Cohérence de données répliquées partagées par un groupe de processus coopérant à distance

Georges Brun-Cottan

THÈSE

présentée à

L'Université Pierre & Marie Curie – Paris VI

pour obtenir le titre de

Docteur de l'Université Paris VI

Spécialité :

Systèmes Informatiques

par

Georges Brun-Cottan

Sujet de la thèse :

Cohérence de données répliquées partagées par un groupe de processus coopérant à distance

Soutenue le 30 septembre 1998, devant le jury composé de :

Messieurs	Michel	Banâtre	*Président*
	Rachid	Guerraoui	*Rapporteurs*
	Michel	Riveill	
	Maarten	van Steen	*Examinateurs*
	Bertil	Folliot	
	Claude	Girault	
	Mesaac	Makpangou	

Résumé

La progression des applications coopératives est confrontée, de façon inhérente, à la latence des moyens de communication. Lorsque celle-ci devient importante, les modèles de cohérence traditionnels entraînent un manque de réactivité rédhibitoire. Nous attaquons ce problème, en réduisant une application coopérative à une collection de réplicats coopérants, puis en se focalisant sur la gestion de cohérence d'un tel groupe de réplicats.

Ce problème est important par deux aspects : par son application dans tous les domaines impliquant la coopération d'individus et par son caractère fondamental dans la structuration et la compréhension des mécanismes de coopération.

Notre étude critique, des critères de cohérence associés aux cohérences dites « faibles », embrasse quatre domaines : les systèmes transactionnels, les mémoires partagées réparties, les objets concurrents et les plate-formes de communication. Notre thèse contribue sur trois points :

1. Notre modèle d'exécution est libre de tout a priori concernant la causalité des opérations. Ce modèle est basé sur des histoires répliquées.
2. Notre modèle de partage, la *réplication coopérante*, dérivée de la réplication active, n'impose pas un ordre commun unique sur l'exécution des opérations. Les réplicats sont autonomes et ne coopèrent que lorsque leur vue de l'histoire globale ne suffit plus à garantir la correction de l'application.
3. Nos principes systèmes permettent de construire un nouveau type de composant, le *gestionnaire de cohérence*. Ce composant :
   - prend en charge la coopération des réplicats. Il implante la partie complexe de la gestion de cohérence : le contrôle de la distribution, de la réplication et de la concurrence ;
   - maintient, sur l'ordre réparti des opérations, des propriétés déterministes. Ces propriétés définissent un *contrat de cohérence* ; elles peuvent être utilisées comme critère de correction ;
   - est choisi *à l'exécution* par l'application ;
   - est *réutilisable*.

Nous avons réalisé **Core**, une plate-forme de développement complète, partiellement documentée et accessible sur FTP, développée au-dessus d'Unix. **Core** offre, outre les services usuels nécessaires à la mise en œuvre de groupes de processus répartis, une bibliothèque extensible de gestionnaires de cohérence. **Core** offre aussi de nombreuses classes, utilisées tant pour la réalisation de nouveaux gestionnaires que pour l'expression de nouveaux types et modèles d'exécution, par les concepteurs d'applications. Nous avons réalisé, avec **Core**, deux applications : une application d'édition coopérative basée sur Emacs et une simulation de ressource partagée.

Mots clés : Systèmes Répartis, Gestion de Cohérence, Objets Répliqués, Contrôle de Concurrence Réparti.

Coherence of Shared Replicated Data within Groups of Remote Cooperating Processes

Cooperative applications must communicate in order to make progress, and are therefore inherently affected by communication latency. In wide-area settings this latency can increase to a point where traditional consistency management no longer provides sufficient reactivity — in some cases making the application unusable. We address this problem by considering a cooperative application to be a collection of cooperating replicas, and then concentrate on the management of consistency within such a group of replicas.

This problem is significant for two reasons: it appears in any application domain where we have cooperation between human agents, and is a fundamental part of the structure of cooperation mechanisms.

Our comprehensive study of "weak consistency" covers four domains: transactional systems, distributed shared memories, distributed shared objects, and platforms for group communication. Our contribution has three significant aspects:

1. We present a new model of execution that is free of causal dependencies. It is based on replicated histories, and is used to formalise the consistency criteria presented in this document.
2. We present a new model of sharing, *cooperative replication* — a derivative of active replication that does not impose a unique global ordering on the execution of operations. Replicas are autonomous except when their local view of the global computation can no longer guarantee correct behaviour of the distributed application.
3. We present a few system principles with which we define a new component, the *consistency manager*. This component:
   - manages the cooperation between replicas and implements the most complex parts of consistency management: the control of distribution, replication and concurrency;
   - provides a *consistency contract* that imposes deterministic properties on distributed operations. These can be considered as the "correctness criteria" for the distributed application;
   - is bound dynamically to the application at *execution time*; and
   - is *reusable*.

We have implemented **Core** — a complete development environment conforming to the above principles — that runs on Unix and is freely available by FTP. **Core** provides the usual services for distributed applications (a name service, group communication and management of group membership, for example), as well as an extensible class library with which specific consistency managers can be developed. It also provides numerous classes with which application designers can implement new kinds of shared object. Two example applications have been built using **Core**: an extension to Emacs that permits collaborative editing, and a simple manager for shared resources.

Keywords: Distributed systems, Consistency Management, Replicated Objects, Distributed Concurrency Control.

Je remercie sincèrement Rachid GUERRAOUI et Michel RIVEILL d'avoir accepté de rapporter sur cette thèse. Je suis très reconnaissant à Michel BANÂTRE, Michel RIVEILL, Maarten VAN STEEN, Bertil FOLLIOT, Claude GIRAULT et Mesaac MAKPANGOU d'avoir accepté d'être membre de mon jury.

Quelques personnes ont eu la tâche, Ô combien laborieuse, de déchiffrer, corriger, relire, suggérer, enfin, de transformer un amas de caractères, en un document lisible voire peut-être compréhensible. Je remercie particulièrement Mesaac MAKPANGOU, bien sûr, mais aussi Claude GIRAULT, Fabrice LE FESSANT, Bertil FOLLIOT, Marc SHAPIRO et ma famille, en particulier mon père, Jean-Claude BRUN-COTTAN, ma mère Francoise BRUN-COTTAN et ma sœur aînée Marguerite-France VU-TRAN pour un travail qui touchait à l'héroïsme.

Je remercie du fond du cœur Claude GIRAULT qui a su être là au bon moment, et qui a su s'accommoder avec une grande tolérance d'un hurluberlu d'autant plus pressé qu'il avait mis longtemps pour toucher bon port.

Que ceux qui ont une femme adorable et deux merveilleux petits bambins imaginent ce que peut être la vie de cette famille lorsque l'individu censé leur servir de père leur consacre moins d'un journée par semaine. Non, mieux ne vaut pas. Permission éternelle leur est donnée de me châtier de toute cette ingratitude. Mes beaux-parents M^{r} et M^{me} GOUINAUD ont évidemment une part de responsabilité importante ; par leur gentillesse et leur disponibilité perpétuelle, ils m'ont permis de poursuivre un travail somme toute, très égoïste.

Trois personnes ont joué un rôle exceptionnel dans cette thèse. Dominique FORTIER, tout d'abord, qui a reconnu dans les pérégrinations laborieuses et sans fin d'un étudiant en cinquième année de médecine, un échec patent pour la médecine mais aussi une passion, l'informatique. Norbert COT, ensuite, directeur à l'époque du Magistère d'Informatique Appliquée d'Île de France (M.I.A.I.F), qui a su faire confiance à un étudiant pétri d'échec et accepté de me prendre dans le magistère.

Quant à Mesaac MAKPANGOU, que dire ? Je lui dois tant. Avant de le rencontrer je terminais un diplôme ; après, je débutait une passion ! Mesaac a su me montrer la science derrière la perpétuelle vitrine technologique de l'informatique. Nous avons communié dans une curiosité sans borne. Il m'a soutenu à bout de bras, avec une extrême patience, en me conservant une confiance intacte, même dans les moments difficiles, qui pourtant ne manquent pas dans une thèse de cette durée. Mesaac est un condensé d'humanité, d'optimisme et d'imagination. Son amitié m'est très chère.

Un thésard est une bête un peu bizarre, un peu un adolescent de la recherche. Tout à la fois clamant son autonomie et recherchant avidement une protection ; constamment sur ses gardes, il fait souvent montre d'une grande ingratitude. Je remercie très sincèrement Marc SHAPIRO d'avoir accueilli et supporté un tel animal, au projet SOR, à l'I.N.R.I.A. Je lui suis infiniment reconnaissant de m'avoir ouvert très généreusement les portes de l'expérience américaine.

Je remercie l'ensemble des membres du projet SOR qui m'ont supporté pendant ces trop longues années. Je serais toujours ému du rôle décisif que chacun a joué jusqu'au dernier instant. Beaucoup sont devenus de bon camarades qui seront toujours là, présents.

L'I.N.R.I.A. est une structure merveilleuse en ce qui concerne la pratique de la recherche. Je suis reconnaissant à tout ceux qui y font vivre l'esprit d'émulation et d'ouverture qui y règnent.

Quand je t'aurais désappris à espérer, je t'apprendrais à vouloir.
– Sénèque –
Trad. libre Compte-Sponville (L'amour, la solitude – page 49)

À Catherine, mon étoile, nettement plus attirée par l'humanité pratique que par les grands délires humanistes,

À Gaëlle et Nicolas mes deux énormes rayons d'amour,

À Francoise et à Jean-Claude, ma mère et mon père, à Christianne et à Jacques, ma grand-mère et mon grand-père et à mes deux sœurs, Marguerite-France et Hélène qui tous m'ont légué cet optimisme inébranlable et cette joie de vivre. Quels plus beaux cadeaux à partager ?

Avant-propos

Les applications coopératives forment le pendant informatique d'un principe d'organisation du travail qui est à la base du développement social et économique de nos sociétés. Le courrier électronique, les forums de discussion thématiques, les jeux multi-utilisateurs ont été de loin, les premières applications développées[1] sur Internet, bien avant même de parler de Web ou de multimédia !

Pour autant, le développement de ce type d'application n'a pas notablement progressé, ceci, malgré le développement prodigieux d'Internet. Un grand nombre d'applications grande échelle voient déjà le jour, telles les applications de téléconférence multimédia à grande échelle ou le commerce électronique. Mais les applications coopératives ne semblent pas progresser de la même façon.

Pourtant, de nombreux éléments de coopération semblent exister dans la vie sociale non informatisée, et même avoir une place prédominante dans le circuit économique (entreprise, stade) alors que leurs pendants informatiques n'existent pas, ou si peu (simulations ou jeux multi-utilisateurs, entreprise virtuelle). Nous pensons que les systèmes informatiques actuels n'ont pas les propriétés qui permettent de développer de tels équivalents. Notre sentiment est que l'évolution technologique d'Internet ne suffit pas, et ne suffira pas, aux applications coopératives : si le débit du réseau de communication grande distance a été multiplié par plus de 1000 en dix ans, si le nombre de sites connectés double tous les ans, et si son taux de pannes diminue constamment, cela permet avant tout d'améliorer le débit d'informations applicatives et la nature de ces informations, en particulier celles dont la taille est importante (sons, images). Cette amélioration bénéficie bien évidemment à toutes les applications, mais ne semble pas résoudre le problème essentiel des applications coopératives : la *latence* de communication.

La latence est le trait qui distingue fondamentalement les applications coopératives des autres applications réparties, telles la « navigation web » ou encore la téléconférence : si les agents coopérants sont géographiquement répartis, ils doivent communiquer pour coopérer et donc pour progresser. Et, alors même que la maîtrise de cette latence de communication est primordiale, elle est, de fait, fondamentalement limitée par la vitesse de la lumière. Si on peut espérer que sa dépendance technologique (partition et surcharge du réseau) puisse disparaître à relativement court terme, il n'est pas prévisible actuellement qu'une information puisse se propager plus rapidement que la vitesse de la lumière : un photon ne peut faire l'aller-retour Paris-Tokyo ($\sim$ 2*10 000 km) en moins de 66 ms. Dans ce même temps, une machine moderne ($\sim$ 300 Mips) exécute 20 millions d'instructions.

Cette limitation suffit, sur un réseau à l'échelle de notre planète, à poser des problèmes pour

[1] Le courrier électronique, première application d'ARPANET, date de 1972 [LCC+97]. *Usenet* date de 1979 [BEH+94], MUD1 ont été écrit par Richard Bartle et Roy Trubishaw entre 1979 et 1980 [Smi97, Sou93] et Sceptre d'Alan Klietz a la même époque [Smi97].

la réalisation d'applications coopératives : une machine concurrente, répartie entre Paris et Tokyo, sérialisant naïvement toutes ses opérations, ne pourrait effectuer plus de 16 opérations par seconde, soit $\sim$ 20 000 fois moins qu'un Apple 2 (300 Kips) du début des années 1980.

En face d'une telle limite, il ne reste qu'une issue : comprendre la coopération. Quand, comment et quelles informations un ensemble de processus doit-il s'échanger, pour conserver son intégrité, pour constituer *un* système ?

Ces questions sont fondamentales. Elles dépassent le cadre de la coopération de processus informatiques. Ce fait explique certainement la fascination qu'a exercé sur nous ce sujet de thèse.

Notre contribution reste infime au regard de ce problème. Nous espérons seulement partager, avec le lecteur, un regard, que nous espérons original, sur la coopération à distance de processus et sur la manière de la mettre en œuvre dans un système informatique réparti.

Table des matières

Table des figures

Chapitre 1

Introduction

Une application coopérative consiste en un groupe d'agents accédant et modifiant concurremment une collection d'informations incarnées dans un système informatique. Il peut s'agir des parties d'un document, des emplacements de temps d'un agenda ou d'avatars dans un monde virtuel. Bien que la notion d'agent soit vague, elle sous-entend une entité douée d'un comportement intelligent et munie d'une certaine autonomie.

Intéressons nous au cas où les agents sont fixes et coopèrent par l'intermédiaire d'un système réparti asynchrone. Un tel cas est représentatif des applications coopératives dont les agents sont des éléments du monde physique, hommes, robots ou machines. Dans ce cadre, une application coopérative se modélise comme un groupe de processus asynchrones répartis.

Dans ce cadre, coopérer à grande distance posent deux problèmes non triviaux :

1. La latence de communication est en moyenne élevée. Elle dépasse couramment 200 msec sur de bonnes liaisons intercontinentales, et la seconde dès que l'on utilise des modems au-dessus du réseau téléphonique commuté.

2. L'environnement d'exécution est non prédictible. De nombreux facteurs concourent à ce manque de prédictibilité. Les applications, et en particulier les applications coopératives, sont développées sans connaître précisément dans quels environnements elles seront utilisées. Les paramètres de communication associés à cet environnement, en particulier le débit et la latence, peuvent varier de plusieurs ordres de grandeur selon que les agents coopérants sont situés sur un même réseau local ou dispersés sur plusieurs continents. De plus, même lorsque cet environnement peut-être anticipé, dès lors que la coopération, s'effectue à grande distance, le débit et la latence peuvent aussi aisément varier d'un ordre de grandeur, et ce au cours de l'exécution même. Ces variations ne sont pas dues uniquement à l'infrastructure, mais aussi au changement de composition du groupe. Les agents coopérants sont autonomes, et peuvent joindre et quitter le groupe à tout moment. L'arrivée dans un groupe, constitué de membres travaillant sur le même réseau local, d'un membre travaillant à domicile, ou bien situé sur un autre continent, change considérablement la latence moyenne. Certains outils algorithmiques, tels ceux utilisés pour réaliser l'agrément entre membres, sont particulièrement sensibles à ces variations.

Ces deux problèmes ont un impact déterminant sur deux aspects de la coopération : la représentation de l'information partagée et la gestion de la cohérence associée à cette information.

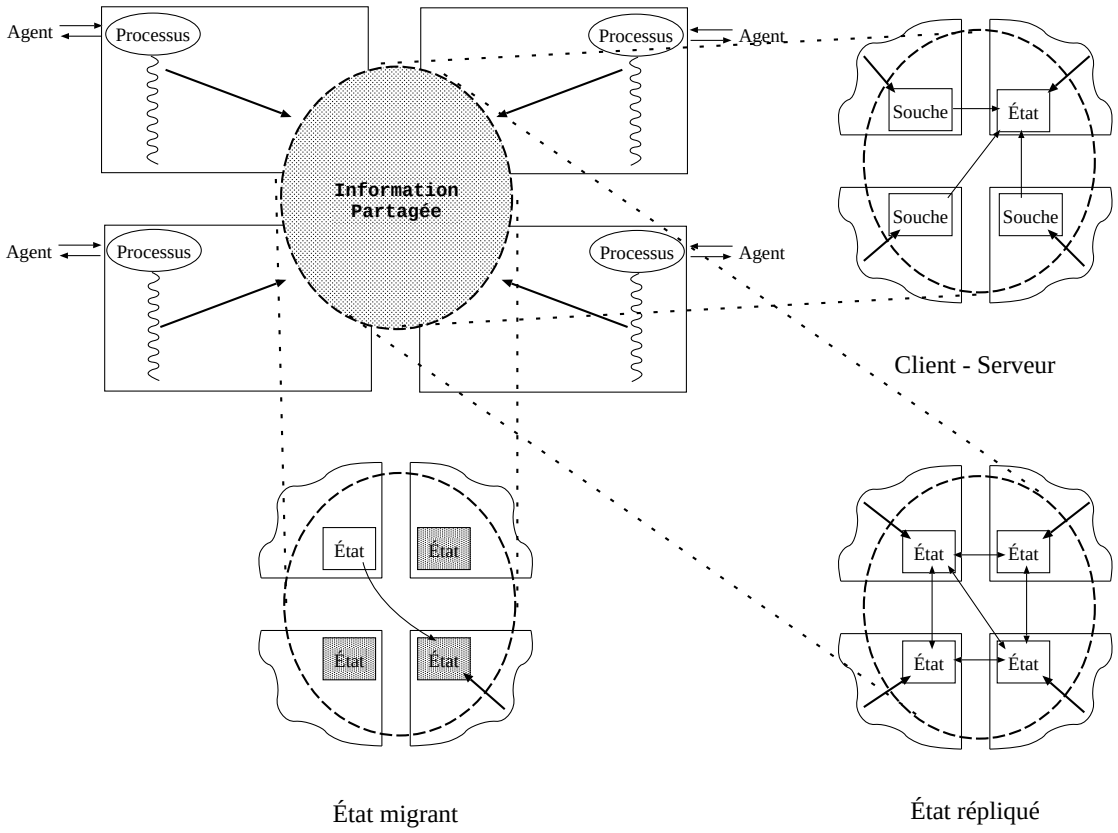


FIG. 1.1: Les diverses représentations traditionnelles d'une information partagée par des agents coopérants. Chaque agent interagit localement (clavier, fenêtre X, ...) avec un processus qui lui est dédié. Les différents processus sont sur des sites distants ; leur interaction est soumise à une latence importante.

La latence moyenne élevée remet en cause le principe d'une représentation centralisée : dans cette représentation, chaque accès à l'information partagée est soumis à cette latence élevée[1]. Les agents sont fixes ; sans autre a priori sur la structure de l'application, seule la réplication de l'information, par chaque agent, permet, par « effet de cache », d'éviter que chacun des accès d'un agent subisse cette latence. Les agents accèdent ainsi localement à l'information partagée (cf. figure 1.1).

Le problème n'est pas résolu pour autant : si les accès à l'information partagée peuvent être locaux grâce à la présence d'un réplicat, encore faut-il que le groupe de réplicats représente encore l'information partagée comme un tout, comme une seule information unique. La concurrence des accès, comme la multiplicité de la représentation de l'information entraîne une divergence entre l'état de l'information tel que représenté par les réplicats et celui attendu par l'application. La gestion de cohérence consiste en l'ensemble des fonctions contribuant à assurer que la *représentation* de l'information partagée reste conforme à l'information elle-même, telle qu'identifiée au travers de la sémantique de l'application. Deux fonctions de cette gestion

[1] Pour les applications non coopérantes, cette latence n'a qu'un effet capacitif ; elle retarde la perception des opérations distantes, mais ne ralentit pas la vitesse de progression de l'application.

de cohérence nous intéressent particulièrement : le contrôle de concurrence et le contrôle de divergence. Ces deux fonctions s'exercent en contrôlant les états de l'information partagée, perçus et modifiés par l'application. Elles s'assurent que la succession de ces états respectent les propriétés qui fondent sa correction. Dans un système asynchrone, ces propriétés reposent sur les ordres dans lesquels chaque agent perçoit ses propres actions et celles des autres agents [MB89, BM93b] et [Bir96, chapitre 13 et 16]. On appelle *modèle de cohérence*, l'ensemble des contraintes imposées par la gestion de cohérence sur ces ordres.

LAMPORT, dans [Lam79], a identifié un modèle de cohérence particulier, la cohérence séquentielle, jouant un rôle central dans les systèmes répartis : très grossièrement, un système réparti est dit séquentiellement cohérent, s'il assure aux applications une exécution produisant un résultat équivalent à ce qu'il serait si le système était centralisé[2]. Ce modèle possède des propriétés remarquables : 1) il impose des contraintes sur le médium représentant l'information partagée (originellement, la mémoire), donc sur le système et non sur l'application, 2) le modèle assure l'équivalence de propriétés applicatives. Ce faisant, ce modèle permet au « système » de prendre en charge la gestion de cohérence, et assure la transparence de la répartition à l'application. Dans une certaine mesure, ce modèle décharge donc l'application de la gestion de cohérence.

Ces bonnes propriétés amènent une question : à quel « prix » sont-elles assurées ? Plus précisément, ce modèle de cohérence est-il compatible avec les contraintes de réplication, de coopération, et de latence de communication exhibées par « notre » environnement d'exécution ?

Une tentative de réponse sera détaillée dans la première partie de la thèse. Peu d'applications coopératives semblent pouvoir conserver une réactivité acceptable en utilisant une cohérence séquentielle ; les exemples sont nombreux, en ce qui concerne les forums, les gestions de courrier ou d'agenda répartis, les jeux ou même des applications systèmes tels que la gestion de fichiers ou des serveurs de noms répartis. De telles applications sont inutilisables sur des implantations séquentielles, en pratique, dès que les agents ne sont pas sur le même réseau local. Ce manque de réactivité semble inhérent à la cohérence séquentielle : les contraintes que ce modèle de cohérence impose sur l'ordre des accès répartis nécessitent, en pratique, des attentes correspondant à l'acquisition d'une autorisation par un ou plusieurs agents distants.

Le problème est détaillé dans la première partie de ce document, mais brièvement, dans le cadre de la coopération distante, la réplication ne peut être bénéfique qu'aux applications se satisfaisant de cohérences non séquentielles, ce que l'on désigne informellement sous le nom de « cohérences faibles ». Nous verrons dans le chapitre suivant quelques exemples d'ordres moins contraignants.

L'élément important de cette première analyse est que les alternatives à la cohérence séquentielle induisent toutes des dépendances avec le code applicatif. Cette dépendance nous fait perdre la plupart des bonnes propriétés de la cohérence séquentielle ; elle a pour effet de complexifier notablement la gestion de cohérence.

Le caractère non prédictible des paramètres de communication tels que la latence, le débit ou le taux de partition, mais aussi celui des schémas d'accès applicatifs, complique notablement le problème. Ces paramètres varient d'une façon que l'on ne peut prévoir lors de la conception, ni même lors de la compilation du programme. Leurs variations ont un impact

[2] la spécification exacte de la cohérence séquentielle est plus restrictive. Elle sera vue précisément dans le chapitre introduisant la cohérence

qui dépend non seulement du modèle de cohérence mais aussi de sa mise en œuvre. Des agents, exécutant à tour de rôle de nombreux accès, exhibent une localité temporelle qui peut être exploitée par l'utilisation d'un jeton. Supposons que ces accès soient mis dans un tampon dont la propagation est contrôlée par la transmission du jeton. La latence ou les partitions du réseau ne seront sensibles que durant les transferts de ce jeton. Mais le gain à attendre d'une telle propagation paresseuse n'est pas trivial. Il dépend lui-même du temps nécessaire à la transmission de ces paramètres ; celui-ci dépend aussi bien du nombre d'accès tamponnés, du débit moyen de la communication au moment de l'exécution, de la taille des fenêtres d'acquittement associées au contrôle de flux ainsi qu'à la retransmission des messages perdus, et de bien d'autres paramètres *de l'exécution*. Prenons le cas d'un jeu réparti. Il est sensé et même usuel de sacrifier, dans une certaine mesure, le réalisme spatial pour conserver une bonne interactivité (un bon réalisme temporel). Ce « sacrifice » est implanté en permettant une certaine divergence sur les positions des avatars telles qu'observées par les divers agents du jeux à un instant donné. Or ces divergences et leurs effets sur l'interactivité sont fonction de ces paramètres non prédictibles.

Finalement, le coût des paramètres fondamentaux impliqués dans la correction de l'application ne sont pas connus avant l'exécution. Ces paramètres dépendent bien sûr des critères de correction, mais aussi de l'infrastructure d'exécution, comme de l'usage qui est fait de l'application. Or ce coût a un impact déterminant sur la réactivité de l'application, donc sur le fait qu'elle soit utilisable ou non.

L'élément important de cette deuxième analyse est que l'application doit pouvoir contrôler, *à l'exécution*, son compromis *cohérence/réactivité* et décider *à l'exécution* de la mise en œuvre de ce compromis.

Nous nous trouvons donc en face d'un code complexe qui doit pouvoir être contrôlé à l'exécution, sur des critères à la fois applicatifs et environnementaux.

1.1 Notre thèse

Dans ce document, nous développons la thèse que la gestion de cohérence associée à l'information répliquée doit être choisie à l'exécution par l'application, ce que nous qualifions *d'adaptable*, et que, du fait de sa complexité, elle doit être implantée dans le système. Elle devient ainsi *réutilisable*.

Nous donnons des éléments de solution permettant d'asseoir cette thèse. Ces éléments se basent sur l'hypothèse forte que les données partagées possèdent une sémantique déterministe, i.e. l'état de chaque donnée est défini de façon unique par son état initial et la séquence ordonnée des opérations appliquées à cette donnée. Les instances de types abstraits de donnée, ce que nous appelons des objets, respectent habituellement cette hypothèse.

D'autre part, nous considérons le critère de correction applicatif comme étant connu du programmeur de l'application. Notre contribution ne consiste pas en la détermination de ce critère de correction mais seulement en une aide à sa mise en œuvre, une fois celui-ci connu.

Enfin, nous ne traitons pas les pannes. Nous considérons que soit leur prise en compte peut se satisfaire de notre architecture, soit leur apparition est supportable par nos applications (jeux/simulations répartis, éditions coopératives, espaces virtuels partagés).

Nous proposons une étude détaillée de l'état de l'art de la gestion de cohérence et de sa mise en œuvre. Cette étude embrasse quatre domaines : les systèmes transactionnels, les

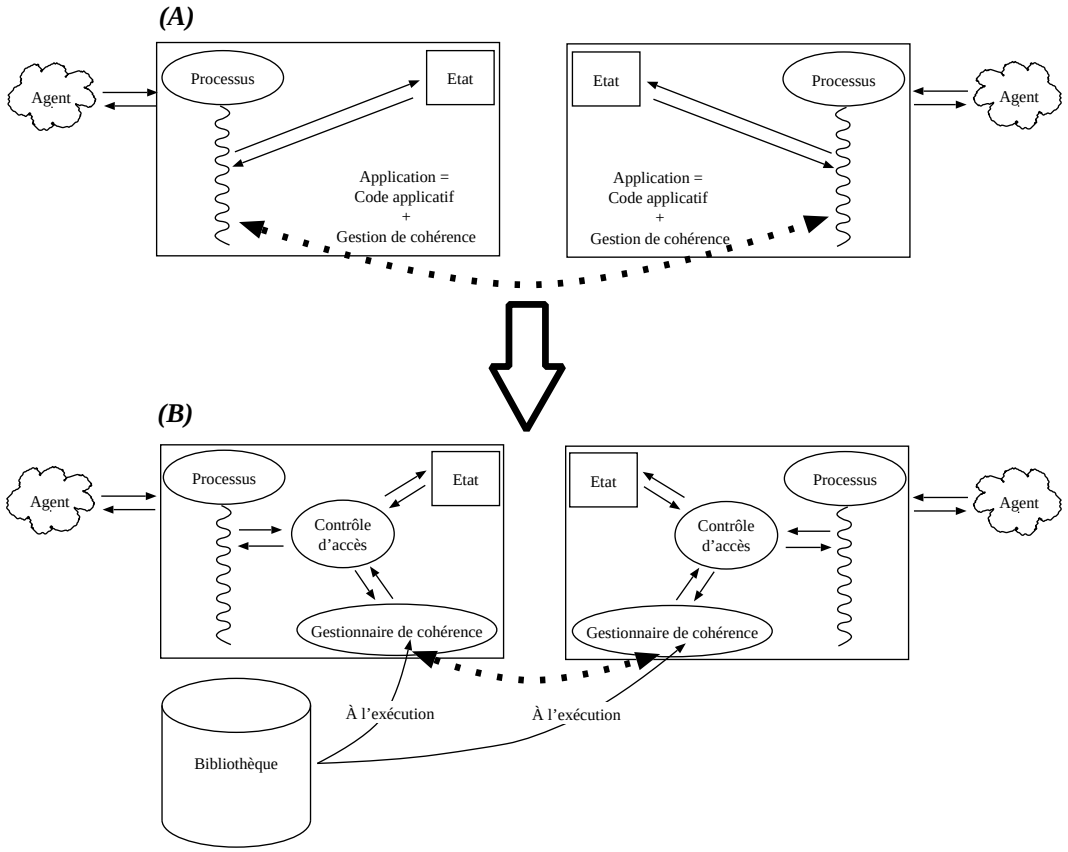


FIG. 1.2: Passer d'un code ***(A)*** applicatif intégrant le problème applicatif et la gestion de cohérence (concurrence, réplication et communication), à un code ***(B)*** où cette gestion de cohérence est factorisée en un composant choisi à l'exécution.

---

mémoires partagées réparties, les objets concurrents et les plate-formes de communication. Cette étude fonde la construction d'un nouveau modèle de partage, la *réplication coopérante*, adapté aux contraintes de nos applications et de notre environnement : chaque agent possède un réplicat ; ces réplicats sont autonomes et coopèrent si besoin est, pour assurer la correction de l'application. Nous avons développé un nouveau modèle d'exécution, basé sur des histoires répliquées. Ce modèle permet de représenter les propriétés pertinentes sur lesquelles baser le choix du modèle de cohérence et sa mise en œuvre. Enfin, nous identifions quelques principes systèmes qui permette de «factoriser» la gestion de cohérence. Cette factorisation découpe la tâche de la gestion de cohérence en trois fonctions : une fonction déterminant l'ordre des opérations perçues localement par le réplicat, une fonction déterminant quand les réplicats doivent coopérer, et une fonction déterminant les caractères de la coopération, en particulier, les contraintes sur l'ordre global de *perception* des opérations. Les deux premières fonctions sont spécifiques des données répliquées et du code applicatif, mais elles n'intègrent pas d'aspects de la répartition dans leur mise en œuvre. La dernière fonction intègre tout le code chargé de la répartition, du contrôle de concurrence et de la réplication. Sa mise en œuvre, par contre, est complètement indépendante du code applicatif.

Ces éléments de solutions permettent de construire un composant «système», le *gestion-*

naire de cohérence, prenant en charge, la partie complexe du code assurant la gestion de cohérence. Ce code assure des propriétés d'ordre, de « bout en bout », sur l'exécution des opérations. Il est choisi à l'exécution par l'application. Enfin, il est réutilisable.

Ce composant est le point clé d'une architecture, où les gestionnaires de cohérence constituent un librairie extensible. Idéalement, dans cette architecture, les applications coopérantes décident, à la compilation, de la structure de donnée représentant l'information partagée, puis choisissent, à l'exécution, dans la librairie, le gestionnaire offrant le modèle et la mise en œuvre adaptés au maintien de la cohérence de la représentation répliquée de cette structure de donnée (cf figure 1.2). Ces gestionnaires de cohérence permettent à une application d'exprimer son compromis cohérence/réactivité tout en garantissant sa correction. La conception de cette architecture donne lieu à une analyse de problèmes de conception peu abordés dans la littérature. Ces problèmes sont, par exemple, les contraintes imposées par l'encapsulation des représentations des réplicats sur les politiques d'emballage, les stratégies de mise à jour et le contrôle de concurrence à grain fin.

Pour valider cette architecture, et ses concepts sous-jacents, nous avons développé, en C++, au-dessus d'Unix, plusieurs prototypes. Le dernier prototype, **Core**, intègre l'essentiel des éléments architecturaux présentés. **Core** est un acronyme de « cohérence d'objet répliqué ». Il constitue un support d'exécution réparti complet, offrant une bibliothèque de tâches portable, un service de nommage, une plate-forme de communication de groupe, un service d'emballage et de déballage, des multi-RPC et de nombreuses classes permettant de développer de gestionnaires de cohérence. Le support d'exécution est partiellement documenté et peut être téléchargé par FTP (cf. annexe B).

Trois gestionnaires de cohérence sont actuellement disponibles dans la bibliothèque. Ils implantent tous les trois, des variantes de la cohérence hybride (« hybrid consistency » [AF92]).

Core intègre de nombreuses classes, permettant d'implanter des objets répliqués au dessus des gestionnaires de cohérence. Ces classes, associées aux gestionnaires de cohérence, constituent une architecture réflexive, adhérant aux principes d'implantation ouverte (« open implementation ») développés par Kiczales [Kic96, KP].

Nous avons réalisé, avec **Core**, deux applications : une application d'édition coopérative basée sur Emacs et une simulation de ressource partagée.

1.2 Plan de la thèse

Ce document est organisé en trois parties :

1. La première partie assoit la problématique de cette thèse : elle précise les termes du compromis cohérence/réactivité et précise les aspects liés à sa mise en œuvre. Elle est composée de trois chapitres qui introduisent la gestion de cohérence, ses optimisations et la technologie de l'adaptabilité qui lui est associée. L'état de l'art associé embrasse les quatre principaux domaines de l'informatique répartie : les systèmes transactionnels, les mémoires partagées réparties, les objets concurrents et les plate-formes de communication.
2. La deuxième partie présente nos éléments de solution. Elle comporte quatre chapitres. Le chapitre 6 présente notre nouveau modèle de coopération, la *réplication coopérante* et étudie ces propriétés. Le chapitre 7 présente les principes « systèmes » permettant de dégager et de formaliser les trois fonctions de la gestion de la cohérence à la base de la factorisation. Le chapitre 8 présente le rôle et l'interface des trois composants centraux

de l'architecture : les *intentions*, les *activités* et le gestionnaire de cohérence. Enfin, le chapitre 9 présente l'architecture **Core** et les aspects les plus marquant de sa mise en œuvre dans le prototype.

3. La troisième partie traite de la validation de nos propositions. Le chapitre 10 présente la mise en œuvre d'une infrastructure pour les objets répliqués au dessus de l'architecture **Core**. Le chapitre 11 présente les éléments les plus significatifs des deux applications mises en œuvre au dessus de **Core**, relativement à la gestion de cohérence. Le chapitre 12 présente une évaluation des performances du prototype, en s'efforcant d'évaluer le coût lié à la factorisation de la gestion de cohérence.

Première partie

Compréhension du problème

Chapitre 2

Introduction à la gestion de cohérence

Notre propos dans ce chapitre est d'introduire ce qu'est la fonction de gestion de cohérence, dans un système réparti asynchrone.

2.1 Cohérence et correction

La cohérence est une propriété de correction, c'est-à-dire une propriété statuant sur la propension d'un système à agir conformément à ses spécifications. La correction est le critère premier auquel s'intéresse tout utilisateur ou concepteur d'un système informatique : elle conditionne l'existence tant du système que de l'application.

Les systèmes d'exploitation répartis traditionnels, de même que les applications coopératives, sont des systèmes réactifs complexes [Weg92]. Nous sommes loin d'avoir les outils qui permettent de les spécifier complètement. Nous voyons tout juste émerger ceux qui permettent de s'assurer de la conformité de leur mise en œuvre vis-à-vis d'une telle spécification.

La gestion de cohérence se cantonne, traditionnellement, au maintien des propriétés de correction remise en cause par la *concurrence*, la *réplication* et les pannes.

Dans ce document nous ne traitons pas le cas des pannes. Nous considérons que soit leur prise en compte peut se satisfaire de notre architecture, soit leur apparition est supportable par nos applications (jeux/simulations répartis, éditions coopératives, espaces virtuels partagés)[1].

Le besoin de spécifier un critère de correction et d'assurer ce critère pour des systèmes concurrents n'est pas nouveau. Les systèmes concurrents sont difficiles à programmer et difficiles à certifier. Depuis leur apparition, la communauté scientifique a cherché à concevoir des outils qui permettent de les programmer facilement et des propriétés sur lesquelles fonder leur correction. En terme de programmation, les sémaphores puis les moniteurs dans les systèmes à processus concurrents et les transactions ACID[2] [GR93] dans les systèmes transactionnels sont les outils les plus connus. En terme de propriétés, l'atomicité [Mos93, Wei84, Her90, HW90] et la séquentialité [Lam79, MRZ94] dans le cadre des systèmes de processus concurrents et la sérialisibilité [EGLT76, BHG87] dans le cadre des systèmes transactionnels sont les plus connues.

[1] Les fautes fréquentes dans les petits groupes distants, i.e. les désordonnancements ou les pertes de messages, peuvent être traitées de façon autonome par l'infrastructure de communication. Les partitions temporaires peuvent être vécues comme une simple dégradation de service.

[2] ACID est l'acronyme de : Atomicité vis à vis des pannes, Cohérence, Isolation, Durabilité.

L'atomicité et la séquentialité sont des propriétés portant sur l'infrastructure de communication entre processus (mémoire, objet, réseau). Ces deux propriétés masquent à l'application la concurrence existant au niveau de la mise en œuvre de cette infrastructure (multiprocesseurs, multimachines, multiprogrammation). La sérialisibilité [EGLT76, BHG87] définit des contraintes sur l'ordre dans lequel peut s'exécuter les opérations appartenant à des transactions concurrentes. Ces contraintes assurent aux transactions concurrentes l'illusion de s'exécuter seules, en isolation. Ce critère d'isolation permet aussi d'assurer la pérennité des contraintes d'intégrité des bases de données, dès lors que chaque transaction est correcte.

Finalement, le terme de cohérence « forte » désigne traditionnellement, de façon informelle et vague ces diverses propriétés. Un des traits communs de ces modèles de cohérences fortes est leur transparence. Ces modèles sont conçus dans la volonté de simplifier la programmation des systèmes concurrents ; ils définissent des propriétés sur les abstractions systèmes (mémoire, transactions) qui assurent cette transparence. Cette transparence rend la correction du code applicatif indépendante de la mise en œuvre de ces abstractions systèmes.

Ces propriétés sont hautement souhaitables ; pourquoi ne sont-elles pas adaptées aux applications conçues comme des groupes de processus coopérants éloignés ? Prenons, par exemple, un ordonnanceur construit sur un verrouillage à deux phases [BHG87, chapitre 3] ; quand un tel ordonnanceur reçoit de son processus une opération, il doit, avant de pouvoir ordonner son exécution, s'assurer qu'aucun ordonnanceur distant n'a déjà ordonné une opération incompatible ; s'assurer d'une telle condition nécessite donc au moins un aller-retour réseau (agrément au dessus du réseau) ; durant ce temps, la progression de l'application est stoppée ; durant ce temps, dans le cas de processus dispersés sur plusieurs continents plus de 10^7 opérations locales peuvent être effectuées.

Les modèles de cohérence « fortes » reposent implicitement sur un coût négligeable de la *perception* des accès concurrents. La latence qui sépare la perception par un processus des actions effectuées par un autre est la raison essentielle pour laquelle ces propriétés ne sont pas adaptées.

A-t-on des alternatives à ces propriétés ? Quelles sont-elles ? Pour comprendre les nombreuses propositions apportées par la communauté scientifique, il nous faut tout d'abord comprendre comment fonctionne un groupe de processus répartis, et tout particulièrement, comment un processus observe les opérations et coopère avec les autres processus membres du groupe coopérant.

2.2 Exemple introductif

Nous donnons un exemple trivial du problème[3] posé par la concurrence puis par la réplication.

Considérons le cas d'un couple, un homme h et une femme f, partageant un compte commun C muni d'une somme $S(C)$, et décidant de faire chacun un retrait (R_h et R_f) dans deux distributeurs de billets distincts. Supposons que ces deux retraits particuliers sont tels que $R_h < S(C), R_f < S(C), R_h \neq R_f, R_h + R_f > S(C)$. Supposons que la banque assigne, à l'application qui pilote la délivrance des billets, la contrainte de ne pas délivrer plus d'argent que le couple n'en a sur son compte. Sur une requête de retrait d'un distributeur de billets, une telle application implante la séquence suivante : lecture de l'état du compte (l), vérification que

[3] Le lecteur trouvera dans les premiers chapitres du livre de Gray et Reuter [GR93] de même que dans celui de Bernstein et al. [BHG87] une introduction détaillée du problème.

la valeur lue est supérieure à celle demandée (v), mise à jour du compte (m) et autorisation du retrait (a).

Considérons la séquence d'opérations suivante :

$$l_h(C) \bullet l_f(C) \bullet v_h \bullet v_f \bullet m_h(C) \bullet m_f(C) \bullet a_h \bullet a_f$$

La banque autorise le retrait, mais la valeur du compte commun est erronée. Le retrait de l'homme n'a pas été décompté, ce bien que l'argent lui ait été distribué. Pour être correcte, chaque séquence $l \bullet v \bullet m$, doit être *atomique*. C'est au contrôle de concurrence qu'incombe la tâche d'assurer cette garantie. On parle, traditionnellement, d'*histoire* pour désigner une séquence ordonnée d'opérations.

Supposons maintenant, que l'application implante chaque séquence $l \bullet v \bullet m$ de façon atomique mais que, pour des raisons de performance, chaque compte soit *répliqué* sur un distributeur de billets ; nous avons donc deux réplicats (C_1 et C_2), c'est-à-dire deux représentations du même compte commun C. Supposons que l'homme fasse son retrait sur le réplicat C_1 du premier distributeur de billet, et la femme sur le second. À l'évidence, les opérations modifiant un réplicat doivent être transmise à l'autre réplicat, pour s'assurer que chacun des réplicats représentent toujours le même compte commun du couple. C'est au protocole de réplication, encore appelé protocole de contrôle de divergence qu'incombe cette tâche.

Notons e_h, l'opération qui consiste à envoyer la valeur du premier réplicat sur le second, après le retrait de l'homme. Notons e_f l'opération correspondante pour la femme. Notons ru_h, la réception de e_h et sa mise à jour correspondante sur C_2. Notons ru_f, l'opération équivalente pour la femme.

Considérons les histoires suivantes des réplicats :

$$\begin{aligned} \mathcal{H}_1 = &\quad l_h(C_1) \bullet v_h \bullet m_h(C_1) \bullet e_h \bullet a_h \bullet ru_f \\ \mathcal{H}_2 = &\quad l_f(C_2) \bullet v_f \bullet m_f(C_2) \bullet e_f \bullet a_f \bullet ru_h \end{aligned}$$

Ni l'atomicité des séquences $l \bullet v \bullet m$, ni même la transmission des modifications aux autres réplicats, ne suffit à assurer la correction : la banque autorise les deux retraits. De plus, les deux réplicats terminent avec des valeurs différentes, et ne représentent plus la valeur du compte commun.

Informellement, pour être correct, l'une des lectures, au choix, effectuée sur un réplicat doit *percevoir* le retrait effectué sur l'autre réplicat, i.e. doit suivre l'opération ru correspondante sur son réplicat. Ainsi, dans le cas général, la propagation des mises à jour ne peut se faire arbitrairement, de façon indépendante des conditions de concurrence. On appelle traditionnellement *gestion de cohérence*, la fonction qui assure *in fine* la correction de l'application, c'est-à-dire le contrôle de concurrence, le contrôle de divergence et leur interaction. Cette fonction est centrée sur la gestion de l'ordonnancement des opérations du système [BHG87, BM93b, MB89, Bir96]. Toute la difficulté de sa mise en œuvre vise à assurer cette gestion en tenant compte des contraintes du système (débit, latence, partition, représentation de la donnée) et en permettant une vitesse d'exécution satisfaisante à l'application.

2.3 Du système et de la cohérence

Poser le problème de la gestion de cohérence en termes de propriétés de correction remises en cause par la concurrence et la réplication, laisse supposer l'existence d'une mise en œuvre de référence, correcte en l'absence d'accès concurrent et en l'absence de représentations multiples

de l'information. C'est clairement le cas de notre exemple introductif. C'est de fait, l'hypothèse implicite sur laquelle repose l'essentiel des travaux de cohérence établis durant ces 20 dernières années. Ces approches définissent des contraintes à imposer au « système » qui lui permettent d'assurer aux applications, une exécution « équivalente » à ce qu'elle serait dans un système centralisé. La cohérence séquentielle et la sérialisibilité sont caractéristiques de cette approche. C'est le fait de demander au « système » d'assurer cette propriété d'équivalence qui garantit la transparence de la répartition aux applications.

Cette approche repose implicitement sur trois hypothèses :

1. Que les applications reposent, pour leur mise en œuvre, sur des fonctions implantées par le système d'exploitation. LAMPORT présente [Lam86] cette approche comme correspondant à la construction d'un système à partir d'un système de plus bas niveau. On peut ainsi identifier une structure système sur laquelle on peut imposer des contraintes permettant d'assurer la correction des applications.
2. Qu'il existe une version de référence de l'application, typiquement non répliquée et accédée de façon non concurrente. C'est bien sûr le cas, lorsque les applications fonctionnent en isolation. La concurrence et la réplication ne sont, alors, que des techniques permettant d'assurer de meilleures performances (parallélisme) et/ou de la tolérance aux fautes.
3. Que les performances obtenues permettent d'utiliser l'application.

L'extension des réseaux de communication a permis d'une part le développement des communications exhibant des latences élevées et d'autre part l'apparition de principes coopératifs dans les applications réparties. Ces deux développements remettent en cause les deux dernières hypothèses sur lesquelles repose l'approche traditionnelle de la transparence.

Tous les domaines de l'informatique se sont trouvés confrontés à cette remise en cause. De nombreuses propositions ont été élaborées, qui prennent en compte tant les aspects coopératifs que le coût des synchronisations inter-processus induites par la latence. Dans le domaine transactionnel, il s'agit d'alternatives à la sérialisibilité [GMW82, RP95], de l'intégration du modèle objet [LCJS87, HW90, HW91, Cat91, PSWL95] ou encore de l'apparition de modèles de transaction étendus [BK91, CR94]. Dans les modèles non transactionnels, il s'agit de l'apparition des modèles de mémoires à cohérence dite faible [Mos93, TK97], des plate-formes de communication de groupe [Sec96], de protocole de réplication *ad hoc* [Mau90, LLG92, LA92]. Nous présentons les propositions les plus significatives dans les deux chapitres suivants. Tous les domaines ont vu, parallèlement, l'émergence de nombreuses propositions explorant les modèles d'exécution asynchrones et les stratégies de gestion de cohérence optimiste [Guy91, GL93, TTP+95].

De façon heureuse, la première de nos trois hypothèses semble persister. Un système complexe est traditionnellement construit au dessus de « sous-systèmes ». Cette hypothèse semble correspondre à une tendance lourde de la démarche humaine. L'homme appréhende les problèmes complexes en les subdivisant en problèmes plus simples. L'homme construit des systèmes complexes à partir de systèmes plus simples.

La question que se pose, dès lors, tout architecte de système réparti est : existe-t-il un noyau de propriétés « universelles » sur lequel reposerait les propriétés applicatives ? Une fois identifiées, ces propriétés pourraient servir à spécifier le comportement des fonctions du système d'exploitation utilisées par le code applicatif. De telle propriétés permettraient de retrouver une partie des bonnes propriétés des modèles de cohérence forte ; en particulier, le confinement de la complexité dans les composants du système.

Cette quête est illustré par l'intérêt porté à l'ordonnancement des opérations dans la correction d'un calcul concurrent réparti. La gestion de la représentation des données, comme la communication entre processus sont des services systèmes, et ces deux services ont un impact sur l'ordre dans lequel les opérations, et plus particulièrement les opérations distantes, sont perçues.

2.4 Modélisation et observation d'un système réparti

Un système réparti est traditionnellement modélisé comme une collection de processus asynchrones communiquant par messages [Lam78, DDS87, CT96] : chaque processus exécute séquentiellement et de façon atomique des événements considérés de trois types : interne au processus, envoi d'un message et réception d'un message. Le rapport des vitesses d'exécution des processus ainsi que les délais d'acheminement des messages sont finis mais non bornés.

LAMPORT a, le premier, introduit ce modèle dans [Lam78]. Il a introduit, dans ce même article, la relation « happened before » qui fonde la plupart des travaux sur l'étude des propriétés globales dans un système réparti, en particulier la causalité [Mat89, BM93a, Sec96]. LAMPORT définit la relation « happened before » (notée $\rightarrow$) comme la relation minimale satisfaisant les quatre[4] conditions suivantes :

1. Si a et b sont deux événements du système exécutés sur un même processus et que a est exécuté avant b, alors $a \rightarrow b$.
2. Si a est l'envoi d'un message par un processus, et b la réception de ce même message par un autre processus, alors $a \rightarrow b$.
3. $\rightarrow$ est transitive.
4. $\rightarrow$ est irréflexive (il n'existe aucun événement a du système tel que $a \rightarrow a$).

$\rightarrow$ définit donc un ordre partiel strict sur l'ensemble des événements du système ; les événements non ordonnés par la relation $\rightarrow$ sont dit *concurrents*. Nous faisons référence à ce modèle comme à ceux basés sur la relation « happened before » en tant que « modèle de Lamport ».

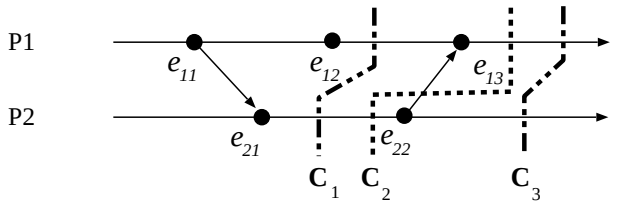
Le modèle de Lamport vise à définir le fonctionnement d'un système réparti. La relation « happened before » définit les propriétés de correction minimales sur les événements du système pour que ce dernier soit « sensé ».

2.4.1 Représentation d'une exécution répartie

Dans le modèle de Lamport, chaque processus exécute séquentiellement une collection d'événements uniques. Une exécution répartie est donc identifiée 1) par l'ensemble E des événements exécutés par chacun des processus et 2) par l'ordre dans lequel ils ont été exécutés par les processus. On désigne traditionnellement par histoire locale, la séquence ordonnée des événements exécutés par un processus. De même, on désigne traditionnellement par histoire globale, $(E, \rightarrow)$, l'ensemble des événements du système, partiellement ordonné (« poset ») par la relation « happened before ».

Lorsque certains événements sont concurrents, il n'est pas possible d'identifier un ordre unique, total, dans lequel ont été *effectivement* exécuté les événements du système. Néanmoins, il est toujours possible de construire un tel ordre total en effectuant un tri topologique

[4] La quatrième condition n'est pas intégrée dans la définition même de la relation par LAMPORT, mais est imposée peu après, dans le même article.



$$
\begin{aligned}
E &= \{e_{11}, e_{12}, e_{13}, e_{21}, e_{22}, \dots\} \\
C_1 &= \{e_{11}, e_{12}, e_{21}\} \\
C_2 &= \{e_{11}, e_{12}, e_{13}, e_{21}\} \\
C_3 &= \{e_{11}, e_{12}, e_{13}, e_{21}, e_{22}\}
\end{aligned}
$$

FIG. 2.1: Diverse coupes de l'exécution d'un système réparti à deux processus P_1 et P_2. Les événements à l'origine et à destination d'une flèche liant deux événements appartenant à deux processus distincts sont respectivement l'émission d'un message et sa réception. C_1 et C_3 sont des coupes causalement cohérentes. C_2 est une coupe mais n'est pas causalement cohérente, car $e_{22} \rightarrow e_{13}$ et e_{22} n'est pas dans la coupe C_2 alors qu'elle contient e_{13}.

respectant la relation « happened before ». Un tel ordre inclut la relation « happened before », et ordonne de façon arbitraire les événements concurrents. Un tel ordre est représenté par l'horloge scalaire de LAMPORT [Lam78].

Cet ordre entretient un rapport étroit avec la cohérence séquentielle. Supposons associée, à chaque processus, une mémoire privée. Supposons que tout événement interne d'un processus consiste soit en une lecture, soit en une écriture sur cette mémoire. Supposons que les processus communiquent deux à deux par des canaux particuliers. Implantons ces canaux de communication comme des éléments d'une mémoire partagée par les processus. Un envoi de message d'un processus p_1 à un processus p_2 se traduit par une écriture sur l'élément de mémoire représentant le canal de communication p_1–p_2. La réception d'un message de p_1 par p_2 se traduit par la lecture de ce même élément de mémoire, par p_2. Considérons la mémoire constituée de l'ensemble de ces éléments de mémoire (privée comme partagée). Cette mémoire est séquentielle si, pour toute exécution du système réparti, les lectures sont légales[5] et si on peut construire un tri topologique respectant la relation « happened before ».

Bien sûr, la mémoire que nous venons de construire est particulière, et de nombreux points sont laissés de côté.[6] Néanmoins, la comparaison fait apparaître ces premiers liens entre la cohérence d'une structure de donnée partagée et l'ordre des événements, ou opérations, effectués par les processus y accédant.

2.4.2 Observation d'une exécution répartie

Le modèle que nous venons de définir ne permet de statuer que sur des exécutions globales *complètes*. Dans la plupart des cas, on veut pouvoir s'assurer incrémentalement de la correction du système ; soit parce que la mise en œuvre envisagée se base sur des mécanismes pessimistes (sémaphore, verrouillage à deux phases) soit plus simplement parce que les processus peuvent ne pas se terminer. On veut donc pouvoir statuer de la correction du système sur l'observation d'états intermédiaires. Or l'absence de synchronisme comme l'absence d'ordonnancement global des événements sont incompatibles avec le principe d'une observation globale intermédiaire précise de l'état du système. Le mieux que l'on puisse faire est de collecter un ensemble d'observations locales, encore appelé prise de vue ou « snapshot ».

[5] Une lecture sur un élément de mémoire est dite légale si elle retourne la *dernière* valeur écrite sur cet élément ; *dernière* est à prendre ici relativement à l'ordre construit par la relation « happened before ».

[6] Le lecteur trouvera une introduction formelle très didactique dans [Lam86, CBCP95]

Les coupes [Mat89, BM93b] représentent de telles prises de vues. Une coupe d'un système réparti est un sous-ensemble de E tel que : si un événement e, exécuté par un processus p_i, est présent dans cet ensemble, alors tous les événements exécutés avant e par ce processus p_i doivent être présents dans cet ensemble ; autrement dit, une coupe garantit pour chaque événement la présence de tous ses prédécesseurs *locaux*. Une coupe est dite cohérente, ou encore causalement cohérente, si pour tout événement, elle assure la présence de son prédécesseur au sens de la relation « happened before » (cf. figure 2.1). Une coupe causalement cohérente garantit donc, pour chaque événement, la présence de tous ses prédécesseurs, *locaux* comme *globaux*. Dès lors que le système réparti répond au modèle de LAMPORT, seules les coupes causalement cohérentes peuvent représenter un état global intermédiaire plausible du système. C'est la raison pour laquelle l'étude des propriétés réparties se base traditionnellement sur ces coupes cohérentes, en écartant d'office les autres coupes.

Les coupes causalement cohérentes sont les outils de base permettant l'étude des algorithmes répartis basés sur des propriétés globales [Mat93, BM93b]. Ces algorithmes sont, par exemple, la construction d'un temps virtuel global, l'implantation d'élections, la détection de terminaison d'un algorithme de ramasse-miettes réparti, la constitution de la vue des membres d'un groupe ou la vue globale de la charge d'un groupe d'unités centrales.

2.5 Les propriétés fondatrices de la correction dans un système réparti

Dans le cadre des systèmes répartis asynchrones, deux principes sont reconnus comme structurant la progression et la correction d'une application : le consensus[7] [HM90, NT93] et la causalité [BJ87, Mat89, RM93, BM93b]. Le consensus permet aux membres d'un groupe de décider sur un choix unique commun à tous, de transcender leur autonomie pour se conduire comme une unité, et ainsi de converger vers une action, un comportement ou un but commun. La causalité [BJ87, Mat89, RM93, BM93b] rend compte du fait qu'un événement est la conséquence d'un autre ; la causalité est perçue comme le caractère porteur de l'irréversibilité d'une exécution ou encore de l'irréversibilité du temps [Pri96]. Dans un système asynchrone, donc sans représentation autonome du temps, l'ordre dans lequel sont exécutées les opérations et, plus finement, la causalité[8] entre les opérations est la propriété porteuse du temps.

La coopération fait implicitement référence à ces deux principes : on construit ces actions en fonction de celle des autres (causalité) ; on construit un but commun (consensus).

L'importance de ces principes dans les systèmes répartis tient à ce qu'ils sont universels, présents d'office dans les systèmes centralisés[9] et perdus d'office dans un système réparti asynchrone.

Le respect de la causalité, comme l'établissement d'un consensus, imposent des contraintes sur l'ordre dans lequel les processus peuvent exécuter leurs opérations. Dès lors que la correction d'une exécution repose sur le respect de ces principes, maintenir une bonne réactivité devient un problème d'optimisation : quand ces principes doivent-ils s'appliquer ? Sur quelles

[7] De nombreuses spécifications du problème du consensus ont été proposées. Nous prenons ici le terme de consensus dans son sens commun d'agrément entre tous les membres.

