Thèse de Doctorat de l'université
Pierre & Marie Curie – Paris 6

Spécialité :

Informatique

présentée par

Jean-Philippe Lesot

pour obtenir le titre de

Docteur de l'université Paris 6

Sujet de la thèse :

De l'amélioration des performances par la réplication des objets : une approche par estampillage

Soutenue le 11 mars 1999 devant le jury composé de :

Rapporteurs :	Michel	Banâtre	IRISA
	Jean-Marc	Geib	LIFL
Examinateurs :	François	Armand	Sun Microsystems
	Guy	Bernard	INT
	Claude	Girault	LIP6
Directeur de Recherche :	Jean-Marie	Rifflet	LIP6

Remerciements

Je tiens à remercier monsieur Jean-Marie RIFFLET, professeur à l'université Denis DIDEROT, pour m'avoir donné goût aux systèmes durant mes études, et avoir accepté d'être mon directeur de thèse durant ces trois dernières années.

Je remercie monsieur Jean-Marc GEIB, professeur à l'université de LILLE, pour avoir accepté d'être rapporteur de cette thèse, ainsi que monsieur Michel BANÂTRE, directeur de recherche à l'IRISA, avec qui j'ai eu grand plaisir à discuter, en regrettant certainement de ne pas l'avoir fait plus tôt.

Je remercie aussi monsieur François ARMAND, ingénieur/chef de projet chez Sun Microsystems, monsieur Guy BERNARD, professeur à l'Institut National des Télécommunications d'EVRY, ainsi que monsieur Claude GIRAULT, professeur à l'université Pierre et Marie CURIE, pour avoir accepté de participer à ce jury.

Stéphane ÉRANIAN, ingénieur chez Hewlett-Packard, et Pierre SENS, maître de conférences à l'université Pierre et Marie CURIE, m'ont apporté le sérieux de leurs commentaires et de leurs réflections, qu'ils en soient ici remerciés.

Je tiens à remercier vivement Pascal CHEVOCHOT, boursier de recherche à l'IRISA, pour les longues discutions, les conseils judicieux ainsi que les heures passées sur ce document, en regrettant peut être les bons moments passés ensemble avant cette thèse. J'espère bien lui être d'une quelconque aide à mon tour.

Il m'est très agréable de remercier Marc ZYNGIER, ingénieur de recherche et développement chez Bull, pour les relectures attentives et les corrections ingrates de mon anglais, mais surtout pour l'aide morale qu'il m'a apportée avec Franck ROGER, ingénieur d'études chez Kadra Consultants, durant les périodes de doute et de désespoir. Qu'ils soient sûrs que je leur en suis redevable.

Enfin, et surtout, je remercie très vivement Céline, de m'avoir fait confiance, peut être à tort, d'avoir supporté ma mauvaise humeur de temps en temps, et de m'avoir inconditionnellement soutenu durant toutes ces années. J'ose espérer que malgré la concision de ces quelques mots, elle les prendra à leur juste valeur.

Table des matières

Table des figures

Liste des tableaux

Introduction

L'AMÉLIORATION DES PERFORMANCES d'un service distribué vis-à-vis de ses utilisateurs est un critère important de satisfaction. La tendance actuelle est d'augmenter la puissance processeur des machines pour réduire le temps de réponse. Mais cette puissance n'est pas le seul intervenant dans la problématique. Les canaux de communication, qui acheminent les requêtes et leurs réponses, sont le point faible du système, introduisant des temps de propagation et de latence qui pénalisent le client. L'utilisation de portables connectés par intermittence ou avec des débits faibles et irréguliers ajoute des contraintes supplémentaires. De plus, contrairement à l'augmentation de la puissance processeur, l'augmentation des débits et la diminution de la latence des réseaux sont encore, de nos jours, relativement difficiles et coûteuses.

