Object management in large-scale distributed systems

Marc Shapiro

Université Paris VI — Pierre et Marie Curie

U.F.R d'informatique

Discipline : informatique

Thèse pour l'habilitation à diriger les recherches

La gestion des objets dans les systèmes répartis de grande échelle

Marc Shapiro
Senior Researcher, Microsoft Research, Cambridge UK
Directeur de Recherche INRIA

Version du 20 nov. 2002

Soutenue le 12 décembre 2002 à l'Université Paris VI, Pierre et Marie Curie, devant le jury suivant :

Rapporteurs	MM.	Sacha Krakowkiak
		André Schiper
		Peter van Roy
Examinateurs	MM.	Claude Girault
		Jean-Jacques Lévy
		Jean-Bernard Stefani
		Andy S. Tanenbaum

La gestion des objets dans les systèmes répartis de grande échelle

Résumé

Les systèmes informatiques répartis prennent une importance toujours croissante, mais restent néanmoins complexes à comprendre à programmer. Les difficultés pratiques et théoriques s'exacerbent à mesure que l'échelle des systèmes s'accroît. Notre recherche a pour but de faciliter le partage de l'information dans de tels systèmes, sans renoncer à leur puissance.

Avec le système SOS, nous avons abordé l'angle génie logiciel, par une structuration en objets partagés répartis, dits fragmentés. Ensuite, nous avons étudié les références réparties et le ramasse-miettes avec le mécanisme des CPSS. Enfin nous avons étudié le ramasse-miettes et la persistance dans une mémoire répliquée avec Larchant et PerDiS. Actuellement nous proposons un algorithme de réconciliation appelé IceCube.

Notre axe actuel porte sur le problème central du partage d'information : la réplication des données partagées et la cohérence des réplicats. Dans la réplication optimiste, le contrôle de concurrence est paresseux et la cohérence est rétablie après coup. Nous exposerons une nouvelle théorie de la «cohérence inéluctable» (eventual consistency). Elle est basée sur deux primitives très simples permettant d'exprimer les relations habituelles comme la transaction, le conflict, ou la dépendance causale. Nous montrons les invariants d'intégrité locales et globales nécessaires à la cohérence inéluctable. L'invariant global est très fort et difficile à garantir.

Nos résultats sont d'ordre pratique, architectural et algorithmique, recouvrant la structuration des systèmes répartis, les mécanismes d'exécution, de désignation, de migration et de ramasse-miettes, de liaison flexible, de persistance, et de réplication pessimiste et optimiste pour le partage des données. Les travaux présentés ont été réalisés en tant que responsable scientifique du projet SOR de l'INRIA Rocquencourt, puis comme «Senior Researcher» du laboratoire Microsoft Research à Cambridge, Royaume-Uni.

Mots-clefs : systèmes répartis, objets répartis, partage de données, données répliquées, cohérence, contrôle de concurrence optimiste, ramasse-miettes réparti.

Object management in large-scale distributed systems

Abstract

Despite being widespread, distributed systems remain hard to understand and to program. Our research aims at easing information sharing in large scale systems, without giving up their power.

The first part of the thesis provides some historical perspective. We first used a software engineering angle, structuring distributed systems as distributed shared objects called Fragmented Objects. Next, we focused on distributed referencing and garbage collection with our Stub-Scion Pair Chains mechanism. This was later generalised, as Larchant and PerDiS, to garbage collection and persistence, in replicated shared memories. Most recently we have been working on the IceCube reconciliation engine.

My current focus is on consistency of mutable replicated shared data. In optimistic replication, concurrency control is lazy and consistency is restored after the fact. I present a new theory of eventual consistency based on two simple primitives that can encode the usual relations such as transactions, conflicts, and causal dependence. I exhibit local and global correctness invariants for eventual consistency. The global invariant is quite strong and hard to ensure.

We present practical, architectural and algorithmic results, in the areas of systems structure, structuring mechanisms, identification, migration, garbage collection, flexible binding, persistence, and pessimistic and optimistic replication for data sharing. This work was performed at INRIA Rocquencourt and Microsoft Research, Cambridge, United Kingdom.

Keywords : Distributed systems, distributed objects, shared data, replicated data, consistency, optimistic concurrency control, distributed garbage collection.

Remerciements

Je tiens à remercier en premier lieu le professeur Claude Girault, de l'université Pierre-et-Marie-Curie (Paris 6), pour m'avoir encouragé très tôt dans ma carrière, pour avoir soutenu mon projet de recherche, et m'avoir proposé de l'enseigner aux étudiants du DEA Systèmes Informatiques.

C'est grâce à la confiance que m'ont accordée M. Jean-François Abramatic, à l'époque responsable du GIP SM-90, M. Marc Guillemont, responsable du projet Chorus, et Mme. Anne Schroeder, directeur de l'Unité de Recherche de l'INRIA Rocquencourt, que j'ai pu rentrer à l'INRIA, et développer cette recherche. Enfin, M. Roger Needham, directeur du laboratoire Microsoft Research à Cambridge (Royaume-Uni) m'a fait confiance pour développer le Cambridge Distributed Systems Group (Camdis). Je les en remercie chaleureusement.

Cette recherche n'aurait jamais abouti sans le travail et les apports de tous les membres du projet SOR de l'INRIA et Camdis à Microsoft Research : doctorants, stagiaires, ingénieurs, visiteurs, et collaborateurs extérieurs. Ils sont trop nombreux pour les remercier tous nommément, mais je tiens à citer dont j'ai encadré les doctorats, et dont l'apport a été essentiel : Mesaac Makpangou [43], Sabine Habert [27], Yvon Gourhant [26], Michel Ruffin [53], Daniel Edelson [17], David Plainfossé [48], Hervé Soulard [63], Paulo Ferreira [19], Julien Maisonneuve [39], Georges Brun-Cottan [12], Aline Baggion [5], Xavier Blondel [10], Fabrice le Fessant [33], Nicolas Richer [52].

Je tiens à remercier MM. Claude Girault, Sacha Krakowiak, André Schiper, Jean-Jacques Lévy, Jean-Bernard Stefani, Andy Tanenbaum et Peter van Roy d'avoir accepté de participer à ce jury.

Enfin, une mention spéciale pour Zoé et Léa, pour leur amour, et pour leur patience avec mes humeurs et mes horaires...

Table des matières

Chapitre 1

Introduction

1.1 Les systèmes informatiques répartis

Un système informatique réparti est un ensemble d'ordinateurs, interconnectés par un réseau de communication, et exécutant des programmes en commun. Ces programmes incluent aussi bien des composants de système d'exploitation, par exemple un système réparti de gestion des fichiers, que des applications parallèles, par exemple la recherche de sous-séquences communes dans une base de génomes, que des applications de partage de l'information, par exemple le World-Wide Web [7] ou un collecticiel de CAO interactive.

Les systèmes répartis, objets de recherche depuis les années 1970, sont devenus réalité industrielle avec l'émergence des réseaux de stations de travail et l'interconnexion à grande échelle par Internet. Le terme recouvre un grand nombre de réalités disparates, depuis les machines multiprocesseur à mémoires disjointes jusqu'aux réseaux de processeurs embarqués, en passant par le World-Wide Web. Nous nous concentrons ici sur les réseaux d'ordinateurs faiblement couplés par un réseau de communication généraliste, excluant donc les multiprocesseurs et les applications parallèles du champ de cette recherche.

1.2 Problématique des systèmes informatiques répartis de grande échelle (SIRGE)

Dans les systèmes de petite échelle les problèmes de base de la communication, du nommage, et du partage de ressources peuvent être considérés à peu près résolues [64]. L'étude devient plus intéressante lorsqu'on s'intéresse aux *Systèmes Informatiques Répartis de Grande Échelle (SIRGE,* en anglais *Large-Scale Distributed Computing Systems, LSDCS).* Ici, la latence est élevée et très variable, les pannes sont relativement fréquentes, et l'hétérogénéité est la règle. Dans les SIRGE, nous nous focalisons sur les problèmes «système» de haut niveau posés par le *partage de l'information.*

Ces questions ont constitué notre sujet de recherche au sein du projet SOR (Systèmes d'objets répartis) de l'INRIA Rocquencourt [6] puis dans le Cambridge Distributed Systems Group (Camdis) chez Microsoft Research.

Un exemple de SIRGE sera le réseau interne de Microsoft. Toutes ses divisions à l'échelle internationale utilisent un système d'informations commun, qui permet

à chacun d'accéder à n'importe quel document (fichier, courrier, page Web, base de données, etc.) quelle que soit sa localisation. Les latences d'accès peuvent être très grandes, car il y a peu de réplication et encore moins de transparence. Il n'est pas rare pour un utilisateur en Europe d'être victime d'une panne située aux USA.

Un autre exemple sera le réseau Akamai [2], un «réseau de transport de contenus» (Content Delivery Network ou CDN). Il s'agit d'un ensemble de serveurs implantés à des emplacements stratégiques à travers le monde, et qui répliquent les pages Web de leurs abonnés. Contrairement à l'exemple précédent, Akamai est conçu spécialement pour masquer les pannes et pour accélérer l'accès par un système de réplication des données.

La différence essentielle entre ces deux exemples est que le premier permet de gérer les données mutables, dont la cohérence est critique, alors que celles d'Akamai sont surtout en lecture, et leur cohérence n'est pas critique.

Les SIRGE se heurtent à leurs limitations intrinsèques, dues en premier lieu aux latences élevées et variables de message, aux pannes de réseau et de machine. Celles-ci exposent les SIRGE à une limitation théorique fondamentale, l'impossibilité du consensus [24], qui explique la difficulté de maintenir la cohérence. Une autre difficulté, d'importance croissante, concerne les problèmes de sécurité. Ces problèmes impactent directement la possibilité de partager l'information, en ralentissant ou en empêchant l'accès à celle-ci.

1.3 Nos travaux : le partage de l'information dans les SIRGE

Afin d'uniformiser le vocabulaire, nous supposerons que l'information est structurée en objets *objets*. Le programmeur d'application peut créer des objets de granularité et de type quelconques. Un objet est supposé passif, c'est-à-dire que la notion de processus est orthogonale à celle d'objet. Un programme d'application peut manipuler tout objet dont il possède la référence, aux limitations près que donnent ses droits d'accès et le contrôle de concurrence.

1.3.1 Systèmes d'objets répartis et objets fragmentés

Du point de vue «système» ce modèle pose un certain nombre de problèmes intéressants. Dans l'ordre chronologique, notre recherche s'est focalisée sur les points suivants. Dans un premier temps, de 1976 à 1990 environ, nous avons proposé une démarche structurée et systématique à la programmation des systèmes répartis, fondée sur la notion d'Objet Fragmenté (OF). Ce modèle a un double intérêt, comme outil de génie logiciel (structuration des systèmes) et comme outil de sécurité (représentation d'une portée de protection), bien que ce deuxième aspect ait été peu traité. Pour les besoins des OF nous avons inventé la notion de *mandataire (proxy)*, devenu ensuite universel dans les systèmes répartis, en premier lieu le World-Wide Web [7]. Le modèle des OF a été mis en œuvre et largement développé dans le système SOS. Ce dernier comportait d'autres aspects novateurs, comme un mécanisme très général de migration d'objets. Le chapitre 2 porte sur les objets fragementés et SOS.

1.3.2 Ramasse-miettes et persistance des objets

De 1990 à 1999 environ, nous nous sommes concentrés sur les mécanismes automatiques de partage et de persistance des objets. Pour cela il fallait étudier de près les mécansismes de références en relation avec le ramasse-miettes réparti. Cette étude a eu lieu en deux temps. D'abord les Chaînes de Paires Souche-Scion (CPSS) sont un mécanisme de référence répartie et d'invocation distante, permettant le ramasse-miettes, dans un modèle de système réparti classique (à communication par message). Les CPSS se distinguent des mécanismes similaires par leur simplicité, leur efficacité, et leur tolérance aux pannes. Ce travail est rapporté dans le chapitre 3.

Dans le modèle plus moderne de mémoire partagée, ce qui inclut les systèmes à cache ainsi que ceux à mémoire répliquée ou persistante, il fallait un mécanisme différent. Nous avons développé un mécanisme de références et de ramasse-miettes, appelé Larchant, qui reste sûr quelles que soient les propriétés de cohérence et de contrôle de concurrence d'une mémoire partagée particulière. Ce modèle a été mis en œuvre dans le cadre du projet européen Esprit de «recherche à long terme» PerDiS, dont j'ai été l'organisateur de 1997 à 2000. Les aspects recherche de ces travaux sont présentés dans le chapitre 4.

Un avantage apparent du modèle à mémoire partagée constitue sa principale limitation. Ce modèle plaît pour sa propriété de *transparence* : la désignation et l'invocation apparaissent identiques à un système centralisé parallèle. Cela a l'avantage de cacher au programmeur certaines propriétés indésirables des SIRGE. Mais en même temps cela dissimule la puissance constitutive de la répartition (ce qui est contraire aux principes de conception de système préconisés par Lampson [32]) et implique une perte de contrôle de la part du programmeur.

1.3.3 Réplication optimiste et réconciliation

Aussi, nos travaux récents utilisent un modèle de mémoire partagée non transparente. Il s'agit de la réplication «optimiste» ou «paresseuse». Les données partagées sont répliquées, mais les réplicats ne sont pas supposés immédiatement cohérents entre eux ; ainsi le contrôle de concurrence et de cohérence est retardé et explicite.[1] Ce modèle a de nombreux avantages. Le premier est la disponibilité : un programme peut accéder aux objets partagés même quand les réplicats distants sont inaccessibles. Ceci convient bien à l'informatique nomade (un PC mobile ne connaît qu'une connectivité intermittente). Cela correspond par ailleurs bien aux habitudes de travail de l'ingénierie coopérative, par exemple du développement de code, où chaque ingénieur a besoin d'un minimum d'isolement. Les mêmes principes sont à l'œuvre dans certains protocoles à haute performance, la synchronisation distante étant trop coûteuse. Enfin, le contrôle de concurrence se faisant à posteriori, il est plus aisé pour l'application de le contrôler.

Depuis 2000, nous nous intéressons donc à la réconciliation des réplicats ayant divergé, c'est-à-dire au contrôle de concurrence à posteriori. L'approche, développée dans notre système IceCube et rapportée au chapitre 5, diffère des travaux précédents sous plusieurs aspects. IceCube est un système d'usage général, permettant l'exécution d'applications diverses, mais paramétré par la sémantique de celles-ci. L'interface d'IceCube permet aux applications d'exporter des informations sémantiques minimales au système, et permet au système contrôler l'exécution des applications. Les informations sémantiques en question sont très simples et générales.

[1] «Réplicat» est la francisation du terme anglais *replica*. Il serait peut-être plus juste de dire «duplicata» [44], ou de former un néologisme comme «réplicata» ou «multiplicata» sur le même modèle.

1.3.4 Modèle et correction de la réplication optimiste

Durant l'exposé de soutenance, nous comptons rapporter nos travaux les plus récents. Partant des primitives utilisées dans IceCube, nous développons une théorie de la réplication et de la cohérence. Cette théorie permet d'expliquer les différences fondamentales entre les différentes politiques de réplication. Elle exprime de façon simple qu'un système de données répliqués est correct, c'est-à-dire vivace, sûr, et durable. S'agissant de travaux encore en cours, ils ne trouvent pas place dans ce mémoire.