[8] Non la relation « happened before », mais bien la causalité *effective* qui peut lier deux opérations.

[9] Une machine de Von Neuman exécute séquentiellement ces opérations ;ses opérations sur la mémoire sont totalement ordonnés. Ces deux propriétés suffisent pour assurer la causalité et permettre la construction de consensus.

opérations et sur quelles histoires doivent s'exercer ces contraintes d'ordonnancement ?

Cette optimisation se heurte à un obstacle inhérent à la nature de ces propriétés : on ne peut déduire quand ces propriétés doivent s'appliquer, à la simple observation des opérations émises par les processus. Une fonction recevant des opérations pour les ordonner et les exécuter, ne peut s'exercer à la fois de façon transparente et être optimale[10]. Implanter de façon optimale un modèle de cohérence nécessite une aide, une spécification explicite de l'application[11] de quand le consensus et la causalité sont nécessaires et sur quels éléments de l'exécution répartie ils s'appliquent.

Une spécification, explicitée par le programmeur, de ces relations, vis à vis de chacune des opérations accédants aux objets partagés, semble peu réaliste. Un *compromis* doit être établi entre le gain apporté par l'optimisation et la complexité qu'elle induit sur le modèle de programmation.

C'est la formulation de ce compromis et les propositions liés à sa mise en œuvre qui a donné naissance à la notion des cohérences dites faibles.

2.6 De la cohérence forte à la cohérence faible

Le terme de cohérences faibles apparu progressivement depuis un quinzaine d'années, désigne un ensemble assez hétérogène et confus de techniques de mises en œuvre (asynchronisme, optimisme), d'utilisation d'heuristiques (référence aux temps dans un univers asynchrone) et de l'intégration de spécifications de type applicatif (commutativité, absence de concurrence, métriques sur les données) dans le critère de correction.

Ériger en critère de cohérence, des stratégies de mise en œuvre ou de simples heuristiques, est à l'origine de nombreux malentendus. Donnons trois exemples :

Indications/contrat de cohérence Les premières tentatives de relâchement de cohérence de données mises en cache se sont basées sur des suggestions concernant une éventuelle durée de la validité de la copie [Ter87] ; ces suggestions se basent sur une estimation du moment où l'original sera mis à jour. La plupart des caches Web actuels se basent encore sur cette notion ainsi que les mises en œuvre de technologies dites « push ». De tels critères ne peuvent être assurés de façon déterministes. Si leur utilité fait peu de doute, peut-être est-il plus sensé de les considérer comme l'expression d'une qualité de service que comme des critères de cohérence. L'extension du Web et son utilisation par des agents non humains entraînent le besoin de propriétés déterministes. L'émergence dans la norme HTTP-1.1 [Din96] de nouvelles propositions en ce sens, ainsi que l'apparition de projets [MrB97] visant à fournir effectivement des garanties aux caches Web en est une conséquence.

[10] Cette assertion est intuitive dans le cadre du consensus, elle l'est peut-être moins de la causalité. Le philosophe anglais David HUME a établit au début du 18ème siècle [Hum86], que la causalité ne peut être établie à partir d'observations. La formulation de cette preuve par Karl POPPER est présentée par Bruno JAROSSON dans [Jar92, chapitre 9]. Cette preuve est illustrée par le paradoxe de l'oie donné par le mathématicien Bertrand RUSSEL à la fin du siècle dernier et repris par Bruno JAROSSON : *Une oie entend tous les jours les pas du fermier qui vient la nourrir. Elle en induit la loi suivante : « Les pas du fermier impliquent l'arrivée de la nourriture. » Malheureusement, un matin, le fermier vient lui tordre le cou. Du danger d'induire une loi générale à partir d'observations particulières.* Jostein GAARDER [Gaa95] commente très simplement cette intuition de la causalité.

[11] De toute fonction générant les opérations en déléguant leur ordonnancement.

Cohérence faible/optimisme Dans la plupart des articles de Bayou [TTP+95, PST+97], la cohérence faible est rattachée à la possibilité qu'ont plusieurs agents d'émettre des écritures de façon asynchrone, donc potentiellement sans se bloquer[12]. Nous pensons que ceci dénote un caractère *optimiste*, non un caractère faible ; Bayou permet à des actions (« updates ») de s'exécuter quitte à les abandonner plus tard ; si tout ensemble d'actions « non sérialisables » [13] était abandonné, où donc serait la cohérence faible ? Le caractère non bloquant ne définit donc pas en soi un modèle de cohérence. L'ambiguïté est d'autant plus forte que Bayou vise effectivement à supporter des modèles de cohérence faible et qu'il possède effectivement des outils pour faciliter leur mise en œuvre : les garanties de sessions[14] [TDP+94] (« session guarantee »). Mais les garanties de session ne définissent pas en soi un modèle de cohérence ; *elles ne suffisent pas* à définir si un ordonnancement d'actions (stabilisées ou non) est correct. Elles permettent à l'utilisateur de spécifier des propriétés d'ordres de *délivrance* (ou de perception) sur les lectures et les écritures. Le critère de cohérence est défini de façon ultime par la procédure de certification (« check procedure ») mise en œuvre par l'utilisateur et que ce dernier associe à toute écriture.

Asynchronisme/observateur global implicite Il est communément admis, même parmi les travaux s'attachant aux cohérences faibles, que, à l'état stationnaire, les réplicats doivent converger vers un état identique. Cette contrainte est admise mais rarement justifiée. Une telle contrainte laisse supposer une indépendance entre les observations effectuées sur un réplicat et l'état de ce réplicat. En fait, l'idée même d'état stationnaire, n'a de sens que dans un référentiel temporel unique. Nous donnons, dans le chapitre suivant, quelques exemples d'applications se satisfaisant fort bien de réplicats ne convergeant pas à « l'état stationnaire ».

Nous pensons qu'une telle contrainte dérive de requis implicites liés à la persistance de l'information partagée. La non convergence de l'information répliquée implique de gérer la persistance de chaque réplicat. De plus, chaque réplicat ne reflétant pas, a priori, l'ensemble des opérations effectuées par les membres, il est nécessaire d'associer, à chaque image persistante de ce réplicat, l'historique des opérations que les processus défunts ont effectués ; à tout le moins l'historique suffisant pour assurer l'initialisation correcte des processus désirant accéder ultérieurement à l'information. S'assurer d'une image persistante unique simplifie clairement la solution. L'image persistante étant unique et, par définition correcte, l'initialisation des processus désirant accéder ultérieurement à la donnée est triviale. Par contre, le coût de cette contrainte « de bon sens » ne nous semble pas souvent évalué.

Nous pensons que la confusion à l'origine de ces malentendus est la conséquence de trois changements : 1) la perte d'une référence simple sur la correction d'un calcul réparti, i.e. de l'équivalence relativement à l'exécution sur un système centralisé, 2) la généralisation des canaux cachés et 3) la disparition du temps local comme critère d'ordonnancement.

Les deux premiers points dérivent d'une évolution « naturelle » de l'informatique à représenter des systèmes d'information humains de plus en plus complexes. Le troisième point dérive de la technologie de construction des systèmes répartis, mais aussi de caractéristiques

[12] Bayou est un système conçu pour supporter des applications coopératives (e-mail, agenda, ...) en mode faiblement connecté voire déconnecté. Il est étudié dans la section 4.2

[13] Bayou n'implante pas de transaction, mais permet de stabiliser chaque action.

[14] Bayou apporte beaucoup d'autres outils d'aide ; en particulier, la possibilité de connaître quand une écriture est stable et la prise en compte de cette stabilité dans le protocole de propagation de ces écritures.

fondamentales de notre univers physique. Le premier point a déjà été introduit au cours de ce chapitre et sera abondamment étudié dans les chapitres qui suivent. Nous nous intéressons dans ce chapitre aux deux derniers points.

2.6.1 Des canaux cachés et de la cohérence

Dans tout système complexe, mais particulièrement dans les environnements « à grande échelle », l'existence de *canaux cachés* pose un sérieux problème à la gestion de cohérence.

Brièvement, un canal caché est une abstraction de communication qui permet à plusieurs entités du système réparti d'échanger de l'information, et ce à l'insu du système. Pour l'application comme pour l'utilisateur, cette information peut avoir un rôle essentiel sur la cohérence de l'information partagée.

Ce problème est identifié de longue date, puisqu'il est déjà analysé dans la dernière partie de l'article fondateur de LAMPORT [Lam78], sous le terme de « comportement anormal » (« anomalous behavior »). La démarche développée par l'auteur est, par contre, loin d'être satisfaisante puisqu'elle résout le problème en construisant un système d'horloge physique synchronisée. L'utilisation d'un tel système d'horloge pour garantir la causalité d'une exécution, par exemple, limite la vitesse de progression de l'application répartie à celle du diamètre temporel du réseau. Cette solution est, dans son principe, conservative : plutôt que d'essayer de découvrir l'information véhiculée dans les canaux cachés, des contraintes sont imposées sur l'évolution du système pour qu'il demeure correct, et ce pour toute caractéristique d'un canal caché plausible dans un univers newtonien (temps absolu).

L'autre démarche, énoncée par LAMPORT dans ce même article, mais non développée, est typique de celle adoptée par les travaux ultérieurs sur les cohérences faibles : réclamer de l'application cette information. Le résultat est meilleur en terme de vitesse de progression, mais au prix d'une complexité accrue. Cette complexité est la source d'un grand nombre de travaux dont certains présentent une confusion commune : ces travaux présentent des heuristiques comme des garanties de cohérence.

L'utilisation d'heuristiques n'est certainement pas en soi une mauvaise chose. Les applications peuvent *sciemment* créer des canaux cachés, pour des raisons de performances : il peut être intéressant de remplacer un protocole d'invalidation de cache par une heuristique sur la durée de validité de la donnée cachée, si les applications utilisant les données de ce cache restent correctes, et ce malgré l'obsolescence potentielle des données. Le problème apporté par la confusion de l'approche heuristique réside dans la correction des systèmes qui les utilisent, de même que dans la possibilité de les changer lorsque l'évolution du système fait qu'elles ne sont plus pertinentes.

2.6.2 Du temps et de la cohérence

Le rôle du temps et ses rapports avec la cohérence ont été étudiés intensivement depuis LAMPORT [Lam78]. De nombreux travaux [Mat89, Mat93, RM93, BM93b, RS95b] ont étudié les propriétés (essentiellement la causalité) portées par le temps dans un système synchrone et ont élaboré des constructions telles que les horloges scalaires, vectorielles ou matricielles (cf. l'excellente synthèse de RAYNAL [RS95b]) permettant d'implanter ces propriétés dans des systèmes asynchrones. Ces travaux semblent être peu considérés dans les domaines où la mise en œuvre et l'utilisation du système sont considérées comme des éléments de validation majeurs, en particulier les systèmes coopératifs à grande échelle (caches Web, édition coopé-

rative, mondes virtuels, jeux et simulations). C'est pourtant dans ces domaines que le sens du temps a une importance majeure, car c'est dans ces environnements qu'apparaissent des indéterminismes difficilement contournables (latence, bande passante) et des requis en terme de réactivité qui imposent de composer avec les contraintes de correction. Connaître celles-ci, c'est pouvoir juger en connaissance de cause de ce que l'on sacrifie et de ce que l'on gagne.

Nous voyons trois utilisations différentes du terme « temps » que nous considérons comme la cause essentielle des ambiguïtés attribuées aux cohérences dites faibles.

- le temps comme identificateur de contrainte d'ordonnancement. Dans ce rôle, il apparaît essentiellement sous deux formes : comme estampille et comme donnée intégrée à l'information partagée. Dans ses deux formes, le caractère monotone croissant du temps est une propriété essentielle.
- le temps comme contrainte de réactivité. Il s'agit ici de contraintes visant à borner la durée d'exécution de certaines tâches locales comme réparties. Bien que la réactivité de l'application soit pour une part essentielle déterminée par les contraintes de cohérence, les contraintes de réactivité ne sont pas en soit des contraintes de cohérence. Tout au plus, ces contraintes limitent les modèles de cohérence envisageables dans un cadre d'exécution donné et imposent certains modèles d'exécution (asynchrone et optimiste, par exemple).
- le temps comme heuristique de cohérence (« time-bounded consistency »). Il s'agit d'utiliser l'approximation du temps réel local, fourni par la plupart des systèmes actuels, comme support de décision pour le contrôle de l'ordonnancement des opérations.

Le première usage recouvre les utilisations où le temps peut être remplacé par des horloges logiques. Il recouvre aussi les utilisations où le temps est intégré à l'état de l'application, par exemple dans les modèles de transactions « t-bound » et « t-vintage » proposé par GARCIA-MOLINÀ et WIEDERHOLD dans [GMW82], et identifie de façon *déterministe* un état ou un ensemble d'états. La prise en compte de la valeur du temps peut aussi être utilisée dans les systèmes répartis synchrones où des bornes sur le délai d'acheminement (latence) des messages peuvent être définies. De tels systèmes reposent sur des synchronisations « raisonnables » des horloges physiques des machines réparties, telles que celle réalisée par le protocole NTP (« Network Time Protocol ») [Mil92]. LAMPORT dans [Lam78] montrent déjà comment exploiter la connaissance de bornes sur les délais de transmission pour éviter les entorses à la causalité provenant de canaux « cachés » de transmission de l'information, c'est-à-dire utilisant des voies non maîtrisées par le système contrôlant l'ordonnancement des événements. Ce principe a été repris dans plusieurs travaux [Lis91, NT93].

Le deuxième usage du temps, comme contrainte de réactivité, ne peut être *garanti*. Une telle contrainte ne peut être assurée, de façon déterministe, que dans un système dit « temps réel » où le délai d'exécution de toute opération, locale comme globale, possède une borne supérieure connue. Une telle contrainte n'est pourtant pas rare dans les systèmes *non* temps réel. Il s'agit dès lors, non plus d'un critère de correction, mais d'un critère représentatif d'une qualité de service, pouvant représenter, par exemple, un compromis cohérence/réactivité particulier.

La troisième utilisation apparaît lorsque certaines propriétés sur le système ou l'application, telles que la période des mises à jour et des accès utilisateur, sont connues de façon approximatives et sont responsables des divergences de l'information répliquée. Le temps est alors une grandeur moyenne qui, en soi, ne définit pas de borne sur la divergence de l'état. Le champ `time-to-live`, attaché aux documents dans le protocole HTTP, et son utilisation dans la durée de validité de ces documents dans les caches Web en est un exemple typique. Les critères de la forme « ... prise en compte des mises à jour n'excédant pas 10 secondes » sont des

formulations courantes, par exemple dans les jeux ou les mondes virtuels. De tels contraintes ne définissent pas un contrat de cohérence. Il s'agit d'heuristiques basées, en général, sur des connaissances que n'ont pas le système ou l'application. Si ces heuristiques ne définissent pas des garanties, elles peuvent, par contre, être d'une grande utilité pour optimiser l'implantation d'un modèle de cohérence.

2.7 Des stratégies optimistes

Nombre de travaux axés sur la coopération à large échelle se sont focalisés, non pas sur la détermination de critères de correction mieux adaptés, mais sur des techniques de mise en œuvre permettant une progression asynchrone des membres coopérants. Le contrôle de concurrence/cohérence optimiste représente une de ces techniques de mise en œuvre. Ces techniques sont très souvent associées aux cohérences faibles [Guy91, GL91, GL93, PST+97]

Sur le principe, pessimisme et optimisme sont des stratégies générales de mise en œuvre de propriétés : une stratégie pessimiste prévient la survenue d'incorrection ; une stratégie optimiste permet la survenue d'incorrection mais assure leur réparation, ceci par des mécanismes dis de compensation ou encore de réconciliation. Ces dernières opérations étant en général coûteuses, les stratégies optimistes ne sont optimisantes que lorsque les incorrections sont rares.

Dans le cadre du contrôle de concurrence, l'optimisme est, a priori, adéquat dans deux occasions :

1. Lorsque le mécanisme de réconciliation s'effectue à un niveau plus fin que le mécanisme de contrôle de concurrence : le mécanisme de réconciliation est alors capable de reconnaître l'innocuité d'accès concurrents que n'a pas pu statuer le mécanisme de contrôle de concurrence (cas d'objets dans une page de mémoire partagée, dans une page de cache d'objets, par exemple).
2. Lorsqu'une donnée fait l'objet de peu d'accès concurrents conflictuels.

Les ordonnanceurs optimistes (appelé aussi certifieurs) sont, en général, moins performants (en terme d'histoires acceptées) que leur pendants pessimistes [MW84, MW85] et [BHG87, chapitre 4]. Ils sont donc utilisés essentiellement pour leur potentialité de progression en cas de latence de communication élevée entre les éléments de l'ordonnanceur réparti, en particulier des déconnexions.

Dans le cadre des applications coopératives envisagées, l'optimisme impose deux remarques : (1) il impose un modèle d'exécution permettant que l'on puisse *défaire* (« roll-back ») des actions, i.e. revenir dans le passé. (2) L'hypothèse que la donnée partagée fait l'objet d'un petit nombre d'accès concurrents conflictuels semble apparemment contradictoire avec l'idée de coopération.

2.8 Vers une définition de la cohérence faible

Nous tentons de fonder une définition rigoureuse de la cohérence faible, en se basant sur les trois points suivants :

1. La cohérence est un critère de correction applicatif. Une application est correcte ou ne l'est pas. Ses données sont cohérentes ou non. La notion de faiblesse n'a a priori pas de sens, concernant la cohérence de l'application ou de ses données .

2. La cohérence est idéalement un critère déterministe. Construire un critère de correction en termes de propriétés non déterministes n'a a priori pas de sens.
3. La cohérence est un critère permettant d'attribuer certaines qualités aux histoires *acceptées* ou encore *committées* ou *certifiées*. L'étude des histoires non pérennes, pouvant être abandonnées ou annulées, ne permet pas de juger de façon pertinente du modèle de cohérence offert par le système.

Il est clair que certaines applications, tel que les jeux, peuvent se passer de critère de correction déterministe. Ces applications peuvent vouloir spécifier une certaine qualité de service. Dès lors que ces qualités de service ne peuvent être assurées de façon déterministe, il s'agit, tout au plus, d'heuristiques données au système pour implanter la gestion de cohérence de cette application. Nous pensons préférable de séparer clairement ces heuristiques de la notion de cohérence.

Nous avons affirmé au début de ce chapitre, notre absence de considération des pannes. Il est clair que cette absence de prise en compte des pannes est en soi une source d'indéterminisme : une panne n'assurera plus la correction de l'application ! Pour satisfaire cette hypothèse, nous considérons une vision de la cohérence, où le seul indéterminisme permis réside dans l'occurence de partitions persistantes du système de communication, d'arrêt (« fail-stop ») de processus ou de la présence de comportement byzantin. Dans les systèmes actuels, ces fautes sont relativement rares. Elles peuvent être diagnostiquées avec un niveau de confiance suffisant, pour les applications que nous ciblons. En particulier, une suspicion erronée de faute, se traduisant par un arrêt de l'application suivie de sa réexécution éventuelle est tolérable.

On constate, et les deux chapitres suivants le montre, que lorsqu'on parle de cohérence faible, on parle de propriétés associées à la fonction de gestion de cohérence, et non de critères applicatifs. Ce point est important pour comprendre l'état de l'art. Une fois ce fait admis, il devient naturel de s'interroger sur les qualités requises par la fonction de gestion de cohérence. Une fonction de gestion de cohérence sera optimale, relativement à une application, si elle n'induit pas plus de contraintes sur l'ordre réparti des opérations qu'il n'est nécessaire pour assurer la correction de cette application.

Supposons que la fonction de gestion de cohérence soit assurée par des composants systèmes. Ces composants vont implanter des propriétés plus ou moins conservatives sur l'ordre des opérations, qui suffiront ou non à assurer la correction de certaines applications. Sur un composant système donné, certaines applications seront correctes, d'autres non.

C'est en se basant sur cette constatation que nous proposons la définition suivante :

Définition 2.1 (Cohérence faible) *Une fonction répartie de gestion de cohérence implante un modèle de cohérence forte si elle assure la correction de toute application qui le serait dans un modèle à cohérence séquentielle [Lam79]. Elle implante un modèle de cohérence faible dans le cas contraire.*

Ainsi, selon notre définition, la cohérence causale est une cohérence faible : certains outils algorithmiques de base, tel le consensus, même en l'absence de panne, ne peuvent être construits au dessus d'une cohérence causale, *ex nihilo*, sans agrément statique préalable entre les processus.[15] En particulier, la simple atomicité locale d'une séquence lecture-écriture (`test-and-set`) sur une mémoire causale [AHJ91], ne peut être utilisée pour mettre en œuvre un sémaphore[16].

[15] Tel qu'un processus dédié (séquenceur), une priorité sur des adresses de sites ou des identificateurs de processus, par exemple.

[16] Une bonne introduction à ce problème sera trouvée dans [RS95a].

Considérons la gestion de cohérence associée à une application utilisant Bayou, par exemple, pour gérer une collection de référence bibliographique.[17] Supposons que la fonction de détection de conflit (« check procedure ») statue un conflit dès que deux écritures sur la base (insertions concurrentes de deux entrées bibliographiques) sont concurrentes. Supposons que la procédure de réconciliation (« merge procedure ») avorte systématiquement (annulation des deux insertions) les opérations conflictuelles. Compte tenu des propriétés de causalité, qu'assure par ailleurs Bayou, une telle gestion de cohérence est séquentielle. Nous ne considérons pas une telle gestion de cohérence comme faible, bien qu'elle permette aux applications de progresser, dans une certaine mesure, de façon asynchrone. Bien qu'une telle mise en œuvre ne soit pas dans l'esprit de Bayou, elle est permise. De façon plus générale, des transactions optimistes avortant toute transaction non sérialisable ne peuvent être considérées comme assurant un cohérence faible, bien qu'elle permette *apparemment* la progression « non cohérente » de l'application. On ne peut statuer de la progression effective de l'application que sur des transactions *committées*.

2.9 Résumé

Nous avons précisé dans ce chapitre le rôle que nous attribuons à la gestion de cohérence.

Nous avons introduit les outils traditionnels développés par la communauté scientifique pour juger des propriétés de cohérence des applications. Nous avons introduit le problème posé par la latence et la coopération. Nous avons dressé un rapide constat des nombreuses propositions adressant ce problème.

Une analyse critique raisonnable de ces propositions nous a permis de distinguer les propositions ayant trait à la correction de l'application de ceux ayant trait à la mise en œuvre de ces principes. Cette analyse nous a permis d'énoncer une formulation « utile » de la cohérence faible. Cette formulation écarte tout indéterminisme du critère de cohérence justifiant son statut comme critère de correction. Elle associe la notion de faiblesse à la fonction de gestion de cohérence et non à l'application. Elle identifie la faiblesse de cette fonction à une optimisation relativement aux deux principes fondateur de la correction en réparti : la causalité et le consensus. Ce problème d'optimisation est posé en terme de compromis entre la complexité nécessaire pour connaître quand ces principes doivent être assurés et la réactivité de l'application.

[17] Cet exemple est cité dans [TDP+94, TTP+95].

Chapitre 3

Des compromis réactivité/cohérence

Notre propos dans ce chapitre est d'identifier les principes qui permettent de maîtriser la vitesse de progression d'un système, ce que nous désignons par sa réactivité. Nous nous proposons d'évaluer la pertinence de ces principes aux systèmes coopérants et d'évaluer leur complexité de mise en œuvre.

La maîtrise de la réactivité d'un système passe par la minimisation de la latence des opérations. Dans un système réparti répliqué concurrent, cette maîtrise conduit essentiellement à minimiser la latence induite par l'*ordonnancement* des opérations.

Maîtriser la latence des opérations n'est ni un problème nouveau ni un problème spécifique à la réplication. Quatre domaines de l'informatique s'y sont intéressés de façon intensive :

- Les bases de données (BD).
- Les systèmes à objets répartis (SOR).
- Les systèmes de communication de groupe (SCG ou « CATOCS »).
- Les mémoires partagées réparties (MPR ou « DSM »).

Un des buts de ce chapitre est de dégager, parmi les solutions adoptées dans ces différents domaines, un petit nombre de principes applicables à notre problématique. Les modèles de programmation utilisés (processus, transactions) dans ces domaines, leurs types d'application (transaction bancaire, ingénierie coopérative, calcul parallèle, contrôle de processus industriel) ainsi que leurs niveaux d'abstraction varient considérablement. Par exemple, la communication de groupe comme les mémoires partagées sont des abstractions de bas niveau (communication), qui sont donc, par nature, limités dans leur exploitation de la sémantique de l'information partagée.

> Nous insistons sur le fait que ce chapitre ne saurait ni ne veut constituer un état de l'art de ces quatre domaines. Un tel travail nécessiterait une encyclopédie, et s'écarterait notablement du problème adressé dans cette thèse.
>
> Par contre, la gestion de la cohérence est une fonction que tous ces domaines ont eu à mettre en œuvre. Et pour la plupart, ces domaines ont été confrontés à des problèmes de performances plus ou moins proches des nôtres. Dans ce chapitre, nous nous contentons de survoler ces domaines, en nous préoccupant uniquement des aspects liés aux optimisations sur les propriétés de cohérence qu'ils ont pu mettre en pratique. Le but ultime étant, bien sûr, de pouvoir juger de la réutilisation de ces optimisations dans le domaine de la coopération à distance.

Les deux premiers domaines (BD, SOR) ont en commun l'utilisation d'un modèle transactionnel et l'intérêt porté au contrôle de concurrence. Les deux derniers domaines (SCG, MPR) se basent sur un modèle à processus et s'intéressent, essentiellement, à la synchronisation répartie.

La latence dans les systèmes transactionnels est essentiellement perçue comme induite par les contraintes de concurrence. La latence dans les systèmes à processus répartis est essentiellement perçue comme induite par les coûts de synchronisation au dessus du réseau.

SKARRA et ZDONIK [SZ89] présentent ces deux lignes d'approche de la gestion de cohérence : l'approche orientée sur la cohérence des données et l'approche centrée sur la cohérence des transactions[1] accédant à ces données.

La première approche concerne les modèles à processus (SCG, MPR). L'absence de toute formalisation de l'entité d'exécution accédant aux données fait que la cohérence ne peut se gérer que sur le plan du « médium de communication » entre les processus,c'est-à-dire la représentation des données (mémoire, segments, objets). Cette approche s'est focalisée essentiellement sur l'interaction entre les schémas de synchronisation entre processus et le protocole de réplication. Dans ce contexte, le critère « fort » de référence est la cohérence mutuelle des réplicats.

La deuxième approche concerne les modèles transactionnels ; elle se focalise sur la maximalisation des ordonnancements corrects en vue de maximaliser la concurrence des transactions[2]. Cette approche déjà perceptible dans la notion des degrés d'isolation [GR93], s'est raffinée énormément avec l'apparition des systèmes répartis à objets [Lis88, DHW88, PSWL95] et des bases de données objets. Dans ce contexte, le critère « fort » de référence est la sérialisibilité, voire la linéarisibilité dans le cas des systèmes à objets atomiques.

Outre leur propriété de transparence, un trait commun aux modèles de cohérence forte est l'indépendance entre les contraintes de corrections sur l'état (le protocole de contrôle de réplication dans le cas des MPR, les contraintes d'intégrité de la base de données dans les mémoires) et les contraintes de correction sur les actions. Cette indépendance révèle en fait le synchronisme responsable du manque de réactivité commun à ces modèles ; dissocier le protocole de réplication du protocole de contrôle de concurrence revient à dire soit que le contrôle de concurrence peut négliger la perception des accès concurrents distants, ce qui parait peu crédible, soit que le contrôle de concurrence a une perception « instantanée » des accès distants, ce qui revient à la condition synchrone induite par l'atomicité des accès.

Les modèles répliqués à cohérence faible [Mos93] comme les modèles transactionnels relâchant la sérialisibilité [RC94, RP95] adoptent en fait une démarche commune : fournir à la gestion de cohérence, des informations plus riches sur les opérations lui permettant de définir, au plus juste, quand les opérations concurrentes doivent être perçues (ou prises en compte) sous peine d'incorrection de l'application. Nous analysons le problème en deux étapes : 1) raffiner la connaissance de la réalité de la concurrence de deux accès, c'est-à-dire découvrir et exploiter la topographie réelle des accès et 2) raffiner la connaissance des ordonnancements

[1] SKARRA et ZDONIK entendent aussi bien les actions atomiques introduites dans Argus [Lis88] et présentes dans la plupart des systèmes répartis à objets [DAM+91, DHW88, PSWL95, LDS92] que les transactions dans les bases de données. La réplication n'est pas abordée.

[2] Vis-à-vis de la réplication, les originalités concernent essentiellement des optimisations ou des extensions de mise en œuvre des protocoles à quorum [Gif79][BHG87, chapitre 8]. Le principe de base reste celui de la cohérence mutuelle des réplicats ; les optimisations et extensions concernent essentiellement le support de type de panne accru (les conditions de progression en fonction des partitions, par exemple).

corrects.

Cette décomposition peut sembler arbitraire dans la mesure où ces problèmes sont souvent confondus ; des outils de contrôle de concurrence tels que les verrous à prédicats [GR93], capturent ces deux connaissances. Tous les travaux sur l'exploitation de la sémantique des opérations associées à des types abstraits présentent, de façon uniforme, l'exploitation de ces deux types de connaissances. Nous verrons plus tard, dans l'exposé de notre solution, que l'exacte nature de l'optimisation est importante pour comprendre l'impact de celle-ci, dès lors que la latence de perception des opérations est importante.

3.1 Exploiter la topographie des accès

Statuer de la concurrence de deux opérations pose un problème dans deux cas : 1) lorsque le contrôle d'accès se fait à une granularité plus grosse que l'opération et 2) lorsque la connaissance de la topographie n'est pas simplement accessible (cas des types de données abstraits).

1. Le contrôle de concurrence ne représente qu'un aspect du traitement de l'information. La représentation de l'information en mémoire et sa persistance en particulier, nécessitent[3] le regroupement de plusieurs données distinctes dans des structures systèmes (pages, « clusters », fichiers, bases). Le contrôle d'accès se fait au niveau de l'accès à ces structures et c'est alors à la fonction de contrôle de concurrence qu'incombe la tâche de distinguer les « faux partages » résultant de données différentes regroupées dans une même structure. Cette différence de granularité entre le contrôle de concurrence et les structures de représentation des données nécessite, dans le cas où les données sont répliquées (cas fréquent en réparti (caches)), des phases de fusion (« merging ») lorsque des données différentes mais regroupées sur une même structure systèmes sont accédées concurremment sur des sites différents. Les systèmes utilisant des caches de pages et du contrôle de concurrence ayant pour grain un objet [CFZ94, CALM97] doivent mettre en œuvre une fonctionnalité identique. L'absence de prise en compte du faux partage se traduit dans le cas des mémoires répliquées par des effets de « ping-pong » désastreux [CBZ91] sur les performances du système.
2. Lorsque les données concurrentes sont membres d'objets complexes (listes, queues, répertoires ...), la connaissance exacte de la donnée (item de la liste, entrée particulière d'un répertoire) fait partie des arguments d'invocation d'une méthode, et peut même dépendre de la sémantique de l'objet complexe. SCHWARZ et SPECTOR [SS84] ont proposé d'utiliser la sémantique des types abstraits pour statuer de cette concurrence ; l'exemple développé concerne l'insertion concurrente de deux noms différents dans un répertoire de noms[4]. Cet exemple est repris dans nombre de travaux sur les systèmes à objets [Wei84, GSW86, Ng89, DHW88, PS88].

Paradoxalement, statuer de la façon la plus fine possible sur la concurrence de deux accès n'est pas toujours souhaitable. Par exemple, dans les moniteurs transactionnels basés sur le verrouillage à deux phases, le temps passé à prendre des verrous devient prépondérant. Moins une transaction prend de verrous, plus elle s'exécute rapidement, moins elle a de chance d'entrer en concurrence avec une autre transaction et plus elle s'exécute rapidement ! Ce cercle vertueux a comme conséquence de diminuer le risque d'apparition de verrou mortel

[3] Contrainte de pagination, de cache ou plus simplement pour des raisons d'occupation mémoire.

[4] Ce papier fondateur propose d'autres optimisations qui seront étudiées par la suite.

[GR93] et donc d'améliorer la vivacité et la performance du système. Un compromis se dessine donc entre la prise rapide d'un petit nombre de verrous à gros grain, et la prise d'un grand nombre de verrous à grain fin, représentant au plus juste l'information utilisée. La plupart [BHG87, GR93, BN97] des moniteurs transactionnels utilisent un verrouillage à deux phases. Ces moniteurs implantent ce compromis en utilisant des verrous dits granulaires (« granular locks ») : la représentation des données est structurée en un arbre dont la racine représente la base de données tout entière et les feuilles les données individuelles[5]. La correction des verrous granulaires nécessite une extension du typage des accès : les accès à grain fin doivent marquer les structures à grain plus gros contenant les données utilisées de leur intention d'accéder à ces données. La plupart des systèmes de gestion de base de données [GR93] font un ajustement dynamique de la granularité de verrouillage de façon opportuniste au moyen d'une technique de promotion de verrou appelée « lock escalation ».

3.2 Exploiter la connaissance des ordonnancements corrects

Dès lors que l'on a déterminé quelles opérations concernent une même donnée, on est amené à étudier les contraintes sur l'ordre de ces opérations nécessaires à la correction du système.

Nous distinguons deux types de contraintes :

1. Celles qui se basent sur l'ordre même d'exécution des opérations. C'est, de loin, la catégorie la plus étudiée.
2. Celle qui se basent sur une fonction de cet ordre. Ce type de contrainte se base typiquement sur une métrique permettant de quantifier une imprécision sur l'information partagée, ou une divergence entre plusieurs représentations de cette information, dans le cas de la réplication. Ce type de contrainte pose de nombreux problèmes, le plus aigu étant d'arriver à maîtriser la correction à long terme du système. La formalisation elle-même de l'imprécision est délicate.

Notre étude se fait en traversant, dans la mesure du possible, les différents domaines de l'informatique répartie évoqués en début de chapitre.

3.2.1 Correction basée sur l'ordre d'exécution des opérations

La plupart des systèmes existants contrôlent la correction de l'ordonnancement des opérations en se basant sur des règles de conflits entre ces opérations : si la correction de l'application dépend de l'ordre dans lequel deux opérations sont exécutées, ces opérations sont considérées comme conflictuelles. La notion de conflit capture à la fois les propriétés topographiques des accès (l'ordre de deux écritures consécutives sur deux données distinctes, n'est, a priori, pas important) et les propriétés opératoires (le fait qu'une lecture ne modifie pas la donnée lue, par exemple). La notion de conflit n'est qu'un outil pour s'assurer que certaines dépendances entre opérations sont respectées. La légalité d'une lecture, par exemple, se définit comme la dépendance entre la valeur retournée par la lecture d'une donnée et l'écriture de cette valeur sur cette donnée : maintenir cette dépendance signifie s'assurer que l'écriture a précédé la lecture et qu'elle est la dernière écriture ayant précédé la lecture sur cette donnée. La causalité définit une dépendance entre une observation (une lecture) et une action (écriture). La sérialisibilité

[5] Des extensions non arborescentes on été proposées pour permettre son utilisation dans le verrouillage sur des intervalles de clés (« key range ») [GR93].

[BHG87] se définit par le respect des dépendances entre opérations conflictuelles (équivalence de conflits) ou entre paires d'écriture-lecture (« read-from ») et un ensemble d'écritures finales (équivalence de vues).

En fait, tous les modèles de cohérence (forts comme faibles) sont caractérisés par le respect de relations de dépendances entre des opérations[6].

Nous distinguons deux types de dépendances :

- Les dépendances « non contextuelle ». Ces dépendances sont induites par la nature même d'une opération, son type (incrément ou décrémentation) ou son appartenance à une catégorie (lecture, accès synchronisant, ...). La nature de ces dépendances peut être déterminée indépendamment de tout contexte.
- Les dépendances « contextuelle ». Ces dépendances sont induites par l'ordre dans lequel un ensemble d'opérations a été initié. La causalité induit des dépendances de ce type. Ces dépendances n'ont de sens que dans le contexte d'une exécution particulière.

3.2.1.1 Dépendances « non contextuelles »

Bases de données. L'utilisation de la commutativité et de la non mutabilité des lectures est une propriété utilisée de longue date. La sérialisibilité par exemple, n'attache pas de dépendances entre deux opérations de lecture. La classification proposée par GRAY [GR93, chapitre 7] distingue quatre degrés d'isolation :

Degré 0 : Respect d'aucune dépendance.

Degré 1 : Respect des dépendances `écriture-écriture` (pas de perte de mises à jour).

Degré 2 : Degré 1 + respect des dépendances `écriture-lecture` (pas de lectures « sales », i.e. non validée)

Degré 3 (isolation) : Degré 2 + respect des dépendances `lecture-écriture` (lectures répétables et gestion des fantômes).

Les divers degrés d'isolation contraignent de façon croissante les ordonnancements entre opérations appartenant à des transactions distinctes, ceci en imposant l'absence de cycle vis-à-vis des dépendances associées au degré d'isolation. La plupart des BD relationnelles assurent par défaut des degrés d'isolation de type 2 [GR93, section 7.6.2], c'est-à-dire ne considérant plus les dépendances `lecture-écriture` (les lectures ne sont pas répétables). Les transactions en lecture uniquement s'exécutent souvent au degré d'isolation 1.

L'extension des règles de conflits sur des opérations d'incrémentation et de décrémentation sur des scalaires est présentée dans [BHG87, section 2.5]. Les seules dépendances représentées dans la table de compatibilité des verrous sont celles concernant les opérations de lecture et d'écriture. L'exploitation de propriétés liées aux types des données est apparue avec les bases de données orientées objets qui manipulent explicitement des types de données abstraits [SZ89, RAeA94]. Cette exploitation peut, en particulier, être couplée efficacement avec un contrôle de concurrence multi-niveau [BR90].

[6] La sérialisibilité comme les contraintes de délivrance dans les SCG sont explicitement définis sous forme de dépendances. Dans une mémoire atomique, toute opération *dépend* potentiellement de l'opération qui la précède, d'où l'ordre total en découlant.

Les systèmes à objets atomiques. WEIHL [Wei84] s'intéresse essentiellement à la construction modulaire de transactions atomiques sur des objets (instances de types de données abstraits)[7]. Dans ce cadre, WEIHL présente de nombreuses utilisations de la commutativité[8] sur des comptes bancaires. De telles optimisations ont été mises en œuvre dans le cadre de l'édition collaborative [GSW86] au dessus d'Argus [Lis88]. La commutativité a bien sûr été reprise dans d'autres systèmes à objets [PS88, PSWL95].

Les systèmes de communication de groupe. L'essence des systèmes de communication de groupe est l'ordonnancement des multi-envois (« multicast »)[9] de messages ; ces ordonnancements définissent des dépendances sur l'ordre de délivrance de multi-envois distincts à des processus distincts :

Diffusion atomique : des diffusions sont dites atomiques (définition adoptée par Isis [Bir85]) si 1) elles sont délivrées dans le même ordre à chaque processus membre du groupe (d'une conversation dans le cas de Psync [PBS89]) et si 2) les délivrances de diffusions en provenance d'un même processus suit l'ordre de leur émission (contrainte FIFO). Dans le cas de groupes disjoints, ces deux contraintes assurent le maintien de la causalité.

Diffusion causale : des diffusions sont dites causales si l'ensemble des diffusions délivrées à un processus précédant l'émission d'une diffusion par ce même processus sont délivrés avant cette dernière diffusion sur tous les processus.

Isis [Bir85] est le premier d'une famille de support système supportant le modèle de synchronie virtuelle [BJ87, Bir96]. La thèse fondatrice d'Isis est que la cohérence d'un système réparti repose sur la connaissance et la maîtrise de l'ordre dans lequel les événements concernant l'ensemble du groupe sont perçus : ces événements vont de la communication d'informations « normales » à celles liés à la composition du groupe (pannes, arrivée ou départ d'un membre). L'idée d'Isis est de caractériser ces événements comme des messages et de caractériser les contraintes reposant sur la perception de ces événements comme des contraintes sur l'ordre dans lequel sont délivrés ces messages. Isis repose sur trois primitives de diffusion (diffusion causale (CBCAST), diffusion atomique (ABCAST) et diffusion globalement ordonnée (GBCAST)) pour assurer ce modèle de synchronie virtuelle. D'autres systèmes dits CATOCS (« Causally and Totally Ordered Communication System ») ont ensuite été construits sur ce principe [Sec96].

Les mémoires partagées réparties. La réponse au problème posé par les performances des MPR de première génération [LH89] a été de prendre en considération le fait que la plupart des programmes préviennent (par des sémaphores ou des moniteurs) l'accès concurrent par plusieurs processus à une même donnée. Lorsqu'une série d'écritures à une donnée se fait en exclusion mutuelle, ces écritures n'ont pas besoin d'être totalement ordonnées vis-à-vis d'autres accès[10]. Clouds [RAK89] est un des premiers systèmes à coupler les transferts de mises à jour

[7] Les différentes formes d'atomicités proposées dans [Wei89] sont le point de départ de ce qui donnera la linéarisibilité [HW90], une propriété permettant d'assurer de façon modulaire, par objets, la correction (linéarisibilité) de transaction accédant à un graphe d'objets atomiques.

[8] Weihl présente [Wei88] plusieurs formes de la commutativité (« forward commutativity » et « backward commutativity »). Ces formes imposent des requis différents sur la gestion de recouvrement (undo log ou liste d'intention).

[9] On parle aussi souvent de diffusions. La distinction est nécessaire dès lors que l'on permet l'existence de plusieurs groupes pouvant partager des membres, comme dans Isis [BJS87].

[10] étant en exclusion mutuelle, les écritures ne peuvent être observées.

avec les relâchements des verrous pris pour assurer l'exclusion mutuelle. La même approche a été adoptée pour la gestion de mémoire des multiprocesseurs. Au chapitre précédent nous avons présenté la « weak consistency » [DSB86, DSB88]. « Release-consistency » [GLL+90] part du même principe mais subdivise les accès synchronisants en deux classes supplémentaires, « acquire » et « release » ; cette catégorisation influe sur la mise en œuvre des opérations et non sur l'ordre des opérations. Munin [CBZ91] est une MPR (basée sur la « release-consistency ») qui exploite les sémantiques de partage au niveau applicatif. Les schémas d'accès aux données partagées sont pris en compte pour déterminer des contraintes d'ordre minimales mais assurant néanmoins la correction de l'application.

HATTIYA et FRIEDMAN dans « hybrid-consistency » [AF92] classifient les accès en « strong » ou en « weak ». Les accès « strong » sont totalement ordonnés alors que les accès « weak » ne sont ordonnées que de façon FIFO entre eux et avec les accès « strong ». Autrement dit, seuls les accès « strong » nécessitent un agrément au dessus du réseau.

« Entry-consistency » [BZ91](EC) se démarque nettement des précédents modèles de cohérence : les modèles de cohérence faible vus jusqu'ici utilisent la classification des accès synchronisants pour reconstruire des sortes de transactions atomiques. EC construit essentiellement des moniteurs au niveau de la mémoire partagée. Il synchronise, comme dans Clouds [RAK89], les mises à jour sur la portion de mémoire contrôlée par le moniteur avec la libération de ce moniteur. Toutefois, EC retarde le transfert des mises à jour à l'acquisition de ce moniteur par un autre processus.

3.2.1.2 Dépendances « contextuelles »

Bases de données. GARCIA-MOLINA et WIEDERHOLD [GMW82] ont proposé très tôt d'exploiter les traits sémantiques des transactions en lecture seules[11]. Les transactions peuvent ne rien spécifier du tout (degré 1 d'isolation encore désigné « browse mode »), spécifier une date permettant d'écarter les effets des transactions ayant été validées plus tard (« t-vintage ») ou de prendre en compte au moins l'ensemble des transactions ayant été validées à cette date (« t-bound »). Ce rapport au temps dans un critère de cohérence, spécialement dans le cadre asynchrone pose quelques problèmes. Le modèle d'ϵ-sérialisibilité [PL90, RP95], présenté au chapitre suivant, synthétise la construction au plus juste de dépendances transactionnelles, en se basant sur des critères dynamiques et sémantiques.

Les systèmes à objets atomiques. TABS [SS84] a introduit la notion de queues faiblement FIFO (encore nommées semi-queues) sur lesquelles des transactions peuvent interagir de façon non isolée. Deux transactions peuvent insérer et retirer mutuellement des éléments insérés par l'autre transaction. Une telle optimisation est formalisée comme la possibilité de considérer comme corrects des ordonnancements présentant des cycles au niveau des insertions et des suppressions d'éléments de la queue[12].

La relation « invalidated by » introduite par HERLIHY et WEIHL [HW91], formalise les relations de dépendances entre opérations. Les auteurs montrent que cette relation est incompa-

[11] Ces transactions ne modifiant pas la base de données, l'incohérence qu'elles peuvent avoir de la base est sans conséquence sur l'intégrité de celle-ci.

[12] SCHWARZ et SPECTOR montrent de même que la sérialisibilité de transactions accédant plusieurs types de données différents n'est pas assurée par le respect individuel, pour chaque type, des « bonnes » dépendances. Ce problème a été posé par WEIHL [Wei89] et HERLIHY [HW90] et une réponse a été apportée par GUERRAOUI [Gue94].

rable avec la commutativité (« failure to commute ») et permet, en général, plus de concurrence que cette dernière[13]. Leur étude reprend tous les exemples classiques (compte bancaire, queue, semi-queue). Ces exemples ont été mis en œuvre au dessus d'Avalon/C++ [DHW88]. Les gains permis reposent, essentiellement, sur la possibilité qu'a la gestion de cohérence de définir le résultat des invocations[14] sur l'objet. Ces travaux, ainsi que ceux postérieurs de WEIHL, sur la « forward commutativity » [Wei88] sont repris par LEONG et AGRAWAL dans un contexte de réplication totale (DSM). Cependant ces auteurs ne prennent pas en considération la latence de la perception des opérations induite par la réplication ; or le calcul dynamique de la relation « invalidated by » nécessite la perception de toutes les opérations.

Les deux systèmes suivants, « Lazy replication » et Bayou, ne sont pas à proprement parler des systèmes à objets atomiques : ils ne sont plus atomiques et la notion d'objet est assez lâche ; par contre, ils ordonnent des opérations *typées*.

« Lazy replication » [LLG92] propose une mis en œuvre originale de services répliqués (une gestion de boites aux lettres réparties) au dessus d'Argus. « Lazy replication » distingue les opérations d'interrogation (« query ») de celles de mises à jour (« update »). Des estampilles sont assignées à chaque mise à jour par les réplicats (serveurs) recevant ces mises à jour. Chaque opération peut spécifier un ensemble d'estampilles identifiant les opérations dont elles dépendent et qui doivent être exécutées avant elles sur les réplicats. « Lazy replication » définit aussi des mises à jour « forcées » (« forced update »), qui sont ordonnées totalement entre elles (sémantique d'ABCAST d'Isis) et des mises à jour immédiates (« immediate update »), qui sont totalement ordonnées vis à vis de toutes les mises à jour (sémantique de GBCAST d'Isis).

Bayou [TTP+95] est une infrastructure répartie pour des applications coopératives (courrier, agenda), conçues pour des agents mobiles. Bayou propose des garanties de session [TDP+94] définissant des contraintes sur l'ordre de perception des opérations. Bayou est un des systèmes flexibles [PST+97] qui sera détaillé dans le chapitre suivant.

Les systèmes de communication de groupe. Psync [PBS89] réifie la précédence des messages sous la forme d'un graphe de contexte. Les utilisateurs de Psync peuvent consulter ce graphe et définir des filtres ou des prédicats permettant de respecter les dépendances de messages lors de leur délivrance. La communication point à point, comme les diffusions causales et atomiques de X-kernel ont été construites au-dessus de Psync.

Les mémoires partagées réparties. La mémoire à cohérence de processeur (encore appelée FIFO) [Mos93] prend en compte uniquement les dépendances d'écriture provenant d'un même processeur. La mémoire à cohérence causale [AHJ91, Mos93] garantit les dépendances de type `écriture-lecture-écriture` caractéristique de la causalité, lorsque seules les lectures sont des observations et que seules les écritures sont des actions.

« Lazy release consistency » [KCZ92](LRC) est une optimisation paresseuse de « release consistency » construite sur le respect de la relation « read-from », ainsi que de l'ensemble des écritures finales sur la mémoire.

[13] Cette augmentation de concurrence est permise parce que la gestion de concurrence maîtrise le résultat des opérations retournées à l'application.

[14] Une invocation n'est pas un événement atomique. Ceci conditionne en fait l'atomicité hybride proposée par ces mêmes auteurs et sur laquelle a été développée ces travaux.

3.2.2 Correction basée sur une fonction de l'ordre d'exécution

Permettre d'observer les informations partagées avec une certaine imprécision et permettre aux processus coopérants de continuer à progresser malgré cette imprécision est une technique très souvent employée dans notre vie de tous les jours et ce pour des raisons identiques à celle des systèmes d'informations : améliorer la réactivité du système. Si, face à certains événements rares importants, telle une transaction bancaire sur un achat de maison ou la détermination de la collision d'avions, nous imposons des contraintes draconiennes sur la précision de l'information, dans de nombreux autres cas, une approximation de la connaissance de l'état l'information partagée suffit à la correction des applications : il en est ainsi de jeux, de simulations distribuées, de système d'équilibrage de charge, de système de réservation ou bien encore, des prises de risque contrôlées dans les systèmes d'aide à la décision.

Par exemple, la plupart des jeux multi-utilisateurs actuels (ACM, XBlast, DOOM, XTank, MiMaze) permettent au processus locaux de progresser sans avoir une connaissance précise, «synchrone», des actions des autres processus, c'est-à-dire sans ordonnancement global des opérations. Les simulations distribuées [JBW+87, Fuj90] ressemblent aux jeux dans leur rapport avec un temps virtuel.[15] On peut donc espérer pouvoir leur appliquer le même type d'optimisation.

Dans un cadre déterministe, la modélisation de l'imprécision repose sur l'existence d'une fonction sur l'histoire des processus concurrents qui permette de quantifier cette imprécision. Dans un contexte répliqué, la divergence entre les états d'un ensemble de réplicats, collectés à certains instants privilégiés, permet de matérialiser cette imprécision.

Nous approfondirons la question, en présentant, au chapitre suivant, le modèle d'ϵ-sérialisibilité [WYP97, PL90, RP95]. Ce modèle fournit un cadre permettant aux transactions de souscrire à des imprécisions sur l'observation d'une base de données, en échange d'une meilleure concurrence. D'autres exemples de contrôle de divergence seront étudiés dans le chapitre 6, une fois développé le formalisme **Core**, un cadre d'étude adéquat.

3.2.3 Quelques remarques

De la commutativité. La commutativité englobe un grand nombre de propriétés : deux accès sur des données différentes commutent (topographie). Deux accès sur une même donnée dont l'ordre n'affecte pas l'état final de cette donnée commutent (agrément) ; cette propriété peut être statique, basée sur le type, ou dynamique, basée sur des arguments particuliers. Deux accès non observants commutent vis à vis de la causalité.

La commutativité est au cœur de la plupart des optimisations, vraisemblablement parce qu'elle agit aussi sur l'équivalence de deux histoires. Elle agit ainsi autant sur le contrôle de divergence associé à la réplication que sur le contrôle de concurrence. Quelques travaux ont utilisé l'exploitation de la commutativité (optimisation sur la constitution des quorums [Gif79, BHG87] d'opérations) ou d'une autre propriété («invalidated by»), simultanément dans ces deux dimensions, dans le cadre des bases de données [KS88] comme dans le cadre des systèmes à objets «atomiques» [Mau90, Her90, LA92].

[15] MATTERN dans [Mat93] introduit remarquablement l'implantation d'un temps virtuel global et plus généralement des moyens d'assurer et de contrôler la croissance globale d'une fonction monotone en se basant sur des coupes non forcément causalement cohérentes.

Du pessimisme ou de l'optimisme de la gestion de cohérence. Pessimisme et optimisme sont des stratégies générales de mise en œuvre de propriétés, non des critères de correction. Bien que ces stratégies jouent un rôle essentiel sur la vivacité de l'application, nous considérons que leur étude est du domaine de la mise en œuvre de la gestion de cohérence, et n'entre pas dans le cadre des propriétés de cohérence. L'étude de ces stratégies sera abordée dans le cadre de l'exposé de la mise en œuvre de la gestion de cohérence.

Des protocoles de réplications. Ils ne sont pas abordés dans ce chapitre. Les protocoles à quorum [Gif79] assurent, de façon inhérente, la convergence mutuelle et donc n'adressent pas, ou que marginalement, notre problème de latence. De plus, dans le cadre du contrôle de divergence, ils sont peu utilisés [CP92, BN97]. Les protocoles de réplication ciblant les systèmes mobiles ou faiblement connectés [GL91, GL93, PST+97] sont, pour la plupart, fondés sur des principes dits anti-entropiques ou proches (propagation de rumeurs « rumor mongering » ...) et visent essentiellement à améliorer l'interactivité, par utilisation de l'asynchronisme (interaction non bloquante). L'amélioration de la vitesse de progression provient du faible taux de conflit, même en présence de critère de cohérence fort.

3.3 Analyse

Quels liens ont les optimisations passées en revue dans les sections précédentes avec les deux principes structurant la correction d'une application répartie, la causalité et le consensus ?

Ces optimisations reflètent des compromis plus ou moins conservatifs réalisés par la gestion de cohérence sur la réalisation de ces principes. Les différentes variantes de ces compromis caractérisent les différents travaux sur la cohérence, effectués dans les différents domaines confrontés à la répartition. Le modèle de cohérence atomique est extrême : il fait dépendre causalement toute opération de ses prédécesseurs et il considère que toute opération peut faire diverger l'état répliqué. Il en résulte un ordonnancement total de toutes les opérations. Aucune optimisation n'est réalisée (au niveau du modèle) en échange d'une complexité induite minimale[16]. Prenant acte de la possibilité de différencier les lectures des écritures sans complexité accrue du modèle de programmation[17], la cohérence de type « ww-constraint » [MRZ94] n'induit pas de causalité entre les écritures (les écritures sont aveugles) et n'attribue pas de potentialité de divergence (anti-consensus) aux lectures (les lectures ne modifient pas l'état). Les implantations usuelles du modèle de synchronie virtuelle [BJ87] optimise la « dimension consensus », en laissant au processus coopérant la possibilité de ne pas communiquer uniquement au travers de diffusions atomiques (« ABCAST »). Mais ces implantations restent conservatives sur la dimension de la causalité, en construisant celle-ci sur la relation « happened before », relation basée sur l'observation de l'ordre dans lesquelles les messages ont été reçus et envoyés par les processus.

Finalement, comment ce lien peut-il être exploité ? Quels gains pouvons nous attendre de ces optimisations vis-à-vis de la latence ? Et que nous coûtent leur mise en œuvre ?

[16] Seule la réplication peut être assurée de façon transparente, au niveau de la gestion de la mémoire, la gestion de la concurrence ne se satisfaisant pas, en général, de l'atomicité au niveau d'opérations individuelles. Ceci dénote la transparence des transactions relativement au contrôle de concurrence tel qu'implanté dans les systèmes ou dans les langages concurrents.

[17] La plupart des applications sont conçues pour le modèle de Von Neuman [vN45, BGN76] et tous les mécanismes « hardware » de gestion de la mémoire distinguent les lectures des écritures.

3.3.1 Quelques éléments de base de l'optimisation

La classification des optimisations présentées au long de ce chapitre matérialise une séparation entre les optimisations « contextuelles » qui reposent sur la connaissance d'abstractions (action, session, conversation, vues, transaction, ...) associées aux modèles d'exécution, et les optimisations « non contextuelles » qui ne se réfèrent pas au modèle d'exécution. Les optimisations « contextuelles » posent, a priori, un problème complexe en ce qui concerne l'implantation « système » de la gestion de cohérence. Les optimisations « non contextuelles » reposent, par contre, sur un petit noyau de propriétés attachées aux opérations. Un gestionnaire peut donc exploiter ces propriétés d'une façon indépendante du modèle d'exécution.

Quatre propriétés constituent notre noyau :

La mutabilité désigne le fait que l'exécution de l'opération change potentiellement l'état d'un objet.

L'observation désigne la propriété qu'a une opération de pouvoir influer sur l'apparition et donc l'exécution d'opérations ultérieures.

L'action désigne la propriété qu'a une opération de modifier le résultat d'observations ultérieures.

L'absorption définit la propriété qu'a une opération de déterminer, par sa seule exécution, un état de la donnée. Autrement dit, elle « efface » l'effet des opérations précédentes dans l'histoire d'un processus. Toute les affectations ont cette propriété. Tous les protocoles à invalidation exploitent cette propriété.

La *mutabilité* est une optimisation du contrôle de la convergence des réplicats, donc de la « dimension » consensus. Une opération non mutable, ne modifiant pas l'état, n'a pas d'effet sur la convergence du réplicat. Deux opérations non mutables sont, en général, commutatives.

La constitution du couple *observation/action* est une optimisation permettant de capturer la causalité plus finement. La causalité repose sur la perception, l'observation des causes. Une opération qui n'observe pas d'état ne peut induire une relation de causalité entre cet état et l'exécution d'autres d'opérations. La propriété de non observation des écritures aveugles[18] est indispensable à la correction de la sérialisibilité basée sur l'équivalence de vues. Or ces écritures constituent la seule optimisation permise par ce type de sérialisibilité sur la sérialisibilité basée sur les conflits, utilisée traditionnellement [BHG87, chapitre 2] dans les systèmes de base de données.

