FOUND . OBJECT_CREATION_ERROR . STATE_TRANSMISSION_ERROR	. L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS . L'identifiant de l'agent destination est incorrect car non encore répertorié par le manager . L'agent destination n'a pas pu créer un objet de même classe que l'objet identifié par objectID . L'état de l'objet n'a pas pu être transmis correctement

Ensemble{Chaîne} = getLocalObjectsIDSet (Chaîne agentID)	
. agentID	. Identifiant de l'agent concerné
Ensemble d'identifiants d'objets ou ensemble vide	
Retourne l'ensemble des identifiants des objets locaux situés sur la machine surveillée par l'agent indiqué en paramètre. Cet ensemble peut être vide lorsqu'aucun objet n'est enregistré chez l'agent (la machine fait partie des machines non utilisées : $agentID \notin UsedAgents$). Les objets en cours de migration vers cet agent sont comptés. Inversement, les objets locaux en cours de migration ne sont pas recensés	
. AGENT_NOT_FOUND	. L'identifiant de l'agent est incorrect car non répertorié par le manager

Ensemble{Chaîne} = getLocalMovableObjectsIDSet (Chaîne agentID)	
. agentID	. Identifiant de l'agent concerné
Ensemble d'identifiants d'objets ou ensemble vide	
Retourne l'ensemble des identifiants des objets locaux, ayant la propriété MOVABLE à VRAI, situés sur la machine surveillée par l'agent indiqué en paramètre. Cet ensemble peut être vide lorsqu'aucun objet migrable n'est enregistré chez l'agent. Les objets en cours de migration vers cet agent sont comptés. Inversement, les objets locaux en cours de migration ne sont pas recensés	
. AGENT_NOT_FOUND	. L'identifiant de l'agent est incorrect car non répertorié par le manager

Booléen = getClassMigrationAbility (Chaîne class)	
. class	. Identifiant de l'objet donné lors de sa création par un DOMSagent
Valeur *VRAI* ou *FAUX*	
Retourne la valeur courante de la propriété *MOVABLE* de la classe considérée. *MOVABLE=VRAI* signifie que tous les objets de la classe sont autorisés à être déplacé, contrairement à la valeur *MOVABLE=FAUX*	
. CLASS_NOT_FOUND	. Le paramètre de classe ne correspond pas à une classe connue par le manager $(class \notin Classes)$

Procédure setClassMigrationAbility (Chaîne class, Booléen ability)	
. class	. Classe des objets
. ability	. Propriété de migration (*VRAI* : l'objet peut être déplacé – *FAUX* : le déplacement n'est pas autorisé)
N/A	
Permet d'assigner à *ability* la valeur de la propriété *MOVABLE* à tous les objets de la classe indiquée en paramètre	
. CLASS_NOT_FOUND	. Le paramètre de classe ne correspond pas à une classe connue par le manager $(class \notin Classes)$

Ensemble{Chaîne,Entier} = getGeneratedExceptionsSetOnNode* (Chaîne objectID)	
. objectID	. Identifiant de l'objet donné lors de sa création par un DOMSagent
Ensemble d'exceptions et nombre d'exceptions ou ensemble vide	
Retourne l'ensemble des exceptions et le nombre de chaque exception générées par l'objet donné depuis son dernier déplacement (ou depuis sa création s'il n'a encore jamais été déplacé). L'ensemble retourné est vide si l'objet n'a pas généré d'exceptions	
. OBJECT_NOT_FOUND	. L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS

Ensemble{Chaîne,Entier} = getGeneratedExceptionsSet (Chaîne objectID)	
. objectID	. Identifiant de l'objet donné lors de sa création par un DOMSagent
Ensemble d'exceptions et nombre d'exceptions ou ensemble vide	
Retourne l'ensemble des exceptions et le nombre de chaque exception générées par l'objet donné depuis sa création. L'ensemble retourné est vide si l'objet n'a jamais généré d'exceptions	
. OBJECT_NOT_FOUND	. L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS

Entier = getGeneratedExceptionsNumberOnNode* (Chaîne objectID)	
. objectID	. Identifiant de l'objet donné lors de sa création par un DOMSagent
Nombre d'exceptions (≥ 0)	
Nombre d'exceptions générées par l'objet depuis son dernier déplacement (ou depuis sa création s'il n'a encore jamais été déplacé)	
. OBJECT_NOT_FOUND	. L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS

Entier = getGeneratedExceptionsNumber (Chaîne objectID)	
. objectID	. Identifiant de l'objet donné lors de sa création par un DOMSagent
Nombre d'exceptions (≥ 0)	
Nombre d'exceptions générées par l'objet depuis sa création	