1.3.5 Autres travaux

Pour des raisons de place, ce mémoire fait l'impasse sur quelques travaux dont l'importance n'est pourtant pas négligeable.

Notre recherche doctorale portait déjà sur le génie logiciel et la structuration des systèmes répartis. Notre solution d'alors, basée sur le langage CSP, ne présente plus qu'un intérêt historique [55].

Nous avons collaboré avec les chercheurs de Chorus Systèmes. Cette collaboration a donné lieu à un article commun, présenté à la prestigieuse conférence SOSP, sur l'architecture de mémoire virtuelle Chorus [1]. Par la suite nous avons conçu en commun le système réparti à objets Cool [28].

Généralisant les mécanismes de SOS, nous avons développé des modèles plus généraux d'objets fragmentés [42, 15, 16] ainsi que des mécanismes de liaison indépendants du système d'exploitation [41, 40, 57].

À la suite des travaux sur les CPSS, nous avons conçu, avec Fabrice le Fessant, plusieurs protocoles avancés de détections de cycles de miettes répartis [23, 61, 34].

Chapitre 2

Les objets partagés fragmentés et le système SOS

Chronologiquement (de la thèse à 1990 environ), mon premier axe a été la conception structurée appliquée aux systèmes répartis. Rappelons qu'à l'époque, la recherche en systèmes se préoccupait principalement, d'une part d'optimiser les primitives de communication et les noyaux, et d'autre part de promouvoir la transparence. Prenant en compte la spécificité du réparti et la difficulté de les programmer, je réfléchissais pour ma part aux moyens de composer les primitives différentes, et d'encapsuler la gestion des non-transparences. C'est ainsi que j'ai proposé le *principe du mandataire (Proxy Principle)* [56] et les Objets Fragmentés.

2.1 Structure et encapsulation dans les systèmes répartis : le principe du mandataire

Notre article de 1986 «Structure and Encapsulation in Distributed Systems : the Proxy Principle» [56] expose deux grandes idées. La première est le concept de mandataire (*proxy* en anglais), objet local représentant un service distribué de façon transparente, tout en mettant en œuvre des fonctions spécifiques de l'objet. C'est ici que ce concept de *proxy* a été exposé pour la première fois.

L'autre grande idée est celle d'une structuration en objets répartis ou *fragmentés* (OF).[1] Les divers morceaux constituant un service réparti sont réunis dans un seul objet réparti. Ceci sépare bien l'interface externe (communication entre clients et service) et interne (entre composants du service). Pour un client, l'OF est une boîte noire, la distribution est cachée, et l'accès est restreint et bien typé. L'intérieur de l'OF constitue un sous-réseau de communication privilégiée ; ses protocoles internes (éventuellement complexes) sont dissimulés derrière l'interface.

2.2 Principe du mandataire et objets fragmentés

Le «principe du mandataire,» illustré par la figure 2.1, s'énonce comme suit : *Afin d'utiliser un service, un éventuel client doit d'abord acquérir un mandataire*

[1] Cette dénomination a été été adoptée postérieurement à cet article, bien que le concept y soit déjà présent.

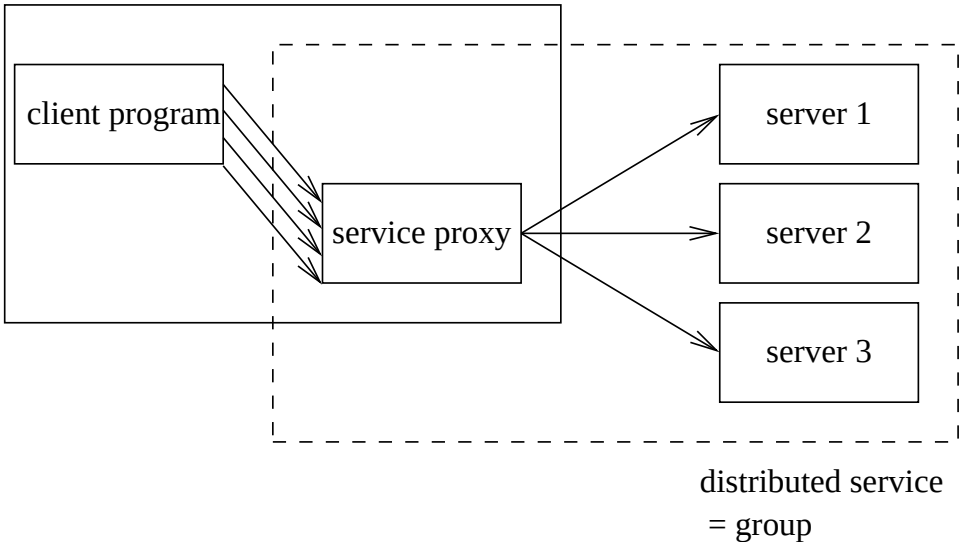

FIG. 2.1 – Un mandataire pour un service réparti

pour ce service ; le mandataire constitue sa seule interface visible au service. Un client ne communique qu'avec son mandataire. Le mandataire, ainsi que les objets qu'il représente, ensemble forment un seul objet fragmenté, dont la décomposition n'est pas visible par les clients. Un éventuel protocole de communicaton de haut niveau reste interne à l'objet fragmenté ; la puissance non-transparente du réseau se gère donc de façon interne à l'OF.

Un objet fragmenté possède deux aspects : une interface externe et une représentation interne. L'interface est le seul moyen d'accès des clients de cet objet, qui sont donc découplés des choix de mise en œuvre interne. Au contraire d'un objet centralisé, monolithique, la représentation d'un objet fragmenté comprend des *fragments* répartis sur tous les sites concernés. Souvent ces fragments remplissent de façon redondante la même fonction, chacun sur son site : par exemple sur un site X et un site Y peuvent être mis en cache les données récemment utilisées sur X et Y, qui peuvent être les mêmes ou différer selon l'historique récent de ces sites. Les fragments étant protégés par l'interface, la répartition reste transparente pour les clients. Les fragments communiquent entre eux par une interface de groupe, privilégiée.

2.3 Quelques exemples

Nous reprenons ici quelques-uns des exemples proposés dans l'article original.

2.3.1 Accès fichier

Le premier exemple concerne la structuration à un système de fichiers similaires à Unix.[2] Un fichier est un objet dont les méthodes sont **read, write, dup** et **close.** Des objets plus spécialisés, comme les sockets, peuvent étendre cette interface standard.

Maintenant supposons que nous voulons accéder au fichier `/users/shapiro/sthg`. La séquence d'ouverture localisera l'emplacement matériel correspondant à ce nom, et démarrera le pilote correspondant :

[2] Il ne faut pas oublier qu'en 1986 l'accès réparti aux fichiers représentait encore une nouveauté.

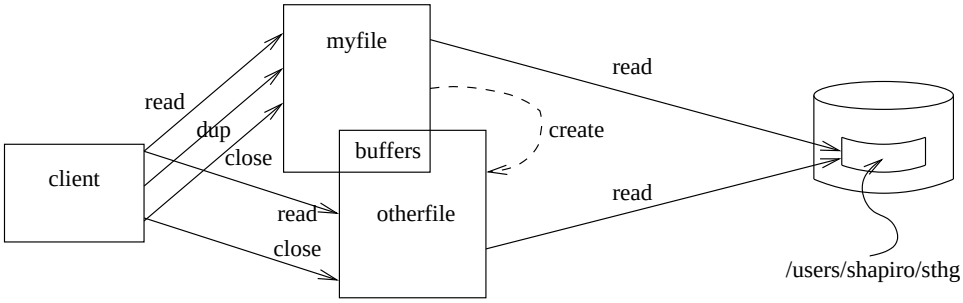

FIG. 2.2 – Fichier accédé comme un objet fragmenté.

file myfile ;
myfile = nameservice.open("/users/shapiro/sthg", "r") ;

La méthode `open` de l'objet système `nameservice` sera invoquée, ce qui retournera un objet gestionnaire de cette localisation. Ce dernier objet est migré vers le contexte du demandeur sous le nom `myfile` et encapsule le pilote et toutes les données de localisation. Ceci reste vrai que le fichier soit local ou distant. Dans ce dernier cas le mandataire s'occupera d'une part de la mise en cache local, et d'autre part avec le pilote distant.

2.3.2 Jeu réparti

Un exemple plus complexe, celui d'un jeu réparti de bataille spatiale, montre comment un objet fragmenté peut utilement encapsuler une communication de groupe. Ceci est illustré par la figure 2.3.

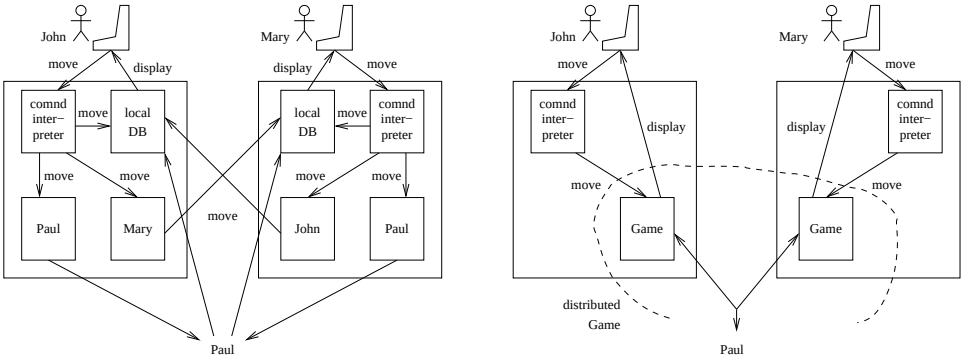

FIG. 2.3 – Communication de groupe dans un jeu réparti.

Les trois joueurs se nomment Paul, John and Mary. Le programme de chaque joueur maintient une base de données des localisation, direction, vitesse et armement de chaque vaisseau spatial. À l'écran le contenu de cette base est affichée. Normalement les positions sont extrapolées des précédentes à chaque top d'horloge, sans trafic réseau. C'est seulement quand un joueur prend une initiative que les nouvelles coordonnées et vélocités doivent être envoyées aux autres sites. Comme le montre la figure l'objet fragmenté jeu peut aussi bien encapsuler une communication point-à-point que multipoint.

Étant donné qu'un message réseau ne peut être envoyé que par un autre membre du même groupe, le jeu peut utiliser un encodage spécialisé et efficace, et il n'est pas nécessaire de vérifier les droits d'accès de l'envoyeur.

2.4 Propriétés des OF

Par rapport aux approches existantes et plutôt ad-hoc de programmation répartie alors utilisées, l'objet fragmenté a les avantages suivants :

1. *Encapsulation.* Le service est une boîte noire, accessible seulement par l'intermédiaire de mandataires. Sa structure interne n'est pas visible.
2. *Localité.* L'OF permet une certaine forme de transparence réseau, car tout accès est local. Certaines requêtes peuvent être satisfaites directement par le mandataire, et les données mises en cache localement. L'état courant de la connexion peut être mémorisée chez le client, ce qui permet au service d'être sans état.
3. *Protocole d'accès.* Le mandataire assure le caractère correct du protocole d'accès (par exemple assure un ordonnancement correct entre requêtes, acquittements, verrouillage, et déverrouillage d'une ressource).
4. *Capacité logicielle.* Le mandataire peut assurer le contrôle d'accès, vérifier la validité des arguments, ou le droit du client d'exécuter les opérations ; au contraire des mécanismes existants il fait cela de façon entièrement programmable.
5. *Souche de communication.* Un mandataire fait office de souche [8] ; il emballe et déballe les données pour l'accès réseau.
6. *Communication vérifiée.* Un mandataire est créé par le service qu'il représente. Il en découle que toute communication reçue par un service provient d'un partenaire digne de confiance, ce qui permet de simplifier le serveur.
7. *Encapsulation de protocole.* Le protocole entre client et service est entièrement encapsulé à l'intérieur de l'OF correspondant. Il en découle qu'il n'est pas nécessaire de vérifier ou de négocier ce protocole (si l'on fait abstraction des fautes et des adversaires éventuels). On pourra profiter des avantages du réparti et masquer ses inconvénients ; le réseau reste transparent au client mais le mécanisme de transparence n'est pas vissé.

2.5 Propriétés d'un noyau pour les objets fragmentés

En même temps que le principe du mandataire, l'article posait les bases de mise en œuvre d'un système réparti autour d'un noyau pour les objets fragmentés.

Le noyau doit fournir les propriétés suivantes. Un mandataire est toujours local à son client. Toute interaction avec le service passe par le mandataire. Du point de vue du client, le mandataire est indistingable de l'objet fragmenté lui-même. Un mandataire est visible du client mais la représentation du mandataire en est caché ; à son tour, le mandataire a la visibilité des autres fragments. Toute forme de communication autre que l'appel de méthode locale est protégée et n'est permise qu'entres fragments d'un même objet. Ceci assure que toute communication interne est d'abord filtrée par un représentant valide de l'OF.

Le noyau d'OF doit donner aux mandataires les propriétés de véritable *capacités* logicielles [18, 46] protégeant la ressource répartie : un mandataire d'un OF ne doit pouvoir être créé que par cet OF, et la communication à l'intérieur de l'OF être réservée aux mandataires.

2.6 Le système SOS

Notre article *SOS : An Object-Oriented Operating System — Assessment and Perspectives* parut en 1989 dans Computing Systems [59]. Il résumait les leçons apprises du développement du système SOS, système d'objets réparti spécifiquement conçu pour la programmation par objets fragmentés.

SOS propose des mécanismes pour la mise en œuvre des objets fragmentés, et est lui-même structuré en OF. SOS contient en outre des mécanismes génériques pour l'invocation, l'identification, la migration et le stockage des objets. SOS constitue un véritable système d'exploitation, car ces services sont complets, de bas niveau, génériques, indépendants d'un langage particulier, et efficaces.

2.6.1 Les Objets Elémentaires

L'élément de base du système est un objet dit *élémentaire* (OE). Un OE est un objet langage, et l'interface entre OE est supposée vérifiée par le compilateur. Un OE est localisé dans un seul contexte. Le système met en œuvre un mécanisme de migration des OE entre contextes.

Un OE possède un identificateur unique, appelé *OID concret*. Il possède aussi un nombre quelconque d'OID de groupe, identifiant les OF dont il fait partie. Un OE possède aussi un nombre quelconque de *prérequis*. Nous verrons sous peu l'importance des prérequis dans la migration : la migration d'un OE vers un contexte n'est finalisée que lorsque tous ses prérequis sont eux-mêmes finalisés.

2.6.2 Les Objets Fragmentés

Un OF est un objet réparti logiquement unique composé d'un groupe d'objets dits *fragments*. Les fragments d'un OF communiquent entre eux de façon privilégiée. Un OF peut se créer et s'ajouter de nouveaux fragments, et le migrer d'un contexte (ou espace d'adressage) à un autre. L'appartenance à un OF est préservée par la migration, ce qui permet à un objet fragmenté de s'accroître par essaimage. Un OF est identifié par un OID de groupe, et un OE appartient à un OF s'il en possède l'OID de groupe.

