Université d'Evry-Val d'Essonne
Institut National des Télécommunications
Institut d'Informatique d'Entreprise

Rapport de Stage
DEA d'Informatique

FÉDÉRATION DYNAMIQUE DE SERVEURS

Nawel SABRI

Responsable de DEA : Gilles Bernot
Responsable de stage : Chantal Taconet

Septembre 1999

Ce stage de DEA a été réalisé au sein du laboratoire Systèmes Répartis du département Informatique de l'Institut National des Télécommunications

Table des matières

Introduction

Le développement des grands réseaux, et plus récemment celui des réseaux à haut débit, ont permis l'utilisation d'applications réparties sur des réseaux étendus. La tendance va vers la décentralisation. Les entreprises répartissent leurs serveurs sur plusieurs sites de manière à être proches des clients potentiels. Afin d'assurer une bonne disponibilité du service, il s'avère nécessaire de faire coopérer les serveurs entre eux.
La présente étude s'inscrit dans le cadre des environnements répartis. Elle s'intéresse à la structure de coopération d'un ensemble de serveurs déployés dans un réseau étendu.
[Tac97] propose un protocole d'administration de ce que nous appelons une **Fédération de Serveurs**. Habituellement, les fédérations de serveurs coopérants sont gérés manuellement par des administrateurs pour chaque serveur. Avec des administrateurs pas toujours coordonnés, la coopération de leurs serveurs n'est pas optimale. Selon le protocole défini par [Tac97], l'évolution de la fédération est automatiquement gérée quasiment sans aucune intervention humaine. De plus, une modélisation, appelée **Graphe de Serveurs Fédérés**, est proposée pour représenter les liens de communication entre serveurs et exploiter ainsi la topologie réelle du réseau afin d'optimiser les coûts de communications.

L'objectif de cette étude est de définir un outil générique de fédération dynamique de serveurs, sur la base de ce protocole. Par la notion de généricité, nous exprimons le fait que cet outil est indépendant du service que peut rendre les serveurs. Il s'applique à n'importe quel type de fédération. Celle-là peut être une fédération de serveurs de cache Internet, serveurs de news, traders CORBA,...etc. Cet outil permet l'administration automatique de la fédération et offre un service de propagation de messages. Nous étudions la réalisation de cet outil dans un environnement middleware qui est l'environnement CORBA.
Ce document est composé de cinq chapitres.

Le premier chapitre situe notre étude en en présentant les bases à savoir le travail de [Tac97]. Nous faisons une synthèse de ce qui a été proposé dans ce travail. Nous décrivons la structure de coopération qui est la Fédération de Serveurs et ses différentes procédures d'administration. Nous introduisons par la suite la fonction de propagation dans la fédération de serveurs avec ses différentes caractéristiques et sémantiques.

Dans le second chapitre, nous étudions les protocoles de communication de groupes. La coopération des serveurs de la fédération étant basée sur leurs capacités de communiquer, ils doivent disposer d'outils leur permettant d'envoyer des messages à un ou plusieurs autres serveurs. Ce qui introduit la notion de communication de groupe. Le chapitre deux est une synthèse d'une étude bibliographique faite sur la communication de groupe. Nous

commençons par présenter les différentes sémantiques et quelques systèmes de communication de groupe existants. Par la suite, nous nous intéressons tout particulièrement à la communication de groupe dans CORBA avec ses différentes approches et modèles.

Le troisième chapitre présente l'essentiel de notre travail. Il expose la conception de l'**Outil générique pour la Fédération Dynamique des Serveurs (OFDS)**. C'est un outil qui permet la fédération dynamique des serveurs. Nous en présentons le modèle. Ce dernier est un modèle orienté objet qui décrit des objets d'administration, qui s'occupent de la gestion automatique de la fédération, et des objets de communication qui s'occupent de la diffusion de messages dans la fédération. Nous détaillons ces deux aspects.

Le quatrième chapitre est une étude de cas. Il présente un projet entre l'INT et l'université Charles de Prague en république Tchèque appelé **projet BARRANDE**. Ce projet décrit une structure de coopération de serveurs de composants logiciels. Nous développons la spécialisation de la fédération de serveurs à ce type de service. Nous décrivons l'environnement auquel va s'appliquer notre modèle à savoir l'environnement basé composants logiciels, appelé "SOFA" (SOFtware Appliances) et la fédération résultante de cette application à savoir une fédération de serveurs de composants logiciels. Nous clôturons ce chapitre par une brève introduction d'une approche orientée composant pour la modélisation de la fédération.

Le cinquième chapitre aborde la mise en oeuvre de l'OFDS. Nous présentons l'environnement de programmation et les outils de mise en oeuvre. Nous exposons également les interfaces IDL de cet outil ainsi que les différents aspects de la partie implantée.

Chapitre 1

La Fédération de serveurs

Nous présentons dans ce chapitre une synthèse du travail de [Tac97] qui constitue les bases de la présente étude. Ce travail présente un protocole d'administration automatique d'une structure de serveurs coopérants que nous appelons **Fédération de Serveurs**. Il définit un modèle appelé: Graphe de réseaux coopérants qui, dans notre cas, va être utilisé sous le nom de **Graphe de serveurs fédérés**. En fait. ce graphe représente une fédération de serveurs, c'est à dire un groupe de serveurs qui coopèrent afin de rendre un service particulier à un ensemble de clients. Nous commençons par développer les différents éléments de ce modèle. Ce graphe de serveurs fédérés est sujet à des événements qui le modifient et contribuent à son évolution. Nous verrons comment est gérée cette évolution.
La fédération de serveurs est, par essence, basée sur la coopération de ses serveurs. Cette coopération est traduite par un échange continuel d'informations. Cette communication peut être de type communication point a point ou de type communication de groupe (diffusion d'informations). Nous abordons cet aspect de communication dans la fédération des serveurs notamment celui de la communication de groupe en en présentant le mécanisme ainsi que les sémantiques.

1.1 Une fédération de serveurs

Grâce aux technologies disponibles aujourd'hui sur les réseaux étendus, Les entreprises n'hésitent plus à répartir leurs serveurs n'importe où dans le monde suivant ainsi les clients potentiels. Afin d'assurer une bonne disponibilité du service, il apparaît nécessaire que les serveurs coopèrent entre eux.
Cette structure coopérante, nous l'avons appelé une **fédération de serveurs**. Une fédération de serveurs est un groupe de serveurs appelés **serveurs fédérés** . Les serveurs fédérés sont répartis sur plusieurs sites qui peuvent être géographiquement distants. Chaque serveur fédéré rend un service particulier à un ensemble de clients. Ce service est le même pour tous les serveurs de la fédération. Toute requête invoquée par un client est traitée soit localement par le serveur invoqué, soit, pour manque d'informations, par la coopération d'un ensemble de serveurs fédérés sollicités pour répondre à la requête.

1.2 L'architecture de coopération

1.2.1 Le Graphe de Serveurs Fédérés (GSF)

La fédération de serveurs est un ensemble de serveurs répartis dans un réseau étendu. Ils sont reliés par des routeurs et des liaisons physiques suivant une certaine topologie. Le GSF représente l'interconnexion des serveurs d'une fédération. Les caractéristiques de ce graphe sont les suivantes :

- Les noeuds de ce graphe représentent les serveurs fédérés.
- Les arêtes représentent les connexions entre les serveurs.
- Chacune des arêtes est valuée par la distance qui sépare ses deux extrémités. Cette distance peut exprimer le temps de latence, la largeur de bande, le coût financier, le nombre d'intermédiaires ou une combinaison de ces informations. Dans tous les cas, elle prend en compte la topologie du réseau sous-jacent. La distance entre deux noeuds n'est pas fixe, elle peut varier dans le temps.
- Le graphe est non orienté. Toutes les liaisons sont bidirectionnelles.
- Le graphe est connexe. A noter que ce graphe est potentiellement complet.

La figure 1.1 donne un exemple d'un graphe de serveurs fédérés répartis dans plusieurs villes.

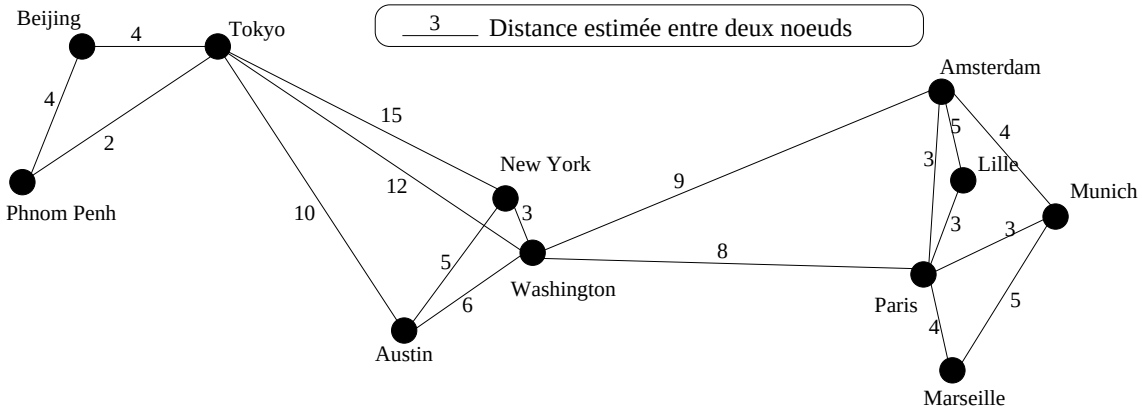

FIG. 1.1 – *Exemple d'un Graphe de serveurs fédérés*

1.2.2 Le serveur fédéré

Le serveur fédéré est l'entité de base de la fédération. C'est un serveur qui remplit deux rôles :

- Un **rôle générique** défini par deux principales tâches :

  1. *Une tâche d'administration*: Cette tâche est basée sur le connaissance du GSF auquel est attaché le serveur. Tout serveur fédéré participe à l'administration de la fédération. Ceci consiste en la gestion des différentes modifications que subit le GSF. Initialement, tous les serveurs possèdent la même vue du GSF. Ce dernier est sujet à des modifications tels que le rajout d'un nouveau serveur ou la suppression d'un serveur existant ou autre. Un protocole inter-serveurs permet de gérer ces modifications. Ce protocole est présenté dans la section 1.4.
  2. *Une tâche de communication*: La fédération de serveurs est basée sur la coopération de ses serveurs. De ce fait, un serveur fédéré est capable de communiquer avec un ou plusieurs autres serveurs. Cette capacité est exploitée à la fois pour administrer la fédération que pour rendre des services aux clients. La communication entre serveurs est détaillée dans la section suivante.

  Nous désignons par **serveur générique** Le serveur dont le rôle se limite au rôle générique.

- **Un rôle spécifique** défini par un service particulier rendu à un ensemble de clients. Ce service constitue une spécialisation du serveur générique. Il est indépendant des tâches génériques d'un serveur fédéré. Néanmoins, il en utilise les services notamment celui de la communication. Ce serveur spécialisé peut être un trader CORBA, un serveur de cache Internet, un serveur de news ou un serveur de composants logiciels...etc.

Dans le cadre de cette étude, on ne s'intéresse pas au service spécifique que peut rendre une fédération de serveurs. Nous définissons une fédération de serveurs génériques valable quelque soit le service qu'elle rend à ses clients.

1.3 La communication dans une fédération de serveurs

La coopération des serveurs de la fédération se traduit essentiellement par leur faculté de communiquer. Ils ont besoin de pouvoir diffuser des informations vers l'ensemble ou un sous ensemble des serveurs. Ceci fait intervenir le concept de communication de groupe que nous présentons au deuxième chapitre. Il est également question de communication point à point puisque, évidemment, les serveurs doivent être capable de s'adresser individuellement les uns aux autres.

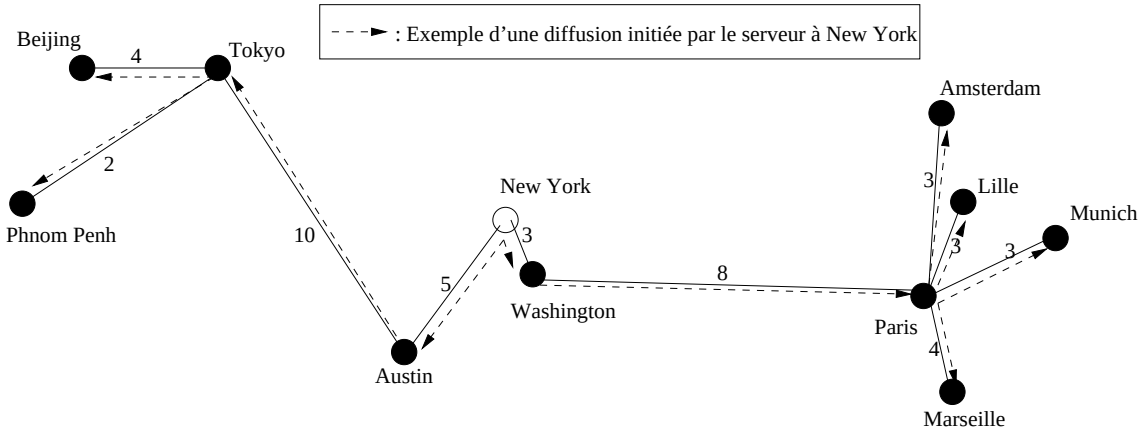

FIG. 1.2 – *Arbre de recouvrement de poids minimum*

La diffusion d'informations se fait en utilisant un arbre de recouvrement de poids minimum.

1.3.1 L'arbre de recouvrement de poids minimum

L'arbre de recouvrement choisi permet de diffuser des informations avec les avantages suivants :

- réduction du coût de communication. L'arbre est calculé de telle manière à ce que la somme des poids de ses arêtes soit minimum.
- répartition de la charge d'une diffusion sur tous les noeuds.
- suppression du contrôle d'arrêt de la diffusion.
- suppression des cycles

A partir du graphe de serveurs fédérés, un arbre de recouvrement de poids minimums est calculé par chaque serveur en utilisant l'algorithme de *Prim* [Pri57] ou l'algorithme de *Kruskal* [Kru56]. Nous appelons cet arbre **Arbre de diffusion**.
La figure 1.2 donne l'arbre de recouvrement de poids minimums de l'exemple de graphe précédent calculé avec l'algorithme de Prim.

Nous appelons **arbre de diffusion local** d'un serveur l'arbre constitué de ses voisins de premier et deuxième niveau dans l'arbre de diffusion. Pour pouvoir faire de la diffusion sur l'arbre de diffusion, un sommet doit connaître ses voisins de premier niveau. Pour pouvoir faire des reconfigurations en cas de panne, un sommet doit connaître ses voisins de deuxième niveau. Nous verrons l'intérêt de ces arbres locaux dans la section suivante.
La figure 1.3 présente l'arbre de diffusion local du serveur à Paris.

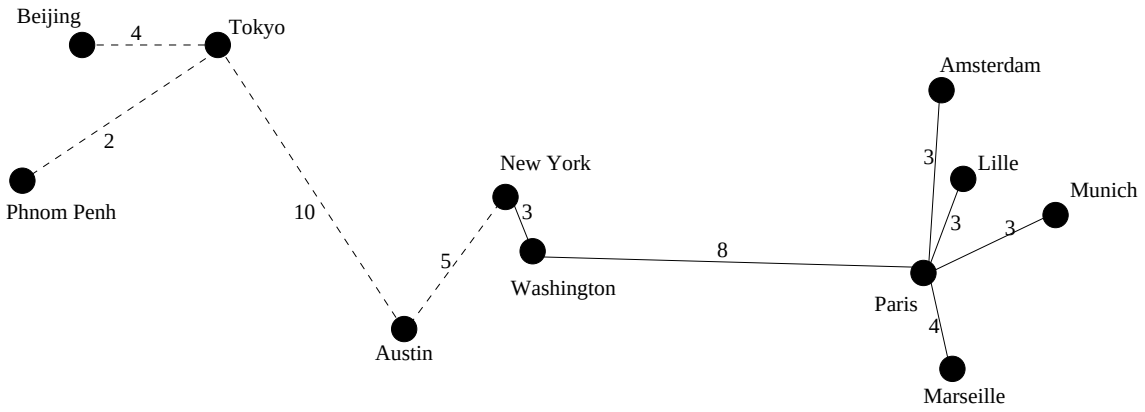

FIG. 1.3 – *L'arbre de diffusion local du noeud "Paris"*

1.4 L'administration d'une fédération des serveurs

Une fédération de serveurs peut évoluer suite à certains événements. Cette évolution se traduit par la modification du GSF et de l'arbre de diffusion par conséquent.
Une fédération de serveurs est caractérisée par un **numéro de version** qui sert à structurer son évolution. Une fédération avec une version i est définie par son graphe de serveurs fédérés GSF_i et son arbre de diffusion AD_i.
Il faut savoir que toute communication entre serveurs est précédée par un ensemble de tests dont la comparaison des numéros de version de la fédération. Ils faut qu'ils appartiennent à la même version de fédération pour pouvoir communiquer. Dans le cas contraire, des procédures de mise à jour de version sont lancées.
Nous présentons dans cette section l'ensemble des événements qui font évoluer la fédération ainsi que leur gestion.

1.4.1 Causes d'évolution de la Fédération de serveurs

Les événements qui peuvent causer la modification du GSF sont les suivants:

- **Ajout d'un serveur**: Quand un nouveau serveur s'ajoute à la fédération, un nouveau noeud doit être créé dans le GSF de tous les serveurs.

- **Suppression d'un serveur**: Un serveur peut être supprimé du graphe soit suite à une défaillance soit suite à une décision externe. Cette modification doit être répercutée sur les GSF de tous les serveurs.