Finalement, l'absorption est une optimisation sur la « dimension » consensus. Elle permet aux processus d'ignorer toute divergence sur l'ordre ainsi que sur les opérations précédant une opération absorbante. Toute opération d'écriture sur un scalaire a cette propriété. La règle de THOMAS (TWR, « Thomas'Write Rule ») utilisée dans les ordonnanceurs hybrides [BHG87, chapitre 4] exploite cette propriété. Ces ordonnanceurs sérialisent les conflits `écriture-écriture` au moyen d'estampilles. La règle de THOMAS permet d'ignorer les écritures arrivant (perçue) tardivement à l'ordonnanceur ; elle exploitent la propriété d'absorption des écritures tardives sur les écritures ayant eu lieu précédemment.

Ces quatre propriétés sont impliquées dans la définition de la causalité, dans les formes de cohérences séquentielles comme de la sérialisibilité basées sur l'équivalence de vues ainsi que sur une bonne partie des formes de la commutativité.

Comme on le devine, la classification usuelle `lecture/écriture` englobe confusément l'ensemble de ces propriétés. On peut parler de lecture pour désigner indifféremment une opération

[18] Lorsque une transaction émet une écriture non précédée par une lecture, cette écriture est dite aveugle.

d'observation, non mutable, et d'écriture pour désigner tout aussi indifféremment une opération mutable, aveugle et absorbante.

3.3.2 Du comportement des optimisations vis-à-vis de la latence

Nous avons fait un tour raisonnablement exhaustif des différentes optimisations pouvant s'appliquer à notre problèmatique mais qu'en est il de leurs rapports vis-à-vis de la latence ? Le bilan est mitigé.

Des optimisations basées sur l'exploitation de la topographie des accès, on ne peut dégager une réponse simple et unique, et ce, même en ce qui concerne la simple granularité du contrôle de concurrence. On peut d'une part penser que les gains en concurrence amenés par des verrous à grain fin compensent largement la prise de verrous à gros grain. Cependant, la prise même des verrous implique des coûts distants (cas pessimiste), ce qui ne nous permet pas de tirer de conclusions définitives. En pratique, le problème reste le même que celui auquel font face les moniteurs transactionnels quant à la décision de pratiquer le changement de granularité des verrous (« lock escalation »). Le choix se fait sur une estimation du ratio entre l'augmentation de la latence induite par la perte de la concurrence et la diminution de celle-ci par la diminution du temps de prise de verrou ; dans le cas pessimiste les deux coûts sont potentiellement distants, i.e. d'ordre de grandeur comparable à la latence de communication entre les processus.

On peut être légèrement plus conclusif quand aux optimisations basées sur la nature des opérations. La conduite de base « maximiser les connaissances globales statiques » est intuitivement payante ; tout consensus défini préalablement entre les processus se fait sans communication et sans délai. Cette conduite est tellement naturelle qu'elle est systématiquement appliquée, de façon plus ou moins implicite[19]. En fait, seules les optimisations qui exploitent des connaissances statiquement connues par tous assurent un gain trivial : elles ne nécessitent pas de synchronisations et donc pas d'envois de messages. Le caractère statique peut porter sur la valeur des données ; la réplication de données non modifiables en est un exemple typique. Le caractère statique peut être de second ordre, concernant par exemple la loi d'évolution de la donnée. L'exploitation de ce trait est plus complexe et dépend de la connaissance de la sémantique de l'information répliquée. Les mises en œuvre de mondes virtuels se servent de l'universalité des principes de la cinématique, comme des lois de la dynamique pour estimer (« dead-reckoning ») localement la position future des avatars distants : les avatars peuvent bouger, mais en l'absence d'interaction, leur mouvement est régi par les lois de la dynamique et les lois de la cinématique, qui sont universelles. Encore faut-il, pour les exploiter, que la gestion de cohérence soit capable d'identifier correctement ce type d'opération : un gestionnaire de cohérence, ne voyant d'un avatar que sa représentation mémoire, ne peut distinguer parmi les opérations de `lecture`/`écriture` celles induites par une interaction utilisateur, de celles résultant de l'application des lois de la cinématique.

Dès que l'on sort du cadre statique, rien n'est plus trivial. Dans le cadre des optimisations basées sur des propriétés dynamiques, de grandes variations de complexité peuvent apparaître. Considérons l'exemple du jeu et supposons que la borne de précision sur la position des objets ne soit plus statique mais dynamique, dépendante par exemple de leurs éloignements des

[19] A un moment ou un autre, toute coopération repose sur un consensus *inné*. Toute communication, aussi simple soit-elle, ne peut être construite que sur des a priori (e.g. protocoles) communs (TCP/IP pour Internet, par exemple).

observateurs ou de leurs vitesses de déplacement[20]. Non seulement l'état du système est dynamique, mais son critère de correction aussi ; de plus, le critère dépend de ce même état ! Quand aux optimisations basées sur des critères d'ordonnancement dynamique (équivalence de vues, par exemple) de méta-opérations, leur complexité analytique[21] est invoquée pour justifier de leur non utilisation (dans les BD). Nous pensons que ces justifications, dont certaines sont très théoriques[22], ne doivent s'appliquer qu'avec prudence dans notre contexte. Ces justifications reposent en partie sur le fait que les coûts locaux d'ordonnancement (les prises de verrous, en particulier) sont le facteur critique de performance. Dans le cas réparti, le coût des communications peut contrebalancer sérieusement les coûts de traitement locaux.

On peut néanmoins se baser sur des traditions de bon sens : l'agrément sur des propriétés dynamiques implique des latences de l'ordre de grandeur de la latence réseau. La causalité est assurément une propriété dynamique plus économique, mais la latence est plus complexe à déterminer ; elle dépend de la distribution des vitesses de propagation dans le groupe et des schémas (« pattern ») d'émission/réception de chacun des membres (ordonnancement relatif des messages, dans le groupe).

3.3.3 De la complexité de l'implantation des optimisations

Une gestion de cohérence assure la correction d'une exécution répartie en se fondant sur un critère de cohérence. La mise en œuvre de cette gestion doit donc maintenir un contexte minimum : les informations nécessaires à l'évaluation de ce critère. Le minimum à maintenir varie selon notre classification ; ceci est sans surprise, dans la mesure où la complexité d'une mise en œuvre est en grande partie dictée par les interactions entre le code gérant les ordonnancements et l'application.

Les optimisations « non contextuelles » n'ont besoin de maintenir que les structures des opérations en cours, i.e. concurrentes. De plus, le contexte associé à une opération étant fixe, une fois l'opération initiée, ce contexte peut être maintenu au moyen de structures efficaces (tables de hash, B-tree), permettant une évaluation peu coûteuse du critère de cohérence.

Les optimisations « contextuelles », basées sur la construction de méta-opérations (action, blocs `acquire`/`release`, ...) ou bien sur la causalité, nécessitent le maintien d'un contexte (histoires, versions, intentions, estampilles, mesures, ...) qui doit survivre à la terminaison locale d'une opération. Ce contexte doit être mis à jour et son expansion contrôlée (troncature de journal ou de contexte, ramasse-miettes). Dans le cas pessimiste, la nature évolutive du contexte rend plus aléatoire une maintenance efficace ; l'évaluation du critère de correction n'en devient que plus coûteuse. Dans le cas optimiste, ce dernier problème est reporté sur la taille des structures nécessaires pour maintenir le contexte.

3.4 Résumé

Ce chapitre répond à deux préoccupations essentielles d'une fonction de gestion de cohérence ; quelles informations dois-je maintenir ? Que me coûte-il de les maintenir ?

[20] Implantation du principe d'*aura* [HLS97] dans les mondes virtuels.

[21] Assurer la sérialisibilité en se basant sur l'équivalence de vue est un problème **NP** complet [BHG87, section 2.6] et [GR93, section 7.7.5]

[22] Nous pensons à l'impossibilité de construire un ordonnanceur « efficace » pour la sérialisibilité basée sur l'équivalence de vue donné par Papadimitriou [Pap79]

Pour ce faire, nous avons brossé un état de l'art des connaissances exploitées par les gestionnaires de cohérences implantés dans quatre grands domaines de l'informatique répartie (BD, SOR, SCG, MPR) pour statuer de la correction d'une exécution. Cet état de l'art a fondé une analyse des rapports de ces connaissances avec l'agrément et la causalité, de leurs rapports avec la latence et de leur coût de mise en œuvre. Cet analyse a posé l'exploitation de ces diverses connaissances en termes d'*optimisations*. Une classification rudimentaire (« contextuelle », « non contextuelle ») de ces optimisations a permis de définir un petit noyau de quatre propriétés génériques (*mutabilité*, *action*, *observation*, *absorption*) qui, associées à toute opération, permet à la gestion de cohérence d'implanter une bonne part des optimisations entrevues. Nous avons présenté une modeste étude de la complexité de mise en œuvre ; cette complexité recoupe les critères de notre classification.

Chapitre 4

De l'adaptabilité

Notre propos dans ce chapitre est de comprendre comment implanter l'adaptabilité sur le plan du modèle de cohérence. Tous les modèles de cohérence faible doivent définir un protocole permettant à aux applications de caractériser leurs opérations[1]. Seuls certains sont conçus pour que les règles qui régissent les ordonnancements des opérations, i.e. le *modèle* de cohérence lui-même, puissent être adaptées. Les degrés d'isolation sont un des premiers exemples d'adaptabilité ; Psync, Munin et « Lazy replication » en sont d'autres. Nous avons sélectionné dans cette étude un représentant de chacun des domaines concernés par la cohérence :

- Les bases de données (BD) : ϵ-sérialisibilité.
- Les systèmes à objets répartis (SOR) : Bayou.
- Les systèmes de communication de groupe (SCG) : Horus.
- Les mémoires partagées réparties (MPR) : Arias.

Parmi les travaux étudiés, ϵ—sérialisibilité constitue une exception : il ne s'agit pas d'une architecture, mais d'un critère de correction qui n'a pas été. À notre connaissance, celui-ci n'a pas été mis en œuvre.

4.1 ϵ-sérialisibilité

L'ϵ-sérialisibilité [PL90, RP95] est un critère de correction étendant la sérialisibilité. Ce critère permet à des transactions en lecture (« query transaction ») de voir, de façon contrôlée, l'effet de transactions non validées ; ce critère rompt donc, partiellement, l'*isolation* des transactions, telle qu'assurée par la sérialisibilité. Alors que les degrés d'isolation [GR93] se définissent vis-à-vis du respect de dépendances de bas niveau, l'ordre des lectures/écritures sur les objets, ϵ-sérialisibilité, en s'appuyant sur l'existence d'une métrique sur les données, se définit sur des critères applicatifs : telle consultation de mon compte peut être imprécise à 100 francs près ; la consultation de mon compte peut ne refléter que 80 % du nombre des opérations effectuées ; la consultation de la position d'un train ou d'un avion peut être précise à 500 m près.

Le principe d'ϵ-sérialisibilité est le suivant : chaque mise à jour sur un item de la base modifie la valeur de cet item d'une grandeur (distance), `export-consistency`, définie par l'opération de mise à jour. Chaque transaction en lecture spécifie une grandeur, `import-limit`,

[1] Sous la forme d'un étiquetage des opérations, de spécifications de dépendances où encore de règles de conflit.

définissant l'imprécision qu'elle tolère sur la valeur (en pratique cumulée) de ses lectures. Les extensions du contrôle de concurrence associées à l'ϵ-sérialisibilité, assurent que l'exécution converge vers une exécution sérialisable, et que les effets de *non* isolation sont bornés par cette grandeur. La convergence de la base est assurée, en interdisant aux transactions pratiquant des mises à jour, de lire des valeurs non validées (`import-limit` $= 0$)[2]. ϵ-sérialisibilité intègre donc, dans son critère de correction, des fonctions supérieures de l'ordre (cf. section 3.2.2). ϵ-sérialisibilité, par contre, impose la convergence terminale (« eventual consistency ») ; cette contrainte limite l'utilisation de cette imprécision à des transactions en lecture uniquement.

Un des grands atouts d'ϵ-sérialisibilité est de pouvoir s'intégrer simplement dans des principes de contrôle de concurrence variés. Wu, Yu et Pu dans [WYP97] montrent comment les ordonnanceurs basés sur les principes classiques (verrouillage à deux phases, ordonnancements par estampilles, et certifieur) peuvent être étendus afin d'intégrer l'ϵ-sérialisibilité. Ces auteurs présentent aussi, en s'appuyant sur des résultats de simulation, les gains à espérer de l'utilisation de ϵ-sérialisibilité.

ϵ-sérialisibilité, par contre, n'est qu'un critère de correction. ϵ-sérialisibilité permet à une application de spécifier ses contraintes de correction mais ne fournit pas d'indications sur leur pertinence vis-à-vis de la latence de la perception. Pu et Leff [PL91] décrive un bon comportement analytique des optimisations permises par ϵ-sérialisibilité dans le cadre répliqué.

4.2 Bayou

Bayou [TTP+95] propose une infrastructure système permettant à des applications coopératives, du type agenda ou courrier réparti, de fonctionner dans un environnement habituellement partitionné. L'information partagée est implantée comme une collection de données répliquées sur un ensemble de serveurs. Un client accède à l'information en contactant un des serveurs. Les accès sont classés selon deux catégories : requêtes ou mises à jour. Les clients sont des nomades potentiels ; ils peuvent accéder et modifier l'information sur différents serveurs. Les serveurs se contactent de façon opportuniste et s'échangent, deux à deux, leur mises à jour manquantes (sessions d'anti-entropie).

Bayou ne traite pas, à proprement parler, d'objets et encore moins d'objets atomiques. Bayou, par contre, associe à chaque mise à jour sur la base, une procédure de certification (« dependency check ») et une procédure de réconciliation (« merge procedure »). En ce qui concerne la gestion de cohérence, ces mises à jour sont comparables à des opérations définies sur des types de données abstraits.

Bayou stabilise lui même les mises à jour : à chaque collection de données est associé un unique serveur dit primaire (« primary ») ; une mise à jour est stabilisée au moment et dans l'ordre[3] où elle est reçue par le serveur primaire. Les mises à jour stabilisées sont ensuite propagées, comme les autres mises à jour, selon le protocole anti-entropique de Bayou [PST+97]. Ce protocole assure que les mises à jour stabilisées précèdent toute autre mise à jour et que l'ordre des mises à jour, stabilisées ou non, respecte l'ordre causal[4].

Les contraintes de correction sont de façon ultime assurées par les procédures de certification et de réconciliation ; le modèle de cohérence n'est donc pas défini par Bayou mais par

[2] Le modèle de spécification permet de *spécifier* des imprécisions de lecture pour de telles transactions, mais l'effet sur la base ne peut plus en être contrôlé.

[3] Aucune garantie d'ordre n'existe entre collections de données distinctes.

[4] Les canaux de causalité cachés induits par le nomadisme sont pris en compte par les garanties de session [TDP+94].

l'utilisateur. Ceci correspond à la stratégie essentiellement optimiste de Bayou. Néanmoins, pour améliorer le comportement et la vivacité de l'application, Bayou implante des sessions. Une session Bayou représente l'histoire des opérations effectuées par un client. Bayou attache cette histoire *au client*. Cette histoire le suit tout au long de ses déplacements. Ces sessions ne jouent aucun rôle sur la propagation des mises à jour. Elles permettent, par contre, de prévenir le client de l'apparition de certaines anomalies lorsqu'au cours de ses pérégrinations, il se connecte à un serveur n'ayant pas perçu l'ensemble des opérations que lui, client, a pu percevoir. Bayou définit des propriétés d'ordre basées sur ces sessions, dénommées garanties de session [TDP+94] . Elles sont au nombre de quatre :

RYW Lectures de ses écritures (« Read Your Writes »). Une lecture d'un client succédant à une écriture de ce même client sur la même donnée au cours d'une même session retourne une valeur au moins aussi récente que l'écriture (des écritures concurrentes par d'autres agents peuvent par contre écraser l'écriture de l'agent en question).

MR Lectures monotones (« Monotonic Reads »). Des lectures successives retournent des valeurs équivalentes ou plus récentes.

WFR Les écritures succèdent les lectures (« Writes Follow Read »). Lorsqu'un client effectue une lecture suivie d'écritures, ces écritures succèdent aux écritures ayant précédé la lecture du client.

MW Écritures monotones (« Monotonic Writes »). L'ordre des écritures effectuées par un client au cours d'une session est conservé.

Les garanties de session jouent deux rôles : (1) elles servent de critère pour choisir un serveur, lorsqu'un client a le choix entre plusieurs serveurs auxquels il peut se connecter. (2) Elles servent de critère pour déclencher une « alarme », signalant au client qu'aucun des serveurs à sa disposition n'assurent les propriétés d'ordre spécifiées. Il s'agit de serveur jugés, par le client, trop en retard pour pouvoir lui être utile.

Chacune des garanties de session représente une forme limitée de la causalité. La possibilité de réaliser ces propriétés sans agrément est la raison fondamentale pour laquelle elles peuvent être utilisées dans un système déconnecté. Ces propriétés permettent de limiter le taux d'abandon et donc de garantir la vivacité de l'application tout en restant compatibles avec la nature essentiellement déconnectée de l'environnement.

En soi, ces propriétés ne définissent pas des contraintes de correction[5] ; elles constituent un compromis pessimiste qui permet de *prévenir* des conflits dont le nombre impliquerait un manque de vivacité, et en fait une démission trop importante de la gestion de cohérence. Dans un système optimiste, les réconciliations qui ne peuvent être prises en charge par le système sont exposées à l'agent. Lorsque celles-ci deviennent trop fréquentes, la gestion de cohérence est en fait assurée non plus par le système mais par l'agent.

Bayou correspond certainement à l'état de l'art en terme de réplication pour les applications coopératives. Bayou représente un peu le pendant optimiste de Isis ou de Horus. Par contre, Bayou ne supporte que des types de coopérations asynchrones pouvant être abandonnées ; mais ceci est le propre des stratégies optimistes.

L'optimisme offre une certaine échappatoire à l'optimisation des propriétés de cohérence. Peu d'opérations peuvent être réconciliées sans intervention humaine. L'optimisme, en se basant sur cette bouée de secours, laisse de coté la maîtrise fine des agréments répartis et se

[5] D'autant plus que ces contraintes concernent l'ordre des écritures de première intention ; cet ordre peut différer de l'ordre des écritures stabilisées.

contente d'assurer une bonne réactivité en implantant des opérations de façon asynchrone. Mais seulement au « commit », i.e. lors de la fusion (« merge procedure »), peut-on vraiment savoir s'il y a eu progression de l'application.

Peut-être Bayou, au même titre qu'Horus, pourrait-il servir de plate-forme pour des systèmes hybrides optimiste/pessimiste, certains travaux [AC92] montrant des résultats prometteurs dans ce type de stratégie.

4.3 Horus

Comme son prédécesseur Isis (voir section 3.2.1.1), Horus [vRBM96] et [Bir96, chapitre 18] fournit une infrastructure modulaire facilitant le développement d'applications tolérantes aux fautes au travers du modèle de synchronie virtuelle.

La vision qu'a un utilisateur de Horus est celle d'un embout de communication évolué ; un embout de communication est constitué d'une pile de protocoles [vRBF+95, RHB94] ; chaque protocole est une réalisation d'un même type abstrait : tous les protocoles partagent les mêmes interfaces (HCPI « Horus Common Protocol Interface ») ascendantes (« upcall ») et descendantes (« downcall »). Ces interfaces représentent, essentiellement, l'union des fonctionnalités de la gestion d'appartenance à un groupe (`join`, `leave`), de celle de l'échange (`send`,`cast`) et de la stabilisation de message (`stable`).

La synchronie virtuelle modélise les critères de correction applicatifs en contrôlant l'ordonnancement de la *livraison* des messages ainsi que l'ordonnancement de la livraison de ces messages vis-à-vis d'événements comme le départ, l'arrivée ou l'initialisation d'un membre. Dans un système asynchrone, le temps n'étant représenté que par l'ordre des événements du système, la synchronie virtuelle assure que les changements de composition du groupe, départ (volontaire ou panne) ou arrivée d'un membre, sont perçus « au même instant » par l'ensemble des membres, i.e. lorsque tous les membres encore présents ont reçu le même ensemble de diffusion (ou « multicast »).

Horus permet de composer la mise en œuvre d'un grand nombre de fonctionnalités (interface réseau, compression, authentification, journalisation, ...). Sur le plan de la cohérence, Horus se focalise sur le thème de la tolérance aux fautes. La boîte à outils Horus propose les trois sémantiques de diffusion `fbcast`, `abcast` et `cbcast`, plus le `gbcast` utilisé essentiellement pour la gestion de la composition du groupe (GMS, « group membership »). Néanmoins, toute autre sémantique, dès lors qu'elle peut se satisfaire de l'interface HPCI de Horus, est à priori implantable comme tous les autres protocoles.

Vis-à-vis de la cohérence, la limitation d'Horus est inhérente à sa fonction. Horus ordonne la *livraison* de messages non typés. Une bonne part des optimisations évoquées au chapitre précédent (commutativité, fonctions supérieures de l'ordre), ne sont pas réalisables avec ces abstractions. Quoiqu'il en soit, l'excellence de ses performances [RHB94, vRBM96] le fait apparaître comme un bon outil pour la mise en œuvre pessimiste de modèles de cohérence plus élaborés.

4.4 Arias

Arias définit une architecture de MPR persistante à cohérence adaptable, conçue pour des applications coopératives [DHMR96]. Nous nous focalisons ici sur les traits liés à la gestion de cohérence dans Arias [PCM97].

Arias implante un espace d'adressage virtuel partagé unique. Cet espace est découpé en segments persistants disjoints constitués de pages. Arias permet à une application de définir une *zone*, i.e. un ensemble d'octets contigus de taille quelconque (pouvant être inférieur à la taille d'une page) associé à un segment. Arias permet à toute application d'associer à une zone un gestionnaire de cohérence de son choix (protocole spécifique (PS)), chargé de la maintenance de la cohérence de cette zone. L'application peut dynamiquement changer le gestionnaire associé à une zone. Chaque site incarnant localement une zone possède une instance du gestionnaire de cohérence associée à cette zone[6]. À tout moment, une zone est associée à un site maître unique ; les incarnations distantes de cette zone sont dites esclaves, de même que les instances des gestionnaires qui en ont la charge.

La couche générique de cohérence (CGC) d'Arias offre aux gestionnaires de cohérence, deux fonctionnalités principales :

- La communication point à point (`CGC_SendMsg`) qui permet aux gestionnaires de cohérence de s'échanger des messages ; les échanges ont lieu typiquement entre les instances réparties d'un même gestionnaire.
- Le changement atomique de maître de zone. Ce changement est effectué par la même primitive de communication (`CGC_SendMsg`) appelée avec un paramètre adéquat. Cette méthode permet de délivrer un message, typiquement une copie fraîche de la zone, atomiquement avec le changement de maîtrise de la zone.

L'interface exportée à l'application par les gestionnaires de cohérence est propre à chaque gestionnaire. Il s'agit typiquement de primitives de mise en exclusion mutuelle (`lock`, `unlock`). Les accès sur la mémoire[7] et leurs contrôles par un gestionnaire de cohérence sont, en ce qui concerne Arias, indépendants. Lorsque le modèle de cohérence repose sur la connaissance précise de ces accès, l'application doit accéder à la mémoire au travers du gestionnaire de cohérence dont l'interface est alors étendue[8] (`Read`(), ou `Write`() dans le cas de « release consistency » [PCM97, section 3.1.5]).

Arias propose en fait une infrastructure spécialisée pour les gestionnaires de cohérence basés sur un privilège tournant. Arias offre à ces gestionnaires l'atomicité du passage du privilège avec les mises à jour susceptibles de l'accompagner.

La limitation d'Arias vient de ce que le privilège tournant est un principe de mise en œuvre d'agrément réparti. Ce principe est excessivement fort pour tous les critères de cohérence reposant sur la causalité, tel que la cohérence causale [AHJ91] ou simplement la cohérence FIFO (« processor consistency » [Mos93]), pour les critères basés sur des fonctions de l'ordre (métrique), et en fait pour une bonne partie des optimisations évoquées au chapitre précédent.

4.5 Résumé

Les travaux qui ont donné lieu à une mise en œuvre proposent un mélange de propriétés de cohérence et de mécanismes. La définition d'une interface entre l'application et le contrat de cohérence n'est pas abordée au profit de la spécification d'interface sur les mécanismes utilisés pour l'assurer. Cette démarche n'est viable qu'à partir du moment où la complexité laissée

[6] L'ensemble des gestionnaires de cohérence forme d'un certaine manière un groupe ; mais Arias n'implante aucun mécanisme de groupe, ni gestion de composition (« group membership » (GMS)) ni diffusion.

[7] Le contrôle de l'incarnation (« mapping ») d'une page se fait de façon traditionnelle, par gestion des défauts de page générée par la MMU lors du premier accès à celle-ci.

[8] Dès lors, Arias ne présente plus une API de mémoire à l'application.

à la charge du programmeur d'application reste suffisamment faible pour que la réutilisation de cette partie de code ne présente pas d'intérêt. Seul Bayou et ϵ-sérialisibilité présentent cette caractéristique. Or ϵ-sérialisibilité n'adresse pas la latence induite par la communication (les calculs et la vérification du critère de divergence sont nécessairement globaux). Bayou n'adresse la latence que sur le seul plan de l'optimisme. En fait, l'aspect propre à la latence de communication n'est pris en compte, en terme de *propriété* de cohérence, que par les travaux situant la gestion de cohérence au niveau de la communication ; c'est-à-dire Horus et Arias. Malheureusement ces travaux ont une représentation de l'exécution, messages ou simple classification en lecture/écriture, qui ne leur permet pas de considérer les optimisations telles que la commutativité générale, les dépendances représentées dans la spécification des semi-queues ou même le contrôle de divergence basé sur des métriques ; autrement dit, des optimisations prometteuses dans le cadre de nos applications coopératives.

Chapitre 5

Synthèse

Au cours de cette première partie, nous avons considéré un modèle de cohérence comme l'ensemble des contraintes prémunissant l'application contre les incorrections engendrées par la concurrence et la réplication. Ces contraintes se définissent soit par des contraintes sur l'ordre d'exécution (l'ordonnancement) des opérations ou des méta-opérations soit par des contraintes sur des fonctions de cet ordre (métriques). La nature de ces contraintes et ce qu'elles requièrent pour leurs mises en œuvre ont été présentés. Les contraintes indépendantes du contexte d'exécution sont les plus simples, les plus complexes étant définies par des fonctions de l'ordre sur des méta-opérations. L'état de l'art des connaissances permettant à une application d'adapter la gestion de cohérence aux contraintes qui lui sont nécessaires et suffisantes a enfin été présenté. Ces connaissances se découpent, grossièrement, selon deux axes : 1) ceux (BD, SOR) qui construisent les contraintes d'ordonnancement au niveau de l'interaction entre les méta-opérations concurrentes, i.e. les conversations, sessions, actions ou transactions. 2) ceux (SCG, MPR) qui construisent les contraintes d'ordonnancement au niveau de la représentation des données partagées par lesquelles ces interactions sont effectuées et perçues, le média de communication en quelque sorte.

Des travaux basés sur l'ordonnancement des méta-opérations, aucun ne considère les rapports entre propriétés réparties et latence. Les systèmes centrés sur les bases de données sont attachés à la notion de transparence, en particulier de transparence de la répartition[1]. Les systèmes basés sur les types abstraits adhèrent à l'optimisme mais minimisent la démission de la gestion de cohérence (prise en charge par l'agent) que cela présuppose. La conséquence en est une spécialisation extrême à quelques applications bien ciblées (gestion d'agenda, gestion de messagerie).

Les travaux visant à munir les média de communication de propriétés de cohérence buttent sur deux problèmes : 1) ils opèrent à un niveau ayant perdu une partie essentielle de la sémantique des opérations, 2) ils sont confrontés au traditionnel argument du « bout-en-bout » (« End-to-End Arguments » [SRC84]) : attacher des propriétés aux couches basses d'un système induit un coût ; ce coût est inutile dès lors que ces propriétés ne sont pas suffisantes et doivent être réimplantées au niveau des couches supérieures [CS93]. Ces travaux se focalisent sur le second problème en s'attachant à maintenir un surcoût faible et délaissent le premier. Ces travaux permettent ainsi de constituer des composants qui, de part leurs performances,

[1] Certains modèles tels les Saga [GR93] se focalisent sur les multi-bases dont le problème crucial est la latence. Mais le modèle recourt à des mécanismes de compensation, qui ne garantissent ni la sérialisibilité ni aucun contrôle de divergence ; le modèle repose donc de fait, comme l'optimisme, sur la présence d'un agent intelligent pour éventuellement assurer la correction finale.

deviennent des briques de construction privilégiées pour les systèmes répartis. Par contre, ces composants ne suffisent pas à assurer une solution de « bout-en-bout ».

Au bout du compte, les plate-formes de communication (SCG, MPR) sont les seuls outils dont disposent actuellement un programmeur d'applications coopératives nécessitant une certaine « réactivité temporelle » (« soft real-time »)[2] et ne se satisfaisant pas des contraintes liées à l'optimisme, en particulier la progression différée de façon arbitraire, et la prise en charge explicite des réconciliations.

Autrement dit, dans le cadre de l'état de l'art, soit on place la gestion de cohérence à un « haut niveau » propre à mettre en œuvre le meilleur des optimisations et l'on se trouve ou bien confronté à la transparence ou bien à une coopération lâche voire une démission de la gestion de cohérence, soit on place la gestion de cohérence à un « bas niveau » assurant un contact direct avec la latence induite par la réplication, mais on a alors perdu les abstractions (méta-opérations, modèle d'exécution) qui permettent de pratiquer de bonnes optimisations.

À l'issue de cette première partie, nous sommes en mesure de répondre à trois questions fondamentales :

- *Qu'est ce qu'un modèle de cohérence ?* Un critère de correction défini comme l'ensemble des ordonnancements admissibles pour une application.
- *Qu'est ce qu'un modèle de cohérence faible ?* Une optimisation des ordonnancements admissibles basée sur l'exploitation de traits applicatifs.
- *En quoi l'état de l'art est-il insuffisant vis-à-vis de l'adaptabilité ?* En ce qu'il ne fournit pas de solution permettant de construire une gestion de cohérence qui, simultanément, assure la responsabilité d'un critère de correction (contrat de cohérence), implante les optimisations requises pour assurer la réactivité des applications coopératives que nous visons à supporter et qui soit enfin réutilisable.

Nous pouvons maintenant reformuler plus précisément le problème de la réactivité d'application coopérant de façon distante. Ce problème découle de deux contraintes apparemment contradictoires :

Contrainte d'adaptation. La réactivité d'une application coopérative repose sur la bonne adéquation entre les ordonnancements permis par la gestion de cohérence et les requis de correction applicatifs. Cette adéquation ne peut être jugée de façon pertinente qu'à l'exécution et doit se baser sur des critères applicatifs.

Contrainte de réutilisation. L'exploitation de traits applicatifs induit des dépendances entre la gestion de cohérence et l'application. Cette dépendance et son exploitation dans un système réparti nécessitent une expertise[3] qu'il est utopique d'attendre de tout programmeur d'application. Ce code complexe doit donc être fourni au programmeur d'application. Ceci nécessite que son coût de développement soit amorti sur plusieurs applications. Il doit donc être réutilisable.

L'adaptation de la gestion de cohérence, à l'exécution et sur des critères applicatifs et environnementaux, permet l'obtention de bonne performance. La réutilisation de code dont la correction est déjà statuée, permet de diminuer les coûts de développement applicatifs, voire même simplement de le permettre.

[2] Jeux répartis, environnements partagés pour entreprises virtuelles, éditeurs coopératifs, ...

[3] L'écriture du code de gestion de cohérence requiert la maîtrise des protocoles de communication, de l'écriture de code concurrent asynchrone et de l'algorithmique répartie complexe (ordonnancement et contrôle de concurrence réparti, gestion de groupe dynamique).

Comme nous l'avons vu dans les deux chapitres précédents, aucune de ces contraintes, prise isolément, n'est nouvelle. MARZULLO et BIRMANN, en particulier, les avaient déjà identifiées [MB89, Bir94]. Ce qui est nouveau, c'est le contexte dans lequel elles apparaissent et leur interdépendance. La contradiction apparente de ces deux contraintes réside en ce que la réutilisation du code de gestion de cohérence suppose une forme de généricité alors même que la complexité de ce code est en partie induite par sa dépendance avec le code applicatif.

Notre thèse est que l'on peut résoudre cette contradiction, en se basant sur une constatation : la partie de la gestion de cohérence qui doit pouvoir être adaptée par l'application n'est pas celle dont la mise en œuvre est dépendante du code applicatif, bien qu'elle interagisse avec cette dernière.

L'architecture **Core**, que nous proposons comme réponse au problème, découle de cette constatation. Le cœur de cette architecture consiste en un composant «système», le *gestionnaire de cohérence*, choisi et instancié à l'exécution par l'application. La nature «système» et la réutilisation de ce composant est obtenue en rendant sa mise en œuvre indépendante du code applicatif. L'efficacité de ce composant est conservée en lui permettant de contrôler, de façon abstraite, deux aspects intimement liés au code applicatifs, et qui ont un rôle clé vis-à-vis de la gestion de cohérence : la nature du contrôle de l'exécution d'une opération (synchrone/asynchrone, pessimiste/optimiste, processus/transaction) et la nature des propriétés d'une opération (mutabilité, observation, action, absorption) vis-à-vis de son interaction avec les autres opérations (ou méta-opérations). La latence est pilotée par l'utilisation de ces propriétés dans le cadre d'un modèle de cohérence.

Nous pensons que, à la manière des graphes de contexte de Psync, l'on peut réifier ces deux aspects moyennant un surcoût acceptable. Cette réification permet de concevoir une gestion de cohérence adaptable assurant des *propriétés* de cohérence répondant bien à la latence et dont la mise en œuvre est isolée de ces structures d'exécution, tout en permettant l'exploitation optimale de leurs sémantiques.

Cette thèse est développée dans la suite de ce document.

Deuxième partie

Éléments de solution

Cette deuxième partie est consacrée à l'exposé de nos propositions.

Ces propositions adressent une catégorie restreinte d'applications coopérantes : celles qui permettent à des agents *fixes* de coopérer en partageant une information au dessus d'Internet. Ce sont ces restrictions qui donnent un sens au choix de la réplication et à la décision d'étudier la cohérence de l'information partagée dans un cadre asynchrone.

Cette partie décrit, en partant du problème de départ, la latence et la coopération, le raisonnement qui nous a amené à définir l'architecture **Core**.

Elle comporte quatre chapitres qui présentent respectivement :

1. La *réplication coopérante*, le modèle de partage utilisé pour la coopération. Nous lui associons un formalisme, basé sur des histoires répliquées. Ce formalisme est l'outil qui nous permet d'étudier, de comprendre et de modéliser les propriétés que l'application attend de la gestion de cohérence.
2. Les principes de décomposition du critère de cohérence qui permettent d'adresser le point clé de notre problématique : la contradiction *adaptation/réutilisation* évoquée au chapitre précédent.
3. Les principes de factorisation qui permettent de mettre en œuvre les différents critères issus de cette décomposition, sous la forme de composants objets.
4. L'architecture **Core**, qui décrit l'intégration d'une version simplifiée de ces composants dans une application répartie.

Le premier chapitre adresse le problème de la performance, ce de deux façons : (1) en utilisant l'effet de cache associé à la réplication totale de l'information partagée ; (2) en permettant de modéliser en toute liberté toutes les formes de contraintes sur l'ordre des opérations, reflétant ainsi les modèles variés de cohérence que nécessitent les applications coopérantes. Il s'agit, en quelques sortes, d'une « modélisation » de l'adaptabilité.

Les trois autres chapitres adressent la faisabilité et la mise en application des solutions qui permettent d'implanter cette adaptabilité. Cette faisabilité repose sur la possibilité de *réutiliser* la partie de la gestion de cohérence sur laquelle porte l'adaptation.

Chapitre 6

Construction de la coopération

Ce chapitre décrit nos propositions concernant la construction de la coopération entre agents fixes distants. Après une rapide présentation de l'environnement d'exécution, des hypothèses que nous posons sur le système et du modèle de l'application, nous présentons un nouveau modèle de partage, la *réplication coopérante*, dans lequel chaque agent possède un réplicat de l'information partagée.

Ce modèle de partage est la première clé qui nous permet de répondre aux contraintes d'adaptabilité identifiées dans la première partie de ce document. L'adaptabilité se caractérise par la liberté que ce modèle permet dans l'ordre réparti des opérations et par la variété des propriétés de cohérence qu'il permet d'exploiter. L'étude de ces propriétés nous a paru, dans le cadre du modèle à processus communiquant de LAMPORT [Lam78], complexe à mener. Dans ce modèle, non seulement la réplication n'est pas aisée à exprimer, mais de plus, tous les événements locaux d'un processus sont causalement liés. Ces contraintes nous ont amenés à développer un nouveau modèle d'exécution représentant explicitement la réplication, et un formalisme, le formalisme **Core**, permettant de raisonner sur ce modèle d'exécution. Ce formalisme permet de représenter et de raisonner sur les seules propriétés liées à l'ordre dans lesquelles les opérations sont exécutées sur les différents réplicats de l'information partagée. Nous familiarisons ensuite le lecteur avec ce formalisme en exprimant les critères de cohérence les plus connus, dans le cadre de la réplication coopérante. Nous montrons ensuite comment utiliser ce formalisme pour construire d'autres critères, plus faibles, basés sur le contrôle de la divergence des réplicats.

Le principe même de la réplication coopérante se base sur une hypothèse essentielle : les données partagées doivent être *déterministes*, i.e. l'état d'une donnée est complètement défini par trois facteurs : son état initial, l'ensemble des opérations qui y ont accédé et l'ordre dans lequel ces opérations y ont accédé.

Cette hypothèse peut sembler contraignante. Elle reste néanmoins compatible, avec notre ambition de pouvoir représenter les réplicats de la donnée partagée par des composants objets standard du marché. Ceux-ci ont, dans leur grande majorité, une sémantique déterministe.

Enfin, nous nous focalisons sur la cohérence de l'information partagée. Autrement dit, nous assimilons les critères de correction de l'application à des contraintes sur l'état des données partagées.

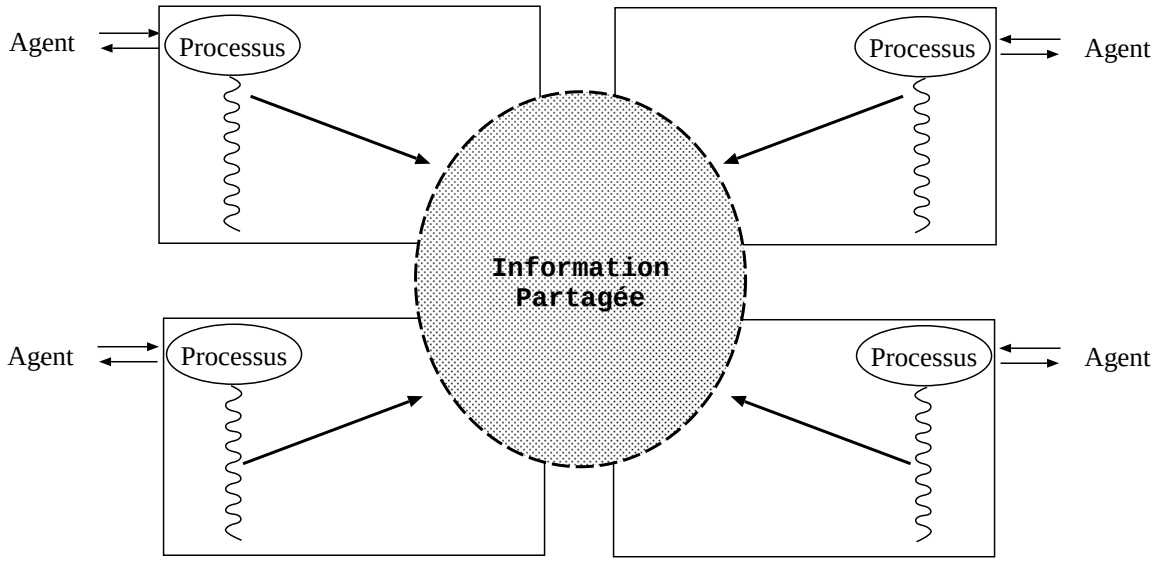


FIG. 6.1: Représentation d'une application coopérative. À chaque agent est associé un processus. Chaque processus réalise un système réactif, dont l'état est modifié (1) par les événements découlant des interactions locales que l'agent entretient avec le processus (clavier, fenêtre X, robot, autre programme, ...) et (2) par les événements découlant de la coopération entre les processus ; ces événements correspondent typiquement aux mises à jours de la représentation de l'information partagée. L'information partagée est partagée par tout processus associé à un agent.

6.1 Modélisation d'une application coopérative

6.1.1 Hypothèses sur le système

Nous faisons deux hypothèses essentielles sur le système. La première concerne l'environnement d'exécution, la seconde l'information partagée :

Environnement d'exécution Notre environnement d'exécution est constitué d'une collection de systèmes d'exploitation traditionnels interconnectés par un réseau. Ces systèmes gèrent l'exécution de processus de type Unix : chaque processus exécute séquentiellement un ensemble potentiellement infini d'instructions. À chaque processus est associé un contexte dans lequel sont représentées les données auxquelles accèdent ses instructions. Les processus communiquent entre eux en échangeant des messages. Nous ne spécifions pas les moyens utilisés pour communiquer. Bien que nous visions essentiellement Internet, nous n'excluons pas l'utilisation de disquettes où de tout autre moyen de communication. Nous ne faisons pas d'hypothèses sur les vitesses relatives d'exécution des processus, ni sur les délais d'acheminement des messages entre processus. Nous nous plaçons donc dans l'hypothèse asynchrone : tout ensemble de processus s'exécutant sur un tel environnement constitue un système réparti asynchrone.

L'information partagée L'information partagée consiste en une collection de données partagées. Ces données doivent pouvoir être représentées sous forme d'état par nos processus. Ces données doivent posséder une sémantique *déterministe*. Les instances de types de donnée abstraits usuels sont pour la plupart munis d'une spécification sérielle déter-

ministe [Wei84], et vérifie notre hypothèse. Nous désignerons le plus souvent par *objet* de telles données. Nous parlerons de même *d'invocations* pour désigner les accès à un objet.

Nous associons à chaque agent un processus. Les agents ne coopèrent que par l'intermédiaire de leur processus.[1] Chaque processus offre, à son agent, une vue de l'information partagée et les moyens de la modifier. Une application coopérative est ainsi représentée par un groupe de processus coopérants, constituant un système réparti asynchrone (cf. figure 6.1).

Typiquement, chaque processus réalise un système réactif [Weg92] à essentiellement deux catégories d'événements : d'une part, ceux qui résultent de l'interaction avec leur agent. Cette interaction se fait au travers d'une interface propre à l'agent (clavier, fenêtre X, robot, programme, ...). D'autre part, ceux qui affectent l'information partagée. Cette interaction dépend du modèle de partage utilisé pour mettre en œuvre la coopération entre processus.

6.1.2 Un modèle de partage : la réplication coopérante

Les deux conditions suivantes caractérisent la *réplication coopérante* :

Réplication totale [Lit91] Toutes les données partagées par le groupe de processus coopérants sont répliquées, selon le schéma suivant : tout processus accédant à la donnée partagée est associé à un unique réplicat de cette donnée. Tout réplicat est incarné dans le contexte d'un unique processus.

Réplicats actifs Chaque réplicat est maître de l'ordre dans lequel les opérations sont appliquées sur le répliquat.

Ces conditions ne présument pas que les réplicats soit des objets actifs, i.e. des objets intégrants un ou plusieurs flots d'exécution propres. Elle suppose seulement l'indépendance des flots d'exécution accédant aux différents réplicats.

Ce modèle possède trois propriétés sympathiques :

- Une prise de décision souple, tantôt autonome, tantôt concertée entre les processus, de l'ordonnancement des opérations de l'exécution répartie. Ce modèle généralise la réplication active [PBS+88, Lit91] en permettant aux réplicats de ne pas se restreindre, a priori, à un ordre unique dans l'exécution des opérations sur la donnée répliquée. L'exacte liberté permise dépend bien sûr du critère de correction applicatif et sera étudiée au chapitre suivant.
- Une liberté dans la forme des mises à jour des réplicats échangées entre les processus. Une mise à jour peut être représentée sous forme d'un état, mais ce choix n'est pas inhérent au modèle de réplication, ce contrairement aux cas de la réplication passive [Lit91] dans laquelle un réplicat maître maintient un état de référence.
- La possibilité d'intégrer dans le contexte d'un même processus, le code accédant au réplicat, le code réalisant le réplicat et le code de gestion de la cohérence. Les interactions entre ces fonctionnalités peuvent ainsi être locales (mémoire partagée, appel de fonctions, ...) donc simples et efficaces.

Ce modèle adresse la partie performance du problème. D'une part, le problème de la latence induite par l'accès à la donnée partagée, ceci en répliquant la donnée partagée (effet de cache). D'autre part, le problème de la latence induite par les agréments imposés par la gestion de cohérence, ceci en ne fixant pas de contraintes, a priori, sur l'ordre des opérations.

[1] Les canaux cachés peuvent être dans un certaine mesure pris en compte. Ce point sera discuté plus avant.

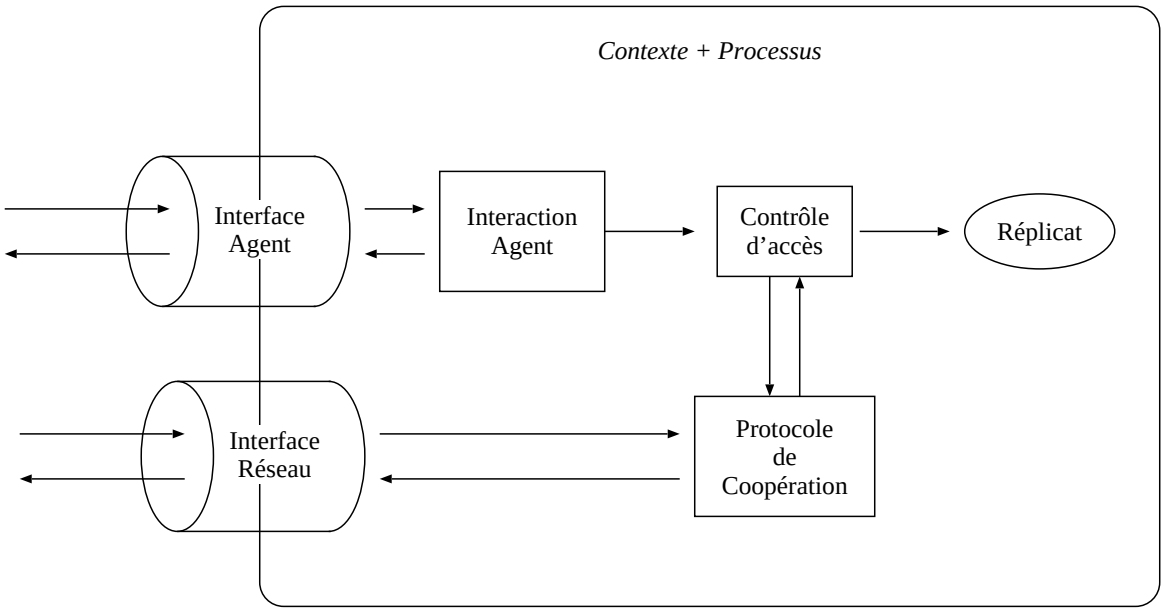


FIG. 6.2: Schéma d'un processus coopérant mettant en œuvre la réplication coopérante. Le réplicat de l'information partagée est un objet *passif* incarné dans le contexte du processus interagissant avec l'agent. L'« interface réseau » représente le média par lequel les processus coopèrent. Il peut s'agir aussi de disquettes ou même de fichiers partagés.

Le modèle possède deux contraintes essentielles : le déterminisme et le caractère total de la réplication. Nous pensons que la première contrainte est acceptable dans un cadre où les réplicats sont typiquement des objets. La deuxième contrainte peut être assouplie ; elle est posée de façon stricte pour simplifier l'étude du problème. L'étude de l'initialisation des réplicats (cf. section 6.3) intègre une petite discussion prospective sur les relâchements possibles de cette contrainte.

6.1.3 Modèle d'une application coopérative

Finalement, nous modélisons une application coopérative comme un groupe de processus formant un système réparti asynchrone. La coopération des agents est obtenue en permettant à chacun des processus « représentant » un agent d'accéder à et de modifier un réplicat de l'information partagée incarné dans son contexte.

Nous considérons que les réplicats sont implantés comme des objets passifs : l'accès à la représentation de l'information partagée est effectué par le processus exécutant le code applicatif.

Ce modèle est illustré par la figure 6.2. Le « contrôle d'accès », l'« interaction agent » et le « protocole de coopération » sont les abstractions identifiant trois tâches réalisées par le code applicatif exécuté par le processus représenté.

- l'« interaction agent » abstrait le code chargé de l'interaction avec l'agent,
- le « protocole de coopération » abstrait le code chargé de la coopération entre processus,
- le « contrôle d'accès » abstrait le code responsable des invocations sur le réplicat.

Deux points importants sont à noter, concernant ce modèle :

1. Le « contrôle d'accès » génère une invocation, en réponse à deux types d'événements : ceux en provenance du code traitant les événements en provenance de l'agent (la fonction « interaction agent »), ceux en provenance des autres processus coopérants (« la fonction protocole de coopération »). Nous parlerons d'événements *initiés* par l'agent pour désigner les événements issus de l'interaction avec l'agent.
2. Le « réplicat » abstrait l'état de l'ensemble des données partagées. L'exacte représentation de ces données est sans importance, dès lors que l'on ne s'intéresse qu'aux contraintes de cohérence. En particulier, le réplicat peut aussi bien représenter un scalaire, une liste de scalaires (mémoire) ou une collection d'objets complexes (graphe).

6.2 Modélisation et observation d'une exécution répartie

Nous venons de construire un modèle d'une application répartie. Notre souhait, maintenant, est de pouvoir exprimer l'ensemble des propriétés que les applications coopérantes peuvent requérir de leur exécution, ceci, sans contraintes additionnelles. Ce souhait s'inscrit dans une démarche qui consiste, tout d'abord, à permettre à l'application d'exprimer ses requis de correction, i.e. avec nos hypothèses, les contraintes de cohérence qu'elle impose à ses données, puis ensuite seulement, de « construire le système » qui lui assure ces propriétés, si possible à moindre coût.

Les propriétés que nous demandons au modèle d'exécution diffèrent sensiblement de celles que l'on trouve dans les modèles d'exécution usuels [Lam86, Mat89, BM93a, RS95b]. Ces derniers s'attachent avant tout à s'assurer de l'existence de certaines propriétés dans une exécution répartie, issue d'un système de Lamport (cf chapitre 2). Notre but est de construire le système répliqué qui assure les propriétés d'une application donnée.

Deux propriétés du modèle d'exécution, comme de son formalisme, reflètent les besoins de notre démarche :

1. ($\mathcal{P}_1$) La réplication doit être explicitement représentée. Cette propriété permet d'exprimer simplement les critères de cohérence basés sur la divergence de réplicat.
2. ($\mathcal{P}_2$) Le modèle ne doit pas spécifier de contraintes sur l'ordre dans lequel les opérations sont effectuées sur les différents réplicats. Cette liberté nous permet d'exprimer les requis de cohérence spécifiques de telle ou telle application. Dès lors qu'il s'agit d'ordre, l'application est totalement libre des contraintes qu'elle peut imposer à son exécution.

Nous présentons dans la section suivante un modèle d'exécution et un formalisme qui permettent de spécifier les propriétés pertinentes pour juger de l'adéquation de la gestion de cohérence d'application utilisant la réplication coopérante.

6.2.1 Le modèle d'exécution Core

Concentrons nous sur l'expression des propriétés de cohérence que l'on peut associer à l'information partagée. L'hypothèse déterministe nous permet de ne considérer, comme événements pertinents, que les seuls accès aux réplicats, i.e. les invocations effectuées par l'abstraction « contrôle d'accès » du code applicatif (cf. figure 6.2).

Notre modèle repose sur trois notions essentielles : l'*opération*, son *exécution* et son *initiation*.

Nous nommons *opération*, l'abstraction de la chaîne d'événements qui amène à la réalisation d'*un* accès particulier à l'information partagée. Une opération caractérise un accès particulier. Une opération détermine, en particulier, le type de cet accès et l'exacte partie de l'information partagée à laquelle accède cette opération.

Chaque opération est « répliquée », i.e. *tout* processus exécute une fois, et une seule, *chaque* opération de l'exécution. Une opération ne désigne *pas* l'accès particulier à un réplicat : dans un système de N processus (et donc N réplicats), une opération est « instancié » N fois, faisant l'objet de N accès particuliers, un sur chaque réplicat distinct. Chaque processus *exécute* donc le même ensemble d'*opérations*.

Parmi les opérations exécutées par un processus, nous en distinguons certaines, celles qui résultent d'une interaction de l'agent avec ce processus, i.e. *son* processus. Nous disons que ces opérations sont *initiées* par le processus associé à cet agent. Une opération est initiée par un et un seul processus.

Les autres opérations résultent aussi d'accès effectués par ce processus, mais dans le cadre du protocole de coopération entre processus. Ces dernières opérations ont, d'une façon ou d'une autre, été initié par les autres processus coopérants.

Une exécution d'un processus est donc complètement définie par trois éléments :

- l'ensemble des opérations qu'il a exécutées,
- l'ordre dans lequel il les a exécutées,
- l'ensemble des opérations qu'il a initiées (sous-ensemble du précédent).

Les deux premiers éléments définissent ce que l'on appelle traditionnellement l'histoire du processus. Les processus exécutant séquentiellement leurs opérations, l'ordre d'exécution est un ordre total strict (irréflexif et transitif) sur l'ensemble des opérations exécutées.

La modélisation de la coopération nécessite deux notions supplémentaires : la donnée partagée, et la partie de cette donnée accédée par une opération. Nous représentons la donnée partagée par un ensemble de parties d'ensemble. Nous associons à toute opération une partie de cet ensemble, que nous désignons par objet[2].

Ce modèle possède clairement les deux propriétés souhaitées : la réplication est explicite et aucun a priori n'est porté sur l'ordre d'exécution.

En particulier, ce modèle d'exécution sépare clairement les *mécanismes* par lesquels une opération émise par un processus en vient à figurer (et à être exécutée) dans l'histoire d'un autre processus, des *propriétés* induites par l'ordonnancement de ces opérations : les mécanismes ne sont pas représentés ; seules sont modélisées les propriétés de cohérence représentées par les ordonnancements potentiellement variés des opérations dans l'histoire de chacun des processus. En fait, nous avons présenté, non un modèle du système, mais bien un modèle de l'exécution répartie répliquée d'une application coopérative. C'est parce que nous ne modélisons pas le système, i.e. les mécanismes mettant en œuvre la répartition, que nous pouvons parler de la correction de l'exécution indépendamment des aspects de la mise en œuvre de la gestion de cohérence, celle-ci dépendant elle-même du système[3].

[2]Ce terme a déjà utilisé pour désigner une instance de type de donnée abstrait. La confusion n'est pas gênante. Il s'agit, essentiellement, de deux visions d'une même entité, une vue applicative et une vue « système ».

[3]En particulier, le caractère pessimiste ou optimiste est un trait de mise en œuvre et ne peut même pas être exprimé dans le modèle **Core**.

N		Nombre de processus coopérants.
$\mathcal{I}_i$		Événements initiés par le processus p_i.
Obset		Ensemble des objets auxquels accède le système.
E	$= \bigcup_N \mathcal{I}_i = \bigoplus_N \mathcal{I}_i$	Événements du système.
$(\mathcal{I}_i, <_i)$		Histoire de l'agent associé aux processus p_i.
$\mathcal{H}_i$	$= (E, \prec_i)$	Histoire du processus p_i.
$\mathcal{H}$	$=(\mathcal{H}_i)_{i\in[1\cdots N]} = (\mathcal{H}_1, \ldots, \mathcal{H}_N)$	Histoire du système.

Tous les processus exécutent le même ensemble E d'opérations.

FIG. 6.3: Résumé des notations les plus utilisées.

6.2.2 Le formalisme Core

Nous introduisant dans cette section, les éléments formels du modèle d'exécution et les notations associées.

Nous notons $S = (p_1, \ldots, p_n)$, la collection de processus formant le système réparti asynchrone. Le système S modélise l'application coopérative.

Les notations suivantes sont utilisées dans le reste de ce document :

- Concernant l'exécution d'un processus p_i :
  - E_i désigne l'ensemble des opérations exécutées par le processus p_i.
  - $\mathcal{I}_i$, désigne les opérations initiées par le processus p_i.
  - $\prec_i$, identifie l'ordre strict, total sur E_i, dans lequel le processus p_i a exécuté ses opérations.
  - $<_i$ désigne la restriction de $\prec_i$ sur $\mathcal{I}_i$. Cet ordre est celui dans lequel le processus p_i a *initié* ses opérations.
  - $\mathcal{H}_i = (E_i, \prec_i)$ désigne la séquence des opérations exécutée par le processus p_i, traditionnellement appelée « histoire du processus p_i ».
  - $(\emptyset, \prec_i)$, i.e. l'histoire vide, désigne l'état initial du processus.
- Concernant la donnée partagée :
  - Obset désigne l'information partagée dans son ensemble.
  - O_{op} désigne l'objet accédé par l'opération op.
  - $\text{Op}(O)$, désigne l'ensemble des opérations ayant accédé à l'objet O.
  - $\mathcal{H}_i(O)$ désigne la restriction de l'histoire $\mathcal{H}_i$ aux opérations ayant accédé à l'objet O, ce que l'on désigne aussi couramment comme la projection de l'histoire $\mathcal{H}_i$ sur l'objet O.
  - De façon générale, pour tout ensemble d'opérations Op et $\text{Op}_1 \subseteq \text{Op}$, pour toute relation $<$ définie sur Op, nous noterons $<\!/_{\text{Op}_1}$ la restriction de $<$ aux opérations de Op_1. Par définition $<\!/_{\text{Op}_1} \subseteq <$.