. OBJECT_NOT_FOUND	. L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS

Procédure deleteObject (Chaîne objectID)	
. objectID	. Identifiant de l'objet donné lors de sa création par un DOMSagent
N/A	
Cette méthode détruit l'objet spécifié et le supprime de DOMS. Une mise à jour des objets connectés à l'objet détruit est effectuée afin de proposer à chacun d'eux un objet de remplacement de même classe. Si l'objet détruit est le seul de sa classe et qu'il a déjà été utilisé par d'autres objets, DOMS s'opposera à sa destruction (l'ensemble $Classes$ ne peut que croître)	
. OBJECT_NOT_FOUND . DELETION_ERROR	. L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS . L'objet ne peut pas être détruit car il est le seul de sa classe et a déjà été invoqué par le passé

Ensemble{Chaîne,Type,Entier,Liste{Valeurs}} = getObjectState (Chaîne objectID)	
. objectID	. Identifiant de l'objet donné lors de sa création par un DOMSagent
Ensemble contenant pour chaque variable de l'objet : son nom, sa taille (nombre d'éléments) et la liste de ses valeurs	
Cette méthode retourne l'état courant de l'objet indiqué en paramètre, c'est à dire l'ensemble des valeurs de ses variables conformément au protocole de transmission de l'état d'un objet	
. OBJECT_NOT_FOUND . STATE_TRANSMISSION_ERROR	. L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS . Certaines variables complexes n'ont pas pu être transmises

Entier = getCalledObjectsClassNumberOnNode* (Chaîne objectID, Chaîne class)	
. objectID . class	. Identifiant de l'objet donné lors de sa création par un DOMSagent . Classe des objets appelés
Nombre d'objets (≥ 0)	
Retourne le nombre d'objets de classe « class » appelés par l'objet indiqué en paramètre depuis son dernier déplacement (ou depuis sa création s'il n'a encore jamais été déplacé)	
. OBJECT_NOT_FOUND . CLASS_NOT_FOUND	. L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS . Le paramètre de classe ne correspond pas à une classe connue par le manager $(class \notin Classes)$

Entier = getCalledObjectsClassNumber (Chaîne objectID, Chaîne class)	
. objectID . class	. Identifiant de l'objet donné lors de sa création par un DOMSagent . Classe des objets appelés
Nombre d'objets (≥ 0)	
Retourne le nombre d'objets de classe « class » appelés par l'objet indiqué en paramètre depuis sa création	
. OBJECT_NOT_FOUND . CLASS_NOT_FOUND	. L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS . Le paramètre de classe ne correspond pas à une classe connue par le manager $(class \notin Classes)$

Ensemble{chaîne} = getCalledObjectsClassesSet (Chaîne objectID)	
. objectID	. Identifiant de l'objet donné lors de sa création par un DOMSagent
Ensemble de classes ou ensemble vide	
Retourne l'ensemble des classes d'objets appelés par *objectID* depuis sa création	
. OBJECT_NOT_FOUND	. L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS

Entier = getCalledObjectsClassesNumber (Chaîne objectID)	
. objectID	. Identifiant de l'objet donné lors de sa création par un DOMSagent
Nombre d'objets (≥ 0)	
Retourne le nombre de classes d'objets appelées par l'objet indiqué en paramètre depuis sa création	
. OBJECT_NOT_FOUND	. L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS

Réel = getObjectMembership (Chaîne objectID)	
. objectID	• Identifiant de l'objet donné lors de sa création par un DOMSagent
Pourcentage	
Mesure du degré d'appartenance de l'objet donné à sa machine locale en fonction du nombre d'appels internes et externes effectués depuis son dernier déplacement (ou depuis sa création s'il n'a jamais été déplacé)	
. OBJECT_NOT_FOUND . OBJECT_NOT_AVAILABLE	. L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS . L'objet est en cours de déplacement, il ne peut pas être contacté

Réel = getObjectReliability (Chaîne objectID, Réel alpha, Réel beta, Réel gamma)	
. objectID . alpha . beta . gamma	. Identifiant de l'objet donné lors de sa création par un DOMSagent . Poids associé à l'indice de fiabilité d'invocations . Poids associé à l'indice de fiabilité de communications . Poids associé à l'indice de fiabilité d'appels
Pourcentage	
Retourne le degré de fiabilité de l'objet, en tenant compte des fiabilités interne et externe décrites dans DOMM	
. WEIGHT_OUT_OF_RANGE . WEIGHT_SUM_ERROR . OBJECT_NOT_FOUND . OBJECT_NOT_AVAILABLE	. L'un des poids n'est pas compris dans l'intervalle [0,1] . La somme des poids n'est pas égale à 1 . L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS . L'objet est en cours de déplacement, il n'a donc temporairement pas d'agent local

Réel = getObjectInvocationsReliability (Chaîne objectID)	
. objectID	. Identifiant de l'objet donné lors de sa création par un DOMSagent
Pourcentage	
Retourne le degré de fiabilité $\Phi^{inv}_{objectID}$ de l'objet	
. OBJECT_NOT_FOUND . OBJECT_NOT_AVAILABLE	. L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS . L'objet est en cours de déplacement, il n'a donc temporairement pas d'agent local