- **Défaillance d'un serveur**: La machine ou le processus qui héberge le serveur peuvent être momentanément arrêtés. Pour assurer la continuité du service, l'arbre de diffusion doit être modifié. Si la défaillance est passagère, elle sera suivie d'un réveil du serveur défaillant, si elle est définitive, elle sera suivie par une suppression du serveur défaillant.
- **Défaillance d'un élément du réseau sous-jacent**: Cet événement peut empêcher la communication entre une ou plusieurs paires de serveurs. Il peut même aboutir à une partition du réseau. Il est aussi question de modification de l'arbre de diffusion. La gestion des défaillances est discutée plus en détail dans [Tac97].
- **Changement de topologie du réseau**: Cet événement touche les arêtes du GSF. Des changements de route conduisent à des rajouts ou suppressions de liens dans le GSF ainsi qu'à des changements de distances (changement des poids des arêtes du GSF).

Nous classons les événements en deux catégories :

1. Evénements passagers : Ce sont des événements dont les effets sont de courte durée (inférieur à 60 min) tels qu'une défaillance qui va être réparée dans le court terme, un changement momentané de route...etc. Les modifications qu'ils produisent sur le GSF ne sont pas définitives.
2. Evénements définitifs : Ce sont des événements tels que le rajout d'un nouveau serveur, la suppression définitive d'un serveur. Les modifications qu'ils produisent sur le GSF sont définitives.

1.4.2 Hypothèses

Sur les caractéristiques des événements cités ci-dessus, et suite à des études menées dans ce sens [Tac97], nous faisons les hypothèses suivantes:

- Les défaillances de machines sont rares (fréquence supérieure à la semaine).
- Les changements de route de communication sont peu fréquents.
- Les événements qui conduisent à des changements définitifs du GSF sont peu fréquents.

1.4.3 Gestion des modifications du GSF

Compte tenu du caractère incertain et passager de certains événements, traiter l'événement immédiatement après son apparition n'est pas envisageable. Trois niveaux de réaction sont prévus : un **fonctionnement de secours** est proposé afin d'assurer la continuité de service dans le cas d'une défaillance passagère; des **modifications locales** dans le cas ou l'événement va durer plus longtemps mais tout en restant passager; un **changement de version** du GSF pour prendre en compte toutes les modifications définitives.

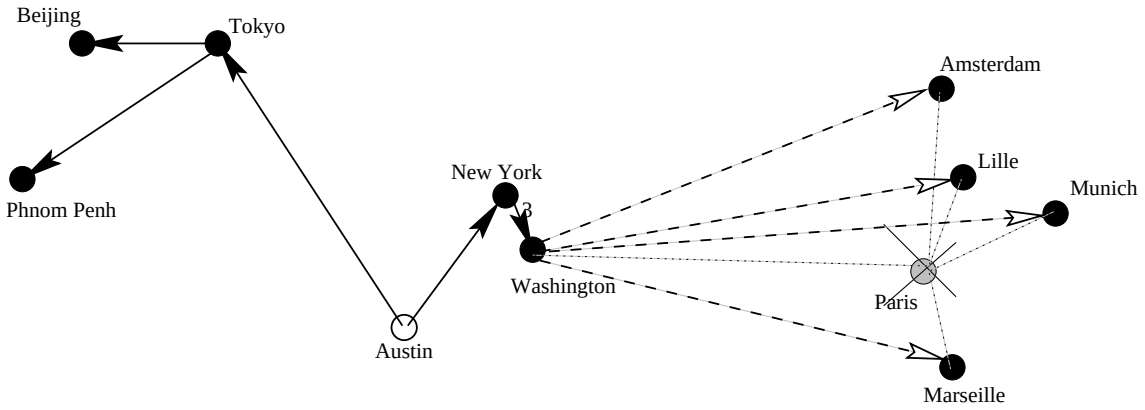

FIG. 1.4 – *Fonctionnement de secours*

Fonctionnement de secours

Le fonctionnement de secours concerne les événements de type défaillances. Quand l'un des serveurs ne répond pas et qu'on suspecte sa défaillance ou la défaillance d'un des éléments du réseau, des procédures de gestion de défaillances sont lancées et le fonctionnement de secours est restauré. Dans le cas ou le serveur défaillant doit propager une information, le serveur qui a découvert sa défaillance se charge de la propager à sa place. La figure 1.4 montre cette procédure en sachant que le serveur à *Austin* est l'initiateur de la diffusion.

modifications locales

Les modifications définitives et celles faisant suite à des défaillances réparables dans le court terme peuvent être effectuée localement. Ceci permet de répondre rapidement à des changements sans avoir à attendre un changement global de version. Ca permet également d'éviter des changements de version qui sont généralement coûteux pour des modifications de courte durée.
Les modifications locales consistent à les effectuer seulement au niveau de l'arbre de diffusion local du serveur en question. Ces modifications sont reportées d'une manière **cohérente** seulement aux voisins de premier niveau. C'est à dire que la modification est soit enregistrée chez "tous" les serveurs concernés soit qu'elle ne l'est pas. C est une opération atomique.
Prenons l'exemple d'un ajout d'un nouveau serveur. Quand il y a un nouveau serveur qui veut joindre la fédération, il demande son ajout à un des serveurs de la fédération : le serveur de contact (le plus proche) . Le serveur de contact diffuse, d'une manière cohérente, le message d'ajout à ses voisins de premier niveau. Si le message est accepté par tous, le nouveau serveur est ajouté comme feuille dans les arbres de diffusion locaux des serveurs impliqués.

Changement de version

Afin de diminuer la fréquence de changement de version du GSF, cette procédure permet de grouper plusieurs modifications définitives et de longue durée pour calculer le nouveau GSF et diffuser à tous les serveurs de la fédération la nouvelle version du GSF. Un nouvel arbre de diffusion est ensuite calculé.
Il existe un serveur appelé **Serveur de Changement de Version (SCV)** qui s'occupe des changements de versions. La procédure de changement de version se passe comme suit : Tous les serveurs fédérés envoient au fur et à mesure leurs modifications locales au SCV. Le SCV calcule au fur et à mesure un taux de dégradation de l'arbre de diffusion. Au delà d'un certain seuil, il décide le changement de version et diffuse la nouvelle version vers tous les serveurs de la fédération. Certains serveurs peuvent ne pas recevoir l'ordre de changement de version pour des raisons d'isolement ou de défaillance. Cet état sera détecté lors de l'étape d'accord sur la version du GSF qui précède tout échange d'information entre serveurs.

Le choix du SCV, La gestion de la défaillance du SCV, le réveil d'un SCV sont discutés en détail dans [Tac97]. Nous en avons présenté juste le principe.

1.5 La diffusion d'informations dans le GSF

Tel qu'on l'a présenté précédemment, la fédération de serveurs a besoin de diffuser des informations vers l'ensemble ou un sous ensemble des serveurs. Ceci se fait selon un arbre de diffusion qui est, en fait, un arbre de recouvrement de poids minimum calculé à l'aide d'un algorithme. Lorsqu'un serveur fédéré (la source) veut diffuser un message, la propagation du message se fait d'une manière répartie. La source diffuse le message vers ses voisins dans l'arbre de diffusion qui, chacun d'eux, le propage à son tour vers ses voisins excepté celui dont il vient de recevoir le message, jusqu'à arriver aux feuilles de l'arbre de diffusion.
Dans le cas des modifications locales, il est question d'une diffusion cohérente. On doit garantir la réception effective des modifications du GSF et attendre donc des acquittements de la part des récepteurs.
Dans le cas d'un changement de version, la sémantique de la diffusion est plus relâchée puisqu'on tolère la non réception du message de changement de version par quelques serveurs.
De ce fait, nous distinguons deux sémantiques à la diffusion d'information dans le GSF : Cohérente et relâchée. Nous retrouverons ces deux notions plus en détail au chapitre trois.
La diffusion d'information fait intervenir une notion plus générale qui est celle de la communication de groupe qui représente un vaste terrain d'investigation pour les chercheurs. Une étude lui est consacrée au deuxième chapitre.

1.6 Un environnement pour la fédération de serveurs : CORBA

On assiste ces dernières années à l'émergeance de plusieurs environnements de programmation qui facilitent le développement d'applications distribuées. Ils sont désignés par le terme de *middleware*. Ils agissent entre l'application et le système d'exploitation. Parmi ces environnements, il y a l'environnement **CORBA** proposé par l'OMG (Object Management Group) [OMG97].
Dans le cadre de cette étude, l'environnement CORBA est choisi comme plateforme de base au modèle proposé. Il permet de s'abstraire des langages de programmation et des systèmes d'exploitation utilisés. Aucune contrainte, qu'elle soit au niveau du réseau, des machines, des système d'exploitation ou du langage de programmation n'est imposée. Nous étudions la fédération dynamique de serveurs dans un environnement CORBA. Ceci implique une prise en compte des caractéristiques de cet environnement pour tous les choix de conception que nous présentons dans les chapitres suivants.

1.7 conclusion

Nous avons présenté un modèle de coopération de serveurs appelé: fédération de serveurs. Ce modèle est décrit par un graphe qui représente l'interconnexion des serveurs fédérés. Ces serveurs ont pour rôle de participer à l'administration de la fédération à laquelle ils appartiennent et rendre un service particulier à un ensemble de clients. Ils offrent également un outil de communication qui permet de les faire coopérer ainsi que pour accomplir leur tâches d'administration de la fédération.
L'administration de la fédération consiste à la faire évoluer d'une manière cohérente suite à des événements tels que le rajout d'un nouveau serveur, la suppression d'un serveur, survenance de défaillances...etc.
La fonction de communication est évidemment essentielle dans la fédération de serveurs. Un serveur doit être en mesure d'émettre des informations vers un ou plusieurs autres serveurs. Ce qui introduit un autre domaine qui est celui de la communication de groupe. Les diffusions d'informations, tel qu'on l'a vu précédemment, peuvent avoir plusieurs sémantiques. Aussi, le choix d'un environnement tel que CORBA donne une autre dimension au problème puisqu'il est dépourvu d'outils de communication de groupe. Toutes ces questions sont soulevées dans le chapitre suivant.

Chapitre 2

La communication de groupe

Aujourd'hui le souci du concepteur de systèmes distribués est d'obtenir des applications qui répondent à des besoins spécifiques tels que la haute disponibilité, la tolérance aux pannes, la fiabilité, le travail collaboratif ou la consistance. On veut rapprocher les serveurs des clients donc on crée des serveurs avec plusieurs répliques qui se répartissent en fonction de la localisation des clients; on veut faire collaborer plusieurs serveurs afin de rendre un meilleur service à un client; on veut qu'un système soit hautement disponible, c'est à dire un système qui assure une continuité de service malgré les éventuelles pannes qui peuvent survenir. La communication de groupe est nécessaire pour répondre à ces besoins.
Dans notre cas, la communication de groupe est nécessaire à la coopération des serveurs d'une fédération. Nous avons déjà expliqué ce besoin au chapitre précédent.
L'idée clé de la communication de groupe est de rassembler un ensemble d'entités communicantes dans un groupe logique et de fournir des primitives de communication dont celles de type un vers plusieurs, autrement dit des primitives de diffusion de messages vers tous les membres du groupe avec différentes sémantiques. La notion de groupe facilite ce type d'interactions : des messages sont envoyés à un groupe sans avoir à connaître le nombre, l'identité ou l'emplacement de ses membres.
Dans le cadre de cette étude, nous allons étudier la communication de groupe avec ses différents outils pour pouvoir en sélectionner ceux qui peuvent s'insérer dans notre environnement, qui est l'environnement CORBA.
Ce chapitre commence par exposer les différentes sémantiques et quelques systèmes de communication de groupe existants. On s'intéresse, par la suite, à CORBA et à ce qu'il offre comme outils de communication de groupe. Nous analysons alors trois types d'outils et jugeons l'intérêt de chacun pour notre étude. Et finalement, nous détaillons le modèle le plus intéressant pour nous qui est le système OGS (Object Group Service) [Fel98].

2.1 Les sémantiques de la communication de groupe

Les sémantiques de la communication de groupe sont nombreuses compte tenu de la diversité des besoins des applications. Nous en exposons les suivantes :

1. La fiabilité
   Il y a trois degrés de fiabilité :
   - Faire de son mieux (best effort):
     A l'envoi d'un message vers les membres d'un groupe aucune vérification de réception n'est effectuée.
   - Quorum:
     L'envoi d'un message n'est validé que si un nombre minimum des membres du groupe ont effectivement reçu le message.
   - Atomique:
     L'envoi d'un message n'est validé que si tous ses destinataires l'ont effectivement reçu.

2. L'ordre
   On entend par ordre l'ordre de réception des messages. Nous avons les catégories suivantes:
   First-In First-Out:
   Garantir que les messages diffusés par le même objet seront reçus dans le même ordre.
   - L'ordre causal:
     Soient deux messages m1 et m2 relié par une relation de cause à effet. L'ordre causal est respecté si et seulement si m1 est reçu avant m2.
   - L'ordre total simple:
     Les messages sont reçus selon leur ordre d'émission.
   - L'ordre total causal:
     Aussi bien l'ordre total que l'ordre causal sont assurés.

3. La dynamique
   Un groupe peut être **statique** ou **dynamique**. Un groupe statique est un groupe dont la composition ne change pas pendant la durée de vie du système. Ses membres restent inchangés. Un groupe dynamique est, à l'opposé du groupe statique, un groupe dont les membres changent. Ceci peut être du à un membre qui, suite à une défaillance, quitte le groupe ou un nouveau membre qui rejoint le groupe ou ...etc.

4. La synchronisation
   Il s'agit de voir de quel manière l'émetteur d'un message se synchronise avec le récepteur.
   - L'émetteur n'attend pas de réponses des récepteurs.

- L'émetteur attend la première réponse. Il se contente de la première réponse reçue.
- L'émetteur attend le maximum de réponses.
- L'émetteur attend autant de réponses que de messages émis.

5. L'autorisation d'émission
   Qui a le droit de diffuser des informations aux membres d'un groupe ?
   - Un émetteur unique :
     Un seul émetteur est autorisé à émettre des messages au groupe. Toute diffusion d'information passe obligatoirement par cet émetteur unique.
   - Un membre du groupe :
     Seuls les membres du groupe peuvent diffuser de l'information au groupe. Ce type de groupes sont dits "groupes fermés".
   - Un émetteur quelconque
     Tout émetteur, qu'il soit membre du groupe ou non, est autorisé à émettre des messages vers le groupe. Ce sont des "groupes ouverts".

Dans le cas de la fédération de serveurs, nous précisons les sémantiques de communication de groupe:

- Fiabilité : il y a deux degrés de fiabilité : "fiable" et "relâchée".
- Dynamique : bien entendu, le groupe de serveurs fédérés est dynamique.
- Synchronisation : Les réponses sont traitées en différé.
- L'autorisation d'émission : est donnée à un émetteur quelconque. Il suffit de connaître le groupe.

Nous reparlerons de ces sémantiques au chapitre suivant.

2.2 Quelques systèmes de communication de groupe

Nous décrirons dans cette section quelques systèmes de communication de groupe :

- **Isis**
  Isis [KB91] est un environnement de programmation développé à Cornell University. Il constitue un support à la communication de groupe de processus et offre une famille de protocoles qui gère ce type de communication avec différentes sémantiques. Parmi eux CBCAST et ABCAST. CBCAST implante la diffusion fiable des messages avec ordre causal. Quant à ABCAST, qui est une extension de CBCAST, il rajoute aux sémantiques supportées par CBCAST l'ordre total simple.
  Ce qui distingue Isis est sa capacité à gérer le recouvrement de groupe c'est à dire l'existence de plusieurs groupes en même temps avec des membres en commun et à garantir, dans ce contexte, l'ordre total et l'ordre causal aux diffusions de messages.

- **Horus**
  Le projet Horus visait à remettre le système Isis sous forme d'un ensemble de modules réutilisables. Ces derniers permettraient de construire d'autres protocoles de communication plus adaptés aux besoins spécifiques d'une application. tel était le cas pour les applications en temps réel. L'intérêt que présente Horus est essentiellement dans sa modularité. Une application développée dans Horus utilise seulement les modules dont elle a besoin.

- **Totem**
  Développé à l'université de Californie, Santa Barbara, il présente une famille de protocoles de communication permettant le développement de systèmes distribués tolérants aux pannes. Il assure la diffusion fiable avec ordre total vers un groupe de processus en utilisant une technique basée sur les jetons circulants. Son utilisation couvre aussi bien un réseau local qu'un ensemble de réseaux interconnectés. Totem gère les défaillances matérielles et leur recouvrement ainsi que le découpage et le raccordement de réseaux.

- **OGS (Object Group Service)**
  Développé à l'EPFL (Ecole Polytechnique Fédérale de Lausanne), il implante un service de communication de groupe dans CORBA. Il offre des primitives d'invocation de requêtes sur un groupe d'objets serveurs. Nous détaillons ce service en fin de chapitre.

2.3 La communication de groupe et CORBA

La spécification CORBA ne prévoit pas d'interfaces pour diffuser une requête vers un groupe de serveurs. Cependant, ces dernières années, plusieurs travaux visaient à définir un mécanisme de communication de groupe dans CORBA. Plusieurs RFP (Request For Proposal) en relation avec la communication de groupe ont été proposé à l'OMG (Object Group Management), dont une sur la tolérance aux pannes [ftc98] qui demande un service prenant en charge la réplication de serveurs, et une autre [Lau99] qui demande un protocole de diffusion de requêtes (MCASTIOP) basé sur le multicast IP.
Globalement, ces travaux ont adopté des approches qui peuvent être classées en 3 catégories [Fel98]:

2.3.1 L'approche "intégration"

Cette approche consiste à intégrer des fonctions de communication de groupe au sein même de l'ORB. Désormais, c'est l'ORB, sans aucun intermédiaire, qui se charge de la gestion du groupe et de la diffusion d'informations vers ses membres.
Deux systèmes ont adopté cette approche :

- **Electra**
  Electra est un environnement conforme à la spécification CORBA, avec des fonction-

nalités de communication de groupe. Ces dernières sont assurées grâce à l'intégration de fonctions de gestion de groupe dans l'adaptateur d'objets (BOA) telles que les fonctions de création de groupe, suppression de groupe....etc.
Electra offre deux modes de communication de groupe. Le mode transparent et le mode non transparent. Dans le mode transparent, les groupes sont complètement cachés aux clients. Ils sont manipulés de la même manière que des objets simples. Quant au mode non transparent, le programmeur manipule le groupe comme étant un ensemble d'objets et il peut accéder individuellement à chacun des objets.