Nous représentons une exécution du système S par le N-uplet $(\mathcal{H}_i)_{i\in[1\cdots N]}$, d'histoires locales $(\mathcal{H}_i = (E, \prec_i))$.

6.2.2.1 Le formalisme et la réplication coopérante

Une exécution du système réparti S respecte les principes de la réplication coopérante si les quatre conditions suivantes sont remplies :

1. Les processus sont déterministes.
2. Les processus ont le même état initial. Nous notons $((\emptyset, \prec_i))_{i\in[1\cdots N]}$ cet état initial.
3. Les processus exécutent le même ensemble fini E d'opérations, i.e. tous les E_i sont égaux et chaque opération est exécutée N fois, une fois sur chaque processus.
4. Les $\mathcal{I}_i$ constituent une partition de E.

Les quelques remarques suivantes aident à la compréhension tant du formalisme que du modèle d'exécution :

- $E = \bigcup_N \mathcal{I}_i = \bigoplus_{i\in[1\cdots N]} \mathcal{I}_i)$.
- $\mathcal{I}_i$ est un sous-ensemble de E_i.
- $<_i$ est un ordre total sur $\mathcal{I}_i$.
- Deux opérations x et y accèdent à deux objets distincts si $O_x \cap O_y = \emptyset$.
- Le cas $<_i = \prec_i$ correspond au cas où seul un processus (p_i) à initié des opérations $(\forall j \neq i, \mathcal{I}_j = \emptyset)$.
- Chaque processus exécute séquentiellement ces opérations. L'exécution *locale* d'une opération est donc atomique.
- Bien que l'ensemble E soit commun à tous les processus, les relations $\prec_i$ de chaque processus peuvent différer.
- Le caractère déterministe des processus permet d'assimiler l'état d'un réplicat à l'histoire du processus auquel est attaché ce réplicat. Ce modèle rend donc compte de façon équivalente de la divergence de l'état des réplicats et de la divergence des histoires locales de chacun des processus.

6.2.2.2 Quelques définitions utiles

Nous définissons la relation « read-from » et la propriété de légalité qui sont toutes deux utilisées par la suite. Ces deux définitions ne s'appliquent qu'à des exécutions dont l'ensemble E des opérations exécutées peut être partitionné en un ensemble $\mathcal{W}$ d'opérations d'écriture et un ensemble $\mathcal{R}$ d'opérations de lecture. Nous supposons aussi que l'information partagée est constituée d'une collection de scalaires et que chaque opération n'accède qu'à un seul de ces scalaires, i.e. $O_x \cap O_y \neq \emptyset \iff O_x = O_y$.

Définition 6.1 (Légalité d'une lecture) *Soit une lecture x sur un objet a ($O_x = a$) initiée par un processus p_i, et y la dernière écriture sur cet objet exécutée sur p_i relativement à $\prec_i$ telle que la valeur retournée par x soit celle écrite par y. x est légale sur le processus p_i, si $\nexists z \in \mathcal{W}, O_z = a$, tel que $y \prec_i z$ et $z \prec_i x$.*

Définition 6.2 (Légalité d'une exécution) *Une exécution est légale si les lectures de chacun des processus de cette exécution sont légales.*

Définition 6.3 (Relation « read-from » ($<_{\mathbf{rf}}$)) *Une opération x, lit une valeur écrite par y, notée $y <_{rf} x$, si et seulement si les deux conditions suivantes sont satisfaites :*

- *$x \in \mathcal{R}$, $y \in \mathcal{W}$ et $\exists a \in Obset$ tel que $O_x = O_y = a$.*
- *sur le processus p_i tel que $x \in \mathcal{I}_i$, $y \prec_i x$ et x est légale.*

6.2.3 Observation d'une exécution répartie

Dans notre modèle, un état intermédiaire du système réparti est représenté par un tuple d'histoires locales. On ne peut donc se baser sur la définition des coupes présentées au chapitre 2 pour représenter un état intermédiaire du système. Nous devons donc redéfinir ce qu'est une coupe.

6.2.3.1 Coupes d'une exécution

Définition 6.4 (Préfixe) *Soit $\mathcal{H}_i = (E_i, \prec_i)$ une histoire locale et une opération $\tilde{e}$ telle que $\tilde{e} \in E_i$. On nomme préfixe de $\tilde{e}$ sur $\mathcal{H}_i$, que l'on notera $P(\mathcal{H}_i)_{\tilde{e}}$, la séquence des opérations précédant,* en l'excluant, *$\tilde{e}$ dans $\mathcal{H}_i$.*

$P(\mathcal{H}_i)_{\tilde{e}}$ est la séquence ordonnée $(A, \prec_i)$ telle que $A \subseteq E_i$ et $\tilde{e}$ est borne supérieure de A relativement à $\prec_i$. $\prec_i$ étant stricte, $\tilde{e}$ ne fait pas partie du préfixe.

Définition 6.5 (Préfixe fermé) *Soit $\mathcal{H}_i = (E_i, \prec_i)$ une histoire locale et une opération $\tilde{e}$ telle que $\tilde{e} \in E_i$. On nomme préfixe fermé de $\tilde{e}$ sur $\mathcal{H}_i$, que l'on notera $P_{fermé}(\mathcal{H}_i)_{\tilde{e}}$, la séquence des opérations précédant,* en incluant, *$\tilde{e}$ dans $\mathcal{H}_i$.*

Soit $P(\mathcal{H}_i)_{\tilde{e}} = (A, \prec_i)$, alors $P_{fermé}(\mathcal{H}_i)_{\tilde{e}} = (A \cup \{\tilde{e}\}, \prec_i)$.

Définition 6.6 (Coupe) *Soit une exécution $\mathcal{H} = (\mathcal{H}_i)_{i\in[1\cdots N]}$ et un N-uplet $(c_i)_{i\in[1\cdots N]}$ d'opérations de cette exécution, i.e. tel que $\forall i, c_i \in E$.*

Un N-uplet $(\mathcal{H}'_i)_{i\in[1\cdots N]}$ est une coupe $(c_i)_{i\in[1\cdots N]}$, notée $C_{(c_i)_{i\in[1\cdots N]}}$, de l'exécution $\mathcal{H}$, si et seulement si : $\forall i \in [1\cdots N], \mathcal{H}'_i = P_{fermé}(\mathcal{H}_i)_{c_i}$.

Les éléments de coupe sont inclus dans la coupe. Nous notons $\mathcal{COUPE}(\mathcal{H})$, l'ensemble des coupes d'une exécution $\mathcal{H}$. Le N-uplet $((\emptyset, \prec_i))_{i\in[1\cdots N]}$ désigne l'état initial du système.

Certaines propriétés requises sur une exécution complète ne le sont pas sur les coupes. Sur une coupe, l'ensemble des opérations exécutées par chacun des processus peut différer. Lorsque cela sera important, nous indexerons supérieurement par le nom de la coupe (C, par exemple) la restriction aux éléments présents dans la coupe, des ensembles et des relations d'ordre définis dans le modèle ($<_i^C$, $\prec_i^C$, $\mathcal{I}_i^C$, etc.). Nous noterons, par exemple, E_i^C, l'ensemble des opérations exécutées sur une coupe C par un processus p_i (en particulier, $\prec_i^C$ est un ordre total strict sur E_i^C).

Définition 6.7 (inf (resp. sup) de deux coupes) *Soit une exécution $((E, \prec_i))_{i\in[1\cdots N]}$ et deux couples de coupes $C = ((E^C, \prec_i))_{i\in[1\cdots N]}$, $C' = ((E^{C'}, \prec_i))_{i\in[1\cdots N]}$ de cette exécution. Nous définissons* $\inf(C, C')$ *(resp.* $\sup(C, C')$*) comme le N-uplet $C'' = ((A_i, \ll_i))_{i\in[1\cdots N]}$ tel que $\forall i \in [1\cdots N], A_i = E_i^C \cap E_i^{C'}$ (resp. $\forall i \in [1\cdots N], A_i = E_i^C \cup E_i^{C'}$), $\ll_i \subseteq \prec_i$ et $\ll_i$ est un ordre total strict sur A_i.*

Définition 6.8 (Précédence de coupe ($\sqsubseteq$)) *Une coupe C' précède une coupe C, que nous notons $C' \sqsubseteq C$, si et seulement si $C' = \inf(C, C')$.*

Nous donnons, sans les démontrer, les propriétés suivantes : l'inf comme le sup de deux coupes d'une même exécution $\mathcal{H}$ sont toujours définis, sont uniques et constituent chacun une coupe de cette exécution $\mathcal{H}$. $\sqsubseteq$ est une relation d'ordre sur $\mathcal{COUPE}(\mathcal{H})$ et pour toute paire de coupes $\{C, C'\}$ d'une même exécution (i.e. $C, C' \in \mathcal{COUPE}(\mathcal{H})$), $\inf(C, C')$ (resp. $\sup(C, C')$) définit une borne inférieure (resp. supérieure) de $\{C, C'\}$ dans $\mathcal{COUPE}(\mathcal{H})$.

Ces propriétés suffisent à établir le corollaire suivant :

Théorème 6.1 *$(\mathcal{COUPE}(\mathcal{H}), \sqsubseteq, \inf, \sup)$ est un treillis.*

Preuve Toute partie $\{C, C'\}$ de $\mathcal{COUPE}(\mathcal{H})$ admet une borne inférieure, $\inf(C, C')$, et une borne supérieure, $\sup(C, C')$. □

6.2.3.2 Observation de quels états ?

Les coupes permettent de collecter des ensembles *arbitraires* de vues locales du système. Dès lors que l'on cherche à statuer de propriétés sur un système, il est sensé de se restreindre à l'observation des exécutions *possibles* du système. Dans le modèle de Lamport, les coupes causalement cohérentes [RM93, Mat89, BM93b] garantissent la prise de vues de toutes les exécutions possibles du système, et uniquement de celles-ci. Brièvement chacune de ces coupes assure, pour tout événement y figurant, la présence dans cette même coupe de ses prédécesseurs, au sens de la relation « happened before ». Le fait que seules les coupes causalement cohérentes définissent un état possible du système et que tout état possible du système soit causalement cohérent est une propriété du modèle de Lamport, en particulier de ce que l'envoi d'un message précède au sens de « happened before » sa réception.

Notre modèle ne possède pas cette propriété. Intuitivement, une telle propriété ne peut être assurée que si le modèle spécifie *comment* un événement initié par un processus parvient à la connaissance des autres. Or, c'est précisément cette spécification que nous avons exclue de notre modèle.

Nous nous proposons de redéfinir une relation similaire dans le principe à celle de « happened before », ne reposant non pas sur le respect de la causalité du couple `envoi/réception` d'un message mais sur celui du couple `perception/initiation` d'une opération ; autrement dit une relation fondée sur le respect de la causalité entre une observation que peut faire un agent (d'un objet) et les actions qu'il effectue ultérieurement.

La définition de cette relation que nous dénommerons « initiée avant » et noterons $<_{\mathcal{IA}}$, nécessite quelques définitions préalables :

Définition 6.9 *Nous associons à toute opération d'une exécution, la propriété d'être ou non une observation. Nous notons $\mathcal{O}_i$, l'ensemble des observations du processus p_i, i.e. les opérations* initiées *par p_i ayant la propriété d'observation ($\mathcal{O}_i \subseteq \mathcal{I}_i$).*

Nous disons qu'une observation o_i observe un objet O_k si et seulement si $o_i \in Op(O_k)$.

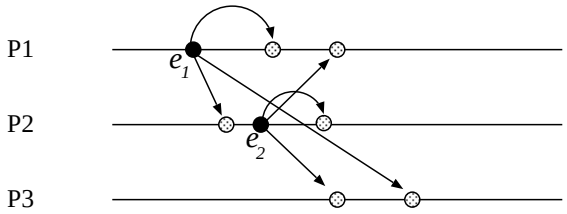
Définition 6.10 *Nous associons à toute opération d'une exécution, la propriété d'être ou non une action. Nous notons $\mathcal{A}_i$, l'ensemble des actions du processus p_i, i.e. les opérations* initiées *par p_i ayant la propriété d'action ($\mathcal{A}_i \subseteq \mathcal{I}_i$).*

Nous disons qu'un action a_i agit sur un objet O_k si et seulement si $a_i \in Op(O_k)$.

Les actions ainsi définies ne présument pas d'une modification quelconque de l'état de l'objet, laissant ainsi la possibilité de modéliser les canaux cachés du système. La propriété d'action caractérise la possibilité qu'a une opération de déterminer l'initiation d'une autre opération. Une opération peut être simultanément une action et une observation.

Définition 6.11 ($<_{\mathcal{IA}}$) *Une opération x précède une opération y au sens de $<_{\mathcal{IA}}$, si et seulement si :*

- *y est une observation sur un objet, x est une action sur ce même objet et x a été exécutée avant y par le processus qui a initié y, soit $\exists p_i, p_j \in S, \exists O_k \in Obset$ tels que $x \in \mathcal{A}_j, y \in \mathcal{O}_i, x \in Op(O_k), y \in Op(O_k)$ et $x \prec_i y$, **ou que**,*
- *x est une observation, y une action, toutes deux initiées par le même processus et x a été exécutée avant y sur ce processus, soit $\exists p_i \in S$ tel que $x \in \mathcal{O}_i, y \in \mathcal{A}_i$ et $x \prec_i y$.*



$$\mathcal{I}_1 = \{e_1\}, \mathcal{I}_2 = \{e_2\}, \mathcal{I}_3 = \emptyset, E = \{e_1, e_2\}$$

$$\left.\begin{array}{l} \mathcal{H}_1 = e_1 \bullet e_2 \\ \mathcal{H}_2 = e_1 \bullet e_2 \\ \mathcal{H}_3 = e_2 \bullet e_1 \end{array}\right) \mathcal{H} = (\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_3)$$

$$\left.\begin{array}{l} e_2 \in \mathcal{I}_2, e_1 \prec_2 e_2 \Rightarrow e_1 <_{\mathcal{IA}_c} e_2 \\ e_2 \prec_3 e_1 \Rightarrow <_{\mathcal{IA}_c} \not\subseteq \prec_3 \end{array}\right\} \Rightarrow \mathcal{H} \notin <_{\mathcal{IA}_c}$$

FIG. 6.4: Notre modèle ne distingue pas l'émission (point noir) d'une opération de ses réceptions (points grisés). e_1 et e_2 sont des diffusions (opérations répliquées). L'exécution $\mathcal{H}$ n'est pas $<_{\mathcal{IA}_c}$ cohérente, en accord avec le modèle de LAMPORT étendu aux diffusions (multicast) représentées sur la partie gauche de la figure.

Nous ne construisons pas, a priori, de dépendance causale entre deux observations, une telle dépendance s'exprime simplement en considérant aussi comme action de telles observations[4]. En général, deux observations consécutives sur le même objet d'un même processus sont causalement liées ; mais le respect de leur ordre chez les processus distincts n'a d'intérêt que si elles peuvent avoir un effet chez ceux-ci, c'est-à-dire déterminer l'initiation ou l'effet d'une opération.

Cette relation repose sur la connaissance des propriétés *d'observation* et *d'action* des opérations, des objets qu'elles accèdent et du processus sur lequel elles ont été initiées. Cette connaissance pouvant être complexe à mettre en œuvre, nous définissons une autre relation de causalité, plus conservative, qui considère que chaque opération initiée par un processus observe et agit simultanément sur l'ensemble de l'information répliquée, Obset.

Définition 6.12 ($<_{\mathcal{IA}}$ conservative ($<_{\mathcal{IA}_c}$)) *Une opération x précède une opération y au sens de $<_{\mathcal{IA}_c}$, si et seulement si : $\exists p_i \in S$ tel que, $x \in E, y \in \mathcal{I}_i$ et $x \prec_i y$.*

Théorème 6.2 *Si x précède causalement y relativement à $<_{\mathcal{IA}}$, il le précède aussi relativement à $<_{\mathcal{IA}_c}$, soit $<_{\mathcal{IA}} \subseteq <_{\mathcal{IA}_c}$.*

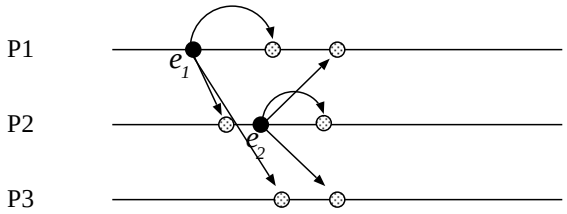
Preuve x $<_{\mathcal{IA}}$y implique $x \in E, x \prec_i y$ et soit $y \in \mathcal{O}_i$ soit $y \in \mathcal{A}_i$. Or $\mathcal{O}_i \subseteq \mathcal{I}_i, \mathcal{A}_i \subseteq \mathcal{I}_i$; donc $x \in E, y \in \mathcal{I}_i$ et $x \prec_i y$ □

Si $\mathcal{O}_i = \mathcal{A}_i = \mathcal{I}_i$, |Obset| = 1 alors $<_{\mathcal{IA}} = <_{\mathcal{IA}_c}$.

6.2.3.3 Formulation de propriétés usuelles

Nous avons, maintenant, les éléments permettant de définir quelques coupes intéressantes. Nous insistons sur le fait que ces coupes ne forment pas, en elles-mêmes, un critère de correction. En l'absence d'une modélisation des mécanismes de perception des événements répartis, on ne peut modéliser (en asynchrone) les coupes d'exécutions intermédiaires possibles, qui ont effectivement pu avoir lieu ; tout au plus peut-on parler de vues globales plausibles. Ces coupes peuvent, par contre, servir de support à des critères de correction adaptés à nos applications cibles.

[4]En particulier, de telles observations induisent alors une dépendance causale avec les observations effectuées par les autres processus.



$$\mathcal{I}_1 = \{e_1\}, \mathcal{I}_2 = \{e_2\}, \mathcal{I}_3 = \emptyset, E = \{e_1, e_2\}$$

$$\left.\begin{array}{l} \mathcal{H}_1 = e_1 \bullet e_2 \\ \mathcal{H}_2 = e_1 \bullet e_2 \\ \mathcal{H}_3 = e_1 \bullet e_2 \end{array}\right) \mathcal{H} = (\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_3)$$

$$\left.\begin{array}{l} e_2 \in \mathcal{I}_2, e_1 \prec_2 e_2 \Rightarrow e_1 <_{\mathcal{IA}_c} e_2 \\ \forall i, <_{\mathcal{IA}_c} \subseteq \prec_i \end{array}\right\} \Rightarrow \mathcal{H} \in <_{\mathcal{IA}_c}$$

FIG. 6.5: Représentation d'une exécution causale dans le modèle de LAMPORT et dans le nôtre.

Définition 6.13 (Coupe FIFO cohérente) *Une coupe C est FIFO cohérente si et seulement si : $\forall i \in [1 \cdots N], \forall j \in [1 \cdots N]$, $\prec_j^C /_{\mathcal{I}_i^C} = <_i^C$. (i.e. si les opérations sont exécutées sur tous les sites dans l'ordre dans lequel elles on été initiées)*

Autrement dit, une coupe est FIFO si les opérations initiées par un processus sont exécutées dans ce même ordre sur tous les autres processus.

Les deux définitions suivantes introduisent la causalité, dans le cadre du modèle d'exécution **Core**.

Définition 6.14 (Coupe $<_{\mathcal{IA}}$ cohérente) *Une coupe C est $<_{\mathcal{IA}}$ cohérente si $\forall i \in [1 \cdots N]$, $<_{\mathcal{IA}} /_{E_i^C} \subseteq \prec_i^C$.*

Toute coupe $<_{\mathcal{IA}}$ cohérente assure la causalité traditionnelle : si une action à été initiée au vu (observation) d'une séquence d'opérations, une coupe $<_{\mathcal{IA}}$ cohérente assure que l'exécution des opérations figurant dans cette séquence précède l'exécution de cette action sur tous les processus, mais pas forcement dans le même ordre[5].

Définition 6.15 (Coupe $<_{\mathcal{IA}_c}$ cohérente) *Une coupe C est $<_{\mathcal{IA}_c}$ cohérente si $\forall i \in [1 \cdots N]$, $<_{\mathcal{IA}_c} /_{E_i^C} \subseteq \prec_i^C$.*

Cette définition garantit que toutes les opérations qui, sur un site, précèdent une opération $\tilde{op}$ initiée sur *ce* site, précèdent aussi cette opération $\tilde{op}$ sur tous les sites.

Théorème 6.3 *Toute coupe $<_{\mathcal{IA}}$ cohérente est $<_{\mathcal{IA}_c}$ cohérente.*

Preuve $<_{\mathcal{IA}} \subseteq <_{\mathcal{IA}_c}$. □

Les coupes $<_{\mathcal{IA}_c}$ cohérentes sont plus conservatives que les coupes $<_{\mathcal{IA}}$ cohérentes mais ne nécessitent pas de distinguer parmi les opérations initiées par un processus celles qui concernent un même objet. Ni de distinguer celles qui sont des observations de celles qui sont des actions. Elles sont donc plus simples à mettre en œuvre.

Les figures 6.4 et 6.5 comparent, sur un exemple répliqué très simple (système de 3 processus), le modèle traditionnel à communication par messages et notre modèle répliqué. La partie gauche de chaque figure représente la représentation de l'exécution sous la forme du

[5]Ceci peut apparaître contre-intuitif si l'on considère que l'on observe un *état*, et donc que l'ordre des prédécesseurs a son importance. Mais conserver l'ordre des prédécesseurs imposerait d'ordonner tous les événements, même concurrents, (potentiellement) prédécesseurs d'une action. Dans un cadre pessimiste, ceci impose d'assurer un ordre global total. En pratique, lorsque un agent ne repose que sur la causalité, il sait que les événements concurrents peuvent être perçus dans un ordre différent sur d'autres sites et son action dépend de cette connaissance *aussi* !

diagramme espace–temps traditionnel, la partie droite représente la même exécution dans le formalisme **Core**. Une opération répliquée dans le modèle **Core** abstrait les quatre événements suivant représentés dans le diagramme espace–temps : la diffusion d'un message (1) et les réceptions de ce message (3). Dans un modèle de communication « point à point », la diffusion d'un message serait elle-même représenté par 3 événements distincts.

Les coupes causales et les coupes FIFO sont toutes construites sur le respect d'un ordre sur des opérations. Ces familles de coupes formant un treillis, elles peuvent être utilisées pour construire une progression croissante de l'exécution répartie, simulant ainsi, un temps virtuel global [Mat93].

6.2.3.4 Formulation de la divergence – Coupes de borne

Nous avons vu, dans la première partie de cette thèse (cf section 3.2.2), que l'existence d'une métrique pouvait être utilisée avec bonheur dans un critère de correction. Dans un cadre répliqué, une telle métrique permet de définir une divergence entre deux états appartenant à des réplicats distincts [PL91, WYP97].

L'asynchronisme pose néanmoins un problème délicat : quels états est-il sensé de comparer ? Dans un système synchrone, la comparaison se ferait au même « moment », mais dans un système asynchrone ?

Dans le modèle d'exécution **Core**, le déterminisme permet d'assimiler la divergence entre deux états à celle des deux histoires qui ont mené à ces deux états.[6] Ainsi, dans le cadre de la réplication coopérante, contrôler la divergence de deux réplicats, c'est contrôler la liberté que leur processus respectifs prennent sur l'ordre d'exécution des opérations.

Nous proposons de modéliser cette liberté en se basant sur le principe suivant : considérons un processus p_i donné, au moment où il *initie* une opération. Une bonne mesure de sa liberté consiste à considérer, à ce moment là, les actions déjà exécutées par les autres processus mais que lui p_i n'a pas encore perçues. Les coupes de borne formalisent se principe.

Définition 6.16 (Coupe de borne) *Soit $c_i \in I_i$ une opération initiée par le processus p_i. Nous disons que C^{c_i} est une coupe de borne de c_i si $C^{c_i} = (P_{fermé}(\mathcal{H}_j)_{c_i})_{j\in[1\cdots N]}$.*

La coupe de borne associée à une opération c_i initiée sur le processus p_i, assure une prise de vue globale au « moment » où cette opération a été exécutée par chacun des autres processus. Une telle coupe permet de collecter les opérations initiées par les autres processus et non connues par p_i au moment où il a initié c_i.

Définition 6.17 (Delta de coupe) *Soit une coupe de borne C^{c_i}. Nous appelons delta de la coupe C^{c_i}, que nous notons $\Delta_{C^{c_i}}$, le $(N-1)$-uplet défini par : $\Delta_{C^{c_i}} = (\mathcal{I}_j^{C^{c_i}} \backslash E_i^{C^{c_i}})_{j\in[1\cdots N], j\neq i}$, où chaque $\mathcal{I}_j^{C^{c_i}} \backslash E_i^{C^{c_i}}$ représente l'ensemble des opérations initiées par p_j non encore perçues par p_i au moment où p_i a initié son opération c_i.*

En un sens, ce $(N-1)$-uplet représente le degré de liberté pris par le processus p_i. Nous montrons, en fin de chapitre, sur des exemples, comment, avec l'aide d'une métrique, on peut utiliser ces outils pour contrôler la divergence des processus. En général, les critères de divergence utilisés sont « aveugles » quant aux opérations exécutées ; seule compte la distance sur les états induits par ces opérations, et non quelle opération particulière dans son type ou ses arguments a engendré cette distance. Les coupes de borne ne se substituent pas, dans le cas général, aux autres coupes basées sur des propriétés d'ordre entre opérations ; elles sont

[6] Il suffit de dériver de la métrique sur l'état, une métrique « équivalente » sur les histoires.

$$\begin{array}{lll} \mathcal{I}_1 = \{r_1^1, w_1^2, r_1^3, w_1^4\}, & \mathcal{O}_1 = \{r_1^1, r_1^3\}, & \mathcal{A}_1 = \{w_1^2, w_1^4\} \\ \mathcal{I}_2 = \{r_2^1, w_2^2\}, & \mathcal{O}_2 = \{r_2^1\}, & \mathcal{A}_2 = \{w_2^2\} \\ \mathcal{I}_3 = \emptyset, & \mathcal{O}_3 = \emptyset, & \mathcal{A}_3 = \emptyset \end{array}$$
$$E = \{r_1^1, w_1^2, r_1^3, w_1^4, r_2^1, w_2^2\}$$

$$\begin{array}{l} \mathcal{H}_1 = r_1^1 \bullet w_1^2 \bullet r_2^1 \bullet w_2^2 \bullet r_1^3 \bullet w_1^4 \\ \mathcal{H}_2 = r_1^1 \bullet w_1^2 \bullet r_2^1 \bullet w_2^2 \bullet r_1^3 \bullet w_1^4 \\ \mathcal{H}_3 = \mathcal{H}_2 \end{array} \qquad \begin{array}{l} \mathcal{H}''_1 = r_1^1 \bullet w_1^2 \bullet r_2^1 \bullet r_1^3 \bullet w_1^4 \bullet w_2^2 \\ \mathcal{H}''_2 = r_1^1 \bullet w_1^2 \bullet r_2^1 \bullet w_2^2 \bullet r_1^3 \bullet w_1^4 \\ \mathcal{H}''_3 = \mathcal{H}''_2 \end{array}$$

$$\begin{array}{l} \mathcal{H}'_1 = r_1^1 \bullet w_1^2 \bullet r_1^3 \bullet r_2^1 \bullet w_2^2 \bullet w_1^4 \\ \mathcal{H}'_2 = r_1^1 \bullet w_1^2 \bullet r_2^1 \bullet r_1^3 \bullet w_2^2 \bullet w_1^4 \\ \mathcal{H}'_3 = \mathcal{H}'_2 \end{array} \qquad \begin{array}{l} \mathcal{H}'''_1 = r_1^1 \bullet w_1^2 \bullet r_2^1 \bullet w_2^2 \bullet r_1^3 \bullet w_1^4 \\ \mathcal{H}'''_2 = r_1^1 \bullet w_1^2 \bullet r_2^1 \bullet w_2^2 \bullet r_1^3 \bullet w_1^4 \\ \mathcal{H}'''_3 = r_2^1 \bullet w_2^2 \bullet r_1^1 \bullet w_1^2 \bullet r_1^3 \bullet w_1^4 \end{array}$$

FIG. 6.6: Système à 3 réplicats et un seul objet répliqué. Le premier processus initie une séquence de deux `lecture-écriture`. Le second processus n'en émet qu'une. Le troisième processus est passif et n'initie aucune opération.
$(\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_3)$ est atomique, causale et séquentielle (« ww-constraint »). $(\mathcal{H}'_1, \mathcal{H}'_2, \mathcal{H}'_3)$ est causale et séquentielle. $(\mathcal{H}''_1, \mathcal{H}''_2, \mathcal{H}''_3)$ est causale mais non séquentielle (w_2^2 et w_1^4 sont ordonnées différemment sur $\mathcal{H}''_1$ et $\mathcal{H}''_2$). $(\mathcal{H}'''_1, \mathcal{H}'''_2, \mathcal{H}'''_3)$ n'est ni atomique ni causale ni séquentielle car, entre autre, $w_1^2 <_{\mathcal{IA}} w_2^2$ et $w_2^2 \prec_3 w_1^2$.

---

incomparables avec les coupes FIFO ou causales. Il existe une exception notable, celui des exécutions séquentielles : sur une exécution séquentielle, toute coupe de borne est FIFO et causale. Cette exception est attendue dans la mesure où ce critère assure une convergence stricte des exécutions réparties.

6.2.4 Correction d'un système répliqué

Les deux hypothèses sur la réplication totale et le déterminisme, énoncées en tout début de chapitre, nous permettent de représenter tout critère de correction comme un ensemble de prédicats globaux[7] sur certaines coupes de l'histoire du système.

Définition 6.18 (Exécutions correctes) *Appelons critère de correction élémentaire, la donnée d'un prédicat et de l'ensemble des coupes de l'exécution sur lesquelles il doit être satisfait. Un critère de correction (définition récursive) est soit un critère de correction élémentaire soit la conjonction ou la disjonction de critères de correction. Nous noterons $\mathcal{C}$ le critère de correction de l'application. Nous noterons $\mathcal{CORRECT}$, l'ensemble des exécutions du système satisfaisant $\mathcal{C}$.*

Par souci de simplification, nous noterons $\mathcal{COUPE}(\{\mathcal{H}\})$ pour $\bigcup_{\{\mathcal{H}\}} \mathcal{COUPE}(H)$, i.e. l'union des coupes de chaque exécution appartenant à $\{\mathcal{H}\}$. Ainsi $\mathcal{COUPE}(\mathcal{CORRECT})$ désigne l'union de toute les coupes d'exécutions correctes. Nous considérons que la coupe initiale $((\emptyset, \prec_i))_{i \in [1 \cdots N]}$ est toujours correcte.

Nous donnons ci-dessous, des exemples d'expression de quelques modèles de cohérence, dont les plus simples sont illustrés sur la figure 6.6 :

[7] i.e.. intégrant au moins une conjonction de prédicat sur des histoires appartenant à des processus distincts.

Cohérence atomique : Une gestion de cohérence assure une cohérence atomique si pour toute exécution du système, toute les relations d'ordre sur les exécutions locales sont identiques ($\mathcal{C}$, critère de correction, $= \forall i, j \in [1 \cdots N], \prec_i = \prec_j$). Autrement dit il existe un ordre total unique sur l'ensemble des opérations.

Cohérence causale : Une gestion de cohérence assure une cohérence causale si toute exécution du système est une coupe $<_{\mathcal{IA}}$ cohérente. Cette cohérence est celle assurée par la mémoire causale [AHJ91]. La cohérence causale telle que définie par MATTERN dans [Mat89] est obtenue en remplaçant la relation $<_{\mathcal{IA}}$ par la relation $<_{\mathcal{IA}_c}$. On notera que dans les deux cas, la causalité est définie sur des critères applicatifs.

Cohérence séquentielle : un multiprocesseur est dit séquentiellement cohérent si [Lam79] :

> *« Le résultat de toute exécution est le même que si l'ensemble des opérations exécutées par les processeurs avaient été exécuté dans un certain ordre séquentiel, que chaque processeur a exécuté ses opérations dans cet ordre, en respectant l'ordre dans lequel ces opérations apparaissent dans le programme. »*
> (traduction libre)

La cohérence séquentielle est le critère de correction de référence s'agissant de mémoires partagées par des multiprocesseurs. Essentiellement pour deux raisons :

1. Elle permet de dériver un calcul réparti correct à partir d'un calcul centralisé. Or nombre d'algorithmes se construisent plus simplement à partir du cas centralisé.
2. Elle définit un contrat en terme de critère de correction *pour le calcul*. Ce contrat est, a priori, indépendant du modèle système et de sa mise en œuvre (y compris donc de la réplication), dans la mesure où la mémoire conserve sa sémantique usuelle.

Lamport considère implicitement dans sa définition que le résultat d'une exécution est déterminée par les lectures intermédiaires effectuées par les processeurs et par l'état final dans lequel cette exécution a laissé la mémoire. C'est en se sens que certains auteurs l'ont comparé [AW94, Mos93, MRZ94, KZ91, AH93] à la sérialisibilité basée sur l'équivalence de vue [Pap86, chapitre 2] et [BHG87, section 2.6].

Nous nous appuyons principalement sur la « séquentialisibilité », une théorie formalisant la cohérence séquentielle, proposée par MIZUNO, RAYNAL et ZHOU dans [MRZ94]. Cette formalisation repose sur l'équivalence de deux histoires, une construction analogue à l'équivalence de vue de transactions proposée par BERNSTEIN, HADZILACOS et GOODMANS dans [BHG87, section 2.6] :

Définition 6.19 (Équivalence de vue) *Deux histoires $(\mathcal{H}_i)_{i\in[1\cdots N]}$ et $(\mathcal{H}'_i)_{i\in[1\cdots N]}$ sont équivalentes si :*

- *Elles sont générées par un même système S, i.e. elles sont assurées par les mêmes processus, et possèdent le même ensemble E d'opérations.*
- *Chaque processus a initié ces opérations dans le même ordre dans les deux histoires, i.e. $\forall i \in [1 \cdots N], <_i = <'_i$.*
- *Les deux histoires ont la même relation « read-from », i.e. $<_{rf} = <'_{rf}$*

Construire un mécanisme optimal s'assurant de la « séquentializibilité » d'une exécution est **NP** complet [MRZ94], de façon analogue à la sérialisibilité basée sur l'équivalence de vue [Pap86, chapitre 2]. De nombreuses approximations efficaces (de complexité polynômiale) ont été proposées dont la plus triviale est l'ordonnancement total de toutes

les opérations. Cette approximation correspond à la cohérence atomique ou encore à une exécution séquentielle[8]

Nous présentons ici, dans notre modèle, la contrainte dénommée « WW-constraint » par MIZUNO et al. dans [MRZ94] :

Définition 6.20 (« WW-constraint ») *Une exécution satisfait à la contrainte « WW-constraint », si l'ensemble des opérations d'écritures, exécutées par l'ensemble des processus, sont totalement ordonnées et, que cet ordre respecte celui dans lequel les écritures ont été initiées.*

Les mêmes auteurs montrent qu'un système assure une cohérence séquentielle si et seulement si chacune de ces exécutions sont légales[9] et respectent « WW-constraint ».

Dans le modèle **Core**, une telle cohérence est satisfaite si toute exécution est *légale* et remplit la condition suivante : $\forall i, j \in [1 \cdots N], \prec_i /_{\mathcal{W}} = \prec_j /_{\mathcal{W}}$, i.e. un ordre global total respectant l'ordre local d'initiation des opérations existe sur les écritures.

Sérialisibilité une copie : Considérons qu'une transaction est attachée à un agent et donc associée à un unique processus. Considérons qu'un processus exécute séquentiellement les opérations qui composent une transaction. Considérons qu'un agent émet ces transactions de façon séquentielle. Considérons les opérations comme étant de deux types, lecture ou écriture. Notons $\mathcal{R}$, l'ensemble des opérations de lecture et $\mathcal{W}$, l'ensemble des opérations d'écriture ; ces deux ensembles constituent une partition de E, l'ensemble des opérations exécutées dans le système.

Définissons une transaction comme une séquence d'opérations *initiées* par un processus. Notons $\mathrm{OP}_{T_i^j}$, l'ensemble des opérations de la $j$ième transaction initiées par le processus p_i. $(\mathrm{OP}_{T_i^j}, <_i)$ est une sous-histoire de $\mathcal{H}_i$.

Soit le cas simple, conservatif, où la réplication est assurée de façon indépendante du contrôle de concurrence, c'est-à-dire nous considérons comme premier élément du critère de correction la cohérence atomique : $\forall i, j \in [1 \cdots N], \prec_i = \prec_j$. Cette cohérence assure l'identité des histoires de chacun des réplicats.

Considérons le critère de sérialisibilité usuel basée sur les conflits [BHG87, chapitre 2] : deux histoires sont équivalentes si elles ont le même ensemble d'opérations, et qu'elles ordonnent de façon identique les opérations conflictuelles. Une histoire est sérialisable s'il existe une histoire sérielle qui lui est équivalente[10].

Deux opérations sont conflictuelles si elles concernent un même objet et que au moins l'une d'entre elles est une écriture.

Les histoires étant identiques, la sérialisibilité une copie est assurée si l'une quelconque des histoires est sérialisable. La sérialisibilité est assurée en l'absence de cycle dans le graphe de sérialisation des transactions ; ce graphe est construit à partir de la fermeture transitive des ordres (partiels) entre opérations conflictuelles. Notons $<_{SA}$, (Sérialisé

[8] PAPADIMITRIOU dans [Pap86] à montré que la sérialisibilité construite sur une équivalence basée sur les conflits est non seulement efficace (i.e. **P** difficile) mais constitue l'approximation **P** difficile, la plus optimale de l'équivalence de vue.

[9] Une lecture sur un objet retourne la dernière valeur écrite sur cet objet.

[10] Par souci de simplification, nous considérons que toute les transactions sont validées. De même nous n'avons pas restreint les histoires correctes aux histoires strictes, ni même recouvrables. Néanmoins, s'agissant de contraintes d'ordre sur des événements de l'histoire, il est clair que ces contraintes peuvent être exprimées dans notre modèle.

$$\mathcal{I}_1 = \{r_1^1, w_1^2, r_1^3, w_1^4\}, \quad \mathcal{O}_1 = \{r_1^1, r_1^3\}, \quad \mathcal{A}_1 = \{w_1^2, w_1^4\}$$
$$\mathcal{I}_2 = \{r_2^1, w_2^2\}, \quad \mathcal{O}_2 = \{r_2^1\}, \quad \mathcal{A}_2 = \{w_2^2\}$$
$$\mathcal{I}_3 = \emptyset, \quad \mathcal{O}_3 = \emptyset, \quad \mathcal{A}_3 = \emptyset$$
$$E = \{r_1^1, w_1^2, r_1^3, w_1^4, r_2^1, w_2^2\}$$

$$\mathcal{H}_1'' = r_1^1 \bullet w_1^2 \bullet r_2^1 \bullet r_1^3 \bullet w_1^4 \bullet w_2^2$$
$$\mathcal{H}_2'' = r_1^1 \bullet w_1^2 \bullet r_2^1 \bullet w_2^2 \bullet r_1^3 \bullet w_1^4$$
$$\mathcal{H}_3'' = \mathcal{H}_2''$$

$$\mathcal{H}_1'' = r_1^1 \bullet w_1^2 \bullet r_2^1 \bullet r_1^3 \bullet w_1^4 \bullet w_2^2$$
$$\mathcal{H}_2'' = r_1^1 \bullet w_1^2 \bullet r_2^1 \bullet w_2^2 \bullet r_1^3 \bullet w_1^4$$
$$\mathcal{H}_3'' = \mathcal{H}_2''$$
$$\mathcal{H}_4'' = \mathcal{H}_2''$$

FIG. 6.7: Arrivée et initialisation d'un nouveau membre (histoire $\mathcal{H}_4$). Nous avons repris la troisième histoire de la figure 6.6 (histoire de gauche), puis avons représenté l'arrivée d'un quatrième réplicat (histoire de droite). L'initialisation représentée est la plus simple : le nouveau membre hérite de l'histoire d'un unique ancien membre ($\mathcal{H}_2''$, du second réplicat).

Avant) la relation suivante entre deux transactions : $T_n^i <_{SA} T_m^j$ si et seulement si les deux conditions suivantes sont satisfaites :

- $\exists a \in \text{Obset}, \exists x \in \text{OP}_{T_n^i}, \exists y \in \text{OP}_{T_m^j}$ tel que $O_x = O_y = a$ et $x \prec y$
- Soit $x \in \mathcal{W}$, soit $y \in \mathcal{W}$.

L'histoire du système est sérialisable si la fermeture transitive de $<_{SA}$ est irréflexive. Ceci suffit à assurer l'absence de cycle dans les dépendances d'opérations conflictuelles et donc de l'absence de cycle dans le graphe de sérialisation des transactions [BHG87].

La sérialisibilité ainsi définie est excessivement conservatrice ; une sérialisibilité une-copie optimale, plus complexe, aurait pu être définie de façon similaire en n'imposant pas la cohérence atomique et en se reposant sur la notion d'équivalence de vue [BHG87, chapitre 8] (même dépendances de lecture et même ensemble d'écritures finales).

Cohérence faible (« weak-consistency ») : Bien que la cohérence faible soit devenue maintenant un terme générique, DUBOIS et al. ont les premiers proposé [DSB86, DSB88] un modèle de mémoire répartie pour multiprocesseurs, dénommé « weak-consistency ». Ce modèle classifie les accès en deux catégories : accès synchronisant A_S et accès non synchronisant A_{NS}. Le modèle « weak-consistency » assure que : 1) tous les accès sont FIFO, et que 2) les accès synchronisants sont totalement ordonnés :

1. $\forall i \in [1 \cdots N], <_i, \forall j \in [1 \cdots N], \prec_j/_{\mathcal{I}_i} \subseteq <_i$.
2. $\forall i, j \in [1 \cdots N], \prec_i/_{A_S} = \prec_j/_{A_S}$ (Un ordre global total existe sur les accès synchronisants).

Si toute paire d'accès concurrents (par deux processeurs distincts) non synchronisants conflictuels (au moins un `write` parmi les accès) est séparée par un accès synchronisant, alors ce modèle est équivalent à la cohérence séquentielle [DSB86, DSB88, Mos93]. Par rapport à une cohérence atomique, une mémoire de type « weak-consistency » n'ordonne que de façon FIFO les accès non synchronisants.

6.3 Initialisation des réplicats

Nous avons jusqu'ici considéré un système composé d'un nombre fixe de processus. Ce trait n'est pas typique d'une application coopérative. Dans une application coopérative, les agents sont autonomes. Ils vont et viennent constamment, observant et modifiant la donnée partagée. Un tel comportement est mieux représenté sous la forme d'un groupe de processus dont les membres rejoignent et quittent le groupe de façon opportuniste [Bir94]. Un tel comportement amène la question de l'initialisation et de la terminaison des nouveaux arrivants.

Le problème posé par l'initialisation des nouveaux réplicats a trait aux contraintes sur l'ordre dans lequel l'arrivée du nouveau membre, son initialisation et les autres opérations effectuées concurremment par les autres membres, sont exécutées. Ce problème n'est pas neuf, mais n'a été étudié essentiellement que dans le cadre d'application principal du modèle de synchronie virtuelle [BJ87] : les applications tolérantes aux fautes. Plusieurs variantes de ce modèle existent selon que les partitions sont supportées et selon les contraintes associées au changements de vues. Dans toutes ces variantes, le nouveau membre est assuré de ne pas percevoir le passé du groupe et de se voir délivrer son état d'initialisation *atomiquement* avec son entrée dans le groupe.

Une telle contrainte d'initialisation se représente simplement dans notre formalisme : initialiser un nouveau venu avec l'état d'un membre revient à faire hériter le nouveau venu de l'histoire locale de ce membre donateur. La figure 6.7 montre un tel exemple.

Le formalisme **Core**, permet de représenter les propriétés de ce type d'initialisation, mais il ne se limite pas à celles-ci : ce protocole induit des synchronisations entre membres a priori inutiles pour les applications n'ayant pas besoin de contraintes aussi fortes que celles définies par la synchronie virtuelle. Ce protocole repose, de plus, sur l'existence d'un réplicat à même de fournir un état de référence complet pour initialiser les nouveaux membres. Nous voulons supporter des contrats de cohérence n'assurant pas l'existence d'un tel réplicat de référence.

Formellement, la contrainte minimale que doit respecter le protocole d'initialisation est la suivante : l'arrivée d'un nouveau membre définit un nouveau système. L'initialisation d'un nouveau membre revient à lui créer ou lui attribuer une histoire locale. L'histoire globale de ce nouveaux système doit être correcte.

Le formalisme, en lui-même, ne modélise pas l'initialisation.[11] L'histoire globale de l'exécution est représentée comme si le nouveau membre avait *toujours* été présent.

La terminaison pose un problème presque similaire. Un processus ne disparaît jamais. Un processus terminé n'initie simplement plus d'opération. Par contre, son histoire doit malgré tout représenter l'exécution des opérations initiées par les autres membres, futurs ou présents, après qu'il a quitté le groupe. Cette contrainte équivaut à imposer que tout processus correct doit posséder un futur *possible* correct dans toute exécution, ce qui nous parait sensé.

6.4 Représentation des méta-opérations

Les méta-opérations sont un élément essentiel de la modélisation d'une exécution : les actions atomiques, les transactions, les sessions, les conversations, la matérialisation de sections critiques (barrières, `acquire/release`, `lock/unlock`, ...) sont des abstractions utilisées dans

[11] Certaines opérations spécifiques de l'initialisation peuvent être présentes, mais elles sont intégrées à une histoire locale dans laquelle figure le passé du ou des membres ayant contribué à l'initialisation du nouveau venu.

de nombreux modèles de cohérence ; il est donc impératif de pouvoir les représenter. Comment construit-on des méta-opérations ? Examinons deux formalismes utilisés dans des domaines très différents (systèmes concurrents et bases de données) :

- Le formalisme introduit par LAMPORT [Lam86] construit une action (« opération ») récursivement au dessus d'un ensemble d'opérations (« événements élémentaires »). LAMPORT pose l'existence d'une relation d'ordre $\prec$[12] sur les opérations. LAMPORT définit deux relations d'ordre sur les actions : *Précède* et *peut affecter*. Une action A_1 *précède* une autre action A_2 si *chacune* des opérations qui définissent A_1 est exécutée avant ($\prec$) chacune des opérations de A_2. Une action A_1 *peut affecter* une autre action A_2 si *certaines* opérations de A_1 sont exécutée avant certaines opérations de A_2. CHARRON-BOST, CORI et PETIT utilisent ce formalisme tout au long de leur *Introduction à l'algorithmique des objets répartis* [CBCP95]. LAMPORT utilise ce formalisme pour prouver des critères de correction pour des exécutions réparties sur plusieurs processeurs [Lam93]. Ce formalisme permet, en particulier, de dériver simplement du code de synchronisation à partir de la preuve de correction d'un algorithme.
- ACTA [CR90, RC94] est un formalisme dérivé de la logique du premier ordre qui permet de représenter l'essentiel des modèles de transactions proposés dans la littérature [CR94]. ACTA définit des classes d'événements « significatifs » [13], essentiellement les opérations sur les objets de la base (`read`, `write`, ...) et les primitives de gestion de transaction (`begin`, `commit`, `abort`, ...), les associe à des transactions et exprime les critères de correction des transactions sous forme de prédicats construits sur des occurrences d'événements ainsi que sur des relations d'ordre entre ces événements. Ces prédicats représentent les relations de dépendances (dépendances de commit, dépendances d'abort, dépendances sur les opérations conflictuelles, ...) entre événements qui définissent les critères de correction associés à ces divers modèles de transaction.

Notre modèle permet de représenter les mêmes abstractions que ces deux formalismes ; par exemple, la construction de la sérialisibilité une-copie présentée à la section 6.2.4 (irréflexivité sur la fermeture transitive de la relation de précédence entre opérations conflictuelles) est une restriction aux conflits `read/write` de la définition donnée en exemple par RAMAMRITHAM et CHRYSANTHIS dans [RC94].

6.5 Exemple d'utilisation du modèle Core

Nous donnons dans cette section, quelques exemples de l'utilisation du modèle pour la spécification de quelques critères de correction, dans le cadre d'applications coopératives simples. Nous commençons par une contrainte de cohérence relativement classique, que nous faisons évoluer vers une contrainte de divergence.

6.5.1 Gestion d'allocation de ressource

Cet exemple se base sur un paradigme classique de coopération autour d'une ressource, munie d'un critère de correction simple : la gestion répartie des entrées/sorties d'un parking

[12] Cette relation est purement axiomatique et ne doit pas être confondue avec la relation « happened before » définie dans [Lam78].

[13] Les classes d'événements et les dépendances construites sur ces événements dépendent des modèles de transaction considérés [CR94].

$$N = 2, P = 1, \mathcal{I}_1 = \{inc_1^1, dec_1^2\}, \mathcal{I}_2 = \{dec_2^1\}$$

$$\begin{array}{rcl} \mathcal{H}_1 & = & inc_1^1\ dec_1^2\ dec_2^1 \\ \mathcal{H}_2 & = & dec_2^1\ dec_1^2\ inc_1^1 \\ \mathcal{H}'_2 & = & dec_1^2\ inc_1^1\ dec_2^1 \\ \mathcal{H}''_2 & = & dec_2^1\ inc_1^1\ dec_1^2 \end{array}$$

$$\begin{array}{rcl} (\mathcal{H}_1, \mathcal{H}_2) & \notin & \mathcal{CORRECT} \\ (\mathcal{H}_1, \mathcal{H}'_2) & \in & \mathcal{CORRECT} \\ (\mathcal{H}_1, \mathcal{H}''_2) & \notin & \mathcal{CORRECT} \end{array}$$

FIG. 6.8: Exemple d'un parc d'une place munis de 2 accès. L'accès 1 laisse entrer puis sortir une voiture. L'accès 2 fait entrer une voiture. $\mathcal{H}_1$ est l'histoire exécutée sur le premier accès. Les histoires $\mathcal{H}_2, \mathcal{H}'_2, \mathcal{H}''_2$ sont des alternatives d'exécution effectués par le deuxième processus contrôlant le deuxième accès.

---

automobile.

Considérons un parking muni d'un nombre P de places, et d'un nombre N d'accès. Soit comme critère de correction, la présence dans le parking d'un nombre de voitures inférieur ou égal au nombre de places P.

Modélisons le problème de la façon suivante : un entier D représente le nombre de places disponibles dans le parking. Cet entier est initialisé par le nombre de places P du parking. Deux opérations, *inc* et *dec*, sont définies sur l'entier. *inc* (resp. *dec*) incrémente (resp. décrémente) D de 1. Le critère de correction est que cet entier D doit rester positif ou nul. Clairement, D représente la valeur associée à un sémaphore.

Modélisons le système de la façon suivante : chaque accès est contrôlé par un unique processus p_i contrôlant les entrées/sorties du parc et garantissant l'invariant applicatif grâce à un sémaphore dont la valeur est localement représentée par un réplicat de l'entier D. La figure 6.8 illustre cet exemple sur une configuration simple du parking.

Soit E_i (resp. S_i) le nombre d'opérations *dec* (resp. *inc*) *exécutées* par le processus p_i. Soit $\mathcal{E}_i$ (resp. $\mathcal{S}_i$) le nombre d'opérations *dec* (resp. *inc*) *initiées* par le processus p_i.

A tout moment (sur toute coupe), pour tout processus p_i, $D_i = P - (E_i - S_i)$ représente la perception (locale) qu'a ce processus du nombre de places libres. *Si les opérations dec sont globalement ordonnées*, $P - \sup_{i \in [1 \cdots N]} E_i$ est une borne inférieure du nombre de places libres. Dans ce cas, la contrainte (locale) qu'une opération *dec* n'est exécutée sur un processus p_i que si $D_i > 0$ suffit donc à satisfaire le prédicat de correction.

Concernant le gestionnaire de cohérence, la spécification de l'ordre total sur les opérations *dec* se formalise comme : soit $\mathcal{D}$, l'ensemble des opérations *dec* exécutées sur le système, $\forall i, j \in [1 \cdots N], \prec_i /_{\mathcal{D}} = \prec_j /_{\mathcal{D}}$, en notant $\prec /_{\mathcal{D}}$, la restriction de $\prec$ aux opérations *dec*.

On se persuadera que les opérations *inc* commutant entre elles aussi bien qu'avec les opérations *dec*, leur ordre n'influe ni sur la correction, ni sur la convergence ultime des réplicats. Elles ont, par contre, un rôle sur la vivacité de l'application ; si aucune sortie n'est prise en compte, l'application arrête de progresser après les P premières entrées ! Tout ordonnancement global des *inc* permet une progression théorique optimale : tout processus désirant exécuter une entrée (*dec*) ayant une vue locale précise du nombre de places libres, aucune opération *dec*

ne sera bloquée alors qu'une place est effectivement disponible. Le choix de l'ordonnancement des *inc* induit, en quelque sorte, un compromis cohérence/réactivité. Sans ordonnancement global des *inc*, on ne synchronise aucune opération de sortie, mais on peut bloquer l'entrée de véhicule alors même que des places sont disponibles. Le caractère optimal en terme de latence dépend de nombreux facteurs tel que l'algorithmique utilisée pour la mise en œuvre de l'ordonnancement total, le profil et la répartition des entrées et des sorties aussi bien que des caractéristiques de la plate-forme d'exécution (débit, latence de la plate-forme de communication).

En terme de réactivité, les gains obtenus sont simples : une cohérence séquentielle [Mos93, RS95a] (ou encore linéarisable [HW90]) entraine une synchronisation globale de tous les processus pour chaque opération. La vitesse de progression de l'application est limitée par la moyenne du `RTT`[14] entre les processus coopérants. Ces deux modèles restent immunes à toute exploitation aussi bien des propriétés algébriques que de propriétés applicatives (borne et schéma d'accès). Considérons la configuration d'un parking où n'existerait qu'une entrée, le reste des accès ne servant que pour la sortie. L'ordre total des *dec* serait garanti sans frais (accès unique). Un ordre FIFO permettrait une progression à vitesse locale. Seule la vitesse de perception des sorties par l'accès d'entrée limiterait la progression de l'application et celle-ci ne pourrait être réduite que par la diminution de la latence point à point[15]. La correction de ce protocole FIFO repose sur trois caractéristiques : 1) la commutativité des opérations, 2) un prédicat dont la correction ne repose que sur le sous-ensemble $\mathcal{D}$ des opérations d'entrée, 3) l'exploitation d'une configuration d'accès propre à l'environnement d'exécution applicatif (un accès d'entrée unique).

6.5.2 Deux critères de divergence

6.5.2.1 La Gestion d'allocation de ressource revisitée.

Reprenons la spécification de la gestion de ressource. Définissons une mesure μ, tel que, $\mu(dec) = 1$ et $\mu(inc) = 0$. Notons $\mathcal{D}$, l'ensemble des opérations *dec* d'une exécution. Le critère de correction de cette application peut s'exprimer comme :

$$\forall \Delta_{C^{c_i}} = (A_j)_{j\in[1\cdots N-1]} \text{ tel que } c_i \in \mathcal{D}, \forall i \in [1\cdots N-1], \mu(A_i) = 0$$

Par définition (cf. définition 6.17, page 65) de $\Delta_{C^{c_i}}$, les A_j représentent les opérations non perçues par le processus p_i au moment où il a exécuté l'opération c_i. Les mesures non nulles étant toutes positives, cette condition impose que p_i ait perçu toute les opérations *dec* distantes avant d'exécuter la sienne. Cette condition entraine donc une divergence nulle des processus vis-à-vis des opérations *dec* ; ceci revient à assurer un ordre total sur les *dec*, et ce critère est donc équivalent à celui présenté en 6.5.1.

Dans cette application, toutes les opérations (*inc* et *dec*) sont commutatives, mutables et observantes (cf. section 3.3.1). En pratique, seul le consensus (ordre global) sur l'ordre d'exécution des opérations *dec* est nécessaire à la correction de l'application. Ce consensus n'a aucun rôle sur la convergence ultime de la valeur du sémaphore, puisque les opérations sont commutatives. Il a un rôle sur le respect d'une borne sur la divergence des divers réplicats,

[14] Temps de parcours aller-retour moyen entre deux sites. Nous considérons comme prédominant le temps de propagation des messages par rapport au temps de traitement locaux (processus fortement éloignés).

[15] Par une diminution du temps d'acheminement des messages et éventuellement par une augmentation du débit lorsque celui-ci est un facteur limitatif.

sur tous les états intermédiaires du calcul répliqué. Il s'agit donc d'une optimisation ; cette optimisation peut être partagée par toute mise en œuvre d'un sémaphore.

