Unification des paradigmes de partage en environnement réparti*

Mesaac Makpangou , Georges Brun-Cottan

INRIA, B.P. 105, 78153 Rocquencourt Cédex, France
tel.: +33 (1) 39-63-52-93, fax: +33 (1) 39 63 53 30
e-mail: mak@sor.inria.fr, télex: 697 033 F

Résumé

Nous présentons le concept d'objet fragmenté, une extension de la méthodologie de programmation par objets à un environnement réparti. Un objet fragmenté est une ressource partagée ; il offre à ses utilisateurs une transparence complète de la répartition. Le concepteur d'un objet fragmenté bénéficie du niveau de visibilité de la répartition qu'il souhaite. La mise en œuvre des objets fragmentés nécessite des outils. Nous présentons le principal d'entre eux : la bibliothèque d'objets de partage. Nous concluons ce papier en identifiant quelques thèmes de recherche relatifs au partage en environnement réparti.

I Introduction

Un système informatique réparti (**SIR**), est un ensemble de composants interconnectés par un ou plusieurs supports de communication (par exemple une mémoire partagée, un bus, des réseaux locaux ou des réseaux grande distance). Dans ce papier, nous nous intéressons plus spécifiquement aux systèmes répartis faiblement couplés, c'est-à-dire ne disposant pas d'une mémoire commune à tous ses composants ; ces derniers communiquent exclusivement par échange de messages. Les principales motivations du développement de tels systèmes répartis sont :

Des raisons économiques : certains composants tels que les imprimantes, les disques ou les processeurs auraient avantage à être partagés par plusieurs utilisateurs situés sur des sites différents. Ce partage aura pour conséquence de réduire le coût de revient du matériel pour chacun des utilisateurs.

Nécessité d'augmenter la disponibilité du système : quand un système centralisé est défaillant, aucun traitement ne peut avoir lieu. Un tel arrêt de fonctionnement n'est pas souhaitable ou supportable pour certaines applications. Par

*Présenté au premier Colloque Africain sur la Recherche en Informatique à Yaoundé, Cameroun.

exemple, l'arrêt du système de réservation peut avoir des graves conséquences pour une compagnie de transport.

L'avènement des systèmes répartis peut permettre de mieux satisfaire les applications ayant besoin d'une disponibilité quasi permanente du système. Ceci nécessite la mise en œuvre des mécanismes permettant de tolérer certaines défaillances des composants du système.

Aujourd'hui, on constate que les progrès importants réalisés dans la mise en œuvre des systèmes répartis n'ont pas entraîné des progrès comparables dans le développement des applications réparties. La principale raison de ce déficit est la complexité des problèmes nouveaux introduits par la répartition, en particulier ceux liés au partage.

Pour faciliter la mise en œuvre du partage, la plupart des systèmes répartis existants offrent plusieurs paradigmes de partage ainsi que des outils spécifiques pour assister les programmeurs. Chacun de ces paradigmes résoud bien certains problèmes qu'on rencontre dans un environnement réparti ; aucun n'est satisfaisant pour tous les problèmes. Les principaux paradigmes ainsi que les avantages et les problèmes posés par chacun sont présentés au paragraphe II.

Offrir à l'utilisateur plusieurs paradigmes n'est pas complètement satisfaisant. Il serait plus satisfaisant d'offrir un seul paradigme qui permet d'adresser convenablement tous les problèmes. Le paragraphe III introduit un nouveau paradigme de partage pour un environnement réparti à objets : le concept *d'objet fragmenté*. Ce concept unifie les paradigmes classiques de partage tout en réduisant considérablement la complexité de la tâche des programmeurs.

Le paragraphe IV conclut ce papier en indiquant quelques thèmes intéressants de recherche sur le partage en réparti.

II Panorama des paradigmes de partage

Le partage est vital aux applications réparties. En effet, des utilisateurs situés en des endroits géographiquement séparés peuvent avoir besoin de partager des ressources telles que : des processeurs, des disques, des imprimantes, des programmes et des fichiers. Toutefois, la mise en œuvre du partage dans un environnement réparti faiblement couplé pose plusieurs problèmes.

Premièrement, l'absence d'une mémoire commune à tous les sites ne permet pas de représenter la ressource en un endroit unique directement accessible par tous les utilisateurs qui la partagent. Ceci soulève plusieurs questions : Où doit-on localiser la représentation de la ressource partagée ? Cette représentation est-elle unique ou dupliquée ? La représentation est-elle stable ou mobile ?