Conceptuellement, un OF constitue un domaine de protection, dans lequel on entre par invocation d'un mandataire local. Un OE peut invoquer un autre OE du même OF (c'est-à-dire ayant un OID de groupe en commun) par les mécanismes classiques, par exemple appel de procédure distante sur un canal de communication non typé. Par contre, deux OE qui ne sont pas membres d'un même OF peuvent communiquer uniquement s'ils ont un mandataire dans leur propre contexte, et uniquement via son interface. Ainsi, la communication inter-contextes est fortement typée, bien qu'elle emprunte des canaux non typés, et sans que SOS n'ait à proprement parler de notion de type.

Nous classifions les fragments d'OF en trois catégories logiques : les serveurs, les mandataires et les fournisseurs. Un mandataire est un objet représentant le service auprès d'un client [56]. Chaque client, désirant accéder à un service, doit acquérir

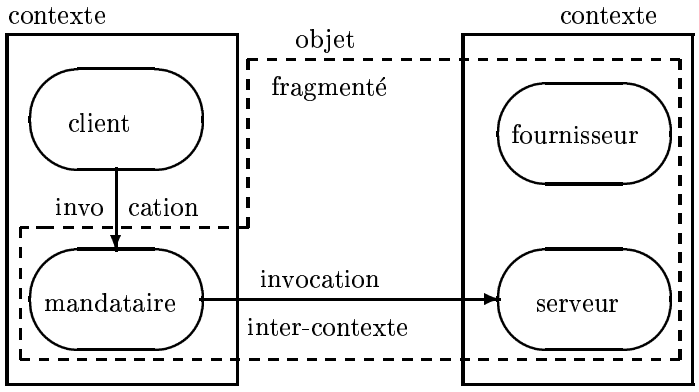

FIG. 2.4 – Les composants logiques d'un objet fragmenté.

un mandataire pour ce service dans son contexte. Le mandataire constitue sa seule interface avec le service. Un mandataire peut traiter les requêtes localement, ou bien les retransmettre au serveur distant (*cf.* figure 2.4). Un serveur est un objet capable de répondre aux requêtes des mandataires. Un fournisseur est responsable de la fourniture de mandataires à la demande des clients.

2.6.3 Migration des Objets Elémentaires

Un contexte peut accéder à un nouvel OF, précédemment inconnu, par migration d'un mandataire pour cet OF vers ce contexte.

Considérons par exemple un programme sous SOS qui demande l'ouverture de fenêtre sur l'écran (voir figures 2.5 et 2.6). Ceci consistera à importer d'un mandataire de fenêtre, par demande adressée au gestionnaire de fenêtrage, fournisseur de mandataires. Ce dernier va créer un mandataire de fenêtre M (qui sera exporté vers le demandeur, comme interface de sa fenêtre), et un serveur de la fenêtre S, qui exécutera les commandes graphiques.

2.6.4 Réinitialisation et prérequis

Nous ne détaillons pas les primitives de création et de recopie binaire d'un contexte à l'autre. Nous nous intéressons par contre au mécanisme de *réinitialisation* et de *prérequis* par lequel la migration conserve la sémantique de l'objet migré.

L'objet Y est prérequis de l'objet X si X a besoin de Y dans le même contexte pour fonctionner. Pour terminer l'importation d'un objet, il faut d'abord importer ses prérequis, et ainsi de suite récursivement. La récursion se termine lorsqu'un objet n'a plus de prérequis à installer. Alors le réinitialisateur de cet objet est invoqué. Au retour de la récursion, chaque objet est à son tour réinitialisé dans le sens du dépilement.

Une fois la valeur binaire d'un objet installé dans le contexte destinataire, le système appelle donc sa *procédure de réinitialisation*, un constructeur de la classe correspondante. Cette procédure se charge par exemple de mettre à jour les pointeurs, ou bien de requérir des importations supplémentaires.

Le premier prérequis d'un objet de données est le code correspondant. Le réinitialiseur d'un objet code exécute une édition de liens dynamique et une vérification

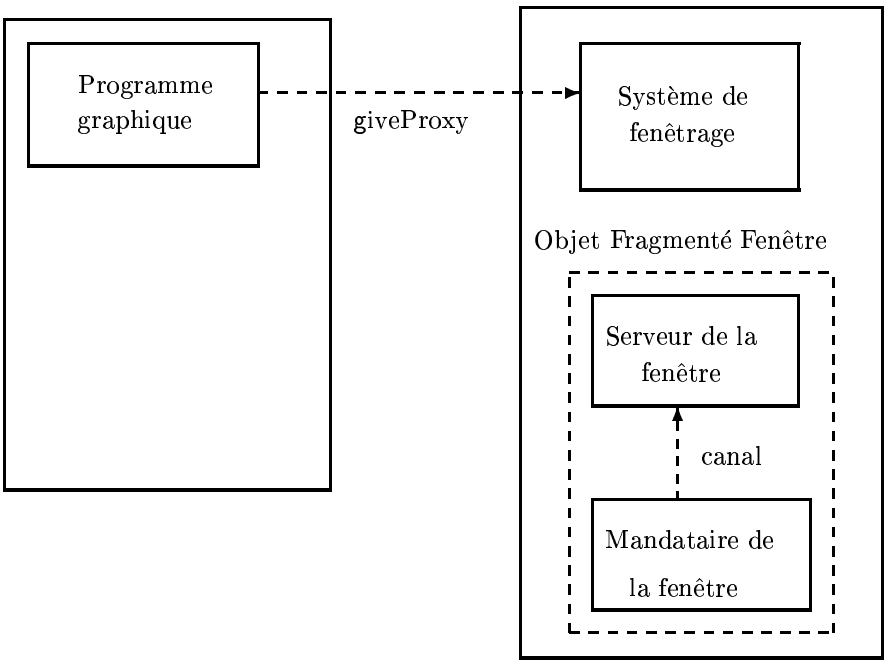

FIG. 2.5 – Avant migration : le gestionnaire de fenêtrage a créé un mandataire et un serveur de la fenêtre, et les a connectés ensemble.

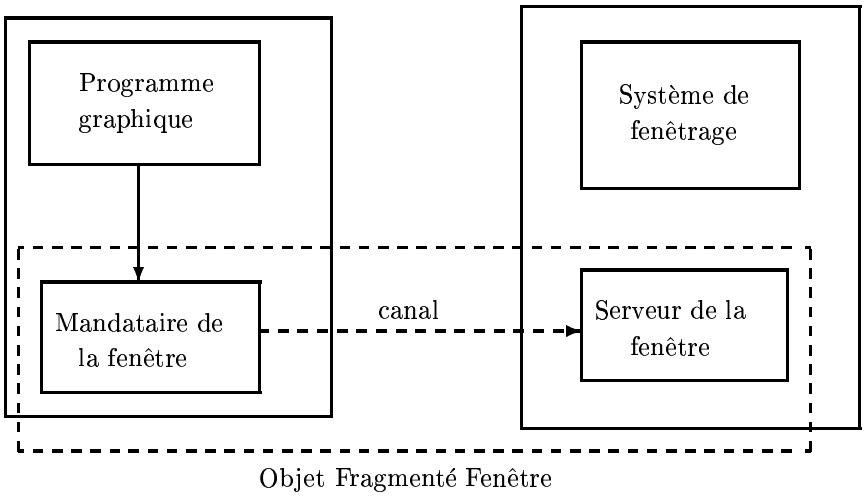

FIG. 2.6 – Après migration : le mandataire de la fenêtre est dans le contexte du client et reste connecté au serveur de la fenêtre

de types [25]. C'est grâce à ce mécanisme que le typage fort des objets, vérifié par le compilateur, est préservé au cours de la migration, même entre processus compilés séparément.

Un prérequis n'est importé que s'il n'est pas déjà présent, même sous forme de mandataire. Ceci permet éventuellement à deux objets importés de partager le même code. De même, le code des mandataires peut être lié statiquement, sans perte de fonctionnalité.

2.6.5 Bilan

Un OF constitue logiquement un sous-réseau de communication privilégié accessible seulement à ses fragments. Ceci suppose résolu le problème de l'authentification répartie, que nous avions à l'époque un peu naïvement ignoré. Cependant, d'une part il ne faut pas oublier que les problèmes de sécurité ne paraissaient pas en 1986 aussi sérieux qu'aujourd'hui. D'autre part il faut souligner que nos hypothèses de sécurité étaient clairement énoncées.

Nous avons inventé un concept aujourd'hui florissant, celui de mandataire. Avec les OF et SOS nous avons posé les bases de la programmation par objet réparti.

Le système SOS, conçu pour permettre aux applications réparties de se constituer en OF, est lui-même constitué d'un ensemble d'OF, dont ce bref résumé a omis la description : service d'accointances (le gestionnaire réparti des objets), de nommage, de persistance, et de communication. Le modèle d'objets de SOS, construisant des objets fragmentés par composition de mécanismes de migration élémentaires est simple et général. L'OF règle d'une façon élégante le problème de l'encapsulation et du typage sûr en réparti. L'OF permet une certaine transparence, grâce au mandataire, sans graver des politiques de transparence dans le marbre, puisqu'un OF se construit de façon autonome et dynamique. Par exemple l'existence de caches ou de protocoles spécifiques entrent naturellement dans le cadre des OF.

Aujourd'hui, le système Jini reprend largement les mêmes concepts et les mêmes mécanismes [66]. SOS a anticipé certains développements ultérieurs. Ainsi le mandataire préfigure une technique utilisée sur les sites Web les plus modernes, la migration transparente de l'état du protocole chez le client [45].

Nous avons par la suite généralisé la migration de SOS en proposant un protocole de liaison générique et flexible [57, 38, 39]. Ce protocole est réalisé comme *intergiciel (middleware)* par-dessus un système d'exploitation existant. Cette couche est le mécanisme minimal et portable offrant le service des OF.

SOS offrait un mécansime de persistance par migration vers le disque [35]. Cette persistance explicite permet de stocker les objets mais est source d'erreurs de programmation. Nous en avons conclu la nécessité d'un mécanisme de persistance implicite, la «persistance orthogonale» [4]. Pour cela il faut coupler un mécanisme d'atteignabilité ou ramasse-miettes, qui a fait l'objet de nos travaux sur les SSPC (chapitre 3), et faire fonctionner correctement le ramasse-miettes en présence de données répliquées, problème résolu avec Larchant (chapitre 4).

Chapitre 3

Le ramasse-miettes dans un système réparti à messages

3.1 Le partage de l'information et la persistance

Le partage de l'information est une fonction essentielle des systèmes informatiques. Ceci inclut bien sûr les messages, mais aussi la persistance, c'est-à-dire la conservation et la récupération des données de façon résistante au temps. Cette seconde fonction étant relativement négligée par les chercheurs en systèmes répartis, j'en fait un axe central de ma recherche, de 1990 à 2000.

La persistance pose deux problèmes aux chercheurs en système. La première l'algorithme qui choisit les objets à persister, problème qui se ramène au ramassage de miettes. Cette question est abordée dans le présent chapitre, dans le cadre d'un système réparti classique (à communication par messages). La seconde, comment gérer les différents niveaux de mémoire (persistante, cache, et réplicats) sera abordée dans le chapitre suivant.

3.1.1 La persistance par atteignabilité

La persistance explicite, comme dans SOS, est sujette à des erreurs de programmation. Un objet doit persister dans son entiéreté. En particulier tout objet pointé par un objet persistant doit lui aussi persister ; c'est ce qu'on appelle l'*intégrité des références (referential integrity)*. En présence de références multiples vers le même objet, cette intégrité est complexe à maintenir manuellement. C'est le même problème qu'on rencontre avec la gestion dynamique de mémoire, et la solution est la même : un *ramasse-miettes (garbage collector, GC)* automatique. Le bon modèle est celui de la persistance automatique, dit *persistance par atteignabilité (persistence by reachability)* : tout objet atteignable, par un chemin de références, depuis une *racine de persistance*, est lui-même persistant [4].

3.1.2 Définitions

Nous considérons le graphe orienté, dit *graphe des objets (object graph)*, dont les nœuds sont les objets et les arcs sont les références. Nous appelerons *origine* d'une référence l'objet pointant, et *cible* l'objet pointé.

Une application accède au graphe uniquement par les objets *racines* et en suivant les références dans le sens des arcs. L'exécution d'un programme, dit alors *mutateur*, peut modifier exécution, par *affectation de référence*, c'est-à-dire la modification d'une référence contenue dans un objet. Les seules mutations légales sont : premièrement, l'affectation d'une référence vers un objet nouvellement créé (que nous noterons X := new) ; deuxièmement, l'affectation, depuis un autre objet, d'une référence existante (notation X := Y).

3.1.3 Le ramasse-miettes

Par le jeu des mutations, un objet peut devenir inaccessible depuis les racines ; il s'ensuit des règles de légalité qu'il ne pourra jamais plus redevenir accessible. On l'appellera *miette (garbage)*. Un *ramasse-miettes (Garbage Collector, GC)* est un algorithme de détection et de destruction des miettes. Le *collecteur* est l'acteur système exécutant le GC afin de récupérer les ressources (en particulier mémoire) utilisées par les miettes. Le collecteur est un acteur privilégié, non tenu par les règles de légalité ci-dessus.

On attend d'un GC qu'il soit à la fois *sûr*, c'est-à-dire que les objets qu'il détecte sont bien inaccessibles ; *vivace*, c'est-à-dire qu'il ne passe jamais dans un état qui l'empêche de détecter des miettes ; et *complet*, c'est-à-dire qu'il détecte bien toutes les miettes. Dans un SIRGE, réunir ces trois propriétés à la fois est infaisable ; on est amené alors à affaiblir au moins l'une d'entre elles. On choisit généralement d'affaiblir la complétude, car elle est la moins critique.

GC centralisé

Les ramasse-miettes centralisés sont bien synthétisés par Wilson [65] et par Jones et Lins [30]. Contentons-nous ici de rappeler les deux grandes familles d'algorithmes de GC.

Le *comptage de références (reference counting)* consiste à attacher à tout objet un compteur du nombre de références pointant sur lui, et à mettre à jour les compteurs à l'occasion des affectations de pointeur. Prenons l'exemple de deux références X et Y pointant vers les objets A et B. Dans l'affectation X := Y, il faut incrémenter de un le compteur de B et décrémenter de un le compteur de A. On peut remplacer le compteur par la liste des origines de références (ou références en arrière). On parle alors de *listage de références (reference listing)*.

Lorsquun compteur atteint zéro, l'objet correspondant est miette. L'inverse n'est pas forcément vrai, en particulier dans le cas d'un cycle de références (par exemple A contient un pointeur vers lui-même). Le comptage de références est un algorithme *acyclique* (ne détectant pas les cycles de miettes), incomplet.

La seconde famille s'appelle *traçage (tracing)*. Ici, le collecteur parcourt le graphe depuis les racines ; un objet rencontré au cours de ce parcours est atteignable, les autres sont des miettes. Le *marquage et balayage (Mark-and-Sweep)* utilise un bit de marquage pour distinguer ces deux sous-ensembles. Il existe de nombreux autres algorithmes de traçage mais ce n'est pas notre propos ici. En principe, le traçage est complet, mais il en existe des variantes incomplètes, par exemple le collecteur «prudent» de Boehm et Weiser [11]. Le défaut du traçage est son caractère global ; son coût est proportionnel à la taille mémoire, et devient prohibitif avec les grandes mémoires.

Le GC *partitionné* combine traçage et comptage, afin de profiter des avantages des deux familles d'algorithmes, tout en minimisant leurs inconvénients respectifs.

