Universite' Pierre et Marie Curie – Paris VI

DEA Systèmes Informatiques

Rapport de stage

Note technique SOR N°132

DRAFT

Mise en oeuvre d'objets fragmentés répliqués à politique pessimiste

Georges Brun-Cottan

Septembre 1992

Directeur de stage : Marc Shapiro

Lieu du stage : projet SOR, INRIA, Rocquencourt

Table des matières

I Introduction **5**

II Problématique du partage et de la réplication **7**
- 1 généralités . . . 7
- 2 Quelques schémas de concurrences . . . 8
- 3 Problématique de la réplication . . . 9
- 4 Approche orienté objet, réplication et cohérence . . . 10

III Architecture générale de BOAR **11**

IV Architecture des objets répliqués **13**
- 1 Présentation . . . 13
- 2 Les objets répliqués . . . 14
- 3 Les Verrous . . . 16
- 4 Les objets de réplication . . . 18
- 5 Exemple . . . 20

V Application Test **22**
- 1 Présentation . . . 22
  - 1.1 Contraintes . . . 22
  - 1.2 Principes de fonctionnement . . . 22
  - 1.3 Algorithme de Réservation . . . 23
- 2 Architecture de l'application . . . 23
  - 2.1 *Console* . . . 24
  - 2.2 *Cortex* . . . 24
    - 2.2.0.1 stratégie pioche . . . 26
    - 2.2.0.2 stratégie profil . . . 27

VI BOAR-V0. Les OF répliqués mis en œuvre. **29**
- 1 Construction des OF - le Factory . . . 29
- 2 Objets de BOAR-Application . . . 30
  - 2.1 Les classes *ReplicatedInteger* et *IntegerVerrou* . . . 30
  - 2.2 Les classes *ReplicatedArray* et *ArrayVerrou* . . . 31
- 3 Objets de BOAR-Réplication . . . 32

VII Travaux comparables **35**
- 1 Argus . . . 35
- 2 Avalon . . . 36
- 3 Clouds . . . 36
- 4 Orca . . . 37
- 5 Munin . . . 37
- 6 Analyse critique . . . 37

VIII Conclusion **39**
1 Constat 39
1.1 Originalités du travail 39
1.2 Parti-pris 40
1.3 Problèmes 40
2 Perpectives 40
2.1 Langage 41
2.2 Architecture de la librairie 41

Annexe A : Le cas des objets complexes **42**
1 Les objets de réplication élémentaires 43
2 Les objets de réplication complexes 43

Annexe B : Les verrous en profondeurs **45**
1 Algèbre de verrou 45
2 Plus loin avec les verrous 46

Table des figures

II.1 Concurrence dans un SGBD 8
II.2 Concurrence dans un langage 9

III.1 Le partage vu par BOAR 11
III.2 couches de BOAR 12

IV.1 Concurrence dans BOAR 14
IV.2 Aperçu d'un OF répliqué 15
IV.3 Un fragment d'OF répliqué 17
IV.4 Interface C++ de la classe d'un objet de réplication 19
IV.5 Exemple d'un entier répliqué *compte* de type *ReplicatedInteger* . . . 20
IV.6 Exemple d'un entier répliqué *compte* de type *ReplicatedInteger* . . . 21

V.1 OF Parc 25
V.2 Vue fonctionnelle de l'application 26

VI.1 Héritage des classes de BOAR 31
VI.2 *Interface*. Classe de base des objets répliqués 32
VI.3 *ReplicatedInteger*. Entier répliqué 33
VI.4 *ReplicatedArray*. Tableau répliqué 34

VII.1 Hiérarchie d'héritage des 3 classes de bases d'avalon 36

Chapitre I

Introduction

La programmation d'applications réparties reste encore maintenant une gageure. Une des raisons est la prise en compte explicite de problèmes et d'outils spécifique à la répartition (mode de communication, panne partielle, outils de synchronisation ou de mise en exclusion mutuelle spécifiques); cette prise en compte «pollue» le problème à l'origine de l'application. Une autre raison, concerne le partage des ressources. Parmi les paradigmes classiques de partage (serveur centralisé, serveur migrant, serveur partitionné, et serveur répliqué), il ne s'en trouve pas un qui soit universellement meilleur. Chacun est adapté à un type de problème. Il n'est pas rare qu'au sein d'une même application, au cours du temps, la politique de partage optimale change.

BOAR, Bibliothèque d'Objets Fragmentés d'Abstraction Répartie veut unifier les paradigmes de partage utilisé en environnement réparti [1]. En étendant la méthodologie de programmation par objets à l'environnement réparti, BOAR assure une transparence de la répartition.

Dans un environnement distribué, la réplication reste un des mécanismes de base permettant d'assurer de bonne performance (principe de cache). Dans un cadre plus général, la tolérance aux pannes est aussi assurée par une réplication des ressources critiques. Fournir des classes d'objets répliqués répond donc à une demande importante de la part de telles applications.

Notre stage de DEA à pour but la réalisation d'objets fragmentés répliqués à politique pessimiste dans BOAR. Ce travail se fait à partir du concept d'objet fragmenté (OF)[2] développé au projet SOR. Ce concept, extension de la programmation objet à l'environnement distribué, propose un paradigme de communication par objet [3, 4] ; de plus, chaque fragment (représentant d'un objet dans un contexte) peut exporter une interface propre, différente de celles des autres fragments. Un OF est vu par ses clients comme un objet unique, réparti, pouvant se présenter sous des formes (interfaces) différentes (selon le contexte/client considéré).

Le choix des objets répliqués et des politiques mises en œuvre à été guidé par la réalisation d'une application de test (gestion des entrées/sortie d'un parc d'attraction) présentée dans la suite de ce rapport. Nous avons choisi, pour une première réalisation et par simplicité, la mise en œuvre d'une politique de contrôle de concurrence pessimiste (pas de problème de réconciliation, pas de mise en œuvre de reprise (roll-back) . Cette réalisation reposera sur une plateforme pour les objets fragmentés, POF, en cours de développement. Cette plateforme fournit les services de base (gestion d'objet fragmenté, communication) nécessaire à notre travail et de façon plus générale aux objets de BOAR.

Le chapitre II apporte une ébauche des problèmes liés au partage et à la réplication.

Le chapitre III introduit l'architecture proposée pour unifier les paradigmes de partage et préciser la place de la réplication dans cette architecture.

Le chapitre IV est la présentation de l'architecture des objets répliqués de BOAR.

Le chapitre V présente un archétype (application test) d'utilisation d'objet fragmentés. Cette application permet d'identifier quelques sémantiques de cohérence et de concurrence intéressante.

Le chapitre VI décrit les objets mis en œuvre dans BOAR-V0.

Le chapitre VII est une analyse comparative des systèmes concurrents avec notre travail.

Le chapitre VIII conclut le rapport, en soulignant les problèmes rencontrés et prévisibles, en proposant une comparaison avec des travaux concurrents, et en fournissant des perspectives de développement dans le cadre d'un thèse.

En annexe sont approfondis les problèmes liés aux objets complexes (non scalaire).

Chapitre II

Problématique du partage et de la réplication

Ce chapitre expose les problèmes apportés par le partage et la réplication en particulier ainsi que les solutions courantes adoptées.

1 généralités

Tout système mettant en œuvre des objets partagés doit proposer des mécanismes permettant une gestion des accès concurrents, ceci afin de pouvoir assurer à tout client de l'objet la propriété de sérialisibilité. Cette contrainte bien que courante n'est pas impérative. Certains systèmes l'imposent, d'autres la proposent optionellement. Récemment, des modèles hybrides sont apparus, offrant un continuum entre l'absence de sérialisibilité et la sérialisibilité complète (offerte par le schéma transactionnel).

Deux grands types de contrôle de concurrence peuvent être envisagés : Les contrôles de concurrence effectué à priori (*approche pessimiste*) et ceux effectués à postériori (*approche optimiste*). L'approche optimiste est basée sur des études empiriques montrant que peu de ressources sont réellement partagées de façon conflictuelles dans les systèmes existants [5, 6]. Cette approche permet un haut degré 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 son coût et la réduction considérable du parallélisme qu'elle induit.

L'approche pessimiste ayant été choisie pour des raisons de simplicité, c 'est sa problématique que nous allons étudier plus avant.

Dans cette approche, le contrôle de concurrence est assuré par la mise en exclusion mutuelle des ressources (objets) partagées pendant les phases d'exécutions (sections critiques) les manipulant. Cette exclusion mutuelle est assurée couramment par verrouillage de l'objet. Le contrôle de concurrence est par essence extrinsèque à l'objet ; Il dépend de l'utilisation qui est faite de cet objet. Traditionnellement, ce contrôle est exercé par des structures externes à l'objet (sémaphore, moniteur).

La garantie de «cohérence» globale de l'application repose alors sur le respect d'un contrat (tester le sémaphore, par exemple, avant d'invoquer l'objet) entre les applications accédants à l'objet.

2 Quelques schémas de concurrences

Les mécanismes développés pour assurer la gestion de concurrence se sont développés principalement dans deux domaines : La gestion de bases de données et les langages de programmation (parallèle ou distribué).

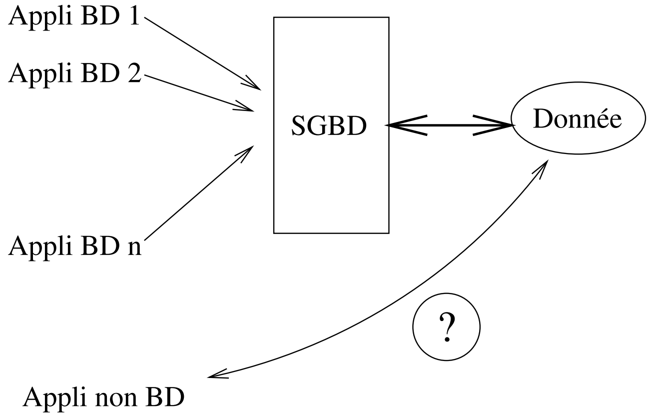

FIG. II.1 - : *Concurrence dans un SGBD*

La concurrence dans une base de données (cf figure II.1) est guidé par le partage d'une donnée persistante, par plusieurs applications indépendantes. Donnée persistante signifiait, jusqu'à il y a peu, temps et débit d'accès (lecture/écriture) faible. Applications indépendantes signifie, absence de consensus sûr inter-application, i.e impossibilité d'utiliser des systèmes de contrôle annexes (moniteur, sémaphore), reposant sur un contrat de confiance. La solution retenue, *la transaction*, est un mécanisme lourd et lent.

L'approche des langages de programmation (cf figure II.2) est presque opposée : On veut introduire du parallélisme dans un problème/programme. Il s'agit donc de co-procédure ou de threads devant assurer un partage de données avec un surcoût en temps d'exécution faible. Les données et les structures de contrôle sont en mémoire rapide. La sécurité est marginale[1]. Les structures classiques offertes sont le sémaphore et le moniteur.

1. Elle ne concerne que l'application.

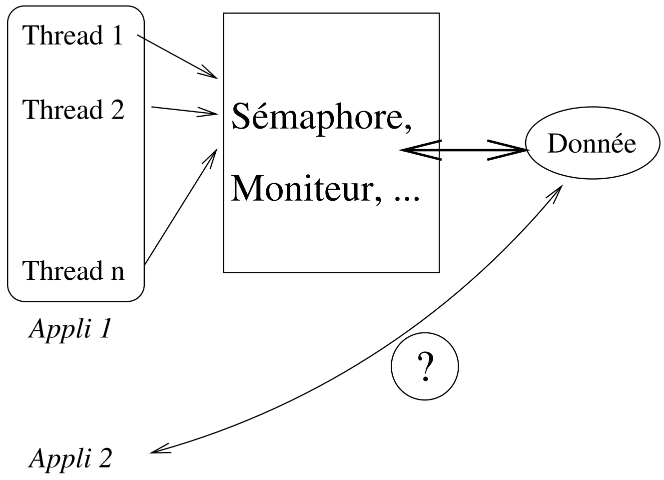

FIG. II.2 - : *Concurrence dans un langage*

3 Problématique de la réplication

En sus des problèmes de concurrence, communs à tout objet partagé, un objet répliqués amène plusieurs questions :

Quelle cohérence pour l'objet ? 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 de meilleures performances.

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 (release-consistency) [7, 8].

Quelle méthode de mise à jour ? En pratique, l'utilisateur d'un objet répliqué n'accède qu'à une seule copie de cet objet à la fois. Toute modification de la copie doit donc être reporté sur les autres copies afin de garantir la cohérence de l'objet répliqué. 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, se pose la question de savoir comment elle doit être effectuée ? Plusieurs stratégies peuvent être réalisées. Les plus connues sont la mise à jour immédiate (write-update) et l'invalidation (write-invalidate).

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.

Le choix du programmeur est guidé par le type de synchronisation et communication existant entre les différents utilisateurs ainsi que par le profil d'accès à la ressource.