Réel = getObjectCommunicationsReliability (Chaîne objectID)	
. objectID	. Identifiant de l'objet donné lors de sa création par un DOMSagent
Pourcentage	
Retourne le degré de fiabilité $\Phi^{comm}_{objectID}$ de l'objet	
. OBJECT_NOT_FOUND . OBJECT_NOT_AVAILABLE	. L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS . L'objet est en cours de déplacement, il n'a donc temporairement pas d'agent local

Réel = getObjectCallsReliability (Chaîne objectID)	
. objectID	. Identifiant de l'objet donné lors de sa création par un DOMSagent
Pourcentage	
Retourne le degré de fiabilité $\Phi^{call}_{objectID}$ de l'objet	

. OBJECT_NOT_FOUND . OBJECT_NOT_AVAILABLE	. L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS . L'objet est en cours de déplacement, il n'a donc temporairement pas d'agent local

PARTIAL/TOTAL/EXCLUSION = getObjectsLinksType (Chaîne objectID$_1$, Chaîne objectID$_2$)	
. objectID$_1$	. Identifiant de l'objet source donné lors de sa création par un DOMSagent
. objectID$_2$	. Identifiant de l'objet destination donné lors de sa création par un DOMSagent
Type de communication entre deux objets indiqués en paramètres	
Cette méthode permet de connaître le type de communication entre objectID1 (source) et objectID2 (destination) donnés selon les différents types mis en évidence par DOMM : invocation partielle (PARTIAL : $objectID_1 \rightarrow objectID_2$), complète (TOTAL : $objectID_1 \Rightarrow objectID_2$) et l'exclusion (EXCLUSION : $objectID_1 \circ\!\!- objectID_2$).	
. SOURCE_OBJECT_NOT_FOUND . DESTINATION_OBJECT_NOT_FOUND	. L'identifiant de l'objet *objectID$_1$* est incorrect car l'objet n'est pas répertorié par DOMS . L'identifiant de l'objet *objectID$_2$* est incorrect car l'objet n'est pas répertorié par DOMS

Procédure setNewTrustedObject (Ensemble{Chaîne objectID}, Chaîne class, Chaîne ref)	
. objectID	. Identifiant de l'objet donné lors de sa création par un DOMSagent
. class	. Classe de l'objet
. ref	. Référence de l'objet que les objets indiqués devront dorénavant appeler
N/A	
Permet de changer d'objet fidélisateur appelé par un ensemble d'objets. Pour chaque objet identifié dans l'ensemble donné en paramètre, l'objet de classe *class* dorénavant appelé sera l'objet dont la référence est *ref*. Note : la méthode ne vérifie pas que la référence indiquée est correcte (il faudrait pour cela appeler l'objet en question)	
. OBJECT_NOT_FOUND . CLASS_NOT_FOUND	. L'identifiant de l'objet est incorrect car l'objet n'est pas répertorié par DOMS . Le paramètre de classe ne correspond pas à une classe connue par le manager $(class \notin Classes)$

Ensemble{Chaîne} = getLastMovedObjectsSet ()	
N/A	N/A
Ensemble d'identifiants d'objets ou ensemble vide	
Cette méthode retourne l'ensemble des objets déplacés par l'algorithme AROD lors de sa dernière exécution. L'ensemble retourné est vide si l'algorithme est exécuté pour la première fois ou si aucun objet n'a été déplacé lors de sa dernière exécution	
N/A	N/A

Ensemble{Chaîne} = getMovedObjectsSet ()	
N/A	N/A
Ensemble d'identifiants d'objets ou ∅	
Cette méthode retourne l'ensemble des objets déplacés par l'algorithme AROD depuis sa première exécution	
N/A	N/A

8.8 AROD par défaut :

Le principe de fonctionnement de l'AROD proposé par défaut par DOMS est le suivant :

- Lorsque la charge moyenne des machines est strictement inférieure à un seuil *MAX_LOAD* fixé (constant), l'algorithme cherche à augmenter le degré d'exclusion des machines
- Si la charge moyenne du SRO est supérieure ou égale au seuil *MAX_LOAD* et que la différence maximum de charge entre deux machines est supérieure à *MAX_LOAD_DIFF*, l'algorithme cherche à équilibrer les charges entre les machines en transférant des objets des machines les plus chargées vers les machines les moins chargées
- Les objets déplacés ne sont pas ceux déplacés lors de la dernière exécution de l'algorithme.