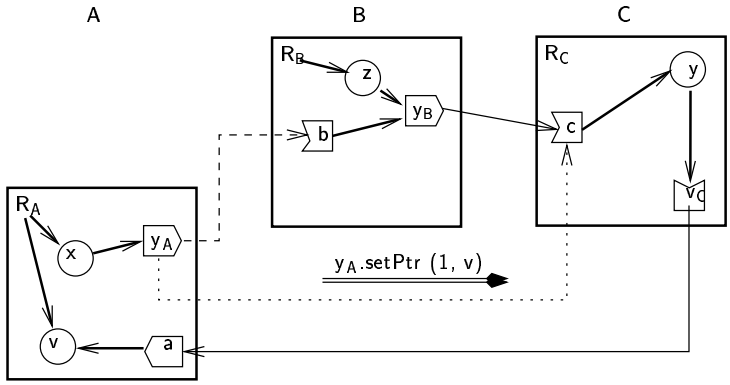

FIG. 3.1 – **Exemple d'envoi de référence.** La souche y_A envoie un message d'appel le long du lien faible vers le scion c. Un des arguments est un pointeur vers v. Le protocole d'emballage créé le scion a du côté émetteur, et la souche v_C chez le récepteur. Une référence vers v dans le processus C utilise l'adresse v_C.

Le principe est de partitionner l'ensemble des objets en sous-ensembles disjoints, que nous appellerons *espaces*. Pour un espace S, l'algorithme maintient des structures de données, que nous appelons *scions* de S, enregistrant les références dont la cible est dans S et l'origine éventuellement hors de S. Les scions sont gérés par comptage ou listage de référence. Chaque espace est collecté par traçage, de façon asynchrone par rapport aux autres espaces, en partant, d'une part des racines locales à l'espace, d'autre part de ses scions. Un espace est choisi suffisamment petit pour faire l'objet d'un traçage.

Utilisations du GC partitionné

Le GC partitionné est bien adapté aux *bases de données d'objets (BDO,* en anglais *Object-Oriented Databases, OODB)* et aux SIRGE. Une BDO contient souvent d'énormes quantités d'objets, dont la plus grande partie réside sur disque. Le coût d'un traçage global serait prohibitif, et ce d'autant plus qu'il doit respecter l'atomicité des transactions. Le GC partitionné permet de concentrer l'effort sur un seul espace de la BDO [3], en choisissant le plus rentable [14].

Le problème est encore plus aigu dans un SIRGE. Il hors de question de tracer, par exemple, tout l'Internet, à cause du coût, des pare-feux, et des pannes. Le GC d'un SIRGE sera partitionné, par exemple par processeur ou par processus [49]. C'est le cas des CPSS que nous présentons dans la suite de ce chapitre. Dans ce cas l'affectation X := Y nécessite l'envoi de messages entre les espaces contenant X et Y. Il s'agit donc du modèle de système réparti à communication par messages.

Dans le chapitre 4 nous étudierons le modèle à mémoire partagée avec l'algorithme Larchant. Le partitionnement par processus n'a plus de sens, et dans Larchant un espace est est un sous-ensemble de la mémoire, susceptible d'être répliqué dans plusieurs processus [21, 22, 61].

3.2 Les Chaînes de Paires Souche-Scion (CPSS)

Nous présentons brièvement notre système de Chaînes de Paires Souche-Scion (CPSS), en nous référant à la figure 3.2.

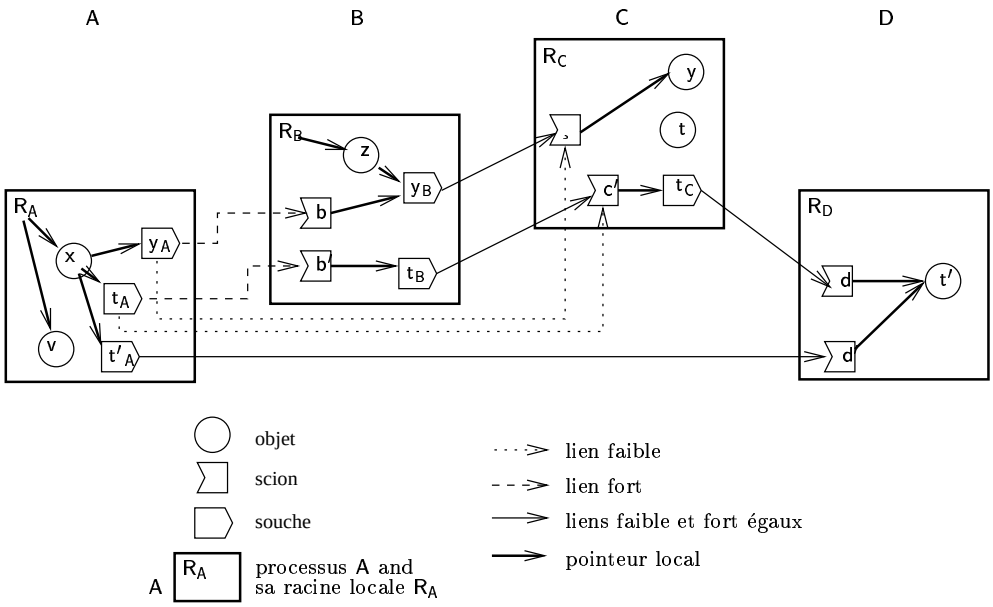

FIG. 3.2 – Architecture des CPSS. Une référence à y a été envoyée du processus C à B, puis renvoyée à A. C a envoyé à B une référence à t, renvoyée ensuite à A. Ensuite, l'objet t a été migrée vers D sous le nom t′. Enfin, une référence à t′ a été envoyée de D à A.

Une référence distante est représentée par un pointeur vers un objet *souche (stub)*, encapsulant la localisation de l'objet distant et la communication avec lui. La souche désigne le scion dans l'espace cible. Pour réaliser une affectation de pointeur distant comme z := y il faut déjà un message, depuis l'espace C contenant y à l'espace B contenant z. À l'émission de ce message, l'espace émetteur crée un scion ; à la réception du message, le récepteur crée la *souche* correspondante. En cas d'affectations successives ou de migration, des chaînes de paires souche-scion se créent. Dans notre figure, x, y et z sont des objets localisés dans trois espaces différents. La séquence d'affectations z := y ; x := z crée une chaîne joignant les trois espaces. Si y est migré vers un autre espace, les chaînes existantes vers y sont prolongées. Les CPSS comportent des protocoles efficaces de raccourcissement de chaîne et de recouvrement après panne d'un espace. Les CPSS ont été mis en œuvre en C++, Smalltalk et Objective-CAML, et sont disponibles de façon publique [47].

L'invocation à travers une CPSS est efficace : en effet chaque niveau d'une CPSS contient une localisation, et les chaînes longues sont raccourcies, éventuellement de façon paresseuse. Ces actions ne requièrent en général aucun message supplémentaire par rapport à ceux de l'application, et aucune synchronisation. Les techniques utilisées sont bon marché, robuste, d'applicabilité générale, et passent à l'échelle.

Le ramasse-miettes est partitionné, utilisant le listage des références. Une souche devenue inaccessible sera détruite par le GC traçant de l'espace d'origine. Un *collecteur inter-espaces* diminuera le compte de références du scion correspondant ; si le compte passe à zéro, le scion est détruit. Finalement, un objet qui n'était référencé que par le scion détruit sera ramassé par la prochaine exécution du collecteur de l'espace destination.

Si l'architecture des CPSS est relativement simple à décrire, l'algorithme contient des aspects assez subtils liés aux croisements de message et aux pannes.

3.2.1 Communication distante

Les processus applicatifs communiquent par messages. Les données transportées sur un message peuvent contenir des références. Quand un processus qui reçoit une référence, l'objet cible devient accessible du récepteur. Aussi l'émetteur du message alloue-t-il un scion et le receveur une souche.

Prenons l'exemple du système dépeint à la figure 3.2. Initialement un message a été envoyé de C à B contenant une référence à y. Quand C a envoyé le message, il a créé le scion $_{,}$; quand B a remis le message, il a créé la souche y_B. Dans un système à appel de procédure distante [8] la souche y_A sert de représentant pour y : pour invoquer y, un client appellera la procédure correspondante de y_A. Celle-ci emballe les arguments, envoie un message d'invocation, récupère le message de réponse, et déballe les résultats. Quand un arguments ou un retour est une référence, le protocole d'emballage crée les souches et les scions correspondants.

3.2.2 Chaînes

Un processus ayant reçu une référence locale peut à son tour l'envoyer comme argument (ou retour) d'une invocation distante. Ceci a pour effet de rallonger la chaîne par ajout d'une paire souche-scion du côté origine. Par exemple dans la figure 3.2, le processus B a reçu une référence à y, à travers la souche y_B. Par la suite, il a envoyé cette même référence au processus A, reliant le scion b à la souche y_B. Une migration a pour effet de rajouter une paire souche-scion du côté de la cible. Ainsi l'objet t a migré vers t'.

Les chaînes trop longues sont raccourcies. Pour éviter les échanges de message supplémentaires, nous préconisons de ne le faire que de façon paresseuse, quand une chaîne est utilisée pour l'invocation. L'information nécessaire est transportée sur le message de retour.

Une optimisation supplémentaire consiste à maintenir un deuxième lien, dit *faible*, dans une souche. Une invocation est toujours envoyée le long du lien faible, qui pointe sur la localisation courante de l'objet cible, à moins que ce dernier n'ait migré récemment. Dans ce dernier cas l'invocation nécessitera un aller-retour de message supplémentaire.

Par exemple, la chaîne *forte* de x à y dans la figure possède une indirectin à travers le processus B. Une invocation suivra le lien faible, directement vers la cible dans son processus C. À l'inverse, une invocation de t' à travers la deuxième référence de x enverra d'abord à C, de façon à localiser la cible effective D. Après raccourcissement de toutes les chaînes et ramassage des miettes, le résultat est celui montré dans la figure 3.3.

3.2.3 Ramassage de miettes

Chaque processus constitue un espace au sens du GC partitionné et exécute un GC traçant local. Si un objet est atteignable, que ce soit d'une racine locale ou que ce soit d'un scion, le collecteur doit le conserver intact. Par exemple, dans la figure 3.2, y ne doit pas être ramassé, bien que n'étant pas atteignable de la racine locale R_C. Quand une souche est désallouée, un message permettra de désallouer le scion correspondant.

Le GC inter-espaces utilise le listage de références, car un compte serait trop difficile à maintenir cohérent. Chaque scion représente un espace source. Le nombre de processus référençant un objet est inférieur ou égal au nombre de scions pointant

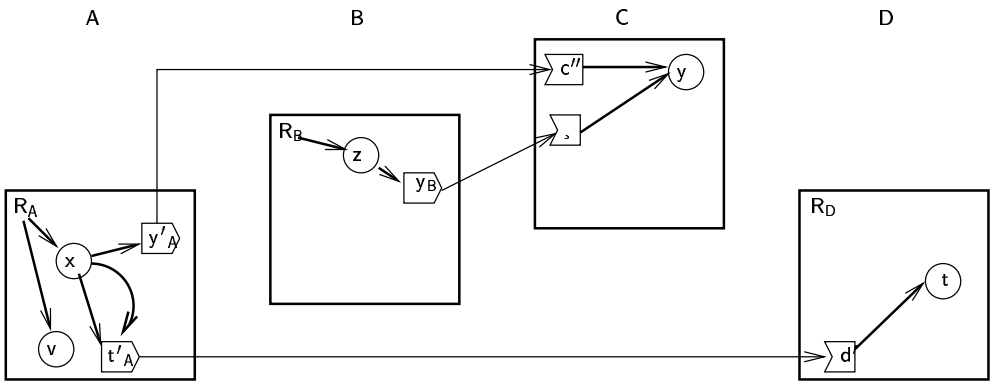

FIG. 3.3 – **Après raccourcissment des indirections et ramasse-miettes** Suite à la figure 3.2 : L'objet x a invoqué y, créant une nouvelle souche y'_A en remplacement de y_A ; y_A et b, devenus inatteignables, ont été ramassés. Ensuite, x ayabt invoqué t', la souche t_A est remplacée par la souche déjà existante t'_A ; la chaîne d'origine t_A, devenue inatteignable, est ramassée. L'objet inatteignable t est ramassé localement.

vers cet objet. L'inégalité est périodiquement renforcée par le raccourcissement des chaînes et par le ramasse-miettes inter-espaces.

Le protocole de ramassage détecte et détruit les scions qui (i) ne sont pointés par aucun lien fort, et (ii) qui ne pourront pas devenir pointés dans l'avenir. Notons qu'à cause de l'asynchronisme des messages (et contrairement au cas des sytèmes centralisés), la seconde propriété ne découle pas directement de la première.

3.2.4 GC et asynchronisme des messages

Comme chaque scion correspond à une souche unique, quand cette dernière est détruite et ne sera pas recréée, il faut détruire le scion. Cependant un simple message au moment de la destruction du scion ne suffit pas. À cela, deux raisons : d'une part le message pourrait être perdu, d'autre part l'asynchronisme des messages. Le premier problème est résolu en remplaçant le message de destruction par un message périodique contenant la liste des souches existantes. Lorsque le scion correspondant à une souche n'apparaît plus dans la liste, le scion est détruit.

Le second problème, plus délicat, est l'asynchronisme entre un message contenant une référence, et un autre message rendant le même objet inaccessible. Prenons par exemple une CPSS du processus A vers le processus B pointant sur l'objet y. Si y est inatteignable dans A, A envoie un message de destruction vers B. Il se peut qu'en même temps B envoie à A un message contenant une référence vers le même y. La solution à ce problème constitue la partie la plus délicate de notre spécification.

Deux mécanismes différents sont nécessaires. Le premier permet à B de ne pas prendre en compte les messages de destruction qui auraient croisé la référence envoyée le plus récemment. Le second permet à A de ne pas prendre en compte les messages, contenant une référence, qui auraient été retardés et auraient croisé un message de destruction déjà pris en compte. Le problème serait simplifié si on supposait le transport de messages entre A et B, mais non résolu entièrement. Le protocole est basé sur des estampilles maintenues à la fois du côté souche que côté scion. Il est relativement subtil mais a été rigoureusement prouvé sûr et vivace.

3.3 Conclusion

Le système le plus directement comparable aux CPSS est Network Objects [9], repris par Java RMI [66]. Network Objects est contemporain des CPSS mais n'a pas les mêmes qualités. Network Objects évite les problèmes d'asynchronisme en sérialisant les opérations. Le système ne permet pas un raccourcissement paresseux et exige un protocole de communication fiable et synchrone. Network Objects ne permet pas la migration des objets.

Il faudrait pouvoir rentrer plus dans le détail pour apprécier les propriétés de tolérance aux pannes, de performance, de passage à l'échelle des CPSS. Les CPSS ont une sémantique bien définie, définie par des invariants qui sont maintenus même en présence de pannes de communication ou de sites. Le ramasse-miettes réparti, partitionné, tolère toutes les incohérences permise par les invariants de CPSS.

Le ramassage des CPSS est acyclique, c'est-à-dire que les cycles répartis de miettes ne sont pas détectées. Nous avons développé par la suite des algorithmes cycliques, que nous ne rapportons pas ici ; nous renvoyons le lecteur intéressé à nos publications avec F. le Fessant [23, 61, 34].