Ensuite, il est de notoriété publique que les programmeurs n'aiment pas changer leurs habitudes. L'un des objectifs des systèmes répartis est d'assurer aux programmeurs la transparence de la répartition. Une telle transparence recouvre la transparence des communications, la transparence des fautes inhérentes à la répartition (corruption, duplication, déséquencement et perte de message, partitionnement du support d'interconnexion, et pannes partielles de sites), la transparence de désignation et la transparence d'accès à la ressource partagée. La mise en œuvre d'une telle transparence coûte cher. La question essentielle est donc : quel degré de transparence de répartition doit-on assurer aux utilisateurs de la ressource ?

Enfin, quand la ressource est dupliquée, deux utilisateurs peuvent voir des copies incohérentes de la ressource. Une telle incohérence est tolérable pour certaines applications, mais pas pour d'autres. Ceci suggère qu'il faille supporter plusieurs types de cohérence. La question qui se pose est donc : quel type de cohérence doit-on garantir afin d'assurer un meilleur compromis entre la cohérence offerte aux utilisateurs et les performances d'accès à la ressource partagée ?

Pour aider les programmeurs à répondre à ces questions, plusieurs paradigmes de partage sont proposés : serveur centralisé, serveur migrant, serveur partitionné, et serveur répliqué. Les systèmes existants supportent des outils (langages de spécification d'interface et pré-compilateurs associés, bibliothèques de protocoles de communication [5, 3, 4, 16]) qui facilitent l'utilisation de ces paradigmes.

Nous passons en revue ces paradigmes dans les paragraphes II.1, II.2, II.3 et II.4. Nous indiquons comment chacun répond aux questions fondamentales énoncées ci-dessus. Nous terminons par un constat concernant l'utilisation effective de ces paradigmes.

II.1 Serveur centralisé

Une seule représentation de la ressource existe à travers tout le système réparti ; cette unique représentation est confiée à un serveur situé sur un site central. Les utilisateurs désirant partager cette ressource adressent leurs requêtes à ce serveur.

Ce type de structuration est souvent mis en œuvre grâce au paradigme d'appel de procédure à distance (Remote Procedure Call – RPC) [17]. L'utilisation du paradigme RPC garantit la transparence des communications entre les clients et le serveur ; elle permet aussi la transparence de localisation et d'accès au serveur.

Le principal avantage de cette solution est sa simplicité. Elle est en outre adaptée pour le partage de ressources devant être protégées des accès non autorisés. Dans ce cas, le serveur est instancié en un endroit protégé et ne laisse passer que les accès autorisés.

Ce type de structuration pose le problème des performances des accès à la ressource partagée. En effet, s'il y a un nombre élevé de clients qui accèdent fréquemment à la ressource, le serveur devient un goulot d'étranglement. De plus, le coût de chaque accès à la ressource partagée comprend en plus le coût de la communication entre le site du client et celui du serveur.

II.2 Serveur migrant

Pour des applications réparties telles que la circulation de document, la ressource partagée est en fait accédée à tour de rôle par les utilisateurs qui la partagent. De plus, très souvent, la ressource peut être accédée plusieurs fois par le même utilisateur avant d'être sollicitée par un autre utilisateur. Pour favoriser ce genre d'utilisation, certains systèmes supportent le concept de serveur migrant [19].

A tout instant, une seule représentation de la ressource partagée existe dans le système. Comme précédemment, cette unique représentation est gérée par un serveur qui répond aux requêtes des clients désirant utiliser la ressource. Toutefois, à la différence de la solution précédente, ce serveur peut migrer de site en site, afin d'être plus proche du client qui utilise la ressource.

Le risque ici est que le nombre d'accès successifs provenant d'un même utilisateur ne soit pas suffisamment élevé pour compenser le coût de la migration du serveur chez cet utilisateur. Des techniques plus intelligentes peuvent être mises en œuvre pour éviter ce genre d'effet. Par exemple, on ne migre le serveur à un endroit donné que si on constate que le pourcentage d'accès à partir de cet endroit dépasse un certain seuil. La mise en œuvre de telles techniques rend plus difficile la tâche du programmeur d'un serveur migrant. Cette difficulté s'ajoute aux problèmes inhérents à la migration de serveur : protocole de localisation pour permettre de retrouver le serveur, atomicité de l'opération de migration, et relai des requêtes arrivant à un site après que le serveur ait été migré vers un autre [10, 12].