- **Orbix+Isis**
  C'est un système de Iona Technologies et Isis Distributed Systems. Il est le produit de l'intégration d'Isis (voir 2.2) à Orbix. Les communications point à point sont gérées par Orbix [Bak97] et les communications de groupe sont gérées par Isis. C'est un système qui assure la tolérance aux pannes et permet l'équilibrage de charge en gérant la réplication d'objets.
  Le système Orbix+Isis est formé de deux programmes daemon : Le daemon d'Orbix (orbixd), et le daemon d'Orbix+Isis (isrd). Le premier se charge d'enregistrer les objets serveurs (dans le répertoire des implantations) et de diriger les requêtes vers ces serveurs. Quant au deuxième daemon, il se charge d'identifier les requêtes et de gérer la communication de groupe. Il enregistre des informations concernant les groupes d'objets dans le répertoire d'Isis (Isis Repository : IsR). Le client consulte le IsR pour en tirer les caractéristiques des groupes d'objets, ainsi que les opérations à invoquer sur ces groupes, leurs modes d'exécution, le nombre de réponses attendues...etc.

2.3.2 L'approche "interception"

Dans cette approche, l'ORB ne sait pas ce qu'est un groupe. Les requêtes sont interceptées, après qu'elles aient été formattées suivant le protocole IIOP, aussi bien au niveau client qu'au niveau serveur, à l'aide d'outils spécifiques au système d'exploitation. Elles sont ensuite passées à un module de gestion de communication de groupe qui s'occupe de les diffuser vers les membres du groupe.
Un exemple de système ayant adopté cette approche est le système Eternal.

- **Eternal**
  Eternal est un système qui offre à un environnement CORBA des capacités de gestion de la réplication d'objets. Il gère aussi bien la réplication des clients que celles des serveurs; et assure la consistance des objets répliqués.
  Eternal agit au dessus du système Unix. Il intervient lors de l'établissement d'une connexion IIOP au dessus de TCP/IP qui se fait par appel système, et ce en interceptant les messages IIOP avant qu'ils atteignent TCP/IP et en les redirigeant vers le système Totem (voir 2.2), qui s'occupe de les diffuser vers le groupe d'objets répliqués. En fait, tout système de communication de groupe offrant les mêmes fonctionnalités que Totem peut être utilisé.

Dans Eternal, il ya deux types de réplication:
- La réplication passive où, lors d'une invocation d'une opération sur un groupe d'objets, seul un objet serveur (serveur primaire) exécute l'opération. Les autres membres du groupe reçoivent seulement la requête et la sauvegardent prévenant ainsi le cas où le serveur primaire échouerait. A la fin de l'opération, Eternal procède à l'envoi des résultats de celle-ci au client via Totem et à la diffusion de l'état du serveur primaire vers toutes les répliques mettant ainsi leur état à jour.
- La réplication active où tous les membres du groupe exécutent l'opération invoquée par le client. Comme Eternal permet une diffusion avec ordre total, tous les serveurs du groupe reçoivent les mêmes messages dans le même ordre. Cette sémantique garantie la consistance des serveurs à la fin de l'opération.

2.3.3 L'approche "service"

L'approche service consiste à définir un service de communication de groupe qui se grefferait au bus CORBA à côté des autres services CORBA. Les fonctions de ce service se trouve ainsi spécifiées avec des interfaces IDL.
Un exemple de système basé sur cette approche est le système Object Group System (OGS).

- **Object Group Service (OGS)**
  Un service CORBA est un ensemble d'interfaces IDL (Interface Description Language) par lesquelles les clients accèdent au service. Techniquement, ces interfaces se traduisent par un ensemble d'objets répartis qui coopèrent entre eux pour assurer le service en question. L'OGS suit exactement la même définition.
  L'OGS est un ensemble d'interfaces qui fournissent des primitives de gestion de groupes d'objets et des primitives de communication avec ces groupes. Aucune hypothèse technique n'est émise quant à l'implantation de l'ORB utilisé. Il permet l'envoie de requêtes vers plusieurs objets serveurs. Il assure la diffusion fiable et non fiable ainsi que la diffusion avec ordre total. On y trouve deux types de communication, la communication typée et la communication non typée. Il est également composé de plusieurs services : service de messagerie, service de diffusion, service de concensus et le service de surveillance. Le modèle OGS sera détaillé ultérieurement.

2.4 Comparaison des différentes approches

Il est clair que ces trois approches remédient à l'absence de fonctions de communication de groupe dans CORBA. Les systèmes conçus sur la base de ces approches prouvent leurs validité.
Nous procédons à la comparaison de ces trois approches essentiellement par rapport à

certains critères qui mesurent l'adéquation de ces approches à notre application.

- **La compatibilité avec CORBA :**
  Un système compatible avec CORBA est un système conforme à la spécification de CORBA établie par l'OMG.
  L'approche d'intégration ne préserve pas cette compatibilité puisqu'elle nécessite la modification de l'ORB. Désormais la sémantique des références CORBA se trouve changée parce que l'ORB doit manipuler des références d'objets répliqués. Ainsi Orbix+Isis et Electra ont, tous deux, dû étendre la spécification du mapping IDL/C++ afin de gérer la communication de groupe.
  Quant aux deux autres approches (interception et service), elles préservent la compatibilité avec CORBA puisque la gestion de la communication de groupe est indépendante de l'ORB. En effet, dans l'approche d'interception, la communication de groupe est gérée au niveau du système d'exploitation. Et dans l'approche de service, elle est gérée grâce à des interfaces IDL et aucune hypothèse n'est émise quant à l'ORB utilisé.

- **La portabilité :**
  Préserver la portabilité d'un code c'est préserver la faculté de l'exécuter sur plusieurs bus CORBA rien qu'en le recompilant.
  L'approche d'intégration et celle de l'interception ne respectent pas cette propriété. Ces deux approches définissent elles-même leurs propres plateforme d'exécution et perdent donc la portabilité. Orbix+Isis et Electra, basés sur la première approche, construisent carrément un nouvel ORB (extension de l'ORB standard). Eternal, basé sur la deuxième approche, utilise des outils de bas niveau propre à Unix qui, par conséquent, dépendent de l'architecture.
  L'approche service, par contre, garantit la portabilité du code. A condition de ne pas utiliser des mécanismes spécifiques à une implantation particulière de l'ORB.

- **La performance :**
  En général, Le degré d'abstraction d'un système est inversement proportionnel à sa performance. Quand un système présente un haut niveau d'abstraction, sa performance diminue. Dans le contexte de la communication de groupe, la performance se mesure également à la qualité de son protocole de diffusion.
  L'approche d'intégration est en principe la plus performante parce que l'intégration de fonctions de communication de groupe au sein même de l'ORB réduit sensiblement le coût d'éventuelles redirections des messages à diffuser. Ceci dit sa performance dépendra directement de l'implantation de l'ORB sous-jacent. La performance de l'approche d'interception dépend, d'une part de l'ORB (temps d'une communication IIOP) puisqu'il faut intercepter les messages IIOP, d'autre part du module de communication de groupe utilisé puisque les messages destinés à un groupe sont dirigés vers ce module.
  Quant à l'approche de service, sa performance dépend uniquement de l'ORB utilisé.

- **La modularité :**
  La modularité est une réponse à la complexité. Elle consiste à décomposer un système en un ensemble de modules, plus ou moins indépendant. Chacun des modules peut être modifié, remplacé sans remettre en question le reste des modules.
  Il est difficile de juger la modularité d'une approche puisque celle-ci décrit plus une philosophie qu'une structure. Pour se faire il faut juger l'implantation de l'approche. Nous allons donc nous pencher sur les systèmes cités en exemple pour chaque approche et voir en particulier la modularité de la fonction de communication de groupe.
  Orbix+Isis et Electra, basés sur l'approche d'intégration, ont une architecture monolithique. Tout changement d'une fonctionnalité peut remettre en question tout le système. La fonction de gestion de la communication de groupe fait partie intégrante de l'ORB. Si on avait à changer par exemple le protocole de diffusion utilisé, tout le système se trouverait affecté.
  Dans Eternal, basé sur l'approche d'interception, la fonction de gestion de communication de groupe est assurée par un module à part, séparé de l'ORB. Mais celui-ci n'est pas conçu de manière à ce qu'il soit réutilisé dans d'autres applications CORBA.
  Quant à l'approche de service, l'OGS favorise la modularité et la réutilisabilité. Il est composé de plusieurs services, spécifiés avec des interfaces IDL, qui peuvent être utilisé dans des contextes autre que la communication de groupe.

2.5 Synthèse

Il est clair que l'évolution de CORBA vers la communication de groupe constituerait une aubaine pour les développeur d'applications distribuées. Ces dernières pourront bénéficier des avantages du concept de groupe. Cependant le rajout de nouvelles fonctionnalités doit préserver les éléments clés qui distinguent une solution middleware telle que CORBA. Ces éléments sont justement les critères de la comparaison précédente.
L'approche qui paraît épouser le plus l'esprit d'une architecture comme CORBA est l'approche service. Les systèmes développés jusque la, basés sur les deux autres approches, utilisent des outils de communication de groupe existant (Orbix+Isis utilise Isis, Electra utilise Horus ou Isis , Eternal utilise Totem) et qui ont été conçus, à l'origine, pour gérer des groupes de processus non pas des groupes d'objets. Il a fallu prévoir une interface logicielle qui fait le lien avec le monde des objets CORBA.
L'approche service respecte tout à fait le modèle de l'OMA (Object Management Architecture). Ainsi, la communication de groupe est un service à part qui se greffe au bus CORBA. Il offre un ensemble d'interfaces qui gèrent les groupes d'objets et la communication au sein de ces groupes.
A ce jour, nous pouvons dire qu'aucun de ces outils n'est standarisé par l'OMG. En attendant les futurs outils de communication de groupe dans CORBA, nous étudions la possibilité d'utiliser l'un d'eux. Nous recherchons une solution simple à utiliser et flexible. Le jour où des outils de communication de groupe seront standarisés, le passage à ces outils ne doit pas affecter notre système. Nous nous orientons donc vers l'approche service.

Nous détaillons dans la section suivante le système OGS afin de voir s'il répond à nos besoins.

2.6 Le modèle OGS (Object Group Service)

OGS est un service réparti qui gère des groupes d'objets CORBA et fournit des primitives de communication avec ces groupes. Les clients n'ont pas besoin de connaître le nombre, l'identité ou la localisation des membres. OGS est basé seulement sur des mécanismes CORBA, il est donc portable à tout ORB compatible avec CORBA et tourne sur tout système supportant le protocole IIOP de CORBA.
Un groupe, dans OGS, possède les caractéristiques suivantes :

- Le groupe est dynamique. Un objet peut joindre ou quitter un groupe à tout moment.
- Les membres du groupe partagent un *état* commun qui, globalement, représente la composition du groupe.
- Lorqu'un objet rejoint un groupe donné, il doit acquérir l'état du groupe. Ceci porte le nom de *transfert d'état*. C'est une opération atomique qui consiste à obtenir l'état d'un membre du groupe pour le transmettre au nouveau membre.
- Un groupe n'est pas un objet CORBA. Il n'est pas localisable parce qu'il est distribué.
- Un groupe est désigné par une référence unique indépendante de la localisation de ses membres et leur nombre.
- La référence des membres du groupe reste inchangée. Ils sont référencés de la manière dont CORBA l'a spécifié.

2.6.1 La structure d'OGS

La conception d'OGS est basée sur une approche orientée objet. Sa structure est composée d'un ensemble d'objets regroupés, selon leurs fonctions, en services CORBA.

Les services d'OGS

OGS est composé de quatre modules conçus sous forme de services CORBA. Chaque service est indépendant de l'autre, mais ceci n'empêche pas le fait qu'ils puissent interagir à travers l'ORB. Ces services sont décrits en détail dans [Fel98].

Ce sont :

- **Le service de Messagerie** qui fournit des primitives de communications point à point asynchrone, fiable ou non fiable. Les communications CORBA sont, par défaut,

synchrones. elles ressemblent aux appels RPC. Ce service permet à un client d'invoquer un serveur sans bloquer son exécution après l'invocation. De plus, il permet de spécifier une certaine qualité de service de la communication. Cette qualité concerne la fiabilité de la communication.

- **Le service de Monitoring** qui s'occupe de la gestion des défaillances des membres du groupe. Il permet de détecter les éventuelles pannes d'un serveur distant.

- **Le service de Consensus** qui permet de régler le problème d'une prise de décision commune par un ensemble d'objets distribués. Il résout les problèmes tels que celui des invocations atomiques ou le multicast avec ordre total. Un protocole de consensus permet à des objets de collaborer en vue d'une prise de décision commune, et ce malgré les éventuelles pannes des uns et des autres.

- **Le service de Groupe** qui fournit les primitives de gestion et de communication avec le groupe. Ce service est celui décrit dans la section suivante.

Le service de Groupe d'OGS

Il est intéressant pour nous de voir comment OGS gère-t-il les groupes d'objets. Cela nous permettra alors de projeter cette gestion sur la gestion des serveurs de la fédération. Dans la structure de ce service, trois familles d'objets sont à distinguer :

1. **Les objets membres**. Ce sont les objets qui forment le groupe. Ils réagissent à des événements tels que l'arrivée d'un message, un changement d'état ou un transfert d'état.

2. **Les objets du service**. Ils sont associés à un groupe donné, et permettent de communiquer avec lui. Ils sont, en quelque sorte, les représentant locaux d'un groupe d'objets. Ils garantissent la transparence du groupe vis-à-vis du client. Ils implantent les protocoles de gestion et de communication avec le groupe.

3. **Les objets de construction**. Ils servent à créer les objets de service. Ils gèrent leur cycle de vie en offrant des opérations de création et de destruction de ces objets.

2.6.2 Les interfaces de gestion de groupe dans OGS

le service de Groupe présente différentes interfaces associées à différentes vues que possèdent les objets les uns des autres. La figure 2.1 représente les classes d'objets de l'OGS.

1. **La vue des clients**. A travers cette vue, un client peut obtenir des informations concernant le groupe ou invoquer une requête sur ce groupe, ou encore communiquer individuellement (communication point à point) avec l'un de ses membres.

Le client a accès au groupe grâce à l'interface *GroupAccessor*. Un objet *GroupAccessor* est créé en invoquant l'opération *create* sur l'objet *GroupAccessorFactory*. Il représente un seul groupe d'objets, c'est à dire qu'il en faut un autre pour avoir accès à un autre groupe. Le *GroupAccessor* définit une opération de diffusion de message: *multicast()*. L'opération *get_view()* permet d'obtenir la composition du groupe.
L'interface *Invocable* décrit un objet qui peut recevoir un message quelconque. Nous verrons son intérêt par la suite.

2. **La vue du groupe**. La vue des membres du groupe encapsule celle des clients. Elle est définie par l'interface *GroupAdministrator* qui hérite du *GroupAccessor* et qui est également associée à un seul groupe. Les objets peuvent joindre le groupe en invoquant l'opération *join_group()*, et le quitter avec l'opération *leave_group()*. Les objets *GroupAdministrator* sont créés grâce à l'opération *create()* de l'objet *GroupAdministratorFactory*.

3. **La vue du service** est définie par l'interface *Groupable*, qui présente l'opération *view_change()* pour la notification d'un changement d'état, et les opérations *get_state()* et *set_state()* pour le transfert d'état. le *Groupable* hérite de l'*Invocable* est donc de sa capacité de recevoir des messages. C'est à travers cet objet que les objets membres sont notifiés d'un changement d'état.
En pratique, lorsqu'un objet rejoint un groupe, il invoque *join_group()* sur un *GroupAdministrator*, préalablement créé, en passant, en paramètre, Les références des objets *Groupable* du groupe. Ceux-ci vont servir à diffuser le changement d'état aux membres du groupe.

2.6.3 La communication dans OGS

Les sémantiques de diffusion

OGS permet plusieurs types de diffusion de requêtes vers un groupe d'objets serveurs, dont:

- **Fiabilité**. OGS garantit la diffusion fiable et non fiable. Avec la version non fiable, plusieurs objets sont invoqués en même temps sans garantir que tous les serveurs reçoivent l'invocation. Avec la version fiable, il est garanti que soit tous les serveurs reçoivent l'invocation soit qu'aucun d'entre eux ne la reçoive.

- **Ordre**. OGS garantit la diffusion avec trois types d'ordres:

  1. Une diffusion *First-In First-Out* (voir la section 2.1).
  2. Une diffusion avec *Ordre total simple* (voir la section 2.1).
  3. Une diffusion avec *Ordre causal* (voir la section 2.1).