4 Approche orienté objet, réplication et cohérence

Le maintien de la cohérence d'une donnée répliquée repose sur l'exécution de séquences d'opérations regroupées en section critiques par l'application. Hors du modèle objet, ces séquences font partie du code de l'application.

L'approche objet encapsule la donnée et le code la manipulant. L'utilisateur de l'objet n'a plus contrôle des séquences d'opérations garantes de la cohérence des données. L'objet se doit donc de fournir lui-même la gestion de sa cohérence. Celle-ci est bien intrinsèque de l'objet, elle dépend de la sémantique de l'objet. Essentiellement, la cohérence est un concept identifiant un «degré» de concordance de vues multiples sur un objet unique. L'utilisateur de l'objet répliqué attend un degré de cohérence ; celui-ci est intrinsèque à l'objet.

Chapitre III

Architecture générale de BOAR

BOAR veut fournir un ensemble d'outils réutilisables, polyvalents, performants, ne remettant pas en cause les habitudes de programmation et s'intégrant dans le cadre de travail du projet SOR. Construire une librairie de classe dans un langage objet offre, via l'héritage, un mode naturel de réutilisation et d'extension. Les performances et l'intégration du langage dans une équipe système ont conditionné la plate-forme de première mise en œuvre (cf le chapitre VI).

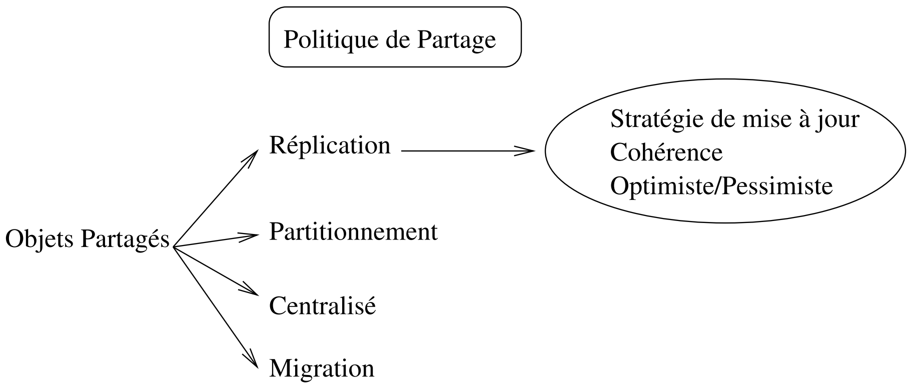

FIG. III.1 - : *Le partage vu par BOAR*

La figure III.1 situe la réplication dans BOAR. De cette vision du partage, une architecture en 3 couches (figure III.2) à été conçue. Cette architecture a comme but l'unification de tous les types de partages envisagés : réplication, partition, migration, centralisation (serveur).

Une couche est composée de classes dont l'instanciation crée un OF. Chaque couche ne dépend que des couches inférieures. BOAR est ouvert dans le sens que les classes fournies par chaque couche sont autonomes : une application peut directement utiliser un gestionnaire de cohérence pour ses besoins propres. De même, par exemple, pour un canal de diffusion.

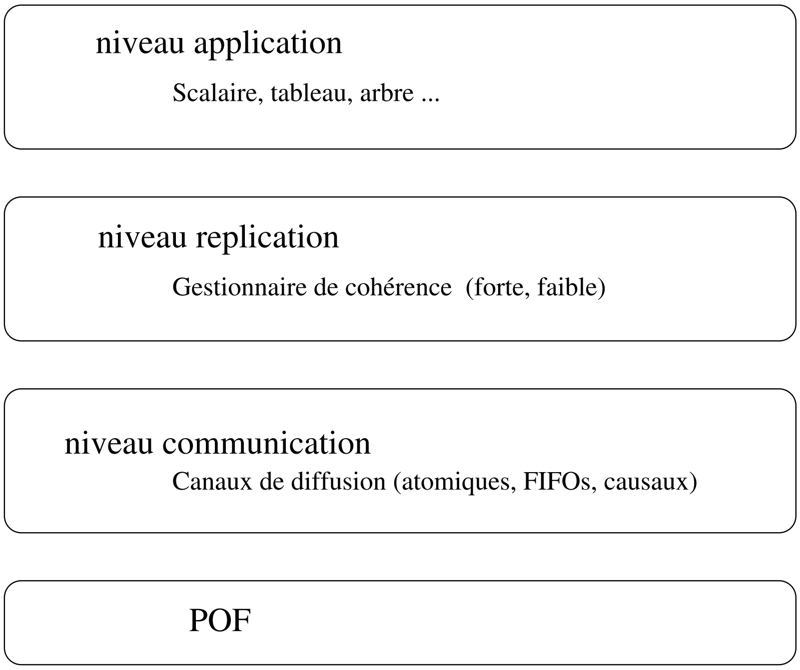

FIG. III.2 - : *couches de BOAR*
Couches composant BOAR et leurs niveaux d'abstractions.

La première couche, BOAR-Communication, offre des canaux mettant en œuvre des protocoles de diffusion FIFO, causal et atomique. Ces canaux sont point à point ou multi-points. Les objets de BOAR-Communication appartiennent soit eux-même à POF soit sont interfacés avec eux. La gestion de la distribution des OF est le fait de POF.

La deuxième couche, BOAR-Cohérence, offre des gestionnaires de cohérence. Ceux-ci se comportent comme des contrôleurs consultatifs. ils assurent l'autorisation d'accès de l'objet contrôlé et la propagation des mises-à-jour. Ils sont consultatifs : ils n'assurent leurs contrôles que sur invocations explicites de l'objet partagé. Un certain nombre de propriétés spécifiques de la politique de partage sont mis en œuvre dans ce gestionnaire (optimisme/pessimisme dans le cadre de la réplication, par exemple). Les objets de BOAR-Cohérence sont des objets conjonctifs : Ils sont vus comme des conduits de communication ; ils imposent à leurs utilisateurs une interface d'appel ascendant (upcall).

La dernière couche, BOAR-Application, offre des versions partagées des abstractions usuelles trouvées dans les langages de programmation : scalaires, tableaux, listes, ...

A tout niveaux, les classes de BOAR construisent des OF.

Chapitre IV

Architecture des objets répliqués

Ce chapitre présente l'architecture que possède tout objet répliqué de la librairie BOAR. Nous espérons que cette architecture pourra s'étendre, sans gros changement, à d'autre type de partage, en particulier, la migration, le partitionnement et le modèle serveur/client.

Après un exposé succint de l'architecture et des idées sous-jacentes suivra une présentation de l'architecture des objets répliqués, de l'objet (gestionnaire) de réplication et une description d'un chemin d'exécution d'une requête arrivant à un objet répliqué à l'aide d'un exemple plus simple que l'application test.

1 Présentation

Nous proposons une architecture d'objet qui permette une utilisation de l'objet répliqué aussi proche que possible (même interface d'accès, sous-typage de la version centralisé) de celle de sa version centralisée.

Le mécanisme de gestion de concurrence proposé se veut polyvalent et sûr (cf figure IV.1), offrant un coût d'utilisation proportionel[1] et compatible avec un système d'exploitation distribué support d'objets à grain fin ou moyen. La polyvalence est de première importance pour un système d'exploitation support d'objets dont le but est d'offrir des abstractions (service de bases) à chacun (les Système de Gestion de Bases de Données (SGBD), en particulier). La sûreté consiste à proposer un modèle de concurrence obligatoire basé sur des verrous (verrous impératifs, potentiellement bloquant sur les méthodes d'accès). Ceci est réalisé en attachant le contrôle de concurrence à l'objet.

Le type de cohérence de l'objet est choisi «à la carte» par l'utilisateur. Notre but est de laisser l'utilisateur décider du type de cohérence voulue et non du mécanisme pour le réaliser.

La délimitation nouvelle (à notre connaissance) et précise entre gestion de concurrence (extrinsèque à l'objet) et gestion de cohérence (intrinsèque à l'objet) à donné

1. De notre point de vue, il s'agira principalement de vitesse d'exécution.

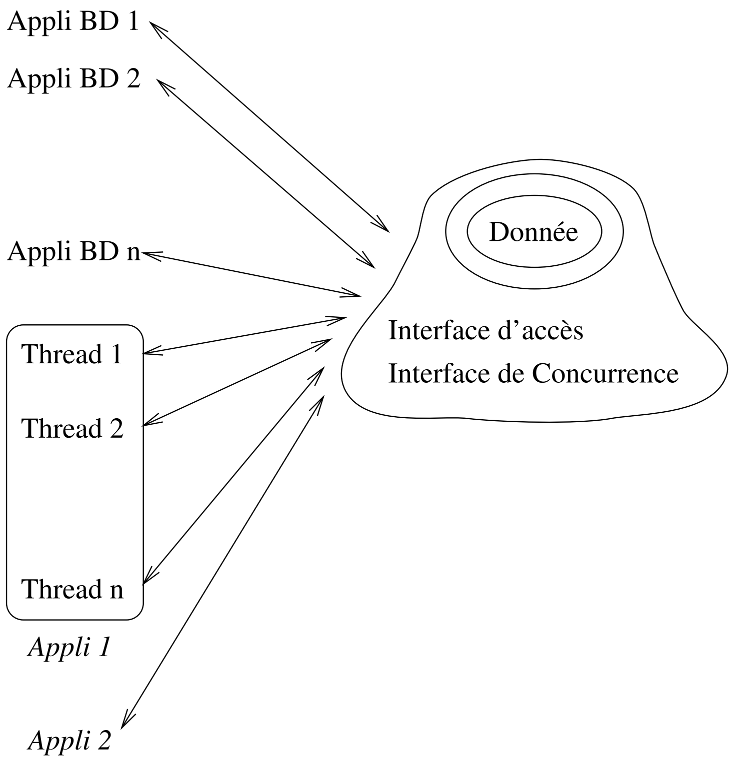

FIG. IV.1 - : *Concurrence dans BOAR*

lieu à deux points originaux de l'architecture :

- Exportation de l'interface de concurrence. Les invocations de verrouillage reste sous le contrôle de l'utilisateur de l'objet, la sémantique de ces méthodes (*beginActivity()* et *endActivity()*) reste dépendante de l'objet. Un verrouillage fin (propre à la sémantique de l'objet, verrouillage de tranche de tableaux, par exemple), et adapté (propre à la sémantique attachée a l'objet par l'utilisateur) est ainsi possible. Les éventuelles synchronisations associées sont encapsulées dans la sémantique de l'objet répliqué (en fait dans l'objet de réplication).
- Encapsulation du protocole de cohérence dans un objet conjonctif générique, l'objet de réplication. Celui-ci propose via une interface se voulant universelle, une sémantique de cohérence. Les mécanismes de réalisation sont encapsulés. Encapsuler la gestion de cohérence dans un objet permet un changement dynamique et automatique de la sémantique de cohérence. Proposer un gestionnaire de cohérence générique laisse entrevoir la possibilité de générer automatiquement (compilateur) une version distribuée d'objets existant en centralisé.

2 Les objets répliqués

Les objets répliqués de BOAR (cf figure IV.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 réplication). L'enveloppe (couche externe) contient le mécanisme d'interfacage avec le langage cible (interface d'accès et de concurrence, mécanisme d'intégration (par sous-typage)), la sémantique de distribution (réplication en l'occurence) et la copie locale. L'interface de contrôle de concurrence est exportée, mais la sémantique du contrôle reste interne à l'objet.