Supposons qu'à des fins d'optimisation, anticipant une sortie massive de véhicules par exemple, nous décidions de relâcher un peu le critère de correction en acceptant que K voitures puissent rentrer dans le parking sans trouver de place. Le nouveau critère de correction s'exprime simplement par :

$$\forall \Delta_{C^{c_i}} = (A_j)_{j\in[1\cdots N-1]} \text{ tel que } c_i \in \mathcal{D}, \forall i \in [1\cdots N-1], \mu(A_i) \leq K$$

Dans un cadre bancaire où les points de retraits d'argent seraient des réplicats des comptes, la banque pourrait ainsi exprimer des bornes de divergences plus ou moins élevées en fonction des risques d'impayés qu'elle est prête à subir et en fonction du gain de réactivité que cela pourrait procurer à l'utilisateur. Le « surbooking », courant en matière de moyens de transports ou d'hôtels, pourrait être modélisé de cette manière.

6.5.2.2 Simulation de combat aérien

Cet exemple illustre comment spécifier la correction d'une application sur un critère de divergence. Nous exploitons une propriété courante, à savoir l'existence d'une métrique [BG83] sur différents états de l'information. Une métrique permet de quantifier la divergence de deux états d'une même information répliquée. Cette idée est loin d'être neuve [GMW82, BGM90] et s'apparente beaucoup à une application de l'ϵ-sérialisibilité [PL91, RP95, WYP97] dans le cadre de caches coopérants d'objets répliqués[16].

Les mondes virtuels dans lesquels des objets, sous la responsabilité d'agents, sont déplacés dans un espace affine sont évidemment de bons candidats à ce genre d'application. Considérons un simulateur de combat aérien (ACM [Rai], par exemple). Chacun des N processus p_i coopérant gère un avion défini par sa position M_i dans un espace affine à trois dimensions $(O, \vec{x}, \vec{y}, \vec{z})$. Chaque agent se satisfait d'une précision approximative de la position des objets mobiles ; toute imprécision ne remettant pas en cause une action importante (collision ou destruction, par exemple) et restant compatible avec des mesures d'évitements réalistes est considérée comme acceptable. Le critère de correction de l'application est que les positions d'un objet telles que perçues par deux agents distincts, à un instant donné, ne doivent pas être éloignés d'une distance supérieure à D_{max}. Notons $d(M_i, M_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$ la distance euclidienne traditionnelle entre deux points d'un espace affine à trois dimensions. Le prédicat de correction se traduit par $\forall i, j \in [1\cdots N], d(M_i, M_j) < D_{max}$ (pour tout objet).

Modélisons le problème de la façon suivante : deux opérations sont définies sur les objets répliqués, *move* et *read*. Les opérations *move* sont relatives et commutatives. Les opérations *read* permettent de calculer l'éventuelle collision d'objets et leurs représentations sur un affichage local associé à l'agent (fenêtre X, par exemple). La position de tout objet est définie par sa position initiale et la séquence d'opérations *move* lui ayant été appliquées. Cette position est représentée par un point M_i. Ce point représente la position de l'objet telle que représentée par le réplicat attaché au processus p_i.

Associons à toute opération du système répliqué, une métrique $\mathcal{M}$ définie par la distance séparant le point original M de celui M' obtenu après exécution de l'opération ($\mathcal{M}(\text{move}) = d(M, M')$ et $\mathcal{M}(\text{read}) = 0$).

[16] Une différence notable, est que la borne de la divergence permise est une propriété de l'objet et non des transactions, ce qui augmente de beaucoup l'efficacité de la solution, mais la restreint au domaine coopératif (Tous les processus agréent [statiquement, dans le cas simple] sur la borne de divergence).

Définissons une fonction μ sur E, en considérant $\mu(\{e_i\}) = \mathcal{M}(e_i)$. μ est une mesure[17]. L'exécution de notre système est correcte si,

$$\forall \Delta_{C^{c_i}} = (A_j)_{j\in[1\cdots N-1]} \text{ tel que } c_i \in \mathcal{D}, \forall i \in [1\cdots N-1], \mu(A_i) < D_{max}$$

autrement dit, si la mesure de l'ensemble des opérations de chacun de ses voisins non perçues par p_i au moment où il a initié c_i reste en deçà de D_{max}.

Ce critère ne reflète pas exactement notre critère de correction ; il moyenne les divergences d'objets distincts. Étendre le critère de correction à un système de K-mesure où K est le nombre d'objet du monde virtuel est trivial.

6.6 Bilan du modèle - bilan du formalisme

Le modèle que nous venons de présenter rend compte simplement des deux fonctions traditionnelles associées à la gestion de cohérence d'objets répliqués : *Le contrôle de divergence* s'assure de « l'équivalence » des histoires locales répliquées ($\mathcal{H}_i$). *Le contrôle de concurrence* s'assure de la correction de l'histoire globale $\mathcal{H}$ (ou de l'une quelconque des histoires locales $\mathcal{H}_i$, lorsque celles-ci sont « équivalentes »). De plus les deux fonctions s'exercent au travers d'un principe commun, unique, l'ordonnancement des opérations. En terme de propriété, la diffusion d'une mise à jour revient à faire percevoir aux processus destinataires, l'exécution de l'opération (potentiellement concurrente) effectuée par le processus émetteur. L'invalidation de la copie d'une donnée associée à un processus est le moyen qui permet d'assurer que ce processus, lors de son prochain accès, aura perçu l'ensemble des opérations (distantes) effectuées par les autres membres sur cette donnée. Le contrôle de réplication peut être vu, en fait, comme un mandataire du contrôle de concurrence, chargé du contrôle de la perception des actions réparties.

Un des traits essentiels du formalisme est sa simplicité : l'ensemble des dépendances des opérations est visible ; nul graphe complexe de dépendances ni d'axiomes additionnels pour spécifier la légalité [MRZ94] des lectures ou la notion d'opérations globalement effectuées (« globally performed » [GLL+90]). Au prix d'une redondance des opérations, naturelle dans le cadre répliqué et du fait que tous les modèles impératifs se basent à un moment ou un autre sur une exécution séquentielle (processus voire tâche), une exécution est composée d'un ensemble d'histoires linéaires.

Certaines caractéristiques peuvent sembler déroutantes, comme la présence de lectures distantes, par exemple, dans une histoire locale. Le fait de représenter explicitement ces lectures distantes ne signifie pas qu'une mise en œuvre doive *transmettre* les lectures, mais seulement que l'exécution de cette mise en œuvre doit donner le même résultat que si elle avait été transmise.

De plus, cette représentation permet de formaliser, simplement, les contraintes liées à la fusion d'histoires ; celles matérialisant l'initialisation de nouveaux membres ou bien l'arrivée d'agents mobiles ayant déjà un vécu (une histoire).

La simplicité d'un formalisme est un point clé dans la compréhension d'un principe. Celui présenté répond à ce principe de simplicité. Il contribue à une meilleure compréhension de la cohérence.

[17] i.e. $\sim$ une application $\sigma-$additive sur un ensemble de parties d'ensemble à valeur dans $\mathbf{R}$ [BG83].

6.7 Résumé

Nous avons présenté dans ce chapitre notre première contribution : comment construire la coopération de façon à prendre en compte la latence des communications entre agents coopérants.

Cette construction s'est fait essentiellement en quatre étapes :

1. Nous avons posé les deux hypothèses essentielles, asynchronisme et déterminisme, caractérisant notre environnement d'exécution et les applications coopérantes.
2. Nous avons introduit un nouveau modèle de partage, la *réplication coopérante.*
3. Nous avons introduit un modèle d'exécution, le modèle **Core**, permettant de rendre compte des propriétés pertinentes vis-à-vis de la correction de l'application et ce, sans modéliser le système lui-même. Ce point est essentiel puisque ce dernier doit être déduit a posteriori du critère de correction applicatif.
4. Nous avons exercé ce modèle d'abord dans un cadre traditionnel (cohérence FIFO, causale, séquentielle, ...), puis enfin sur des contraintes de divergence. Cette catégorie de contrainte est importante pour les applications coopératives, et pourtant bien difficile à formaliser dans les modèles faisant abstraction de la réplication.

Le modèle **Core** a une caractéristique essentielle. En faisant abstraction des *mécanismes* de coopération entre processus, il définit les seules *propriétés* de correction attachées à cette coopération. Cette abstraction est à l'origine de deux enseignements importants :

- Le contrôle de divergence, ne constitue qu'un élément du contrôle de concurrence. Les deux ne constituent que les éléments qui permettent de contraindre l'ordre réparti dans lequel sont appliquées les opérations accédant à la donnée partagée. Cet enseignement redécouvre, mais sous une forme moins empirique, certaines des optimisations développées dans le cadre des mémoires partagées réparties (cf chapitre 3).
- Le protocole d'initialisation ne peut être réellement isolé du protocole de coopération « normal », i.e. celui en exercice lorsque la composition du groupe est stable. Ce n'est que dans des protocoles d'initialisation « simples », basés sur le transfert atomique de l'état complet d'un membre de référence, qu'une séparation relativement claire en deux modes de fonctionnement peut être effectuée.

Chapitre 7

Décomposition du critère de correction

Nous présentons dans ce chapitre, les principes de décomposition du critère de correction qui permettent d'adapter et de réutiliser le code de gestion de cohérence. **Core** vise la gestion économe de la latence de la perception des opérations distantes ; les principes définis concernent donc essentiellement les implantations pessimistes de la gestion de cohérence.

Nous considérons qu'un seul et unique gestionnaire de cohérence doit être en charge de la cohérence de l'ensemble des données répliquées. Ceci, en sus des hypothèses sur le déterminisme et sur le modèle de réplication formulées au début de la problématique. Bien que **Core** permette la présence de plusieurs gestionnaires de cohérence, **Core** ne sait ni gérer leur coopération, ni composer leurs propriétés.

7.1 Idée de base

Reprenons le modèle présenté au chapitre précédent. L'idée intuitive de **Core** est de déterminer quand un processus au vu de son histoire locale, donc partant d'une coupe dont il sait qu'elle appartient à une histoire correcte, peut décider localement que l'adjonction d'une opération à cette histoire ne remet pas en cause la correction de l'application. Cette connaissance peut servir ensuite à prendre *localement* la décision d'ordonner et d'exécuter cette opération. Déterminer cette connaissance revient à identifier parmi les multiples relations d'ordre qui gouvernent la correction de l'exécution, celles qui contrôlent la perception des opérations distantes, et donc la divergence des réplicats. Dans un système pessimiste, ces «sous-ordres» sont typiquement assuré par des mécanismes de synchronisation réparti. Ces mécanismes sont sensibles aux conditions d'exécution de l'application, particulièrement de la latence de communication. Pouvoir adapter leur mise en œuvre est donc important.

Ces «sous-ordres» restreignent la liberté des histoires locales appartenant à une même exécution et définissent un cadre à l'intérieur duquel la décision de correction peut s'effectuer localement. Les relations d'ordres restantes qui définissent le reste des contraintes de correction, peuvent être implantées par des prédicats locaux. Ces ordres influent sur la réactivité de l'application, mais de façon relativement déterministe : leur coût sur les performances de l'application n'est pas soumis aux variations de l'environnement d'exécution propre à la coopération distante. La réutilisation du code implantant ces contraintes est probablement intéressante. Son adaptation, à l'exécution ne l'est probablement pas autant.

Il doit être clair, que **Core** ne vise pas à minimiser le coût des communications nécessaires à la satisfaction des contraintes de cohérence. **Core** permet seulement à un utilisateur de

pouvoir adapter sa gestion de cohérence. Les critères utilisés par cet utilisateur, ne sont pas du ressort de **Core**.

Exemple 7.1 *Considérons l'application de gestion de ressource présentée dans la section 6.5.1 : l'information partagée est l'entier représentant la valeur d'un sémaphore. La commutativité des opérations inc et dec n'est pas spécifique de l'utilisation dans le cadre d'un sémaphore, ni même des contraintes d'intégrités associées à la gestion de la ressource (parc) ; la commutativité fait partie des propriétés algébriques de l'addition et de la soustraction sur des entiers. Prendre en compte la commutativité de ces opérations permet à des agents distincts d'exécuter concurremment ces opérations, assurant ainsi une réactivité supérieure à ce qu'elle serait si les agents avaient dû se synchroniser pour les exécuter. La précision que l'on peut se permettre d'avoir sur la valeur de cet entier lorsqu'une opération de lecture, non commutative, est effectuée dépend par contre de l'application ; cette précision est, dans l'exemple décrit, construite en associant une métrique aux opérations mutables (inc et dec) et en utilisant celle-ci pour borner la divergence des histoires des agents (donc l'état de l'entier). L'application de simulation de combat aérien présentée dans la section 6.5.2.2 présente des caractéristiques proches.*

Le choix, à l'exécution, de la borne contrôlant la divergence de l'information partagée est important. Une application s'exécutant dans un contexte où tous les membres sont sur le même réseau local, peut choisir une borne d'imprécision nulle, alors que la même application s'exécutant avec certains membres reliés par un modem, choisira une borne plus adaptée, afin de conserver une réactivité appropriée. Le fait de *garantir* une telle borne permet de définir un contrat rigoureux et tangible avec les utilisateurs. L'adaptabilité ne se borne pas aux métriques ; elle concerne toutes les propriétés pilotant la perception des opérations distantes. La consultation (`finger`) des personnes ayant une session active sur les machines d'un laboratoire peut faire l'objet de propriétés différentes selon que les membres du projet sont sur un même réseau local, ou réparti, par exemple chez eux (travail à domicile, unités de recherche décentralisées).

7.2 Gestionnaire de cohérence – Contrat de cohérence

Nous désignons par *gestionnaire de cohérence* la machinerie, i.e. tous les aspects du code d'application, qui a la charge d'ordonner et de propager les opérations accédant à et modifiant l'information partagée. De part le modèle d'application coopérative présenté au chapitre précédent, une telle machinerie est *répartie*.

Le but ultime de ce chapitre est de poser les bases qui permettent de réunir en un unique *composant* réparti, une telle machinerie, ceci en respectant une contrainte : ce composant doit contrôler de bout-en-bout et de façon *déterministe*, l'ordre dans lequel les processus coopérants accèdent à l'information partagée.

Cet ordre définit ce que nous appelons un *contrat de cohérence*. Il représente le modèle de cohérence particulier que ce composant assure aux applications l'utilisant.

La figure 7.1 illustre la place et le rôle du gestionnaire de cohérence dans la coopération de deux processus. Le gestionnaire est implanté sous forme de deux mandataires, M_1 et M_2, un pour chaque contexte. Les accès *initiés* par chacun des agents ($Agent_1$ et $Agent_2$) sont capturés par une voie non spécifiée et soumis au mandataire local du gestionnaire de cohérence. Celui-ci contrôle ensuite leur application sur le réplicat.

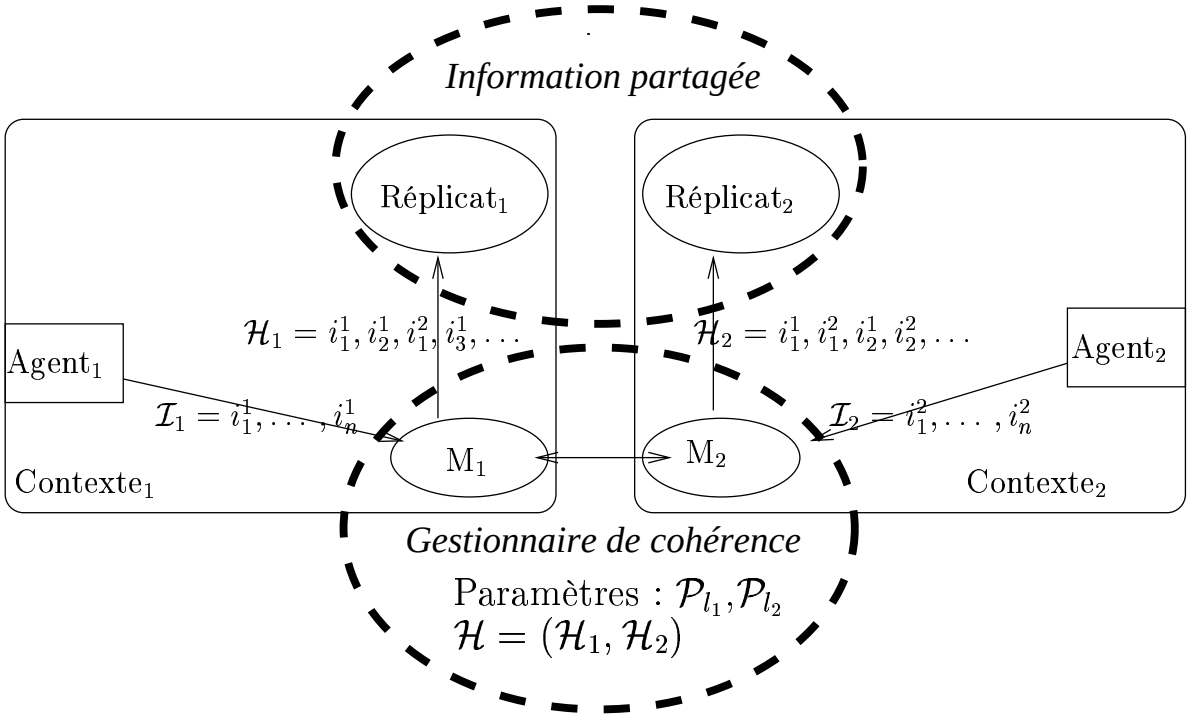


FIG. 7.1: Gestionnaire de cohérence. L'information partagée et le gestionnaire de cohérence sont deux entités répartie. L'information partagée n'est répartie que grâce au service du gestionnaire de cohérence. M_1 et M_2 sont les deux mandataires réalisant le gestionnaire de cohérence. $\mathcal{H}_1$ n'est connu que de M_1 implanté dans le contexte 1. $\mathcal{H}_2$ n'est connu que de M_2 implanté dans le contexte 2. Les deux mandataires coopèrent pour s'assurer que l'histoire globale, H, est correcte.

7.3 Principes de Core

Regardons un tel gestionnaire de cohérence comme un automate. L'idée qui sous-tend les principes de **Core** est de laisser l'application pouvoir paramétrer ce gestionnaire par deux prédicats évaluables *localement.*

Un tel gestionnaire fonctionne d'une manière similaire aux ordonnanceurs hybrides [BHG87, section 4.5] : chacun des prédicats définit des contraintes sur l'ordre d'exécution des opérations ; l'union de ces contraintes permet au gestionnaire d'assurer la correction de l'histoire globale. Nous nous appuyons sur une définition et quelques notations supplémentaires pour définir ces prédicats.

Définition 7.1 *(séquence plausible) Une séquence d'opération $\mathcal{H}$ est* plausible *sur un processus p_i si et seulement si il existe une exécution correcte $(\mathcal{H}_j)_{j\in[1\ldots N]} \in \mathcal{CORRECT}$ pour laquelle $\mathcal{H}$ est préfixe fermé de l'histoire locale $\mathcal{H}_i$, de ce processus.*

Une séquence locale d'opérations est plausible sur un processus p_i si cette séquence d'opérations peut évoluer en une histoire locale figurant dans au moins une histoire globale correcte. La plausibilité détermine le critère d'autonomie que l'on peut prendre dans une décision ; au vu d'une histoire locale *non* plausible, on peut décider localement de la *non* correction de l'histoire globale.

Une coopération entre les mandataires locaux d'un gestionnaire de cohérence est néces-

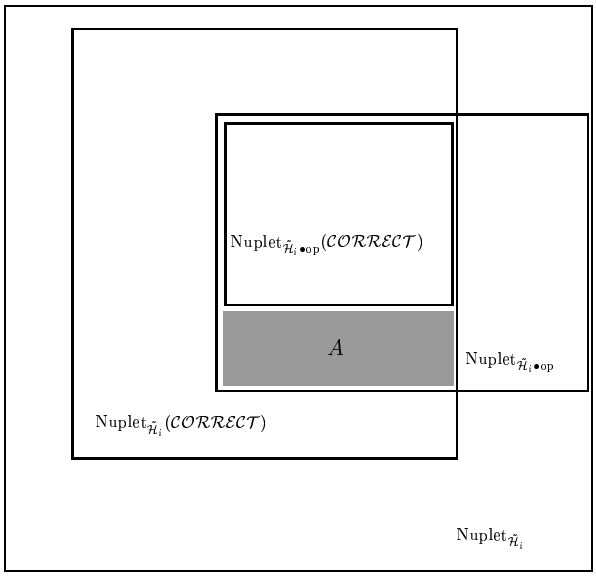


FIG. 7.2: Inclusion des coupes

saire dès lors que les histoires locales de plusieurs processus distincts sont plausibles, donc acceptables localement, mais ne peuvent appartenir à la même exécution globale. **Core** ne fait qu'implanter ce principe : le mandataire local d'un gestionnaire de cohérence teste si la vue de la seule histoire locale du processus permet de déterminer la correction de l'application.

Notons $\text{Proj}_i(\mathcal{H})$, l'ensemble des préfixes fermés de la séquence locale d'opération du processus p_i dans la coupe $\mathcal{H}$. Notons $\text{Nuplet}_{\tilde{\mathcal{H}}_i}$ l'ensemble des N-uplets $(\mathcal{H}_i)_{i\in[1\cdots N]}$ dont l'histoire sur p_i a pour préfixe $\tilde{\mathcal{H}}_i$; soit $\text{Nuplet}_{\tilde{\mathcal{H}}_i} = \{\mathcal{H} = (\mathcal{H}_i)_{i\in[1\cdots N]}, \tilde{\mathcal{H}}_i \in \text{Proj}_i(\mathcal{H})\}$. Notons $\text{Nuplet}_{\tilde{\mathcal{H}}_i}(\mathcal{CORRECT}) = \mathcal{COUPE}(\mathcal{CORRECT}) \cap \text{Nuplet}_{\tilde{\mathcal{H}}_i}$, l'ensemble des coupes d'exécution correctes ayant exécuté le préfixe $\tilde{\mathcal{H}}_i$ sur le processus p_i.

Nous avons les relations suivantes (cf. figure 7.2) :

- $\text{Nuplet}_{\tilde{\mathcal{H}}_i \bullet \text{op}} \subseteq \text{Nuplet}_{\tilde{\mathcal{H}}_i}$,
- $\text{Nuplet}_{\tilde{\mathcal{H}}_i \bullet \text{op}}(\mathcal{CORRECT}) \subseteq \text{Nuplet}_{\tilde{\mathcal{H}}_i}(\mathcal{CORRECT})$,
- $\text{Nuplet}_{\tilde{\mathcal{H}}_i}(\mathcal{CORRECT}) \subseteq \text{Nuplet}_{\tilde{\mathcal{H}}_i}$,
- $\text{Nuplet}_{\tilde{\mathcal{H}}_i \bullet \text{op}}(\mathcal{CORRECT}) \subseteq \text{Nuplet}_{\tilde{\mathcal{H}}_i \bullet \text{op}}$,
- $A(\tilde{\mathcal{H}}_i, \text{op}) = (\text{Nuplet}_{\tilde{\mathcal{H}}_i \bullet \text{op}} \backslash \text{Nuplet}_{\tilde{\mathcal{H}}_i \bullet \text{op}}(\mathcal{CORRECT})) \cap \text{Nuplet}_{\tilde{\mathcal{H}}_i}(\mathcal{CORRECT})$

L'ensemble A représente certaines coupes d'exécutions dont le processus p_i a exécuté $\tilde{\mathcal{H}}_i \bullet \text{op}$. Ces coupes appartiennent à des exécutions qui étaient correctes après que le processus p_i ait exécuté la séquence $\tilde{\mathcal{H}}_i$ mais qui ne le sont plus une fois que p_i a exécuté *op*. Si $\text{Nuplet}_{\tilde{\mathcal{H}}_i \bullet \text{op}}(\mathcal{CORRECT})$ est vide, le préfixe $\tilde{\mathcal{H}}_i \bullet \text{op}$ n'est pas plausible et p_i peut décider localement de l'incorrection de l'histoire générée. Par contre, si $\text{Nuplet}_{\tilde{\mathcal{H}}_i \bullet \text{op}}(\mathcal{CORRECT})$ et A sont non vides, p_i ne peut décider localement de la correction de l'exécution globale s'il exécute $\tilde{\mathcal{H}}_i \bullet \text{op}$; c'est le cas où une coopération entre plusieurs membres est requise pour s'assurer de la correction.

Considérons les trois prédicats suivants :

$\mathcal{P}_{l_1}$: $\mathcal{P}_{l_1}(\tilde{\mathcal{H}}, \text{op})$ est satisfait sur p_i, ce que nous notons $\mathcal{P}^i_{l_1}(\tilde{\mathcal{H}}, \text{op})$, si et seulement si $\tilde{\mathcal{H}} \bullet \text{op}$ est plausible sur p_i.

$\mathcal{P}_{l_2}$: $\mathcal{P}_{l_2}(\tilde{\mathcal{H}}, \text{op})$ est satisfait sur p_i, ce que nous notons $\mathcal{P}^i_{l_2}(\tilde{\mathcal{H}}, \text{op})$, si et seulement si $A(\tilde{\mathcal{H}}, \text{op}) = \emptyset$.

$\mathcal{P}_g$: Soit une opération op et une histoire locale $\tilde{\mathcal{H}}$. $\mathcal{P}_g(\tilde{\mathcal{H}}, \text{op})$ est satisfait si dans toute exécution telle que $\tilde{\mathcal{H}}$ est préfixe de op sur le processus p_i ayant initié op, $\mathcal{P}^i_{l_1}(\tilde{\mathcal{H}}, \text{op})$ vrai implique que la coupe de borne C^{op} appartient à une exécution correcte : $\forall \mathcal{H} = (\mathcal{H}_i)_{i \in [1 \cdots N]}, i$ tel que $\text{op} \in \mathcal{I}_i, \tilde{\mathcal{H}} = P(\mathcal{H}_i)_{\text{op}}$,
$[\mathcal{P}^i_{l_1}(\tilde{\mathcal{H}}, \text{op})] \implies C^{\text{op}} \in \mathcal{COUPE}(\mathcal{CORRECT})$.

$\mathcal{P}_{l_2}$ joue notre rôle « d'ancrage » : s'il est satisfait, la satisfaction de $\mathcal{P}_{l_1}$, le critère local, assure la correction de l'application au moment opportun, i.e. lorsque qu'une opération *initiée* par un processus doit être ajoutée à l'histoire locale.

Informellement, $\mathcal{P}_{l_1}$ s'assure de la correction « locale » de l'exécution, i.e. relativement à toute les opérations perçues « localement », $\mathcal{P}_{l_2}$ contrôle quand la perception des opérations distante est nécessaire tandis que $\mathcal{P}_g$ offre des garanties sur l'ordre dans lequel les opérations sont perçues.

Définition 7.2 (Critère Core) *Soit $\mathcal{H} = (\mathcal{H}_i)_{i \in [1 \cdots N]}$ une coupe d'exécution correcte, op une opération initiée par le processus p_i et $\tilde{\mathcal{H}}_i$, le préfixe de cette opération op sur p_i. Nous appelons critère **Core**, noté $CritCore(\tilde{\mathcal{H}}_i, op)$, le prédicat suivant :*

$$\mathcal{P}^i_{l_1}(\tilde{\mathcal{H}}_i, op) \wedge \left[\mathcal{P}^i_{l_2}(\tilde{\mathcal{H}}_i \bullet op) \vee \mathcal{P}_g(\tilde{\mathcal{H}}_i \bullet op)\right]$$

Le critère **Core** n'est défini que sur certaines coupes ; celles dont au moins une histoire locale est le préfixe fermé d'une opération initiée sur le processus ayant exécuté cette histoire locale. Le critère **Core** n'est donc pertinent qu'au moment où le mandataire local d'un gestionnaire ayant reçu une opération de son agent doit l'insérer dans l'histoire locale du processus.

Le critère **Core** permet de construire une coupe correcte à partir d'une coupe de borne jugée correcte. Il permet d'implanter un *contrat de cohérence* de façon incrémentale.

Par définition, la satisfaction du critère **Core** sur toute coupe de borne assure la correction de toute application dont le critère de correction ne dépend que des coupes de bornes. C'est le cas, en particulier, des critères basés sur le contrôle de divergence. Nous conjecturons que le critère **Core** permet d'assurer d'autres critères de correction, en particulier la causalité et la séquentialité.

Conjecture 1 *Si le critère **Core** est satisfait sur toute coupe d'une exécution où $\mathcal{P}_g$ est défini, alors cette exécution est correcte.*

Selon cette conjecture, un gestionnaire de cohérence qui construit ses histoires locales en satisfaisant le critère **Core**, ne produit que des histoires correctes.

Le critère **Core** n'est sûrement pas optimal en ce qui concerne la maximisation du nombre d'histoires permises. Seule une décision basée sur une perception de toutes les opérations concurrentes peut être optimale. C'est justement ce que nous avons voulu éviter, le but étant de minimiser la latence moyenne des exécutions.

7.4 Quelques exemples

Exemples de l'utilisation des principes **Core** à des modèles de cohérence connus :

Exemple 7.2 *Considérons la mise en œuvre d'une mémoire répliquée à cohérence hybride («hybrid consistency») [AF92, ACFW93]). Dans ce modèle de mémoire, les accès sont étiquetés soit comme* `strong` *soit comme* `weak`*. Le contrat de cohérence garanti à l'application par ce type de mémoire est le suivant : les accès* `strong` *sont globalement ordonnés et respectent l'ordre FIFO ; les accès* `weak` *sont FIFO.* ATTIYA *et* FRIEDMAN *montrent dans [AF92] que si une application étiquette toutes ses écritures comme* `strong` *et toute ses lectures comme* `weak`*, une mémoire à cohérence hybride donne les mêmes résultats qu'une cohérence séquentielle.*

Supposons une telle mémoire conçue selon le modèle système ***Core****. Construisons* $\mathcal{P}^i_{l_1}(\tilde{\mathcal{H}}, op)$ *de telle sorte qu'il soit satisfait si l'histoire locale* $\tilde{\mathcal{H}} \bullet op$ *est légale sur* p_i*. Un contrat de cohérence assurant un ordre global total sur les opérations d'écriture et un ordre FIFO sur les lectures implantera une cohérence hybride si* $\mathcal{P}^i_{l_2}(\tilde{\mathcal{H}}, op)$ *est satisfait pour toute opération op étiquetée* `strong`*. Une application étiquetant ces opérations comme requis aura le même résultat que sur une mémoire à cohérence séquentielle.*

Exemple 7.3 *Soit un entier répliqué muni des quatre opérations inc, dec, read et write.*

Considérons, tout d'abord, l'étiquetage où les opérations read sont étiquetées `weak` *et toutes les autres* `strong`*. Avec cette étiquetage, le contrat développé dans l'exemple précédent assure que toutes les opérations conflictuelles (non commutatives) sont ordonnées de la même façon sur tous les sites. Clairement, ceci suffit à assurer un contrat équivalent à celui d'un entier sur une mémoire séquentielle.*

Considérons, maintenant, l'étiquetage où les opérations inc et dec sont étiquetées `weak` *et les opérations read et write sont étiquetées* `strong`*. Pour les mêmes raisons que précédemment, ordonnancement identique des opérations non commutatives, le contrat précédent assure la correction de l'application.*

L'exemple 7.3 illustre un point particulier : la construction d'un prédicat $\mathcal{P}_{l_2}$, local, n'est pas unique. Dans cet exemple, l'étiquetage le plus adapté dépend des fréquences relatives des opérations dans l'exécution. il faut étiqueter `weak`, le groupe d'opérations le plus fréquent. LEONG et AGRAWAL dans [LA92] ont étudié ce phénomène sur deux types d'information : un compteur, identique à celui présenté dans l'exemple et un ensemble. Ces deux auteurs ont proposé deux algorithmes paramétrables par un ensemble d'opérations, qu'ils dénomment, «core set». Ces algorithmes se basent, l'un sur la relation «forward commutativity» définie par WEIHL dans [Wei88] et le second sur la relation «invalidated by» définie par HERLIHY et WEIHL dans [HW91] ; ces deux relations ont été présentées brièvement au chapitre 3. Le «core set» identifie l'ensemble des opérations qui, dans un cas, commutent entre elles, et dans l'autre, peuvent invalider une opération précédemment exécutée.

Exemple 7.4 *Utilisons maintenant l'entier répliqué de l'exemple 7.3 pour l'application de gestion de ressource avec borne supérieure de divergence nulle sur les places libres (cf. 6.5.1). Si l'on étiquette comme* `weak` *les opérations read, la correction de l'application est assurée par la cohérence hybride, car les opérations dec sont ordonnées totalement ; cette correction se fait au prix d'un ordonnancement total des inc. Par contre, l'étiquetage* `weak` *des opérations inc et dec, n'assure plus la correction de l'application. La correction de l'application implique que les opérations dec soit totalement ordonnées. En utilisant les mêmes prédicats que dans les exemples précédents, l'étiquetage* `weak` *des opérations inc et* `strong` *de toutes les autres assure la correction de l'application.*

Exemple 7.5 *Raffinons l'exemple précédant afin de supporter une borne supérieure de divergence non nulle. Nous reprenons les spécifications présentées dans la section 3.2.2. Rappelons le critère de correction de l'application :* $\forall \Delta_{C^{c_i}} = (A_j)_{j\in[1\cdots N-1]}$ *tel que* $c_i \in \mathcal{D}, \forall i \in [1\cdots N-1], \mu(A_i) \leq K$.

Soit $\mathcal{P}^i_{l_1}(\tilde{\mathcal{H}}, op)$ *satisfait pour tout* $\tilde{\mathcal{H}} \bullet op$ *légale sur* p_i. *Soit* $\mathcal{P}_g$ *tel que* $\mathcal{P}^i_{l_2}$ *est faux dès qu'un processus exécute plus de* K *opérations dec non acquittées. Un gestionnaire de cohérence satisfaisant au critère* ***Core*** *garantit le critère de correction de l'application. Un tel gestionnaire de cohérence peut être implanté en maintenant une fenêtre d'acquittement dont la taille est définie par la borne de divergence* K *et construite en utilisant la mesure des opérations non acquittées.*

7.5 Principes Core

Nous pouvons maintenant résumer les principes de construction de **Core** de la manière suivante :

(P_1) $\mathcal{P}_{l_1}$ et $\mathcal{P}_{l_2}$ doivent pouvoir être évalués localement.

(P_2) Le contrat de cohérence doit représenté $\mathcal{P}_g$. De P_1, on déduit trivialement que $\mathcal{P}_g$ est le seul prédicat dont le calcul soit global.

(P_3) La *mise en œuvre* du contrat qui assure $\mathcal{P}_g$ doit être indépendante de l'application.

C'est l'application de ces trois principes qui amène à la construction des gestionnaires de cohérence de l'architecture **Core**.

Les gestionnaires de cohérence de l'architecture **Core** implantent le critère **Core**. Chaque instance d'un gestionnaire réalise, conceptuellement, un prédicat $\mathcal{P}_g$ particulier. Ces gestionnaires sont paramétrés, au moment de leur instanciation, par les prédicats $\mathcal{P}_{l_1}$ et $\mathcal{P}_{l_2}$ dont la mise en œuvre dépend de l'application.

Toute la difficulté de la mise en œuvre repose, en particulier, sur la mise en application du troisième principe, P_3. Ce point fait l'objet du chapitre suivant.

7.6 Résumé

Nous avons montré, pas à pas, le raisonnement qui nous a amené à décomposer de façon «générique», certain type de critère de correction, ceux construit sur les coupes de borne, selon le critère **Core**. Ce critère est construit comme la combinaison de trois prédicats :

- $\mathcal{P}_{l_1}$ s'assure de la correction «locale» de l'exécution d'un processus, i.e. relativement à toute les opérations perçues «localement» par ce processus.
- $\mathcal{P}_{l_2}$ contrôle quand la perception des opérations distantes est nécessaire.
- $\mathcal{P}_g$ offre des garanties sur l'ordre (ou une fonction de l'ordre) dans lequel les opérations distantes sont perçues.

Bien que ce critère ne concerne que la correction des critères s'exprimant sur les seules coupes de borne, donc typiquement les critère de divergence (cohérence atomique et cohérence séquentielle comprise), nous conjecturons que cette décomposition peut s'étendre à d'autres critères de corrections tel que la cohérence causale.

Les gestionnaires de cohérence de l'architecture **Core** implante ce critère. Chaque instance d'un gestionnaire réalise, conceptuellement, un prédicat $\mathcal{P}_g$ particulier. Ces gestionnaires sont

paramétrés, au moment de leur instanciation, par les prédicats $\mathcal{P}_{l_1}$ et $\mathcal{P}_{l_2}$ dont la mise en œuvre dépend de l'application.

C'est la possibilité de décomposer les contraintes de correction sous la forme du critère **Core** qui conditionne le fait de pouvoir construire un composant qui, tout à la fois, présente un contrat de cohérence déterministe, adapté à des applications particulières, et qui soit réutilisable.

Ces principes de décomposition ne garantissent *pas* l'optimalité de ce composant, ils permettent seulement à un tel composant d'exister préalablement à l'application tout en lui étant adapté.

Chapitre 8

Factorisation de la gestion de cohérence

Dans ce chapitre, nous présentons les principes permettant de factoriser la gestion de cohérence dans un composant réutilisable, le *gestionnaire de cohérence*. Nous spécifions le rôle et l'interface des divers composants résultant de cette factorisation et les contraintes que leur utilisation impose au programmeur d'application.

8.1 Principe

Nous distinguons quatre aspects dans la mise en œuvre de la gestion de la cohérence :

1. la perception locale des opérations,
2. la décision de l'ordonnancement de ces opérations,
3. la diffusion de cette décision (y compris, la diffusion de l'opération, elle-même),
4. les mises en œuvre locales de cette décision (y compris l'exécution locale sur le réplicat).

Notre principe conducteur est de laisser les fonctions 2 et 3 à la charge du gestionnaire de cohérence et les autres (1 et 4) à la charge de l'application. Ce principe se fonde sur le fait que les fonctions 2 et 3 sont les seules ayant trait à la répartition et à la réplication, et que les deux autres concentrent l'ensemble des problèmes de *mise en œuvre* spécifique du code applicatif.

Informellement, un gestionnaire de cohérence **Core** gouverne la perception répartie des opérations et sur cette base, décide de l'ordonnancement des opérations. Ce rôle correspond à celui joué par les ordonnanceurs de moniteurs transactionnels. Formellement, un gestionnaire de cohérence implante le critère **Core** et le prédicat $\mathcal{P}_g$, les deux autres prédicats $\mathcal{P}_{l_1}$ et $\mathcal{P}_{l_2}$ lui étant fourni par l'application.

Selon cette décomposition, le gestionnaire de cohérence s'interface à trois fonctions laissées à la charge de l'application, bien que nécessaire à son bon fonctionnement :

1. la fonction de perception des accès qui contrôle les accès du processus à son réplicat local,
2. la fonction d'emballage/déballage qui permet de préparer la diffusion des opérations vers les membres distant, et
3. la fonction locale de contrôle du flot d'exécution du processus, qui représente l'application de la décision de l'ordonnancement.

Pouvoir s'interfacer à ces trois fonctions, indépendamment de leurs mises en œuvre particulières, conditionne le fait de pouvoir *réutiliser* le gestionnaire de cohérence. Nous proposons d'assurer cette indépendance en appliquant les trois principes suivants :

1. L'application doit abstraire pour chaque opération, ses caractéristiques opératoires et les réifier dans un composant que nous nommons *intention*[1]. Idéalement, une *intention* conserve d'une opération, toute l'information pertinente pour son ordonnancement. Ces propriétés peuvent dépendre aussi bien d'information statique (type de l'opération) que d'information dynamique (paramètres d'invocation d'une méthode, indice d'un tableau). Plus l'information contenue dans cet objet est riche, en terme de sémantique, plus grande sont les chances d'optimisation de l'ordonnancement. La table de compatibilité de verrou, ou la réalisation du prédicat de contrôle de concurrence granulaire sont le genre d'information que doit implanter cet objet.

   Cette intention est utilisée, indirectement par le gestionnaire de cohérence, au travers des mises en œuvre du prédicat $\mathcal{P}_{l_1}$ et $\mathcal{P}_{l_2}$.[2]
2. L'application doit abstraire pour chaque opération, le contrôle de son exécution, et le réifier dans un composant que nous nommons, *activité*. Une *activité* est une structure système représentant l'exécution d'une opération sur un réplicat particulier au niveau du gestionnaire de cohérence. Une activité est implantée comme une transaction, sans propriétés spécifiques, d'un processus sur son réplicat.
3. L'application doit réaliser l'emballage de ses opérations, ce de façon transparente pour le gestionnaire de cohérence. Les opérations sont transmises sous forme de type de données opaque. Déléguer l'emballage pose des problèmes qui seront abordées dans la section 8.3.

Avant de rentrer dans le détail de ces composants, nous présentons rapidement (figure 8.1) comment ils sont typiquement utilisés par une application. Une présentation plus détaillé sera effectué dans le chapitre suivant.

Dans un premier temps, l'application instancie un gestionnaire de cohérence et lui confie le contrôle de l'ordonnancement des opérations. C'est la phase de liaison. Les autres étapes, énumérées de façon chronologique, correspondent aux actions typiquement mises en œuvre pour *chaque* accès sur la donnée répliquée :

1. L'application identifie les caractéristiques de son accès et en fait un objet de première classe en instanciant une *intention*.
2. L'application crée une activité en lui passant l'intention caractérisant l'accès et le gestionnaire de cohérence en charge de la cohérence.
3. L'application *initie* l'opération (`activity.begin()`). Dans une mise en œuvre pessimiste cette opération est typiquement bloquante.
4. L'application accède au réplicat.
5. L'application emballe puis transmet au gestionnaire de cohérence (`activity.update()`), les informations nécessaires à la mise à jour des réplicats distants. La forme choisie pour l'emballage est libre, pour peu que celle-ci soit sémantiquement équivalente à ce qui figure dans l'*intention*. Ce point est détaillé dans la section 8.3.2.

[1] Ce terme ne recouvre précisément, ni la notion d'intention introduite dans la construction d'objets atomiques par WEIHL [Wei84], ni la notion d'intention introduite dans les variantes conservatives des ordonnanceurs, en particulier ceux basés sur le verrouillage à deux phases qui permettent d'éviter l'apparition de dead-lock et donc de mécanisme de retour-arrière, spécifiquement pour l'isolation.

[2] Dans le prototype réalisé, c'est l'intention elle même qui réalise ces prédicats, ce qui induit des limitations sur la forme de ces prédicats.

```
 (* Création d'un gestionnaire *)
aGC <- Core.CreateGC("Nom du Gestionnaire");
        .....
 (* Création d'une Intention *)
(1) intention <- Intention.Create(<Type>, <arguments>);
 (* Création d'une activité *)
(2) activity <- Activity.Create(aGC, intention);

 (* Interaction avec le gestionnaire de cohérence *)
(3) activity.begin();
                {
                  aGC.beginActivity(activity);
                                {(upcall) activity._begin()}
                }

(4) (* Les accès au réplicat se font ici *)

(5) activity.update(<Opération emballée>);
                {
                  aGC.update(activity, <Opération emballée>)
                }
(6) activity.end();
                {
                  aGC.end(activity)
                                {(upcall) activity._end()}
                }
```

FIG. 8.1: Interaction avec le gestionnaire de cohérence. Les accolades montrent les actions «types» exécutées, en interne, par une activité et par le gestionnaire de cohérence.

---

6. L'application termine son opération (`activity.end()`).

Le gestionnaire permet de nombreuses variantes mais toutes doivent respecter la condition suivante : l'accès à la donnée ne peut se faire qu'après que l'activité ait été ordonnée (`Activity._begin()`) par le gestionnaire, et ne peut se continuer après que la fin de l'activité (`Activity._end()`) lui en ait été notifiée.

La section qui suit détaille le rôle et l'interface de ces composants.

8.2 Les acteurs de l'ordonnancement

8.2.1 Activité

La raison de l'activité est de pouvoir séparer la décision de l'ordonnancement, qui définit les règles de correction de l'application, de l'exécution de cette décision qui dépend souvent d'aspects proprement applicatifs. Citons parmi les plus aspects significatifs : le caractère synchrone ou asynchrone des invocations, et l'exécution de l'application dans un cadre transactionnel.

Une activité comprend donc une interface qui permet de changer, de façon abstraite, l'état d'exécution de l'application et ce de façon répartie : une activité est un objet qui peut être transféré sur des sites distants et dont la mise en œuvre est isolé de la répartition. Une activité constitue une réification du flot d'exécution de l'application accédant à la donnée.

Une activité peut être dans l'un des cinq états suivants : `initialisée`, `active`, `suspendue`, `terminée` ou `abandonnée`.

Une activité nouvellement crée est dans l'état `initialisée`. L'état `active` identifie le seul état durant lequel l'application peut accéder à la donnée répliquée. Les changements d'états sont effectués par le gestionnaire de cohérence, en invoquant l'une des cinq méthodes de l'interface ascendante («upcall») d'une activité définit par **Core** (Cf. figure 8.2) :

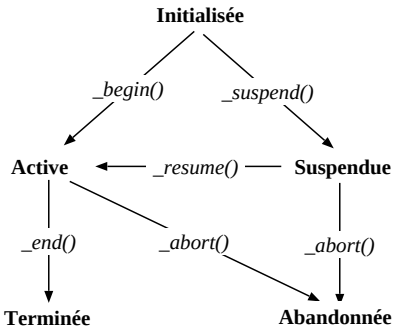


FIG. 8.2: Les cinq états que peut prendre une activité sont en gras. Les transitions sont étiquetées par le nom de la méthode de l'activité qui la déclenche. Ces méthodes définissent l'interface d'une activité.

`_begin()` : autorise les accès à la donnée répliquée.

`_suspend(activity)` : est appelé lorsque l'activité ne peut être exécutée dans le cadre du contrat de cohérence. Le paramètre passé est la première des activités (opérations) incompatible avec celle demandant l'autorisation d'accéder la donnée répliquée.

`_resume()` : autorise les accès à la donnée répliquée, après que ceux-ci aient été retardés.

`_end()` : acquitte la fin de l'activité.

`_abort()` : acquitte l'abandon de l'activité.

Bien que la mise en œuvre des méthodes ne soit volontairement pas spécifiée, **Core** impose deux contraintes sur celle-ci :

- L'accès à la donnée répliquée ne doit se faire que lorsque l'activité correspondante est dans l'état `active`.
- L'exécution *locale* des activités doit être sérialisable. Cette garantie permet de considérer une activité comme une opération exécutée localement de façon atomique, et une histoire locale comme l'exécution séquentielle des opérations ayant accédé aux données partagées (l'ordre de sérialisation local des activités). Ceci assure la conformité de l'architecture avec le modèle d'exécution de **Core**. Ceci permet d'exprimer le contrat de cohérence dans le formalisme **Core**.

8.2.2 Intention

La raison de l'intention est d'isoler les propriétés de cohérence de la donnée répliquée, nécessaires à l'implantation du contrat de cohérence, d'autres aspects, tels sa sémantique, qui sont la raison d'être de la donnée.

Le gestionnaire de cohérence, proprement dit, n'interagit pas avec une intention. Un gestionnaire a seulement besoin de pouvoir passer les intentions aux objets locaux ou distants, implantant les prédicats $\mathcal{P}_{l_1}$ et $\mathcal{P}_{l_2}$. Néanmoins, certaines connaissances sur la topologie de l'accès sont indispensables pour obtenir de bonnes performances. Malheureusement ces connaissances dépendent du type de contrat implanté par le gestionnaire de cohérence.

Nous proposons un modèle de représentation des opérations qui puisse satisfaire les modèles de cohérence présentés en première partie de cette thèse, et en particulier, ceux basés sur l'ordre

des opérations, ceux basés sur une fonction de cet ordre (métrique), ou ceux faisant appel à une combinaison des deux.

Ce modèle distingue trois dimensions dans une opération, chacune identifiant une propriété utile au contrôle de cohérence :

1. L'agent initiateur de l'opération. Plusieurs flux de contrôle peuvent partager le même identifiant. Ceci peut être utilisé dans les applications interactives où les opérations d'un même agent peuvent s'exécuter dans des flots d'exécution distincts [HJT+93]. Les identifiants des effecteurs d'une opération peuvent être de simples scalaires (identificateurs traditionnels des transactions plates) ou être des instances d'un espace plus complexe permettant d'exprimer, par exemple, une hiérarchie ; une telle hiérarchie peut servir à représenter les relations ascendant/descendant d'activités emboîtées ou bien une hiérarchie de niveaux applicatifs.
2. Le type de l'opération. Il peut s'agir d'un simple typage statique (lecture/écriture, addition/soustraction), d'un typage utilisant des paramètres de l'opération (arguments) ou plus généralement d'une fermeture (« closure ») de l'exécution locale. Cette dernière peut être quelconque, dans la limite où elle peut être évaluée sur les processus distants.
3. La donnée utilisée par l'opération. Il peut s'agir aussi bien d'une partie de l'objet : **Core** ne fait pas de différence entre une donnée implémentée comme un tout, i.e. un objet, ou une collection de scalaires (e.g. une mémoire).

Dans chacune de ces dimensions, une opération définit un domaine. La mise en œuvre du prédicat $\mathcal{P}_{l_1}$ peut se construire de façon géométrique en fonction de l'intersection sur chaque dimension des domaines associés aux intentions (opérations) [Pap86, chapitre 6].

Core paramètre le prédicat $\mathcal{P}_{l_1}$ avec les dimensions qui doivent être prises en compte dans le calcul du prédicat (cf. section 9.1.3.2).

En terme de performance, la raison essentielle de cette modélisation est de pouvoir « mettre en cache » des privilèges logiques. Par exemple, l'acquisition d'un jeton couvrant des droits en écriture sur une partie de l'arborescence d'un document. Une fois le jeton acquis, le mandataire local du gestionnaire de cohérence peut garantir seul, i.e. de façon locale, des propriétés globales sur l'ordre des accès à cette partie du document.

8.2.3 Gestionnaire de cohérence

Un gestionnaire de cohérence **Core** assure trois fonctions :

- (« group membership ») : la gestion du groupe des processus constituant l'application.
- (Communication de groupe) : la propagation des opérations, que lui soumet chaque membre, à l'ensemble des autres membres.
- (Ordonnancement) : la décision de l'exécution des opérations qui lui sont soumises par les membres.

Dans une mise en œuvre pessimiste, les deux premières fonctions sont assurées traditionnellement par les plate-formes de communication de groupe [Sec96]. Les requis que le gestionnaire pose sur ces deux fonctions, en particulier l'ordre dans lequel sont perçus l'arrivée de nouveaux membres, l'initialisation de ceux-ci et les messages échangés par les membres, dépendent du contrat de cohérence. Nous reviendrons sur ces fonctions au chapitre suivant, lors de la description de l'architecture. Le point essentiel est que ces fonctions étant *in fine* responsables de

la transmission des opérations entre processus, elles doivent être pilotées par l'ordonnanceur pour assurer une mise en œuvre efficace [CD89].

La fonction d'ordonnancement est la seule fonction perçue par l'utilisateur de **Core** ; elle encapsule tout ce qui a trait à la communication entre les membres. C'est son architecture que nous détaillons maintenant.

Le gestionnaire de cohérence reçoit à sa création deux objets implantant l'un, $\mathcal{P}_{l_1}$, le prédicat local de « plausibilité », l'autre, $\mathcal{P}_{l_2}$, le prédicat local implantant l'autonomie de décision du mandataire local du gestionnaire de cohérence.

Le prédicat de « plausibilité », $\mathcal{P}_{l_1}$, a un rôle similaire à celui du prédicat implantant les règles de conflits dans les verrous à prédicat. Il est invoqué avec trois paramètres :

- l'histoire du processus,
- l'intention associée à l'activité pour laquelle le gestionnaire doit prendre une décision d'ordonnancement, et
- un filtre identifiant quelles dimensions de l'intention doivent être prises en compte.

Le prédicat $\mathcal{P}_{l_2}$ est appelé avec :

- l'histoire du processus, et
- l'intention associée à l'activité pour laquelle le gestionnaire doit prendre une décision d'ordonnancement.

En pratique, l'histoire locale peut être représentée par un pointeur sur le réplicat local.

Core impose une condition sur la mise en œuvre de ces prédicats : le respect du critère **Core** (cf. définition 7.2, page 81) doit suffire à garantir la correction de l'application.

L'application interagit avec le gestionnaire de cohérence selon deux modes :

- En tant que client, en lui soumettant des opérations à ordonner. Cette interaction se fait au travers de l'interface des appels descendants (« downcalls ») du gestionnaire.
- En tant que serveur, d'une part lorsque le gestionnaire invoque les prédicats et notifie l'application de la décision d'ordonnancement, d'autre part lorsque le gestionnaire lui soumet une opération, initiée par un membre distant, devant être appliquée sur le réplicat local. L'aspect ordonnancement s'effectue au travers de l'interface des prédicats, des intentions et des activités, l'autre aspect au moyen de l'interface des appels ascendants (« upcalls ») de **Core**.

Core définit quatre appels descendants :

`beginActivity(activity)`	: soumet l'activité au gestionnaire.
`endActivity(activity)`	: notifie la fin de l'activité au gestionnaire.
`abortActivity(activity)`	: notifie l'abandon de l'activité au gestionnaire.
`update(updateMessage, activity)`	: soumet les opérations devant être effectués sur les réplicats distants.

Tous les appels ascendants du gestionnaire de cohérence sont construit comme des notifications d'événements. Un gestionnaire de cohérence génère deux types d'événements :

`UpdateEvent` : encapsule les opérations initiées par un membre distant. Une instance de cet événement contient ce qui est nécessaire à l'application pour mettre à jour son réplicat.

`InitEvent` : correspond à une requête d'initialisation d'un nouveau membre. Les contraintes d'ordre entre l'arrivée des membres et les opérations effectuées dépendent du contrat de cohérence. Par exemple, dans le cas d'un modèle assurant une synchronie virtuelle forte (« strong virtual synchrony »), c'est le gestionnaire qui s'assure que l'arrivée d'un membre et son initialisation sont perçues comme une opération atomique.

La méthode `subscribe()` du gestionnaire, permet à l'application de souscrire[3] aux événements qu'elle peut gérer. L'application souscrit à un événement en invoquant `subscribe()` avec en paramètre un objet (« handler ») prenant en charge l'événement au niveau de l'application.

8.3 L'exécution répartie des opérations

Core possède les contraintes propres au contrôle de concurrence abstrait. Un gestionnaire de cohérence ne contrôlant pas la partie de la représentation de l'information *réellement* accédée, **Core** impose une contrainte sur la mise en œuvre du code applicatif chargé de l'exécution de l'accès sur le réplicat local. Cette contrainte est triviale : l'accès sur ce réplicat doit avoir les mêmes propriétés que son abstraction, l'*intention*, utilisée par le gestionnaire pour décider de son ordonnancement.

L'exécution d'une opération répliquée comporte deux aspects : l'invocation locale sur le processus ou a été initié l'opération et la mise à jour distante. La contrainte dérivant de l'ordonnancement abstrait a un impact sur ces deux aspects :

- sur l'invocation locale sur le réplicat, i.e. sur la mise en œuvre de l'opération elle-même,
- sur l'emballage de l'opération qui détermine, pour une part, la sémantique de la mise à jour distante.

Cette contrainte peut être illustrée simplement, au moyen des deux exemples suivants :

Exemple 8.1 *Reprenons l'entier muni des deux opérations inc et dec utilisées dans la gestion de ressource (cf. section 6.5.1). Ces deux opérations commutent entre elles. Supposons que l'application implante chacune de ces opérations comme une paire d'opérations consécutives lecture• écriture. La correction de l'application impose que cette paire d'accès soit exécutée de façon atomique*[4].

Exemple 8.2 *Supposons que l'information partagée soit représentée par un nombre complexe. Le* module *d'un nombre complexe est indépendant de son* argument. *Nommons* `setModule` *(resp.* `setArgument`*), la méthode modifiant le module (resp. l'argument) d'un nombre complexe. Ces deux méthodes peuvent, a priori, s'exécuter concurremment. Ce n'est en fait pas le cas si la représentation choisie pour mettre en œuvre le complexe est la représentation cartésienne. La propriété de commutativité de ces deux opérations est, par contre, toujours maintenue.*

[3] L'application peut choisir les événements d'intérêts. Ceci donne une certaine souplesse au modèle de réplication : **Core** permet la création de réplicat « virtuel » à des fins de contrôle ou de déboggage par exemple.

[4] La sérialisibilité locale sur les activités suffit, en fait.

8.3.1 L'invocation locale sur le réplicat

Les deux exemples ci-dessus montrent que les contraintes de concurrence, sur l'exécution locale d'une opération, dépendent de la mise en œuvre du réplicat. La contrainte de sérialisibilité locale des activités (cf. 8.2.1) découle de ces contraintes de concurrence.

Core ne prend pas en compte les contraintes locales de concurrence sur le réplicat, en particulier d'exclusion mutuelle, que peuvent nécessiter certaines mises en en œuvre du réplicat. **Core** délègue à l'application la responsabilité que l'exécution locale des activités sur le réplicat donnent un résultat conforme à l'ordonnancement des intentions dans le cadre du modèle d'exécution de **Core**. Cette délégation est inhérente à l'ordonnancement abstrait et donc à la factorisation de la gestion de la cohérence. Cette indépendance est *souhaitable* ; elle possède deux bonnes propriétés :

- Elle préserve la gestion de cohérence des aspects dépendant de l'implantation de l'information partagée. Cette propriété favorise la réutilisation, tant de la gestion de cohérence que du réplicat.
- Elle traite de façon distincte les contraintes de concurrence locales, des contraintes de concurrence globales qui jouent sur des ordres de grandeur plusieurs fois supérieurs. Cette propriété se marie bien avec les principes de **Core** qui se focalisent sur l'adaptabilité des contraintes globales.