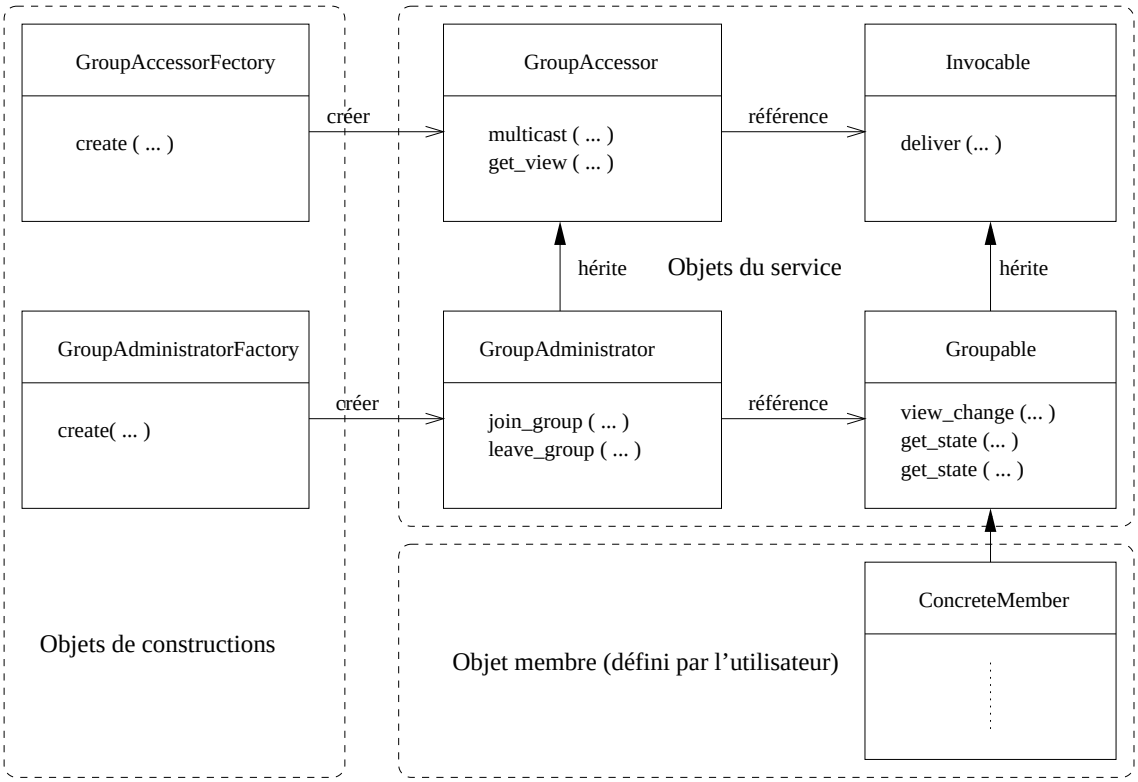

FIG. 2.1 – *Les objets de gestion de groupe dans OGS*

- **Synchronisation**. Le nombre de réponses attendues pour une requête diffusée est à spécifier parmi les options suivantes :

  1. Toutes les réponses. Toutes les réponses des membres du groupes sont attendues.
  2. La majorité des réponses.
  3. Une seule réponse. Seulement la première réponse est attendue.
  4. Pas de réponse. Ce sont des requêtes sans réponses.
  5. synchrone. Les invocations synchrones attendent la fin de l'opération associée mais elles sont sans réponses.

La communication typée

La communication typée offre au client une transparence par rapport à la gestion du groupe. Les clients peuvent ainsi invoquer un groupe d'objets serveurs comme si c'était un seul objet. C'est aux objets d'OGS de diffuser la requête invoquée vers le groupe, et filtrer les réponses afin de n'en retourner qu'une seule. Bien entendu, tous les serveurs du groupe doivent avoir la même interface IDL.
Ce type de communication a l'avantage d'épargner aux clients d'avoir à diffuser explicitement leurs requêtes, ce qui demanderait des opérations d'encodage et décodage de la requête avant l'envoi. Cependant, elle présente l'inconvénient de ne pas être assez flexible, puisqu'un client ne peut pas spécifier la sémantique de la diffusion.

La communication non typée

Ce type de communication permet d'envoyer des messages non typés. Un client qui veut envoyer un message à un groupe d'objet doit transformer son message en une valeur de type *any* et le diffuser explicitement avec *multicast ()* (voir la section 2.6.2). Au niveau de chaque serveur, le message est reçu à travers l'opération *deliver()* (voir figure 2.1).
Contrairement à la première, la communication non typée est plus flexible puisqu'un client peut envoyer n'importe quel type de données, et avec la sémantique qu'il désire. Seulement avec l'inconvénient d'avoir à gérer la présentation du message avant sa diffusion.

2.7 conclusion

Nous avons présenté les diverses sémantiques de communication de groupe et précisé celles dont nous avons besoin pour supporter la fédération de serveurs. Nous avons également exposé quelques systèmes de communication de groupe. Quelques uns de ces systèmes ont été utilisés afin de concevoir des outils de communication de groupe dans un environnement CORBA. La spécification CORBA ne prévoit pas d'outils de communication de groupe. Quelques travaux menés ces dernières années ont aboutit à des systèmes de communication de groupe dans un environnement CORBA. Nous pouvons classer ces travaux selon trois approches “intégration”, “interception” et “service”. Nous avons présenté ces

trois approches, et en avons fait la comparaison, et nous avons opté pour l'approche "service". Désormais, nous voyons la fonction de communication de groupe dans la fédération de serveurs comme étant un ensemble d'interfaces IDL décrivant un service à part, indépendant de son contexte d'utilisation. Un système qui implante cette approche est le système OGS (Object Group service). Nous nous sommes penché sur la gestion de groupes dans OGS pour étudier la possibilité de l'utiliser dans notre système.
Dans la fédération de serveurs, L'ensemble des serveurs constitue notre groupe d'objets. L'administration de ce groupe est déjà assurée et définie par le protocole que nous avons présenté au premier chapitre. Par conséquent, nous n'avons pas besoin de la gestion des groupes imposée par OGS, mais uniquement des interfaces de diffusion de messages. De ce point de vue, et si nous considérons un groupe comme étant un ensemble d'objets auxquels on diffuse des informations, nous n'avons pas un seul groupe mais un groupe de groupes qui diffèrent d'un serveur à un autre, et d'une diffusion à une autre. En effet, dans notre cas, la diffusion de messages suit la structure de l'arbre de recouvrement. De ce fait, chaque serveur diffuse vers un groupe constitué par ses voisins de premier niveau, différent pour chaque serveur et pour chaque diffusion (selon le serveur source de la diffusion). En fait, chaque serveur appartient à autant de groupes qu'il a de voisins dans l'arbre de diffusion. A cause de la nature de cette structure et la dynamique des groupes de l'arbre de diffusion, l'utilisation de OGS alourdirait considérablement le fonctionnement de la fédération.
Nous décidons donc de ne pas utiliser OGS tel quel. Nous prévoyons de nous inspirer de son modèle pour définir un module de gestion de la diffusion de messages dans la fédération de serveur. Ce module est présenté au chapitre suivant.

Chapitre 3

Outil générique pour la fédération dynamique des serveurs (OFDS)

Ce chapitre représente l'essentiel de notre étude. Il présente l'outil que nous proposons pour la fédération dynamique de serveurs (FDS) et que nous appelons **Outil générique pour la Fédération Dynamique de Serveurs (OFDS)**. Nous commençons par rappeler les objectifs de notre étude. Nous présentons par la suite une approche orientée objet pour décrire le modèle de fédération de serveurs présenté au premier chapitre qui est à la base de l'OFDS. Nous détaillons les différents objets du modèle ainsi que les relations qui les relient. Nous abordons encore une fois, par rapport à l'étude faite sur la communication de groupe au deuxième chapitre, la gestion de la communication dans l'OFDS. Nous présentons un modèle simplifié de communication de groupe inspiré du modèle OGS (voir la section 2.6). Et enfin, nous nous intéressons aux applications de la FDS: comment appliquer le modèle que nous proposons sur une fédération de traders CORBA par exemple.

3.1 Objectifs : fédération dynamique de serveurs (FDS)

Au premier chapitre nous avons présenté un modèle de fédération de serveurs. Ce modèle permet de gérer un groupe de serveurs déployés dans un réseau étendu, et susceptibles de coopérer afin de rendre un service aux clients d'un organisme décentralisé. Il décrit principalement un protocole d'administration suivant lequel l'évolution de la fédération de serveurs peut être automatiquement gérée. c'est à dire que tout changement dans la composition ou la structure de la fédération est pris en compte sans aucune intervention humaine. Ce protocole est basé sur le Graphe de Serveurs Fédérés (GSF) et la stratégie de modification de ce graphe.
Généralement, dans les systèmes à base de serveurs coopérants comme le système de news *USENET* [BK86], chaque serveur est géré manuellement par un administrateur qui fixe lui même la liste des serveurs voisins. Les administrateurs de deux serveurs voisins n'étant pas forcément coordonnés, il peut y avoir des cycles dans les relations de voisinage. De plus, ces dernières ne prennent pas en compte la topologie du réseau sous-jacent [Tac97].

Deux serveurs géographiquement proches l'un de l'autre peuvent utiliser un chemin de communication complexe. Une administration automatique avec le GSF prend en compte tout cela.

L'objectif de cette étude est de définir un outil générique de fédération dynamique de serveurs sur la base du protocole cité ci-dessus. Par la notion de généricité, nous exprimons le fait que la nature du service rendu par cette fédération est quelconque. Aucune hypothèse n'est émise concernant ce service. La fédération que nous définissons est censée s'appliquer sur n'importe quel groupe de serveurs.

3.2 Conception orientée objet de l'OFDS

Pour répondre à la complexité d'un système, on a souvent recours à la décomposition. C'est ce qui a donné lieu au concept d'objet. Un objet est une abstraction d'une entité du monde réel; il possède certaines propriétés et un certain comportement. Contrairement à une approche procédurale classique, une conception orientée objet décompose le système en une collection d'objets qui interagissent représentant ainsi son comportement. Ceci donne flexibilité, abstraction et facilité de développement à un système. Puisque nous utilisons l'environnement CORBA, il est intéressant de préciser la notion d'objet tel que la définit l'OMG: un objet est une entité identifiable qui fournit un ou plusieurs services à un ou plusieurs clients [OMG97].

L'étude que nous proposons vise essentiellement à reformuler le modèle présenté au premier chapitre en adoptant une approche orientée objet. Il s'agit de trouver une décomposition de ce modèle en un ensemble d'objets qui modélisent les différents acteurs d'une fédération de serveurs et offrent ses services.
Il est important de préciser que tous les objets dont nous allons parler dans ce chapitre ne sont pas forcément des objets CORBA. Nous aborderons cet aspect d'implantation au dernier chapitre.

3.3 Structure de la FDS

Tel qu'on l'a présenté au premier chapitre, une fédération de serveurs génériques remplit deux principales tâches: une tâche d'administration et une tâche de communication. Ce qui justifie l'existence de deux familles d'objets:

- **Objets d'administration** qui gèrent le cycle de vie de la FDS. Ils détiennent toutes les données concernant la fédération dont le GSF (Graphe de serveurs fédérés) et l'arbre de diffusion, et assurent toutes les opérations d'administration.

- **Objets de communication** qui fournissent un support aux communications multipoints entre serveurs avec différentes sémantiques. Leurs connaissances se limitent aux références des objets auxquels ils doivent envoyer les messages dont le contenu

leur est également inconnu. Ils sont complètement indépendant des objets d'administration. Ils peuvent être utilisés aussi bien dans une fédération de serveurs que dans une toute autre application.

3.3.1 Le modèle d'administration

Ce modèle décrit les objets d'administration ainsi que les différentes relations qui existent entre eux. La figure 3.1 montre le diagramme des classes du modèle d'administration en utilisant la notation UML (Unified Modeling Language).

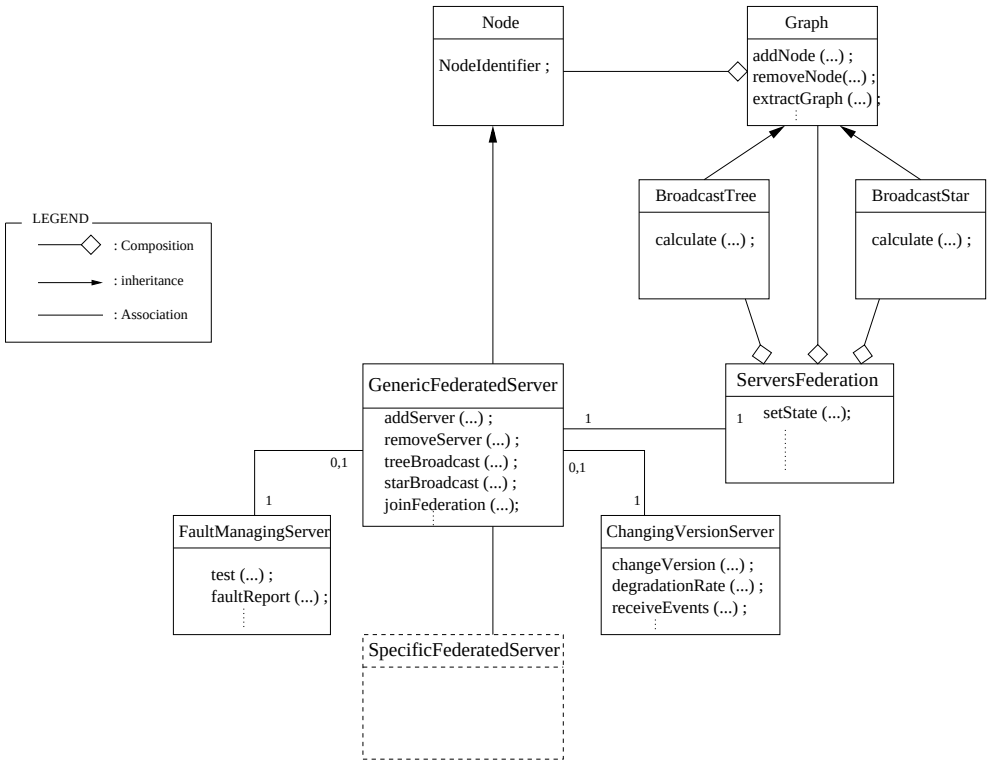

FIG. 3.1 – *Diagramme des classes du modèle d'administration*

Les objets *Node, Graph, BroadcastTree, BroadcastStar* permettent de gérer la FDS en tant que graphe, en faisant abstraction du fait que cela soit un réseau de serveurs. Ils gèrent les aspects en relation avec la théorie des graphes de la FDS.

– L'objet *Graph* représente un graphe quelconque composé d'un ensemble de sommets. ce qui justifie son lien de composition avec l'objet *Node*. Il assure la gestion des différentes opérations effectuées sur un graphe dont l'ajout et la suppression d'un sommet, l'extraction d'un sous graphe, réduction d'un graphe...etc.

- L'objet *BroadcastTree* représente un graphe en forme d'arbre. ce qui explique sa relation d'héritage avec *Graph*. Il représente l'arbre de diffusion de la FDS. Il implante à travers sa méthode *calculate()* l'algorithme de calcul de l'arbre de recouvrement à poids minimum.

- L'objet *BroadcastStar* représente un graphe en forme d'étoile. Il représente un arbre de diffusion en étoile. Il peut être aussi bien un arbre de diffusion locale qu'un arbre de diffusion.

- L'objet central de ce modèle est l'objet *GenericFederatedServer*. Il est au coeur du modèle d'administration. Il modélise le comportement d'un serveur fédéré et en assure les fonctions. Il est associé à l'objet *ServersFederation* qui représente sa vue de la fédération.

- L'objet *ServersFederation* contient toutes les structures de données qui décrivent le GSF, l'arbre de diffusion ainsi que l'arbre de diffusion locale et permet de les modifier. Son lien de composition avec les objets *Graph*, *BroadcastTree* et *StarBroadcast* fait que son cycle de vie est le leur. Nous voulons ainsi exprimer le fait que la vue que possède un serveur fédéré de la fédération à laquelle il appartient est définie par son GSF, son arbre de diffusion et son arbre de diffusion local.

- L'objet *ChangingVersionServer* modélise le comportement du serveur de changement de version. c'est lui qui se charge des procédures de changement de version.

- L'objet *FaultManagingServer* est prévu pour modéliser le serveur de détection de défaillance. La gestion des défaillances n'étant pas prévue dans cette étude, nous n'abordons pas cette question.

- L'objet *SpecificFederatedServer* représente l'application du *GenericFederatedServer* à un service bien précis. Cet objet peut être un trader, un serveur de cache Internet ou autre. Cela dépend de la nature de la fédération. Sa relation avec le serveur générique peut être soit un héritage soit une association. cette question est discutée dans la section 3.5.

Désormais la fédération de serveurs s'avère être une fédération d'objets qui interagissent pour permettre l'administration dynamique de la Fédération.

L'échange de messages entre serveurs se trouve être des invocations de méthodes locales ou distantes entre objets.

Les services (méthodes) de ces objets seront détaillés dans le dernier chapitre ainsi que l'architecture répartie de ce modèle.

3.3.2 Le modèle de communication

Le modèle de communication est inspiré du modèle OGS présenté au deuxième chapitre. Il décrit des objets de communication qui fournissent des primitives de diffusion de messages vers un groupe d'objets. Désormais nous parlons de "groupe d'objets".

Tout serveur fédéré a accès aux fonctionnalités de communication de groupe grâce à deux types d'objets: les objets *GroupAccessor* qui s'occupent de l'émission du message à diffuser et les objets *Invocable* qui s'occupent de la réception du message diffusé.

- *GroupAccessor*: un objet de ce type est le représentant local d'un groupe d'objets. Il intègre les mécanismes nécessaires à la diffusion de messages vers ce groupe avec une certaine sémantique.

- *Invocable*: il définit un objet capable de recevoir un message. Un objet de ce type est l'intercepteur d'un message diffusé. Il joue un rôle symétrique à celui du GroupAccessor. Il fait office de proxy. L'existence de cet objet permet d'effectuer certains traitements propres à certaines sémantiques de diffusion qui doivent précéder la réception du message par son destinataire final.

Nous avons évoqué au premier chapitre l'existence de deux sémantiques de diffusion que nous appelons: sémantique **fiable** et sémantique **relâchée**.

1. *Diffusion fiable* (reliable multicast): cette sémantique est utilisée afin d'effectuer les modifications locales de l'arbre de diffusion d'une manière cohérente. Elle permet de diffuser un message vers un groupe de serveurs en garantissant que le message soit délivré soit à tous les serveurs soit à aucun d'entre eux. Si l'un des serveurs n'est pas prêt à accepter le message, son traitement est annulé dans tous les autres serveurs.

2. *Diffusion relâchée* (best effort multicast): quant à cette sémantique, elle est utilisée pour la propagation de messages. Dans l'administration de la fédération, elle est utilisée pour diffuser le message de changement de version. Elle permet de diffuser un message sans l'obligation que tous les destinataires le reçoive. Ce sont en général des messages sans réponse.

La figure 3.2 montre le diagramme des classes du modèle de communication.

De l'interface *GroupAccessor* (classe non instantiable) hérite les objets *ReliableGroupAccessor* et *BestEffortGroupAccessor* qui représentent une spécialisation du rôle du *GroupAccessor* aux deux sémantiques, respectivement fiable et relâchée. Même raisonnement pour l'objet *ReliableInvocable* par rapport à son interface *Invocable*.