Notre travail a comme objectif principal d'améliorer les performances des serveurs dans un système distribué à objets. La solution proposée est d'utiliser la réplication des serveurs. Cela permet de multiplier les processeurs qui exécutent un service et d'augmenter la proportion d'accès locaux (sur une même machine, ou une même hiérarchie de réseaux) en rapprochant les copies d'un serveur des clients qui l'utilisent. Pour améliorer les performances, la réplication peut être accompagnée d'autres mécanismes :

– L'utilisation de la réplication active apporte la possibilité aux diverses copies de manipuler simultanément les mêmes données, améliorant ainsi le degré de concurrence dans l'exécution des requêtes ;
– L'emploi d'un schéma d'accès aux copies de type « *read-any / write-any* » permet à un client de changer de copie, entre autre, en fonction du temps de réponse. Ainsi, un client trouvant la copie choisie trop longue à répondre peut en changer immédiatement, ou à une prochaine requête ;
– Le contrôle du degré de synchronisation entre le client et les copies du serveur permet au client d'être synchrone avec les copies les plus rapides, et d'être asynchrone avec le reste. Ainsi, une copie surchargée, ou particulièrement lente, ne pénalise pas l'ensemble des clients ;
– L'utilisation de plusieurs copies simultanément pour exécuter la même requête permet à un client d'obtenir une réponse de la plus rapide d'entre elles. Cette méthode pénalise l'ensemble des clients, mais peut s'avérer utile pour des utilisations ou des clients particuliers ;
– Choisir le modèle de cohérence le plus adapté à l'application permet aux copies d'un serveur de maintenir leur état cohérent sans coût réseau et sans synchronisation inutile. À fortiori, l'utilisation de modèles de cohérence spécifiques à l'application, « de bout en bout », est la solution la plus efficace ;

- Permettre aux protocoles de cohérence et à ceux de synchronisation de partager les mêmes mécanismes autorise l'optimisation des messages réseau, et donc de diminuer leur coût ;
- L'adaptation aux réseaux des protocoles de communication permet de tirer partie de leur topologie et de leurs caractéristiques.

Néanmoins, la disponibilité, la fiabilité et le passage à l'échelle sont intimement liés aux systèmes distribués, et ne peuvent pas être ignorés. C'est pourquoi les solutions que nous proposons ne sont dépendantes ni du nombre de clients, ni du nombre de copies d'un serveur et fournissent seulement le support pour la tolérance aux fautes, sans la mettre directement en œuvre.

Propositions

Pour satisfaire à ces objectifs, ce document propose d'une part un cadre de travail, ou architecture partielle d'un système à objets, utilisant la réplication active, et d'autre part un protocole d'estampilles d'ordonnancement.

Le protocole des estampilles d'ordonnancement est un protocole qui permet à une requête de transporter avec elle toutes les informations nécessaires à sa livraison dans un ordre imposé par le modèle de cohérence. L'implantation du modèle de cohérence se résume alors à utiliser ce protocole pour déterminer un ordre sur les requêtes en les identifiant et les estampillant. Cela dissimule les problèmes de propagation, de stockage, de remise en ordre et de livraison des requêtes à l'application. L'utilisation du modèle de cohérence strictement nécessaire, voire l'utilisation d'un modèle spécifique « de bout en bout », est donc facilitée par l'utilisation de ce protocole.

Dans notre cadre de travail, un client peut utiliser les services d'un objet répliqué activement avec un schéma d'accès de type « *read-any / write-any* ». Il contrôle le degré de synchronisation avec les copies du serveur et l'utilisation de requêtes parallèles. Le protocole de cohérence est encapsulé dans un objet, facilitant l'implantation de différents modèles de cohérence, ou protocoles de synchronisation. Les protocoles de communication entre les copies d'un serveur sont contenus dans un autre objet, facilitant leur adaptation à l'environnement.

Enfin, dans cette étude, nous restreignons le schéma de relation entre les clients et les serveurs pour éviter la duplication des invocations, et donc la mise en place des filtrages. De tels mécanismes sortent du cadre de ce document, tout comme le placement des objets ou la répartition de la charge qui peuvent être utilisés conjointement pour améliorer les performances.

Plan du document

Le chapitre 1 introduit quelques définitions d'un système distribué, de ses modèles possibles, et des modèles de programmation. Ainsi, nous décrivons le passage de messages, l'appel de procédures à distance, l'invocation de méthodes à distance et la mémoire virtuellement partagée. Au travers de ces descriptions, nous introduisons le vocabulaire nécessaire à la compréhension de ce document, comme l'ordre, la fiabilité et l'hétérogénéité.

Nous traitons alors au chapitre 2 de la réplication, en introduisant les principales méthodes, leurs coûts, leur répercution sur la tolérance aux fautes, mais surtout en quoi ils peuvent améliorer les performances des applications. Nous décrivons alors les principaux modèles de cohérence qui permettent à un serveur répliqué de maintenir cohérents les différents états introduits par ses différentes copies. Enfin, nous terminons ce chapitre en examinant quelques systèmes représentatifs d'une technique de réplication particulière, mais aussi d'une méthode d'intégration de cette réplication dans un cadre plus large.