8.3.2 L'emballage des opérations

La mise à jour des réplicats distants nécessite l'encodage de l'opération sous une forme permettant sa communication aux processus accueillant les réplicats. On distingue, en pratique, deux formes d'encodages :

- L'encodage de l'opération : l'opération est transmise, avec ses arguments, aux processus distants, décodée puis réexécutée sur chacun des réplicats.
- L'encodage de l'état : l'état du réplicat, ou éventuellement la partie modifiée par l'opération, est transmis et remplace, par recopie, la partie correspondante du réplicat distant. Cette forme d'encodage est systématiquement utilisée, dans les implantations de mémoire partagée répartie.

L'exemple 8.2 montre les limitations de la seconde forme d'encodage : l'envoi de l'état aboutit à un état divergent ; ceci bien que les deux opérations soient supposées commuter et aient été exécutées sur les deux processus. L'exécution des *deux* opérations ne sera reflétée sur aucun des deux réplicats, en contradiction avec l'état abstrait construit par l'ordonnanceur. Cette incorrection découle d'un fait général : la mise à jour par copie d'état a la propriété d'absorption (cf. 3.3.1), i.e. a une sémantique d'affectation. Dans le cas général, seule la réexécution des opérations garantit la concordance entre les intentions et l'encodage des opérations.

Core laisse l'emballage à la charge de l'application. Ceci est requis pour conserver l'indépendance entre le gestionnaire de cohérence et la donnée répliquée ; cette indépendance permet la réutilisation de ce gestionnaire. La délégation de l'emballage à l'application a un autre avantage : il permet à l'application d'optimiser l'emballage en fonction de la donnée répliquée ; or cette optimisation est un facteur reconnu pour la limitation des surcoûts locaux des exécutions réparties [BNOW93].

8.3.3 De l'invalidation et de l'initialisation

La remarque précédente concernant l'encodage des opérations a un impact sur l'utilisation de l'invalidation dans le protocole de réplication. L'invalidation repose *implicitement* sur la possibilité de pouvoir recevoir un état cohérent sans passer par les états intermédiaires qui y ont conduit. Cette propriété est acquise de fait lorsque que toute les opérations modifiant la donnée ont la propriété d'absorption, i.e. se comportent comme des écritures. Dans le cas général, un processus dont le réplicat a été invalidé requiert l'ensemble ordonné des opérations qu'il n'a pas perçues pour être à jour. Vis-à-vis de la diffusion de chacune des opérations, l'invalidation devient simplement un procédé de tamponnage.

En pratique la situation est meilleure. **Core** est conçu pour gérer des groupes dynamiques [Bir96, sections 13.12.3 et 16.1] où les réplicats sont créés à la demande dès qu'un agent joint le groupe. Un tel modèle nécessite un mécanisme d'initialisation des nouveaux réplicats. Le modèle d'exécution de **Core** n'exige qu'une chose : que l'histoire locale attachée à ce nouveau réplicat soit compatible avec une exécution globale correcte. Une solution simple est de définir une opération d'initialisation absorbante, transférant la copie d'un réplicat à jour[5]. Cette opération peut être utilisée par un mécanisme à invalidation.

8.4 Core en tant que moniteur transactionnel

Core assure certaines des fonctions traditionnelles des moniteurs transactionnels. Nous comparons rapidement dans cette section, les composants de **Core** avec ces fonctions. La comparaison est faite d'une part dans le modèle abstrait de SGBD définis par Bernstein, Hadzilacos et Goodman [BHG87, section 1.4], d'autre part avec les spécifications des services de transaction et de contrôle de concurrence de CORBA [COSb, COSa].

8.4.1 Core dans le modèle abstrait de SGBD

Le modèle abstrait de système de gestion de base de données (centralisé) présenté dans [BHG87, section 1.4] comprend trois composants :

- Le gestionnaire de transaction (« transaction manager ») : il crée, termine ou abandonne les transactions, soit sur requête de l'utilisateur, soit sur requête de l'ordonnanceur. C'est le gestionnaire de transaction qui associe les opérations sur les données aux transactions.
- L'ordonnanceur (« scheduler ») : il décide de l'ordre d'exécution des opérations sur les données partagées ; ceci en contrôlant l'ordre dans lequel les opérations sont délivrées au gestionnaire de données.
- Le gestionnaire de données (« data manager ») qui lui même comprend deux modules : le gestionnaire de recouvrement (« recovery manager ») et le gestionnaire de cache (« cache manager »)). Cette décomposition ne nous concerne pas. Le gestionnaire de données maintient la représentation de la base et assure son évolution en appliquant les opérations qui lui sont soumises par l'ordonnanceur.

Dans **Core**, les transactions sont représentées par les activités. Le rôle du gestionnaire de transaction, comme celui du gestionnaire de données, est laissé à l'application, plus précisément au code contrôlant les accès à la donnée répliquée. Enfin le rôle de l'ordonnanceur est rempli

[5] Un tel réplicat peut ne pas exister dans les contrats de cohérence faible ; l'initialisation devient alors singulièrement plus complexe.

par le gestionnaire de cohérence. Les intentions ne sont pas explicitées et sont intégrées dans l'ordonnanceur.

8.4.2 Core dans le modèle CORBA

CORBA distingue la gestion du contrôle de concurrence (« Concurrency Control Service ») de la gestion des transactions (« Transaction Service »). Cette distinction permet d'utiliser le service de contrôle de concurrence en dehors d'un cadre transactionnel.[6]

Le service de contrôle de concurrence ne considère que le verrouillage à deux phases, strict [BHG87, chapitre 3]. Chaque entité accédée est représentée de façon abstraite par un objet `LockSet` gérant les verrous contrôlant cette donnée. C'est à l'application, ou à la transaction, de s'assurer que les accès à la donnée sont bien précédés d'une prise de verrou sur l'objet `LockSet` correspondant. CORBA définit les types de verrous usuels (`read`, `write`, `upgrade`, `intention_read`, `intention_write`) avec leur sémantique usuelle [GR93]. Cette interface permet, en particulier, le verrouillage granulaire [GR93, section 7.8]. CORBA définit aussi un objet `LockCoordinator` chargé de coordonner le relâchement de l'ensemble des verrous pris par une tâche ou une transaction. Cet objet assure le respect du protocole de verrouillage à deux phases.

Le service de transaction définit les composants nécessaires à la mise en œuvre de transactions ACID en réparti. CORBA permet de nombreuses formes d'interactions avec le gestionnaire de transaction. L'interaction la plus simple se fait au travers de l'interface `Current`. Cette unique interface rassemble les interfaces des services d'un moniteur transactionnel les plus fréquemment utilisés. Nous ne décrivons ici que les éléments auxquels nous limitons la comparaison avec **Core**, i.e. ceux liés au maintien de l'isolation des transactions concurrentes.

Une transaction CORBA est représentée par un objet de type `Control` ; cet objet donne accès à deux composants :

- Un objet de type `Coordinator` maintient l'ensemble des *ressources* accédées, y compris les verrous (`LockSet`). Chaque ressource accédée par une tâche peut retrouver la transaction dans le contexte de laquelle cette tâche s'exécute et s'enregistrer (`register_resource`) dans la liste des ressources accédées, maintenue par le `Coordinator` de la transaction.
- Un objet de type `Terminator` contrôle le devenir, `commit` ou `rollback`, de la transaction.

Vis-à-vis de l'isolation, un gestionnaire de cohérence **Core** assure le rôle de `LockCoordinator`. L'interface **Core** d'une activité est analogue à celle des « call-backs » de X/Open et représente l'interface `Resource` additionnée des deux méthodes `suspend` et `resume` de l'interface `Current`. Les intentions n'ont pas d'équivalent dans CORBA, si ce n'est les types de verrous définis par `lock_mode`.

Quelques remarques

- La sémantique de contrôle de concurrence de CORBA ne peut être adaptée : les verrous de CORBA ne sont pas des objets de première classe, les types et les règles de conflit sont spécifiés et fixés.

[6] Comme dans Clouds [CD89], CORBA permet à un objet d'être accédé concurremment par des transactions et par des entités (tâches, processus) n'appartenant pas à une transaction.

– Le modèle de réplication coopérante se traduit par des implantations réparties des objets `Resource` et `Coordinator`. CORBA permet de telles implantations mais ne précise pas comment les réaliser.

8.5 Core et le modèle objet

L'objet est un outil d'abstraction introduit pour concevoir et créer des systèmes complexes. Une des propriétés essentielles de l'objet est l'encapsulation : un objet « décrit » ce qu'il fait, non comment il le fait ; il encapsule ainsi sa mise en œuvre qui devient dès lors *transparente* à son utilisateur.

La gestion de la concurrence comme de la répartition impliquent des mécanismes de mise en œuvre complexes. L'encapsulation de ces mécanismes à l'intérieur d'objets partagés et/ou répartis permet de confiner cette complexité, et préserve l'utilisateur de celle-ci. L'intégration du contrôle de concurrence a donné lieu aux *objets atomiques* [Wei84, LCJS87, PSWL95]. L'intégration de la répartition a donné lieu aux *objets fragmentés* [MGlNS94].

Core intégrant la problématique de concurrence et de répartition (réplication), nous le positionnons dans cette section, vis-à-vis de ces deux axes de recherche.

8.5.1 Objets atomiques et spécifications non déterministes

Les *objets atomiques* sont une réponse à deux défis que pose un objet utilisé dans un cadre transactionnel :

– La représentation d'un objet est arbitrairement complexe ; en particulier, sa représentation peut être discontinue (graphe d'objets) et sa taille peut varier.

– La sémantique d'un objet peut être exploitée, permettant une concurrence plus importante que celle obtenue au travers de la vision d'un simple scalaire muni d'opérations de lecture et d'écriture (cf chapitre 3).

Le contrôle de concurrence et de recouvrement des objets atomiques a fait l'objet de nombreux travaux [Wei84, Wei89, Ng89, HW91, Gue92, Gue94]. Dans tout ces travaux, le type de l'objet atomique est construit en augmentant le type abstrait de la version centralisée par des propriétés de concurrence. Le résultat en est ce que Weilh dans [Wei84] nomme une spécification sérielle *non* déterministe : le résultat d'une opération ne dépend pas seulement de l'état de l'objet et de l'invocation mais aussi des invocations qui ont lieu concurremment et du contexte (transaction) dans lesquelles elles sont effectuées. Outre qu'elle répond aux deux défis préalablement définis, cette structuration, mise en œuvre dans Avalon [DHW88], a des avantages conséquents :

– Elle permet de construire un système transactionnel réparti selon une architecture modulaire simple de type client/serveur : chaque objet atomique gère sa concurrence et son recouvrement en s'appuyant sur le contexte des transactions qui les accèdent, en particulier de leur ordre de sérialisation.

– La concurrence est intégrée au type de l'objet, parachevant l'encapsulation.

Core à deux inconvénients majeur vis-à-vis des objet atomiques :

– La sémantique des objets partagés, n'est définie que partiellement par leurs représentations, i.e. les réplicats. Un objet répliqué est un « couple » réplicats/gestionnaire de

cohérence. La sémantique de cet objet n'est déterminée que lorsque le programmeur lie le réplicat, à un gestionnaire de cohérence particulier.

- Le fait d'abstraire l'ordonnancement d'une opération de sa mise en œuvre rend difficile l'utilisation de techniques de concurrence telles que l'atomicité dynamique ou le contrôle de concurrence multiversion ; ces dernières nécessitent que l'ordonnanceur puissent contrôler des versions multiples de l'objet[7]. Un tel requis, dans **Core**, reviendrait à attribuer au gestionnaire de cohérence la possibilité de gérer plusieurs histoires locales (autant que de versions).

Nous pensons que ces inconvénients sont le prix à payer pour construire des applications coopératives sur des environnements à latence élevée. En contrepartie, **Core** permet d'implanter l'information partagée à partir de composants standards du marché.

Schématiquement, les objets atomiques se focalisent sur une construction modulaire et transparente de la concurrence. **Core** se focalise sur l'adaptation et la réutilisation des composants.

8.5.2 Core et les objets fragmentés

Les *objets fragmentés* sont construits à partir de deux constatations :

- La gestion de la répartition d'une information partagée est complexe : rendons son utilisation simple en confinant cette gestion à l'intérieur d'un objet qui offre l'encapsulation de sa mise en œuvre. La répartition devient transparente aux clients de l'objet fragmenté.
- La mise en œuvre performante de la distribution d'une information partagée nécessite la prise en compte de critères applicatifs et dynamiques : offrons au programmeur les moyens de décider, à l'exécution, du type de fragment le mieux adapté.

De ces deux remarques est issu le modèle de programmation des objets fragmentés : l'information est représentée par des objets gérant eux-mêmes la répartition et la concurrence. La représentation des fragments est définie à l'exécution.

Sur le plan architectural, un objet fragmenté est constitué d'un groupe de mandataires coopérants, les *fragments* ; un fragment est un objet traditionnel, passif, incarné dans le contexte d'un processus. L'interface du fragment est celle de l'objet fragmenté [MGlNS94, Gou91].

La mise en œuvre de la représentation de l'objet partagé n'est pas spécifiée par le modèle d'objets fragmentés. En particulier, la représentation peut être centralisée, partitionnée, répliquée ou migrante.

Un point essentiel du modèle concerne la liaison d'un client/processus avec un objet fragmenté : la création locale d'un mandataire se fait par requête sur un objet spécifique, un serveur de liaison[8], appelé « factory ». Ce serveur de liaison permet de choisir à l'exécution la mise en œuvre du fragment représentant localement l'objet fragmenté.

Le mécanisme de liaison des objets fragmentés peut être utilisé pour définir, lors de l'instanciation du fragment, le gestionnaire de cohérence le mieux adapté à l'environnement d'exécution. **Core** peut donc être utilisé pour implanter des objets fragmentés répliqués.

[7] Les intentions proposée par WEIHL et HERLIHY constituent implicitement des versions de l'objet.
[8] Ce travail a été ultérieurement développé dans [Sha94, Mai96].

8.6 Résumé

Nous avons définis dans ce chapitre, des principes architecturaux qui permettent l'adaptation et la réutilisation du code de gestion de cohérence. Ces principes définissent trois composants principaux ainsi que les requis minimaux concernant leur interface :

- L'intention encapsule toutes les informations propres à une opération, nécessaires à toute décision relative à son ordonnancement.
- L'activité encapsule le modèle d'exécution. Cet encapsulation concerne le caractère synchrone ou asynchrone de l'exécution d'une opération. Elle concerne le fait qu'une opération s'exécute dans un cadre transactionnel ou dans un simple processus. Elle concerne aussi les informations que l'application peut avoir besoin de lier à l'état d'une activité, par exemple, pour gérer la conscience du partage (« cooperation awareness ») (nom ou statut de l'agent). Ces caractères applicatifs sont encapsulés, préservant la gestion de cohérence d'une dépendance qui rendrait impossible sa réutilisation.
- Le gestionnaire de cohérence encapsule la partie non déterministe de la réplication. Cette encapsulation est indépendante de la mise en œuvre de l'information partagée, mais utilise ses propriétés aussi bien que celles de l'application qui y accède.

Concernant l'utilisation de **Core** dans un cadre objet, deux traits distinguent **Core** des autres modèles à objets partagés répartis :

- La sémantique d'un objet répliqué n'est fixée qu'à l'exécution, en fonction du gestionnaire de cohérence associé au réplicat.
- Les mises en œuvre de l'information partagée et des mécanismes de réplication sont indépendantes et peuvent être réutilisées séparément.

Chapitre 9

Architecture Core

Dans ce chapitre, nous présentons l'architecture **Core** telle que mise en œuvre dans le prototype développé en C++.

Cette architecture comporte de sévères limitations comparée aux principes exposés au chapitre précédent. Les plus notables sont liées à l'implantation des prédicats $\mathcal{P}_{l_1}$ et $\mathcal{P}_{l_2}$. Dans le prototype, ces prédicats sont encapsulés dans les intentions et l'interface de celles-ci ne permettent pas de tenir compte ni de l'histoire locale du processus ni de l'état de son réplicat. *Les contrats basés sur les métriques ne peuvent donc pas être mis en œuvre dans cette implantation de l'architecture.*

Nous présentons tout d'abord une vue globale de l'architecture, montrant les relations du gestionnaire de cohérence avec l'application ainsi qu'avec les principaux composants de la plate-forme. Cette vue est précisée en présentant l'interface des principaux composants tels que mis en œuvre dans le prototype.

Nous montrons dans un deuxième temps la construction de la liaison d'un gestionnaire de cohérence avec un réplicat.

9.1 Architecture Core

La figure 9.1 montre les éléments principaux de l'architecture et leurs relations. Les seuls composants pertinents, en ce qui concerne l'application (zone grisé «code applicatif»), sont les trois composants du contrôle d'accès. Un coup d'œil comparatif avec la figure 6.2 montre que l'ensemble « plate-forme–gestionnaire de cohérence » assure le protocole de coopération de l'application répartie.

Nous présentons successivement les trois grandes zones de la figure : la plate-forme, le gestionnaire de cohérence puis le code applicatif.

9.1.1 La plate-forme

Le prototype a été développé au-dessus d'Unix. Tous les composants décrits sont implantés comme des classes C++.

La mise en œuvre et l'utilisation d'un gestionnaire de cohérence s'appuie sur trois services du prototype : un service de nommage, un service de communication de groupe et un service d'emballage/déballage.

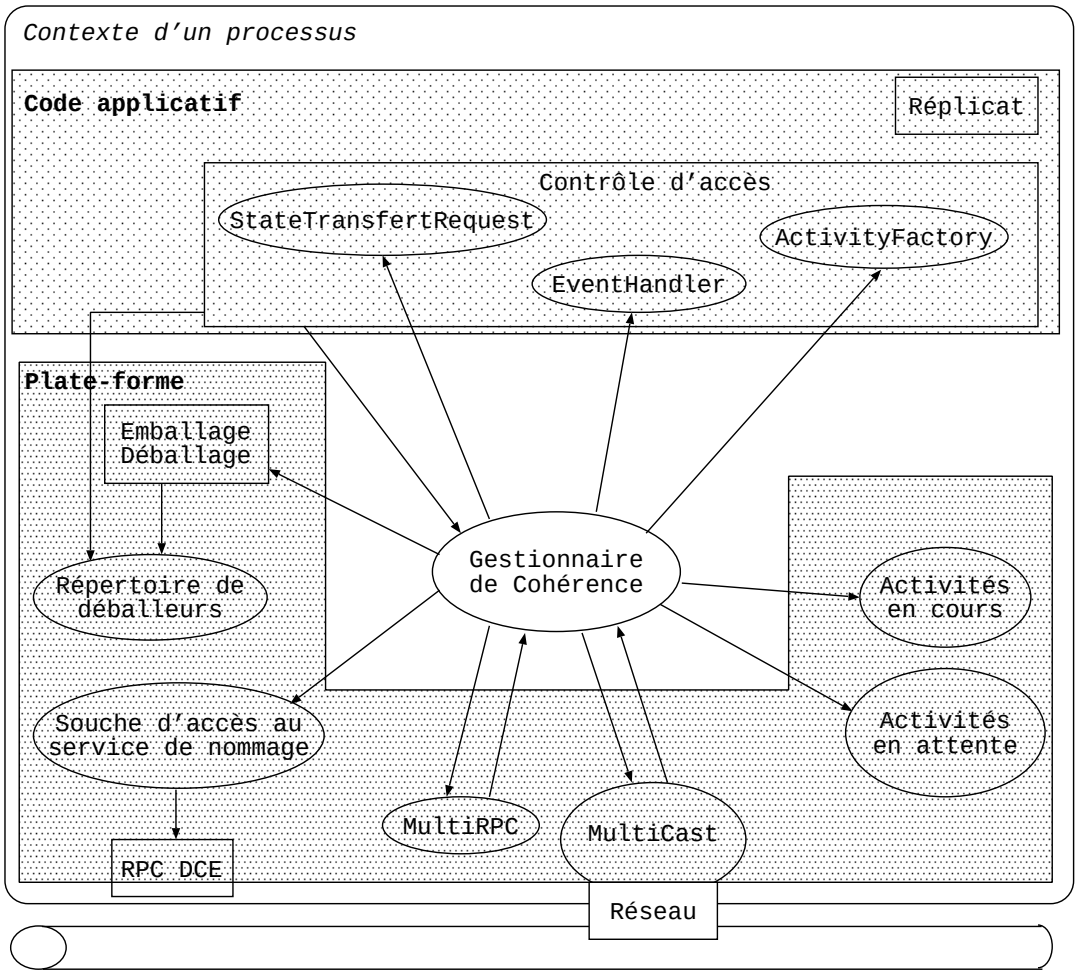


FIG. 9.1: Architecture **Core** au niveau d'un processus. Les ovales sont des instances d'objets, les rectangles représentent des fonctions dont ni la mise en œuvre ni l'interface ne sont spécifiées. Les flèches correspondent aux références que maintiennent entre eux les différentes entités. Le gestionnaire de cohérence encapsule toute la partie répartition à l'exception du déballage.

Le service de nommage Le service de nommage est spécialisé pour le support de la communication de groupe : il maintient des listes d'associations (`<identificateur du groupe> [<adresse IP> <port TCP>]+`).

L'identificateur de groupe est une chaîne ASCII identifiant de façon unique le groupe. Chaque couple `[<adresse IP> <port TCP>]` représente les coordonnées d'un membre de ce groupe.

Le service de nommage est assuré par un serveur centralisé implanté à l'aide des RPC DCE. Ce serveur maintient les listes d'associations de façon persistante afin de faciliter le déboggage des applications.

Trois méthodes `lookup`, `register` et `unregister` sont fournies.

La fonction `register` enregistre les coordonnées d'un nouveau membre et retourne à ce dernier, sous la forme d'une liste de couple `[<adresse IP> <port TCP>]*`, les coordonnées

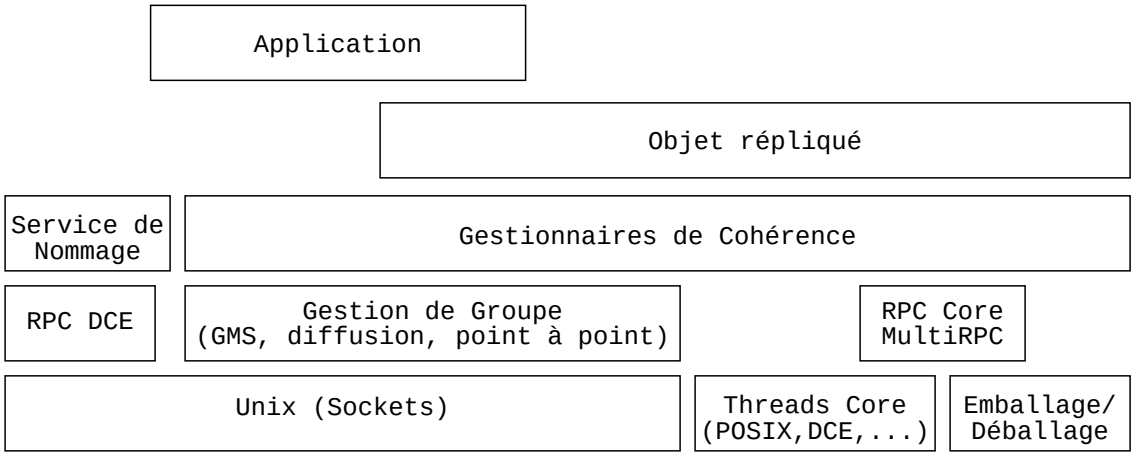


FIG. 9.2: Plate-forme prototype **Core**. Cette plate-forme n'est pas structurée en couches comme le laisse croire la figure. Elle consiste en une collection de services implantés sous forme de classes C++, utilisables directement par l'application. L'application peut s'interfacer directement avec un gestionnaire de cohérence sans utiliser les composants de **Core** utilisés pour la construction des objets répliqués.

---

des membres précédents enregistrés. Cette dernière propriété est essentielle pour la gestion du groupe. Elle permet d'ordonner totalement les enregistrements des membres, et rend ainsi aisée la constitution d'un maillage complet (toutes les communications sont effectuées au-dessus de TCP) entre les membres.

Le service de communication de groupe La collection d'agents coopérants est gérée comme un groupe. Cette structure de groupe se reflète récursivement au niveau des réplicats, des gestionnaires de cohérence et des embouts de la communication de groupe. A chaque agent correspond un réplicat, un gestionnaire de cohérence et un embout de communication.

La gestion de groupe fournie par la plate-forme assure la gestion de la composition du groupe (« group membership »), trois primitives de diffusions (*abcast*,*fbcast* et *gbcast*) et une primitives de communication de point à point. Cette plate-forme n'est pas virtuellement synchrone : en particulier, les nouveaux membres peuvent voir certaines diffusions initiées avant leur arrivée dans le groupe.

Les appels ascendants, correspondant à l'occurence d'événements asynchrones (délivrance de message, changement de composition du groupe), sont implantés sous forme de notifications. L'application souscrit à différents types d'événements. Chaque souscription s'accompagne du passage d'une référence sur un objet (`EventHandler`). L'occurence d'un événement pour lequel l'application a souscrit, génère un invocation sur cet objet avec l'événement notifié correspondant, passé en paramètre. Ce point est revu dans la section suivante et illustré dans la partie droite de la figure 9.3.

Le choix du développement d'une plate-forme de communication de groupe s'est faite pour des raisons historiques, mais toute autre plate-forme, tel Ensemble/Maestro, par exemple, pourrait être utilisée.

Le service d'emballage/déballage Les services d'emballage et de déballage du prototype gèrent un répertoire de type extensible. A chaque type d'objet est associé un identificateur unique. Ce répertoire est constitué d'un simple fichier utilisé à la compilation.

L'emballage et le déballage sont à la charge du programmeur de tout objet désirant être migré. Le prototype implante l'emballage de tous les types de bases C++. Le gestionnaire de déballage de **Core** gère dynamiquement la liste des types qu'il connaît. Le service de déballage maintient un ensemble d'associations `([TypeID <Déballeur>])`.

L'application et le gestionnaire de cohérence doivent enregistrer à l'exécution et avant la liaison, tous les types d'objets susceptibles d'être déballés par les gestionnaires de cohérence. Ceci est impératif pour tous les objets tels que les activités ou les intentions qui sont utilisés et donc déballés par les gestionnaires de cohérence sans que ces derniers ne connaissent leur type exact.

9.1.2 Le gestionnaire de cohérence

Un gestionnaire de cohérence est un composant réparti. Ce composant est implanté comme une collection de mandataires constitués en groupe. Un groupe est créé implicitement lors de l'instanciation du premier mandataire. La création d'un groupe consiste en deux étapes :

1. l'instanciation du premier mandataire,
2. l'enregistrement de ce mandataire (nom du groupe, site et port de connexion) dans le service de nom de la plate-forme. Le nom du groupe est une simple chaîne ASCII. C'est à l'application de s'assurer que ce nom ne rentre pas en conflit avec un autre groupe.

Notre modèle de *réplication coopérante* impose à chaque processus désirant accéder aux données répliquées d'instancier dans son contexte un mandataire. Tous les mandataires d'un même groupe, doivent être des instances de la même classe. C'est cette classe qui identifie le comportement du gestionnaire de cohérence et donc le contrat de cohérence. Le composant intitulé « gestionnaire de cohérence » ne représente donc que le mandataire local du processus considéré.

La figure 9.3 montre la classe abstraite du prototype qui définit l'interface d'un gestionnaire de cohérence. On distingue deux interfaces :

- Une interface « descendante », correspondant aux invocations synchrones de l'application sur le gestionnaire de cohérence. Il s'agit :
  - des déclarations de début (`beginActivity`), de fin (`endActivity`) et d'abandon (`abortActivity`) d'activité,
  - des mises à jours du réplicat (`update`), et
  - de la demande de liaison/initialisation (`init`).
- Une interface « ascendante » (« upcalls »), correspondant à l'occurence d'événements asynchrones. Ces événements sont de quatre types :
  - mises à jour (classe `UpdateEvent`). Ces événements correspondent à un appel de la méthode `update` d'un mandataire distant du gestionnaire de cohérence.
  - réception d'une opération d'initialisation (classe `InitEvent`). Cet événement est généré, de façon asynchrone par le gestionnaire, après l'appel de la méthode `init` de ce même mandataire, dans la cadre de la liaison.

```
class StateTransfertRequest {
public :
    virtual MBuff* invoke () = 0;
};

extern const KindOfEvent CM_UPDATE_EVENT;
extern const KindOfEvent CM_INIT_EVENT;

class Coherence : public SmartNotifier {
protected:
    const OFname      theOFname;
    Predicate         initialised;
    StateTransfertRequest *stateTransfertRequest;
    ActivityFactory *activityFactory;

public:

    class UpdateEvent : public MBuffEvent {
    public:
        UpdateEvent (MBuffEvent &mb) : MBuffEvent(mb) {}
        virtual const KindOfEvent& kind() const {return CM_UPDATE_EVENT;}
    };


    class InitEvent : public MBuffEvent {
    public:
        InitEvent (MBuffEvent &mb) : MBuffEvent(mb) {}
        virtual const KindOfEvent& kind() const {return CM_INIT_EVENT;}
    };


    Coherence (const OFname of, StateTransfertRequest *str, ActivityFactory *af)
        : theOFname(of),
          stateTransfertRequest(str),
          activityFactory(af),
          initialised(false)
        {}
    virtual ~Coherence (){}

    virtual const Id& getId         ()          = 0;
    virtual void      init          (ActivityCMI&, ActivityFactory* = NULL) = 0;
    virtual void      beginActivity (ActivityCMI&)  = 0;
    virtual void      endActivity   (ActivityCMI&)  = 0;
    virtual void      abortActivity (ActivityCMI&)  = 0;
    virtual void      update        (MBuff&, ActivityCMI&)  = 0;
};
```

```
/*
 * 'Event' are the basic event type passed to EventHandlers. They may
 * be extended to encapsulate context dependent of the current use of
 * the Notifier mechanism.
 */
class Event {};
extern const Event voidEvent; // ONLY this very instance is considered void.

class EventHandler {
public:
    virtual void invoke(Event&) = 0;
};

class KindOfEvent {
public:
    virtual bool operator <  (const KindOfEvent&) const;
    virtual bool operator == (const KindOfEvent&) const;
};
extern const KindOfEvent voidKindOfEvent;

class SmartEvent : public Event {
public:
    virtual const KindOfEvent& kind() const;
};
extern const SmartEvent voidSmartEvent;

class SmartNotifier {
public:
    SmartNotifier();
    ~SmartNotifier();

    virtual void subscribe   (EventHandler&,
                              const KindOfEvent& = voidKindOfEvent);

    virtual void unsubscribe (EventHandler&,
                              const KindOfEvent& = voidKindOfEvent);

    void         notify      (SmartEvent& = voidSmartEvent);
    void         waitForNotification ();

    unsigned int nbKind () const; // number of notifying class
    unsigned int size    (const KindOfEvent& = voidKindOfEvent);
    bool         empty   (const KindOfEvent& = voidKindOfEvent);

      // ... implementation ...
};
```

FIG. 9.3: Interface du gestionnaire de cohérence. La partie gauche présente l'interface des appels ascendants synchrones ainsi que les types d'événements notifiés de façon asynchrone par le gestionnaire à l'application. La partie droite présente l'interface du service de notification qui permet à l'application de souscrire à ces événements. Ce service est utilisé de façon interne par **Core** aussi bien par le gestionnaire et la communication de groupe que par le service de MultiRPC.

- demande d'une opération d'initialisation. Cette dernière n'utilise pas le système de notification standard (classes `Notifier` et `SmartNotifier`) de **Core**. La demande se fait par invocation d'une instance de la classe `StateTransfertRequest` que toute application doit fournir au mandataire d'un gestionnaire au moment de sa liaison.
- réincarnation d'un activité distante. Cette dernière notification se fait par invocation sur la classe `ActivityFactory` que toute application doit fournir au mandataire d'un gestionnaire au moment de sa liaison.

Les appels ascendants correspondant à la mise en œuvre des prédicats $\mathcal{P}_{l_1}$ et $\mathcal{P}_{l_2}$ ne sont pas représentés. Dans le prototype, ils sont intégrés à l'interface des intentions et illustrés plus loin dans la figure 9.7 (méthodes `conflictP` et `weakP`).

Toutes les mises à jour, images ou opérations d'initialisation sont passées par l'application au gestionnaire de cohérence sous la forme d'un bloc de données opaque (classe `MBuff`).

Tous les appels ascendants de **Core**, autres que les prédicats du contrat de cohérence, sont construits autour d'un système de notification d'événements. L'application souscrit à des

```
class StateTransfertRequest {
public :
    virtual MBuff* invoke () = 0;
};
/*
 * 'Event' are the basic event type passed to EventHandlers. They may
 * be extended to encapsulate context dependent of the current use of
 * the Notifier mechanism.
 */
class Event {};
extern const Event voidEvent; // ONLY this very instance is considered void.

class EventHandler {
public:
    virtual void invoke(Event&) = 0;
};
```

```
/*
 * We need this to cope safely with various form of C++ inheritance
 * (multiple, virtual). This is *essential* to allow pointer
 * arithmetic often required in that case. This arithmetic is done
 * transparently by the compiler along the chain of explicit then
 * implicit casts from the raw buffer.
 */
class Coherence;

class ActivityFactory {
 public:
    virtual Activity*    create (IntentionPtr, Coherence* = NULL) = 0;
    virtual ActivityCMI *unmarshal (MBuff&) = 0;
};
```

FIG. 9.4: Interfaces des composants applicatifs.

catégories d'événements en fournissant une instance de la classe `EventHandler` ; le gestionnaire de cohérence invoque la méthode `invoke()` de cette instance lorsqu'un événement de ce type doit être notifié à l'application. L'application souscrit aux événements qui l'intéressent. Si un événement doit être soumis à l'application et que celle-ci n'a pas souscrit à ce type d'événement, l'événement est éliminé.

9.1.3 Code applicatif

Les trois classes `StateTransfertRequest`, `EventHandler` et `ActivityFactory` sont des classes abstraites. Ce ne sont que des interfaces d'appels ascendants spécialisés par la fonction qui leur est attribuée par le gestionnaire de cohérence (cf. figure 9.4) .

Ni les intentions ni les activités ne sont représentées dans la figure 9.1 représentant l'architecture. Ces éléments sont transients, générés par la fonction de contrôle d'accès, en effet de bord des accès au réplicat. Nous terminons cette section en présentant quelques exemples de mise en œuvre de ces éléments importants de l'architecture.

9.1.3.1 Activités

La figure 9.5 montre les classes abstraites définissant les interfaces d'une activité. La classe `ActivityCMI` est la seule qui importe pour **Core**. Elle définit l'ensemble des appels utilisés par le gestionnaire de cohérence. La classe abstraite `ActivityBI` définit une interface de référence pour les appels descendants de l'application. Elle n'est pas requise par **Core**.

L'instanciation comme la réincarnation des activités nécessite un composant qui connaisse leur type exact (le plus dérivé) ; c'est le rôle des instances des classes implantant l'interface `ActivityFactory`.

Chaque instance d'une activité est encapsulée dans une continuation dès qu'elle est passée au gestionnaire de cohérence. Cette encapsulation permet à l'ordonnanceur de gérer de façon uniforme les activités, indépendamment de leur caractère local ou distant ; ceci simplifie considérablement le code de l'ordonnanceur. Le bon fonctionnement de cette gestion repose sur des mécanismes de délégation : la méthode `self()` permet de prendre en compte le cas des opérateurs binaires polymorphiques de l'activité.

La figure 9.6 montre comment sont implantées ces méthodes dans le cas des activités synchrones fournies par le prototype.

```
class ActivityCMI {
public:
    virtual ~ActivityCMI(){}

      // Scheduling interface
    virtual void       run       ()          = 0;
    virtual void       resume    ()          = 0;

      /*
       * CM calls the sequence 'suspend' before 'waitingOn' to cope
       * with race condition than may occur between 'waiting' and
       * 'resume'. Every call to suspend is guaranteed to be in a
       * critical section.  Only 'waitingOn' is assumed to be
       * blocking.
       */
    virtual void       suspend   ()          = 0;
    virtual void       waitingOn (ActivityCMI&) = 0;

      // Finalisation interface
    virtual void       _end      ()          = 0;
    virtual void       _abort    ()          = 0;

      // Predicate interface
    virtual bool       nestedP    (const ActivityCMI&) const = 0;
    virtual bool       abortedP() const             = 0;
    virtual bool       committedP () const          = 0;

      // Needed for delegation
    virtual const ActivityCMI& self() const = 0;
    virtual const Intention&   getIntention() const = 0;

    virtual MBuff&     marshal    (MBuff&)         const = 0;
    virtual ostream&   print      (ostream&)       const = 0;

    virtual bool     operator == (const ActivityCMI &_act) const;
    virtual bool     operator <  (const ActivityCMI&) const;
};
```

```
class ActivityBI {
public:
    virtual ~ActivityBI(){}

    virtual void       begin     ()          = 0;
    virtual void       end       ()          = 0;
    virtual void       abort     ()          = 0;
    virtual void       update    (MBuff&)    = 0;
};

class  Activity : public ActivityBI, public ActivityCMI {};
class  NestedActivity :  public Activity {
public:
    virtual Activity*  createNested()  = 0;
};

/*
 * We need this to cope safely with various form of C++ inheritance
 * (multiple, virtual). This is *essential* to allow pointer
 * arithmetic often required in that case. This arithmetic is done
 * transparently by the compiler along the chain of explicit then
 * implicit casts from the raw buffer.
 */
class Coherence;
class ActivityFactory {
 public:
    virtual Activity*   create (IntentionPtr, Coherence* = NULL) = 0;
    virtual ActivityCMI *unmarshal (MBuff&) = 0;
};
```

FIG. 9.5: Interface d'une activité.

```
inline
void SynchronousActivity::begin ()
{ consistencyManager->beginActivity(*this); }

inline
void SynchronousActivity::end ()
{ consistencyManager->endActivity(*this); }

inline
void SynchronousActivity::abort ()
{ consistencyManager->abortActivity(*this); }

inline
void SynchronousActivity::update (MBuff &mb)
{ consistencyManager->update(mb, *this); }
```

```
inline
void SynchronousActivity::_end ()
{ committed = true; }

inline
void SynchronousActivity::_abort ()
{ aborted = true; }

inline
void SynchronousActivity::run ()
{}

inline
void SynchronousActivity::resume ()
{ suspended = false; }

inline
void SynchronousActivity::suspend ()
{ suspended = true; }

inline
void SynchronousActivity::waitingOn(ActivityCMI&)
{ suspended.wait(false); }
```

FIG. 9.6: Interface de l'activité synchrone fournie dans le prototype. La variable `suspended` est un objet encapsulant un mutex et une variable de condition. La clause `suspended.wait(false);` suspend la tâche exécutant l'appel.

```
typedef unsigned int Dimension;

const Dimension DOMAIN = 0x1;
const Dimension TYPE   = 0x2;
const Dimension OWNER  = 0x4;

#define ISDOMAIN(x) (x & DOMAIN)
#define ISTYPE(x)   (x & TYPE)
#define ISOWNER(x)  (x & OWNER)

class Intention {
public:

    Dimension activeDimension; // bitmap of active dimension

    Intention (Dimension = (DOMAIN|TYPE|OWNER));
    virtual ~Intention();

    virtual bool    initP     () const = 0;
    virtual bool    weakP     () const = 0;
    virtual bool    mutableP  () const = 0;
    virtual bool    conflictP (const Intention&) const = 0;

    virtual MBuff&  marshal       (MBuff&) const = 0;
    virtual ostream& print        (ostream&) const = 0;

    virtual bool operator == (const Intention&) const = 0;

    static bool filter ( bool domainp,
                         bool typep,
                         bool ownerp,
                         Dimension criterion);
};
```

```
class IntentionAlgebra : public Intention {
public:
    TypePtr   accessType;
    DomainPtr accessDomain;
    OwnerPtr  accessOwner;

    IntentionAlgebra ();
    IntentionAlgebra (Dimension, TypePtr, DomainPtr, OwnerPtr);
    virtual ~IntentionAlgebra();

    virtual bool    initP     () const;
    virtual bool    weakP     () const;
    virtual bool    mutableP  () const;
    virtual bool    conflictP (const Intention&) const;

    virtual MBuff&  marshal       (MBuff&) const;
    virtual ostream& print        (ostream&) const;

    virtual bool operator == (const Intention&) const;

    static caddr_t unmarshal (MBuff&);
};
bool IntentionAlgebra::weakP () const
{ return accessType->weakP(); }

bool IntentionAlgebra::conflictP (const Intention &i) const
{
    return filter( accessDomain->intersectP(*i.accessDomain),
                   accessType->conflictP(*i.accessType),
                   !( accessOwner->inP(*i.accessOwner) ||
                      i.accessOwner->inP(*accessOwner) ),
                   (activeDimension & i.activeDimension) );
}
```

FIG. 9.7: La classe `Intention` définit l'interface des intentions. La classe `IntentionAlgebra` représente explicitement sous forme de composants séparés, chacune des trois dimensions d'un accès.

9.1.3.2 Intentions et prédicats

La figure 9.7 montre la classe de base des intentions. Seules les méthodes `weakP()` et `conflictP()` sont utilisées par le gestionnaire de cohérence. La classe `IntentionAlgebra` représente explicitement les dimensions présentées dans la section 8.2.2 ; `TypePtr`, `DomainPtr` et `OwnerPtr` sont des pointeurs sur des entités représentant, pour chaque dimension, les parties auxquelles accède l'intention. Les pointeurs intègrent un comptage de références qui permet le partage des différentes dimensions par plusieurs intentions. Le prototype met à disposition du programmeur plusieurs variantes de chacune des dimensions : des domaines (`Domain`) scalaires ou des domaines arborescents (cf. classe `TreeDomain` dans la figure 9.8), par exemple dans la dimension « domaine ».

La méthode `conflictP()` implante le prédicat $\mathcal{P}_{l_1}$ et la méthode `weakP` implante le prédicat $\mathcal{P}_{l_2}$. La méthode `conflictP()` implantée suffit dans tout les cas où chaque dimension a la propriété d'un algèbre d'ensemble. De façon informelle, il faut que le gestionnaire de cohérence puisse paver l'ensemble de chaque dimension.

Limitations La méthode `weakP()` est unaire. Cela impose que $\mathcal{P}_{l_2}$ ne soit défini que sur la base de l'opération en question, sans pouvoir tenir compte de l'histoire en cours comme exécutée ; dans l'exemple de la figure 9.7, `weakP()` ne prend en compte que le type de l'opération.

La méthode `conflictP` ne se compare qu'avec une autre intention. Ceci ne permet pas au prédicat $\mathcal{P}_{l_1}$ de prendre en considération des dépendances mettant en jeu plus de deux opérations de l'histoire locale. De plus, dans la mise en œuvre actuelle, les seules opérations

```
template <int VectorSize>
class TreeDomain : public Domain {
    int  getColored() const;
public:
    vector<int> domain;

    TreeDomain();
    TreeDomain(int *init);
    TreeDomain(char* initText);

    virtual bool emptyP      () const;
    virtual bool unityP      () const;
    virtual bool inP         (const SetAlgebra&) const;
    virtual bool intersectP  (const SetAlgebra&) const;
    virtual bool operator <  (const TreeDomain&) const;
    virtual bool operator == (const SetAlgebra&) const;

    virtual ostream& print    (ostream&) const;
    virtual MBuff&  marshal  (MBuff &mb) const;
    static caddr_t  unmarshal (MBuff &mb);
};
```

```
class ScalarDomain : public Domain {
public:
    static const int EMPTYSET = -1;
    static const int UNITYSET = -2;
    int domain;

    ScalarDomain(int _domain) : domain(_domain) {}

    virtual bool emptyP      () const;
    virtual bool unityP      () const;
    virtual bool inP         (const SetAlgebra&) const;
    virtual bool intersectP  (const SetAlgebra&) const;
    virtual bool operator == (const SetAlgebra&) const;

    virtual ostream& print    (ostream&) const;
    virtual MBuff&  marshal  (MBuff &mb) const;
    static caddr_t  unmarshal (MBuff &mb);
};
```

FIG. 9.8: `TreeDomain` représente un domaine arborescent. Cette classe est utilisée par le prototype de l'éditeur pour représenter la partie utilisée d'un document LaTeX répliqué. `ScalarDomain` représente l'index de l'élément accédé d'un tableau répliqué.

---

testées sont celles faisant l'objet d'activités *en cours* d'exécution ; l'histoire passée n'est pas utilisée.

9.2 Liaison et initialisation d'un réplicat

La liaison est l'opération qui associe un réplicat à un gestionnaire de cohérence. La liaison n'est considérée comme terminée, qu'une fois le réplicat nouveau venu initialisé.

Dans l'architecture **Core**, la liaison est explicite (méthode `init`). Le paramètre essentiel de cette méthode est le nom du groupe identifiant le gestionnaire de cohérence et donc l'information partagée à joindre. Deux autres paramètres essentiels pour l'établissement de la liaison sont le passage des références des trois composants gérant les différents appels ascendants du gestionnaire de cohérence, i.e. les instances dérivant des classes `EventHandler`, `StateTransfertRequest` et `ActivityFactory`.

L'initialisation est modélisée comme une opération ; comme telle, une activité et une intention lui sont associées. Quatre points la distinguent cependant des autres opérations :

- Le prédicat `initP()` de son intention est satisfait.
- Son invocation, par le processus joignant le groupe, se fait par une méthode spécifique, `Coherence::init()`.
- Le transfert d'état qui est associé à l'exécution distante de l'opération n'est pas nécessairement effectué sur tous les réplicats.
- Sa terminaison indique au gestionnaire de cohérence qu'il peut accepter les invocations locales du processus et transmettre les opérations distantes au processus (les appels ascendants sont rendus passants).

Le protocole d'initialisation est intégré au contrat de cohérence : le prédicat de conflit (`Intention::conflictP`) et le contrat implanté par le gestionnaire de cohérence détermine le protocole d'initialisation.

Le chapitre 11 fournit des exemples de code de liaison minimal dans le cadre d'un entier répliqué.

9.3 Résumé

Nous avons présenté, dans ce chapitre, l'architecture **Core** proprement dite. Le gestionnaire de cohérence prenant en charge toute la distribution, seule l'interface avec le contrôle d'accès de l'applicatif fait partie de la spécification :

- Les instances de la classe `StateTransfertRequest` sont chargés de fournir les éléments (états, opérations) nécessaires à l'initialisation d'un nouveau venu.
- Les instances de la classe `EventHandler` sont chargées de répercuter les opérations distantes ordonnées par le gestionnaire et reçues par le mandataire local sur le réplicat local.
- Les instances de la classe `ActivityFactory` sont chargés de déballer les activités reçues par le mandataire local du gestionnaire de cohérence.
- Les instances de la classe `Intention` sont chargés d'implanter les prédicats $\mathcal{P}_{l_1}$ et $\mathcal{P}_{l_2}$ du critère **Core**.
- Les instances de la classe `Activity` sont chargés d'implanter les changements d'état du flot d'exécution applicatif, correspondant aux décisions prises par le gestionnaire de cohérence.

Le reste des éléments ne sont que des éléments internes tel que réalisés dans le prototype et présentés pour concrétiser la présentation de l'architecture.

Troisième partie

Éléments de validation

Cette dernière partie est consacrée à la validation de nos propositions. Elle est constituée des trois chapitres qui suivent.

Core se compose de trois volets : un modèle d'exécution, des principes architecturaux et un prototype.

En ce qui concerne la validation du modèle d'exécution de **Core**, nous espérons que son utilisation tout au long de cette thèse est convaincante : ce modèle est, à la fois, assez puissant pour exprimer tous les critères de cohérence évoqués dans ce document et suffisamment sélectif pour ne représenter que des propriétés d'ordre.

Quant à la validation des principes architecturaux, elle pose le problème commun à toutes les approches architecturales, il faut identifier les métriques pertinentes.

L'appréciation du bénéfice obtenu par la réutilisation est difficile, car aucune métrique simple ne permet de la représenter. Sur ce point, nous nous en remettons à l'appréciation du lecteur, en appuyant simplement sur le fait que l'ensemble du code réparti et l'essentiel des problèmes de concurrence sont confinés dans le composant de gestion de cohérence.

L'appréciation de l'adaptabilité de la gestion repose sur le prototype. Nous avons construit notre validation en deux phases :

- D'abord une preuve de faisabilité. La factorisation a été montrée au chapitre précédent, au travers de la description des éléments pertinents du prototype. La possibilité d'utiliser ce prototype, de façon conforme au principe de **Core**, fait l'objet des deux chapitres qui suivent.
- Ensuite une « preuve d'adéquation ». Nous montrons que l'approche de **Core** n'induit pas un *surcoût* rédhibitoire. Ce faisant, nous évaluons les surcoûts de la factorisation au travers des performances de la plate-forme prototype de **Core**. Cette étude fait l'objet du dernier chapitre de cette thèse.

Chapitre 10

Construction d'objets répliqués

Dans ce chapitre, nous montrons comment l'architecture **Core** a été utilisée pour implanter des objets répliqués et comment, dans ce contexte, cette architecture se compare avec les principes d'implantation ouverte (« Open Implementation ») développés par KICZALES dans [Kic96].

10.1 Une architecture pour les objets répliqués

Les applications que nous avons développées sur le prototype de **Core** sont basées sur des objets répliqués[1]. Construire des objets répliqués demande, en sus des services précédemment décrits, la réalisation du réplicat et la capture des accès à l'objet. Les accès sur le réplicat doivent être interceptés afin de pouvoir être contrôlés et diffusés par le gestionnaire de cohérence. Pour le programmeur d'application, l'idéal est de pouvoir utiliser comme réplicat un composant standard du marché et d'assurer de façon transparente la capture et le contrôle des accès sur celui-ci. Le programmeur interagit ainsi avec l'objet répliqué comme s'il s'agissait d'un objet local (cf. figure 10.1). L'intrusion de la distribution, de la réplication et de la concurrence dans le code applicatif est réduite.

La figure 10.2 montre quelques composants proposés au programmeur d'application par **Core**, pour développer des objets répliqués. La classe `AccessObject` définit une interface minimale pour interagir avec un gestionnaire de cohérence, i.e. l'interface de liaison, et l'interface pour les appels ascendants. La figure 10.3 montre un cas d'utilisation simple mais montrant toutes les étapes de l'utilisation d'un objet répliqué (entier).

10.2 Principes réflexifs de l'architecture

Plusieurs approches ont été adoptées pour construire des objets répartis pouvant être utilisés par le programmeur de façon similaire aux objets purement locaux. Les premières approches se sont basées sur des langages dédiés ; Orca [BK89], les langages utilisés au-dessus d'Argus [Lis88], d'Emerald [BHJ+87] ou de Clouds [DAM+91] en sont des exemples. Les approches architecturales sont apparues dès que les besoins d'adaptabilité dans la mise en œuvre de la répartition, de la concurrence et de la persistance sont devenus évidents. Les premières

[1] Dans le cas de l'éditeur, le réplicat est incarné dans autre contexte que celui du gestionnaire de cohérence.

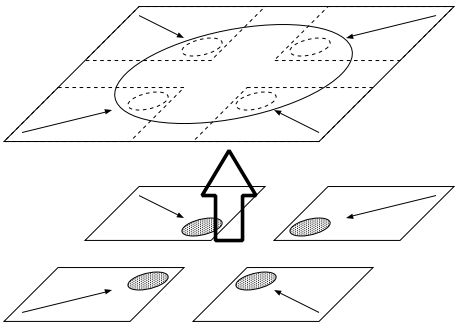


FIG. 10.1: Construction d'un objet répliqué à partir de composants standards. Le composant standard réalise le réplicat (en grisé). Le rôle de l'architecture est de définir les éléments qui transforment cet ensemble de réplicats en l'illusion d'un unique objet réparti : l'application peut alors invoquer cet objet (répliqué) comme un objet local.

---

approches tels Avalon [DHW88] et Arjuna/Shadows [PSWL95, CPS93] se sont basées sur l'héritage pour associer ces trois grandes propriétés aux objets implantant l'information partagée. Dans le cadre majoritaire des langages typant statiquement les objets, l'héritage pose deux problèmes :

- L'héritage crée une relation de sous-typage[2]. L'objet répliqué ne présentent plus le même type que son réplicat ; en pratique, l'interaction locale du programmeur avec l'objet s'en trouve considérablement modifié.
- L'héritage est défini statiquement à la compilation. On ne peut donc utiliser l'héritage pour composer des propriétés à l'exécution. Tout au plus, comme dans Shadows, peut-on ou non les activer.

Même si ces problèmes peuvent être contournés dans les langages à typage non statique tel que Smalltalk ou CLOS, le problème de fond reste entier : composer des propriétés systèmes en fusionnant les types des composants systèmes avec les types applicatifs compromet la réutilisation, même interne à cette application, des classes de l'application ; le cas de l'héritage des contraintes de synchronisation («inheritance anomaly») en est un exemple [Frø92, MTY93]. L'adaptation à l'exécution de propriétés systèmes réclame en fait quelques considérations supplémentaires.

10.2.1 Introduction à l'implantation ouverte

Le principe «d'implantation ouverte» («Open Implementation» (OI)) est proposée par KICKZALES dans [Kic96]. Ce principe se base sur une analyse des problèmes architecturaux, que pose aux applications, l'adaptation, à l'exécution, des propriétés de leur mise en œuvre [KTW92, KL92]. Le principe «d'implantation ouverte» consiste à adjoindre à une application ou à ses composants, une interface appropriée, dénotée «méta-interface», permettant de manipuler les aspects de sa mise en œuvre impliqués dans des propriétés système, telles que par exemple, la persistance, le contrôle de concurrence, la répartition ou la représentation

[2]Dans le cas général ce n'est pas systématiquement le cas (héritage privé en C++, par exemple) ; ça le devient lorsqu'il s'agit de composer des propriétés : l'objet doit pouvoir être passé à des fonctions/méthodes n'agissant que sur les types associés à une propriété particulière (migrable, verrouillable, partageable, persistant, ...).

```
class AccessObject : public EventHandler, public StateTransfertRequest {
public:
    Coherence          *consistencyManager;

    AccessObject(Coherence *cm = NULL)
        : consistencyManager(cm)
        {}
    virtual ~AccessObject() {
        delete consistencyManager;
        consistencyManager = NULL;
    }

    virtual void bind (Coherence* cm) = 0;

     // upcalls.
    virtual void    invoke(Event&) = 0;//   updates
    virtual MBuff* invoke ()       = 0; // state transfert request
};
```

```
class ReplicatedInteger : public ReplicatedScalar<int> {
    IntentionPtr    readIntention;
    IntentionPtr    writeIntention;
    IntentionPtr    read_writeIntention;

public:
    ReplicatedInteger(int initValue = 0);

    virtual void bind(Coherence* cm){}
    virtual void bind(ActivityFactory*, Coherence* cm);

    int  get() {
        SCALARACCESS(readIntention,int);
        return *value;
    }

    void set(int _value) {
        SCALARACCESS(writeIntention,int);
        *value = _value;
    }

    void inc() {
        SCALARACCESS(read_writeIntention,int);
        (*value)++;
    }

    void dec() {
        SCALARACCESS(read_writeIntention,int);
        (*value)--;
    }

    operator int ()
        { return get(); }
};
```

```
/*
 * Template access object for scalar objects. Data shipping is implemented.
 */
template <class T>
class ReplicatedScalar : public AccessObject {
    ActivityFactory *factory;

public:
    friend class Access;

    class Access {
        Activity     *act;
        IntentionPtr intention;
        ReplicatedScalar *rs;
    public:
        Access(IntentionPtr _intention, ReplicatedScalar* _rs) {
            intention = _intention;
            rs = _rs;
            act = rs->factory->create(intention, rs->consistencyManager);
            act->begin();
        }

        ~Access() {
            if (intention->mutableP()) {
                MBuff mb;
                mb << *rs->value;
                act->update(mb);
            }
            act->end();
            delete act;
        }
    };

    T*    value;                // The replica itself

    ReplicatedScalar(const T& initValue = 0)
        : AccessObject(NULL), factory (NULL), value(new T(initValue))
        {}

    ~ReplicatedScalar() {
        consistencyManager->unsubscribe(*this, CM_UPDATE_EVENT);
        consistencyManager->unsubscribe(*this, CM_INIT_EVENT);
    }

    void bind (Coherence*) = 0;
    void bind (ActivityFactory *_factory, IntentionPtr initIntention, Coherence* cm) {
        factory = _factory;
        consistencyManager = cm;
        cm->subscribe(*this, CM_UPDATE_EVENT);
        cm->subscribe(*this, CM_INIT_EVENT);
        Activity *act = factory->create(initIntention, consistencyManager);
        consistencyManager->init(*act, factory);
        delete act;
    }

      // Upcalls.
    virtual void    invoke(Event &_ev) {    // Updates
        MBuffEvent &mev = (MBuffEvent&) _ev;
        if ( (mev.kind() == CM_UPDATE_EVENT) ||
             (mev.kind() == CM_INIT_EVENT) ) {
            *mev.message >> *value;
        } else
            assert (0);
    }

    virtual MBuff* invoke () { // state transfert request
        MBuff *mb = new MBuff;
        *mb << *value;
        return mb;
    }
};

#define SCALARACCESS(intention, type) \
  ReplicatedScalar<type>::Access aReplicatedScalarAccess(intention, this)
```

FIG. 10.2: La classe abstraite `AccessObject` défini les services minimaux devant être assurés par le contrôle d'accès. La classe `ReplicatedScalar` implante le code typique nécessaire pour un scalaire; bien que le type des intentions ne soit pas spécifié, le « data shipping » impose en pratique une sémantique de concurrence de type lecture/écriture. La classe `ReplicatedInteger` montre son utilisation.

```
/*
 * Test a simple replicated integer.
 */
#include <stdlib.h>
#include <fstream.h>
#include "initcore.h"
#include "linearisable.h"
#include "linear2.h"
#include "debug.h"
#include "rinteger.h"
#include "synchronous.h"

ostream& operator << (ostream &out, const ReplicatedInteger &ri)
{ out << *ri.value; }

OFname ofName = "ReplicatedInteger";
ReplicatedInteger          *theReplicatedInteger;
Coherence     *cm;
SynchronousActivityFactory *sacFactory;
```

```
void main (int argc, char **argv)
{
    // Trace::currentDebugLevel = TR_CM|TR_UP_DOWN_CALL;
    if (argc > 1)
        coreLog = new ofstream(argv[1]);

    char * consistency = getenv("CONSISTENCY");

    assert( (consistency != NULL) &&
            ( (strcmp(consistency, "LinearisableConsistency") == 0) ||
              (strcmp(consistency, "Linear") == 0)) );

    initCore();
    theReplicatedInteger = new ReplicatedInteger(10000);

    if (strcmp(consistency, "LinearisableConsistency") ==0)
        cm = new LinearisableConsistency (ofName, theReplicatedInteger, NULL);
    else
        cm = new Linear2 (ofName, theReplicatedInteger, NULL);

    sacFactory = new SynchronousActivityFactory(cm->getId());
    theReplicatedInteger->bind(sacFactory, cm);

    while (theReplicatedInteger->get() > 0) {
        cerr << "(* " << *theReplicatedInteger << " *)" << endl;
        theReplicatedInteger->dec();
    }
    cerr << "That,s all folks\n";
    delete theReplicatedInteger;
    delete coreLog;
}
```

FIG. 10.3: Exemple complet d'une application très simple : l'application décrémente un entier jusqu'à zéro. L'exemple montrent le choix du gestionnaire de cohérence, la liaison et la boucle de décrémentation dans laquelle l'objet est invoqué de façon similaire à toute invocation sur un objet local. La plate-forme peut journaliser les actions et intègre des outils pouvant tester quelques propriétés d'ordre, locales ou réparties, sur une collection de journaux locaux.