- L'objet *ReliableGroupAccessor* s'occupe de la diffusion de messages avec la sémantique fiable.

- L'objet *ReliableInvocable* se charge d'intercepter les messages diffusés avec la sémantique fiable. Le *ReliableGroupAccessor* possède un lien de référence vers un groupe d'objets de ce type, ce qui lui permet de leur diriger les messages.

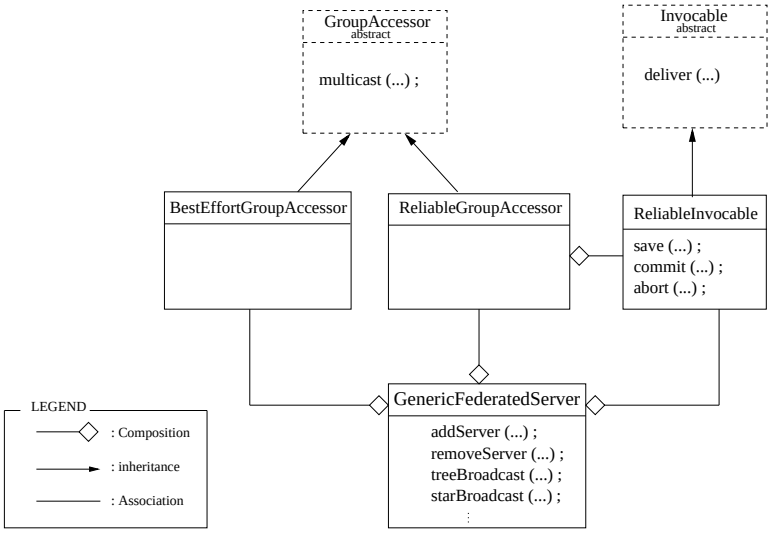

FIG. 3.2 – *Diagramme des classes du modèle de communication*

– L'objet *BestEffortGroupAccessor* se charge de la diffusion de messages avec la sémantique relâchée.

Tous ces objets sont créés lors de la création de l'objet *GenericFederatedServer*. Le lien de composition entre ce dernier et les trois objets de communication exprime cela.
A noter que nous n'avons pas besoin d'un objet *Invocable* pour la sémantique relâchée parce aucun traitement n'est requis avant la réception du message par son destinataire final. Il lui est délivré directement.
Nous détaillerons les mécanismes de diffusion associés aux deux sémantiques dans la section suivante.

3.3.3 Structure complète de l'OFDS

Le modèle d'administration et de communication interagissent fortement. Le schéma 3.3 montre la structure complète de l'outil de fédération dynamique des serveurs (OFDS).

La figure 3.3 présente deux packages *mDynamicFederation* qui regroupe les objets d'administration et *mGroupAccess* qui regroupe les objets de communication.

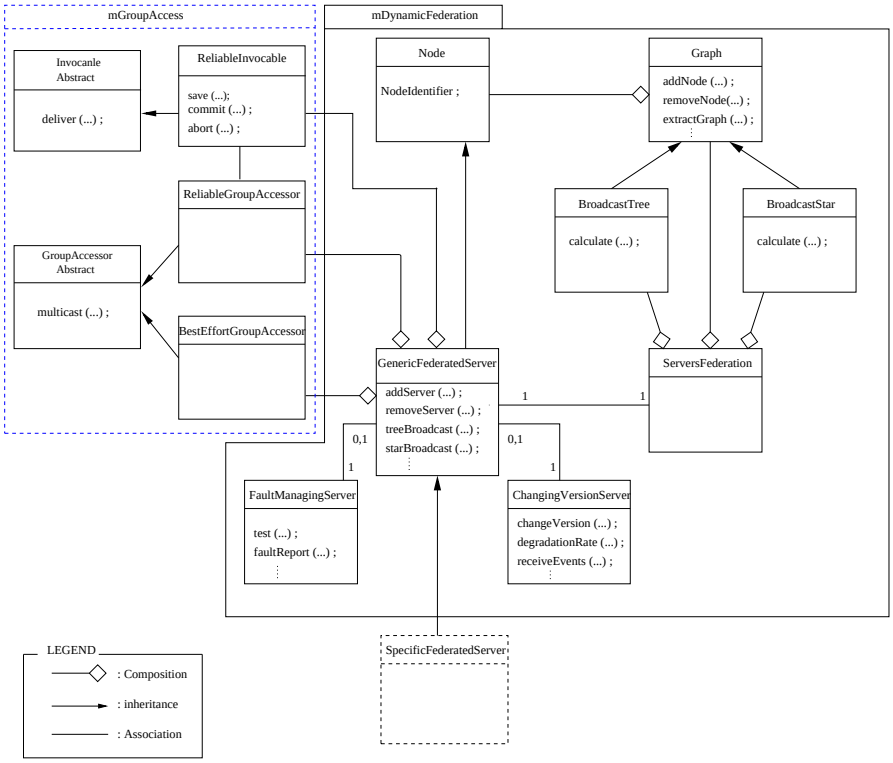

FIG. 3.3 – *Diagramme des classes de l'OFDS*

3.4 Gestion de la communication dans la FDS

La communication est une fonction vitale dans la FDS parce qu'elle est à la base de la coopération de ses serveurs. En fait la communication dans la FDS se traduit par une séquence de requêtes/réponses ou bien des requêtes sans réponses. Ces requêtes sont en réalité des invocations de méthodes locales ou distantes. Ces méthodes peuvent être avec ou sans valeurs de retour.

Il existe deux types de communication dans la FDS. Une communication point à point qui permet à deux serveurs d'échanger des informations et une communication multipoint qui permet de diffuser des informations vers un groupe de serveurs. Cette dernière soulève une problématique assez classique dans le domaine des systèmes répartis. Des questions telles que la terminaison de la diffusion, la gestion des réponses, la gestion du temps, la réception de messages dupliqués...etc y sont posées. Certes nous ne répondons pas à toutes ces question dans cette étude mais quelques unes d'entre elle y sont abordées.

3.4.1 La communication point à point dans la FDS

Bien entendu une paire de serveurs peuvent s'échanger des requêtes. Cet échange se traduit par de simples invocations CORBA. Ayant la référence du serveur cible, une requête peut être invoquée en synchrone ou en asynchrone, selon les besoins. Le bus CORBA se charge de faire parvenir le requête à son destinataire et faire de même pour la réponse.

3.4.2 La diffusion dans la FDS

L'OFDS offre le moyen de diffuser des informations vers un groupe de serveurs. Il permet d'invoquer une méthode sur un groupe de serveurs ayant la même interface. Ceci est fait suivant deux sémantiques :

La Diffusion fiable dans la FDS

La diffusion fiable permet d'invoquer une méthode qui doit être exécutée d'une manière cohérente. C'est le cas des méthodes de modifications des arbres de diffusion locaux des serveurs fédérés.

Invoquer une méthode sur un groupe de serveurs d'une manière fiable exige un mécanisme à deux phases. Dans un premier temps, une fois la requête diffusée aux membres du groupe, si elle peut être acceptée, elle est sauvegardée et un acquittement positif est envoyé à la source de diffusion. Si tous les membres du groupe envoient un acquittement positif disant qu'ils sont prêts à accepter la requête, un ordre d'exécution de la requête sauvegardée leur est émis. Dans le cas où au moins un serveur envoie un acquittement négatif ou ne répond pas, un ordre d'annulation de la requête est émis et le traitement est différé.

La figure 3.4 illustre le mécanisme d'invocation d'une méthode fiable telle que la méthode *addServer* sur un groupe de serveurs. Cette méthode consiste à rajouter un nouveau

serveur dans l'arbre de diffusion local d'un des serveurs et ses voisins.

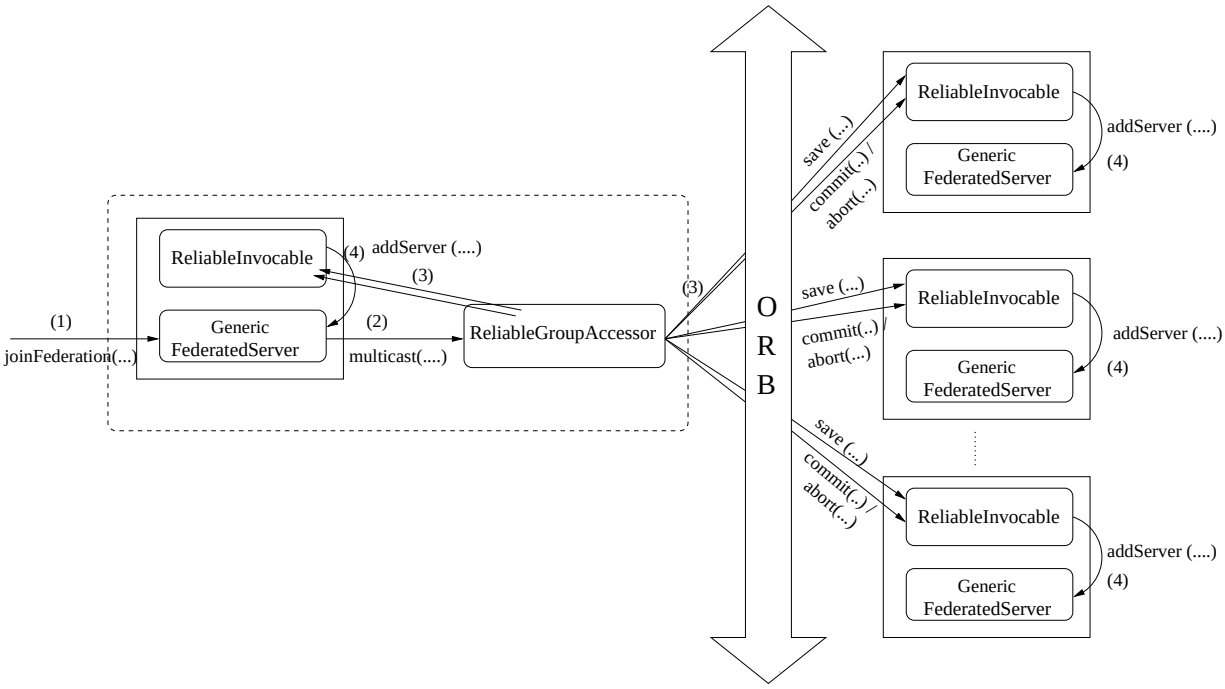

FIG. 3.4 – *Mécanisme de diffusion fiable*

Les différents objets intervenants dans ce mécanisme sont: le *GenericFederatedServer* qui, du côté gauche de la figure, représente le serveur de contact d'un nouveau serveur, et du côté droit, représente le groupe de serveurs à qui il doit notifier l'ajout d'un nouveau serveur. Ce sont en fait ses voisins de premier niveau dans l'arbre de diffusion local. Le *ReliableGroupAccessor* est utilisé par le serveur de contact pour s'occuper de la diffusion, et au niveau de chaque serveur cible, le *ReliableInvocable* qui s'occupe de la réception des requêtes diffusées. Le mécanisme est décrit comme suit :
(1) Un serveur voulant intégrer la fédération invoque la méthode *joinFederation* sur un des serveurs de la fédération (après une procédure de choix de ce serveur [Tac97]), (2) L'objet *ReliableGroupAccessor* se charge de diffuser la requête d'ajout d'un nouveau serveur (la méthode *addServer*) vers les objets *ReliableInvocable*. (3) Le protocole de diffusion fiable est alors déclenché. Dans un premier temps, la capacité d'accepter la requête est testée par *save* qui, dans le cas d'un succès du test, sauvegarde la requête et retourne un acquittement positif. Selon les acquittements reçus, la requête est soit validée (par la méthode *commit*), soit annulée (par la méthode *abort*). (4) Si la requête est validée, elle est finalement invoquée sur chaque serveur fédéré du groupe.

La Diffusion relâchée pour la propagation d'informations

La diffusion relâchée permet d'invoquer une méthode donnée sur un groupe de serveurs sans exiger qu'elle soit exécutée par tous les serveurs. Si l'un des serveurs, pour des raisons d'indisponibilité ou de défaillances, ne peut pas exécuter la requête demandée, sa propagation est tout de même enchaînée.

Cette sémantique est utilisée essentiellement pour la propagation d'informations dans l'arbre de diffusion, tel le cas d'un changement de version de la fédération. Une requête de changement de version est diffusée vers tous les serveurs de la fédération sans la nécessité que tous les serveurs la prennent en compte.

La figure 3.5 montre le mécanisme de diffusion d'une requête de changement de version (la méthode *changeVersion*). Le changement de version est déclenché par le serveur de changement de version (SCV). Cette figure montre la propagation de la requête par tout serveur l'ayant reçu soit de la part du SCV lui même, soit de la part du serveur père dans l'arbre de diffusion.

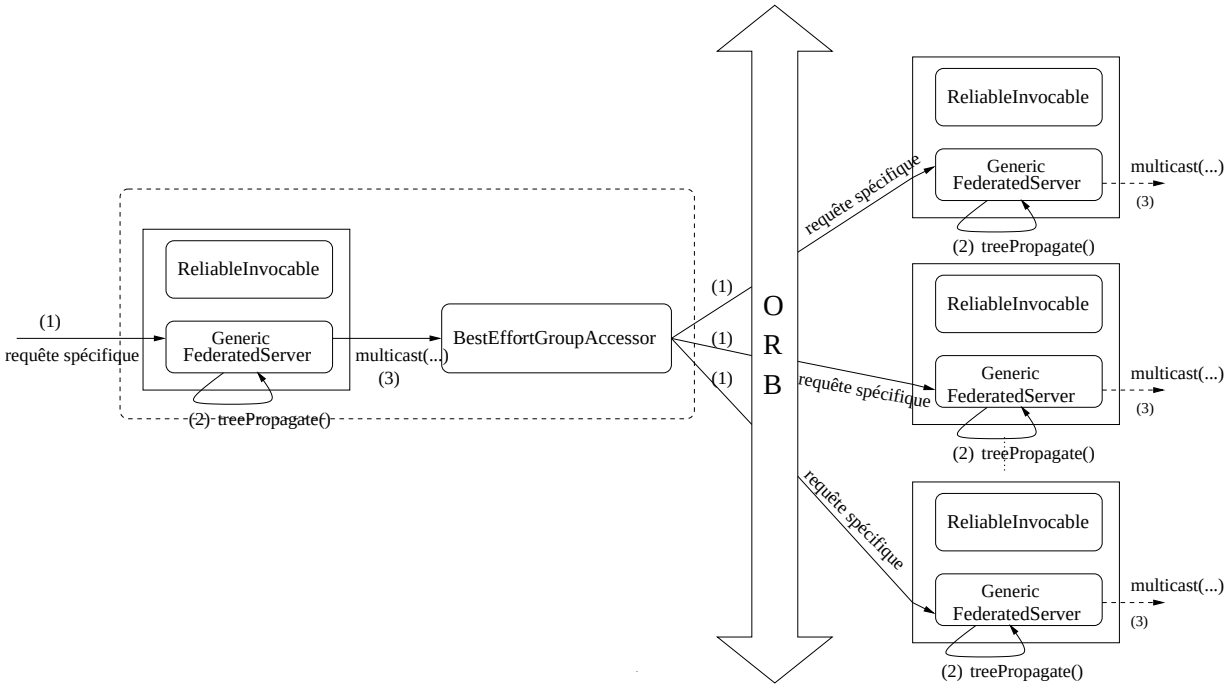

FIG. 3.5 – *Application de la diffusion relâchée à la propagation de requête*

(1) Le serveur fédéré reçoit la requête de changement de version. Il l'exécute et (2) la propage vers ses voisins dans l'arbre de diffusion (avec la méthode *treePropagate*). (3) l'objet *BestEffortGroupAccessor* se charge de la diffuser vers le groupe de serveurs voisins. Cette fois-ci, contrairement à la sémantique précédente, la requête est invoquée directement

sur l'objet serveur sans passer par un objet intermédiaire. Et le processus est réitéré jusqu'à arriver aux feuilles de l'arbre de diffusion.

Caractéristiques de la propagation

La propagation d'informations est un problème classique dans le domaine des systèmes répartis. Sa problématique reste complexe et difficile à résoudre. Nous en abordons quelques questions.

- **Gestion des réponses :**
  Telle une requête invoquée sur un objet simple, une requête invoquée sur un groupe d'objets peut avoir des valeurs de retours. Seulement dans ce cas, nous avons autant de valeurs de retours que d'objets serveurs dans le groupe. Ceci soulève beaucoup de questions dont: faut-il invoquer des requêtes synchrone ou asynchrone ? faut-il attendre toutes les réponses ?

  Dans le contexte d'un réseau étendu avec des délais de communication très variables et un risque d'échec de la communication assez présent, une requête synchrone paraît inenvisageable. Avec une solution asynchrone, un client peut retarder sa réception des réponses de sa requête et effectuer d'autres traitements entre temps. Aussi, une éventuelle défaillance d'un des serveurs ne va pas le bloquer. Dans notre cas, l'invocation d'une requête sur un groupe de serveurs est asynchrone et sans réponses. Il est important de distinguer entre les **acquittements** et les **réponses**. Les acquittements, pour une diffusion fiable ou non fiable, sont attendus pour confirmer ou infirmer la réception du message. Cette attente n'est pas bloquante et est limitée par un **timeout**. Concernant les réponses aux requêtes, elles sont traitées en tant que requêtes et non pas en tant que valeurs de retours à des requêtes. C'est à dire qu'un serveur qui veut répondre invoque une autre requête sur le serveur source de la propagation.

- **Gestion de la terminaison de la propagation**
  La diffusion de requêtes se fait suivant un arbre de diffusion qui est un arbre de recouvrement de poids minimum. Bien entendu, cette structure assure la terminaison de la diffusion puisque la requête est propagée d'un noeud source de l'arbre vers ses feuilles sans duplication. Donc par construction, la terminaison de la diffusion est assurée. Ceci dit, nous avons néanmoins prévu un moyen d'arrêter la propagation d'une requête avant d'atteindre les feuilles de l'arbre. Ceci vise à limiter la propagation à un sous ensemble des serveurs de la fédération. Deux techniques sont prévues à cet effet :(1) fixer explicitement la portée de la propagation en terme de profondeur, (2) spécifier une condition d'arrêt de la propagation qui sera testée à chaque noeud de l'arbre.