II.3 Serveur partitionné

Certaines ressources complexes comme des tableaux ou des structures arborescentes peuvent avantageusement être partitionnées. Les partitions sont placés en plusieurs endroits et sont gérées par des serveurs coopérants. Les différents serveurs coopèrent pour permettre aux utilisateurs d'accéder à n'importe quelle partition de la structure.

L'ensemble des serveurs coopérants peut être logiquement vu comme un serveur partitionné sur plusieurs localisations. Cette structuration est viable si les partitions ne sont pas utilisées de façon concurrante par tous les utilisateurs.

Le principal problème ici est de déterminer le partitionnement adéquat. Comme pour le serveur migrant, une bonne politique de partitionnement doit prendre en compte les statistiques d'accès par les différents clients. Cette nécessaire adéquation entre statistiques d'accès et politique de partitionnement doit être prise en compte par le programmeur de l'objet partagé. Comme pour le serveur migrant, il est nécessaire de mettre en œuvre un protocole de localisation des différentes partitions.

II.4 Serveur répliqué

La représentation de la ressource partagée peut être dupliquée en plusieurs endroits. Chaque réplique est alors confiée à un serveur particulier. Ces serveurs coopèrent pour répondre aux requêtes des utilisateurs partageant la ressource. Comme pour le cas précédent, l'ensemble des serveurs peut être logiquement vu comme un serveur unique.

La réplication permet une meilleure disponibilité de la ressource. Elle permet aussi de tolérer des fautes partielles du système. Toutefois, cette solution pose plusieurs problèmes :

Quelle cohérence pour la ressource ?

Du fait de l'existence de plusieurs copies de la représentation, les utilisateurs situés à des endroits séparés peuvent observer des états incohérents de la ressource. Certaines applications ne tolèrent aucune incohérence. D'autres au contraire, préfèrent un degré de cohérence moindre et des performances meilleures.

Pour satisfaire les besoins divers des programmeurs, plusieurs types de cohérence ont été définis. Les principaux sont : cohérence séquentielle ou forte, cohérence causale et cohérence contrôlée par des verrous [8, 2].

Quelle méthode de mise à jour ?

En pratique, un utilisateur accède à une seule copie de la ressource à la fois. Par conséquent, s'il modifie cette copie, la modification doit être reportée sur les autres afin de les maintenir cohérentes. Ce report de mise à jour peut se faire soit en réexécutant l'opération sur les autres copies, soit par simple recopie de la copie modifiée.

La bonne solution dépend de la taille de la ressource, du coût des communications et du coût du traitement.

Quelle stratégie de mise à jour ?

Quelque soit la méthode de mise à jour adoptée, il se pose le problème de savoir comment on doit l'effectuer ? Plusieurs stratégies peuvent être réalisées. Les plus connues sont la mise à jour immédiate et l'invalidation.

Dans le premier cas, toute mise à jour sur une des copies est immédiatement reportée sur les autres. Dans la méthode par invalidation, une mise à jour d'une copie provoque l'invalidation des autres. Quand une copie invalidée est sollicitée, elle doit être rafraîchie à ce moment.

Le choix du programmeur est guidé par le type de synchronisation et de communication qu'il y a entre les différents utilisateurs.

Quelle stratégie de contrôle de concurrence ?

> Quelque soit le type de cohérence choisi, le programmeur a besoin d'un mécanisme de contrôle de concurrence pour ordonnancer les accès concurrents. Le problème ici est de savoir si on fait un contrôle à priori (*approche pessimiste*) ou à postériori (*approche optimiste*).
>
> L'approche optimiste est basée sur le fait que des études empiriques montrent que peu de ressources sont réellement partagées dans les systèmes existants [18, 11]. Elle permet un haut degé de parallélisme et une meilleure disponibilité de la ressource. Elle nécessite cependant le support des mécanismes de recouvrement et de réconciliation pour essayer de déterminer à postériori un ordonnancement correct des accès conflictuels.
>
> L'approche pessimiste évite toute introduction d'incohérence. Son principal inconvénient est qu'elle est coûteuse et qu'elle réduit considéralement le parallélisme.