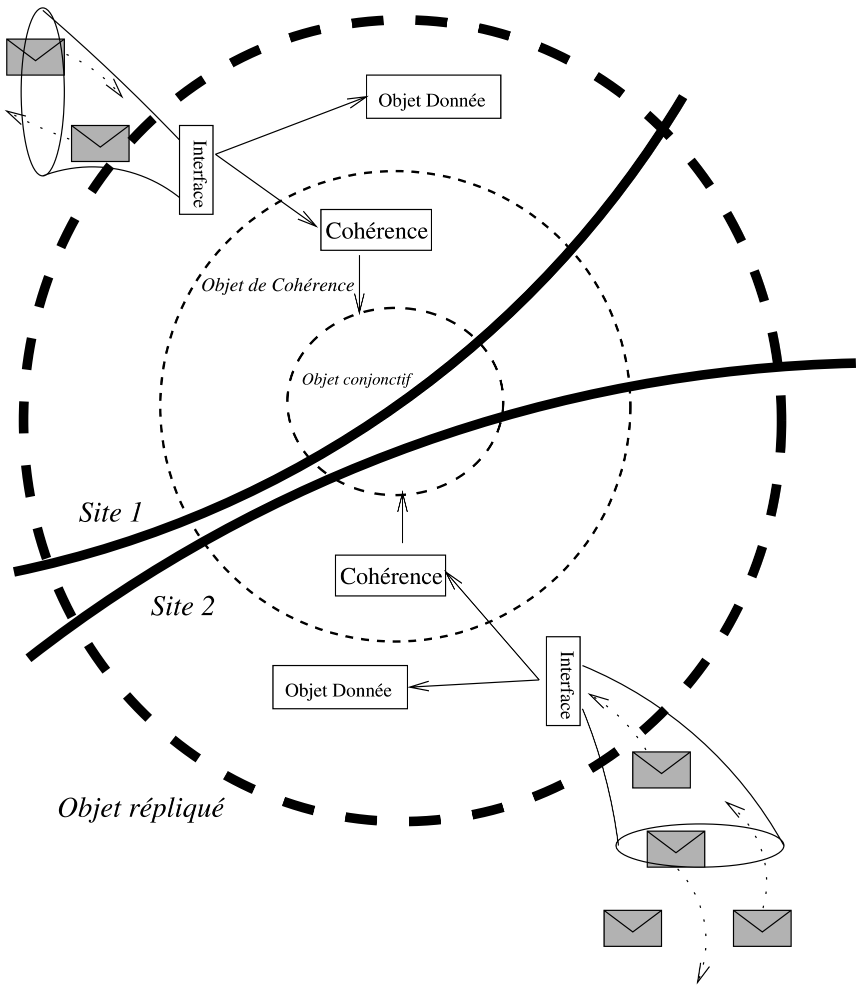

FIG. IV.2 - : *Aperçu d'un OF répliqué*

Toutes les E/S avec l'utilisateur de l'objet sont gérées par Interface. Objet Donnée *est typiquement une version centralisée de l'objet. Un objet conjonctif est un objet appartenant à BOAR-Communication.*

Dans la figure IV.2, les rectangles représentent des objets (instances de classes de BOAR), les flèches inter-objets représentent des relations de composition, les cercles, la vue extérieure qu'à un utilisateur de l'objet. Les cercles représentent, de fait, des objets des différentes couches de BOAR. Seul l'objet interface est visible (invocable) de l'utilisateur d'un OF répliqué.

Soit un programmeur ayant un objet centralisé ; la couche Cohérence lui fournit les moyens (sorte de colle) avec lesquels il pourra «simplement» mettre en œuvre une version répliquée, par exemple, de cet objet. Le programmeur pourrait reprendre telles que, les classes des versions centralisées des abstractions classiques : scalaire, tableau, liste, et autres. Ultérieurement, la génération automatique (compilateur) de la version répliquée des objets est envisagée.

Un fragment d'un OF répliqué de niveau application est composé de 3 objets :(cf figure IV.3):

– *Interface*. Exporte l'interface d'accès de l'objet ainsi que l'interface de contrôle de concurrence. Il supporte l'intelligence de la répartition : il s'assure (dialogue avec le gestionnaire de cohérence) que les accès et modifications locales sont conformes, avant de les déléguer à *ObjetDonnée* (cf ci-dessous). Il présente un point d'entrée (upcall) *Interface::update()* pour l'objet de réplication *Cohérence*; Cet upcall permet la mise à-jour de la copie locale par l'objet de réplication. Le contrôle de concurrence est délégué à l'objet de réplication.

– *ObjetDonnée*. Autant que possible, il s'agirait de la version non répartie de l'objet. Tous les accès, cast, etc, arrivent, in fine, à lui.

– *Cohérence*. C'est l'objet de réplication ; c'est un objet conjonctif. Il diffuse les mises à jour aux autres fragments de l'OF selon un schéma spécifique au protocole de cohérence. Il possède l'état de verrouillage du fragment. C'est le point de concours entre la gestion de cohérence et la gestion de concurrence.

Bien que *Interface* soit un sous-type de la version centralisée *Objet Donnée*, une relation de composition entre *Objet Donnée* et *Interface* s'avère mieux adaptée ; elle permet le changement dynamique d'*Objet Donnée*. Ceci est, en particulier, nécéssaire à la gestion d'un objet persistant. Une relation de composition entre *Interface* et *Cohérence* permet un changement dynamique de l'objet de cohérence.

Les objets *Interface*, *ObjetDonnée* et *Verrou* (cet objet est étudié par la suite, cf 3) sont spécifiques de l'objet mis en œuvre. *Cohérence* faisant partie de BOAR-cohérence est indépendant du type de l'objet répliqué.

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 réplication réalise la fusion de ces deux contrôles.

Avant d'étudier l'architecture de l'objet de réplication, nous allons introduire les objets-verrous. Le verrouillage est le principe choisi comme mécanisme de contrôle de concurrence dans la mise en oeuvre des objets répliqués. Les objets de BOAR permettent un contrôle fin de la concurrence. Un contrôle fin nécessitant la connaissance sémantique de l'objet partagé (répliqué), comment le faire gérer par un objet générique (l'objet de réplication *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 réplication, sa sémantique est du niveau de celle de l'objet répliqué, son interface est générique.

3 Les Verrous

Le verrou est le mécanisme (classe) qui encapsule la sémantique de concurrence d'un objet répliqué de BOAR. Son interface est générique, ce qui permet son utilisation par le gestionnaire de cohérence (*Cohérence* (cf figures IV.2 et IV.3).

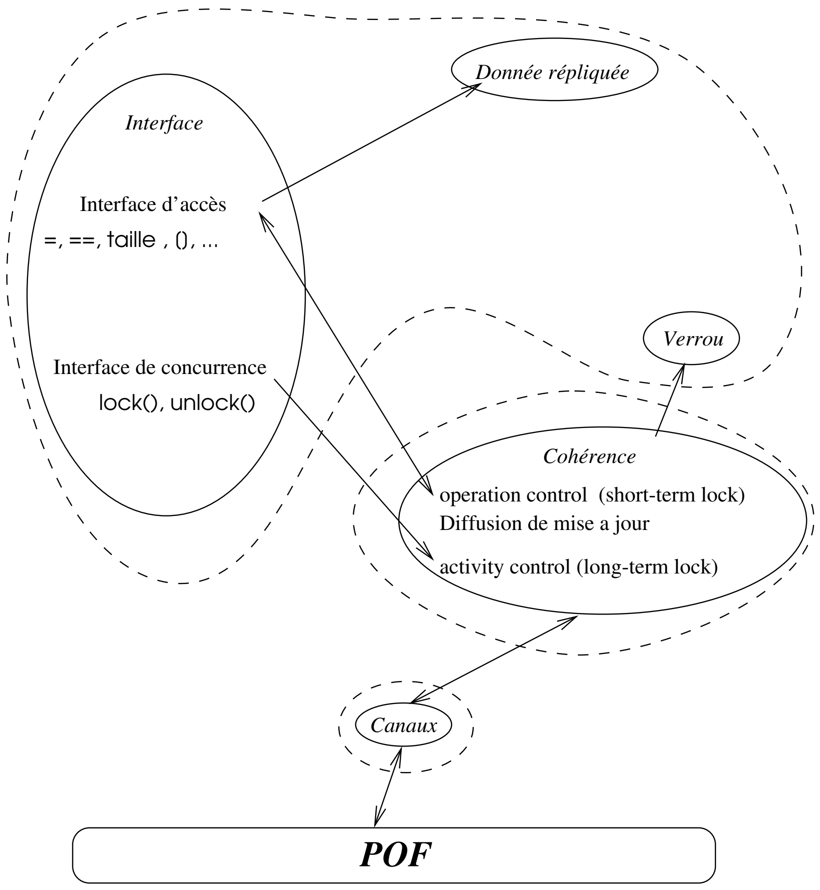

FIG. IV.3 - : *Un fragment d'OF répliqué*

Les pointillés délimitent un groupe d'objets de même niveau de sémantique. Cohérence *appartient à la couche 2 de BOAR, il n'a pas connaissance de la sémantique de l'OF répliqué.*

Les verrous sont issus de plusieurs désirs :

- Permettre un contrôle de concurrence fin d'un objet.
- Conserver la généricité au gestionnaire de cohérence.
- Rendre autonome les objets de réplication (gestionnaires de protocole de cohérence) afin de limiter les appels ascendants.

Le contrôle de concurrence découpe l'objet répliqué en granules, chaque granule pouvant-être verrouillé indépendamment des autres. A chaque granule est associé son type de verrouillage et son propriétaire (initiateur du verrouillage). On définit naturellement un élément de verrou comme étant un triplet (granule, accès, propriétaire). Un objet-verrou peut englober un ensemble d'éléments de verrou. Pouvoir manipuler les objet-verrou à l'aide d'opérateur d'union et de différence représente un moyen naturel pour verrouiller ou déverrouiller des tranches de tableaux, par exemple. Moyennant un test de compatibilité de verrou supplémentaire, ces opérations semblent suffisantes et naturelles. On remarque qu'une définition légérement plus étendue, permet de construire une structure d'algébre, garantissant ainsi, la complétude des opérations (cf annexe 1). Cette complétude garantit la généricité de l'interface de verrou proposée. Les opérations proposée par le verrou sont les suivantes :

Construction d'un verrou. Sont donnés en paramètre, le granule ou l'ensemble des granule concernés et le type d'accès a effectuer. Le propriétaire est déduit (thread-id, fragment-id, OF-id).

Union de deux verrous. Renvoie un Verrou dont les éléments de verrous sont l'union des éléments de verrous de chacun des opérandes.

Différence de deux verrous. Opérations indispensables ou déverrouillage.

Intersection de deux verrous.

Complémentaire. Dans les appels non bloquants, permet (avec l'intersection) de donner des verrous compatibles avec l'état de verrouillage de l'OF. L'objet répliqué peut en profiter pour subdiviser sa demande de verrouillage afin d'augmenter la concurrence.

Verrou UN. Nécessaire pour l'opération de complémentarité.

Verrou ZERO. Verrou neutre.

Test de validité/compatibilité de verrou.

La mise en œuvre des verrous repose sur la création d'une classe de base abstraite *Verrou* exportant les opération de verrous. Chaque classe de BOAR-Application, construit une version dérivée apropriée à sa sémantique.

Le verrou en lui-même n'a aucune interaction avec l'objet répliqué. C'est son passage à l'objet de réplication et l'utilisation que celui-ci en fait qui accomplit les contrôle de concurrence et de cohérence. Le verrouillage proprement dit consiste au passage d'un objet-verrou à l'objet de réplication (*Cohérence*).

4 Les objets de réplication

Les rôles de l'objet de réplication (*cohérence*, cf figure IV.2) sont :

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

La diffusion des mises-à-jour met en valeur le coté conjonctif de l'objet de cohérence. La modification de la copie locale nécessite un appel ascendant à l'objet *Interface*; lui seul a connaissance de la sémantique de l'objet et de la stratégie de mise à jour (recopie vs rééxecution). La propagation des mises-à-jour locale se fait par invocation de l'objet *Cohérence.update(Bloc Opaque)* (cf figure IV.4) par l'objet *Interface* (cf figure IV.2); la sémantique de propagation est propre à un type de cohérence. La propagation, en elle-même, utilise des canaux conjonctifs de niveaux inférieurs ; ceux-ci sont choisis en fonction de l'adéquation entre les protocoles qu'ils

```
class Coherence {
Verrou *verrou;
Interface *interface; // pour le upcall

public :     // group:

  Coherence (Interface *p, Verrou& v) : interface(p), verrou(&v){}

  void update(void *block);  // block

  const Verrou& beginOperation(const Verrou& v);
  void endOperation(const Verrou& v);

  int beginActivity(const Verrou& v);
  void endActivity(const Verrou& v);

  const Verrou& getVerrou();    // fournit le verrou possede par coherence
};
```

FIG. IV.4 - : *Interface C++ de la classe d'un objet de réplication*

proposent (atomicité, causalité, FIFO) et la sémantique de cohérence voulue par l'objet de cohérence.

En ce qui concerne la gestion de la cohérence et la gestion de la concurrence, deux familles de fonction sont proposées : La famille *beginOperation/endOperation* et la famille *beginActivity/endActivity*. Les appels sont pour l'instant bloquant : Le retour n'a lieu que si le type d'accès demandé ne viole pas la cohérence de l'objet tel que definit par la sémantique de l'objet de réplication.

La famille *beginOperation/endOperation* offre une «sorte d'atomicité» d'opération tel que décrit dans Orca [9] ; cet atomicité dépend du type de cohérence mis en œuvre par l'objet de réplication: en ce sens, cette famille est une mise en œuvre des *short-term lock* d'Avalon [10]. L'atomicité offerte par un objet de réplication proposant une sémantique de cohérence forte se veut celle définie dans Orca.

La famille *beginActivity/endActivity* permet la mise œuvre de transactions légères. Le principe de base de *beginActivity* est la réalisation d'une synchronisation-verrouillage atomique. Cette famille est conçue pour pouvoir garantir la sérialisibilité aux utilisateurs d'objets répliqués ; elle peut éventuellement garantir l'atomicité. Elle représente les *long-term lock* d'Avalon [10]. Seul l'utilisateur d'un objet répliqué, selon la sémantique de l'application, est à même de contrôler cette propriété. Cette famille doit donc en principe être sous le contrôle de l'utilisateur de l'objet répliqué. C'est le point d'entrée du contrôle de concurrence.