3.5 Application d'une FDS

Plusieurs applications distribuées peuvent bénéficier des services de la FDS. Le système de news *USENET* [BK86] utilise une fédération de serveur pour la distribution des news. Le service de trading de CORBA [OMG96] offre la possibilité de lier des traders entre eux. On peut alors parler de fédération de traders qui permettra une recherche optimisée des offres de services.

L'OFDS offre le moyen d'appliquer la FDS à tout type de fédération comme celles citées ci-dessus. Ces fédérations hériteront de la gestion dynamique de la FDS tout en rendant le service qu'elles rendent à leurs clients.

L'application de la FDS se traduit par la spécialisation du rôle de son objet principal à savoir le serveur fédéré générique autrement dit l'objet *GenericFederatedServer*. Techniquement, Cette spécialisation peut se faire de deux manières :

1. L'héritage
   Cette notion est celle utilisée dans la programmation orientée objet. Le serveur spécifique peut hériter de l'objet *GenericFederatedServer* et donc de toutes ses fonctions tout en gardant ses propres services. le serveur générique et le serveur spécifique se trouvent confondus en un seul objet.

2. L'association
   Quant à l'association, elle consiste à associer les deux objets. Chacun d'eux possède un lien de référence vers l'autre. le *GenericFederatedServer* et le *SpecificFederatedServer* sont deux objets à part.

L'intérêt de **l'héritage** est dans le fait que nous ayons un seul objet actif. Il s'occupe et de l'administration de la fédération et du service spécifique qu'il rend à ces clients. Ce qui est intéressant sur le plan de l'espace mémoire occupé. Cependant, cette technique exige que le serveur spécifique hérite du serveur générique. Ceci implique, dans certains environnements qui ne tolèrent pas l'héritage multiple, que le serveur spécifique ne puisse pas hériter d'un autre objet. Ce qui peut être limitant. Cette limitation n'existe pas avec **l'association**. Avec cette technique, il y a deux objets actifs en mémoire. Les rôles sont bien séparés. Le serveur générique s'occupe de l'administration, et le serveur spécifique s'occupe de rendre ses services aux clients tout en utilisant les services de communication du premier.

3.5.1 Une fédération de traders CORBA

Le but de cette section est d'appréhender les modifications nécessaires pour qu'une fédération spécifique utilise les services de la FDS.
Prenons le cas d'une fédération de traders CORBA. Le service de trading définit une hiérarchie de traders suivant les interfaces supportées. Prenons l'exemple d'un *Query Trader* qui supporte l'interface *Lookup*. Cette dernière contient une seule méthode *query* qui permet de rechercher un service particulier.

Pour qu'un *query Trader* utilise les services de la FDS, son interface doit être enrichie. Nous définissons pour cela l'interface **ExtndedLookup**:

```
module FederatedQuery_Trader
{
interface ExtendedLookup : CosTrading::Lookup,[mDynamicFederation::GenericFederatedServer]
{

//void query(in ServiceTypeName type, in Constraint constr, ..., out OfferSeq offers,
  //...) raises (...);

void queryToPropagate (in mDynamicFederation::RequestHeader head, in ServiceTypeName type,
 in Constraint constr, ...);

void queryAnswers (out OffSeq offers);

};
}
```

Cette interface hérite la méthode *query* de l'interface *Lookup* du service de Trading de l'OMG, et définit la méthode *queryToPropagate*. Notons la différence de signature de ces deux méthodes. En fait, *queryToPropagate* permet de propager la méthode *query* en spécifiant les caractéristiques de cette propagation dans son premier argument. *RequestHeader* est une structure qui regroupe toutes les caractéristiques de la propagation qu'on veut effectuer telle que la portée de la propagation, la source de la propagation...etc. Le *RequestHeader* est détaillé dans la section 5.3.2.
Le client d'un trader fédéré n'utilise que l'interface *Lookup* standard. Un *query* sur un trader générera un *queryToPropagate* sur ses voisins dans l'arbre de la fédération. Les traders, proposant des offres, répondront directement avec *queryAnswers*.

3.6 Conclusion

Nous avons présenté les différents éléments de l'OFDS (Outil générique pour la fédération dynamique des serveurs). C'est un outil qui permet de gérer dynamiquement une fédération de serveurs. Il est basé sur une hiérarchie d'objets définis à partir du modèle GSF décrit au premier chapitre. Cette hiérarchie comprend deux types d'objets, des objets d'administration qui permettent d'accomplir les différentes opérations d'administration de la fédération; ainsi que des objets de communication qui permettent aux serveurs de communiquer et de diffuser des informations selon deux sémantiques (fiable et relâchée). Nous avons abordé certaines questions en relation avec la diffusion dont la gestion des réponses et la terminaison de la diffusion. Rappelons que notre principal objectif n'est pas de développer un outil de communication de groupe. Devant l'absence d'outils standards, nous avons conçu un module simple de communication de groupe qui nous permette de diffuser nos messages dans la FDS. En dernier lieu, nous avons évoqué la procédure d'application de la FDS. Nous verrons au chapitre suivant une FDS spécifique. C'est dans le cadre d'un

projet entre l'INT et l'université Charles de Prague : Le projet BARRANDE. Il s'agit d'une fédération de serveurs de composants logiciels.

Chapitre 4

Etude de cas : le projet BARRANDE

Le projet BARRANDE est un programme d'échange entre deux équipes de recherche: l'équipe *Système répartis* du Département INF de l'INT et une équipe du Département Informatique de l'Université Charles de Prague (République Tchèque).

Le département Informatique de l'Université Charles travaille sur les projets **SOFA** (SOFtware Appliances) et **DCUP** (Dynamic Component UPdating) [FP98]. Ces projets proposent une architecture permettant la définition et le développement de composants logiciels. Dans cette architecture, les composants logiciels sont téléchargés et mis à jour dynamiquement. Le département INFormatique de l'INT travaille sur les techniques permettant la distribution de logiciels sur grande échelle : technologie Web, partage de charge, FDS (Fédération Dynamique de serveurs), agents mobiles, service de trading.

Parmi les actions de ce projet, l'utilisation de la FDS dans l'architecture SOFA. Nous avons effectué, dans le cadre de cette collaboration, un voyage à l'Université Charles durant lequel nous avons étudié la possibilité d'intégrer la FDS au SOFA. Ce voyage a permis de développer la procédure de spécialisation de la FDS, et de découvrir une autre approche pour la représentation de la FDS, c'est une approche orientée composant. Vu l'enrichissement qu'a apporté cet échange, nous avons préféré consacrer un chapitre entier à ce projet.

Nous présentons dans ce chapitre une synthèse du travail effectué lors de ce voyage. Nous commençons par introduire la notion de composant logiciel afin de fixer les idées. Nous décrivons par la suite l'architecture SOFA, ses domaines d'application ainsi que son besoin d'une structure telle que la FDS. Et nous proposons brièvement un modèle orienté composant pour la FDS (c'est un modèle qui reste à étudier).

4.1 Les composants logiciels

Le concept d'objet a trouvé son succès pour les raisons que nous connaissons. Cependant, aujourd'hui, avec la tendance de construire des applications à partir de bouts de logiciels hétérogènes [VM96] (librairies, frameworks, hiérarchies d'objets...) conçus pour différentes applications, systèmes et langages, on a besoin d'une entité qui puisse intégrer

ces bouts logiciels et favoriser la réutilisation tout en offrant l'homogénéité d'une composante logicielle unique. De ceci, découle le concept de **composant logiciel**. Les avantages d'une conception basée composant [VM96] sont les suivants :

- Réutilisabilité et intégration de composants logiciels existants.
- Applications configurables.
- facilité de conception, développement et maintenance des applications.

4.2 Introduction de l'architecture SOFA (SOFtware Appliances)

L'architecture SOFA s'inscrit dans le cadre des environnements distribués basés composants. Cette architecture vient de l'idée que, dans l'avenir, le développement d'applications sera basé sur le réutilisation de composants logiciels existants qu'on pourra récupérer directement à travers un réseau, ce qui donnera lieu à un vrai commerce de composants logiciels. En plus de tous les aspects de cette architecture en relation avec le commerce électronique, elle traite la nécessité d'adapter les composants et leurs interconnexions selon les spécificités de l'application hôte. Il y a une autre question que l'architecture SOFA aborde, c'est l'idée de mettre à jours dynamiquement les composants logiciels d'une application , quasiment sans intervention humaine, pour remplacer par exemple une version boguée ou simplement mettre à jour une ancienne version. Cette question est encore plus complexe quand il s'agit de toucher à des composants d'une application en cours d'exécution.

L'environnement SOFA touche à plusieurs aspects [FP98] dont :

- Le commerce électronique. Cela concerne l'achat des composants logiciels, leur payement, la gestion des licences, la sécurité...etc.
- Un protocole de transmission de composants logiciels via un réseau. Parmi les éléments de ce protocoles, il y a l'utilisation de modèles transactionnels pour la transmission des composants.
- La mise à jour dynamique des composants logiciels avec une architecture spécifiques de composants appelées *DCUP (Dynamic Component UPdating)* et qui est une extension des composants SOFA. L'architecture DCUP présente une technique de mise à jour des composants d'une application en cours d'exécution. Cette technique repose sur le concept d'agents mobiles et l'approche *Aglet* [Lan97].
- Un formalisme de définition de l'architecture des composants logiciels ainsi que leurs comportements. La spécification d'un composant SOFA est exprimée dans un langage de définition de composant *SOFA CDL (Component Description Language)* basé sur la syntaxe IDL de CORBA. La syntaxe complète de ce langage est détaillée dans [Men98].

– Le trading des composants logiciels qui permet de rechercher un composant logiciel à partir d'un ensemble de caractéristiques.

Ce n'est pas du ressors de cette étude de détailler tous ces aspects. Nous nous intéressons à la structure sur laquelle s'appuie l'environnement SOFA à savoir *une fédération de serveurs de composants logiciels.*

4.3 Structure du SOFA

Dans SOFA, une application est vue comme une hiérarchie de composants logiciels téléchargeables et qui peuvent être mis à jour dynamiquement par leur fournisseurs à travers un réseau. Ce dernier représente la structure de base du SOFA, c'est la structure **SOFAnet**. Le *SOFAnet* est un réseau étendu de type réseau d'entreprise. Il est composé de noeuds dont le terme général est **SOFAnode**. Les *SOFAnodes* jouent plusieurs rôles dont celui de fournisseurs de composants et de clients. Il y a un autre type de *SOFAnode* qui est le **SOFAproxy**. Ce dernier est le représentant des fournisseurs auprès d'un ensemble de clients. Les *SOFAproxy* sont reliés entre eux grâce à la FDS (Fédération Dynamique de Serveurs). Un *SOFAproxy* participe au téléchargement et la mise à jour des composants et joue le rôle d'un cache, on y stocke les dernières versions des composants.

La figure 4.1 montre la structure générale d'un SOFAnet.

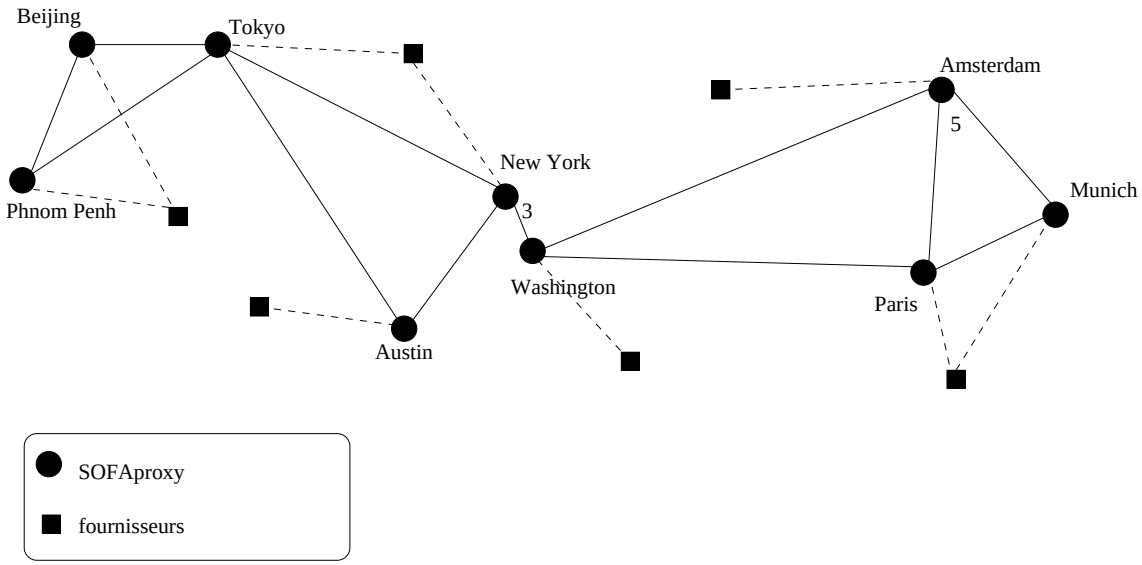

FIG. 4.1 – *Structure générale du SOFAnet*

Les *SOFAproxy* sont des serveurs qui coopèrent entre eux afin de fournir aux clients un service de vente et d'après vente de composants logiciels. Cette coopération permet essentiellement le trading des composants. Elle permet de rechercher par exemple la localisation de la dernière version d'un composant logiciel en diffusant un avis de recherche de celui-ci.

4.4 Une fédération de *SOFAproxy*

Une fédération de *SOFAproxy* est un groupe de *SOFAproxy* déployés dans le réseau d'un organisme décentralisé. Ils sont en relation, à la fois avec les clients et les fournisseurs de composants logiciels. Grâce à cette fédération, la recherche d'un composant peut être diffusée vers l'ensemble des serveurs, une nouvelle version de composant peut leur être notifiée, l'invalidation des composants peut être effectuée...etc. Les *SOFAproxy* ont besoin d'un support de coopération. L'OFDS leur offre ce support. Il suffit de spécialiser le serveur fédéré générique.

Nous présentons, dans ce qui suit, la partie de l'interface d'un *SOFAproxy* qui utilise la propagation:

```
interface SOFAproxy: mDynamicFederation::GenericFederatedServer
{

void findComponent(in TemplateM tm); // Rechercher un composant dans la fédération.
SOFAproxyRef[] retrieveFromFindComponent(); // Récupérer les résultats
 de la recherche.
void notifyNewComponent (in NotificationMsg msg); // Notifier la présence d'une
 nouvelle version.
void invalidateComponent (in InvalidateMsg msg); // Il y a un composant bogué.
void findComponentToPropagate(in RequestHeader head,in TemplateM tm); // propager
 la recherche d'un composant.
 // propager la notification.
void notifyNewComponentToPropagate (in RequestHeader head, in NotificationMsg msg);
....

}
```

Chacune de ces requêtes utilisent les primitives de diffusion de l'OFDS.

4.5 Une approche basée composants pour la modélisation de la FDS

Il nous paraît intéressant d'exposer une autre approche pour modéliser la FDS. C'est une approche orientée composant. En réalité, cette approche ne change en rien le modèle

orienté objet présenté au chapitre précédent. Globalement, elle en change la structuration en introduisant un autre niveau d'abstraction qui est celui du composant et qui va encapsuler les objets déjà définis dans le modèle.

La figure 4.2 montre la nouvelle structure de la FDS.

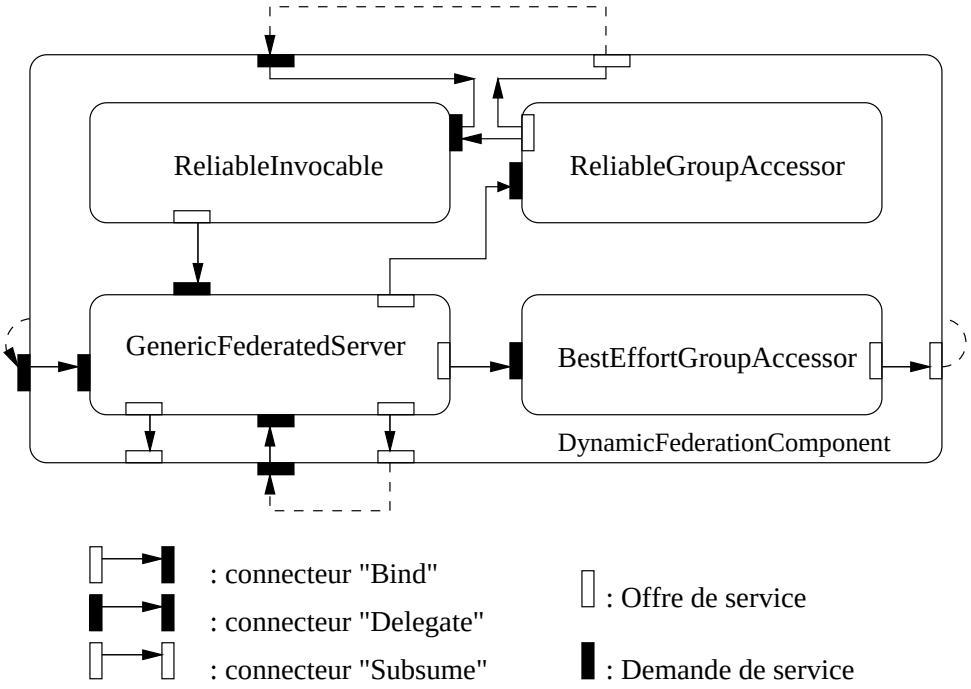

FIG. 4.2 – *Un composant FDS*

Tout composant peut offrir un service, ce qui est exprimé par les rectangles noirs, et peut demander un service, ce qui est exprimé par les rectangles blancs. Les composants sont liés à l'aide de trois types de connecteurs :

1. Les connecteurs *"Bind"* lient une offre de service à sa demande.
2. Les connecteurs *"Delegate"* lient une offre de service d'un composant "composé" à celle de l'un de ses composants "composant" (un sous composant).
3. Les connecteurs *"Subsume"* lient une demande de service d'un composant à celle de son composé.