Le grand nombre de problèmes à résoudre et la diversité des options offertes pour chaque problème rendent très difficile la mise en œuvre d'une représentation dupliquée. Depuis quelques années, la mise en œuvre de la réplication en environnement réparti est devenue un des thèmes majeurs de la recherche en système réparti.

II.5 Constat

La plupart des environnements répartis supportent des paradigmes de partage ainsi que des outils pour assister les programmeurs. Malgré cela, la programmation d'une ressource partagée reste une tâche difficile pour au moins trois raisons.

A l'exception du paradigme de serveur centralisé accédé par des appels de procédure à distance, les autres paradigmes de partage sont mal maîtrisés. Leur utilisation exige la prise en compte des problèmes de localisation, de maintien de cohérence et de traitement de fautes dues à la répartition qui sont tous inexistants dans les systèmes centralisés. Ces problèmes s'ajoutent donc aux problèmes spécifiques de l'application.

Ensuite, l'utilisation des paradigmes de migration, de partitionnement et de réplication nécessite une analyse fine du type de partage de la ressource. Ce genre d'analyse demande un effort supplémentaire au programmeur. De plus, il faut des outils pour permettre au programmeur de spécifier le type de partage attendu. De tels outils sont encore rares aujourd'hui.

Enfin, les systèmes doivent adapter leur comportement au type de partage de la ressource [6, 1]. Des systèmes ayant un comportement versatile en fonction du type de partage désiré sont rares.

Pour favoriser l'émergence et l'utilisation d'autres paradigmes que l'appel de procédure à distance, il faut trouver un modèle de programmation qui permette de réduire la complexité du travail des programmeurs. La méthodologie de programmation par objets semble être une bonne candidate pour cela.

III Unification des paradigmes de partage

Aujourd'hui, la méthodologie de programmation par objets est largement acceptée. Dans un environnement réparti à objets, l'objet est l'unité de partage. Le programmeur dispose donc d'un modèle uniforme de communication aussi bien en local qu'en réparti : c'est l'invocation d'objet. On peut tirer avantage de cette uniformité pour rendre plus accessibles les paradigmes de partage tels que le partitionnement, la réplication et la migration. Par exemple, les protocoles de communication nécessaires pour le maintien de divers degrés de cohérence d'un objet répliqué, pour la localisation des partitions d'une ressource, pour la circulation d'un objet migrant, peuvent être encapsulés dans des *objets de partage* prédéfinis ; par la suite, le programmeur aura simplement à choisir l'objet de partage qui convient à ses besoins.

C'est cette approche que nous développons à l'INRIA au sein du projet SOR [13, 14, 15, 7, 9]. Le concept d'objet fragmenté étend la méthodologie de programmation par objets à un environnement réparti faiblement couplé. Un objet fragmenté est un objet réparti, partagé par des utilisateurs localisés dans des espaces d'adressage séparés. Du point de vue des utilisateurs, un OF offre une transparence complète de la répartition. Le programmeur d'un OF bénéficie de la visibilité de la répartition qu'il souhaite. Cette visibilité de la répartition permet au programmeur de faire les choix qui conviennent le mieux à la sémantique de son objet.

Le paragraphe III.1 présente plus formellement le concept d'objet fragmenté. Ensuite le paragraphe III.2 présente le principal outil qui permet l'unification des paradigmes classiques de partage.

III.1 Le concept d'objet fragmenté

Du point de vue externe, un objet fragmenté est un objet partagé. Il offre à ses clients une ou plusieurs interfaces d'accès. Chaque client ne voit qu'une seule interface.

Du point de vue interne, un objet fragmenté est un groupe d'objets élémentaires (ou *fragments*) coopérants. Sa représentation est donc fragmentée en plusieurs endroits. Nous distinguons grossièrement trois types de fragments : *fournisseur*, *serveur* et *référence*.

Les fragments *fournisseur*s ont trois rôles importants : ils ont la charge de créer les fragments *serveur*s ; ils traitent les demandes de liaison de nouveaux clients ; enfin, ils se chargent de connecter chaque fragment de l'objet fragmenté avec les autres fragments avec qui il doit communiquer. L'ensemble des fragments fournisseurs coopèrent entre eux pour remplir leurs rôles. Ils peuvent être vus comme un groupe d'objets isolés du reste de leur objet fragmenté.