Voici l'AROD proposé (le résultat de la méthode « run » est la liste des commandes de migration à appliquer à l'issue de l'exécution de l'algorithme) :

```
run()
    MAX_LOAD ← 80
    MAX_LOAD_DIFF ← 30
    // Soit A l'ensemble des machines du SRO
    A ← getAgentsIDSet()
    Si (getAgentLoad(0.5,0.5) < MAX _ LOAD) alors
        // Déplacement des objets afin d'augmenter l'exclusion
        // des machines
        // La charge de ces dernières n'est que secondaire
        E ← nodesExclusion(A)
    Sinon
        // Choisir les deux machines ayant la plus grande
        // différence de charge
        {diff,src,dest} ← maxNodesDiff(A,0.5,0.5)
        Si (diff > MAX_LOAD_DIFF) alors
            // Le SRO est déséquilibré
            // Déplacement de certains objets des machines les
            // plus chargées vers les machines les moins chargées
            E ← loadBalance(src,dest)
        Fsi
    Fsi
    Retourner E
FinRun
```

```
nodesExclusion(A)
    E ← {}
    // trier chaque machine de la moins à la plus exclue du SRO
    F ← TrierExclusionCroissante(A)
    Pour src = tête(F) à queue(F) faire
        // L est l'ensemble des objets locaux à la machine src
        // suceptibles d'être migrés
        L ← getLocalMovableObjectsIDSet(src)
        // Supprimer de cet ensemble les objets ayant été déplacés
        // lors de la dernière exécution de l'algorithme
        LMO ← getLastMovedObjectsSet()
        L ← L – LMO
        // Prendre l'objet (non demandeur) ayant le degré
        // de membership le plus faible et lui chercher une
        // autre machine
        objID ← minMembership(L)
        Si (getObjectMembership(objID) < seuil) alors
            dest ← destNode(objID,A)
            Si ( dest ≠ null ) alors
                // En déduire une nouvelle commande de migration
                E ← E ∪ {objID, src, dest}
            Fsi
        Fsi
    Fait
    Retourner E
FinNodesExclusion

minMembership(L)
    objID ← null
    // La valeur maximum de membership est 100%
    // Il va falloir la minimiser
    miniMembership ← 101
    // Sélectionner l'objet ayant le plus petit degré
    // de membership
    Pour tout o de L faire
        Si (getObjectMembership(o) < miniMembership) alors
            MiniMembership ← getObjectMembership(o)
            objID ← o
        Fsi
    Fait
    Retourner objID
FinMinMembership
```

```
destNode(objID,A)
    // La machine destination est la machine qui communique
    // le plus avec l'objet sélectionné
    // Pour la choisir, il faut compter le nombre d'objets qui
    // ont un lien avec l'objet sélectionné, par machine
    // La machine choisie est celle qui a une charge lui
    // permettant d'accepter l'objet objID
    // E est la liste des machines qui communiquent avec l'objet objID
    // triée par ordre décroissant du nombre de communications
    E ← nodesCommunicationList(objID, A – getObjectResponsibleAgentID(objID))
    Pour m = tête(E) à queue(E) faire
        Si (getAgentLoad(m,0.5,0.5) < 80) alors
            // la machine sélectionnée n'est pas trop chargée
            Retourner m
        Fsi
    Fait
    // Aucune machine n'est en mesure de recevoir objID
    Retourner null
FinDestNode

maxNodesDiff(A,alpha,beta)
    {max,min} ← selectMaxMinNodes(A,alpha,beta)
    diff ← getAgentLoad(max, alpha, beta) - getAgentLoad(min, alpha, beta)
    Retourner {diff,max,min}
FinMaxNodesDiff

selectMaxMinNodes(A,alpha,beta)
    max ← null
    min ← null
    loadmax ← -1
    loadmin ← 101
    Pour tout m ∈ A faire
        // Sélectionner deux machines :
        // la plus et la moins chargée du système
        load ← getAgentLoad(m,alpha,beta)
        Si (load > loadmax) alors
            max ← m
            loadmax ← load
        Sinon Si (load = loadmax) alors
            // Sélectionner la machine ayant le plus
            // fort gradient de charge
            Si (getAgentLoadGradient(m) > getAgentLoadGradient(max)) alors
                max ← m
```

```
                Fsi
            Fsi
            Si (load < loadmin) alors
                    min ← m
                    loadmin ← load
            Sinon Si (load = loadmin) alors
                // Choisir la machine ayant le plus faible
                // gradient de charge
                Si (getAgentLoadGradient(m) < getAgentLoadGradient(min)) alors
                    min ← m
                Fsi
            Fsi
        Fait
        Retourner {max,min}
    FinSelectMaxMinNodes

    nodesCommunicationList(objID,F)
        E ← {}
        // Insérer les machines dans la liste E en utilisant
        // la méthode nbCommunications ci-après
        Pour tout m de F faire
            E ← insérer(m,nbCommunications(objID,m))
        Faire
        Retourner E
    Fin NodesCommunicationList

    nbCommunications(objID,m)
        // Par défaut, aucune communication entre les objets de
        // la machine m et objID
        C ← getObjectClass(objID)
        nbComm ← 0
        Pour tout o de getLocalObjectsID(m) faire
            nbComm ← nbComm + getCalledObjectsClassNumber(o,C)
        Fait
        Retourner nbComm
    Fin NbCommunications

    loadBalance(src,dest)
        // Choisir un objet ayant le plus faible membership
        // parmi ceux de la machine la plus chargée du SRO
        // Celle-ci est sélectionnée à l'aide de ses indices de charge
        // et de gradient moyen de charge courants
        Si ((src = null) ∨ (src = dest)) alors
```

```
            Retourner {}
        Sinon
            // Choisir l'objet migrable de plus faible membership de src
            L ← getLocalMovableObjectsIDSet(src)
            Si (L = {}) alors
                // src ne contient aucun objet
                Retourner null
            Sinon
                objID ← minMembership(L)
                Retourner {objID,src,dest}
            Fsi
        Fsi
    FinLoadBalance
```