Suivant ce formalisme [FP99], la figure 4.2 exprime notre modèle. Elle représente l'architecture d'un composant que nous appelons "*DynamicFederationComponent*", et qui constitue véritablement un serveur fédéré. Il regroupe les principaux objets qui participent à

l'accomplissement du rôle du serveur fédéré, ainsi que les différentes relations qui existent entre eux exprimées à l'aide des trois types de connecteurs. Les liens en pointillés sont des liens externes qui lient deux composants *DynamicFederationComponent* distants.

Ce formalisme décrit seulement la structure des composants. Nous n'allons pas détailler cette approche plus que ça. Cette section fait office d'une introduction d'une bonne perspective de continuation.

4.6 Conclusion

Nous avons présenté au cours de ce chapitre le projet BARRANDE qui était à la fois une application du modèle de la FDS et une découverte d'un formalisme de modélisation orienté composant.

L'environnement SOFA est un ambitieux projet qui demande encore beaucoup de travail. Son alliance avec la fédération Dynamique de Serveurs constitue un enrichissement des deux parties.

Nous avons présenté les différents aspects de l'architecture SOFA. Cette architecture repose sur une fédération de SOFAproxy. De ce fait, elle utilise l'OFDS pour l'administration de son organisation ainsi que ses services de communication.

Nous avons également proposé brièvement une autre approche de modélisation de la FDS. C'est une approche basée composant, décrite dans [FP99]. [FP99] propose une technique de spécification de l'architecture et du comportement des composants logiciels. Nous avons appliquer cette technique à notre modèle. Cette démarche, pour le moment, a juste amélioré la structuration du modèle et la lisibilité de son protocole. Une étude plus approfondie est à prévoir.

Chapitre 5

Mise en oeuvre de l'OFDS

Dans ce chapitre, nous nous intéressons à la mise en oeuvre de l'OFDS. Nous présentons l'environnement de programmation qui a servi au développement ainsi que les différents outils et concepts dont on eu besoin. De plus, nous exposons la spécification des interfaces IDL de notre outil et quelques aspects de son implantation.

5.1 L'environnement de programmation

Nous utilisons l'environnement le bus CORBA **ORBacus** avec le langage **Java**.

5.1.1 ORBacus

ORBacus [ORB], plus connu sous le nom de *OmniBroker*, est un ORB (Object Request Broker) dédié à une utilisation non commerciale (gratuit), développé par *Object Oriented Concepts (OOC)*[OOC]. Il est compatible à cent pour cent avec la spécification CORBA 2.0, et implante le mapping CORBA IDL/JAVA et IDL/C++.
Nous avons choisi cet ORB, tout d'abord, parce qu'il est gratuit, et qu'il intègre un certain nombre de fonctionnalités, intéressantes pour nous, dont :

- Les mécanismes dynamiques de CORBA, à savoir le *Dynamic Invocation Interface (DII)* et le *Dynamic Skeleton Interface (DSI)* que nous détaillerons par la suite.
- Différents modèles de concurrence.
- Des services de base de CORBA tels que le service de nommage, le service des événements, le service des propriétés, le service de notification. Nous en utilisons le service de nommage.

L'intérêt de cet ORB est aussi dans la disponibilité de ses sources et la qualité se son support technique. Il a été adopté par différentes compagnies dans différents secteurs dont DEUTSCHE TELEKOM qui l'utilise dans ses projets d'orientation vers les environnements middlewares.

5.1.2 Java

Outre le fait que le langage Java [Fla97] est un langage orienté objet, ce qui est une propriété nécessaire à l'implantation de l'OFDS, il présente d'autres avantages dont :

- C'est un langage concurrent. nous verrons l'intérêt de cette propriété dans la section suivante.
- Il est conçu pour un maximum de portabilité. Grâce au format *byte-code* généré par le compilateur Java et qui est indépendant de la plate forme, les programmes Java tournent sur n'importe quel système implantant la machine virtuelle Java.

5.2 Outils de mise en oeuvre

Afin de mettre en oeuvre l'OFDS, nous avons besoin d'outils qui nous permettent les actions suivantes :

- Invoquer une requête asynchrone. c'est à dire une requête non bloquante.
- Invoquer plusieurs requêtes en parallèle pour optimiser la diffusion de messages.
- Répondre aux demandes de service de plusieurs clients en même temps. autrement dit, gérer la concurrence au niveau des serveurs de la fédération.
- Gérer la généricité des serveurs fédérés. C'est à dire que l'on doit s'attendre à ce que des requêtes inconnues aux serveurs vont circuler dans la fédération. Ce sont les requêtes des serveurs spécifiques qui vont utiliser l'OFDS et dont on ne sait rien du tout. Il doit y avoir un mécanisme qui permet aux serveurs fédérés génériques d'accepter des requêtes qu'ils ne connaissent pas.

Pour répondre à ces besoins, nous faisons appel à deux mécanismes de CORBA qui sont le *"DII"* et le *"DSI"* [AV98, Bak97] ainsi qu'au *multithreading* [AV98].

5.2.1 Dynamic Invocation Interface (DII)

Le mécanisme DII permet à une application cliente d'invoquer dynamiquement des requêtes sans utiliser les souches IDL prégénérées. Le client découvre l'opération en cours d'exécution. Il n'a pas besoin de connaître cette opération au moment de la compilation. Les nouveaux services offerts par les serveurs se trouvent immédiatement utilisables par les applications clientes sans nécessiter leur modification.
Ce mécanisme nous permettra de réaliser deux choses :

1. Invoquer des requêtes asynchrones. Car les invocations CORBA sont, par défaut, synchrones. Le DII permet de construire une requête de toute pièce, en spécifiant le nom de l'opération, le type du résultat et les paramètres, et de l'invoquer d'une manière asynchrone grâce à la primitive *send_deferred()*. La fin de l'invocation peut être testée par la primitive *poll_response()*, et les résultats obtenus par *get_response()*.

2. Invoquer plusieurs requêtes en même temps. Cette fonctionnalité découle de la première. Toujours en asynchrone, le DII permet d'invoquer plusieurs requêtes d'un coup. Ceci grâce à la primitive *send_multiple_requests_deferred()* qui accepte en argument la séquence de requêtes à invoquer. Ce qui va nous permettre de diffuser des messages dans la FDS.

3. Invoquer des requêtes sur des serveurs d'interfaces inconnues.

5.2.2 Dynamic Skeleton Interface (DSI)

Le DII permet à un client d'invoquer une requête inconnue au moment de la compilation. De manière analogue, le DSI permet à un serveur d'accepter une requête inconnue au moment de la compilation. On peut invoquer n'importe quelle opération sur n'importe quel objet. A l'origine, ce mécanisme fut défini pour permettre la mise en place de passerelles (bridge) entre différents bus CORBA. Lorsque deux bus CORBA n'utilisent pas le même format de requêtes. Les conversions de protocoles sont effectuées par l'intermédiaire de passerelles: Les requêtes arrivent dans un certain format, elles sont reçues via le mécanisme DSI et converties. N'importe quelle requête peut être ainsi convertie.
Dans notre cas, le DSI permet aux serveurs fédérés d'accepter les requêtes des serveurs spécifiques quelle que soit leur nature.

5.2.3 Multithreading

Un *thread* est un système d'exécution séparé dans un même programme. Le *multithreading* est le lancement de plusieurs threads en même temps. Conceptuellement, les threads ressemblent à des processus, à la différence que plusieurs threads partagent le même espace d'adressage (ils peuvent partager des variables et des méthodes) et sont moins coûteux en terme de ressources. Le *multithreading* offre le moyen d'effectuer plusieurs tâches différentes en même temps.
Un serveur offrirait un meilleur service à ses clients s'il arrivait à traiter plusieurs requêtes en parallèle. Dans un contexte tel que celui de la FDS, un contexte dans lequel:

- les serveurs doivent à la fois répondre à leurs propres clients et aux autres serveurs de la fédération.

- certaines opérations peuvent durer un certain temps avant d'être achevée. c'est le cas d'une opération qui demanderait la coopération d'un certain nombre de serveurs, qui, rappelons-le, sont déployés dans un réseau étendu. le temps de latence n'est donc pas négligeable.

Il est clair que le multithreading s'impose de lui même dans ce contexte.

5.3 Les spécifications IDL de l'OFDS

L'ensemble des interfaces IDL de l'OFDS sont présentées, au complet, en annexe. Nous en exposons, dans cette section, les plus importantes.
Les interfaces de l'OFDS sont définies dans deux modules. Le premier est *"mDynamicFederation"* qui regroupe les objets d'administration, le deuxième est *"mGroupAccess"* qui regroupe les objets de communication de groupe.

5.3.1 Le module *mGroupAccess*

Ce module contient les deux interfaces *GroupAccessor* et *Invocable* qui s'occupent respectivement de la diffusion et la réception d'une requête. Un *GroupAccessor* diffuse une requête avec la méthode *multicast* qui accepte en argument la requête à diffuser, et la description du groupe d'objets serveurs auxquels la requête est destinée. Cette description comprend les références des objets du groupe. Un GroupAccessor, quand il ne sert plus à rien, est détruit avec la méthode *destroy()*. Cette interface va servir à implanter les deux sémantiques de diffusion (fiable et relâchée) à travers les deux objets *ReliableGroupAccessor* et *BestEffortGroupAccessor*.
L'interface *Invocable* définit un objet capable de recevoir un message à travers sa méthode *receive*. Tout *Invocable* est associé à un objet serveur dès sa création.
L'interface *ReliableInvocable* est une spécialisation de l'*Invocable* à la sémantique "fiable". La méthode *save* permet de sauvegarder une requête qui, avec son identificateur, sont reçus en argument. La méthode *commit* exécute la requête sauvegardée. La méthode *abort* l'annule.

```
module mGroupAccess
{
...

interface GroupAccessor
    {
BoolSeq multicast ( in any Req, in GroupDescription group); //diffuse une requête vers un groupe de serveurs
void destroy ();
    };

interface Invocable
    {
Any deliver (in any Req);
}

interface ReliableInvocable : Invocable
{
boolean save(in any req, in short req_idf); // sauvegarde la requête reçue en mémoire stable.
void abort(in short req_idf); // supprime la requête
any commit(in short req_idf); // exécute la requête
    };
...
};
```

5.3.2 Le module *mDynamicFederation*

Ce module contient les objets d'administration, nous en citons le plus important, représenté par l'interface *GenericFederatedServer*.

```
module mDynamicFederation
```

```
{
....
struct RequestHeader
{
RequestIdentifier idf;
FederatedServerIdentifier source;
FederatedServerIdentifier fromServer;
FederationState state;
short broadcastDepth;
short currentDepth;
mGroupAccess::RequestDescription stopping_condition;
mGroupAccess::RequestDescription delivery_condition;
Flag FaultReport;
}

interface  GenericFederatedServer
     {
attribute FederatedServerDesc FDesc;

FederationDescription getFederation();

void joinFederation (in FederatedServerDesc f, in DistanceSeq d)
raises(UnknownFederation, NewerVersion);

void addServer (in RequestHeader h , in AddServerEvent e)
raises (NewerVersion, OlderVersion, NotNeighbour);

void removeServer (in RequestHeader h, in RemoveServerEvent e)
raises (UnknownServer,NewerVersion, OlderVersion, NotNeighbour);

void changeVersion(in RequestHeader h, in FederationDescription newVersion);

void updateVersion(in FederationDescription newVersion);

void treeBroadcast (in RequestHeader h, in mGroupAccess::RequestDescription req)
raises (NewerVersion, OlderVersion, NotNeighbour);

void starBroadcast (in RequestHeader h, in mGroupAccess::RequestDescription req, in mGroupAccess::GroupDescription gr )
raises (NewerVersion, OlderVersion, NotNeighbour);

void treePropagate (in RequestHeader h, in mGroupAccess::RequestDescription req)
raises (NewerVersion, OlderVersion, NotNeighbour);

void propagate (in RequestHeader h, in mGroupAccess::RequestDescription req, in mGroupAccess::GroupDescription gr)
raises (NewerVersion, OlderVersion, NotNeighbour);

boolean  is_alive(); // tester la vivacité d'un serveur

EventList getEvents(out DistanceSeq localWeight);  // envoyer les événements au serveur de changement de version

void CVSAgreement();  // procédure d'accord sur le choix du SCV (Serveur de Changement de Version)

void oneway you_are_CVS ();  //procédure d'élection d'un SCV
    };
...

};
```

Cette interface définit toutes les opérations d'administration de la fédération.

- *getFederation* permet de demander une description de la fédération à savoir son identificateur, sa version, ses serveurs en plus des autres informations.
- *joinFederation* permet à un nouveau serveur de joindre la fédération.

- *addServer* et *removeServer* permettent respectivement l'ajout et la suppression d'un serveur dans un arbre de diffusion local.

- *changeVersion* permet le changement de version de la fédération.

- *updateVersion* permet à un serveur qui n'a pas reçu le message de changement de version de mettre à jour le numéro de version de la fédération. La différence entre ces deux dernières méthodes est que la première est une requête diffusée et à diffuser, la deuxième est une requête point à point.

- *treeBroadcast* déclenche une diffusion dans l'arbre de diffusion. Cette méthode permet à un serveur source de diffuser une requête vers tous ses voisins.

- *starBroadcast* déclenche une diffusion en étoile vers le groupe passé en argument.

- *treePropagate* permet à un serveur de propager la requête reçue vers tous ses voisins excepté celui qui l'a envoyé.

- *propagate* diffuse simplement une requête vers un groupe de serveurs spécifiés en argument.

Toutes les méthodes qui demandent une propagation admettent un argument de type *RequestHeader*. Ce dernier est une structure qui regroupe toutes les caractéristiques de la diffusion. Elle contient les informations suivantes :

- *source* : l'identificateur du serveur source de la diffusion.

- *fromServer* : l'identificateur du serveur ayant envoyé la requête, ceci permet de ne pas lui retransmettre la requête lors de sa propagation.

- *FederationState* : c'est une structure qui contient entre autre La version de la fédération.

- *CurrentDepth* : la profondeur courante de la diffusion.

- *BroadcastDepth* : la profondeur maximum de la diffusion.

- *stopping_condition* : une opération de test d'arrêt de la diffusion.

- *delivery_condition* : une opération d'identification du récepteur de la requête. Cette information permet de ne délivrer la requête qu'à des serveurs répondant à certains critères.

- *faultReport* : un drapeau qui, mis à 1, indique que le serveur source de la diffusion désire recevoir une notification dans le cas de la non réception du message par l'un des serveurs.

5.4 Quelques aspects de l'implantation de l'OFDS

5.4.1 Les classes Java

La figure 5.1 montre la projection de la spécification IDL en classes Java après compilation.

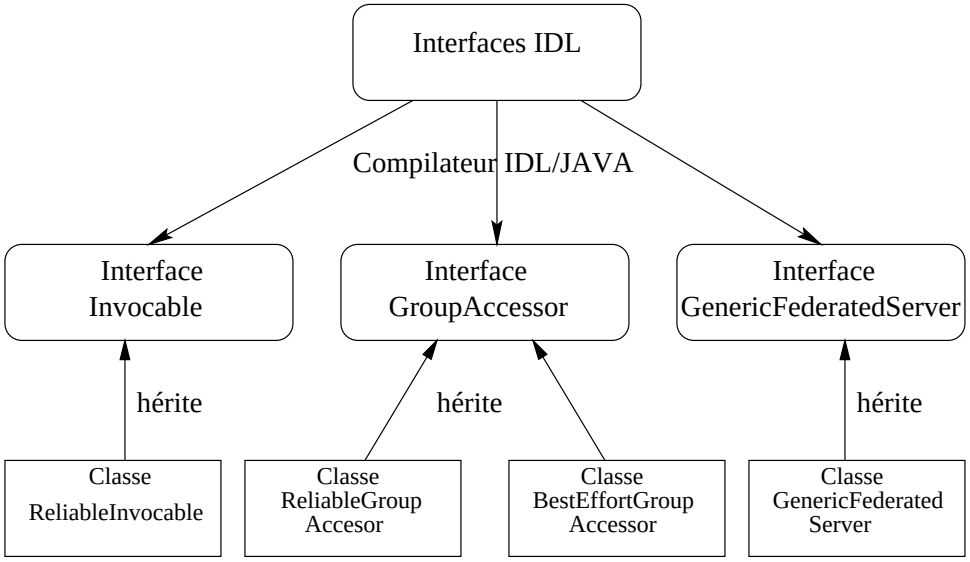

FIG. 5.1 – *Les classes*

Toutes les interfaces IDL sont mappées en interfaces Java qui sont ensuite implantées en en dérivant les classes associées. Toutes ces classes sont des objets CORBA.