Les fragments *serveurs* coopèrent pour réaliser les services proposés par les différentes interfaces de l'objet fragmenté. La création des fragments serveurs, leur placement et les inter-connexions sont de la responsabilité des fragments fournisseurs.

Les *références* sont des fragments assimilables à des **stubs client** [17] ou des pointeurs intelligents. Pour pouvoir accéder à un objet fragmenté, un utilisateur doit disposer au préalable d'une référence fournissant l'interface qu'il désire. Cette référence peut pointer sur un fragment serveur localisé dans son espace ou non, ou sur un fragment fournisseur.

Nous désignons par *raffinement* de référence le fait de construire une référence pointant sur un fragment serveur, à partir d'une référence qui pointe sur un fragment fournisseur. Du point de vue de l'utilisateur d'un objet fragmenté, le raffinement est obtenu par effet de bord de l'appel au mécanisme de liaison (*binding*) classique.

Les fragments serveurs ou fournisseurs ont besoin de communiquer entre eux. Ces communications ont lieu au travers d'autres objets fragmentés de plus bas niveau. Nous les appelons des objets conjonctifs. Un objet conjonctif est privé à l'objet fragmenté qui le contient.

La mise en œuvre du concept d'objet fragmenté repose d'une part sur le support d'un ensemble de mécanismes systèmes et d'autre part sur des outils pour assister les programmeurs. Le principal outil est BOAR, une bibliothèque d'outils pour les abstractions réparties. Le paragraphe suivant présente en détail cette bibliothèque.

III.2 BOAR

BOAR (Bibliothèque d'Objets fragmentés d'Abstractions Réparties) est une bibliothèque d'objets fragmentés prédéfinis [15]. Ces OFs réalisent les abstractions réparties les plus couramment utilisées.

Nous distinguons grossièrement trois niveaux d'abstractions. Les abstractions de bas niveau comprennent notamment les canaux, les queues et les objets de synchronisation. La mise en œuvre de ces abstractions repose sur les abstractions système qui sont offertes par une plateforme de base (POF).

Les abstractions de niveau intermédiaire comprennent essentiellement les objets de réplication. Ces objets réalisent différentes stratégies de réplication et garantissent différents types de cohérence entre les répliques. Ce niveau comprend aussi les objets de partitionnement, qui mettent en œuvre des stratégies de fragmentation de données et des mécanismes de coopération entre les différentes partitions. Les objets de migration sont aussi de ce niveau ; ils mettent en œuvre le protocole de circulation de l'objet migrant ainsi que le protocole de localisation nécessaire au bon fonctionnement. La mise en œuvre de ce niveau repose sur les objets du niveau inférieur.

Enfin, le niveau application est constitué des objets encapsulant des abstractions spécifiques à une classe d'applications. Les abstractions de ce niveau peuvent être réalisées en utilisant aussi bien les objets de bas niveau ou du niveau intermédiaire. Les

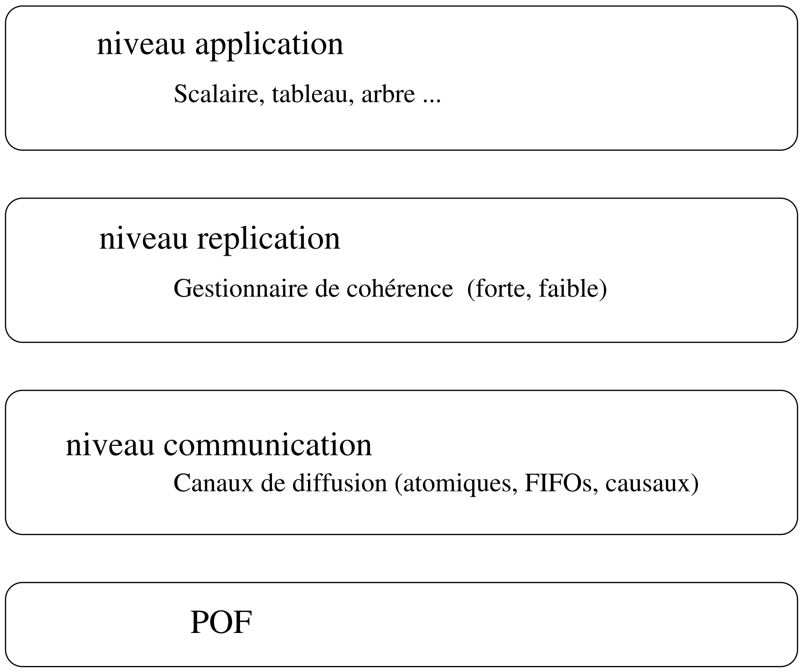

Figure 1 : couches de BOAR

Couches composants BOAR et leur niveau d'abstraction.

abstractions telles que des tableaux répliqués, des structures arborescentes partitionnées sont de ce niveau.