Chapitre 4

Mémoire répartie d'objets et ramasse-miettes

4.1 Persistance, partage de mémoire et ramasse-miettes

Dans le chapitre précédent, nous nous placions dans un modèle de système réparti à communication par messages. Partant des besoins de la persistance, nous en avons déduit le besoin du ramasse-miettes, et conçu un nouvel algorithme particulièrement efficace, celui des CPSS.

Or nous n'avons en fait pas traité la persistance. La raison principale est que le modèle à messages n'y est pas adapté. [1] Le partage d'objets persistants implique une communication par effet de bord (affectation), compliquée par l'existence de réplicats de ceux-ci : copie sur disque, copie locale en mémoire cache, copies en cache distantes éventuelles. La mutation du graphe est implicite (et non par messages) et n'est pas forcément visible tout de suite à cause des effets du protocole de cohérence.

La persistance impose donc de prendre en considération un modèle différent, celui de communication par mémoire partagée. Un protocole tel que les CPSS, conçu pour le modèle classique, ne fonctionne pas dans le nouveau modèle.

Nous avons été les premiers à souligner l'importance qu'il y a à découpler le ramasse miettes de la cohérence. Cela est impératif pour des raisons de performance applicative et simplifie les protocoles de cohérence comme de ramassage en évitant les interactions entre eux. Nous avons ainsi conçu l'algorithme de ramasse-miettes Larchant spécialement pour une mémoire partagée, éventuellement à cohérence faible. Il est simple et élégant. Il a ensuite été mis en œuvre dans le cadre du système PerDiS, une mémoire d'objets persistante répartie pour l'ingénierie coopérative.

4.2 Larchant

Nous nous plaçons dans le cadre de l'abstraction la plus générale, celle de *mémoire virtuelle répartie persistante (MVRP,* en anglais *Persistent Distributed Shared Memory, P-DSM).* Le ramasse-miettes dans une mémoire partagée est un

[1] Une autre raison est qu'il faut pouvoir distinguer entre objets atteignables depuis une racine de persistance, et ceux atteignables seulement d'une racine ordinaire. Or un GC à comptage (comme celui des CPSS) ne permet pas de faire cette distinction.

problème plus difficile que dans le modèle classique (chapitre 3). Les raisons en sont les suivantes :

- L'application modifie le graphe de façon concurrente, et par affectation de pointeur en mémoire, et non par envoi de messages. L'affectation étant une opération très fréquente, elle ne doit pas être ralentie ; par exemple un listage de référence intégré avec l'affectation serait inacceptable.
- Les réplicats ne sont pas instantanément cohérents entre eux. Il est difficile et coûteux d'observer une image cohérente du graphe des objets.
- Le graphe des objets est très grand et réparti. Une grande partie de ce graphe est distant ou stocké sur le disque. Il serait infaisable de tracer tout le graphe en une seule fois.
- Une affectation peut avoir des effets sur des portions lointaines et volumineuses du graphe. Ceci a un impact sur l'ordonnancement global des opérations.
- Le ramasse-miettes ne doit pas entrer en compétition avec l'application. En particulier, le collecteur ne doit pas prendre des verrous, ni être la cause d'opérations de cohérence, ni d'entrées-sorties.

Les travaux précédents sur le ramasse-miettes dans une mémoire partagée [3, 36, 37, 67] ne sont pas applicables car ne prenant pas en compte ces problèmes. Nous avons donc conçu l'algorithme Larchant, spécifiquement pour une MVRP persistante dans un SIRGE. Les applications envisagées sont les logiciels d'ingénierie coopérative. Larchant est conçu comme une MVRP de grande échelle. Tout objet atteignable depuis une racine de persistance est persistante et accessible depuis n'importe quel site.

Les objectifs de conception de l'algorithme sont son caractère correct, le passage à l'échelle, un surcoût faible, et l'indépendance par rapport au protocole de cohérence. Des objectifs secondaires sont d'éviter les modifications des applications ou des compilateurs.

Notre algorithme divise le ramasse-miettes global en sous-problèmes indépendants et de petite échelle, qui peuvent s'exécuter de façon asynchrone. Ainsi ils pourront être exécutés de façon différée ou en tâche de fond.

- La mémoire est partitionnée en «bunches» (grappes) lesquels peuvent être répliqués. Le bunch est un espace au sens du GC partitionné, avec traçage dans un réplicat de bunch, et listage entre bunches.
- Chaque site exécute un collecteur traçant standard [65] qui trace un ou plusieurs réplicats de bunches, présents sur ce site, à la fois.
- Le protocole entre collecteurs n'implique aucune synchronisation.
- Un collecteur n'examine que la partie locale du graphe, sans causer d'entrées-sorties, et sans prendre de verrous.
- Un collecteur peut s'exécuter même lorsque les réplicats locaux de bunches ne sont pas cohérents.

Larchant s'articule autour de cinq règles de correction. L'algorithme est sûr (seuls les objets inatteignables sont détruits) et vivace (l'algorithme ne se met pas dans un état où il ne peut plus détecter de miettes). Malheureusement, mais inévitablement, il n'est pas complet.

L'algorithme a été modélisé et prouvé correct par Ferreira et Shapiro [21]. Le lecteur trouvera des informations complémentaires dans nos précédentes publications [19, 20, 58].

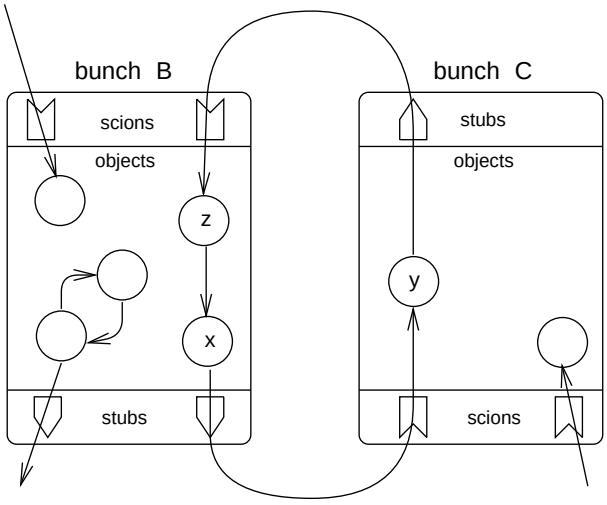

FIG. 4.1 – Deux bunches et leur contenu : objets, souches et scions

4.3 Algorithme de GC et règles de sûreté

La mémoire Larchant est structurée selon le schéma de la figure 4.1. Pour les besoins du GC partitionné, il est divisé en *bunches* disjoints et de gros grain (typiquement, ensembles de pages mémoire) dans lesquels les objets sont alloués. Ce qui est nouveau par rapport au chapitre précédent, c'est qu'un bunch peut être répliqué dans plusieurs processus. Le GC est correct même si les réplicats ne sont pas cohérents entre eux.

Une référence est toujours un simple pointeur. Ainsi l'affectation de référence ou son suivi, opérations extrêmements fréquentes, ne subissent aucun surcoût.

La figure 4.2 montre un example de fonctionnement. À l'intérieur d'un bunch Larchant exécute un GC traçant, et utilise le listage de références entre bunches. Le collecteur découvre les affectations de pointeur après coup, lors du traçage. Lorsqu'une référence traverse une frontière de bunch, le collecteur gère des méta-données supplémentaires, une souche sur la frontière sortante, et un scion sur celle entrante.

Malgré la similarité de noms, la fonctionnalité des souches et scions diffère sensiblement des méta-données correspondantes des CPSS. En effet ici les méta-données sont privées au collecteur, ne sont pas visibles du mutateur, et sont gérés de façon asynchrone (et non dans le chemin critique d'une affectation).

Voyons brièvement le fonctionnement du système et nos notations Lorsqu'un mutateur fait l'affectation de pointeur $<\mathsf{x} := \mathsf{y}>_i$, c'est-à-dire recopie la valeur du pointeur y dans le pointeur x dans le processus i, un, deux ou trois processus sont impliqués. Supposons que les objets x, y, z et t sont localisés dans les bunches X, Y, Z et T respectivement, et qu'initialement, x pointait sur z et y pointait sur t. Postérieurement à l'affectation, le collecteur s'exécutant dans le processus i incrémente le compte de références pour t par l'opération locale $<\mathsf{increment.stub}(\mathsf{X},\mathsf{x},\mathsf{T},\mathsf{t})>_i$ et envoie le message $\mathsf{increment.scion}(\mathsf{X},\mathsf{x},\mathsf{T},\mathsf{t})$ au collecteur d'un processus j gérant T. Il décrémente le compte de références de z en exécutant localement $\mathsf{decrement.stub}(\mathsf{X},\mathsf{x},\mathsf{Z},\mathsf{z})$ et en envoyant le message $\mathsf{decrement.scion}(\mathsf{X},\mathsf{x},\mathsf{Z},\mathsf{z})$ au collecteur du processus k, gestionnaire de Z.

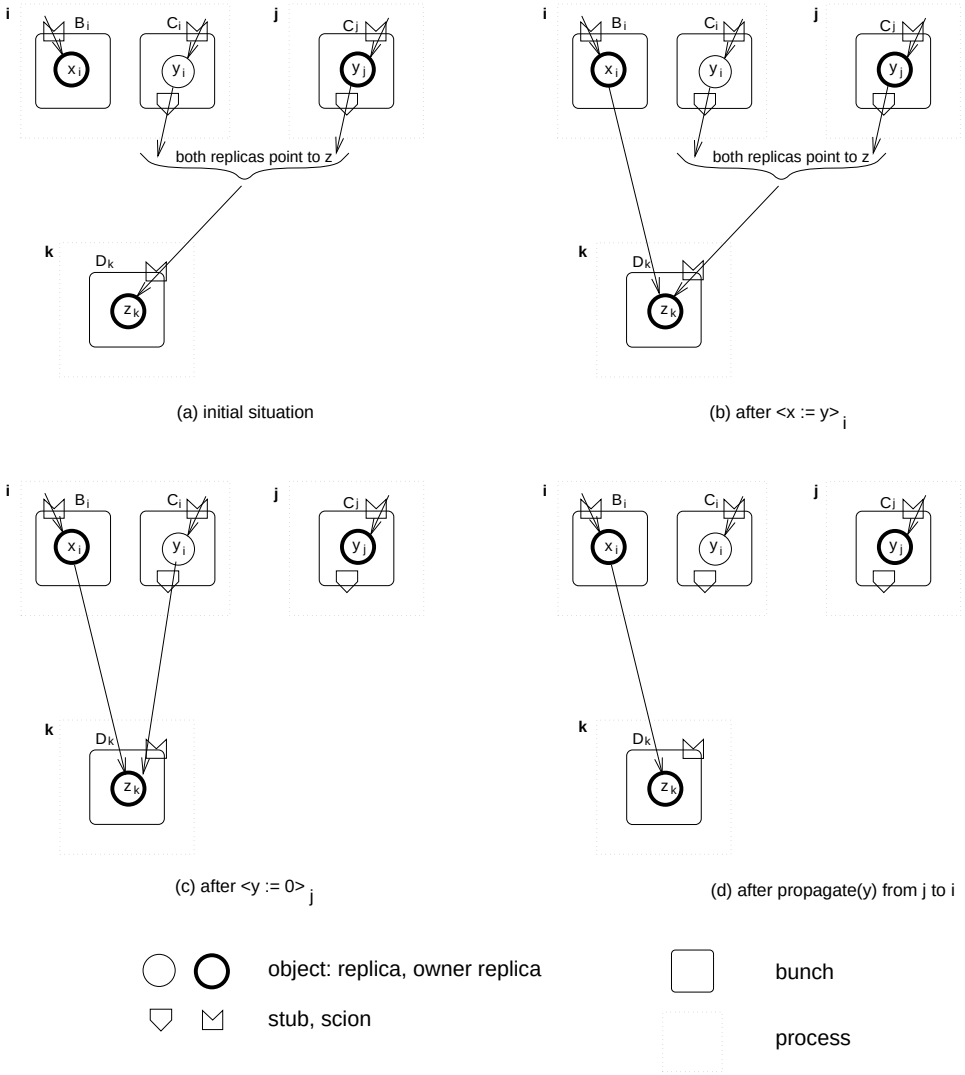

FIG. 4.2 – Prototype problématique de l'exécution de mutateur. Notez que les souches et scions deviennent temporairement incohérents par rapport aux pointeurs. Grâce aux règles décrites dans le texte, cela ne met cependant pas en cause la sûreté de l'algorithme.

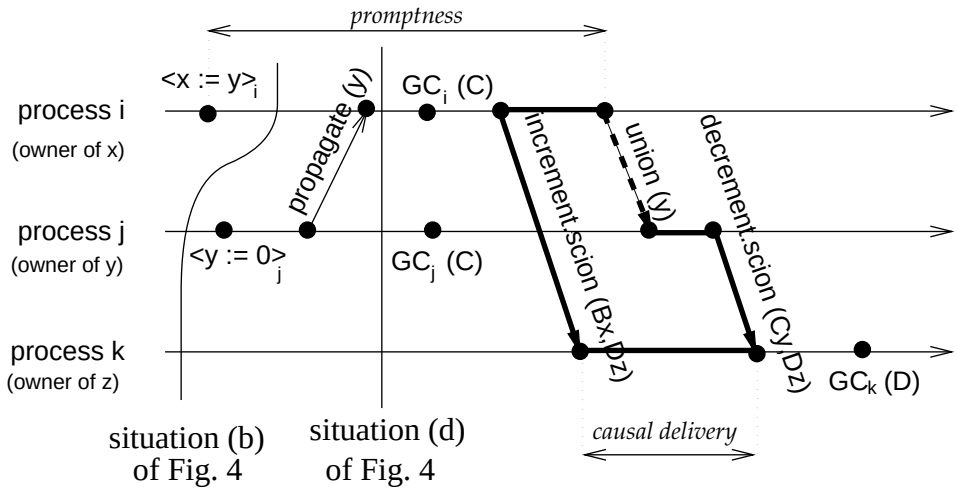

FIG. 4.3 – Effets des règles de sûreté pour l'exemple de la figure 4.2. Sur le site i, l'envoi de increment.scion(B, x, D, z) peut être retardé au plus tard jusqu'à l'envoi de union(y). Noter la dépendance causale (indiqué par les traits gras) entre les messages increment.scion et decrement.scion, portée par le message union.

4.3.1 Le traçage en présence de réplicats : règle de l'union

Nous décrivons maintenant la coopération entre collecteurs traçants de chaque processus, afin de tenir compte du partage et de la réplication. Notre but est de permettre l'indépendance entre collecteurs, ce qui implique de les exécuter sans attendre que la mémoire soit cohérente.

À un moment donné, le collecteur du processus i pourrait observer x_i en train de pointer sur z, alors que le collecteur de j observe x_j qui pointe t. Le protocole de cohérence devrait rendre les deux réplicats égaux, mais le collecteur ne sait pas quelle est la valeur correcte de x. Ce dernier doit donc accepter tous les réplicats comme valides et ne jamais détruire un objet, tant qu'il n'est pas inatteignable dans l'union de tous les réplicats. Nous généralisons cet exemple selon la règle suivante :

> **Condition de sûreté I : Règle de l'union.** *S'il existe un réplicat de* x x_i *pointant sur* z, *et s'il existe un réplicat de* x *atteignable* x_j, *alors* z *est atteignable.*

Le plus intéressant est que cette règle reste vraie même si $i \neq j$.