L'objet de réplication réalise la fusion des contrôles de concurrence et de cohérence. Pour ce faire, l'objet de réplication possède un objet-verrou (*Coherence::verrou*). Ce verrou représente l'état de verrouillage du fragment de l'objet de répliqué. Ce verrou est en permanence remis à jour, par l'objet de réplication en fonction des invocations des familles de fonction *beginOperation/endOperation* et *beginActivity/endActivity* par le fragment (Objet *Interface*) et des messages de diffusion en provenance des autres fragments de l'objet répliqué (dépendant du type de cohérence mis en œuvre). Les mises à jour du verrou sont effectués par les opérations définit par l'algèbre de verrou.

La suite du chapitre donne à travers un exemple simple, le chemin d'exécution d'une invocation d'un objet répliqué.

5 Exemple

Soit un entier répliqué, objet de BOAR. Supposons l'exécution du code ci-dessous (figure IV.5). Ce code représente l'exemple classique du retrait d'un compte en banque.

```
.
.
ReplicatedInteger somme; -- compte en banque répliqué
.
.
-- retrait de 'somme' F
procedure Retrait (somme : entier);
if  (compte > somme)
  then
     compte.decremente (somme);
  else
     -- retrait impossible
if
```

FIG. IV.5 - : *Exemple d'un entier répliqué* compte *de type* ReplicatedInteger

De façon évidente, dans un environnement permettant la multiprogrammation, le code ci-dessous sera incorrect. La validité de ce code nécessite : L'atomicité de l'opération de décrementation. La mise en exclusion mutuelle de l'ensemble de la procedure Retrait (ou tout du moins de la séquence test-décrémentation).

Decremente est une méthode de l'entier répliqué, le caractère atomique ou non de l'opération fait partie du contrat de l'objet répliqué (type de cohérence choisi).

La procédure *Retrait* attribue à l'entier répliqué une sémantique particulière (celle d'un compte en banque avec ses contraintes). Seul l'utilisateur de *compte* est à même de définir la section critique nécessaire et éventuellement optimale.

De facon intuitive, l'atomicité de l'opération *compte.decremente()* sera assuré par la famille *beginOperation/endOperation* de l'objet de réplication. L'exclusion mutuelle sera assuré par la famille *beginActivity/endActivity.*

Le code suivant (figure IV.6) est le code dont nous allons suivre l'exécution.

La première ligne de *Retrait* consiste en la création d'un verrou représentant un accès exclusif (en écriture). La lecture reste permise. A ce stade, l'accès à l'objet est totalement libre.

La suite consiste en la mise en exclusion mutuelle de l'objet. Tout objet répliqué exporte les appels de la famille *beginActivity/endActivity.* Le verrou est passé en paramètre. L'objet *interface* de l'objet répliqué délègue l'invocation directement à l'objet de réplication (*Cohérence::beginActivity()*).

```
.
.
ReplicatedInteger somme; -- compte en banque répliqué
.
.
-- retrait de 'somme'
procedure Retrait (somme : entier);
ReplicatedIntegerVerrou verrou(WRITE); -- create an exclusive access (write) lock.

compte.beginActivity(verrou) -- début exclusion mutuelle
if  (compte > somme)
  then
     compte.decrement (somme);
  else
     -- retrait impossible
if
compte.endActivity(verrou) -- fin exclusion mutuelle
```

FIG. IV.6 - : *Exemple d'un entier répliqué* compte *de type* ReplicatedInteger

Le test en lui-même nécessite un changement de type (en entier, dans l'exemple). Ce changement de type est mis en œuvre (surcharge de l'opérateur de cast en C++) par l'objet répliqué. Il est à noter que cette opération doit se faire en accord avec l'état de concurrence de l'objet, i.e implique un demande d'accès en lecture (verrou en lecture).

Supposons l'opération de décrémentation mis en œuvre par l'objet répliqué comme une addition suivie d'une affectation. Cette séquence d'opération doit être atomique. L'objet répliqué (l'objet *Interface* en fait) demande donc un verrouillage en écriture à l'objet de réplication. Pour ce faire, il fait appel au fonction de la famille *beginOperation/endOperation* de l'objet de réplication. Ceci assure (La sémantique est fait sous la dépendance de l'objet de réplication) l'exclusion mutuelle du fragment local de l'objet de réplication. Intuitivement, cela assure que des mises à jour provenant de fragments distants ne pourront avoir lieu pendant l'exécution de décrémentation. L'opération de décrémentation se poursuit par une diffusion des mises à jour [2] au fragments distants (*Cohérence::update()*). Le verrou «opérationnel» (short-term lock) est ensuite relaché (*Cohérence::endOperation(verrou)*); l'opération de décrémentation est alors terminée.

La dernière ligne de *Retrait* libère le verrou «transactionnel» (long-term lock).

2. Soit copie locale, soit opération avec ses paramètres. Le bloc est opaque pour l'objet de réplication.

Chapitre V

Application Test

1 Présentation

L'application présentée a en charge la gestion répartie d'un parc d'attraction. Un parc est constitué de plusieurs portes d'entrée. L'application a en charge la gestion des entrées/sorties de visiteurs avec le respect d'un quota maximal (contrainte de sécurité) de visiteurs dans le parc.

1.1 Contraintes

Les contraintes de l'application sont les suivantes :

(C1) L'application est répartie, i.e une application tourne sur chaque site. Un site correspond à une porte d'accès du parc. L'arrêt d'un site[1] permet de continuer la gestion du parc en mode dégradé : Seule, une porte d'accès est condamnée.

(C2) L'application doit satisfaire l'invariant suivant : Le nombre de visiteurs dans le parc a tout instant doit être inférieur a un quota max préétabli.

Les contraintes suivantes sont fournies pour des raisons de performances et de réalisme :

(C3) Le mode de décision ne doit pas, la plupart du temps, reposer sur un consensus global. Par mode de décision, on entend l'autorisation ou non de laisser rentrer des personnes a une porte.

(C4) Le rendement du parc doit être le plus élevé possible, i.e le nombre de visiteurs dans le parc doit être le plus proche possible du quota.

1.2 Principes de fonctionnement

Un certain nombre de principes découlent des contraintes énoncées au dessus :

1. Tolérance au panne.

(C1) présuppose une architecture symétrique [2].

(C2) fournit intuitivement la notion de ce qui doit être partagé : La vision qu'a chaque site, du nombre de visiteurs dans le parc à tout instant. En fait pour des raisons de performances (C3), le choix se portera sur un autre partage.

(C3) est satisfait de la façon suivante : Chaque site réserve un certain nombre de places, se constituant un stock de réservation. Le stock est tel que (C2) doit-être vérifié, i.e $\sum entree_i \leq QuotaMax$ ($entree_i$ représentant «à tout instant» le nombre de visiteur dans le parc rentré par la porte i). Tant qu'un site n'a pas épuisé son stock de réservation, la décision de laisser entrer des personnes dans le parc est locale. Lorsque qu'un site n'a plus de place libre dans son stock de réservation, les demandes d'entrées deviennent bloquantes, tant que des personnes (en nombre suffisant) ne sont pas sorties ou que le site n'a pas augmenté son stock de réservation.

1.3 Algorithme de Réservation

Le principe de réservation est mis en œuvre de deux façons différentes :

Réservation statique : Le stock de place alloué a chaque site est prédéterminé. Ce modèle est simple mais présentent de gros inconvénient : nombre de site du parc supposé connu et fixe, mauvais comportement en cas de panne (rupture de (C4)), pas de prises en compte des différences de flux d'entrée/sortie entre portes d'accès (Accès principaux, accès secondaires).

Réservation dynamique : Le stock de place est évolutif. Ce modèle n'a aucun des inconvénient de la réservation statique ; il est plus coûteux sur le plan communication (charge du réseau).

Une analyse de ces deux classes d'algorithmes montre que les inconvénients de la réservation statique sont prohibitifs. Nous avons donc décidés de mettre en œuvre, tout d'abord, la réservation dynamique en fonction de trois arguments principaux :

- Elle est meilleure (en terme de simulation du réèl).
- Le coût reste, en fait, modéré si le nombre de places libres échangées unitairement, pour la constitution des stocks, par les Cortex n'est pas trop petit.
- Ce mécanisme met en valeur le concept d'OF, ce qui est le but de l'application.

Avec le mécanisme de réservation, la contrainte d'intégrité (C2) s'exprime de la façon suivante :

$$\left(\sum_{i=0}^{nbSitesActifs} stock_i\right) \leq QuotaMax$$

2 Architecture de l'application

Schématiquement, l'application converse avec le parc. Celui-ci propose actuellement les services de contrôles d'accès. L'application consiste principalement en un

2. Le code tournant sur chaque site est identique.

interpréteur de commande construit à l'aide de lex et yacc. Les commandes interprétées seront des ouvertures/fermeture d'un accès, des demandes d'entrées/sortie de personne, et éventuellement, un ordre de commutation en mode simulation. Ce mode simulation, en fait, génèrerait des entrées/sorties pilotées par une loi statistique (poissonniennes).

Le parc reçoit les demandes d'entrées/sorties effectuées par l'application. Le parc est représenté par un OF *Parc*. Conceptuellement, l'ensemble des services susceptibles d'être offert par le parc devrait être accédés via cet OF.

Design de l'OF *Parc* L'idée est d'associer à chaque type de service du parc un type de fragment mettant en œuvre ce service et exportant son interface. l'OF parc serait constitué par autant de type de fragments qu'il y aurait de services. La mise en œuvre actuelle, présente un seul type de service : L'accès d'entrée/sortie.

Un seul type de fragment est donc décrit, mis en œuvre dans la classe Acces. C'est cette classe qui exporte l'interface du service d'accès, constituée des méthodes *Acces::entre()*et *Acces::sort()*.

Le service d'accès a, en fait, été décomposé en deux services. Le premier, *Console*, met en œuvre la gestion locale des entrées/sorties. Le deuxième, *Cortex*, met en œuvre la gestion des réservations ; c'est lui qui met en œuvre la répartition des accès. De facon naturelle, *Cortex* tournera en tâche de fond, *Console* représentant la tâche dans laquelle tournera l'interpréteur de commande. Un mécanisme de préemption étant nécessaire, la librairie utilisée sera LWP de SUN.

La figure V.1 résume, la structure de l'application. Les variables *entree* et $stock_i$ sont partagées par les deux tâches *Console* et *Cortex*.

2.1 *Console*

Cette tâche met en œuvre principalement l'interface utilisateur. Elle représente, en fait, la version centralisé des accès. En ce sens, elle ignore la répartition.

La structure de donnée principale dans *Console* est l'instance de la classe Acces. Chaque requète d'entrée (appel à $Acces :: entre()$) génère une tâche exécutant une attente active sur le critère $entree + entreeVoulue \leq stock$. Ces tâches sont gérées en FIFO sur une même base de priorité (pas d'inter-préemption).

2.2 *Cortex*

Cortex représente à la fois une tâche et une classe. En fait , la classe *Cortex* n'a pas d'interface publique et la tàche est lancée dans le constructeur de la classe. Un moniteur gère les accès concurrents de *Console* et *Cortex* sur le stock du site. *Cortex* est lancée lors de l'instanciation d'un Acces.

Les *Cortex* collaborent entre eux, pour assurer une répartition des stocks de réservation, parmi les sites, la plus optimale possible. Par optimale, on entend la vérification de toute les contraintes et en particulier de (C2) et (C3).

La réservation dynamique est pilotée par un système de priorité basé sur les

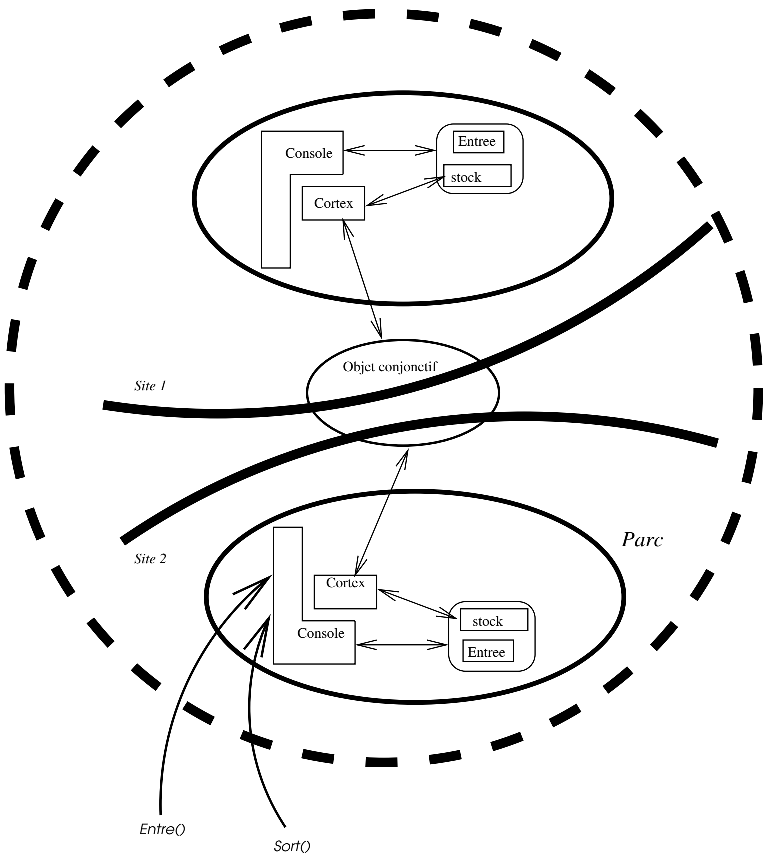

FIG. V.1 - : *OF Parc*
Design de Parc. L'interface est relayé à la classe Console. Seul Cortex a des liens avec l'OF de partage.

règles suivantes :

- La priorité doit être proportionelle au rapport $entree_i/stock_i$. Le but est de minimiser le blocage des portes qui ont du monde.
- La priorité doit être pondérée par les variation de flux d'entrée, i.e par les variations de $entree_i/base\_temps$. Ce mécanisme à pour but d'anticiper les demandes fortes. Un contrôle des oscillations est à prévoir.