Note : cet algorithme ne respecte par la troisième règle lors du déplacement des objets. Il faudrait pour cela choisir une machine destination ne disposant pas d'objet de la classe de l'objet à migrer.

8.9 ARID par défaut :

L'algorithme de répartition d'invocation par défaut cherche à augmenter l'exclusion des machines en valorisant les communications locales entre les objets.

Cet algorithme se compose de trois méthodes, « run », « runMultiple » et « selectReplacingTrustedObjects », utilisées par le module LBCM ainsi que par le protocole de migration des objets répartis.

Les méthodes proposées par défaut sont les suivantes :

```
run(objID,E)
    // Cette méthode est utilisée lors de la première communication d'un objet
    // avec un objet de classe C
    // objID est l'identifiant d'un objet souhaitant appeler un objet de classe C
    // E est l'ensemble des objets de classe C susceptibles de répondre à objID
    // run permet de choisir un objet trustedObject parmi ceux de E
    // L'objet sélectionné est celui qui aura reçu le moins d'appels jusqu'à maintenant
    nbObj ← getObjectsFromClassNumber(getObjectClass(objID))
    Si (Card(E) = 1) alors
        Retourner getObjectReference(trustedObject), trustedObject ∈ E
    Fsi
    // Il y a plus d'un objet de classe C
    trustedObject ← null

    Soit o ∈ E
    E ← E − {o}
    TrustedObject ← o
    mincalls ← getObjectRecvInvNumber(o)
    Pour tout o ∈ E faire
        Si (objectIsMigrating(o) = FAUX) alors
```

```
                    nb ← getObjectRecvInvNumber(o)
                    Si (nb < mincalls) alors
                        mincalls ← nb
                        trustedObject ← o
                    Fsi
                Fsi
            Fait
            // retourner la référence de l'objet que objID doit appeler
            // null si aucune référence n'est disponible
            Si (trustedObject = null) alors
                Retourner null
            Sinon
                Retourner getObjectReference(trustedObject)
            Fsi
        FinRun

        runMultiple(objID,C,trustedObject)
            // Cette méthode est exécuté lorsqu'un objet objID
            // a appelé un objet trustedObject de classe C
            // un nombre de fois multiple de NB_CALLS
            // Soit E l'ensemble des objets de classe C
            E ← getObjectsFromClassSet(C)
            // Assigner à objID l'objet ayant reçu le moins d'appels
            // jusqu'alors : trustedObject
            newTrustedObject ← minReceivedCallsObject(E)
            Si (newTrustedObject ≠ trustedObject) alors
                setNewTrustedObject({objID},C,trustedObject)
            Fsi
        FinRunMultiple

        minReceivedCallsObject(E)
            Choisir obj tel que obj ∈ E
            min ← getObjectRecvInvNumber(obj)
            E ← E − {obj}
            Tant que (E ≠ ∅) faire
                Choisir o tel que o ∈ E
                nb ← getObjectRecvInvNumber(o)
                Si (nb < min) alors
                    min ← nb
                    obj ← o
                Fsi
            Fait
```

Retourner *obj*
FinMinReceivedCallsObject

selectReplacingTrustedObjects(*objID,E*)
// Cette méthode est exécutée lors de la migration d'un objet
// *objID* est l'identifiant de l'objet devant être déplacé
// *E* est l'ensemble des objets ayant déjà appelé *objID*
$F \leftarrow \{\}$
// Soit *C* l'ensemble des objets de même classe que *objID*
class ← *getObjectClass(objID)*
$C \leftarrow$ *getObjectsFromClass(class)* - *objID*
// Dans cet exemple, seule la méthode de « round robin » est utilisée
$G \leftarrow E$
Tant que $(G \neq \varnothing)$ faire
$D \leftarrow C$
Tant que $((D \neq \varnothing) \wedge (G \neq \varnothing))$ faire
Choisir $p \in G$
Choisir $o \in D$
$D \leftarrow D - \{o\}$
// Remplacer *objID* par *o* chez *p*
$F \leftarrow F \cup \{p, class, o\}$
$G \leftarrow G - \{p\}$
Fait
Retourner *F*
FinSelectReplacingTrustedObjects

8.10 Description des modules ODFL :

Ces modules sont intégrés au module OCEM des agents. Leur rôle est de créer des objets répartis lorsqu'ils sont sollicités par OCEM. Chaque module ODFL est un objet réparti (accessible via une interface indépendamment de son langage de programmation) local à OCEM.