Dans le cas d'un protocole de cohérence à «propriétaire» les collecteurs centralisent l'information sur les pointeurs ayant pour origine x chez le propriétaire de x, par un message union.

4.3.2 Autres règles de sûreté

Comme nous l'avons déjà indiqué le comptage de référence est asynchrone par rapport à l'affectation de pointeurs. Ceci permet de ne pas surchager l'affectation, opération très fréquente. L'asynchronisme est permis tant qu'il préserve le caractère atteignable d'un objet. Reprenons l'exemple de la figure 4.2. Au moment de $<x := y>_i$, l'objet z est atteignable et possède un scion, par exemple scion(T, t, Z, z). En principe (mais pas nécessairement) T = Y et t = y. Tant que le scion de z conserve un compte non nul, on peut retarder l'incrément de scion(B, x, D, z). Il reste cependant un problème. Dans notre exemple, dès que le point (d) est atteint, le

message decrement.scion(C, y, D, z) pourrait être livré sur le site k avant increment.scion(B, x, D, z) ; dans ce cas z serait désalloué à tort. Pour éviter cette violation de la sûreté, il suffit de donner priorité aux messages increment.scion sur ceux de decrement.scion et d'union, comme le montre la figure 4.3 : l'intervalle marqué *«promptness»* montre de combien le message increment.scion(B, x, D, z) peut être retardé par rapport à l'affectation ($<x := y>_i$). Les règles de sûreté suivantes déterminent jusqu'à quand le comptage peut être retardé tout en restant sûr.

Nous présentons ces règles sans preuve. Le lecteur intéressé trouvera leur justification formelle ailleurs [21, 61].

> **Condition de sûreté II : Incrément avant décrément.** *Quand un objet est tracé les messages* increment.scion *correspondants sont envoyés immédiatement.*
>
> **Condition de sûreté III : Traçage complet.** *Quand le processus* i *envoie un message* union *ou* decrement.scion*, tous les réplicats présents sur* i *ont été tracés depuis leur dernière affectation.*
>
> **Condition de sûreté IV : Propagation propre.** *Quand le processus* i *envoie* propagate(x), x_i *a été tracé depuis sa dernière affectation.*
>
> **Condition de sûreté V : Livraison causale des message de GC.** *Les messages* increment.scion, union *et* decrement.scion *sont livrés dans l'ordre causal.*

Ces règles donnent les conditions de correction en présence d'asynchronisme, d'une part entre affectation et comptage, d'autre part entre comptage et traçage, et enfin entre traçage et cohérence. En particulier la règle IV implique que lorsque le protocole envoie la nouvelle valeur d'une zone mémoire dans un message de cohérence, cette zone doit avoir précédemment visitée par le GC traçant. Elle entraîne la création de scion au moment de l'émission d'un message contenant un nouveau pointeur, comme dans les CPSS. La règle III indique que le comptage doit prendre en compte tous les pointeurs distants nouvellement créés ou détruits récemment. Enfin les règles II et V évitent que les scions deviennent incohérents par rapport à leurs souches.

4.3.3 L'entrepôt persistant réparti PerDiS

Le projet Esprit PerDiS [60, 51], dont j'étais le coordinateur, est basé sur les résultats de Larchant. Il visait à construire une MVRP pour des applications d'ingénierie coopératives fonctionnant à grande échelle. Le domaine d'application de PerDiS est l'industrie du bâtiment et particulièrement la collaboration inter-entreprises dans ce qui est appelé une «entreprise virtuelle». Les concepts et les algorithmes de Larchant ont donc fait l'objet d'une mise en œuvre pratique et d'une valorisation industrielle.

4.4 Conclusion

Notre travail sur Larchant a porté sur les interactions entre le GC et la gestion de cohérence. Nous avons montré que ses deux composantes, le traçage local et le comptage réparti, peuvent être découplés de la cohérence. Ainsi le GC n'a pas besoin que la mémoire soit cohérente, ne rentre donc pas en compétition avec les éventuels verrous applicatifs, et (dans le cas des données persistantes) ne génère pas

d'entrées-sorties. L'asynchronisme permet des optimisations, comme un traitement par lots du comptage réparti. Les conditions de sûreté imposent une seule interaction impactant les performances de l'application : les messages du protocole de cohérence contenant des mises à jour doivent être tracés par le collecteur.

Nos mesures de performance [19] montrent que le surcoût du traçage est indépendant du nombre de réplicats (il passe à l'échelle) et que l'asynchronisme permet une amélioration significative des performances.

La livraison causale des messages de cohérence (règle V) ne passe pas à l'échelle en général. Cependant cela ne constitue pas un problème réel car les protocoles de cohérence courants assurent déjà la causalité. C'est le cas de notre mise en œuvre basé sur la cohérence à l'entrée.

Le ramasse-miettes réparti n'était pas un domaine vierge, mais notre recherche a apporté un éclairage nouveau. Notre préoccupation était de proposer des algorithmes à la fois corrects, performants dans des applications réalistes, et passant à l'échelle. Avec Larchant, nous avons été les premiers à explorer le découplage entre cohérence et ramasse-miettes, qui est indispensable si on veut des performances acceptables.

Le découplage a l'avantage supplémentaire que les applications et les compilateurs n'ont pas à être modifiés pour profiter du ramasse-miettes. Une application existante peut être portée sur Larchant sans modification et sans surcoût significatifs. En particulier Larchant ne surcharge pas l'opérateur d'affectation pour les besoins du comptage réparti, ce qui aurait pour effet de ralentir considérablement les applications.

La motivation initiale de cette recherche, la persistance par atteignabilité des objets partagés, n'avait pas été satisfaite avec les CPSS. L'algorithme Larchant a apporté des solutions satisfaisantes, dont l'applicabilité industrielle est démontrée par la mise en œuvre PerDiS.

Enfin, nous avons été les premiers à formaliser les règles de sûreté d'un ramasse-miettes en mémoire partagée avec réplication.

Chapitre 5

Protocoles optimistes pour les objets partagés répliqués

Dans un SIRGE une même donnée existe souvent en plusieurs copies ou *réplicats (replicas)*. Par exemple une copie persistante sur disque et une copie de travail (dite cache ou antémémoire) en mémoire, ou une copie distante principale et un cache local. Les réplicats existent, soit pour des raisons de performance, le cache étant beaucoup plus rapide d'accès que la copie distante, soit pour la disponibilité, le réplicat local restant accessible même si une panne de réseau empêche l'accès à la copie distante.

En cas de modification d'un réplicat, il faut maintenir la *cohérence (consistency)* avec les autres. Le problème se pose surtout en cas de mise à jour concurrente sur plusieurs réplicats ; la cohérence se confond alors avec le contrôle de concurrence.

Une première classe d'algorithmes de cohérence, dite *pessimiste*, fait du contrôle de concurrence à priori, afin d'éviter toute divergence entre réplicats. Nous nous intéressons à la classe plus générale des algorithmes *optimistes* [29] où le contrôle peut être fait de façon «paresseuse» ou asynchrone, laissant éventuellement les réplicats diverger, et réparant cette divergence après coup.

Nous décrivons maintenant la réconciliation (contrôle de concurrence à posteriori) dans notre système IceCube [62, 31, 50]. Pour une approche d'ensemble des systèmes à réplication optimistes, nous renvoyons à la synthèse de Saito et Shapiro [54].

5.1 Le système IceCube

Dans la réplication optimiste, les différents utilisateurs peuvent lire et mettre à jour leur réplicat local sans se synchroniser avec les autres. Une mise à jour sera propagée aux autres réplicats quand la communication sera possible. Cependant une mise à jour n'est que provisoire, puisqu'il peut y avoir conflict avec une mise à jour concurrente et prioritaire. Étant donné un ensemble de mises à jour concurrentes, un algorithme de *réconciliation* sélectionne une combinaison sans conflit parmi toutes celles possibles [54]. Dans une approche dite «basée sur les opérations» les actions de mise à jour sont enregistrées dans un journal. La réconciliation rejoue une combinaison des actions, à partir de l'état initial, selon un *plan d'exécution (schedule)*.

Au contraire des autres systèmes existants, IceCube prend en compte la sémantique des opérations pour générer un plan d'exécution, sans être limité à une seule sémantique

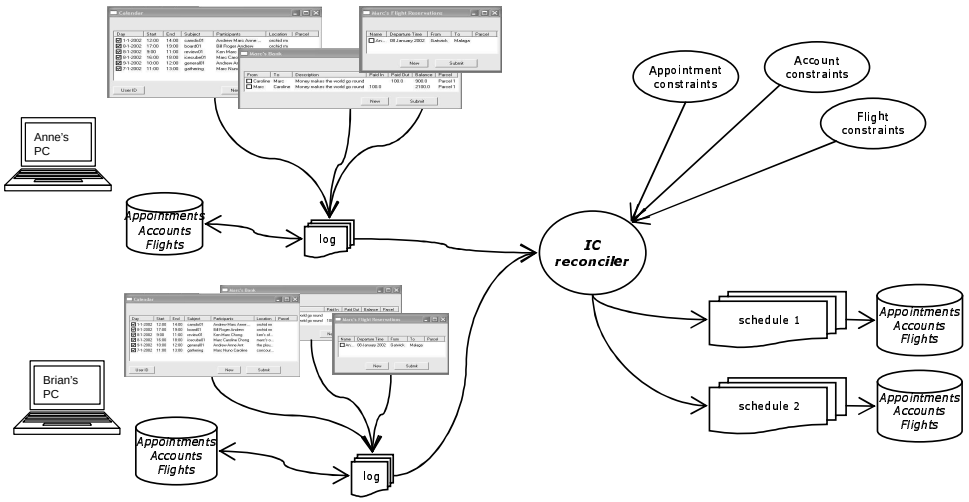

FIG. 5.1 – Structure d'une installation IceCube

applicative. IceCube constitue un intergiciel de réconciliation générique, paramétré par la sémantique. IceCube réconcilie de façon harmonieuse entre utilisateurs, applications et objets, qui ne sont pas forcément conscients les uns des autres. Nous donnons maintenant quelques informations sur IceCube. Le lecteur intéressé pourra se reporter à Shapiro, Preguiça et Matheson [50] pour des explications plus approfondies.

5.1.1 Contributions

IceCube innove sur plusieurs plans. Il possède une interface générale et puissante pour capturer les invariants sémantiques, sous la forme de *contraintes*. Contrairement aux systèmes existants (qui ordonnent les actions selon des critères syntaxiques) IceCube n'abortera une action que si cela se justifie sémantiquement. Notre approche traite la réconciliation comme un problème d'optimisation, minimisant (de façon heuristique) la valeur des actions abortées. Nous traitons les problèmes d'efficacité et de passage à l'échelle en encourageant le programmeur d'application à utiliser les contraintes dites *statiques*. L'efficacité est encore améliorée en décomposant en sous-problèmes de réconciliation indépendants. Nos mesures montrent que IceCube réconcilie en temps raisonnable et passe bien à l'échelle. Enfin, notre approche fonctionne en pratique. Nous avons codé plusieurs applications différentes au-dessus d'IceCube. Il est beaucoup plus simple d'écrire une application réconciliable en utilisant IceCube qu'en partant de zéro. De plus IceCube coordonne les interactions de l'utilisateur avec des applications différentes.

5.1.2 Modèle de système

La figure 5.1 montre la structure d'une installation IceCube typique. Deux utilisateurs, Anne et Brian, partagent des données de rendez-vous, de compte bancaire et de vols, par réplication des objets correspondants sur leur PC respectif. Les applications lisent et mettent à jour ces réplicats localement mais de façon provisoire. Les mises à jour sont journalisées. Le système envoie les journaux au composant de réconciliation. Ce dernier calcule une combinaison des mises à jour provisoires, qu'il

ré-exécute selon un plan d'exécution à partir de l'état initial. Pour cela il prend en compte les informations sémantiques, d'une part celles enregistrées dans le journal, dites *contraintes de journal* et celles caractérisant les objets partagés, *les contraintes d'objet* (voir par exemple «Appointment constraints» sur la figure). S'il y a conflit, le réconciliateur génère des plans sans conflit. L'utilisateur peut alors, soit confirmer l'un des plans, soit demander de nouveaux plans, soit éditer les journaux et recommencer.

5.2 Les contraintes dans IceCube

La détection de conflits et le calcul de plans d'exécution sont basés sur les *contraintes*. Une contrainte peut exprimer aussi bien une intention utilisateur qu'un invariant sémantique. Les contraintes sont soit statiques (ou déclaratives), soit dynamiques (ou impératives). Les premières limitent l'espace de recherche ; les secondes sont constitués des préconditions et postconditions des actions, et se traduisent par l'échec de celles-ci à l'exécution d'un plan.

5.2.1 Contraintes statiques primitives

Le système est basé sur deux contraintes primitives, Order notée $\rightarrow$ et MustHave notée $\bowtie$. Tout plan d'exécution S doit satisfaire les conditions d'intégrité suivantes :

1. Respect de Order : Si a et b sont des actions de S, si $a \rightarrow b$ alors a précède (pas forcément immédiatement) b dans S,
2. Fermé pour MustHave : Si a est une action de S, toute action b telle que $a \bowtie b$ est aussi dans S (pas forcément dans cet ordre, et pas forcément contiguement).

Si deux actions ne sont pas reliées par une contrainte Order, elles peuvent s'exécuter dans un ordre quelconque.

Il découle de la première condition qu'un plan ne contient pas de relation Order cyclique. Si un cycle de Order apparaît entre journaux, il doit être brisé en excluant une ou plusieurs actions des plans d'exécution. Or la recherche de sous-graphes acycliques optimaux est un problème NP-hard. Nous avons développé une heuristique de recherche spécialisé, extrêmement efficace et de très haute qualité.

Bien que très simples, en combinant ces contraintes on a un pouvoir expressif très fort. Nous en présentons maintenant quelques exemples.

5.2.2 Contraintes de journal

Une *contrainte de journal* est une relation entre deux actions et est enregistrée avec eux dans le journal et exprime une intention utilisateur.

La contrainte predecessorSuccessor indique une relation causale. La seconde action ne peut avoir lieu qu'après que la première se soit exécutée avec succès. La relation predecessorSuccessor(a, b) (où a et b sont des actions) est équivalente à la formulation $a \rightarrow b \wedge b \bowtie a$, c-à-d la conjonction des primitives Order and MustHave dans des directions opposées.

La contrainte alternative demande au système de choisir entre deux actions. Elle s'exprime comme une contradition, alternative$(a, b) \equiv a \rightarrow b \wedge b \rightarrow a$, ce qui empêche a et b de faire partie du même plan.

La contrainte parcel groupe des actions de façon indivisible. Soit toutes ses actions sont exécutées avec succès, soit le «paquet» échoue. $\mathsf{parcel}(a, b)$ est équivalent à $a \triangleright\triangleright b \wedge b \triangleright\triangleright a$.

5.2.3 Contraintes d'objet

Une *contrainte d'objet* est une relation entre types d'action et reflète la sémantique d'un objet. A l'initialisation, IceCube extrait les contraintes d'objet en comparant entre elles deux à deux toutes les actions journalisées (ce sont des méthodes des actions).