La figure V.2 donne une vue fonctionnelle de l'application.

Cortex utilise pour le partage associé à la réservation, des OF de BOAR-Application. Cet objet joue le rôle d'objet conjonctif de l'OF *Parc*

Deux stratégies de mise en œuvre de la réservation dynamique ont été isolées afin de mieux apprécier l'éventail des possibilités offertes par la programmation utilisant le concept d'OF :

- La première consiste à créer un pool (une pioche) de places libres disponibles. En fonction de sa priorité, un site à droit à un certain nombre de places libres d'avance. L'information à partager est, dans ce cas, le nombre de places libres.

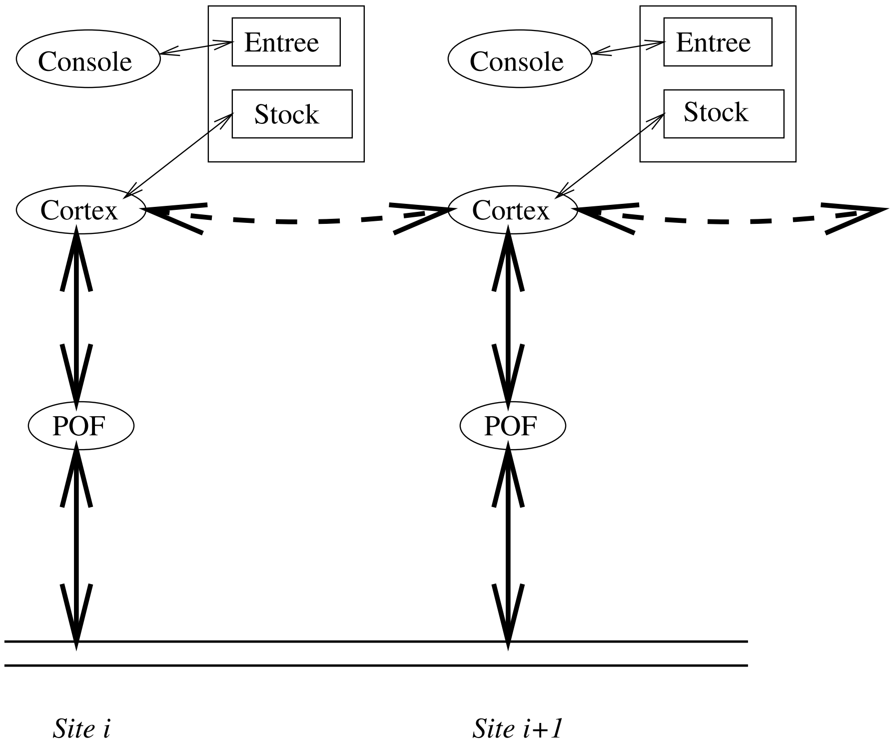

FIG. V.2 - : *Vue fonctionnelle de l'application*

Cette vue schématise les appels effectués. Cortex et POF peuvent être des taches. Les flux de services sont représentés en gras. Les accès au variables sont en fin. Le pointillé est la vue de service logique qu'on les Cortex (cf protocole en couche).

– La seconde est plus complexe. Elle reste équitable en cas de charge élevée du parc. L'idée consiste à minimiser le nombre de places libres non réservé, i.e ne faisant pas partie d'un stock de reservation sur un site. Chaque site se bat pour avoir un stock de réservation le plus grand possible: la règle du jeu est sous la dépendance de priorités. L'information à partager est, dans ce cas, un profil de réservation du parc (cf ci-dessous).

2.2.0.1 stratégie pioche

L'information à partager est simplement une variable scalaire représentant le nombre de places libres (et hors de tout stock) disponibles dans le parc.

Dans cette stratégie, toute modification de la priorité d'un site entraîne un test et une affectation (en exclusion mutuelle) du nouveau nombre de places libres hors stock. Le type de cohérence souhaité est forte, au niveau opération sur l'entier, pour assurer la contrainte (C2) ; i.e la lecture de l'entier doit retourner la dernière valeur écrite. BOAR doit donc fournir un entier répliqué à cohérence forte. Une étude plus fine montre qu'une cohérence à relâchement de verrou est suffisante, si l'objet répliqué garantit l'atomicité de ces opérations, et les encadre par des verrous.

Le nouveau stock voulu est donnée par :

$stockVoulu = (deltaE + entreeVoulue) * (1 + beta * (1 - 2 * taux())) + entreeVoulue;$

$entreeVoulue$ représente le nombre de places voulues pour satisfaire des personnes qui attendent (font la queue à l'entrée). $(deltaE + entreeVoulues) * (1 + beta * (1 - 2 * taux()))$ représente la consommation prévisionnelle. $deltaE$ représente le flux d'entrée/sortie effectif sur la dernière "période"; $(1 + beta * (1 - 2 * taux()))$ la pondération en fonction de la charge du parc; on tempère ainsi les appétits prévisionnels des sites lorsque le parc arrive à saturation. Ceci garanti une meilleure équité.

Une modération de l'activité intersite (coûteuse en communication) est obtenue en ne validant que les requêtes entrainant une modification notables de l'état du site demandeur d'un accroissement ou d'un diminution de son stock : $((stock < stockVoulu * (1 - alpha))||(stock > stockVoulu * (1 + alpha)))$.

La concurrence des 2 tâches *Cortex* et *Console*, impose le contrôle de l'accès au stock du site par un moniteur. Celui-ci est mis en œuvre par la librairie LWP de SUN.

2.2.0.2 stratégie profil

Chaque site se maintient une vue locale (le «profil») de l'état de réservation et de priorité de chacun des autre sites. De là, est construite une stratégie visant à réclamer à des sites moins prioritaires, une partie de son stock. L'intérêt de cette mise en œuvre est d'intégrer les mécanismes de requête/obtention sous la forme d'accès à une structure partagée.

L'information partagée est un tableau dont chaque entrée i est un agrégat reflétant la priorité et le stock du site i et son état (présent ou non). Le tableau contient une entrée par site potentiel.

Les accès aux tableaux suivent les principes suivants :

- On lit et modifie son entrée sans contrainte (pas d'exclusion mutuelle).
- On lit l'entrée de chacun sans contrainte, mais avec une cohérence faible ; i.e la valeur lue n'est pas forcément à jour.
- On accroit son stock en piochant dans le stock de son voisin. Cette pioche doit se faire en exclusion mutuelle. Le fait de passer en exclusion mutuelle doit entrainer pour le demandeur une vue cohérente de l'entrée du tableau.

Ces règles assurent le respect de (C2). Les deux premières règles permettant le support d'une cohérence faible, doivent augmenter notablement le parallélisme, et concomitamment la vitesse d'éxécution, et diminuer le traffic de communication.

BOAR met en œuvre une objet de réplication muni d'une telle politique de cohérence (*LazzyConsistency*). Un tableau de structure (priorite, stock, etat), fourni par BOAR-Application, associé avec cet objet de réplication est donc choisi.

Régulièrement , chaque *Cortex* i vérifie que son stock de places réservées est suffisant ($stock_i > stockVoulu * (1 - alpha)$). Dans le cas négatif, *Cortex* lance une quête chez ses voisins. Le nombre de places prises dans le stock du voisin correspond à l'égalité des priorités des deux sites (recquérant-recquis).

La priorité est donné par $stockVoulu/stock$. Le calcul du nouveau stock voulu est identique à celui mis en œuvre dans la version Pioche, à l'exception de la charge du parc qui n'intervient pas.

La concurrence des 2 tâches *Cortex* et *Console*, impose le contrôle de l'accès au stock du site par un moniteur. Celui est mis en œuvre par la librairie LWP de SUN.

Chapitre VI

BOAR-V0. Les OF répliqués mis en œuvre.

BOAR-V0 est constitué des classes créées pour les besoins de l'application test. Ce chapitre décrit les objets répliqués et les objets de réplication mis enœuvre.

Construire une librairie de classe dans un langage objet offre, via l'héritage, un mode naturel de réutilisation et d'extension.

La mise en œuvre de BOAR-V0 se fait en C++. Ce choix est justifié par sa grande diffusion dans les équipes systèmes (et dans le projet SOR, en particulier) et ses performances à l'exécution. Le choix d'un langage existant permet de plus un temps de développement raisonnable. Le support des types paramétriques et ses évolutions attendues (type-checking dynamique, en particulier) à venir en fait un bon candidat.

Les objets répliqués et les objets de réplication sont des OF. Comme tout OF, leurs créations est basé sur les Factory[11]. Nous allons dans un premier temps décrire la mise en œuvre des Factory ; la suite du chapitre sera consacré au particularisme des objets de réplication et des objet répliqués.

1 Construction des OF - le Factory

Les OF sont des objets distribués potentiellement sur des contextes (espaces d'adresses, site) . Les mécanismes d'instanciations des langages utilisées sont locaux à un contexte. Un mécanisme d'attachement et de gestion (migration, destruction) intercontexte des fragments composants l'objet doit être mis en œuvre : c'est le rôle des Factory. Le Factory est le mécanisme qui met en place le lien entre les fragments : Ce lien est l'objet conjonctif. Un objet conjonctif est lui-même, un OF. Les objets conjonctifs à la base de la construction des OF sont ceux proposés par le run-time system *POF*, i.e les «canaux» (cf [12]).

Le Factory à besoin de deux paramètres pour effectuer l'attachement : un identifiant unique sur le système de l'OF (OFid) et la liste des objets conjonctifs (leurs types, en fait).

Deux mécanismes d'attachements sont envisageables : Un attachement «topdown», où chaque fragment s'instancie localement et instancie récursivement, ses objets conjonctifs. L'attachement est réalisé ensuite, par le gestionnaire d'OF de POF. Un attachement «bottom-up» où le gestionnaire d'OF de POF commence par créer les objets conjonctifs ; dans un second temps, les fragments s'instancient et se relient (lors du raffinement, si l'étape est distincte) aux fragments locaux des objets conjonctifs (créés lors de la première étape).

La première approche est adaptée aux applications symétriques, où tous les contextes sont succeptibles de créer (enregistrement dans POF) l'OF. Cette approche est celle typiquement utilisée lors de l'initialisation du système.

La deuxième approche, plus simple à mettre en œuvre, est adaptée aux applications asymétriques, où un contexte à la charge de créer l'OF, les autres s'attachant par la suite.

La suite du chapître concerne la mise en œuvre des Factory dans BOAR-V0.

Les classes *Factory* sont en cours de mise en œuvre.

2 Objets de BOAR-Application

BOAR-Application propose deux classes : Une classe *ReplicatedInteger* représentant la version répliquée d'un entier. Une classe *ReplicatedArray* mettant en œuvre un tableau statique répliqué. Ce dernier est paramétrique (template C++) permettant l'instanciation de tableau de tout type d'élements et de taille quelconque (mais fixe).

Ces deux classes d'objets répliqués reposent sur un ensemble de classes satellites : Leurs classes parentes, mettant en œuvre les caractéristique communes des objets répliqués (cf *Factory* et *Interface* et figure VI.1) et les verrous correspondants (*IntegerVerrou* et *ArrayVerrou*).

La classe abstraite *Interface* sert de classe de base à toutes les classes d'objets répliqués de BOAR. Cette classe (cf figure VI.2) exporte l'interface de concurrence (*Interface::beginActivity()* et *Interface::endActivity()*) et l'interface d'apppel ascendant (*Interface::update()*) utilisée par les objets de réplication. *Interface* dérive elle-même de *Factory*.

Cette approche permet d'effectuer la création et le raffinement d'un objet répliqué en une seule opération utilisateur. La conséquence en est que l'objet répliqué et l'objet Factory sont confondus.

2.1 Les classes *ReplicatedInteger* et *IntegerVerrou*

La classe *ReplicatedInteger* (cf figure VI.3) met en œuvre la version répliqué d'un entier. Les opérations mathématiques usuelles sont mises en œuvre par surcharges des opérateurs (+, -, *, /, =, ==, ++, −, +=, -=). Un opérateurs de conversion **(int)** est fournit. Les méthodes de contrôle de concurrence *ReplicatedInteger::beginActivity* et *ReplicatedInteger::endActivity* sont surchargés et prennent en paramètre un verrou d'entier *IntegerVerrou* .