Dans BOAR, tous les objets encapsulant une même abstraction offrent une interface publique identique. Cette approche unifie les interfaces d'accès aux abstractions réparties. Cette unification permet, d'une part de faire évoluer le système et la bibliothèque sans affecter les programmes d'application, et d'autre part, de faciliter la migration des applications existantes vers des techniques nouvelles.

Pour réaliser son application, un programmeur doit juste choisir dans BOAR les objets qui répondent le mieux à ses besoins. Il n'a pas à se préocuper de la réalisation des abstractions réparties. Il peut par conséquent se consacrer aux algorithmes spécifiques de son application. Le reste de ce paragraphe présente les objets de partage de BOAR.

Les objets partagés de BOAR (voir figure 2) sont structurés en couches concentriques. Les couches internes sont composées d'objets conjonctifs. Plus une couche est externe, plus son niveau d'abstraction est élevé. L'ensemble des couches internes peut être vu comme un conduit, chaque couche de ce conduit apportant son protocole (communication, puis partage). L'enveloppe (couche externe) contient le mécanisme d'interfaçage avec le langage cible (interface d'accès et de concurrence, mécanisme d'intégration (par sous-typage)), la sémantique de distribution et éventuellement (sauf dans le cas d'un serveur centralisé) la copie locale. L'interface de contrôle de concurrence est exportée, mais la sémantique du contrôle reste interne à l'objet.

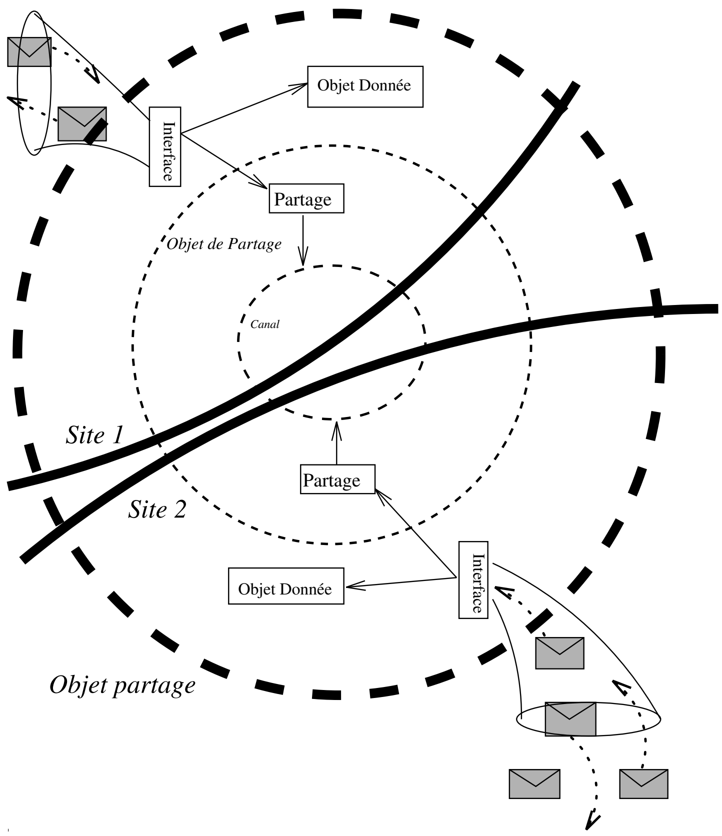

Figure 2 : Aperçu d'un OF répliqué

Toutes les entrées/sorties avec l'utilisateur de l'objet sont gérées par *Interface*. Les cercles pointillés donnent la vue virtuelle unique d'un objet fragmenté. Les cercles sont concentriques indiquant (quant aux couches internes) la construction d'un objet conjonctif de haut niveau (protocole de partage) à partir d'objet(s) conjonctif(s) de sémantique plus basse (protocole de communication : canaux atomiques, causaux,...). L'enveloppe (couche externe) assure l'intégration dans le langage cible (Interface d'accès et de concurrence, sous-typage).