Chaque ODFL prend en charge la création d'une classe d'objets implémentée dans un langage de programmation donné, si tant est que la fabrique de cette classe fût disponible. La création des fichiers sources des fabriques d'objets est prise en charge par le DOMSparser (décrit dans le paragraphe suivant).

Ainsi, les ODFL sont écrits dans des langages hétérogènes (en fonction des langages utilisés par les objets répartis) et suivent tous le pseudo-algorithme suivant (*C* désigne la classe de l'objet à créer et *state* l'état d'un objet de classe *C*, décrit selon la grammaire fournie) :

ODFL(*Class C, State state*)
Création d'un objet de type « Class » générique
Chargement dynamique de la fabrique d'objets de classe *C* demandée
Si pas d'erreur de chargement alors
Exécution de la fabrique qui crée alors un nouvel objet de classe *C* en lui donnant en paramètre l'état *state* fourni

```
        Sinon
            // La fabrique n'a pas pu être chargée
            // Elle doit être écrite dans un langage différent de celui supporté par cet ODFL
            Génération de l'exception CLASS_NOT_FOUND à destination de OCEM
        Fsi
    FinODFL
```

Le module OCEM invoque chaque ODFL tant que l'objet n'a pas été créé. Si aucun ODFL ne permet de créer un objet de la classe demandée, OCEM génère un message de type *OBJECT_CREATION_ERROR* et l'envoie au manager.

8.11 Description sommaire du DOMSparser :

Le « DOMSparser » est un organe facultatif dont le rôle est d'automatiser la modification du code des objets originaux en y ajoutant :

- Un composant autonome (objet délégateur autonome) de la classe *DOMSobjectSpec*, qui permet à chaque objet de conserver les mesures de QoS effectuées localement et de communiquer avec son agent local.
- De nouvelles méthodes destinées aux DOMSagents : la modification est effectuée à l'aide du mécanisme d'héritage de l'interface *DOMSobjectAPI* . Le corps des méthodes ajoutées fait appel à des méthodes du composant qui instancie la classe *DOMSobjectSpec* afin de limiter la taille et la complexité du code ajouté
- Des calculs de temps de réponse des invocations (pour les objets demandeurs et délégateurs) et de temps de traitement des requêtes (pour les objets délégateurs et fournisseurs).

Le second rôle du parser est de créer des fabriques d'objets correspondant aux différentes classes des objets composant l'application surveillée.

Ce modificateur de code est ici décrit de manière générale, mais un DOMSparser spécifique à chaque langage de programmation utilisé par l'ARO doit être écrit.

8.11.1 Ajout du composant autonome DOMSobjectSpec :

Soit une classe d'objets répartis appelée *Provider*. Le parser modifie le code de cette classe afin de rendre chacune de ses instances instrumentable par DOMS.

```
class Provider {
    DOMSobjectSpec DOMSobjSpec ;
    // Suite des déclarations des variables de classe
    …
    Provider() {
        // Constructeur par défaut : l'objet n'a pas
        // encore d'identifiant donné par DOMS
        DOMSobjSpec = new DOMSobjectSpec() ;
    }

    Provider(ObjectID id, State S, State S') {
        // Constructeur utilisé lors de la migration d'un objet Provider
        // S et S' désignent l'état d'un objet de même classe, conforme
        // à la grammaire proposée par DOMSP :
        // 1. Décomposer S selon la grammaire pour recouvrir
```

```
                // l'état de l'objet. Cette phase est propre à chaque objet
                // 2. Créer une instance de DOMSobjectSpec en lui fournissant
                // l'identifiant de l'objet source et l'état S' de ses variables DOMS
                DOMSobjSpec = new DOMSobjectSpec(id,S') ;
        }
        // Suite du code de la classe
        …
}
```

8.11.2 Modification de l'implémentation des méthodes de l'interface :

```
<type> methodName(…) {
        DOMSobjSpec.beginMethodExecution("methodName") ;
        // suite des actions produites par la méthode
        …
        DOMSobjSpec.endMethodExecution("methodName") ;
}
```

8.11.3 Gestion des invocations :

```
<type> methodName(…) {
        DOMSobjSpec.beginMethodExecution("methodName") ;
        // suite des actions produites par la méthode
        …
        // invocation d'un objet de classe C
        try {
                String ref = DOMSobjSpec.trustedObject("C") ;
                // obtension d'un objet de classe C à partir de
                // la référence ref
                try {
                        Object genericObject = orb.string_to_object(ref) ;
                } catch (InvalidReference e) {
                        ref = DOMSobjSpec.newObjectCall("C") ;
                        Object genericObject = orb.string_to_object(ref) ;
                }
                C objC = new genericObject.narrow() ;
                // invocation d'une méthode de objC
                …
        } catch (InvalidRefException e) {
        } catch (InvalidObjectClass e) {
                // aucune référence à un objet n'a pu être
                // trouvée par le manager
        }
        DOMSobjSpec.endMethodExecution("methodName") ;
}
```