La classe *IntegerVerrou* met en œuvre des verrous «élémentaires» (un seul granule, cf annexe 1). Les droits d'accès appartiennent à $\mathcal{P}\{READ, WRITE, NONE\}$; le WRITE est exclusif.

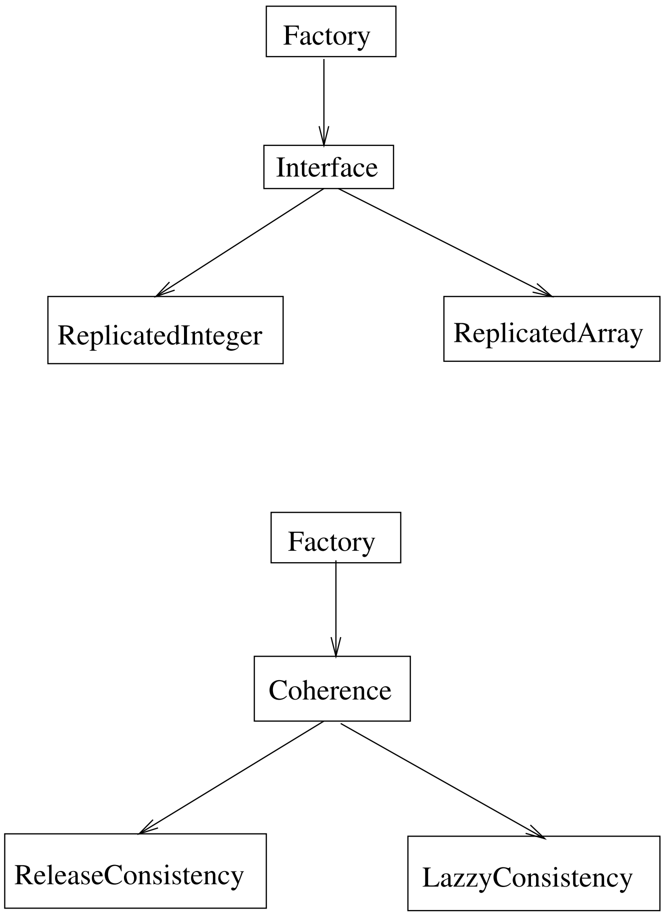

FIG. VI.1 - : *Héritage des classes de BOAR*

2.2 Les classes ***ReplicatedArray*** et ***ArrayVerrou***

La classe *ReplicatedArray* (cf figure VI.4) met en œuvre un tableau répliqué paramétrique. Les opérations usuelles sur les tableaux sont mises en œuvre par surcharges des opérateurs (=, ==, [ ],). Un opérateur de conversion dans le type centralisé est fournit (**Array** $< T >$) . Le type centralisé (template $< classT >$ Array) est lui même mis en œuvre dans ce projet de DEA. Les méthodes de contrôle de concurrence *ReplicatedArray::beginActivity* et *ReplicatedArray::endActivity* sont surchargés et prennent en paramètre un verrou de tableau *ArrayVerrou* .

La classe *ArrayVerrou* met en œuvre des verrous «élémentaires». Néanmoins, un verrouillage fin, d'éléments de tableaux est possible. Un *ArrayVerrou* associe de façon interne chaque élément du tableaux à un verrouillage. La sémantique du verrouillage d'un élément, est celle d'un entier, i.e les droits d'accès appartiennent à $\mathcal{P}\{READ, WRITE, NONE\}$; le WRITE est exclusif. Le constructeur du verrou offre une «syntaxe» permettant le verrouillage d'une tranche de tableau.

```
typdef char * OF_ident\,;

class Interface : Factory {
    Coherence *coherence; // replication manager

friend class Coherence; // upcall. Used by Coherence
    virtual void update(const void *invokeMessage) = 0;

public:
    Interface(OF_ident id); // instance Connective object coherence then bind it

// concurrence interface
    virtual void beginActivity (const Verrou& v) = 0;
    virtual void endActivity (const Verrou& v) = 0;
};
```

FIG. VI.2 - : Interface. *Classe de base des objets répliqués*

3 Objets de BOAR-Réplication

Deux classes de réplication pessimistes sont mis en œuvre. Une classe *ReleaseConsistency* assurant une cohérence au relâchement de verrou (release Consistency). Une classe *LazzyConsistency* assurant une cohérence paresseuse (cf plus bas).

Les caractéristiques de *ReleaseConsistency* sont les suivantes :

- Les verrouillages exercés par les familles *beginOperation/endOperation* et *beginActivity/endActivity* sont identiques et atomiques sur l'OF.
- La diffusion des mises à jour est retardée. Celle-ci est effectuée au relâchement du verrou (endOperation/endActivity).

Les caractéristiques de *LazzyConsistency* sont les suivantes :

- Les verrouillages exercés par la famille *beginOperation/endOperation* sont locaux aux fragments sur lesquels ils sont demandés. Il assurent l'exclusion mutuelle de la copie locale (vis à vis d'éventuels autres clients du fragment, et des autres fragments de l'OF).
- Les verrouillages exercés par la famille *beginActivity/endActivity* sont atomiques sur l'OF. Ils sont précédés d'une synchronisation, assurant la cohérence de l'OF à la prise de verrou.
- La diffusion des mise à jour aux fragments distant est paresseuse. Elle s'effectue de façon asynchrone périodiquement. Une prise de verrou atomique (*beginActivity/endActivity*) induit une diffusion immédiate (précédant l'accord (ACK) de la prise de verrou) des modifications des copies locales.
- Le fragment demandeur d'un verrou atomique est mis-à-jour à l'obtention du verrou (avant le retour d'invocation de *Coherence::beginActivity()*).

```
class ReplicatedInteger : public Interface {
    int *objetDonnee;

    virtual void update(void *invokeMessage); // upcall for coherence

public:
    ReplicatedInteger (char *s, int data = 0);
    ~ReplicatedInteger ();

    ReplicatedInteger& operator ++ ();
    ReplicatedInteger& operator -- ();
    ReplicatedInteger& operator += (const int data);
    ReplicatedInteger& operator -= (const int data);
    ReplicatedInteger& operator = (const int data);
    int operator == (const int data) const;

    operator int () const;

    friend ostream& operator << (ostream& out, ReplicatedInteger& s);

// concurrency interface
    IntegerVerrou *beginActivity (const IntegerVerrou& v);
    void endActivity (const IntegerVerrou * v);
};
```

FIG. VI.3 - : ReplicatedInteger. *Entier répliqué*

L'objet (gestionnaire) de réplication met en œuvre *LazzyConsistency* en réalisant les règles suivantes :

- Le verrouillage atomique est effectué par un classique mécanisme à deux phases : diffusion de la demande, puis attente de l'accord de chacun. Les communications sont supposées fiable (Cette condition est garanti par la run-time *POF*).
- Le dernier gestionnaire de réplication possesseur d'un verrou atomique détient la version la plus récente de l'objet (ou portion de l'objet) considérée. ce statut est connu du gestionnaire.
- Sur une demande de prise de verrou en provenance d'un autre fragment, le gestionnaire de réplication «flushe» la liste des mises à jour qu'il pourrait avoir en suspens, avant de renvoyer l'acquittement.
- Sur réception d'un message distant de mise à jour, le gestionnaire répercute la mise à jour immédiatement sur le fragment (appel ascendant de *Interface::update()*).

Les deux types de cohérence mis en œuvre repose sur des canaux FIFO (assurant l'ordonnancement des mises-à-jour successives et des séquence mises-à-jour/verrouillage atomique).

Les classes *ReleaseConsistency* et *LazzyConsistency* dérive de *Coherency*. C'est

```
class ReplicatedArray : public Interface {
    Array *objetDonnee;

    virtual void update(void *invokeMessage);

public:
    ReplicatedArray (char *s, Array& data);
    ~ReplicatedArray ();

    ArrayElem& operator [] (const int i) const;

    int operator == (const Array& data) const;
    int operator == (const ReplicatedArray& t) const;

    ReplicatedArray& operator = (const Array& data);
    ReplicatedArray& operator = (const ReplicatedArray& t);

    operator Array () const;

    int taille() const {return objetDonnee.taille();}

// concurrency interface
    ArrayVerrou *beginActivity (const ArrayVerrou& v);
    void endActivity (const VerrouReplicatedArray* v);
};
```

FIG. VI.4 - : ReplicatedArray. *Tableau répliqué*

cette architecture qui permet le remplacement dynamique de l'une par l'autre. L'interface des objets de réplications est donnée ci-dessous (figure IV.4).

L'objet de réplication est un OF et hérite donc de la classe *Factory*. L'objet conjonctif utilisé est un canal de diffusion FIFO. Le mécanisme d'instanciation de cet objet est similaire à celui décrit pour les objets répliqués.

On notera que le choix s'est porté implicitement sur un mode de mise à jour de type write-update. Ce choix semble découler naturellement des politiques de cohérence faible. Les bonnes performances obtenues par Orca, et l'étude comparative effectuée par Bal[9], conforte ce choix.

Chapitre VII

Travaux comparables

Ce chapître s'intérresse aussi bien au support de la réplication, qu'à la conception de l'architecture des objets partagés et du support offert quant à la gestion de la concurrence.

Les premiers travaux sur la réplication ont été effectué dans le domaine des bases de données. Initialement, moyen de fiabilité sur bases de données centralisées, la réplication a ouvert la porte à de nouveau gain de performance dans les bases de données distribuées.

De cette origine, les premiers travaux sur les objets partagés (Argus, Avalon, Clouds) ont hérité d'une approche basé transaction. Orca et Munin sont deux systèmes mettant en œuvre la réplication d'objet.

1 Argus

Argus [13] est construit en architecture client/serveur. Un serveur, *Guardian*, encapsule un service (gestion de ressource, en général). Argus assure de facon interne (encapsulé dans le *Guardian*) le support de la concurrence et la tolérance aux pannes.

Un *Guardian* réside sur un seul site, mais peut migrer. Il survit au panne de son site.

L'objet (*Guardian*) proposé par Argus est gros : A un *guardian* peut être associé plusieurs objets et process. Chaque requête (invocation de méthode) de la part d'un client génère une tâche en charge du service dans l'espace du serveur (rpc). Argus propose au guardian des mécanismes de contrôle de concurrence classique (lock). Il apporte l'atomicité à un groupe d'opération par un mécanisme de transaction tolérant les pannes. Il apporte la propriété d'objets stables (persistant, tolérant aux pannes).

La politique (pessimiste), le type de cohérence (forte), la stratégie de mise en oeuvre est imposée. Le contrôle de cohérence est simplifié par le fait que la cohérence est forte et que le serveur n'est pas distribué.

Le modèle propose avant tout un mécanisme de distribution de service, proche de celui de Chorus [14].

2 Avalon

Avalon [15] est un langage associé à une librairie permettant une gestion d'objets partagés en environnement distribué. La librairie est conçue au-dessus de Camelot [16], qui lui offre la gestion de transaction. Plusieurs langage cible, C++ (avalon/C++) et lisp (avalon/Common Lisp), ont servit de base de mise en œuvre.

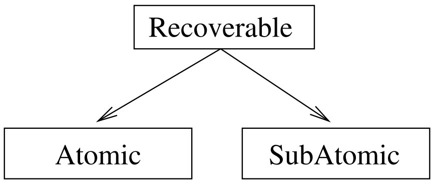

FIG. VII.1 - : *Hiérarchie d'héritage des 3 classes de bases d'avalon*
La classe Recoverable *met en œuvre (via Camelot) la propriété d'atomicité et de toléance aux pannes.* Recoverable *étant classe de base, tout contrôle de concurrence/cohérence impose le coût d'un système transactionnel.*

La figure VII.1 montre la structure d'héritage d'un objet avalon [10]. La classe *Recoverable* apporte la tolérance aux pannes, via la gestion de transaction de Camelot. La classe *Atomic* offre un contrôle de concurrence par verrouillage (*long term lock*); Elle assure la sérialisibilité au niveau transactionnel. La classe *SubAtomic* offre un contrôle de concurrence plus fin, assurant une sérialisibilité au niveau opérationnel(*short term lock*).

Les objets proposés par Avalon sont petits. Le surcoût est néanmoins important du à l'héritage minimum du système transactionnel (La classe *Recoverable* est classe de base).

Avalon est le premier à apporter la notion d'atomicité hybride [17] (*short term lock*). Cette approche est le premier pas vers la distinction entre cohérence et concurrence.

3 Clouds

Clouds [4] est un système d'exploitation support d'objets.

La gestion de la cohérence est assurée par une mémoire virtuelle partagée (DSM) qui assure une cohérence forte. L'algorithme de maintien de cohérence utilise un directory-scheme avec invalidation [18] [19, Fixed Distributed Manager Algorithm, p 333]. Cette mémoire distribuée est persistante.