La contrainte mutuallyExclusive indique si l'exécution de deux actions violerait un invariant. Nous avons la relation $\mathsf{mutuallyExclusive}(a, b) \equiv a \rightarrow b \wedge b \rightarrow a$. Là où existe un ordre favorable d'exécution de deux actions, celui-ci est indiqué par la contrainte bestOrder Nous avons la relation $\mathsf{bestOrder}(a, b) \equiv a \rightarrow b$.

5.3 Reconciliation scheduler

Les contraintes statiques limitent la taille de l'espace de recherche mais celui-ci reste trop grand pour un algorithme exhaustif. Aussi IceCube utilise-t-il une heuristique. Nos mesures montrent que, d'une part, l'heuristique est très efficace, et d'autre part, que les solutions proposées sont très proches de l'optimum.

5.4 Heuristique

Initialement, à chaque action est associé un mérite, estimé à partir de ses contraintes. À chaque itération IceCube sélectionne (de façon aléatoire) entre les actions ayant le mérite le plus élevé. Le mérite des actions restantes est alors mis à jour.

Le mérite d'une action est calculé selon le bénéfice qu'il y a à ajouter cette action au plan courant. Une action aura un mérite d'autant plus élevée qu'elle permettra à d'autres actions d'être ajoutées par la suite. Plus précisément, le mérite d'une action candidate a sera d'autant plus élevé :

1. que la valeur des actions pouvant être exécutées seulement avant a est plus faible,
2. que la valeur des alternatives à a est plus basse,
3. que la valeur des actions mutuellement exclusives à a est plus basse,
4. que la valeur des actions qui ne peuvent être exécutées qu'après a est plus élevée.

Ces facteurs sont listés par ordre d'importance décroissante. Quand une action échoue dynamiquement, son mérite décroît pour les plans suivants.

5.5 Applications sur IceCube

Plusieurs applications ont été développées sous IceCube. Une démonstration réunit trois applications assez simples (réservation d'avion, calendrier et banquaire) en un scénario ad-hoc de réservation de voyages. Deux applications plus complexes sont un gestionnaire de boîtes aux lettres, et un système de fichiers, tous deux répliqués et réconciliables. Nous décrivons brièvement cette dernière application.

DirectoryNode	link/link	link/unlink	unlink/unlink
Different parent, name	¬overlap	¬overlap	¬overlap
Same parent&name	Mut.Excl.	bestOrder	commute

Contrainte dynamique : nom n'existe pas déjà

File	Read/Read	Read/Write	Write/Write
Other file	¬overlap	¬overlap	¬overlap
Same file	commute	bestOrder	Mut.Excl.

Contrainte dynamique : néant

DirectoryNode/File	Unlink/Write	other
Other file	¬overlap	¬overlap
Same file	Mut.Excl.	¬overlap

TAB. 5.1 – Contraintes d'objets dans RFS

5.6 Reconcilable File System (RFS)

L'application Reconcilable File System (RFS) gère un système de fichier à la Unix (un arbre de répertoires et de fichiers), sous réplication optimiste. Deux mises à jour concurrentes sont en conflit si elles violent la sémantique d'un système de fichier. Par exemple le fait de créer deux fichiers sous le même répertoire modifie deux fois l'objet répertoire, mais ne constitue un conflit que si les deux fichiers ont le même nom.

5.6.1 Principes

RFS décompose une commande utilisateur en un parcel d'actions élémentaires. Une action préambule vérifie et analyse les arguments, chacune des suivantes soit connecte soit déconnecte un nœud dans l'arbre. Par exemple pour move, le préambule décide lequel des différents sous-cas possibles (déplacement d'un fichier ou d'un répertoire, vers la même destination ou une autre, etc.) il faut exécuter. Les actions suivantes seront une ou deux déconnexions et une connexion.

Les actions élémentaires sont donc les suivantes : au niveau répertoire, connecter ou déconneter un nœud à un répertoire ; au niveau fichier, lire ou écrire.

Les contraintes d'objet sont résumées dans le tableau 5.1. Elles interdisent de lier dans un même répertoire deux objets (fichier ou sous-répertoire) de même nom. Celles sur les fichiers sont : deux écritures du même fichier sont en conflit, relation bestOrder ordonnant une lecture avant une écriture. Il y a une contrainte supplémentaire reliant les objets fichier et les objets répertoire : il est interdit d'écrire dans un fichier et concouramment de détruire le répertoire contenant le fichier. Enfin, les actions préambule peuvent échouer dynamiquement mais il n'y a pas de contraintes statiques entre elles.

5.6.2 Mise en œuvre

Le code de RFS est basé sur une version dite «solo» qui met en œuvre la fonctionnalité d'un système de fichiers centralisé. Une interface «répliquée» offre la sémantique réconciliable. Chaque commande de la version répliquée exécute d'abord le code solo sur le réplicat local, puis (si le code solo a réussi) journalise les actions correspondantes. Pour chaque effet de bord du code solo (création de nœud, connexion, déconnexion, lecture ou écriture) la version répliquée journalise l'action correspondante. Le code de journalisation est en gros une version modifiée du code solo.

Décomposons par exemple la commande mkdir, qui est suffisamment simple mais néanmoins représentative. La version solo vérifie ses arguments, crée un nouvel objet répertoire, puis connecte celui-ci au répertoire père. Dans le cas de la version solo, ces mêmes effets ont lieu, avec mémorisation de l'identificateur d'objet du nouveau répertoire. Les actions journalisées sont : un préambule (vide dans ce cas particulier très simple), la création d'un nouveau répertoire sous le même identificateur d'objet qu'à l'exécution initiale, enfin la connexion dans le même répertoire père sous le même nom. Ces actions font partie du même parcel et reliées entre elles par predecessorSuccessor.

5.6.3 Discussion

Grâce à la notion de parcel il est possible de décomposer une commande complexe en actions simples. L'avantage est qu'il est beaucoup plus simple de raisonner sur les contraintes d'objet des actions simples.

Nous avons simplifié la réconciliation du système de fichier en identifiant les nœuds par des identificateurs d'objet. Ainsi par exemple un utilisateur pourra renommer un fichier pendant qu'un autre écrit dedans, sans que cela constitue un conflit.

Dans la mise en œuvre actuelle, deux écritures dans le même fichier sont en conflit. Dans le cas d'un fichier de type connu, il serait facile d'empiler des contraintes d'objet spécifiques et moins fortes.

RFS ne comporte que 2 974 lignes de code Java, plus de la moitié pour le code solo : seulement 1 283 lignes (43%) sont spécifiques à la réplication. Le développeur de RFS a donc rajouté la réconciliation par un code très petit, grâce à notre approche systématique.

5.7 Conclusion

IceCube est un système de réconciliation multi-application. Le principe d'IceCube diffère des systèmes précédents sur deux points : d'une part la réconciliation est un problème d'optimisation dirigé par la sémantique (et non par des contraintes temporelles) et de l'autre l'application exprime ses contraintes de façon déclarative et impérative. Les contraintes élémentaires d'IceCube ont un grand pouvoir expressif. Nous avons rendu réconciliables des applications complexes de façon relativement simple.

Ce qui n'a pas été montré ici, faute de place, c'est que l'heuristique d'IceCube est extrêmement efficace (il réconcilie des journaux d'une dizaine de milliers d'actions en quelques secondes) et d'excellente qualité (au moins à 99% de l'optimum).

Enfin, IceCube est indépendant de l'application et réconcilie de façon harmonieuse des applications différentes, qui n'ont pas nécessairement été conçues pour travailler ensemble.

En conclusion, IceCube apporte une amélioration substantielle au domaine des applications réparties à réplication optimiste. IceCube est en phase d'adoption par des produits Microsoft.

Chapitre 6

Conclusion et perspectives

Nous concluons ce bref mémoire avec quelques indications sur nos travaux en cours, qui seront approfondies au cours de l'exposé de soutenance, et un bref résumé des acquis et de l'impact de notre recherche passée.

6.1 Modèle de système répliqué

Le système IceCube, présenté au chapitre précédent, a pour mission de choisir, entre mises à jour concurrentes, un plan non conflictuel optimal. Son fonctionnement en centralisé ne pose aucun problème particulier, mais si plusieurs sites exécutent IceCube (ou tout autre algorithme de choix), rien ne garantit que le système reste cohérent. Il existe plusieurs solutions à ce problème mais aucune n'est intuitivement satisfaisante.

Le problème est plus complexe qu'il n'y paraît. Afin de comparer entre eux les protocoles de cohérence existants, et peut-être d'en inventer de nouveaux, une approche formelle s'impose. Nous définissons de façon formelle les notions d'action, de multi-journal, et de plan d'exécution, soumises au contraintes MustHave $\bowtie$ et Order $\rightarrow$. Nous justifions ce choix par la puissance expressive, démontrée par les applications IceCube, de ces primitives.

Chaque site i a une vue locale M_i du multi-journal. Un sous-ensemble de ce dernier, noté C_i, dénote les actions dont le site i sait qu'elles sont définitivement acquises («committed»). Tout plan d'exécution S sur le site i est soumis à des contraintes d'intégrité :

- Fermé pour MustHave : $\forall a, b \in S : a \in S \wedge a \bowtie b \Rightarrow b \in S$
- Ordonné par Order : $\forall a, b \in S : a \rightarrow b \Rightarrow (a; \ldots; b)$ dans S
- Respecte les contraintes dynamiques : $\forall a \in S$: réussite(a, S)
- Contient les actions acquises : $a \in C_i \Rightarrow a \in S$

Il s'ensuit que l'ensemble des actions acquises C_i est lui-même soumis à des contraintes similaires :

- Fermé pour MustHave : $\forall a, b \in C_i : a \in C_i \wedge a \bowtie b \Rightarrow b \in C_i$
- Ne contient pas de cycle Order : $\forall a, b \in C_i : a \rightarrow b \Rightarrow \neg(b \rightarrow a)$

Une des règles de vivacité impose que tout action acquise sur un site est fatalement acquise sur tous les sites : $\forall a, i, j : a \in C_i \Rightarrow \Diamond a \in C_j$. La combinaison de cette dernière et de la contrainte «Ne contient pas de cycle Order» entraînent la contrainte globale suivante :

$$\forall a, b \in \bigcup_{j \in Sites} C_j : a \rightarrow b \Rightarrow \neg(b \rightarrow a)$$

Cela signifie qu'un site ne peut confirmer une action que s'il est certain qu'aucun autre site ne confirmera une action en conflit avec cette dernière. De plus si les action sont dans des parcel cette prohibition s'étend à l'ensemble des actions d'un parcel.

Le nœud du problème de la cohérence est donc que la dernière contrainte, globale, est difficile à réaliser. Les systèmes existants que nous avons pu analyser, soit utilisent des algorithmes triviaux (par exemple, à tout instant $C_i = \emptyset$), soit centralisent la décision de «commit», soit utilisent un protocole de consensus global. Toutes posent problème, soit pour être trop peu générales, soit pour passer difficilement à l'échelle.

Si aucune des solutions existantes n'est satisfaisante, on voit mal comment faire mieux. Nous proposons nous-mêmes un nouveau protocole, basé sur un compromis entre partitionnement des objets, centralisation du commit à l'intérieur d'une partition, et consensus dans le cas des transactions entre objets appartenant à des partitions différentes. Ce compromis semble bien adapté aux applications réelles mais cela reste à valider.

6.2 Bilan

Au cours des années nous avons exploré des domaines de recherche différents, mais organisés de façon cohérente autour d'un besoin réel : le partage des informations dans un système réparti de grande échelle. Du génie logiciel nous sommes passés au ramasse-miettes réparti, puis à la persistance, enfin à la réplication optimiste.

Dans chaque cas nous sommes partis de besoins applicatifs réels. Sans fausse modestie, notre démarche alliait pragmatisme (car nous avions besoin de solutions effectives et nous les avons mises en œuvre) élégance (car nous avons toujours cherché des solutions simples, générales, et efficaces, aussi bien sur le plan pratique que théorique), et ténacité (car ce n'est que convaincus d'avoir traité un domaine à fond que nous sommes passés à un autre).

L'impact de mon travail le plus aisément mesurable est dans le domaine de la formation, puisque j'ai encadré 14 thèses et j'ai créé et enseigné le module «Systèmes Répartis Avancés» du DEA de Systèmes Informatiques de Paris-6, qui a formé plusieurs dizaines d'étudiants. En ce qui concerne l'animation de la communauté scientifique, j'ai été à l'origine de la création de plusieurs conférences (I-WOOOS, ERSADS et même un peu ECOOP), j'ai été vice-président de l'ACM SIGOPS, j'ai contribué à amener les conférences SOSP et HotOS en Europe, et j'ai été le créateur de l'ASF, Association SIGOPS de France. Enfin, on se référera à ma liste de publications pour compter les conférences de qualité où j'ai publié.

Sur le plan pratique enfin, ma contribution est non négligeable. J'ai été l'inventeur du concept de mandataire *(proxy)* qui est devenu incontournable dans les systèmes répartis. Le système SOS a eu une influence non négligeable sur les développements ultérieurs de systèmes à objets. Nos algorithmes de ramasse-miettes et de persistance ont souligné les problèmes réels du passage à l'échelle, et ont fait l'objet d'applications industrielles avec le système PerDiS. Enfin, les résultats d'IceCube sont en phase d'industrialisation dans des produits Microsoft.

6.3 Perspectives

Avec le développement d'Internet, la recherche sur les systèmes répartis est plus importante que jamais. Nous avons fait le tour de plusieurs problèmes importants mais néanmoins partiels, et nous sommes loin d'en avoir épuisé les possibilités.

Le passage à l'échelle heurte bien des obstacles. Certains sont techniques, comme on vient de le voir avec notre succinte analyse de la cohérence de la réplication optimiste. D'autres sont sociétales ou organisationnels, comme les problèmes d'équité, de sécurité, d'héterogénéité, ou la multiplication des pare-feux. Ces obstacles rendent illusoire l'idée d'un système universel, d'égal à égal *(peer-to-peer)* de partage de l'information, dont on a pu rêver.

Heureusement ce ne sera pas nécessaire. En pratique le partage se divise entre données largement lues, dont la diffusion ne pose aucun problème pratique (le Web fait très bien l'affaire) et des données à la fois lues et écrites, qui ne sont partagées qu'à un petit nombre d'exemplaires. Si le nombre de participants simultanés ne semble donc pas problématique, la sécurité, la latence, le nombre d'objets, et les transactions inter-objets restent des difficultés.

Enfin, comme toujours, le chemin de la recherche au consommateur est long. Une recherche pointue et approfondie se focalise sur un problème à la fois, au risque de négliger le système dans son ensemble, bref regarde l'arbre au lieu de la forêt. Pour maximiser son impact, le chercheur en système doit rechercher un délicat compromis, entre innovation (travail en profondeur) et adoption (travail en largeur).

Bibliographie

[1] V. Abrossimov, M. Rozier, et M. Shapiro. Generic virtual memory management for operating system kernels. Dans *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 123–136, Litchfield Park AZ (USA), décembre 1989. ACM.

[2] Akamai. Site web. `http://www.akamai.com`.

[3] Laurent Amsaleg, Olivier Gruber, et Michael Franklin. Efficient incremental garbage collection for workstation-server database systems. Dans *Proc. 21st Very Large Data Bases (VLDB) Int. Conf.*, Zürich (Switzerland), septembre 1995.

[4] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, et R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4) :360–365, 1983.