---

en mémoire des données. Dans un cadre orienté-objet, ces aspects de la mise en œuvre de l'application (classes, méthodes, invocation, ...) sont représentés par des « méta-objets » ; la « méta-interface » de l'application, constituée de l'ensemble des interfaces des « méta-objets », définit comment ces méta-objets coopèrent et prend le nom de protocole à méta-objets (« MetaObject Protocol », ou MOP). Le lecteur est invité à consulter [KdRB91] et l'excellente introduction [KP] sous forme de tutoriel de KICZALES et PAEPCKE sur ce sujet.

Les principes d'implantation ouverte sont, en fait, une application du traditionnel « separation of concerns » [Aks96] aux relations application/système d'exploitation ; une « implantation ouverte » définit deux niveaux conceptuels :

- Un niveau de référence (« functional level »), celui du problème applicatif et de sa mise en œuvre (choix des structures de données, algorithmique, ...).
- un « méta-niveau » (« behavior level ») dans lequel sont décrites et définies les propriétés de l'implantation elle-même : concurrence, persistance, répartition et les techniques associées (mise en cache, journalisation, ...).

Dans une « implantation ouverte », les différents niveaux collaborent à la réalisation d'un code permettant l'exécution de l'application. Cette collaboration entre niveaux, i.e. cette faculté qu'ont les composants d'un méta-niveau d'interagir avec les composants du niveau de référence repose sur deux facultés d'un code applicatif : la possibilité d'introspecter (« introspection ») et de modifier (« intercession ») son propre état à l'exécution. Un système est dit réflexif s'il possède ces deux facultés [MF93].

La mise en œuvre de ces deux facultés repose sur une technique de base, la réification [GWB91, Weg92]. La réification consiste à incarner dans un objet, la propriété implicite de l'application que l'on désire manipuler : l'invocation d'une méthode ou bien l'accès à la repré-

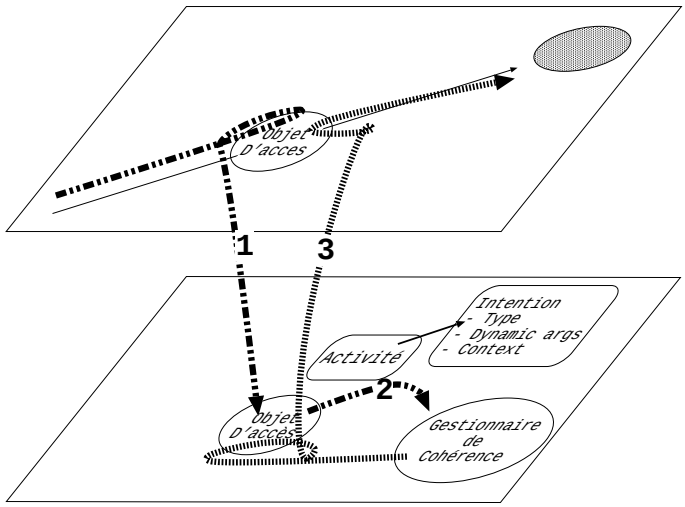


FIG. 10.4: Tous les méta-objets sont des éléments de **Core**. Les flèches représentent le va et vient entre les deux niveaux dans le cas d'une invocation (1) et dans le cas de la réception d'une opération distante (3). La flèche (1) représente la capture de l'invocation. L'activité réifie le contexte d'exécution de l'invocation et l'intention les aspects sémantiques (type et arguments) de cette même invocation. Ces deux méta-objets sont passés au gestionnaire de cohérence en début d'activité (2).

sentation d'un objet, par exemple.

10.2.2 Core et les principes d'implantation ouverte

Core fournit les composants qui permet à une application d'être structurée selon les principes d'implantation ouverte : au niveau de référence, le programme utilise un objet partagé. La concurrence, la répartition et la réplication sont assurés par les objets du méta-niveau : intention, activité et gestionnaire de cohérence. L'application adapte les éléments de la gestion de cohérence en choisissant son gestionnaire de cohérence et les classes auxiliaires, activité et intention, qui réifient les éléments de l'exécution manipulés par ce gestionnaire. L'interface des intentions, activités et gestionnaires de cohérences et leur coopération (l'architecture **Core**) constitue le protocole à métaobjet (MOP) qui assure la gestion de cohérence d'une application : ce MOP prend en charge tous les aspects de l'exécution qui permettent à l'information d'être partagée, répartie et répliquée.

Le prototype de **Core** est réalisé dans un langage, C++, qui n'offre aucun support pour la réflexivité, à tout le moins, aucun support équivalent à ce que propose des langages comme Smalltalk [GR83], CLOS [GWB91] ou OpenC++ [Chi95, Chi96]. Pour permettre aux éléments du méta-niveau d'exercer leur fonction, **Core** se base sur la délégation [Wol92]. Les objets d'accès sont des souches interposées devant les réplicats ; ces souches délèguent la réalisation locale de la méthode à ce réplicat, réifie l'invocation (activité et intention) et collaborent avec le gestionnaire de cohérence, contribuant ainsi au respect du MOP **Core**. La figure 10.4 montre comment ce MOP s'insère dans une architecture à implantation ouverte.

L'instanciation de l'objet d'accès et du gestionnaire de cohérence fait partie du protocole de liaison et n'est pas spécifiée par **Core**. Le protocole de liaison flexible des objets fragmentés peut parfaitement convenir [MGlNS94, Sha94] ; ceux-ci peuvent même être implantés

sous forme de MOP. Dans le prototype de **Core**, la liaison est effectuée explicitement par l'application (cf. figure 10.3).

10.2.3 GARF : un représentant de l'état de l'art

De nombreux travaux ont utilisé les techniques d'implantation ouverte, la plupart en construisant des protocoles à métaobjet : PCLOS [Pae90] implante un MOP pour la persistance en CLOS, CodA [McA95] propose un MOP prenant en compte, entre autres, certains aspects de la répartition et STROUD et WU ont implanté un MOP [SW95] en OpenC++ rendant les objets atomiques ; Muse [YTM91] et son successeur Apertos [Yok92] sont des systèmes d'exploitation complets structurés selon ces principes. Nous nous concentrons sur GARF, un travail ayant comme point commun avec **Core** de gérer la réplication d'objet partagés.

GARF est une architecture à méta-niveau implantée en Smalltalk visant à simplifier le développement d'applications tolérantes aux fautes. La tolérance aux fautes est assurée, en répliquant les objets accédés par l'application, sur un nombre configurable de serveur. Dans [GGM95], GARBINATO, GUERRAOUI et MAZOUNI présentent son utilisation dans le cadre de la réplication active [PBS+88, Lit91]. GARF se base sur Isis [Bir85] pour tout ce qui concernent les aspects « systèmes » de la tolérance aux fautes, en particulier, l'implantation du modèle de synchronie virtuelle. Le modèle de réplication proposé par GARF est le modèle traditionnel utilisé dans le cadre de la tolérance aux fautes : l'objet répliqué est représenté par un groupe de serveur, chacun gérant un réplicat ; les clients invoquent l'objet en diffusant les requêtes à l'ensemble du groupe. Chaque objet répliqué est représenté par un groupe Isis.

GARF implante la tolérance aux fautes de façon transparente : le code client comme l'objet réalisant le service sont des composants standards. GARF rajoute des souches (encapsulateur) au niveau des points d'invocations comme autour des réplicats ; ce sont ces souches, `ActiveReplica`, dans l'exemple décrit, qui implante la réplication active et gère l'interaction avec Isis. GARF capture les invocations sur l'objet, grâce à des modifications apportées à l'environnement d'exécution Smalltalk ; une fois capturée par GARF, le traitement de l'invocation est délégué à l'encapsulateur local de l'objet dans le contexte de l'invocation. Les encapsulateurs `ActiveReplica` implantent la réplication active de la façon suivante : chaque invocation est transformée en un multi-RPC ; l'invocation est diffusée (*abcast()* d'Isis) aux encapsulateurs serveurs, qui invoquent leur réplicat et renvoient la réponse. L'encapsulateur initiateur de l'invocation collecte les réponses de tous les serveurs et retourne la première d'entre-elles au code client. Chaque multi-RPC est mis en œuvre par une instance de la classe `Abcast` qui gère l'interaction avec Isis, initie la diffusion, attend et collecte les réponses.

Dans le cadre des objets répliqués, **Core** possède de nombreux points communs avec GARF : son architecture à méta-niveaux, la préservation du réplicat des aspects liés à la concurrence et la réplication et l'adaptabilité de la mise en œuvre de la réplication. L'objet d'accès de **Core** joue le rôle de la classe `ObjectName` réalisant la souche implanté par GARF pour capturer les accès destinés à l'objet. La classe `ActiveReplica` pourrait être une instance d'un gestionnaire de cohérence ; tout comme les encapsulateurs de GARF, les mandataires des gestionnaires de **Core** forment un groupe.

Néanmoins, sur le plan architectural, **Core** diffère significativement de GARF : **Core** gère la cohérence et non uniquement la réplication ; le contrôle du flux d'exécution local `Activité` est flexible ; les activités peuvent s'intégrer simplement dans un contexte transactionnel, et la politique d'emballage est découplée de la machinerie qui gèrent la cohérence ; d'autres différences existent, comme la délégation des prédicats de conflits ou le choix du « core set »,

mais qui découlent de la nature différente des buts adressés par les deux travaux.

10.3 Résumé

Nous avons montré dans ce chapitre comment **Core** avait été utilisé pour construire des objets répliqués et comment, dans ce contexte, il respectait les principes réflexifs à la base des principes d'implantation ouverte (« Open Implementation »).

Nous avons brièvement montré en quoi **Core** constituait un protocole à méta-objet spécialisé dans la gestion de cohérence.

Nous avons finalement comparé l'approche architecturale de **Core** à celle de GARF, un représentant de l'état de l'art dans le cadre des objets répliqués.

Chapitre 11

Deux applications de Core

Dans ce chapitre, nous montrons l'utilisation du prototype de **Core** par deux applications réparties. La première application consiste en un simple compteur répliqué utilisé de façon concurrente. La seconde application est une extension d'Emacs qui assure la réplication d'un tampon (« buffer ») Emacs et permet son édition collaborative.

11.1 Environnement de développement

Une description succincte du prototype **Core** est fournie dans l'annexe B. Actuellement, la plate-forme propose trois gestionnaires de cohérence, un service de nommage, un service d'emballage/déballage et quelques classes d'objets d'accès utilisées par les applications prototypes, en particulier, celles utilisées pour établir les mesures de performance (scalaire répliqué et tableau de scalaires répliqué).

Les gestionnaires de cohérence réalisés implantent le même contrat, la cohérence hybride proposée par ATTIYA et FRIEDMAN dans [AF92] : les activités sont partitionnées selon deux groupes, les activités `strong` et les activités `weak`. Les activités `strong` sont ordonnées totalement entre elles ; les activités `weak` sont ordonnées de façon FIFO entre elles. Les activités initiées par un même processus sont ordonnées de façon FIFO.

Dans le modèle d'exécution de **Core**, un tel contrat se formalise de la façon suivante :

Définition 11.1 (Cohérence Hybride) *Soit un système S de N processus, E, l'ensemble des opérations exécutées sur ce système, $\mathcal{S}$, l'ensemble des opérations* `strong` *et $\mathcal{W}$, l'ensemble des opérations* `weak`*. Une exécution vérifie la cohérence hybride si :*

- $E = \mathcal{S} \oplus \mathcal{W}$ *($\mathcal{S}$ et $\mathcal{W}$ forment une partition de E).*
- *Toute coupe de l'exécution est FIFO cohérente (cf définition 6.13).*
- $\forall i, j \in [1 \cdots N], \prec_i/_{\mathcal{S}} = \prec_j/_{\mathcal{S}}$ *(ordre total sur les opérations* `strong`*).*

Les gestionnaires diffèrent par leur protocole d'initialisation, l'ordre des `update()`, l'ordre des `endActivity()` et certaines optimisations sur le traitement des activités `weak` : l'un des gestionnaires (classe `LinearisableConsistency`) diffuse systématiquement toutes les activités ; les deux autres ne diffusent les activités `weak` que si elles ont modifié la donnée partagée (invocation de la méthode `Coherence::update()`). Les activités `strong` sont systématiquement diffusées.

En deux mots, l'ordre total des opérations `strong` est assuré en diffusant le `beginActivity()` d'une activité `strong` à l'aide d'un *abcast()* et en attendant la réponse (point-à-point FIFO)

de chacun des membres ; l'activité n'est pas « activée » tant que les réponses (positives) ne sont pas parvenues. Le reste des opérations, `update()` et `endActivity()` sont diffusées de façon FIFO ou totalement ordonnées, selon les gestionnaires ; ceci change les propriétés liées au changement de vue, et donc l'initialisation des nouveaux arrivants.

11.2 Entier répliqué

Les figures 11.1 et 11.2 exposent la totalité du code à la charge du programmeur d'application qui désire utiliser un entier répliqué. Ce code est destiné à montrer la flexibilité permise par **Core**. Seul le respect des interfaces présentées au chapitre précédent est requis ; le reste, i.e. la majeure partie de ce qui est présenté est affaire de choix du programmeur de l'application. Le choix qui nous a guidé est celui de la réutilisation des classes fournies par **Core** et la démonstration de la flexibilité de la plate-forme.

Le code de cette version est notablement plus complexe que celui présenté au chapitre précédent (cf. figure 10.2) ; essentiellement, parce que le contrôle d'accès et l'emballage sont devenus spécifiques de l'interface de l'entier ; ces services ne peuvent donc plus être assurés par le support générique pour scalaire fourni par la classe `ReplicatedScalar`. Là encore, tout est affaire de choix du programmeur. La spécialisation du code d'accès et d'emballage est plus complexe mais permet de meilleure performance ; le programmeur peut sélectionner à l'exécution le choix du noyau d'optimisation (« core set ») présenté au chapitre 7 ; la politique d'emballage choisie (« function shipping ») permet son exploitation.

Ce code peut sembler complexe pour un simple entier. Il s'agit pour l'essentiel du code de liaison et du code d'emballage/déballage. Ce coût est quasiment indépendant de la complexité de l'information partagée, et correspond au ticket d'entrée à payer pour la répartition. En outre, le code lié à l'objet répliqué, comme le code du réplicat lui-même, est indépendant de l'application ; ce code est donc réutilisable et peut servir à construire une librairie extensible d'objets d'accès, pour les types d'information courant. Le coût de programmation de l'objet répliqué est donc amorti.

11.3 Édition collaborative

La figure 11.3 montre une application de **Core** dans le cadre de l'édition collaborative. L'application permet à plusieurs personnes d'éditer concurremment un document sous Emacs.

L'application est composée de trois parties, deux librairies Emacs-Lisp[1] et un programme C++ contenant **Core** :

- La première librairie parcourt le tampon Emacs contenant le document devant être édité et construit des données (liste d'association de « marker » Emacs) correspondant à sa structure logique. Actuellement, seuls des documents LaTeX sont syntaxiquement analysés et la structure construite représente la décomposition en chapitre, section, sous-section, ... du document LaTeX.
- La deuxième librairie contrôle les accès au document. Emacs propose de nombreux crochets (« hook ») qui permettent de capturer et de contrôler les déplacements du curseur et les modifications du tampon. Les interactions de l'utilisateur et les modifications que celui-ci fait sont capturées et soumises à **Core**. Les accès locaux et les verrouillages de

[1] Elles sont actuellement fusionnées dans un seul « package » Emacs-Lisp.

```
#define SCALARACCESS(intention,ac)\
ReplicatedInteger::Access aReplicatedScalarAccess(intention, this, ac)

class ReplicatedInteger : public AccessObject {
protected:
    friend class Access;

      // Intentions are instanciated just once by 'bind', then reused.
    IntentionPtr    getIntention;
    IntentionPtr    setIntention;
    IntentionPtr    incIntention;
    IntentionPtr    decIntention;

    ActivityFactory *factory;

public:
    int     value;              // The replica itself :-)

    typedef IntType::weak_pattern weak_pattern;
    typedef IntType::access_t     access_t;
    static const access_t GET  = IntType::GET;
    static const access_t SET  = IntType::SET;
    static const access_t INC  = IntType::INC;
    static const access_t DEC  = IntType::DEC;
    static const access_t INIT = IntType::INIT;

    class Access {
        Activity      *act;
        IntentionPtr intention;
        ReplicatedInteger *rs;
        access_t      access;
    public:
        Access(IntentionPtr, ReplicatedInteger*, access_t);
        ~Access();
    };

    ReplicatedInteger  (int initValue = 0);
    ~ReplicatedInteger ();

    virtual void bind (Coherence* cm){}
    virtual void bind (ActivityFactory*, Coherence*, weak_pattern);

      // Upcalls.
    virtual void   invoke (Event &_ev);
    virtual MBuff* invoke ();

      // Functional Interface
    int  get () {
        SCALARACCESS(getIntention, GET);
        return value;
    }

    void set (int _value) {
        SCALARACCESS(setIntention, SET);
        value = _value;
    }

    void inc () {
        SCALARACCESS(incIntention, INC);
        value++;
    }

    void dec () {
        SCALARACCESS(decIntention, DEC);
        value--;
    }

    operator int () { return get(); }
};
```

```
class IntType : public Type {
public:
    typedef unsigned int access_t;
    typedef unsigned int weak_pattern;
    static const access_t GET  = (1<<0);
    static const access_t SET  = (1<<1);
    static const access_t INC  = (1<<2);
    static const access_t DEC  = (1<<3);
    static const access_t INIT = (1<<4);

    access_t      accessType;
    weak_pattern wp;

    IntType(access_t _access, weak_pattern _wp = (GET))
        : accessType(_access), wp(_wp) {}

    virtual bool conflictP (const Type&) const;
    virtual bool weakP      () const;
    virtual bool initP      () const;

    virtual bool operator == (const Type&) const;
    virtual ostream& print (ostream&) const;
    virtual MBuff&   marshal (MBuff&) const;

    static  caddr_t  unmarshal (MBuff&);
};

inline bool IntType::conflictP (const Type &_set) const {
    IntType set = (IntType&) _set;
    access_t merge = (set.accessType | accessType);
    return ( (merge & (INIT | SET)) ||
             ((merge & (INC | DEC)) && (merge & ~(INC | DEC))) );
}

inline bool IntType::weakP () const
{ return (accessType & wp); }

ReplicatedInteger::Access::Access( IntentionPtr _intention,
                                   ReplicatedInteger* _rs,
                                   access_t ac) {
    intention = _intention;
    access = ac;
    rs = _rs;
    act = rs->factory->create(intention, rs->consistencyManager);
    act->begin();
}

ReplicatedInteger::Access::~Access () {
    if (access != GET) {
        MBuff mb;
        if (access == SET)
            mb << rs->value;
        mb << access;
        act->update(mb);
    }
    act->end();
    delete act;
}
```

FIG. 11.1: Entier répliqué implantant les principes de **Core** comme la notion de « core set » (cf. exemple 7.3). En pratique {inc, dec} et {get} sont les deux « core set » viables ; le client choisit, lors de la liaison (`bind()`), celui qu'il considère comme le plus adapté.

```
ReplicatedInteger::ReplicatedInteger(int initValue)
    : AccessObject(NULL), factory (NULL), value(initValue) {}

ReplicatedInteger::~ReplicatedInteger() {
    consistencyManager->unsubscribe(*this, CM_UPDATE_EVENT);
    consistencyManager->unsubscribe(*this, CM_INIT_EVENT);
}

void ReplicatedInteger::bind(ActivityFactory* _factory,
                             Coherence* cm,
                             weak_pattern wp) {
    OwnerPtr        anOwner;
    DomainPtr       domain;

    anOwner  = OwnerPtr(new ScalarOwner(cm->getId(),
                                        getBoarGenerator()->getUniqueId()));
    domain   = DomainPtr(new Domain);

    getIntention = IntentionPtr (new IntentionAlgebra(DOMAIN|TYPE|OWNER,
                                                      TypePtr(new IntType(GET,wp)),
                                                      domain,
                                                      anOwner));
    setIntention = IntentionPtr (new IntentionAlgebra(DOMAIN|TYPE|OWNER,
                                                      TypePtr(new IntType(SET,wp)),
                                                      domain,
                                                      anOwner));
    incIntention = IntentionPtr (new IntentionAlgebra(DOMAIN|TYPE|OWNER,
                                                      TypePtr(new IntType(INC,wp)),
                                                      domain,
                                                      anOwner));
    decIntention = IntentionPtr (new IntentionAlgebra(DOMAIN|TYPE|OWNER,
                                                      TypePtr(new IntType(DEC,wp)),
                                                      domain,
                                                      anOwner));

    IntentionPtr initIntention (new IntentionAlgebra(DOMAIN|TYPE|OWNER,
                                                     TypePtr(new IntType(INIT,wp)),
                                                     domain,
                                                     anOwner));

    getBoarRepository()->registerU(TID_INTTYPE, IntType::unmarshal);
    factory = _factory;
    consistencyManager = cm;
    cm->subscribe(*this, CM_UPDATE_EVENT);
    cm->subscribe(*this, CM_INIT_EVENT);
    Activity *act = factory->create(initIntention, consistencyManager);

      // The real Core bind.
    consistencyManager->init(*act, factory);
    delete act;
}

//  Updates (function shipping)
void   ReplicatedInteger::invoke(Event &_ev) {
    MBuffEvent &mev = (MBuffEvent&) _ev;

    if (mev.kind() == CM_INIT_EVENT)
        *mev.message >> value;
    else if (mev.kind() == CM_UPDATE_EVENT) {
        access_t      access;
        *mev.message >> (int&) access;
        switch (access) {
          case SET :
              *mev.message >> value;;
              break;
          case INC:
              value++;
              break;
          case DEC :
              value--;
              break;
          default :
              assert(0);
        }
    } else
        assert (0);
}

// state transfert request
MBuff* ReplicatedInteger::invoke () {
    MBuff *mb = new MBuff;
    *mb << value;
    return mb;
}
```

```
#include <stdlib.h>
#include <fstream.h>
#include "debug.h"
#include "initcore.h"
#include "rinteger.h"
#include "synchronous.h"
#include "linearisable.h"
#include "linear2.h"
#include "hybrid.h"

OFname ofName  = "ReplicatedInteger";
SynchronousActivityFactory *sacFactory;
ReplicatedInteger          *theReplicatedInteger;
Coherence                  *cm;

ostream& operator << (ostream &out, const ReplicatedInteger &ri)
{ out << ri.value; }

void main (int argc, char **argv) {
    bool passive  = (argc < 2);
    unsigned int nbInvocation = (passive ? 0 : atoi(argv[1]));

    if (getenv("CORELOG"))
        coreLog = new ofstream(getenv("CORELOG"));

    char *consistency = getenv("CONSISTENCY");
    char *coreset     = getenv("WEAKSET");

    assert( (consistency != NULL) &&
            ( (strcmp(consistency, "LinearisableConsistency") == 0) ||
              (strcmp(consistency, "Linear") == 0) ||
              (strcmp(consistency, "Hybrid") == 0) ));

    initCore();
    theReplicatedInteger = new ReplicatedInteger(nbInvocation);

    if (strcmp(consistency, "LinearisableConsistency") == 0)
        cm = new LinearisableConsistency (ofName, theReplicatedInteger);
    else if (strcmp(consistency, "Linear") == 0)
        cm = new Linear2 (ofName, theReplicatedInteger);
    else if (strcmp(consistency, "Hybrid") == 0)
        cm = new Hybrid (ofName, theReplicatedInteger);
    else
        assert (0);

    IntType::weak_pattern myCoreSet;
    if (coreset && (strcmp(coreset, "INCDEC") == 0))
        myCoreSet = ( ReplicatedInteger::INC | ReplicatedInteger::DEC );
    else
        myCoreSet = ( ReplicatedInteger::GET );

    sacFactory = new SynchronousActivityFactory(cm->getId());
    theReplicatedInteger->bind(sacFactory, cm, myCoreSet);

    if (!passive) {
        while (nbInvocation-- > 0) {
            if ((nbInvocation%100) == 0)
                cerr << "(* "
                     << theReplicatedInteger->value
                     << " *)" << endl;
            theReplicatedInteger->dec();
        }
    } else
        thread_suspend();

    cerr << "That,s all folks\n";

    delete theReplicatedInteger;
    delete coreLog;
}
```

FIG. 11.2: À gauche, le code de liaison et les appels ascendants d'un entier répliqué. À droite, le code de l'utilisateur de cet entier répliqué.

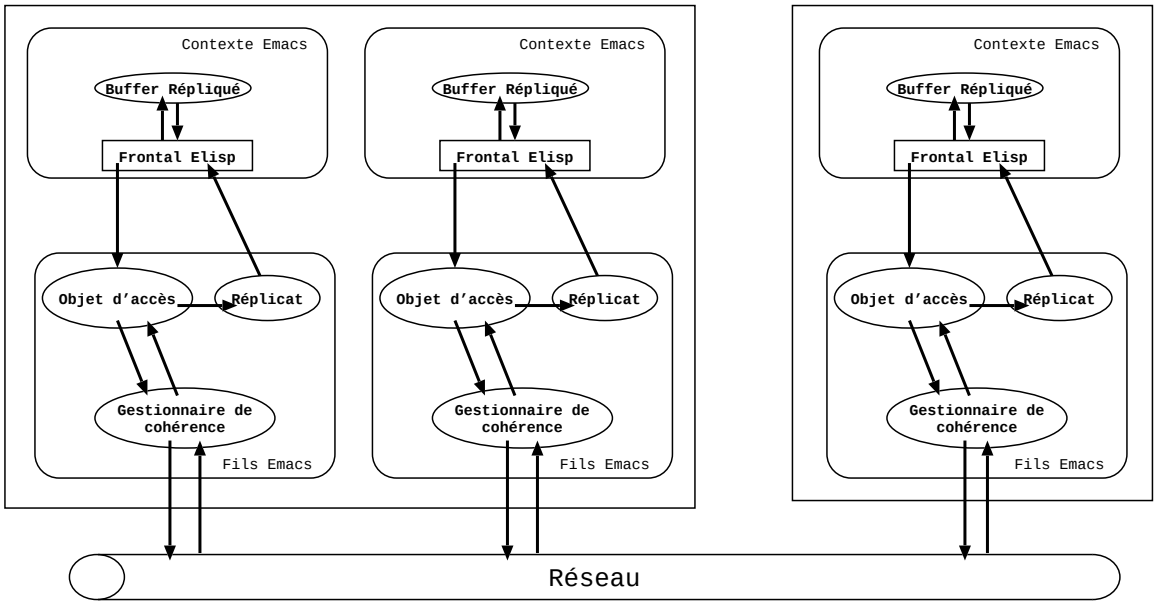


FIG. 11.3: Trois processus Emacs partagent un document. Chaque processus Emacs possède un réplicat du document (un « buffer »). **Core** n'est pas implanté dans le processus Emacs mais dans un processus fils. Emacs et le fils communiquent par des « pipes » Unix. Dans le processus fils, l'objet « réplicat » est une simple souche qui transmet à Emacs les invocations en provenance de l'objet d'accès. De la même façon, Emacs interagit avec le réplicat au travers de l'objet d'accès. Le contrôle de l'accès est en fait réparti entre Emacs et son processus fils.

sections du document sont transmis au programme C++ où ils sont représentés sous forme d'activités **Core**. Cette librairie reçoit de **Core** les modifications faites par les utilisateurs distants et les applique au tampon local.

- Le programme C++ contient le gestionnaire de cohérence, maintient la représentation des interactions effectuées par l'utilisateur dans Emacs et coopère, via des pipes Unix, avec la librairie implantant le contrôle d'accès dans Emacs.

En ce qui concerne l'interaction utilisateur, chaque agent lance son propre Emacs et charge dans un tampon Emacs le fichier associé au document. Ce fichier correspond à l'image persistante du document. Ce fichier est utilisé pour initialiser le premier membre du groupe. L'agent joint alors le groupe des membres en invoquant une commande Emacs-Lisp. À la fin de cette commande, le tampon Emacs contenant son fichier est initialisé de façon correspondante au contrat et l'agent perçoit les modifications que font les autres agents ; de façon symétrique, les autres membres perçoivent ses modifications.

Les règles qui régissent les perceptions des actions sont contrôlées par le contrat de cohérence, i.e. la cohérence hybride dans le prototype. L'agent peut verrouiller en lecture ou en écriture une partie de la hiérarchie du document. Toute modification du document construit implicitement une activité en écriture sur le paragraphe modifié.

Par rapport à l'entier répliqué, cette application montre d'autres propriétés de **Core** :

- Le découplage entre la représentation du réplicat et la gestion de cohérence : le gestionnaire de cohérence et le document Emacs sont dans deux contextes/processus distincts. Cette architecture a permis de ne pas toucher au noyau de l'éditeur d'Emacs et d'utiliser les mécanismes extensibles d'Emacs (environnement et bibliothèque Lisp) pour implanter la réplication du document et son édition collaborative.
- L'abstraction de l'intention dans deux autres axes :
  - L'agent initiateur de l'opération (`Owner`) : une tâche est créée pour chaque requête en provenance d'Emacs. Toutes ces tâches représentent le *même* agent, implanté par la classe `SiteOwner` fournie dans le prototype.
  - Le domaine de l'opération (`Type`) : le document partagé (un texte sous LaTeX) est structuré en arbre. La classe `TreeDomain` fournie dans le prototype permet une concurrence adaptée à la partie du document effectivement accédée (contrôle de concurrence à grain fin).

Le code ne montre pas une autre propriété attendue, l'abstraction du contexte d'exécution dans l'activité. Néanmoins, à l'usage, il apparaît que l'activité synchrone n'est pas adéquate : la demande d'un verrouillage sur une portion du document déjà verrouillée par un autre membre bloque l'interaction utilisateur dans Emacs ; ce comportement n'est pas supportable. La mise en place d'activité asynchrone notifiant Emacs et/ou l'utilisateur quand sa requête a abouti correspond mieux aux standards d'interaction dans une application interactive. Nous pensons que l'abstraction de l'activité permet une telle implantation, ainsi que d'autres stratégies, telles que la notification de la personne détenant le verrou (« call-back »), par exemple.

Enfin, les gestionnaires de cohérences utilisés par cette application, sont les *mêmes* (i.e. les instances des mêmes classes) que ceux utilisés pour l'entier répliqué.[2]

[2] La simulation de gestion de ressource basé sur un sémaphore répliqué n'utilise pas cet entier. Elle implante un nouveau type permettant une lecture/écriture « atomique ». Construire le sémaphore en utilisant l'entier répliqué nécessite le support des activités emboîtées (get/inc) qui ne sont plus disponibles dans cette version de la plate-forme.

11.4 Résumé

Nous avons montré dans ce chapitre la faisabilité des deux propriétés clés de **Core** :

- Chacune des deux applications présentées peut choisir, à l'exécution, l'un quelconque des gestionnaires de cohérence. Une application peut donc *adapter*, à l'exécution, sa gestion de cohérence.
- Chacun des gestionnaires peut être utilisé, *tel que*, dans l'une ou l'autre des applications. La gestion de cohérence est donc *réutilisable*.

De plus, l'essentiel des classes implantant certains traits d'une application, tels que la dimension d'un accès, le contrôle de l'accès, l'emballage des opérations, l'asynchronisme des opérations, ne sont pas spécifiques des applications présentées ; ces classes peuvent être placées dans une bibliothèque et réutilisées ultérieurement. Les classes utilisées dans toutes les applications développées au dessus de **Core** ont été construites de cette façon ; elles constituent une bibliothèque dans laquelle puise directement le programmeur d'application, comme le programmeur de l'objet répliqué.

Chapitre 12

Éléments de performance

Dans ce chapitre, nous évaluons les surcoûts liés à la factorisation de la gestion de cohérence. Par surcoûts nous entendons l'augmentation du temps d'exécution que peut subir une application utilisant **Core**, par rapport à une application utilisant la même gestion de cohérence, mais sans souci d'adaptabilité ni de réutilisation.

12.1 Conduite des mesures

Les mesures ont été effectuées sur sept stations DEC 3000/500 munies d'un processeur Alpha 21064 cadencé à 150 MHz[1] tournant sous OSF/1 3.0 et connectées par un réseau FDDI à 100 Mbits/secs.

Les mesures ont été faites sur des stations et un réseau local peu chargé. Les bancs de test sont suffisamment longs, plusieurs dizaines de secondes à plusieurs minutes, pour pondérer les coûts de chargement des programmes de test. De plus, les programmes ont été lancés plusieurs fois avant d'être mesurés, ceci afin de minimiser les défauts de pages initiaux[2].

À l'usage, les différentes commandes `time` d'OSF/1 *sous* évaluent de façon très importantes les temps CPU des programmes multi-tâches (seul le temps CPU de la tâche `main` semble être comptabilisé). La figure 12.1 montre le code utilisé pour la mesure des divers temps d'exécution.

Pour un même banc de test, les mesures ont montré une variance notable ; nous ne montrons pas des résultats moyens, mais les résultats des bancs de test les meilleurs.

Toute les applications utilisées pour les mesures sont disponibles avec la plate-forme prototype (cf. annexe B).

12.2 Évaluation du support de communication

Les implantations des gestionnaires de cohérence utilisent de façon intensive deux services fournis avec le prototype : la plate-forme de communication de groupe et le service d'appel à distance (RPC). Toutes les mises en œuvre de la cohérence hybride reposent, pour le

[1] L'une d'entre elle est cadencée à 160MHz et munie de 128 Mo de mémoire vive. Les mesures obtenues ne diffèrent pas sensiblement de celles obtenues sur les autres machines, dès lors que l'application utilise plusieurs réplicats.

[2] **Core** ne génère pas de librairies partagées ; les exécutables sont donc gros, de l'ordre de 10 Mo sur l'environnement mesuré.

```
#include <sys/types.h>
#include <unistd.h>
#include <iostream.h>
#include <sys/resource.h>

#if defined(FUNCTION_H)
//  'times' conflict with the STL library !
#warning  Do not forget to link with chrono.o !!
#define times hacked_times
#endif

#include <sys/times.h>

class Chronometre {
    static time_t tickToTime (time_t tick) {
        long long t = tick*1000; // sec -> msec
        return t/sysconf(_SC_CLK_TCK);
    }

      // (t2 -t1) in msec
    static long timevalDiff (const timeval &t1, const timeval &t2) {
        long long _t1 = t1.tv_sec*1000 + t1.tv_usec/1000;
        long long _t2 = t2.tv_sec*1000 + t2.tv_usec/1000;
        return _t2 - _t1;
    }

public:
    rusage start_rusage, stop_rusage;
    time_t start_time, stop_time;
    tms    start_tms, stop_tms;

    start() {
        start_time = times (&start_tms);
        getrusage(RUSAGE_SELF,&start_rusage);
    }

    stop() {
        stop_time = times (&stop_tms);
        getrusage(RUSAGE_SELF,&stop_rusage);
    }

    ostream& print (ostream &out) const {
        time_t real_t, user_t, system_t;
        long   user_delta, system_delta;

        real_t   = tickToTime(stop_time - start_time);
        user_t   = tickToTime(stop_tms.tms_utime - start_tms.tms_utime);
        system_t = tickToTime(stop_tms.tms_stime - start_tms.tms_stime);

        user_delta = timevalDiff(start_rusage.ru_utime, stop_rusage.ru_utime);

        system_delta = timevalDiff(start_rusage.ru_stime, stop_rusage.ru_stime);

        out << "real ("   << real_t   << ")\t"
            << "user ("   << user_t   << ")\t"
            << "system (" << system_t << ")\n";
        out << "User ("   << user_delta   << ")\t"
            << "System (" << system_delta << ")\n";
    }
};

ostream& operator << (ostream  &out, const Chronometre &chrono)
{ return chrono.print(out); }

#if defined(FUNCTION_H)
#undef times
#endif
```

FIG. 12.1: Code de l'instrument de mesure. Ce code est celui ayant servi à la relève de tous les temps fournis ci-après.

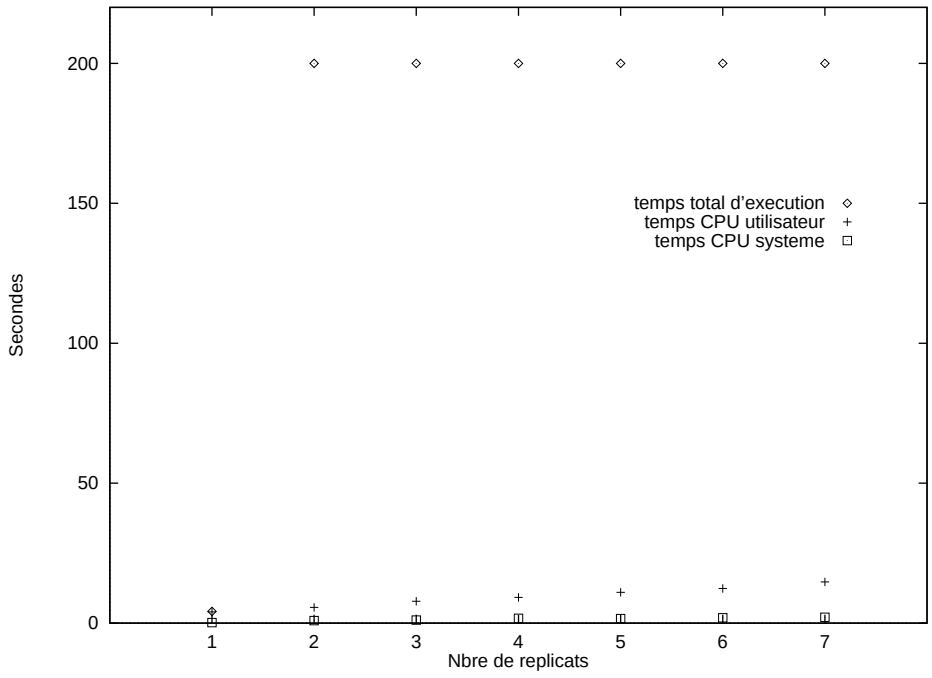


FIG. 12.2: Évolution du coût de multi-RPC totalement ordonné, en fonction du nombre de réplicats. Chaque point correspond à 1 000 multi-RPC nuls.

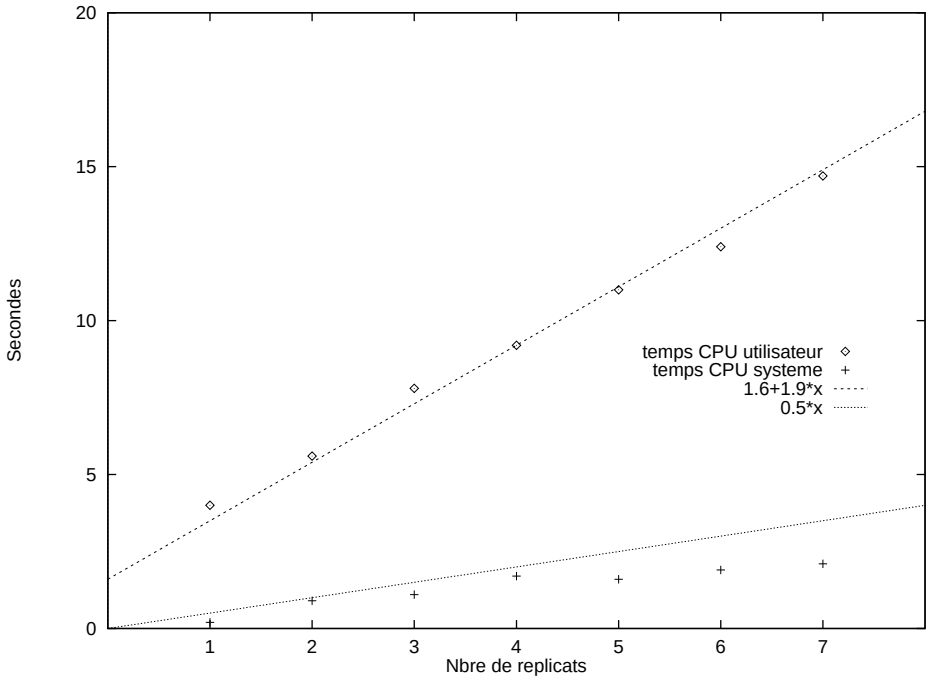


FIG. 12.3: Évolution du coût des multi-RPC totalement ordonnés en fonction du nombre de réplicats. Agrandissement de la figure 12.2 montrant les temps d'exécution utilisateur et système.

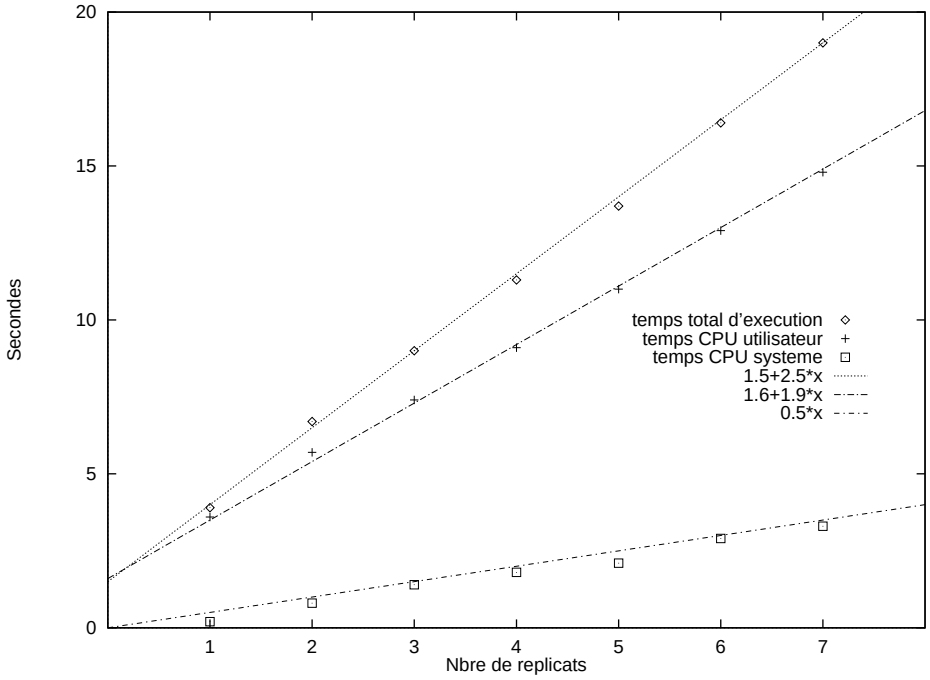


FIG. 12.4: Évolution du coût des multi-RPC FIFO en fonction du nombre de réplicats. Chaque point correspond à 1 000 multi-RPC nuls. Les droites permettent de visualiser l'évolution du coût en fonction du nombre de réplicats.

verrouillage des activités `strong`, sur l'utilisation de multi-RPC basés sur un *abcast()*. Les requêtes de début d'activité sont ordonnées totalement en les diffusant de façon atomique (*abcast()*) à l'ensemble du groupe. L'activité est ordonnée lorsque la réponse de chacun à été reçue. Un membre attend pour renvoyer une réponse que l'activité demandée ne soit en conflit avec aucune des activités en cours d'exécution dont il a connaissance. La connaissance du coût de ces mécanismes permet de faire la part entre le coût associé à la plate-forme de communication de groupe et le coût associé à la gestion de cohérence.

La diffusion atomique est implantée par un séquenceur : le site émetteur envoie le message au séquenceur qui le diffuse ensuite de façon FIFO (*fbcast()*) (N point à point), à l'ensemble du groupe. Le temps minimum correspond donc à 1 RTT ; celui-ci est inférieur à 1 msec sur notre environnement. Le séquenceur est un membre du groupe. Lorsque les mesures sont effectuées avec un seul réplicat, ce réplicat sert de séquenceur. Dans tous les autres cas, le réplicat abritant le séquenceur n'est pas celui sur lequel s'effectue la mesure.

Le coût d'un multi-RPC nul avec un seul réplicat est de 3.5 msec. Ce coût monte à 200 msec avec plusieurs réplicats, et reste indépendant du nombre de réplicats (cf. figure 12.2). La figure 12.4 montre ce coût lorsque les multi-RPC sont basés sur une diffusion FIFO (*fbcast()*) au lieu d'une diffusion atomique (*abcast()*). Les coûts du multi-RPC avec *abcast()* sont supérieurs d'un ordre de grandeur (200 msec / 14 msec) à ceux obtenus dans des conditions similaires en utilisant un multi-RPC avec *fbcast()*. Simultanément, le temps processeur utilisé reste faible (cf figure 12.3), du même ordre de grandeur que ceux obtenus avec la diffusion FIFO. Ce dernier point montre que le temps d'exécution d'un multi-RPC basé sur un *abcast()* est dominé par la latence. Cette latence est, par contre, bien supérieure à 3 demi RTT, valeur correspondant à la latence minimum permise par le protocole. Ce point, associé à la dissociation du temps CPU, signifie qu'il s'agit de latence locale. Celle-ci est vraisemblablement induite par la politique d'ordonnancement des tâches communicantes à l'intérieur de la plate-forme[3].

Le plateau, en lui même, n'est pas important par sa présence, une optimisation soigneuse de la plate-forme peut le supprimer. Ce plateau nous donne par contre, de façon opportune, un signal facilement lisible du surcoût des opérations qui nécessitent un agrément au dessus du réseaux : ces opérations utilisent la primitive responsable de ce plateau (multi-RPC avec *abcast()*) et exhiberont ce surcoût brutal.

La figure 12.4 apporte un autre élément important qui concerne l'évolution des coûts d'exécution. Ces coûts évoluent linéairement en fonction du nombre de réplicats ; d'autres mesures non présentées montrent que ce coût évolue aussi linéairement en fonction du nombre de multi-RPC. Le coût total d'exécution d'un multi-RPC nul évolue grossièrement selon la droite affine $1.6 + 1.9N$, N représentant le nombre de réplicats. Dans la plate-forme, les éléments qui dépendent du nombre de réplicats sont, la diffusion de la requête (N communications point à point FIFO), la délivrance des réponses et leur déballage. Ces opérations coûtent donc de l'ordre de 1.9 msec par réplicat. Le coût restant identifie l'emballage de la requête, sa soumission à la plate-forme de communication par le service de RPC et la latence de la réponse. Ce coût est de l'ordre de 1.6 msec par RPC.

[3]D'autres mesures, présentées en annexe, montrent que le débit des *abcast* dans nos essais est proche de celui des *fbcast*. Ceci tend à montrer que la latence n'est pas induite par le séquenceur mais par la réponse locale à un multi-RPC diffusé de façon atomique. Les bonnes performances du cas à un réplicat, non seulement ne sont pas contradictoires, mais constituent plutôt un argument : ce cas est traité comme un *fbcast* par la plate-forme de communication. Un autre point en faveur de cette interprétation est que ce phénomène plateau n'apparaît pas sur les mesures non publiées, effectuées sur une autre plate-forme (3 stations Solaris 2.5.1).

12.3 Coût de la gestion de cohérence

Les coûts d'exécution présentés correspondent, à une variante près (l'opération `get` à été supprimée), à l'exécution du compteur répliqué présenté au chapitre précédent (cf. figures 11.1 et 11.2, pages 123 et 124). Le code instrumenté est celui de la figure 12.5. Les temps d'exécution correspondent à l'exécution de 1 000 invocations.

Les seules invocations sur le compteur sont des opérations `dec`, donc commutatives. Les figures 12.6 et 12.7 montrent les temps d'exécution lorsque les opérations sont étiquetées `strong`, i.e. lorsque un agrément entre les membres du groupe est nécessaire pour ordonner une opération. Cet agrément utilise un multi-RPC basé sur *abcast()*. La figure 12.8 montre les temps d'exécution lorsque les opérations sont étiquetées `weak`. Les coûts et leur évolution diffèrent d'un ordre de grandeur entre ces deux bancs. Le banc de test où les opérations sont étiquetées `strong` exhibe l'aspect en plateau caractéristique du multi-RPC utilisé.

Dans les deux cas, le contrôle de concurrence tient compte de la commutativité des opérations `dec`. Seul diffère l'étiquetage `strong` ou `weak` des opérations, i.e. le fait que l'ordonnancement d'une activité `strong` fasse l'objet d'un agrément ou non. Le choix des opérations appartenant au noyau d'optimisation (« core set ») a un impact clair sur les performances. Bien que cet effet soit attendu, c'est, à notre connaissance, la première fois que ce concept est implanté et son effet mesuré. L'exemple montre comment **Core** permet à l'application de choisir ce noyau à l'exécution.

La figure 12.9 expose le coût associé à la gestion de cohérence dans le cas où les opérations `dec` sont étiquetées `weak`. Le traitement d'une telle opération consiste à créer une activité, à parcourir les queues locales pour s'assurer que cette activité n'est conflictuelle avec aucune des activités en cours, à emballer l'activité, à diffuser (*abcast()* dans le cas mesuré) son début, à diffuser les mises à jour associées puis à diffuser sa terminaison. Aucune optimisation, en particulier de type « piggy-backing » n'est effectuée sur le prototype. Le coût induit par le code du gestionnaire proprement dit consiste donc en le coût total d'exécution moins celui lié à la diffusion des messages, temps de traitement de la réception et de la délivrance des messages compris. Le temps de la diffusion d'un message ne représente qu'une borne minimum de ce temps mais est le seul que l'on puisse mesurer de façon simple et fiable sur notre plate-forme. La courbe `f3()` de la figure 12.9 représente donc une borne *maximum* du coût associé à la gestion de cohérence proprement dit.

Les mesures présentées jusqu'ici ne suffisent pas à déterminer le surcoût de la factorisation. Celui-ci est difficile à évaluer car la structure même du code est modifiée : la gestion des files d'attentes, la représentation des activités en cours d'exécution et les coûts associés à la manipulation de ces structures peuvent s'en trouver considérablement modifiés ; ceci alors même que ces coûts sont traditionnellement sur le chemin critique de la gestion de concurrence dans les moniteurs transactionnels. Malgré tout, ces mesures permettent de se faire une idée du coût maximal d'une telle gestion. Ce coût dans notre prototype est comparable à la valeur de la latence d'un réseau local. Ces coûts sont clairement optimisables ; il suffit de se référer à des supports de communication de groupe appartenant à l'état de l'art, comme Horus. Alors même que ces coûts sont facilement optimisables, les variations des temps d'exécutions obtenues selon la sélection du noyau d'optimisation montre que ces coûts sont déjà largement amortis dès lors que les gains obtenus concernent les latences appartenant aux environnements ciblés (plusieurs dizaines à plusieurs milliers de millisecondes).

```
/*
 * Test a concurrent replicated integer.
 */
#include <stdlib.h>
#include <fstream.h>
#include "initcore.h"
#include "com.h"
#include "debug.h"
#include "rinteger.h"
#include "synchronous.h"
#include "linearisable.h"
#include "linear2.h"
#include "hybrid.h"
#include "chrono.h"

ostream& operator << (ostream &out, const ReplicatedInteger &ri)
{ out << ri.value; }

class BenchBarrier : public EventHandler {
    Predicate ready;
    int nbMember;
    int triggerNb;

    BenchBarrier (const BenchBarrier&) : ready(false) {}

public :
    BenchBarrier (int _triggerNb)
        : nbMember(0), triggerNb(_triggerNb), ready(false) {}
    ~BenchBarrier () {}

    void invoke(Event &_ev) {
        MultiCast::ChangeEvent &ev = (MultiCast::ChangeEvent&) _ev;
        nbMember = ev.group.size();
        if (nbMember == triggerNb)
            ready = true;
    }

    void wait(int nb) {
        if (nb < triggerNb)
            ready.wait(true);
    }
};

OFname ofName  = "ReplicatedInteger";
SynchronousActivityFactory *sacFactory;
ReplicatedInteger          *theReplicatedInteger;
Coherence                  *cm;
MultiCast                  *com;
```

```
void main (int argc, char **argv)
{
    bool passive  = (argc < 2);
    unsigned int nbInvocation = (passive ? 0 : atoi(argv[1]));
    Chronometre chrono;

    if (getenv("CORELOG"))
        coreLog = new ofstream(getenv("CORELOG"));

    char *consistency = getenv("CONSISTENCY");
    char *coreset     = getenv("WEAKSET");

    assert( (consistency != NULL) &&
            ( (strcmp(consistency, "LinearisableConsistency") == 0) ||
              (strcmp(consistency, "Linear") == 0) ||
              (strcmp(consistency, "Hybrid") == 0) ));

    initCore();
    theReplicatedInteger = new ReplicatedInteger(nbInvocation);

    com = new MultiCast(ofName);

    if (strcmp(consistency, "LinearisableConsistency") == 0)
        cm = new LinearisableConsistency (ofName,
                                          theReplicatedInteger, NULL, com);
    else if (strcmp(consistency, "Linear") == 0)
        cm = new Linear2 (ofName, theReplicatedInteger, NULL, com);
    else if (strcmp(consistency, "Hybrid") == 0)
        cm = new Hybrid (ofName, theReplicatedInteger, NULL, com);
    else
        assert (0);

    IntegerCoreSet myCoreSet;
    if (coreset && (strcmp(coreset, "INCDEC") == 0))
        myCoreSet.coreset =  IntegerCoreSet::IncDecWeakSet;
    else
        myCoreSet.coreset = IntegerCoreSet::ReadWeakSet;

    sacFactory = new SynchronousActivityFactory(cm->getId());
    theReplicatedInteger->bind(sacFactory, cm, myCoreSet);

    if (!passive) {
        BenchBarrier bb = BenchBarrier(atoi(argv[2]));
        com->subscribe(bb, groupChangeEvent);
        bb.wait(com->size()); // after the bind so we are member.
        com->unsubscribe(bb, groupChangeEvent);

        chrono.start();

        while (nbInvocation-- > 0) {
            if ((nbInvocation%100) == 0)
                cerr << "(* " << theReplicatedInteger->value << " *)" << endl;
            theReplicatedInteger->dec();
        }

        chrono.stop();
        cerr << chrono;
    } else
        thread_suspend();

    cerr << "That,s all folks\n";

    delete theReplicatedInteger;
    delete coreLog;
}
```

FIG. 12.5: Code du compteur répliqué instrumenté ayant servi à la collecte des mesures montrées ci-après.

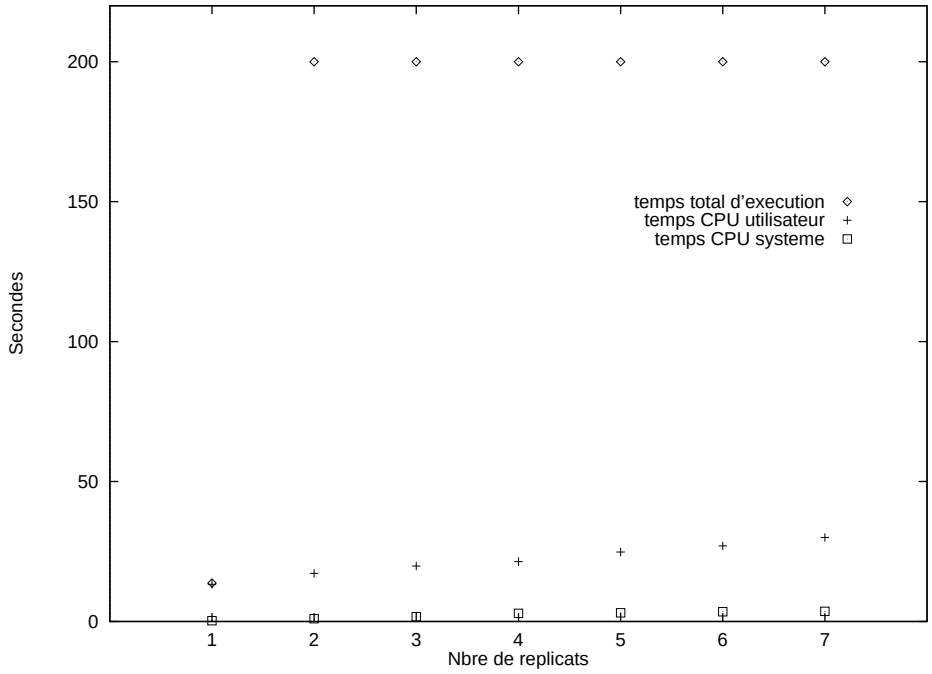


FIG. 12.6: Évolution du coût d'exécution de 1 000 invocations `strong` en fonction du nombre de réplicats.