8.11.4 Création de fabriques d'objets :

Pour chaque classe d'objet *Class*, le parser crée une fabrique d'objets portant le nom <*Class*>*Factory*. La classe contient le code nécessaire à la création d'un nouvel objet de classe *Class* ainsi que la création d'objets ayant le même état que des objets de cette classe créés auparavant (copies d'objets).

Ces fabriques utilisent les constructeurs d'objets modifiés par le parser. En particulier, la création de copies d'objets utilise le second constructeur présenté afin de modifier les variables internes de l'objet nouvellement créé ainsi que les variables de surveillances ajoutées par DOMS (introduites par le composant « DOMSobjectSpec » de DOMS).

9 Bibliographie :

[Adoud 00] H. Adoud, E. Rondeau, T. Divoux, « Configuration Of Network Architectures For Co-operative Systems », Proc. Of the 26th Euromicro Conference (Eurmicro Workshop On Multimedia and Telecommunications), Maastricht, the Netherlands, September 5-7, 2000

[Arabe 95] J. N. C. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, P. Stephan, « Dome : Parallel programming in a heterogeneous multi-user environment », Technical Report CMU-CS-95-137, School of Computer Science, Carnegie Mellon University, April 1995

[Badidi 98] E. Badidi, R. K. Keller, P. G. Kropf, V. Van Dongen, « LoDACE : une architecture de partage de charge dans les systèmes distribués objet », Proc. of the Colloque International sur les NOuvelles TEchnologies de la REpartition (Notere'98), pp 281-296, Montreal, QC, Canada, October 1998

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

[Becker 92] W. Becker, « Lastbalancierung in heterogenen Client-Server Architeckturen », Fackültätsbericht Nr. 1, Institut für Parallele und Verteilte Höchstleistungsrechner, Universität Stuttgart, 1992

[Bouchi 00] A. Bouchi, E. Lepretre, P. Lecouffe, « Un mécanisme d'observation des objets distribués en Java », Proc. of the Twelfth REncontres francophones du PArallélisme des ARchitectures et des systèmes (RENPAR'12), Besançon, France, June 2000

[Casavant 87] T. L. Casavant, J. G. Kuhl, « Analysis of three Dynamic Distributed Load Balancing Strategies with Varying Global Information Requirements », Proc. of the Seventh International Conference on Parallel and Distributed Computing Systems (PDCS'87), pp 185-192, Berlin, Germany, 1987

[Chatonnay 96] P. Chatonnay, B. Herrmann, L. Philippe, F. Bourdon, « Placement dynamique dans les systèmes répartis à objets », Calculateur parallèle, pp 11-30, 1996

[Cottin 00] N. Cottin, O. Baala, J. Gaber, M. Wack, « Management and QoS in Distributed Systems », Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'00), Volume III, pp X-Y, Las Vegas, NV, June 2000

[Daniel 00] J. Daniel, « Au cœur de CORBA avec Java », pp 9-12, Vuilbert, Paris, 2000, ISBN 2-7117-8659-5

[Fünfrocken 98] S. Fünfrocken, « Transparent Migration of Java-based Mobile Agents », Proc. of the Second International Workshop on Mobile Agents (MA'98), September 9-11, Springer-Verlag LNCS 1477, pp. 26-37, Stuttgart, Germany, 1998

[Geib 97] J.M. Geib, C. Gransart, P. Merle, « Corba, des concepts à la pratique », pp 9-11, pp 2-5, pp 29-30, pp 20-21, Masson, Paris, 1997

[Gelenbe 91] E. Gelenbe, « Product form queuing networks with negative and positive customers », Journal of Applied Probability, Vol. 28, pp 656-663, 1991

[Gelenbe 96] E. Gelenbe, A. Labed, « Esprit LTR Project 8144 – LYDIA, Load Balancing and G-networks : Design, Implementation and Evaluation », Technical Report, IHEI, Univ. René Descartes, Paris V, August 1996

[Lambert 98] A. B. Lambert et al., « Adaptative Analysis for the Design of Hardware Agents Using the Artificial Immune System Model for Resource Management of Heterogeneous Systems », Technical Report, MSSU-COE-ERC-98-10, Mississippi, August 1998

[Meyer 88] B. Meyer, « Object-Oriented Software Construction », Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1988

[Mowbray 95] T.J. Mowbray, R. Zahavi, « The Essential CORBA: Systems Integration Using Distributed Objects », John Wiley & Sons, New York, NY, USA, 1995