[5] Aline Baggio. *Adaptable and Mobile-Aware Distributed Objects.* Thèse de doctorat, Université Paris 6, Paris, France, juin 1999.

[6] Aline Baggio et Guillaume Pierre. Welcome to SOR. `http://www-sor.inria.fr/`.

[7] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, et Arthur Secret. The World-Wide Web. *Communications of the ACM*, 37(8) :76–82, août 1994.

[8] A. D. Birrell et B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Programming Languages and Systems*, 2(1), février 1984.

[9] Andrew Birrell, Greg Nelson, Susan Owicki, et Edward Wobber. Network objects. *Software Practice and Experience*, S4(25) :87–130, décembre 1995. `http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr%-115.html`.

[10] Xavier Blondel. *Gestion de la mémoire dans un environnement réparti persistant à grande échelle : l'exemple de PerDiS.* Thèse de doctorat, Conservatoire National des Arts et Métiers (CNAM), Paris, France, octobre 2000.

[11] Hans-Juergen Boehm et Mark Weiser. Garbage collection in an uncooperative environment. *Software — Practice and Experience*, 18(9) :807–820, septembre 1988.

[12] Georges Brun-Cottan. *Gestion de cohérence dans de (petits) groupes de processus coopérants éloignés.* Thèse de doctorat, Université Pierre et Marie Curie — Paris 6, jan 1998.

[13] Luis-Felipe Cabrera, Vince Russo, et Marc Shapiro, éditeurs. *1991 International Workshop on Object Orientation in Operating Systems*, Palo Alto CA (USA), octobre 1991. IEEE, IEEE Computer Society Press. IEEE Computer Society Press Order Number 2265.

[14] Jonathan E. COOK, Alexander L. WOLF, et Benjamin G. ZORN. Partition selection policies in object database garbage collection. Dans *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 371–382, Minneapolis MN (USA), mai 1994. ACM SIGMOD.

[15] Peter DICKMAN et Mesaac MAKPANGOU. A refinement of the fragmented object model. Dans *1992 Int. Workshop on Object Orientation and Operating Systems*, pages 230–234, Dourdan (France), octobre 1992. IEEE Comp. Society, IEEE Comp. Society Press.

[16] Peter DICKMAN, Mesaac MAKPANGOU, et Marc SHAPIRO. Contrasting fragmented objects with uniform transparent object references for distributed programming. Dans *5th European SIGOPS Workshop, on "Models and Paradigms for Distributed Systems Structuring"*, Mont Saint-Michel (France), septembre 1992. ACM SIGOPS, IRISA, INRIA-Rennes.

[17] Daniel EDELSON. *Type-specific storage management.* Phd thesis, University of California at Santa Cruz, juin 1993.

[18] R. S. FABRY. Capability-based addressing. *Communications of the ACM*, 17(7) :403–412, juillet 1974.

[19] Paulo FERREIRA. *Larchant : ramasse-miettes dans une mémoire partagée répartie avec persistance par atteignabilité.* Thèse de doctorat, Université Paris 6, Pierre et Marie Curie, Paris (France), mai 1996. `http://www-sor.inria.fr/publi/ferreira_thesis96.html`.

[20] Paulo FERREIRA et Marc SHAPIRO. Garbage collection and DSM consistency. Dans *Proc. of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 229–241, Monterey CA (USA), novembre 1994. ACM. `http://www-sor.inria.fr/publi/GC-DSM-CONSIS_OSDI94.html`.

[21] Paulo FERREIRA et Marc SHAPIRO. Modelling a distributed cached store for garbage collection. Dans *12th Euro. Conf. on Object-Oriented Prog. (ECOOP)*, Brussels (Belgium), juillet 1998. `http://www-sor.inria.fr/publi/MDCSGC_ecoop98.html`.

[22] Paulo FERREIRA, Marc SHAPIRO, Xavier BLONDEL, Olivier FAMBON, Joâo GARCIA, Sytse KLOOSTERMAN, Nicolas RICHER, Marcus ROBERTS, Fadi SANDAKLY, George COULOURIS, Jean DOLLIMORE, Paulo GUEDES, Daniel HAGIMONT, et Sacha KRAKOWIAK. PerDiS : design, implementation, and use of a PERsistent DIstributed Store. Rapport technique QMW TR 752, CSTB ILC/98-1392, INRIA RR 3525, INESC RT/5/98, QMW, CSTB, INRIA and INESC, octobre 1998. `http://www-sor.inria.fr/publi/PDIUPDS_rr3525.html`.

[23] Fabrice Le FESSANT, Ian PIUMARTA, et Marc SHAPIRO. An implementation of complete, asynchronous, distributed garbage collection. Dans *Conf. on Prog. Lang. Design and Impl. (PLDI)*, Montreal (Canada), juin 1998. ACM SIGPLAN. `http://www-sor.inria.fr/publi/ICADGC_pldi98.html`.

[24] M. FISHER, N. LYNCH, et M. PATTERSON. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2) :274–382, avril 1985.

[25] Philippe GAUTRON et Marc SHAPIRO. Two extensions to C++ : A dynamic link editor and inner data. Dans *Proceeding and additional papers, C++ Workshop*, Berkeley, CA (USA), novembre 1987. USENIX.

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

[27] Sabine HABERT. *Gestion d'objets et migration dans les systèmes répartis.* Thèse de doctorat, Université Paris-6, Pierre-et-Marie-Curie, Paris (France), décembre 1989.

[28] Sabine HABERT, Laurence MOSSERI, et Vadim ABROSSIMOV. COOL : Kernel support for object-oriented environments. Dans *ECOOP/OOPSLA'90 Conference*, volume 25 de *SIGPLAN Notices*, pages 269–277, Ottawa (Canada), octobre 1990. ACM.

[29] Maurice HERLIHY. Optimistic concurrency control for abstract data types. Dans *Proceedings of the 5th Symposium on the Principles of Distributed Computing*, pages 206–217, Vancouver (Canada), août 1986. ACM.

[30] Richard JONES et Rafael LINS. *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, Royaume-Uni, 1996. ISBN 0-471-94148-4.

[31] Anne-Marie KERMARREC, Antony ROWSTRON, Marc SHAPIRO, et Peter DRUSCHEL. The IceCube approach to the reconciliation of divergent replicas. Dans *20th Symp. on Principles of Dist. Comp. (PODC)*, Newport RI (USA), août 2001. ACM SIGACT-SIGOPS. `http://research.microsoft.com/research/camdis/Publis/podc2001.pdf`.

[32] Butler W. LAMPSON. Hints for computer system design. *IEEE Software*, 1(1) :11–31, janvier 1984.

[33] Fabrice le FESSANT. *Conception et mise en œuvre d'un système à agents mobiles*. Thèse de doctorat, École Polytechnique, Palaiseau, France, décembre 2001.

[34] Fabrice le FESSANT, Ian PIUMARTA, et Marc SHAPIRO. A detection algorithm for distributed cycles of garbage. Dans *OOPSLA W. on Garbage Collection and Memory Management*, Atlanta, GA, USA, octobre 1997. `http://www-sor.inria.fr/publi/DADCG_gcmm97.html`.

[35] Jean-Pierre LE NARZUL. *Le nommage de ressources réparties sur de grandes étendues géographiques*. Thèse de doctorat, Université de PARIS XI-Orsay, Orsay, France, décembre 1993.

[36] T. LE SERGENT et B. BERTHOMIEU. Incremental multi-threaded garbage collection on virtually shared memory architectures. Dans *Proc. Int. Workshop on Memory Management*, numéro 637 dans Lecture Notes in Computer Science, pages 179–199, Saint-Malo (France), septembre 1992. Springer-Verlag.

[37] Barbara LISKOV, Mark DAY, et Liuba SHRIRA. Distributed object management in Thor. Dans *Proc. Int. Workshop on Distributed Object Management*, pages 1–15, Edmonton (Canada), août 1992.

[38] Julien MAISONNEUVE. Hobbes : liaison flexible avec des objets répartis. Dans *Actes des deuxièmes journées des jeunes chercheurs en systèmes répartis*, Rennes (France), octobre 1995. Institut de Recherche en Informatique et Systèmes Aléatoires.

[39] Julien MAISONNEUVE. *Hobbes : un modèle de liaison de réfénces réparties*. Thèse de doctorat, Université Paris 6, Paris, France, octobre 1996.

[40] Julien MAISONNEUVE et Marc SHAPIRO. Implementing efficient indirections. Dans *Broadcast 1st Open Workshop*, Newcastle England, octobre 1993.

[41] Julien MAISONNEUVE, Marc SHAPIRO, et Pierre COLLET. Implementing references as chains of links. Dans *1992 Int. Workshop on Object Orientation and Operating Systems*, pages 236–243, Dourdan (France), octobre 1992. IEEE Comp. Society, IEEE Comp. Society Press.

[42] Mesaac MAKPANGOU, Yvon GOURHANT, Jean-Pierre LE NARZUL, et Marc SHAPIRO. Fragmented objects for distributed abstractions. Dans T. L. CASAVANT et M. SINGHAL, éditeurs, *Readings in Distributed Computing Systems*, pages 170–186. IEEE Computer Society Press, juillet 1994.

[43] Mesaac Mounchili MAKPANGOU. *Protocoles de communication et programmation par objets : l'exemple de SOS.* Thèse de doctorat, Université Paris VI, Paris (France), février 1989.

[44] Karim R. MAZOUNI. *Étude de l'invocation entre objets dupliqués dans un système réparti tolérant aux fautes.* Thèse de doctorat, École Polytechnique Fédérale de Lausanne (EPFL), Lausanne (Suisse), novembre 1996.

[45] MICROSOFT. Asp.net. Site Web. `http://www.asp.net/`.

[46] S. J. MULLENDER et A. S. TANENBAUM. The design of a capability-based distributed operating system. *The Computer Journal*, 29(4) :289–299, 1986.

[47] Ian PIUMARTA. Stub-Scion Pair Chains. `http://www-sor.inria.fr/projects/sspc/`.

[48] David PLAINFOSSÉ. *Distributed Garbage Collection and Reference Management in the Soul Object Support System.* Thèse de doctorat, Université Paris-6, Pierre-et-Marie-Curie, Paris (France), juin 1994. Available from INRIA as TU-281, ISBN-2-7261-0849-0.

[49] David PLAINFOSSÉ et Marc SHAPIRO. A survey of distributed garbage collection techniques. Dans *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), septembre 1995. `http://www-sor.inria.fr/publi/SDGC_iwmm95.html`.

[50] Nuno PREGUIÇA, Marc SHAPIRO, et Caroline MATHESON. Efficient semantics-aware reconciliation for optimistic write sharing. Rapport technique MSR-TR-2002-52, Microsoft Research, Cambridge (UK), mai 2002. `http://research.microsoft.com/scripts/pubs/view.asp?TR_ID=MSR-TR-2002-5%2`.

[51] PROJET PERDIS. `http://www.perdis.esprit.ec.org/`, 1997.

[52] Nicolas RICHER. *Étude du comportement mémoire d'applications persistantes coopératives.* Thèse de doctorat, Université Paris 6, Paris, France, mai 2002.

[53] Michel RUFFIN. *Kitlog : Un Service de Journalisation Générique.* Thèse de doctorat, Université Pierre et Marie Curie, Paris VI, Paris (France), septembre 1992. Available from INRIA as TU-205, ISBN-2-7261-0759-1.

[54] Yasushi SAITO et Marc SHAPIRO. Replication : Optimistic approaches. Rapport technique HPL-2002-33, Hewlett-Packard Laboratories, mars 2002. `http://www.hpl.hp.com/techreports/2002/HPL-2002-33.html`.

[55] Marc SHAPIRO. *Une méthode de conception progressive des systèmes parallèles utilisant le langage C.S.P.* Thèse de docteur-ingénieur, Institut National Polytechnique de Toulouse, E.N.S.E.E.I.H.T., Toulouse, France, septembre 1980.

[56] Marc SHAPIRO. Structure and encapsulation in distributed systems : the Proxy Principle. Dans *Proc. 6th Intl. Conf. on Distributed Computing Systems*, pages 198–204, Cambridge, Mass. (USA), mai 1986. IEEE.

[57] Marc SHAPIRO. A binding protocol for distributed shared objects. Dans *Proc. Int. Conf. on Distributed Computing Systems*, Poznan (Poland), juin 1994.

[58] Marc SHAPIRO et Paulo FERREIRA. Larchant-RDOSS : a distributed shared persistent memory and its garbage collector. Dans J.-M. HÉLARY et M. RAYNAL, éditeurs, *Workshop on Distributed Algorithms (WDAG)*, numéro 972 dans Springer-Verlag LNCS, pages 198–214, Le Mont Saint-Michel (France), septembre 1995. `http://www-sor.inria.fr/publi/LRDSPMGC_wdag95.html`.

[59] Marc SHAPIRO, Yvon GOURHANT, Sabine HABERT, Laurence MOSSERI, Michel RUFFIN, et Céline VALOT. SOS : An object-oriented operating system — assessment and perspectives. *Computing Systems*, 2(4) :287–338, décembre 1989.

[60] Marc SHAPIRO, Sytse KLOOSTERMAN, et Fabio RICCARDI. PerDiS — a persistent distributed store for cooperative applications. Dans *Proc. 3rd Cabernet Plenary W.*, Rennes (France), avril 1997. `http://www-sor.inria.fr/publi/PPDSCA_cabernet97.html`.

[61] Marc SHAPIRO, Fabrice LE FESSANT, et Paulo FERREIRA. Recent advances in distributed garbage collection. Dans S. KRAKOWIAK et S. K. SHRIVASTAVA, éditeurs, *Recent Advances in Distributed Systems*, volume 1752 de *Lecture Notes in Computer Science*, chapitre 5, pages 104–126. Springer-Verlag, février 2000. `http://www-sor.inria.fr/publi/RAIDGC_lncs1752.html`.

[62] Marc SHAPIRO, Antony ROWSTRON, et Anne-Marie KERMARREC. Application-independent reconciliation for nomadic applications. Dans *Proc. SIGOPS European Workshop : "Beyond the PC : New Challenges for the Operating System"*, Kolding (Denmark), septembre 2000. ACM SIGOPS. `http://www-sor.inria.fr/~shapiro/papers/sigops-ew-2000-logmerge.html`.

[63] Hervé SOULARD. *Adaptation des systèmes de stockage aux besoins des utilisateurs : l'approche micro-systèmes de stockage et sa mise en œuvre dans BOSS.* Thèse de doctorat, Université Paris 6, Pierre et Marie Curie, Paris (France), novembre 1995.

[64] Andrew S. TANENBAUM. *Distributed Operating Systems.* Prentice Hall, 1995.

[65] Paul R. WILSON. Uniprocessor garbage collection techniques. Dans *Proc. Int. Workshop on Memory Management*, numéro 637 dans Lecture Notes in Computer Science, Saint-Malo (France), septembre 1992. Springer-Verlag. `ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps`.

[66] Ann WOLLRATH, Roger RIGGS, et Jim WALDO. A distributed object model for the Java system. Dans *Conf. on Object-Oriented Technologies*, Toronto, Ontario (Canada), juin 1996. Usenix.

[67] V. YONG, J. NAUGHTON, et J. YU. Storage reclamation and reorganization in client-server persistent object stores. Dans *Proc. Data Engineering Int. Conf.*, pages 120–133, Houston TX (USA), février 1994.