– La classe *ReliableGroupAccessor*
Cette classe, tel qu'on l'a explique ci-dessus, permet de diffuser une requête avec la sémantique fiable, à travers la méthode *multicast*. Cette méthode utilise le DII. Cette méthode invoque la méthode *deliver* sur le groupe des *ReliableInvocable*, analyse les valeurs de retours, si tous les objets retournent "vrai", elle invoque *commit*, sinon elle invoque *abort*. Nous exposons l'utilisation du DII dans une partie du code de la méthode *multicast*.

```
public class  ReliableGroupAccessor extends _GroupAccessorImplBase
{
...

 public boolean[] multicast(any Req, GroupDescription gr)
{

Request r[] = new Request[gr.Members.length];
org.omg.CORBA.Object GrObj[] = new Object[gr.Members.length];
// récupérer les références des objets du groupe
```

```
for(int i; i<gr.Members.length, i++)
GrObj[i]= orb.string_to_object(gr.Members[i].IOR);

// préparation des requêtes ''deliver'' à diffuser avec le DII
for(int i; i<gr.Members.length, i++)
{
// le nom de l'opération
r[i]= GrObj[i]._request(''deliver'');
// les paramètres
Any AnyReq = r[i].add_in_arg();
AnyReq.insert_any(Req);

Any AnyIdf = r[i].add_in_arg();
//insérer req_idf qui est un identificateur calculé de la requête
AnyIdf.insert_short(req_idf);

r[i].set_return_type(orb.get_primitive_tc(TCKind.tk_boolean));
...
}

// invocation des requêtes
 orb.send_multiple_requests_deferred (r);

while (_orb.poll_next_response()==false) {}
Request reply=_orb.get_next_response();

... récupérer toutes les valeurs de retours dans return_deliver[]...
... si toutes les valeurs de retours sont ''true'', préparer, de la même manière que  deliver ci-dessous,
les requêtes  commit, sinon préparer les requêtes  abort et les invoquer.
...

return return_deliver;
}
...
}
```

– La classe *BestEffortGroupAccessor*
Cette classe implante la même interface que la précédente. Toujours à travers *multicast*, elle permet de diffuser une requête à un groupe d'objets avec la sémantique relâchée. Cette fois-ci, la requête est directement invoquée sur les objets *GenericFederatedServer* sans passer par des objets intermédiaires. Nous exposons une partie de son code :

```
public class  BestEffortGroupAccessor extends _GroupAccessorImplBase
{
...

 public boolean[] multicast(any Req, GroupDescription gr)
{
...
RequestDescription requete = orb.RequestDescriptionHelper.extract(Req);
...
for(int i; i<gr.Members.length, i++)
r[i]=GrObj._request(requete.op_name);

for (int i; i<requete.params.length; i++)
{
switch(requete.params[i].mode)
{
case 0 : // paramètre in
for(int j; j<gr.Members.length, j++)
{
```

```
Any anyIN =r[j].add_in_arg();
insert(anyIN, requete.param[i].value);break;
}
case 1 : // paramètre OUT
for(int j; j<gr.Members.length, j++)
{
Any anyIN =r[j].add_out_arg();
insert(anyIN, requete.param[i].value);break;
}
case 2 : // paramètre INOUT
for(int j; j<gr.Members.length, j++)
{
Any anyIN =r[j].add_inout_arg();
insert(anyIN, requete.param[i].value);break;
}

}

}
for(int j; j<gr.Members.length, j++)
r[j].set_return_type(requete.result.type());

// invocation
 orb.send_multiple_requests_deferred (r);

// récupérer les acquittements dans return_ack[]
...

return return_ack;
}
...
}
```

– La classe *ReliableInvocable*
Cet objet s'occupe de la réception des requêtes diffusées avec la sémantique fiable. Il est associé dès sa création à un objet *GenericFederatedServer*. Il est invoqué à travers les trois méthodes suivantes.

```
public class  ReliableInvocable extends _ReliableInvocableImplBase
{
...
 public boolean save (Any Req, short req_idf )
{
... sauvegarder la requête Req en mémoire persistente...
}
 public void abort(short req_idf)
{
... supprimer la requête dont l'identificateur est req_idf
}
 public void commit(short req'idf)
{
.. invoquer la requête identifiée par req_idf sur l'objet serveur auquel il est associé.
}
...
}
```

– La classe *GenericFederatedServer*
Cette classe implante toutes les méthodes d'administration citées dans la spécification IDL. Son interface dynamique est implantée à l'aide du DSI, comme suit:

```
public class  GenericFederatedServer implements  _DynamicImplementation
{
```

```
...
 public void invoke (ServerRequest r)
{
...
switch (r.op_name())
{
case ''addServer'' : ... ; addServer(...); break;
case ''removeServer'' : ... ; removeServer(...); break;
case ''changeVersion'' : ... ; changeVersion(...); break;
...
defaults : ... traiter les requêtes des serveurs spécifiques...
}
...
}
}
```

Cet objet a accès aux structures de données décrivant le GSF (Graphe de serveurs fédérés) et l'arbre de diffusion.
il est sera "multithreadé". chaque requête arrivée est traitée dans un thread à part. Ce qui permet de traiter plusieurs requêtes en même temps.

Les objets cités ci-dessus sont présents au niveau de chaque noeud du réseau de la fédération. Les objets de communication sont créés et détruits au fur et à mesure de leur utilisation par l'objet *GenericFederatedServer*. La figure 5.2 montre cela.

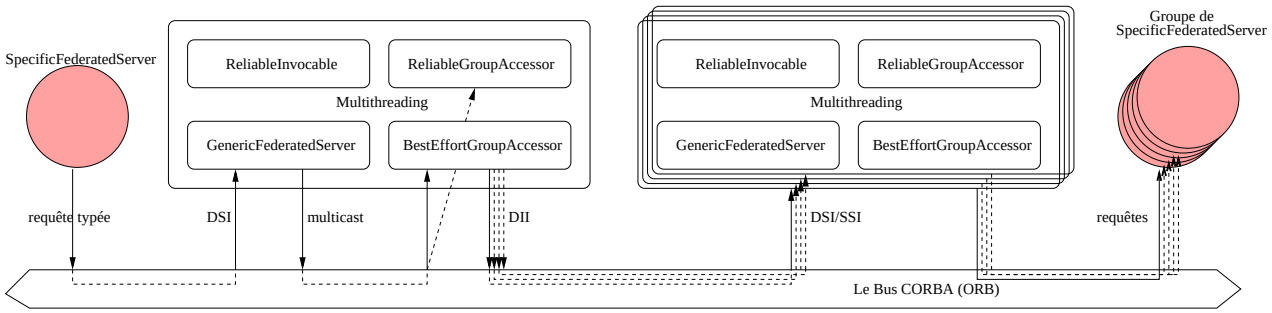

FIG. 5.2 – *mécanismes d'invocation dans la FDS*

5.5 Conclusion

La mise en oeuvre complète de l'OFDS n'est pas encore achevée. C'est une application qui demande un long travail de développement. Cependant, nous avons mis en oeuvre le mécanisme de diffusion de requêtes à l'aide de l'outil DII de CORBA. Cet outil permet d'invoquer une requête sur un groupe de serveurs d'un coup. Il suffit de donner les références de ces serveurs. Une partie du modèle d'administration de l'OFDS a été développée

en C++ dans la cadre de la thèse de [Tac97]. Il va falloir porter cette partie à l'environnement actuel pour pouvoir valider et évaluer son fonctionnement.

Conclusion générale

Le travail effectué tout au long de ce stage a conduit à la conception d'un outil de coopération d'un ensemble de serveurs déployés dans un réseau étendu. L'étude présentée recouvre les aspects suivants :

- Modélisation orientée objet de la structure de coopération proposée dans [Tac97]. Cette structure appelée **Fédération Dynamique de Serveurs** (FDS) est composée d'un groupe de serveurs qui coopèrent afin de rendre un service particulier à leurs clients. Le modèle présenté décrit essentiellement une hiérarchie d'objets qui permet l'administration automatique de cette fédération. Tout événement tels que l'ajout, la suppression, la défaillance d'un serveur ...etc, est pris en compte automatiquement par ce modèle. Aucune hypothèse n'est émise quant à la nature de ces serveurs. Il s'agit d'une fédération de serveurs génériques. Cette étude s'inscrit dans un environnement bien précis qui est celui de l'environnement CORBA.

- Etude des outils de communication de groupe dans CORBA. Cette étude s'est imposée en sachant que la communication de groupe est à la base de la coopération des serveurs d'une fédération. La fonction de communication est nécessaire à l'administration de la fédération ainsi qu'en tant que service offert aux applications qui utilisent la FDS afin de diffuser des messages. Par manque d'outils de communication de groupe standards, nous en avons conçu un modèle simple qui complète le modèle d'administration de la fédération pour donner lieu à l'OFDS: **Outil générique de Fédération Dynamique des Serveurs**.

- Une étude de cas dans le cadre du projet BARRANDE. Ce projet, entre le département INFormatique de l'INT et l'université Charles de Prague (République Tchèque), introduit un environnement appelé **SOFA** (SOFtware Appliances) qui met en oeuvre un service de vente et d'après vente de composants logiciels à travers un réseau. Le SOFA touche à plusieurs domaines en relations avec le commerce électronique et les composants logiciels. Dans cette étude, nous nous sommes intéressés à la structure de coopération sur laquelle s'appuie l'environnement SOFA. Il s'agit d'une fédération de serveurs de composants logiciels. Cette fédération constitue une application de la FDS générique que nous avons défini. Le projet BARRANDE a également permis de découvrir une technique de spécification de l'architecture et le comportement d'un composant logiciel. Un composant n'étant rien d'autre qu'une structure particulière d'objets, nous avons remodeler notre modèle orientée objet, en utilisant cette

technique, pour donner lieu à un modèle basé composant. Cette démarche reste à l'étude. Ce que nous avons présenté constitue une simple introduction d'une bonne perspective de continuation.

Cependant, La mise en oeuvre de notre outil s'est limité à l'implantation d'un module de communication qui permet la diffusion d'une requête vers un group de serveurs. Une partie du modèle d'administration a été développé dans le cadre de la thèse de [Tac97] et qui doit être portée à l'environnement actuel.

Perspectives

L'étude que nous avons traité est très complexe. Tel qu'on l'a vu au cours des différents chapitres de ce rapport, elle pose une multitude de questions en relation avec les systèmes répartis et l'algorithmique distribuée.
Nous prévoyons en perspective les éléments suivants :

- Continuer la développement de l'OFDS et valider son fonctionnement sur réseau étendu.
- Développer un service de gestion des défaillances des serveurs. Ceci renforcera la fiabilité de la diffusion de messages.
- Utilisation d'un service de transactions. Ce service sera utilisé aussi bien pour l'administration de la fédération, notamment pour garantir l'atomicité des modifications des arbres de diffusion locaux, que par les applications qui utiliseront la FDS.
- Analyser la possibilité d'utiliser des composants logiciels dans la conception de l'OFDS.

Annexe A :
Les interfaces IDL de l'OFDS

```
/*****************************************
       Le module d'administration
*****************************************/

module  mDynamicFederation
{

interface FederatedServer;

enum STATE {NORMAL, FAULTY, ISOLATED, PARTITIONNED};
enum EventType ServerAddition, ServerRemoval, DistanceChange, Partition;

typedef short FederationUniqueIdentifier;
typedef short FederatedServerIdentifier;
typedef short FederationVersion;
typedef string NetworkAddress;
typedef short FederationServerIdentifier;
typedef sequence<FederatedServer> FederatedServerSeq;
typedef short EventIdentifier;
typedef unsigned long  Distance;
typedef sequence <Distance> DistanceSeq;


/*************************************************************************/

struct FederatedServerDesc
{
string server_name;
FederatedServerIdentifier id;
NetworkAddress NetAddr;
string IOR;
};

typedef sequence <FederatedServerDesc> FederatedServerDescSeq;

struct FederationState
{
string federation_name;
FederationUniqueIdentifier FId;
FederationVersion ver;
FederatedServerIdentifier CurrentVersionCVS;
FederatedServerIdentifier CVS;
};

struct FederationDescription
{
FederationState FedS;
```

```
FederatedServerDescSeq Servers;
DistanceSeq d;
};

struct AddServerEvent
{
mDynamicServersFederation::FederatedServer newServer;
FederatedServerIdentifier toServer ;
DistanceSeq d;
};

struct RemoveServerEvent
{
FederatedServerIdentifier ServerToRemove;
};

struct DistanceChangeEvent
{
FederatedServerIdentifier ServerIdf;
DistanceSeq d;
};

//struct PartitionEvent ;

union Event switch (EventType)
{
case ServerAddition :AddServerEvent add;
case ServerRemoval : RemoveServerEvent mv;
case DistanceChange : DistanceChangeEvent dc;
      //case Partition : PartitionEvent e;
};

struct EventDescription
{
EventIdentifier id;
Event event;
};

typedef sequence<EventDescription> EventList;

struct RequestHeader
{
RequestIdentifier idf;
FederatedServerIdentifier FromServer;
FederationState state;
short BroadcastDepth;
short CurrentDepth;
mGroupAccess::RequestDescription stopping_condition;
mGroupAccess::RequestDescription delivery_condition;
};

/*****************************  Les exceptions

exception UnknownFederation
{};

exception UnknownServer
{
FederatedServerDesc s;
};

exception NotNeighbour
{
//the sender must invoke a joinFederation
};
```

```
exception NewerVersion //The receiver have a version newer than that of the sender
{
FederationDescription desc; //To update the version
};

exception OlderVersion
{
// the sender must invoke an updateVersion
};

exception Partition
{};

/*****************************  Les interfaces


interface  GenericFederatedServer
{
FederationDescription getFederation();

void joinFederation (in FederatedServerDesc f, in DistanceSeq d)
raises(UnknownFederation, NewerVersion);

void addServer (in RequestHeader h , in AddServerEvent e)
raises (NewerVersion, OlderVersion, NotNeighbour);

void removeServer (in RequestHeader h, in RemoveServerEvent e)
raises (UnknownServer,NewerVersion, OlderVersion, NotNeighbour);

void changeVersion(in RequestHeader h, in FederationDescription newVersion);

void updateVersion(in FederationDescription newVersion);

void treeBroadcast (in RequestHeader h, in mGroupAccess::RequestDescription req)
raises (NewerVersion, OlderVersion, NotNeighbour);

void starBroadcast (in RequestHeader h, in mGroupAccess::RequestDescription req, in mGroupAccess::GroupDescription gr )
raises (NewerVersion, OlderVersion, NotNeighbour);

void treePropagate (in RequestHeader h, in mGroupAccess::RequestDescription req)
raises (NewerVersion, OlderVersion, NotNeighbour);

void propagate (in RequestHeader h, in mGroupAccess::RequestDescription req, in mGroupAccess::GroupDescription gr)
raises (NewerVersion, OlderVersion, NotNeighbour);

boolean  is_alive(); // tester la vivacité d'un serveur

EventList getEvents(out DistanceSeq localWeight);  // envoyer les événements au serveur de changement de version

void CVSAgreement();  // procédure d'accord sur le choix du SCV (Serveur de Changement de Version)

void oneway you_are_CVS ();  //procédure d'élection d'un SCV

};


interface  ChangeVersionServer
{
void ChangingVersion();
EventList  receiveEvents(out EventList e, out DistanceSeq localWeight);
boolean DegradationRate();
};
```

```
interface  FaultManageServer
{
void FaultReport ();
void test();
};


};

/*****************************************
       Le module de communication
                  de groupe
*****************************************/

module  mGroupAccess
{

typedef sequence<boolean> AnySeq;

struct MemberDescription
{
string IOR;
string name;
};

struct GroupDescription
{
string GroupName;
sequence <MemberDescription> Members;
};

struct ParameterDesc
{
string par_name;
short mode;
any value;
};


struct RequestDescription
{
short idf;
string op_name;
sequence <ParameterDesc> params;
any result;
};

/************************* Les interfaces

interface  Invocable
{
boolean deliver (any msg);
}

interface  ReliableInvocable : Invocable
{
boolean save(in any req, in short req_idf);
void abort(in short req_idf);
any commit(in short req_idf);
void awake();
boolean is_commited (any req);
};


interface  GroupAccessor
```

```
{
AnyBool multicast ( in any Req, in GroupDescription gr)
void Destroy ()        // pour detruire le groupAccessor
};

};
```

Bibliographie

[AV98] keith Duddy Andreas Vogel. *Java Programming with CORBA*. Wiley Computer Publishing, 1998.

[Bak97] Seàn Baker. *CORBA Distributed Objects Using Orbix*. Addison-Wesley, 1997.

[BK86] hil Lapsley Brian Kantor. Network News Transfer Protocol. RFC 977. http://www.clizio.com/Connected/RFC/977/index.htm, Février 1986.

[Fel98] Pascal Felber. *The CORBA Object Group Service : A service Approach to Object Groups in CORBA*. PhD thesis, Lausanne, EPFL, 1998.

[Fla97] David Flanagan. *JAVA in a Nutshell, a Desktop Quick Reference*. O'Reilly, 1997.

[FP98] Radovan Janecek Frantisek Plasil, Dusan Balek. SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. In *Proceedings of ICCDS'98*, Maryland (USA), Mai 1998.

[FP99] Miloslav Besta Frantisek Plasil, Stanislav Visnovsky. Bounding Component Behavior via Protocols. TOOLS USA'99 conference, Août 1999.

[ftc98] Fault Tolerant CORBA using entity redundancy. RFP, http://www.omg.org/cgi-bin/members/doclistm.pl, Octobre 1998.

[KB91] Pat Stephenson Kenneth Birman, André Schipper. Lightweight Causal and atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3), août 1991.

[Kru56] J.B. Kruskal. On the Shortest Spanning Subtree of a Graph and the travelling salesman problem. *In Proc. Am. Math. Soc.*, 1956.

[Lan97] Danny B. Lange. Java aglet Application Programming Interface. White paper. http://www.trl.ibm.co.jp/aglets/JAAPI-whitepaper.html, Février 1997.

[Lau99] Marc Laukien. Multicast Inter-ORB Protocol. RFP, http://www.omg.org/cgi-bin/members/doclistm.pl, Mai 1999.

[Men98] Vladimir Mencl. Component definition language. Technical report, Université Charles, 1998.

[OMG96] Trading Object Service. OMG Document, Mai 1996.

[OMG97] Common Object Request Broker: Architecture and Specification. OMG Document, Août 1997.

[OOC] Object Oriented Concepts. http://www.ooc.com.

[ORB] Orbacus for c++ and java. Technical report, Object Oriented Concept.

[Pri57] R.C. Prim. Shortest Connection Networks and some Generalizations. *Bell Syst. Techn. J.*, 36, 1957.

[Tac97] Chantal Taconet. *Graphes de Réseaux Coopérants et Localisation Dynamique pour Les systèmes Répartis sur Réseaux étendus* . PhD thesis, Université d'EVRY VAL D'ESSONNE, 1997.

[VM96] Luc Bellissard Vladimir Marangozov. Component-Based Programming of distributed Applications. http://www.twente.research.ec.org/cabernet/research/radicals/1996/papers/comp-marangozov.html, Mai 1996. Projet SIRAC, IMAG-LSR.