Un objet Clouds est un espace d'adressage. Un espace d'adressage est découpé en segments. A chaque segment est associé un tas (heap) et une zone de données statiques. Le segment est le granule de gestion du contrôle de concurrence automatique assuré par le noyau Râ. La gestion de concurrence est le fait de deux mécanismes: Une maintenance de verrou au niveau des segments (dont le mapping est définit par l'utilisateur), des threads munies d'attributs indiquant le type d'atomicité souhaité. L'utilisateur dispose de threads sans garantie de cohérence (consistency) *s-threads*, avec cohérence locale à un site *cp-threads* et avec cohérence globale *gcp-threads*.

Tout objet est persistant. L'utilisateur précise un niveau de tolérance aux pannes. Celle-ci est assurée par une réplication couplée avec un système transactionnel (mis en oeuvre par des *gcp-threads*).

Des mécanismes de synchronisation classique sémaphore reste a la disposition du programmeur d'objet.

4 Orca

Orca[9] est un langage objet. Sa mise en œuvre repose sur un compilateur et un environnement d'exécution (run-time system) tournant au-dessus d'AMOEBA [20]

Le seul type de partage offert est la réplication. L'unité de réplication est l'objet. Le système choisit, sur des statistiques effectuées en continu, la meilleure stratégie (Mise à jour/ Invalidation) et la meilleure méthode (réexécution/recopie) de mise à jour. Seul sont partagés les objets hérités des processus parents (passés en paramètre) lors du fork.

5 Munin

Munin [21, 8] est un gestionnaire de mémoire virtuelle distribuée tournant au-dessus de système V [22]. Son originalité consiste en le support de politiques de maintien de cohérence diverses adaptées à la sémantique d'un segment de mémoire partagé. Il permet au programmeur d'associer à un objet, à une portion d'objet ou à un groupe d'objet , un type de partage. Définit par des mots clés[1] associés à la déclarations des objets, ces type de partages ne sont que des suggestions ; celles-ci peuvent être remisent en causes aussi bien par la run-time MUNIN que dynamiquement (appel à des fonction de la librairie de system-service de MUNIN) par le programmeur.

Les types de partages proposés par MUNIN sont : Lecture-Uniquement, Migration, Producteur-Consommateur, Ecriture-concurrentes, Réduction, Résultats. A ces types de partages sont associés des politiques de cohérence se voulant optimales, justifiée par une étude fine et poussée du comportement d'applications réèlles.

Munin est un des premier à mettre en œuvre la politique de cohérence à relâchement de verrou (Release-consistency) [7].

6 Analyse critique

L'objet d'Argus est assez loin de la notion de type abstrait. Son coût important (process, espace d'adressage propre) le rend inadapté au partage de petites entités (type scalaire, petits tableaux, etc).

Comme celui d'Argus, l'objet de Clouds est gros. Il offre une approche très originale de part l'existence d'une mémoire virtuelle partagée et de son modéle object/threads. Cette approche permet un contrôle fin du type de cohérence souhaitée par le programmeur de l'objet partagé. Néanmoins, la cohérence repose, in fine, sur

1. préprocessé au dessus de PRESTO pour la première implémentation.

une cohérence forte au niveau de la mémoire. Ce choix nous parait exclure des types de cohérence paraissant interressant tels qu'une cohérence causale. Le contrôle de concurrence n'est pas exporté et, malgré une mise en oeuvre originale et adaptable, recouvre une sémantique transactionnelle, i.e en marge de l'utilisation qui est faite de l'objet.

Avalon est le seul a exporter l'interface de concurrence, à distinguer la gestion de cohérence du contrôle de concurrence et à adopter une sémantique de concurrence similaire à la nôtre Néanmoins, Le coût de base (transaction) est élevé.

Munin est le seul à proposer de prendre en compte la sémantique de la donnée partagée pour la politique de partage. Néanmoins, l'ensemble des politiques proposées est fixe. Le niveau d'interaction de MUNIN (segment de mémoire) ne permet pas d'étendre les politiques de partages lors de la mise en œuvre de l'objet partagé ; des connaissances relevant du contrôle de concurrence, ou d'une mise en œuvre particulière ne peuvent être pris en compte.

Orca est, dans l'esprit, le plus proche de notre travail. Ses hypothèses de travail,: objets passifs, petits, rapides se préocupant peu de la tolérance au pannes, sont semblables aux nôtre [2]. Orca ne fournit qu'une politique de partage, la réplication. De plus, sa mise en œuvre et sa gestion purement statistique, fait abstraction complète de la sémantique de l'objet, de sa mise en œuvre et de son utilisation. Le partage de données par un arbre d'héritage processus père/processus fils, nous semblent restrictif.

L'importance de la mise en œuvre d'un mécanisme transactionnel, nous parait contradictoire avec un système d'exploitation support d'objets à grain moyen. Ce type de système d'exploitation est celui auquel BOAR est destiné (cf [23]). Les travaux récents [24, 25], soulignant le besoin de structure de contrôle plus polyvalente à juste coût, semblent confirmer ce point de vue.

L'absence d'objets répliqués génériques dans Avalon et Clouds imposent un coût de développement important pour chaque classe utilisateur (Scalaire, tableau, ...).

L'approche bibliothèque est, nous l'espèrons, plus performante (par diminution des appels inter-contexte).

2. L'aspect tolérance aux pannes, journalisation, reste en arrière plan ; Néanmoins, nous espèrons que notre architecture pourra facilement incorporer ces services, «à la carte».

Chapitre VIII

Conclusion

1 Constat

1.1 Originalités du travail

Les quelques idées apportées dans ce travail sont :

- Classe de réplication générique. A une politique de réplication est associé une classe. Une classe d'objet de réplication est partagée par l'ensemble des objets répliqués adoptant la politique de réplication représentée par la classe. La réutilisation est maximale.
- Interface d'accès et typage identique à l'objet version centralisé. Ceci permet la transparence de la réplication (au contrôle de concurrence près).
- Attachement du contrôle de concurrence à l'objet. Le contrôle est obligatoire (mandatory) offrant une sécurite accrue.
- Publication de l'interface de contrôle de concurrence. La puissance du contrôle est maximale : la sémantique de l'application vis-à-vis de l'objet répliqué est exploitée au mieux et contrôlée par l'utilisateur de l'objet répliqué. La sémantique du contrôle de concurrence reste encapsulée dans l'objet répliqué, restant ainsi du ressort de l'objet de réplication.
- Contrôle de concurrence fin, i.e fonction de l'objet répliqué : un arbre binaire répliqué peut verrouiller son sous-arbre gauche par exemple ; Un tableau peut verrouiller une tranche d'élément.
- Distinction nette entre cohérence et concurrence. Ceci donne la souplesse nécessaire pour permettre le support des divers types (politiques) de cohérence. L'architecture accepte les optimisation permise par la connaissance de l'état de concurrence.
- Principe du verrou. (cheval de troie). Encapsulation d'une sémantique dans un objet offrant une interface d'algèbre. Cela permet une prise de décision par un objet générique (de niveau inférieur).

1.2 Parti-pris

Le partis-pris le plus important à consisté en l'approche librairie dans la conception de BOAR (BOAR est constitué par une architecture de classe). L'approche librairie peut théoriquement être langage-indépendante; elle s'intègrent mieux dans une conception système.

L'architecture proposé, lie fortement la sémantique de concurrence au mécanisme de cohérence utilisé par l'objet : La prise de verrou synchronise tous les fragments (Release Consistency), dans le cas de l'objet *ReplicatedArray*. Elle ne permet pas a l'utilisateur de spécifier de façon indépendante un mécanisme de cohérence et une sémantique (ou un mécanisme) de concurrence. Nous pensons que ceci n'est pas un réèl handicap.

1.3 Problèmes

Le manque de maturité des outils utilisés dans le support de la généricité n'a pas permis de mettre en œuvre des classes génériques (templates peu fiables).

Les problèmes de contravariance rencontrés naturellement lors de la conception des verrous ont imposés un mécanisme de type-checking dynamique intégrés à la classe de verrou.

La covariance des types de retour des méthodes virtuelles n'est pas gérée par les compilateurs actuels. Ceci impose la présence de mécanisme de coercition pour toutes les méthodes virtuelles de verrou. Ce problème devrait disparaitre avec des versions ultérieures des compilateurs.

L'optimisation des invocations locales n'a pas été pleinement étudié : le mécanisme de communication entre fragments sur un même site devrait utilisé la mémoire partagée. Une étude doit être engagée pour voir si une telle optimisation peut-être faite et à quel niveau (couche) de BOAR. L'utilisation des familles de POF, doivent permettre de résoudre ce problème de façon transparente pour le programmeur de l'objet de réplication.

2 Perpectives

Les perspectives touchent trois domaines : Le langage, l'architecture de la librairie, la journalisation.

La nature de la librairie, la portabilité et le type de généricité requise doivent faire l'objet d'une étude à court terme.

Les perpectives à long terme sont celles faisant l'objet de la thèse. En particulier, la mise en œuvre de nouveaux objets répliqués, l'évolution de FOG permettant une génération automatique d'objet répliqués, le changement dynamique de politique de partage, et le support de la journalisation.

Nous pensons, à moyen terme, mettre en œuvre un mécanisme de journalisation issus de KitLog, outil de journalisation générique mis en oeuvre dans le projet SOR [26]. Au niveau des objets de réplication, la journalisation apportera la tolérance aux pannes requises par beaucoup d'applications distribuées. Au niveau des objets répliqués, elle apportera un gain de performance quant au partage de grosses structures (e.g multimédia) sur des architectures faiblement couplées en permettant le passage des opérations et leurs paramètres en lieu de la copie modifié de l'objet répliqué.

2.1 Langage

Au fur et à mesure de notre expérience, il apparait de plus en plus probable qu'une approche mixte langage-librairie doit être utilisé. Tant pour des problèmes actuels (interface de groupe, structure répliqués générique ?) que pour l'évolution de BOAR. La génération automatique de classe d'objet répliqués reste en effet un de nos projet majeur.

La mise en œuvre d'un langage permettant de choisir un objet répliqué de BOAR par heuristique n'est envisagé qu'à très long terme (au delà de la thèse).

2.2 Architecture de la librairie

L'aspect interchangeable de l'objet de partage (type de distribution, i.e migration, partitionnement, réplication, serveur) n'a pas été exploré assez profondement. Il est probable, que l'interface de l'objet de réplication va évoluer. L'unification des paradigmes de partages, but de BOAR, devant passer par une interface commune des objets de partages [1]. Ce n'est que lorsqu'une interface générique des objets de partage aura été identifié, qu'une modification dynamique et éventuellement automatique des objets de partage pourra être envisagé.

Annexe A : Le cas des objets complexes

Ce chapitre après une brève approche de l'objet complexe, expose les mécanismes mis en œuvre pour répondre aux problèmes liés a ces objets (principalement la fuite de référence) et les retentissements sur l'architecture des objets répliqués.

Un objet complexe est classiquement représenté par les tableaux, les agrégats de façon plus générale, et les structures arborescentes. Logiquement, un objet complexe est composé d'élément simple (non décomposable). Les langages de programmation moderne (en fait, ceux permettant directement ou non (voir les union en C ou les COMMON en fortran) de manipuler des adresses de variables), offre la possibilité d'isoler un élément d'un objet complexe :

```
/* en C */

/* exemple1 */
int tab[10], *element;
element = &tab[4];
  .
  .
  .
/* exemple2 */
Tableau<int> tab(20);

 tab[i] = 40; // Attention !
  /*
 Ceci est equivalent a :
 int& temp = tab[i]; // Ok, contrôle correct en lecture. [] est méthode
                     // de Tableau. Il n'y a même pas de
                     // lecture donc pas de prise de verrou.

 temp = 40;     // Erreur, temp est un int. pas de contrôle.
 */
```

Ce mécanisme permet d'accèder et de modifier un objet complexe en «courscircuitant la sémantique» de cet objet. Les problèmes qu'il pose au compilateur sont classiques et maitrisés (optimisation en présence d'alias en C, par exemple).

Dans une architecture orienté objet, un certain nombre de mécanisme comme la persistance, la tolérance aux pannes, ou le partage, peuvent être géré par l'objet (architecture de l'objet) lui-même en-lieu d'une librairie ou d'un mécanisme langage. Avalon (resilient container) [10] s'est heurté au problème pour le contrôle de la tolérance aux pannes d'objets complexes.

Le but de BOAR étant d'offrir à l'utilisateur d'un objet répliqué la transparence de la réplication, le mécanisme d'alias doit pouvoir être supporté. Hors d'une approche langage, la seule solution (compatible avec un contrôle de concurrence correct) est de répliquer les objets élémentaires composant un objet complexe. Cette solution est très vraisemblablement coûteuse : toutes les fonction de la famille memcpy, memcmp, etc, par exemple, ne pouvant plus être utilisée, sans passer par une copie [1].