[OMG 97a] OMG, « CORBAservices: Common Objet Services Specification », Updated version : July 1997

[OMG 97b] OMG, « The Common Object Request Broker : Architecture and Specification », Rev.2.1, August 1997

[OMG RFP5] OMG, « RFP5 Submission Trading Object Service », OMG Document orbos/960506, OMG, Framingham, MA, 1996

[Orfali 95] R. Orfali, D. Harkey, J. Edwards, « Essential Distributed Objects Survival Guide », John Wiley and Sons Inc., New York, USA, 1995

[Pellegrini 00] M. C. Pellegrini, « Reconfiguration d'applications réparties : application au bus logiciel CORBA », Ph.D. Thesis, Institut National Polytechnique de Grenoble, October 2000

[Rackl 97] G. Rackl, « Load Distribution for CORBA Environments », Ph.D. Thesis, Technische Universität München, Januar 1997

[Robinson 96] J. Robinson et al., « A Task Migration Implementation for the Message-Passing Interface », Proc. of the Fifth Hight Performance Distributed Computing Conference (HPDC-5), pp 61-68, IEE Computer Society Press, Los Alamitos, CA, USA, 1996

[Russ 99] S. H. Russ, K. Reece, J. Robinson, B. Meyers, R. Rajan, L. Rajagopalan, C.-H. Tan, « Hector : An Agent-Based Architecture for Dynamic Resource Management », IEEE Concurrency, pp 47-55, April-June 1999

[Schnekenburger 97] T. Schnekenburger, G. Rackl, « Implementing Dynamic Load Distribution Strategies with Orbix », Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97), Volume II, pp 996-1005, Las Vegas, NV, USA, 1997

[Siqueira 99] F. Siqueira, « Quartz : A QoS Architecture for Open Systems », Ph.D. Thesis, Trinity College, University of Dublin, December 1999

[Stevens 92] W. R. Stevens, « Advanced Programming in the Unix Environment », Addison-Wesley, Reading, Mass., 1992

[Wegner 90] P. Wegner, « Concepts and Paradigms of Object-Oriented Programming », ACM OOPS Messenger, August 1990

[Zeltserman 99] D. Zeltserman, « A Practical Guide to SNMPv3 and Network Management », pp 108-109, Prentice-Hall, Inc., Upper Saddle River, New Jersey 07458, 1999, ISBN 0-13-021453-1

[Zinky 97] J. A. Zinky, D. E. Bakken, R. E. Schantz, « Architectural support for quality of service for CORBA objects », April 1997, BBN System and Technologies

[Zweiacker 97] M. Zweiacker, « The Persistent Object Group Service – An approach to fault tolerance of open distributed applications », Proc. of the IFIP/IEEE Intern. Conference on Open Distributed Processing and Distributed Platforms, pp 224-235, May 1997, Toronto, Canada

10 Table des illustrations :

11 Acronymes utilisés :

ANICM : Agent Node Information Collection Module ....p 44
AOCEM : Agent Object Command Execution Module ....p 45
AOICM : Agent Object Information Collection Module....p 44
API : Application Programming Interface ....p 39
ARID : Algorithme de Répartition d'Invocation Dynamique ....p 48
ARO : Application Répartie Objet....p 5
AROD : Algorithme de Répartition d'Objet Dynamique ....p 48
ATM : Asynchronous Transfert Mode ....p 19
CORBA : Common Object Request Broker Architecture....p 10
DCP : DOMS Communication Protocol....p 53
DIP : DOMS Invocation Protocol ....p 56
DMP : DOMS Migration Protocol ....p 70
DOMM : Distributed Object Management Model....p 26
DOMS : Distributed Object Management System ....p 39
DOMSP : Distributed Object Management System Protocol....p 52
ERO : Environnement Réparti Objet....p 6
GUI : Graphical User Interface....p 40
IDO : Invocation Statique d'Objet....p 8
IHM : Interface Homme Machine ....p 27
IP : Internet Protocol ....p 19
ISO : Invocation Dynamique d'Objet ....p 8
LBCM : Load Balancing Command Module ....p 45
NICM : Node Information Collection Module ....p 47
OCEM : Object Command Execution Module ....p 47
ODFL : Object Dynamic Factory Loader....p 47
OICM : Object Information Collection Module ....p 46
OMENS : Objects Manager, Environment and Network Supervisor ....p 60
OMG : Object Management Group....p 10
OOC : Object Oriented Concepts....p 51
PDPTA : Parallel and Distributed Processing Techniques and Applications....p 51
PVM : Parallel Virtual Machine....p 14
RMI : Remote Method Invocation ....p 5
RPC : Remote Procedure Call....p 6
RSVP : Resource Reservation Protocol....p 19
SGBD : Système de Gestion de Base de Données ....p 14
SRO : Système Réparti Objet ....p 6
TCP : Transmission Control Protocol....p 19
UDP : User Datagram Protocol….p 20