Les objets de partage se veulent génériques : ils doivent être indépendants de l'objet partagé (ils n'en connaissent pas la sémantique, ils ne savent pas s'ils servent un tableau ou un entier partagé). Chaque type de partage (centralisation, migration, partitionnement, réplication) propose une interface unique, masquant les mécanismes utilisés pour la mise en œuvre. Par exemple, un objet de cohérence (partage par réplication) proposant une cohérence forte aura la même interface que celui proposant une cohérence causale ou à relâchement de verrou.

Les paragraphes III.2.1, III.2.2 et III.2.3 décrivent plus précisement les objets de partage pour la réplication, la migration et le partitionnement.

III.2.1 Objet de partage pour la réplication

Les rôles de cet objet (appelé *objet cohérence*) sont :

- ❒ Le maintien de la cohérence des répliques.
- ❒ La gestion, in fine, du contrôle de concurrence.
- ❒ La diffusion des mises à jour.

La diffusion des mises à jour met en valeur le coté canal de communication de l'objet de cohérence. La modification de la copie locale nécessite un appel ascendant[1] à l'objet *Interface* ; lui seul a connaissance de la sémantique de l'objet et de la stratégie de mise à jour (recopie vs réexecution). La propagation des mises à jour locales se fait par invocation de l'objet de cohérence par l'objet *Interface* (cf figure 2) ; la sémantique de propagation est propre à un type de cohérence. La propagation, en elle-même, utilise des canaux de niveaux inférieurs ; ceux-ci sont choisis en fonction de l'adéquation entre les protocoles qu'ils proposent (atomicité, causalité, ...) et la sémantique de cohérence voulue par l'objet de cohérence.

La gestion de cohérence et le contrôle de concurrence sont deux concepts qui pilotent les droits d'accès, et particulièrement de modification, de l'objet répliqué. L'objet de cohérence réalise la fusion de ces deux contrôles.

Les objets de BOAR permettent un contrôle fin de la concurrence. Un contrôle fin nécessitant la connaissance sémantique de l'objet partagé, comment le faire gérer par un objet générique (*objet cohérence*) ? La solution a consité à encapsuler la connaissance de l'état de concurrence de l'objet partagé dans un objet *verrou*. Cette objet est possédé par l'objet de cohérence, sa sémantique est du niveau de celle de l'objet partagé, son interface est générique (Par définition d'opérations universelles sur les verrous). La définition d'un algèbre de verrou offrant la fusion, la suppression et les test de compatibilité de verrous, fournit le caractère universel à l'interface et permet son utilisation par un objet générique (*objet cohérence*).

Les autres objets de partage sont construits sur un schéma similaire. On ne donnera que leur spécificité.

III.2.2 Objet de partage pour la migration

Il a pour rôle de déterminer la présence de l'objet et de gérer dans le cas contraire le défaut d'objet, c'est-à-dire l'importation (instanciation locale, localisation, initialisation, destruction de l'ancienne copie) de l'objet partagé (ses données). La sémantique de l'importation (localisation, synchronisation, atomicité, tolérance aux pannes) est encapsulée dans l'objet de migration.

[1] La présence d'une interface d'appel ascendant (*upcall*) est une caractéristique importante d'un objet conjonctif.

III.2.3 Objet de partage pour le partitionnement

De même que pour la migration, il a pour rôle de déterminer la présence de l'objet et de gérer dans le cas contraire le défaut d'objet. La réponse à ce défaut est propre à la sémantique de partitionnement. En cas de partitionnement statique, l'objet de partitionnement doit localiser la partition et relayer l'invocation (*upcall*) à l'objet *interface* du fragment possédant la partition. En cas de partitionnement dynamique, outre la localisation, l'éventuelle importation de la partition pourra être effectuée.

IV Conclusion

Nous avons présenté un panorama des principaux paradigmes de partage. Chacun de ces paradigmes permet de prendre facilement en compte des classes différentes de problèmes auxquels sont confrontés les programmeurs d'applications réparties.

D'une part, la complexité des mécanismes nécessaires pour la mise en œuvre de la migration, du partitionnement et de la réplication, la rareté des outils permettant d'adapter le comportement du système au type de partage d'autre part, n'ont pas permis l'émergence de ces paradigmes de partage pourtant très utiles.