Nous avons décidé de laisser disponible dans la bibliothèque, une version légère de tableaux répliqués, n'assurant pas un contrôle de concurrence sur les objets élémentaires. L'utilisateur d'un tel tableau, peut encore, via les primitives de contrôle de concurrence exporté par tout objet répliqué, assurer un contrôle de concurrence correct. Le surcoût parait un problème uniquement pour les structures d'objets complexes à type de bloc mémoire (*struct*, *union* et tableau en C) bien adaptés aux architecture machines actuelles.

Les changements, important, concernent uniquement les objets de réplications. La suite du chapitre présente les modifications apportées aux objets de réplication associés aux objets élémentaires (objet de réplication «élémentaire») et aux objets complexes (objet de réplication «complexe»).

1 Les objets de réplication élémentaires

Intrinsèquement, l'objet de réplication pourrait être inchangé. Pour des raisons d'optimisation (limitation du nombre de canaux), nous envisagons de multiplexer les canaux de communications. Le multiplexage statique consiste lors de la construction de l'OF à passer comme canal de communication, non pas un objet de BOAR-communication mais l'objet de réplication complexe ; cela implique que ce dernier exporte l'interface correspondante [2]. Le multiplexage dynamique nécessite une modification dynamique des canaux de communication. Un tel mécanisme permettrai de modifier dynamiquement le statut simple/composant d'un objet élémentaire. Cela permettrait d'incorporer dynamiquement des OF répliqués dans un OF répliqués complexe (graphes répliqués d'objets répliqués ou tableaux extensible, par exemple).

2 Les objets de réplication complexes

Un objet de réplication n'a intrinsèquement pas besoin de mettre en œuvre de moyens de communication. Néanmoins, pour les raisons invoqués plus haut, il devrait évolué vers un multiplexeur.

Le changement principal réside dans la gestion du contrôle de concurrence/cohérence (familles *Coherence::beginActivity()*, *Coherence::beginOperation()*). L'objet de réplication complexe doit analyser le verrou reçu, le scinder en granules (correspondant

1. Tout objet répliqué mets en œuvre un opérateur de conversion en la version centralisé de l'objet.

2. Le mécanisme de construction des OF sont en cours de spécification (cf factory [11] et POF [12]).

aux objets élémentaires) et les diffuser aux objets de réplication élémentaires concernés. Cette opération de diffusion consiste en une invocation des objets de réplication élémentaires (via les familles de gestion de cohérence standards) avec les verrous «élémentaires».

L'objet de réplication complexe garantit l'atomicité de la prise de verrou (Evitement des dead-lock reposant sur le caractère complexe d'un objet répliqué). Cette atomicité impose la mise en œuvre de prise de verrou asynchrone, par les objets de réplication élémentaire.

Le caractère générique (vis à vis de la sémantique de l'objet répliqué et celle des objets élémentaires composants) est conservé. On remarquera qu'il impose le sens du flux du contrôle (objet de réplication complexe vers objet de réplication élémentaire).

Annexe B : Les verrous en profondeurs

Ce chapitre donne une description plus précise des verrou. Après une spécification succinte des verrous sous forme d'algèbre de boole, une discussion pratique permettra de comprendre et de faire partager (nous l'espèrons) notre optimisme dans le concept d'objet-verrou.

1 Algèbre de verrou

Formaliser les verrous par une structure d'algèbre (de boole dans notre cas) offre les avantages suivants :

- Une garantie d'opérations stable. Idéalement, cela permet de se passer de la mise en œuvre d'exceptions.
- La complétude des opérations. Ce point est fondamental pour une classe de base. C'est la garantie que les opérations définies par cette classe de base sont suffisantes pour construire n'importe quel type de verrou dérivés.

On notera que la sémantique des opérations n'entre pas en compte. Ce que garanti l'algèbre est que toute mise en œuvre «correcte» (par exemple, l'union de deux verrou renvoie un verrou et ce verrou représente bien l'union des verrous) des opérations apporte la complétude et la stabilité.

Soit $\mathcal{V}$ l'ensemble des verrous associés à un objet répliqué, $(\mathcal{V}, +, -, not)$ forme un algèbre de boole. $+$ et $-$ sont respectivement l'union et l'intersection de verrou. *not* est le complémentaire.

ZERO verrou nul est l'élément neutre de l'union. UN verrou complet est l'élément neutre de l'intersection.

Il est important de noter que, tel que définit, l'union de deux verrous renvoie *toujours* un verrou. La structure de ce verrou (granule, accès, propriétaire) est valide, sa sémantique ne l'est pas obligatoirement :

> Exemple : L'union de deux verrous présentant chacun un droit d'accès en écriture sur un même élément (granule) d'un objet, mais de propriétaires différents retournera un verrou (granule, accès, $\{propritaire1, propritaire2\}$).

La validité d'un granule doit donc être vérifié. Une fonction de test de validité à été rajouté. On remarque que ce problème est indépendant de la structure du verrou ; il est apporté par la sémantique de verrou et ne dépend pas d'un type de verrou particulier. Il est dans la sémantique d'un verrou, de restreindre les droits d'accès à un objet et donc d'imposer le notion de «compatibilité» de verrous.

2 Plus loin avec les verrous

Ce chapître présente les verrous élémentaires et complexes. Ceux-ci sont le pendant des objets de réplication élémentaires et complexes (cf chapitre 2.2).

Un verrou élémentaire est un verrou ne concernant qu'un granule. Lui-même n'a pas de notion de granule ; il est l'objet-verrou géré par un objet de réplication élémentaire (ou simple). Seuls sont donc définis dans ce type de verrou, les types d'accès à l'objet répliqué élémentaire ou simple et le propriétaire correspondant. Sa sémantique est celle d'un objet simple.

Un verrou complexe est un verrou possédant la sémantique d'un objet complexe. Il est géré par un objet de réplication complexe. Les phases d'une opération de verrouillage consistent en :

- Une décomposition du verrou complexe (*VerrouComplexe::split()*). Cette décomposition retourne une liste de structure associant un verrou élémentaire et l'élément de l'objet complexe auquel il s'adresse [1]
- Une demande de verrouillage, à chacun des objets élémentaires concernés. Cette étape nécéssite un appel ascendant à l'objet *Interface* de l'objet répliqué. cet appel retourne l'adresse de l'objet de réplication concerné. L'invocation (*beginOperation()* ou *beginActivity*) est alors réalisé en mode asynchrone.
- En cas de non succès du verrouillage de l'un des objets élémentaires, l'ensemble des verrous obtenus avec succès sont libérés. Selon le mode (asynchrone ou synchrone), soit le verrouillage complexe a échoué, soit il est recommencé.

La mise au point d'un verrouillage asynchrone avec détection de dead-lock est envisagé ultérieurement.

1. Il ne s'agit pas de l'adresse de l'objet de réplication, mais de sa «position» dans l'objet complexe.

Bibliographie

[1] Makpangou (Mesaac) et Brun-Cottan (Georges). – *Unification de paradigmes de partage en environnement réparti.* – Note technique n° SOR–129, Rocquencourt (France), Projet SOR, INRIA, octobre 1992. Présenté au Premier Colloque Africain sur la Recherche en Informatique, Yaoundé (Cameroun).

[2] Makpangou (Mesaac), Gourhant (Yvon), Le Narzul (Jean-Pierre) et Shapiro (Marc). – Fragmented objects for distributed abstractions. *In: Readings in Distributed Computing Systems*, éd. par Casavant (T. L.) et Singhal (M.), pp. 170–186. – IEEE Computer Society Press, juillet 1994.

[3] Makpangou (Mesaac), Gourhant (Yvon) et Shapiro (Marc). – BOAR: A library of fragmented object types for distributed abstractions. *In: Proceedings of the 1991 International Workshop on Object Orientation in Operating Systems.* – Palo Alto, CA (USA), octobre 1991.

[4] Dasgupta (Partha), LeBlanc Jr. (Richard J.), Ahamad (Mustaque) et Ramachandran (Umakishore). – The Clouds distributed operating system. *IEEE Computer*, vol. 24, n° 11, novembre 1991, pp. 34–44.

[5] Eggers (Susan J.) et Katz (Randy H.). – A characterization of sharing in parallel programs and its application to coherency protocol evaluation. *In: Proceedings of the 15th International Symposium on Computer Architecture*, pp. 373–382. – Honolulu (Hawai), 1988.

[6] Popek (Gerald J.), Guy (Richard G.), Page, Jr. (Thomas W.) et Heidemann (John S.). – Replication in Ficus distributed file systems. *IEEE Computer Society Technical Committee on Operating Systems and Application Environments Newsletter*, vol. 4, n° 3, 1990, pp. 24–29.

[7] Gharachorloo (Kourosh), Lenoski (Daniel), Laudon (James), Gibbons (Phillip), Gupta (Anoop) et Hennessy (John). – Memory consistency and event ordering in scalable shared-memory multiprocessors. *In: Proceedings of the 17th Annual International Symposium on Computer Architecture.* ACM SIGARCH, pp. 15–26. – Seattle, WA (USA), mai 1990.

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

[9] Bal (Henri E.) et Kaashoek (M. Frans). – *Experience With Distributed Progamming in Orca.* – Rapport technique n° IR-200, Amsterdam (The Netherlands), Dept. of Math. and Computer Science, septembre 1989.

[10] Detlefs (David), Herlihy (Maurice) et Wing (Jeannette). – Inheritance of synchronization and recovery properties in Avalon/C++. *Computer*, vol. 21, n° 12, décembre 1988, pp. 57–69.

[11] Dickman (Peter) et Makpangou (Mesaac). – A refinement of the fragmented object model. *In : Proceedings of the Second International Workshop on Object Orientation and Operating Systems*. pp. 230–234. – Dourdan (France), octobre 1992.

[12] Makpangou (Mesaac). – *Fragmented Object Support Platform*. – Note technique n° SOR–128, Rocquencourt (France), Projet SOR, INRIA, octobre 1992. Submitted to BROADCAST open workshop oct 1993.

[13] Liskov (Barbara), Curtis (Dorothy), Johnson (Paul) et Scheifler (Robert). – Implementation of Argus. *In : Proceedings of the 11th ACM Symposium on Operating Systems Principles*. ACM. – Austin TX (USA), novembre 1987.

[14] Rozier (M.), Abrossimov (V.), Armand (F.), Gien (M.), Guillemont (M.), Hermann (F.), Kaiser (C.), Leonard (P.), Langlois (S.) et Neuhauser (W.). – *Overview of the Chorus Distributed Operating System*. – Rapport technique n° CS/TR-88-7, Montigny-le-Bretonneux (France), Chorus Systèmes, juin 1988.

[15] Herlihy (Maurice P.) et Wing (Jeannette M.). – Avalon: Language support for reliable distributed systems. *In : Proceedings of the 17th International Symposium on Fault Tolerant Computing*. IEEE, pp. 89–94. – Pittsburgh PA (USA), juillet 1987.

[16] Carnegie-Mellon University, Pittsburgh PA (USA). – *Guide to the Camelot Distributed Transaction Facility: Release 1*, mai 1988. A. Z. Spector and K. R. Swedlow, editors.

[17] Herlihy (Maurice. P.) et Weihl (William E.). – Hybrid concurrency control for abstract data types. *Journal of Computer and System Sciences*, vol. 43, n° 1, août 1991, pp. 25–61.

[18] Dasgupta (Partha), Ananthanarayanan (R.), Menon (Sathis), Mohindra (Ajay) et Chen (Raymond). – Distributed programming with objects and threads in the Clouds system. *Computing Sytems*, vol. 4, n° 3, 1991, pp. 243–276.

[19] Li (Kai) et Hudak (Paul). – Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, vol. 7, n° 4, novembre 1989, pp. 321–359.

[20] Mullender (Sape J.), van Rossum (Guido), Tanenbaum (Andrew S.), van Renesse (Robert) et van Staveren (Hans). – Amoeba: A distributed operating system for the 1990s. *Computer*, vol. 23, n° 5, mai 1990, pp. 44–53.

[21] Carter (John B.), Bennett (John K.) et Zwaenepoel (Willy). – Implementation and performance of Munin. *Operating Systems Review*, vol. 25, n° 5, octobre 1991, pp. 152–164.

[22] Cheriton (David R.). – The V distributed system, mars 1987.

[23] Shapiro (Marc). – Soul: An object-oriented OS framework for object support. *In : Workshop on Operating Systems for the Nineties and Beyond*. pp. 251–255. – Dagstuhl Castle, Germany, juillet 1991.

[24] Birman (Kenneth P.). – *Maintaining Consistency in Distributed Systems.* – Rapport technique n° TR91-1240, Ithaca NY (USA), Dept. of Computer Sc., Cornell U., novembre 1991.

[25] Herlihy (Maurice P.) et Wing (Jeannette M.). – *Linearizability: A Correctness Condition for Concurrent Objects.* – Rapport technique n° CMU-CS-88-120, Pittsburgh PA (USA), Department of Computer Science, Carnegie-Mellon University, mars 1988.

[26] Ruffin (Michel). – Kitlog: A generic logging service. *In : Proceedings of the 11th Symposium on Reliable Dist. Syst.*, pp. 139–146. – Houston TX (USA), octobre 1992.