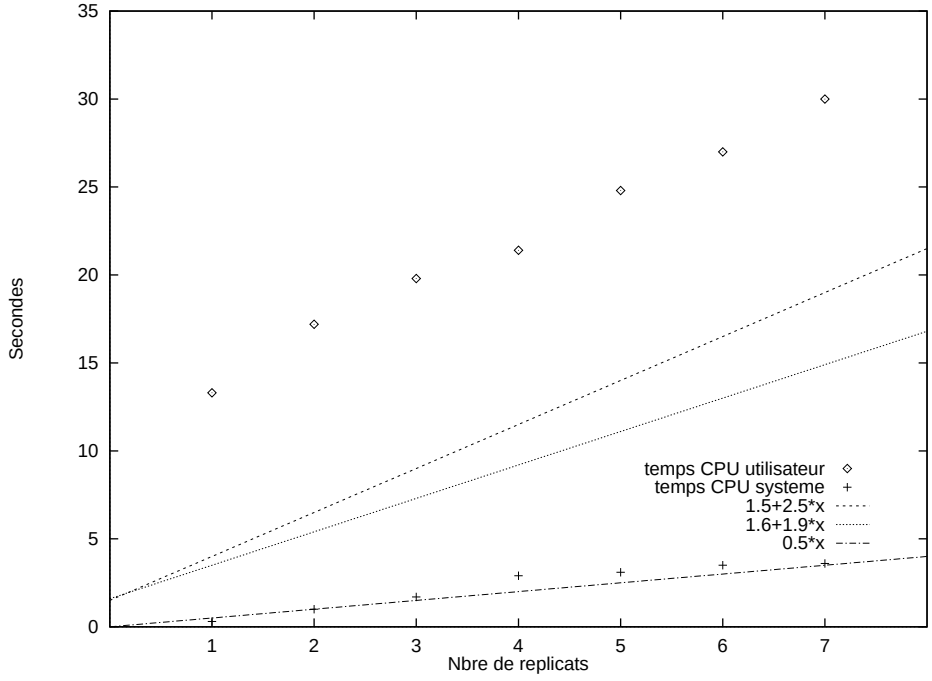


FIG. 12.7: Évolution du coût d'exécution de 1 000 invocations `strong` en fonction du nombre de réplicats. Agrandissement de la figure 12.6 montrant les temps d'exécution utilisateur et système.

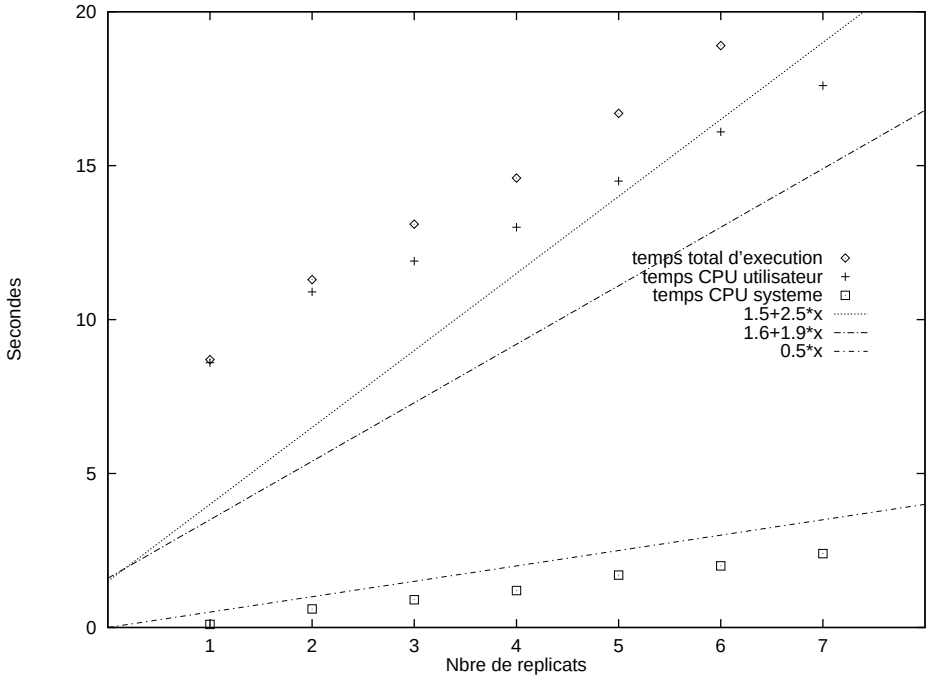


FIG. 12.8: Évolution du coût d'exécution de 1 000 invocations `weak` en fonction du nombre de réplicats.

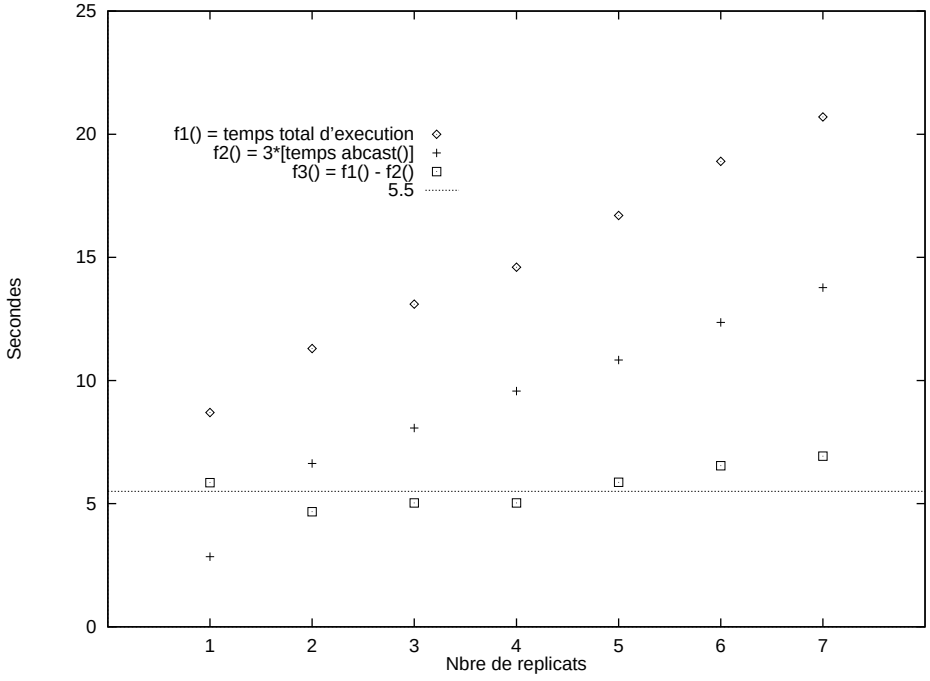


FIG. 12.9: Évolution comparée du coût d'exécution total et du coût associé à la primitive de diffusion employée, sur l'exécution de 1 000 invocations `weak`. L'échantillon *f3()* est obtenu en soustrayant les valeurs de ces deux coûts. Le surcoût semble à peu près constant, de l'ordre de 5 msec par invocation.

12.4 Les surcoûts de l'architecture

Deux éléments de la factorisation génèrent clairement un surcoût : la création des activités et la création des intentions. Le coût cumulé de création et de suppression d'une activité est de l'ordre de 250 micro-secondes, sur notre environnement. Celui associé à la création des intentions peut varier considérablement en fonction des choix du programmeur, mais devrait être du même ordre que celui des activités. Dans les applications mesurées, les intentions sont instantiées une seule fois, lors de la liaison, puis réutilisées. Ceci n'est bien sûr possible que parce que les intentions ne réifient pas les arguments d'invocation.

Ce surcoût est obligatoire et constant, même en présence d'un unique réplicat. Il n'est pas négligeable en regard des temps d'exécution locaux, mais le devient dès lors que plusieurs réplicats sont présents.

12.5 Résumé

Nous avons dans ce chapitre, présenté les coûts significatifs de la gestion de cohérence, tels qu'obtenus sur notre prototype. Ces coûts ont permis d'illustrer les gains que peut apporter l'adaptabilité de la gestion de cohérence dans un cadre pourtant relativement classique : la spécification à l'exécution d'un noyau d'optimisation dans le cadre de la cohérence hybride.

Les mesures présentées ont été produite sur un prototype loin d'être optimisé : les choix de mise en œuvre ont systématiquement été fait en faveur de la facilité du déboggage et la compréhension du fonctionnement de la plate-forme. Si les performances sont dans l'absolu médiocres, les mesures montrent clairement que le responsable en est la plate-forme de communication de groupe et non la factorisation.

Malgré cette immaturité de la plate-forme, les surcoûts sont dans des ordres de grandeur nettement inférieurs aux gains obtenus en adaptant la gestion de cohérence. Ceci alors même que ce code de gestion de cohérence a pu être réutilisé tel quel pour d'autres applications.

Chapitre 13

Conclusion

Dans ce document, nous avons présenté une architecture que nous pensons adaptée à la mise en œuvre d'applications coopérants à distance. La démarche qui nous a guidé est schématisée par la figure 13.1.

L'architecture développée se caractérise par la présence d'un composant système choisi à l'exécution par l'application. Ce composant intègre la partie la plus complexe du code d'une application répartie : celle ayant trait à l'algorithmique répartie et en particulier, à la gestion de la concurrence, à la gestion de la réplication et à la gestion de la communication.

Cette architecture offre deux propriétés importantes : l'adaptabilité et la réutilisabilité. L'adaptabilité permet de coopérer de façon performante à grande distance. La réutilisabilité permet de diminuer les coûts de développement et de produire des applications plus robustes et mieux optimisées.

Au-delà de ces deux propriétés, notre thèse contribue à l'état de l'art sur trois points :

- la conception d'un modèle de partage, que nous désignons par *réplication coopérante* ;
- la description de principes de *factorisation*, qui permettent de mettre en œuvre ce modèle en assurant les deux propriétés clé, adaptabilité et réutilisabilité ;
- la conception et la réalisation de l'architecture **Core** qui intègre ces principes.

Chacun de ces trois points a été développé avec le souci d'adéquation au problème original : la coopération au-dessus d'un réseau étendu. Ils forment le premier maillon permettant de construire des caches coopérants à grande échelle, plus particulièrement ceux gérant des informations évolutives non pérennes.

Bien que la problématique soit en elle-même relativement nouvelle, elle intègre de nombreux aspects déjà très étudiés, en particulier dans le domaine de la concurrence, de la répartition et de la réplication. La conception de chacun de ces trois points dans **Core** s'appuie constamment sur l'état de l'art de ces différents domaines d'études.

À ceci s'ajoute une dernière contribution : une étude des propriétés associées aux cohérences dites faibles. La conception du modèle de partage, de même que celle des principes de factorisation nous a imposé une réflexion sur ce qu'est la cohérence d'un groupe d'agents coopérants. Cette réflexion intègre le développement d'un nouveau formalisme, le *modèle d'exécution* **Core**, adapté à l'expression de propriétés réparties d'ordre, dans le cadre de la réplication coopérante. Cette réflexion nous a permis de structurer le champ des cohérences faibles ; elle nous a permis de cerner et de décrire quelques propriétés qui pilotent les interactions entre membres coopérants.

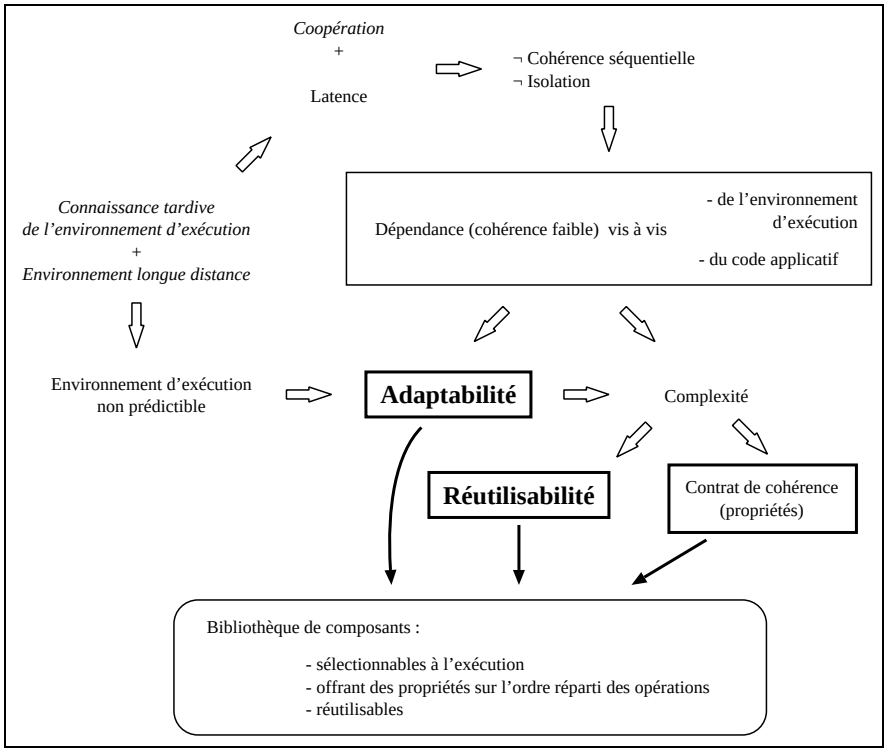


FIG. 13.1: Synopsis de la thèse. Les données originales du problème sont en italiques. Les flèches représentent notre démarche intellectuelle. Les rectangles identifient les aspects dans lesquels la thèse apporte une contribution. Les rectangles gras correspondent aux réalisations qui permettent d'arriver à notre solution, la bibliothèque de composants, représentée par un rectangle à bord rond

13.1 Limitations – Problèmes ouverts

Le sujet défriché dans ce document est vaste, et notre contribution ne forme que quelques éléments de la solution. Nous présentons brièvement quatre limitations, dont la levée peut permettre une application plus satisfaisante de notre solution.

13.1.1 Composition de gestionnaires de cohérence

Nous avons considéré, dans cet ouvrage, le cas d'un gestionnaire de cohérence unique. Ceci présuppose une information partagée structurée comme une collection restreinte et homogène en termes de contraintes de cohérence. Si ceci semble adapté à la collaboration en groupes de taille limitée, d'autres applications telles que les systèmes d'entreprise, les simulations à large échelle de champs de bataille ou les mondes virtuels sur Internet manipulent de grandes collections d'informations disparates ; ces applications intègrent, de façon évidente, des aspects coopératifs. Il est vraisemblable, que ces applications nécessitent un regroupement (« cluster ») de ces informations, selon des critères de cohérence adaptés à leurs rôles ou à leur importance dans l'application. Si chacun de ces regroupements est géré par un gestionnaire de cohérence, que peut-on dire de la cohérence qui résulte de cette composition de contrats, comme de cette composition de mises en œuvre ?

Nous ne pouvons répondre en l'état actuel à cette question. Certains éléments de réponse

existent, car le problème s'est déjà posé dans le cadre des objets atomiques : WEIHL, lorsqu'il présente ses trois propriétés d'atomicité locale, *statique*, *atomique* et *hybride* [Wei89], pose le problème de transactions qui interagiraient avec une collection d'objets implantant ces différentes formes d'atomicité, et ce dans un même contexte d'exécution. GUERRAOUI dans [Gue94] en définissant la propriété d'o-atomicity (« object atomicity ») apporte une réponse à ce problème dans le cadre atomique.

13.1.2 Duplication des opérations

La réplication coopérante partage avec la réplication active le fait de réexécuter les opérations sur chacun des réplicats. Dans le cas ou un objet répliqué peut invoquer un autre objet répliqué, cette réexécution introduit des invocations parasites : la réexécution d'une opération distante génère une invocation sur le réplicat d'un objet répliqué distinct.

Ce problème a été identifié dans le cadre traditionnel de la tolérance aux fautes assurée de façon transparente. Nos applications ainsi que l'architecture **Core** s'y prêtent moins. D'une part, l'information répliquée est circonscrite, d'autre part l'architecture ne la réplique pas de façon transparente. En dernier recours, nous pouvons utiliser les techniques de filtrage présentées par MAZOUNI [Maz96]. Bien que le modèle de réplication coopérante diffère notablement de celui envisagé par MAZOUNI, nous pensons que la structure des objets d'accès, et la liberté permise par l'architecture dans la réification des activités, permettent l'implantation des différentes formes de filtrages évoquées. Là encore, ce problème nécessite d'être étudié.

13.1.3 Stratégies optimistes

Le développement de l'architecture s'est effectué autour de stratégies pessimistes. Ce choix a été délibéré, en particulier parce qu'il permet de séparer clairement les effets liés aux propriétés d'ordre sur les opérations de l'asynchronisme de la mise en œuvre de ces opérations.

Bien que nous pensions qu'à long terme les systèmes seront totalement et en permanence connectés, il est probable que l'approche optimiste reste une stratégie adaptée à certaines applications, y compris dans le domaine collaboratif. L'architecture **Core** se prête-elle à de telles stratégies ?

Nous ne pouvons répondre de façon définitive à cette question. Tout au plus, peut-on constater que les *activités* comme les *intentions* de l'architecture **Core** peuvent représenter, *a priori*, toutes les informations relatives à une opération. Dans une stratégie pessimiste, elles sont utilisées essentiellement comme des verrous typés. Elles peuvent a priori intégrer les estampilles ou les vecteurs de version utilisés dans les stratégies optimistes. Les besoins requis pour les phases de certification et de réconciliation doivent clairement être étudiés. En particulier, les rapports étroits que la réconciliation entretient avec la représentation de données peut poser des problèmes non triviaux ; la définition systématique d'opérations de compensation peut éviter ces problèmes. Une étude approfondie, assortie d'une mise en œuvre convaincante, est nécessaire pour conclure.

13.1.4 Intégration avec le recouvrement et avec la persistance

Nos travaux se sont focalisés sur la gestion de la cohérence. La cohérence interagit avec d'autres propriétés, en particulier le recouvrement et la persistance de l'information partagée. La définition d'un composant autonome, le gestionnaire de cohérence, ne suffit pas en

soi à assurer l'intégration avec ces fonctions qui, toutes deux, ont la vocation de modifier l'ordonnancement des opérations sur la donnée partagée.

Ce problème a été étudié dans le cadre des objets atomiques [Wei84] et, plus récemment, dans CORBA. La spécification du service de transaction [COSb] statue des interactions entre la gestion de l'ordonnancement, du recouvrement et de la persistance. Bien que nous ayons accordé une attention certaine à ce problème, en particulier dans la définition de l'interface des activités, une mise en œuvre complète de transactions, avec support du recouvrement, est indispensable pour conclure.

Notre intuition est que le problème est difficile. Rendre persistante une information, nécessite a priori une certaine forme de synchronisation : le système doit pouvoir obtenir une image unique de l'information qui puisse assurer la correction des processus qui y accéderont par la suite. Cette simple contrainte est à même de faire perdre tout le bénéfice des cohérences faibles.

13.2 Conclusion

Il nous a fallu près de six ans et deux prototypes complets, assortis de nombreuses modifications, pour mener à bien ce travail. Le premier prototype comprenait plus de 40 000 lignes de code ; le dernier, librement accessible sur `ftp` (cf. annexe B), en comprend moins de 20 000, et ce, en offrant une facilité d'utilisation, des fonctionnalités et une robustesse sans commune mesure avec le premier prototype.

Nous avons conscience des limites de notre validation ; celle-ci réclamerait des mesures en présence de latences importantes. Néanmoins, nous avons retiré de ce travail de thèse, de nombreux enseignements. Nous aimerions en partager cinq qui, bien qu'ils puissent paraître triviaux, nous semblent importants :

- Dans le cadre de la réplication totale, le protocole de réplication contrôle la perception des opérations distantes.
- Le contrôle de concurrence repose sur la perception de toutes les opérations accédant à l'information partagée, locales comme distantes. Un contrôle de concurrence réparti doit donc intégrer la gestion de la réplication.
- Si l'on entend par cohérence, des propriétés de correction, alors la gestion de cohérence se doit d'assurer des *propriétés* sur l'ordre des opérations. Ces propriétés sont traditionnellement obtenues en limitant la concurrence des opérations. À ce titre, la gestion de cohérence embrasse le contrôle de concurrence, et donc la gestion de la réplication.
- Les propriétés d'initialisation font partie intégrante des propriétés de cohérence. Dans les faits, on ne peut construire le protocole d'initialisation des nouveaux membres, indépendamment du protocole gérant la coopération entre les membres déjà initialisés.
- La programmation multi-tâches, dans le domaine de la communication de groupe, apporte infiniment plus de problèmes qu'elle n'en résout. C'est une dure expérience que nous avons comprise avec lenteur. Nous ne saurions trop conseiller aux futurs concepteurs impliqués dans la communication de groupe de suivre les évolutions de l'état de l'art dans ce domaine, tel Isis, Horus et Ensemble.

Annexe A

Mesures brutes

Cette annexe expose les chiffres bruts ayant servi à l'établissement des courbes présentées dans la thèse. Tous les temps sont en secondes.

Les mesures ont été effectuées sur sept stations DEC 3000/500 munies d'un processeur Alpha 21064 cadencé à 150 MHz[1] tournant sous OSF/1 3.0 et connectées par un réseau FDDI à 100 Mbits/secs.

A.1 Plate-forme de communication

A.1.1 Primitives de diffusion

Temps d'exécution de 10 000 diffusions. Ces temps correspondent à l'envoi de 10 000 messages d'une donnée utilisateur de 4 octets (taille d'un entier sous DEC Alpha) mais aussi au traitement correspondant à la réception d'un nombre indéterminé de ces messages (les messages sont délivrés aussi à l'émetteur).

Le séquenceur est toujours l'un des membres du groupe, les temps obtenus avec un unique réplicat intègrent donc le coût du séquenceur. Dans tous les autres cas, le séquenceur n'est pas géré par le processus mesuré.

Nombre de réplicats	Temps en secondes Total/CPU utilisateur/CPU système	
	fbcast()	*abcast()*
1	6.2 / 6.1 / 0.1	9.5 / 9.5 / 0.0
2	12.2 / 11.5 / 0.6	22.1 / 14.4 / 1.7
3	17.1 / 15.7 / 1.2	26.9 / 18.2 / 1.7
4	21.5 / 19.5 / 1.7	31.9 / 22.2 / 1.7
5	25.8 / 23.2 / 2.1	36.1 / 26.4 / 1.7
6	31.2 / 27.9 / 2.8	41.2 / 30.6 / 1.8
7	36.4 / 32.3 / 3.5	45.9 / 35.5 / 1.7

[1] L'une d'entre elle est cadencé à 160MHz et munie de 128 Mo de mémoire vive. Les mesures obtenues ne diffèrent pas sensiblement de celles obtenues sur les autres machines, dès lors que l'application utilise plusieurs réplicats.

A.1.2 Primitives de multi-RPC

Temps d'exécution de 1 000 multi-RPC nuls. Un multi-RPC est un appel de procédure à distance invoquant chacun des membres du groupe. Dans le cas du multi-RPC nul, l'invocation se fait sans argument et la réponse est attendue bien qu'elle soit de taille nulle. La colonne de gauche est obtenue lorsque la diffusion de l'invocation se fait avec la primitive *fbcast()*. La colonne de droite est obtenue lorsque la diffusion de l'invocation se fait avec la primitive *abcast()*.

Nombre de réplicats	Temps en secondes Total/CPU utilisateur/CPU système	
	fbcast()	*abcast()*
1	3.9 / 3.6 / 0.2	4.2 / 4.0 / 0.2
2	6.7 / 5.7 / 0.8	200 / 5.6 / 0.9
3	9.0 / 7.4 / 1.4	200 / 7.8 / 1.1
4	11.3 / 9.1 / 1.8	200 / 9.2 / 1.7
5	13.7 / 11.0 / 2.1	200 / 11.0 / 1.6
6	16.4 / 12.9 / 2.9	200 / 12.4 / 1.9
7	19.0 / 14.8 / 2.3	200 / 14.7 / 1.6

A.2 Gestion de cohérence

Temps d'exécution de 1 000 invocations de l'opération `dec` sur un compteur répliqué. La colonne de gauche est obtenue lorsque l'opération est étiquetée `weak`. La colonne de droite est obtenue lorsque l'opération est étiquetée `strong`. Le gestionnaire utilisé est une instance de la classe `LinearisableConsistency` : chaque opération consiste en une requête de début d'activité, la diffusion des mises à jour et la diffusion de la fin de l'activité. Les diffusions se font avec la primitive *abcast()*. Les requêtes d'opérations `strong` se font avec un multi-RPC basé aussi sur *abcast()*. Les requêtes d'opérations `weak` se font localement puis sont diffusée avec un *abcast()*.

Nombre de réplicats	Temps en secondes Total/CPU utilisateur/CPU système	
	`weak`	`strong`
1	8.7 / 8.6 / 0.1	13.7 / 13.3 / 0.3
2	11.3 / 10.6 / 0.6	200 / 17.2 / 1.0
3	13.1 / 11.9 / 0.9	200 / 19.8 / 1.7
4	14.6 / 13.0 / 1.2	200 / 21.4 / 2.9
5	16.7 / 14.5 / 1.7	200 / 24.8 / 3.1
6	18.9 / 16.1 / 2.0	200 / 27.0 / 3.5
7	20.7 / 17.6 / 2.4	200 / 30.0 / 3.6

Le tableau suivant présente les temps d'exécution de 1 000 invocations de l'opération `dec`, étiquetée `weak` sur un compteur répliqué. Le gestionnaire utilisé est une instance de la classe `Hybrid` : les requêtes d'opérations `strong` sont traitées de façon identique au gestionnaire `LinearisableConsistency`. Par contre, toutes les autres diffusions utilisent la primitive *fbcast()*.

Nombre de réplicats	Temps en secondes Total/CPU utilisateur/CPU système
1	8.6 / 8.5 / 0.1
2	11.0 / 10.4 / 0.5
3	13.0 / 11.7 / 0.9
4	14.5 / 12.8 / 1.2
5	16.5 / 14.4 / 1.6
6	18.8 / 16.0 / 2.0
7	20.0 / 17.0 / 2.4

Annexe B

Diffusion du prototype Core

L'environnement prototype **Core** est téléchargeable à l'adresse suivante : `ftp://ftp.inria.fr/INRIA/Projects/SOR/misc/boar/core.tar.gz`. Ce prototype est sous copyright I.N.R.I.A. Il est distribué selon les licences « GNU Public License » (GPL) et « Library GNU Public License » (LGPL). Bien que la version actuelle de **Core** soit le second prototype réalisé, il comporte encore de nombreux bogues, qui le rendent difficile d'utilisation. Deux parties de ce prototype, la librairie de tâches et le service de nom sont, par contre, diffusées séparément depuis plus d'un an, et paraissent robustes.

Core est fourni avec une documentation partielle, de nombreux jeux de tests, quelques programmes écrits en Emacs-Lisp pour tester le respect de divers ordonnancements d'opérations sur les journaux d'exécutions. Les deux applications prototypes décrites dans le manuscrit sont incluses dans le prototype.

Core tourne sur SunOS 4.1.3, Solaris 2.5.1 et DEC OSF/1 3.0. Outre la librairie de gestionnaires de cohérence (2), il comporte un service de nommage, un service d'emballage et de déballage, une plate-forme de communication de groupe garantissant un modèle de synchronie virtuelle faible avec de la communication point à point, de la diffusion FIFO, de la diffusion atomique et de la diffusion totale (*gbcast()*). De nombreux outils sont disponibles avec la plate-forme, comme une librairie de tâches portable, une bibliothèque de multi-RPC flexibles, des objets d'accès, des activités et des intentions prédéfinies pour les types de données courants.

Pour plus d'informations, le lecteur est invité à télécharger le logiciel et à consulter la documentation, en cours d'élaboration et disponible en de nombreux formats (texinfo, info, HTML et Postscript).

Bibliographie

[AC92] M. S. Atkins et M. Y. Coady. Adaptable concurrency control for atomic data types. *ACM Transactions on Computer Systems*, 10(3):190–225, août 1992.

[ACFW93] Hagit Attiya, Soma Chaudhuri, Roy Friedman, et Jennifer L. Welch. Shared memory consistency conditions for non-sequential execution: Definition and programming strategies. Dans *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 241–250, 1993.

[ACM87] ACM SIGOPS. *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, volume 21 des *Operating Systems Review*, Stouffer Austin Hotel, Austin, Texas, USA, novembre 8–11 1987. ACM Press.

[ACM90] ACM SIGMOD. *Proceedings of the 1990 International conference on the Management of Data*, ACM SIGMOD Records, Atlantic City, NJ, USA, mai 1990. ACM Press.

[ACM93] ACM SIGOPS. *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Operating Systems Review (27)5, The Grove Park Inn and Country Club, Asheville, NC, USA, décembre 5–8 1993. ACM Press.

[ACM97] ACM SIGOPS. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Operating Systems Review (31)5, Saint-Malo, France, octobre 5–8 1997. ACM Press.

[AF92] Hagit Attiya et Roy Friedman. A correctness condition for high-performance multiprocessors. Dans *Proceedings of the 24th ACM Symposium on Theory of Computing*, pages 679–690, Victoria, Canada, 1992.

[AH93] S. V. Adve et M. D. Hill. A unified formalization of four shared-memory models. *IEEE Trans. on Parallel and Distributed Systems*, 4(6):613–624, juin 1993.

[AHJ91] Mustaque Ahamad, Phillip W. Hutto, et Ranjit John. Implementing and programming causal distributed shared memory. Sous la direction de IEEE, *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 274–281, Arlington, TX, USA, mai 1991.

[Aks96] Mehmet Aksit. Composition and separation of concerns in the object-oriented model. *ACM Computing Surveys*, 28A(4), décembre 1996.

[AW94] Hagit Attiya et Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, mai 1994.

[BEH+94] Steve Bellovin, Ray Essick, Mark Horton, Brian Kantor, Phil Lapsley, Bob Page, Rich Salz, Tom Truscott, et Larry Wall. Usenet history. [news.announce.newusers], novembre 1994. Voir `http://www.arnes.si/usenet/history.html` et `http://www.md.chalmers.se/~thomas/KF1/usenet/hist.html`.

[BG83] Alain Bouvier et Michel George. *Dictionnaire des Mathémathiques*. Presses Universitaires de France, 1983.

[BGM90] Daniel Barbará et Hector Garcia-Molina. The case for controlled inconsistency in replicated data. *IEEE Computer Society Technical Committee on Operating Systems and Application Environments Newsletter*, 4(3):8–11, 1990.

[BGN76] A. W. Burks, H. H. Goldstine, et J. V. Neumann. *Collected Works*, volume 5, chapitre Preliminary Discussion of The Logical Design of an Electronic Computing Instrument – (1946), pages 34–80. Pergamon – Oxford, 1976.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, et Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[BHJ+87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, et Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, janvier 1987.

[Bir85] Kenneth P. Birman. Replication and fault-tolerance in the ISIS system. Dans *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, volume 19 des *Operating Systems Review*, pages 79–86, Orcas Island, Washington, USA, décembre 1–4 1985. ACM SIGOPS, ACM Press.

[Bir94] Kenneth P. Birman. Integrating runtime consistency models for distributed computing. *Journal of Parallel and Distributed Computing*, 23:158–176, 1994.

[Bir96] Kenneth P. Birman. *Building Secure and Reliable Network Applications.* Manning, 1996.

[BJ87] Kenneth P. Birman et Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. Dans *Proceedings of the 11th ACM Symposium on Operating Systems Principles* [ACM87], pages 123–138.

[BJS87] Kenneth P. Birman, Thomas A. Joseph, et Frank Schmuck. Isis documentation: Release 1. Technical Report 87–849, Department of Computer Science, Cornell University, Ithaca, New York (USA), juillet 1987.

[BK89] Henri E. Bal et M. Frans Kaashoek. Experience with distributed progamming in Orca. Technical Report IR-200, Department of Mathematics and Computer Science, Amsterdam (The Netherlands), septembre 1989.

[BK91] Naser S. Barghouti et Gail E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, septembre 1991.

[BM93a] Özalp Babaoğlu et Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. Technical Report UBLCS-93-1, University of Bologna, 1993.

[BM93b] Özalp Babaoğlu et Keith Marzullo. *Distributed Systems*, chapitre 4 – Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms, pages 55–96. Frontier Series. Addison-Wesley, acm press edition, 1993.

[BN97] Philip A. Bernstein et Eric Newcomer. *Principles of Transaction Processing.* Morgan-Kaufmann, 1997.

[BNOW93] Andrew Birrell, Greg Nelson, Susan Owicki, et Edward Wobber. Network objects. Dans *Proceedings of the 14th ACM Symposium on Operating Systems Principles* [ACM93], pages 217–230.

[BR90] B. R. Badrinath et Kriti Ramamritham. Performance evaluation of semantics-based multilevel concurrency control protocols. Dans *Proceedings of the 1990 International conference on the Management of Data* [ACM90], pages 163–172.

[BZ91] Brian N. Bershad et Matthew J. Zekauskas. Midway : Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Department of Computer Science, Carnegie Mellon University, septembre 1991.

[CALM97] Miguel Castro, Atul Adya, Barbara Liskov, et Andrew C. Myers. HAC: Hybrid adaptive caching for distributed storage systems. Dans *Proceedings of the 16th ACM Symposium on Operating Systems Principles* [ACM97], pages 102–115.

[Cat91] R.G.G. Catell. *Object Data Management.* Addison–Wesley, 1991.

[CBCP95] Bernadette Charron-Bost, Robert Cori, et Antoine Petit. Introduction à l'algorithmique des objets partagés. Technical Report LIX/RR/95/10, Laboratoire d'informatique de l'Ecole Polytechnique, 1995. `http://www.labri.u-bordeaux.fr/~cori/Articles/registres.dvi`.

[CBZ91] John B. Carter, John K. Bennett, et Willy Zwaenepoel. Implementation and performance of Munin. Dans *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, volume 25 des *Operating Systems Review*, pages 152–164, Pacific-Grove, USA, octobre 1991. ACM SIGOPS, ACM Press.

[CD89] Raymond Chen et Partha Dasgupta. Linking consistency with object/thread semantics. Sous la direction de IEEE, *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 121–128, Newport Beach, CA, USA, juin 1989.

[CFZ94] M. J. Carey, M. J. Franklin, et M. Zaharioudakis. Fine-grained sharing in a page server OODBMS. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(2):359–370, juin 1994.

[Chi95] Shigeru Chiba. A metaobject protocol for C++. Dans *OOPSLA '95, Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, ACM SIGPLAN Notices 30(10), pages 285–299, Austin, Texas, USA, octobre 1995. ACM SIGPLAN, ACM Press.

[Chi96] Shigeru Chiba. OpenC++ programmer's guide for version 2. Technical Report SPL-96-024, Xerox PARC, 1996.

[COSa] Object Management Group. *Concurrency Control Service.*

[COSb] Object Management Group. *Object Transaction Service.*

[CP92] Shu-Wie F. Chen et Calton Pu. An analysis of replica control. Dans *Proceedings of the 2nd Workshop on the Management of Replicated Data*, pages 22–25, Monterey, CA, USA, novembre 1992. IEEE.

[CPS93] Steve J. Caughey, Graham D. Parrington, et Santosh K. Shrivastava. SHADOWS - A flexible support system for objects in distributed systems. Technical Report BROADCAST/TR93–30, ESPRIT Basic Research Project BROADCAST, août 1993.

[CR90] Panayiotis K. Chrysanthis et Kriti Ramamritham. ACTA: A framework for specifying and reasoning about transaction structure and behavior. Dans *Proceedings of the 1990 International conference on the Management of Data* [ACM90], pages 194–203.

[CR94] P. K. Chrysanthis et K. Ramamritham. Synthesis of extended transaction models using acta. *ACM Transactions on Databases Systems*, 19(3):450–491, septembre 1994.

[CS93] David R. Cheriton et Dale Skeen. Understanding the limitations of causally and totally ordered communication. Dans *Proceedings of the 14th ACM Symposium on Operating Systems Principles* [ACM93], pages 44–57.

[CT96] Tushar Deepak Chandra et Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, mars 1996.

[DAM+91] Partha Dasgupta, R. Ananthanarayanan, Sathis Menon, Ajay Mohindra, et Raymond Chen. Distributed programming with objects and threads in the Clouds system. *Computing Sytems*, 4(3):243–276, 1991.

[DDS87] Danny Dolev, Cynthia Dwork, et Larry StockMeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, janvier 1987.

[DHMR96] P. Dechamboux, D. Hagimont, J. Mossiere, et X. Rousset de Pina. The Arias distributed shared memory: An overview. *Lecture Notes in Computer Science*, 1175:56–73, 1996.

[DHW88] David Detlefs, Maurice Herlihy, et Jeannette Wing. Inheritance of synchronization and recovery properties in Avalon/C++. *IEEE Computer*, 21(12):57–69, décembre 1988.

[Din96] Adam Dingle. Cache consistency in HTTP 1.1 proposed standard. Dans *Web Caching on Internet*. ICM Workshop on Web Caching (Varsovie), octobre 1996. see `http://www-sor.inria.fr/mirrors/wwc96/talk4/`.

[DSB86] Michel Dubois, Christoph Scheurich, et Fayé Briggs. Memory access buffering in multiprocessors. Dans *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, juin 1986.

[DSB88] Michel Dubois, Christoph Scheurich, et Fayé Briggs. Synchronisation, coherence and event ordering in multiprocessors. *IEEE Computer*, pages 9–21, février 1988.

[eco95] *ECOOP '95, Proceedings of the 9th European Conference on Object-Oriented Principles*, volume 952 des *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[EGLT76] K. Eswaran, Jim Gray, R. Lorie, et I. Traiger. The notion of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[Frø92] Svend Frølund. Inheritance of synchronisation constraints in concurrent object-oriented programming languages. Sous la direction de Ole Lehrmann Madsen, *ECOOP '92, Proceedings of the 6th European Conference on Object-Oriented Principles*, volume 615 des *Lecture Notes in Computer Science*, pages 185–196, Utrecht, Netherlands, 1992. Springer-Verlag.

[Fuj90] Richard M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, octobre 1990.

[Gaa95] Jostein Gaarder. *Le monde de Sophie*, chapitre David Hume. Seuil, 1995.

[GGM95] Benoît Garbinato, Rachid Guerraoui, et Karim R. Mazouni. Implementation of the GARF replicated object plateform. *Distributed Systems Engineering Journal*, 2:14–27, 1995.

[Gif79] David K. Gifford. Weighted voting for replicated data. Dans *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, Operating Systems Review, pages 150–162, Pacific Grove, CA, USA, décembre 1979. ACM SIGOPS, ACM Press.

[GL91] Richard Golding et Darrell D. E. Long. Accessing replicated data in a large-scale distributed system. Technical Report UCSC-CRL-91-01, Computer Research Laboratory, UCSC, Santa Cruz, CA, USA, janvier 1991.

[GL93] Richard A. Golding et Darell E. Long. Modeling replica divergence in a weak-consistency protocol for global-scale distributed data bases. Technical Report CRL-93-03, UCSC, 1993.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, et John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. Dans *Proceedings of the 17th International Symposium on Computer Architecture*, ACM SIGARCH Computer Architecture News (18)2, pages 15–26, Seattle, WA, USA, mai 1990. ACM SIGARCH, IEEE Computer Society Press.

[GMW82] H. Garcia-Molina et G. Wiederhold. Read-only transactions in a distributed database. *ACM Transactions on Databases Systems*, 7(2):209–234, juin 1982.

[Gou91] Yvon Gourhant. *Outils pour la Programmation d'Objets Fragmentés*. Thèse de doctorat, Université Paris 6 Pierre-et-Marie-Curie, Paris 6 – France, juin 1991.

[GR83] Adele Goldberg et David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley, 1983.

[GR93] Jim Gray et Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan-Kaufmann, 1993.

[GSW86] Irene Greif, Robert Seliger, et William Weihl. Atomic data abstractions in a distributed collaborative editing system. Dans *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 160–172. ACM, ACM, janvier 1986.

[Gue92] Rachid Guerraoui. *Programmation répartie par objets : études et propositions.* PhD thesis, Université Paris XI-Orsay, octobre 1992.

[Gue94] Rachid Guerraoui. Modular atomic objects. *Theory and Practice of Object Systems*, 24(4):1–12, mai 1994.

[Guy91] Richard G. Guy. *Ficus: a very large reliable distributed file system.* PhD thesis, UCLA Computer System Department, 1991.

[GWB91] Richard P. Gabriel, Jon L. White, et Daniel G. Bobrow. CLOS: Integrating object-oriented and functional programming. *Communications of the ACM*, 34(9):29–38, septembre 1991.

[Her90] Maurice Herlihy. Concurrency versus availability: Atomicity mechanisms for replicated data. *Journal of the ACM*, 37(2):257–278, avril 1990.

[HJT+93] Carl Hauser, Christian Jacobi, Marvin Theimer, Brent Welch, et Mark Weiser. Using threads in interactive systems: A case study. Dans *Proceedings of the 14th ACM Symposium on Operating Systems Principles* [ACM93], pages 94–105.

[HLS97] Olof Hagsand, Rodger Lea, et Märten Stenius. Using spatial techniques to decrease message passing in a distributed VR system. Sous la direction de Rikk Carey et Paul Strauss, *VRML 97: Second Symposium on the Virtual Reality Modeling Language*, New York City, NY, février 1997. ACM SIGGRAPH / ACM SIGCOMM, ACM Press.

[HM90] Joseph Y. Halpern et Yoram Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, juillet 1990.

[Hum86] David Hume. *A Treatise of Human Nature*, chapitre Part I. Sect. IV. – Of the Connexion or Association of Ideas. Green and Grose, 1886. En ligne à `http://www.utm.edu:80/research/hume/wri/treatise/treatise.htm`.

[HW90] Maurice. P. Herlihy et Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, juillet 1990.

[HW91] Maurice P. Herlihy et William E. Weihl. Hybrid concurrency control for abstract data types. *Journal of Computer and System Sciences*, 43(1):25–61, août 1991.

[Jar92] Bruno Jarrosson. *Invitation à la philosophie des sciences*, chapitre David Hume. Points Sciences. Seuil, 1992.

[JBW+87] David Jefferson, Brian Beckman, Fred Wieland, Leo Blume, Mike DiLoreto, Phil Hontalas, Pierre Laroche, Kathy Sturdevant, Jack Tupman, Van Warren, John Wedel, Herb Younger, et Steve Bellenot. Distributed simulation and the time warp operating system. Dans *Proceedings of the 11th ACM Symposium on Operating Systems Principles* [ACM87], pages 77–93.

[KCZ92] Peter Keleher, Alan L. Cox, et Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. Dans *Proceeding of the 19th International Symposium on Computer Architecture*, pages 13–21, Gold Coast (Australia), mai 1992.

[KdRB91] Gregor Kiczales, Jim des Rivières, et Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[Kic96] Gregor Kiczales. Beyond the black box: Open implementation. *IEEE Software*, 13(1):8–11, janvier 1996.

[KL92] Gregor Kiczales et John Lamping. Operating systems: Why object-oriented? Dans *Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems*, pages 25–30, Asheville, NC, USA, décembre 9–10 1992. IEEE Computer Society Press.

[KP] Gregor Kiczales et Andreas Paepcke. Open implementations and metaobject protocols. `http://www.parc.xerox.com/oi-at-parc/ourpapers/tutorial.ps.gz`.

[KS88] A. Kumar et M. Stonebraker. Semantics based transaction management techniques for replicated data. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 17(3):117–125, septembre 1988.

[KTW92] Gregor Kiczales, Marvin Theimer, et Brent Welch. A new model of abstraction for operating system design. Dans *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*, pages 346–349, Dourdan, France, septembre 24–25 1992. IEEE Computer Society Press.

[KZ91] Peter Keleher et Willy Zwaenepoel. Detecting data races in software distributed shared memory. Technical Report TR91-171, Departement of Computer Science, Rice University, Houston, Texas, USA, novembre 1991.

[LA92] H. V. Leong et D. Agrawal. Type-specific coherence protocols for distributed shared memory. Sous la direction de IEEE, *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 434–441, Paris, France, 1992.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, juillet 1978.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, septembre 1979.

[Lam86] Leslie Lamport. On interprocess communication, Parts I and II. *Distributed Computing*, 1:76–101, 1986.

[Lam93] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. Technical Report 96, DEC SRC, février 1993.

[LCC+97] Barry M. Leiner, Vinton G. Cerf, David D. Clark, Robert E. Kahn, Leonard Kleinrock, Daniel C. Lynch, Jon Postel, Larry G. Roberts, et Stephen Wolff. A brief history of the internet. *Communications of the ACM*, février 1997. `http://www.isoc.org/internet-history/`.

[LCJS87] Barbara Liskov, Dorothy Curtis, Paul Johnson, et Robert Scheifler. Implementation of Argus. Dans *Proceedings of the 11th ACM Symposium on Operating Systems Principles* [ACM87], pages 111–122.

[LDS92] Barbara Liskov, Mark Day, et Liuba Shrira. Distributed object management in Thor. Dans *Proceedings of the International Workshop on Distributed Object Management*, pages 1–15, Edmonton (Canada), août 1992.

[LH89] Kai Li et Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, novembre 1989.

[Lis88] Barbara Liskov. Distributed programming in ARGUS. *Communications of the ACM*, 31(3):300–312, mars 1988.

[Lis91] Barbara Liskov. Practical uses of synchronized clocks in distributed systems. Dans *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, Canada, août 1991. ACM SIGOPS, ACM Press.

[Lit91] Mark C. Little. *Object Replication in a Distributed System.* PhD thesis, University of Newcastle upon Tyne, Computing Laboratory, septembre 1991.

[LLG92] Rivka Ladin, Barbara Liskov, et Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, novembre 1992.

[Mai96] Julien Maisonneuve. *Hobbes : un modèle de liaison de références réparties.* Thèse de doctorat, Université Paris 6, Pierre et Marie Curie, Paris (France), octobre 1996.

[Mat89] Friedmann Mattern. Virtual time and global states of distributed systems. Dans *Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.

[Mat93] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, août 1993.

[Mau90] Herlihy Maurice. Type specific replication algorithms for multiprocessor. Dans *Proceedings of theWorkshop on the Management of Replicated Data*, pages 70–74, Houston, TX (USA), novembre 1990. IEEE.

[Maz96] K. R. Mazouni. *Étude de l'invocation entre objets dupliqués dans un système réparti tolérant aux fautes*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1996. Number 1578.

[MB89] Keith Marzullo et Kenneth P. Birman. The role of order in distributed programs. Technical Report TR89-1001, School of Computer Science, Cornell University, mai 1989.

[McA95] Jeff McAffer. Meta-level programming with coda. Dans *ECOOP '95, Proceedings of the 9th European Conference on Object-Oriented Principles* [eco95], pages 190–241.

[MF93] Anurag Mendhekar et Daniel P. Friedman. Towards a theory of reflective programming languages. Dans *OOPSLA'93 Workshop on Reflection and Meta-level Architectures*, 1993.

[MGlNS94] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre le Narzul, et Marc Shapiro. Fragmented objects for distributed abstractions. Sous la direction de T. L. Casavant et M. Singhal, *Readings in Distributed Computing Systems*, pages 170–186. IEEE Computer Society Press, juillet 1994.

[Mil92] David L. Mills. Network Time Protocol (version 3) specification, implementation and analysis, mars 1992. En ligne à `http://www.eecis.udel.edu/~mills/database/rfc/rfc1305/rfc1305{a,b,c}.ps%`. Voir aussi `http://www.eecis.udel.edu/~mills/bib.html`.

[Mos93] David Mosberger. Memory consistency models. *Operating Systems Review*, 27(1):18–26, janvier 1993.

[MrB97] Mesaac Makpangou et Éric Bérenguier. Relais : un protocole de maintien de cohérence de caches web coopérants. Dans *NoTeRe'97 Colloquium*, Pau, France, novembre 1997. `http://www-sor.inria.fr/publi/RPMCCWC_notere97.html`.

[MRZ94] Masaaki Mizuno, Michel Raynal, et James Z. Zhou. Sequential consistency in distributed systems. Sous la direction de Kenneth P. Birman, Friedemann Mattern, et André Schiper, *Theory and Practice in Distributed Systems*, volume 938 des *Lecture Notes in Computer Science*, pages 224–241. Springer-Verlag, septembre 1994.

[MTY93] Satoshi Matsuoka, Kenjiro Taura, et Akinori Yonezawa. Highly efficient and encapsulated re-use of synchronisation code in concurrent object-oriented languages. Dans *OOPSLA '93, Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, ACM SIGPLAN Notices, pages 109–126. ACM SIGPLAN, ACM Press, 1993.

[MW84] R. J. T. Morris et W. S. Wong. Performance of concurrency control algorithms with nonexclusive access. Sous la direction de E. Gelenbé, *Performance'84*, pages 87–101. Elsevier Science Publisher, 1984.

[MW85] R. J. T. Morris et W. S. Wong. Performance analysis of locking and optimistic concurrency control algorithms. Dans *Performance Evaluation 5*, pages 105–118. Elsevier Science Publisher, 1985.

[Ng89] Tony P. Ng. Using histories to implement atomic objects. *ACM Transactions on Computer Systems*, 7(4):360–393, novembre 1989.

[NT93] Gil Neiger et Sam Toueg. Simulating synchonized clocks and common knowledge in distributed systems. *Journal of the ACM*, 40(3):334–367, avril 1993.

[Pae90] Andreas Paepke. PCLOS: Stress testing CLOS (experiencing the metaobject protocol). Dans *OOPSLA '90, Proceedings of the 5th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, ACM SIGPLAN Notices (25)10, pages 194–211, Ottawa, Canada, octobre 1990. ACM SIGPLAN, ACM Press.

[Pap79] C. H. Papadimitriou. Serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, octobre 1979.

[Pap86] Christos Papadimitriou. *The Theory of Database Concurrency Control.* Principles of Computer Sciences Series. Computer Science Press, 1986.

[PBS+88] D. Powell, G. Bonn, D. Seaton, P. Verissimo, et F. Waeselynck. The delta-4 approach to dependability in open distributed computing systems. Dans *International Symposium on Fault-Tolerant Computing (FTCS'88)*, pages 246–251, Washington, D.C., USA, juin 1988. IEEE Computer Society Press.

[PBS89] Larry L. Peterson, Nick Buchholz, et Richard D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, août 1989.

[PCM97] Elisabeth Pérez-Cortès et Jacques Mossière. La cohérence sur mesure dans une mémoire partagée répartie. *Technique et science informatiques*, 16(10):1283–1310, décembre 1997.

[PL90] Calton Pu et Avraham Leff. Epsilon-serialisibility. Technical Report CUCS-054-90, Columbia University, 1990.

[PL91] Calton Pu et Avraham Leff. Replica control in distributed systems: An asynchronous approach. Dans *Proceedings of the 1991 International conference on the Management of Data*, ACM SIGMOD Records, pages 377–386, Denver, CO, USA, mai 1991. ACM SIGMOD, ACM Press.

[Pri96] Ilya Prigogine. *La fin des certitudes.* Odile Jacob, 1996.

[PS88] Graham D. Parrington et Santosh K. Shrivastava. Implementing concurrency control in reliable distributed object-oriented systems. Dans *ECOOP '88, Proceedings of the 2nd European Conference on Object-Oriented Principles*, volume 322 des *Lecture Notes in Computer Science*, pages 233–249. Springer-Verlag, 1988.

[PST+97] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, et Alan J. Demers. Flexible update propagation for weakly consistent replication. Dans *Proceedings of the 16th ACM Symposium on Operating Systems Principles* [ACM97], pages 288–301.

[PSWL95] Graham Parrington, Santosh K. Shrivastava, M. Stuart Wheater, et Mark C. Little. The design and implementation of Arjuna. *Computing Systems*, 8(3):255–308, 1995.

[RAeA94] Rodolfo F. Resende, Divyakant Agrawal, et Amr el Abbadi. Semantic locking in object-oriented database systems. Dans *OOPSLA '94, Proceedings of the 9th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, ACM SIGPLAN Notices, pages 388–402, Portland, Oregon, USA, octobre 1994. ACM SIGPLAN, ACM Press.

[Rai] Riley Rainey. ACM: Air combat simulation. `http://pw1.netcom.com/~rrainey/acm.html`.

[RAK89] Umakishore Ramachandran, Mustaque Ahamad, et M. Yousef A. Khalidi. Coherence of distributed shared memory: Unifying synchronisation and data transfer. Dans *International Conference on Parallel Processing*, pages 160–169, 1989.

[RC94] Kriti Ramamritham et Panos K. Chrysanthis. *Distributed Object Management*, chapitre In Search of Acceptability Criteria: Database Consistency Requirements and Transaction Correctness Properties, pages 212–230. Morgan-Kaufmann, 1994.

[RHB94] Robbert Van Renesse, Takako M. Hickey, et Kenneth P. Birman. Design and performance of Horus: A lightweight group communications system. Technical Report TR94-1442, School of Computer Science, Cornell University, août 1994.

[RM93] Michel Raynal et Masaaki Mizuno. How to find his way in the jungle of consistency criteria for distributed shared memories (or how to escape from the Minos'labyrinth). Dans *Fundamentals Concepts*, volume 1 des *First Year Report.* BROADCAST, ESPRIT Basic Research Project 6360, octobre 1993.

[RP95] Krithi Ramamritham et Calton Pu. A formal characterization of epsilon serializability. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):997–1007, décembre 1995.

[RS95a] Michel Raynal et André Schiper. From causal consistency to sequential consistency in shared memory systems. Technical Report 926, Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes (France), mai 1995.

[RS95b] Michel Raynal et M. Singhal. Logical time: A way to capture causality in distributed systems. Technical Report 2472, Institut National de Recherche en Informatique et Automatique, mars 1995.

[Sec96] Special Section. Group communication. *Communications of the ACM*, 39(4):50–97, avril 1996.

[Sha94] Marc Shapiro. A binding protocol for distributed shared objects. Sous la direction de IEEE, *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 134–141, Poznan, Poland, juin 21–24 1994.

[Smi97] Jennifer Smith. Faq 1/3: Muds and mudding, décembre 1997. Posté tous les mois dans rec.games.mud, voir `http://www.apocalypse.org/pub/u/lpb/muddex/bartle.txt`.

[Sou93] Finnegan Southey. Where in the virtual world are we?, décembre 1993. `http://www.uoguelph.ca/~finnegan/thesis/thesis1.txt`.

[SRC84] J. H. Saltzer, D. P. Reed, et D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, novembre 1984.

[SS84] Peter M. Schwarz et Alfred Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–250, août 1984.

[SW95] R. J. Stroud et Z. Wu. Using metaobject protocol to implement atomic data types. Dans *ECOOP '95, Proceedings of the 9th European Conference on Object-Oriented Principles* [eco95], pages 168–189.

[SZ89] Andrea H. Skarra et Stanley B. Zdonik. *Object-Oriented Concepts, Databases, and Applications*, chapitre 16 – Concurrency Control and Object-Oriented Databases, pages 395–421. Frontier Series. ACM Press, 1989.

[TDP+94] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, et Brent B. Welch. Session guarantees for weakly consistent replicated data. Dans *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 140–149, Austin, Texas, septembre 1994.

[Ter87] Douglas B. Terry. Caching hints in distributed systems. *IEEE Transactions on Software Engineering*, 13(1):48–54, janvier 1987.

[TK97] Gérard Thia-Kime. *Critère de cohérence pour données partagées à support réparti.* PhD thesis, Université Rennes 1, octobre 1997.

[TTP+95] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, et Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. Dans *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Operating Systems Review (29)5, pages 172–183, Copper Mountain Resort, Colorado USA, décembre 3–6 1995. ACM SIGOPS, ACM Press.

[vN45] J. von Neumann. First draft of a report on the EDVAC. Technical report, University of Pennsylvania, 1945.

[vRBF+95] Robbert van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, et David A. Karr. A framework for protocol composition in Horus. Dans *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 80–89, Ottawa, Ontario, Canada, 2–23 août 1995.

[vRBM96] Robbert van Renesse, Kenneth P. Birman, et Silvano Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, avril 1996.

[Weg92] Peter Wegner. Dimensions of object-oriented modeling. *IEEE Computer*, 25(10):12–22, octobre 1992.

[Wei84] William E. Weihl. *Specification and implementation of atomic data types.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1984.

[Wei88] William E. Weihl. Commutativity-based concurrency control for abstract data type. *ACM Transactions on Computer Systems*, 37(12):1488–1505, décembre 1988.

[Wei89] William E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–282, avril 1989.

[Wol92] Mario Wolczko. Encapsulation, delegation and inheritance in object-oriented languages. *IEEE Transactions on Software Engineering*, 7(2):95–101, mars 1992.

[WYP97] Kun-Lung Wu, Philip S. Yu, et Calton Pu. Divergence control algorithms for epsilon serializability. *IEEE Transactions on Knowledge and Data Engineering*, 9(2):262–274, mars/avril 1997.

[Yok92] Yasuhiko Yokote. The Apertos reflective operating system: The concept and its implementation. Dans *OOPSLA '92, Proceedings of the 7th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, ACM SIGPLAN Notices (30)10, pages 414–434. ACM SIGPLAN, ACM Press, 1992.

[YTM91] Yasuhiko Yokote, Fumio Teraoka, et Atsushi Mitsuzawa. The Muse object architecture: A new operating system structuring concept. *Operating Systems Review*, 25(2):22–46, avril 1991.