L'environnement réparti à objets fragmentés en cours de développement à l'INRIA (projet SOR) permet d'offrir aux utilisateurs un modèle de partage unique : le modèle objet. Ce modèle est bien connu et aujourd'hui bien accepté de tous.

La bibliothèque BOAR offre des types d'objets fragmentés de partage prédéfinis. En particulier, elle offre des objets de partage encapsulant les protocoles de circulation d'un objet migrant, de maintien de cohérence entre les répliques d'un objet, et d'éclatement d'un objet complexe en plusieurs partitions. Dans l'environnement objets fragmentés, le programmeur choisit dans BOAR l'objet de partage qui convient le mieux à ses besoins. Il se consacre donc exclusivement aux algorithmes spécifiques à son application.

Le développement de BOAR doit être completé et un certain nombre de thèmes de recherche sont envisagés. Pour permettre aux applications de bénéficier pleinement de BOAR, nous devons déterminer dynamiquement le bon type de partage et adapter alors le comportement de notre système en conséquence. Un autre thème intéressant est celui du placement intelligent des fragments. Nous aimerions tenir compte de la charge des différentes machines.

Références

[1] H. E. Bal and A. S. Tanenbaum. Orca: A language for distributed object-based programming. *SIGPLAN Notices*, 25(5):17–24, may 1990.

[2] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proc. 2nd Annual Symp. on Principles and Practice of Parallel Programming*, Seattle, WA (USA), March 1990. ACM SIGPLAN. In SIGPLAN Notices 25(3).

[3] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[4] Kenneth P. Birman, Robert Cooper, and Barry Gleeson. Programming with process groups: Group and muticast semantics. Technical Report TR 91-1185, Dept. of Comp. Sc., Cornell University, Ithaca, New York (USA), January 1991.

[5] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 123–138, Austin TX (USA), November 1987. ACM.

[6] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, volume 25 of *Operating Systems Review*, pages 152–164, Pacific Grove CA (USA), October 1991.

[7] Peter Dickman and Mesaac Makpangou. A refinement of the fragmented object model. In *1992 Int. Workshop on Object Orientation and Operating Systems*, pages 230–234, Dourdan (France), October 1992. IEEE Comp. Society, IEEE Comp. Society Press.

[8] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the 17th Annual International Symp. on Comp. Architecture*, volume 18, pages 15–26, Seattle, WA (USA), May 1990. ACM SIGARCH.

[9] Yvon Gourhant. *Outils pour la Programmation d'Objets Fragmentés*. Thèse de doctorat, Université Paris 6 Pierre-et-Marie-Curie, Paris (France), June 1991.

[10] Sabine Habert. *Gestion d'objets et migration dans les systèmes répartis*. PhD thesis, Université Paris-6, Pierre-et-Marie-Curie, Paris (France), December 1989.

[11] Rivka Ladin, Barbara Liskov, and Liuba Shrira. Lazy replication: Exploiting the semantics of distributed services. *Operating Systems Review*, 25(1):49–55, January 1991.

[12] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[13] Mesaac Makpangou and Yvon Gourhant. Fragmented objects: a building block for distributed object-support operating systems. In *Newsletter of the Computer Society Technical Committee on Operating Systems and Application Environments*, volume 5, pages 22–25. IEEE, 1991.

[14] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Fragmented objects for distributed abstractions. In T. L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, pages 170–186. IEEE Computer Society Press, July 1994.

[15] Mesaac Makpangou, Yvon Gourhant, and Marc Shapiro. BOAR: A library of fragmented object types for distributed abstractions. In *Proc. of the International Workshop on Object-Orientation in Operating Systems*, Palo Alto, CA (USA), October 1991.

[16] Mesaac Mounchili Makpangou. *Protocoles de communication et programmation par objets : l'exemple de SOS.* PhD thesis, Université Paris VI, Paris (France), February 1989.

[17] Bruce J. Nelson. *Remote Procedure Call.* PhD thesis, Carnegie-Mellon University, May 1981.

[18] Thomas W. Page, Jr., Richard G. Guy, Gerald J. Popek, and John S. Heidemann. Architecture of the Ficus scalable replicated file system. Technical Report UCLA-CSD 910005, University of California, Los Angeles, CA (USA), March 1991.

[19] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A general-purpose programming language. *Software—Practice and Experience*, 21:91–118, January 1991.