Dans le chapitre 3, nous introduisons pas à pas notre protocole d'estampilles d'ordonnancement. Ce protocole permet à une requête de transporter avec elle toutes les informations nécessaires pour déterminer l'ordre de sa livraison au serveur destinataire. Nous commençons par introduire les bases du protocole en décrivant comment il peut représenter une relation d'ordre strict. Nous exposons alors des optimisations indispensables à l'efficacité de ce protocole, en autorisant certaines requêtes à ne pas être diffusées, et en introduisant des mécanismes pour gérer des motifs d'utilisation fort récurrents. À chaque étape, nous présentons des exemples d'utilisation pour faciliter la compréhension des différents mécanismes intervenant dans ce protocole.

Le chapitre 4 décrit un cadre de travail pour la réplication active en vue d'améliorer les performances s'appuyant sur l'utilisation du protocole des estampilles d'ordonnancement. Ainsi, il permet de clarifier l'utilisation de ce protocole, et de comprendre ce qu'il apporte à une telle architecture. Chaque gestionnaire ou objet introduit est le plus indépendant possible, permettant à chacun de tirer partie de son environnement, et d'être facilement interchangeable par rapport aux autres.

Enfin, l'implantation d'un prototype de ce cadre de travail est décrit dans le chapitre 5. Il nous permet de revenir sur le protocole des estampilles d'ordonnancement pour le compléter avec un mécanisme supplémentaire indispensable à son implantation. La description de l'implantation de deux gestionnaires particuliers et d'une librairie nous permet de montrer le coût limité du protocole des estampilles d'ordonnancement, et sa facilité d'utilisation pour implanter des modèles de cohérence et un mécanisme de synchronisation.

Chapitre 1

Communication dans les systèmes distribués

L'ESSOR DES SYSTÈMES DISTRIBUÉS est essentiellement dû à l'extension des réseaux, à leur modularité intrinsèque, à leur bon rapport coût/performance ainsi qu'à leur possibilité de partager des ressources.

Autant le matériel évolue rapidement (processeurs de plus en plus puissants, réseaux de plus en plus rapides, etc), autant une application distribuée reste encore délicate à programmer et à mettre au point, qui plus est si elle doit tolérer les fautes. Les couches logicielles qui cachent la distribution (communication de groupes, courtier d'objets, etc) se heurtent à des problèmes conséquents de pertes de performance. Des méthodes, des outils, des optimisations ont été proposés, fournissant une multitude de solutions, difficilement portables et rarement interchangeables. Le programmeur d'une application distribuée est donc face à un choix important qui influence les caractéristiques et les performances de son travail.

Ce chapitre présente différents moyens de communication dans les systèmes distribués. Après la définition d'un système distribué (§ 1.1), nous examinons leurs motivations (§ 1.2) et les différents modèles utilisés (§ 1.3). Enfin, nous traitons des modèles de programmation (§ 1.4) avec le passage de messages (§ 1.4.1), l'appel de procédures à distance (§ 1.4.2), l'invocation de méthodes à distance (§ 1.4.3) et enfin la mémoire virtuellement partagée (§ 1.4.4).

1.1 Définition

De multiple définitions d'un système distribué existent. Certaines, comme dans [Tanenbaum, 1995], définissent un système distribué comme un système qui s'exécute sur un ensemble de machines sans mémoire partagée, mais vu comme une seule et unique machine par les utilisateurs. Pour notre part, nous le définissons ainsi :

Définition 1.1
Un système distribué est un ensemble de machines qui communiquent entre elles par échange de messages au travers d'un canal de communication. Elles possèdent leur propre mémoire, éventuellement des périphériques, mais ne possèdent ni mémoire partagée, ni horloge globale.

1.2 Motivations

Durant ces dernières années, les systèmes distribués ont été largement étudiés, conçus et utilisés. Par rapport à un système centralisé, les systèmes distribués ont les atouts suivants :

1.2.1 Partage de ressources

Grâce au réseau, une machine peut mettre à disposition des autres ses ressources matérielles ou logicielles. Ainsi des ressources coûteuses peuvent être utilisées par un ensemble de machines.

1.2.2 Amélioration des performances

Dû au nombre important de processeurs dans un système distribué, celui-ci est capable de fournir un temps de réponse faible, et une capacité importante de traitement. Tant des critères économiques que des critères physiques tendent à favoriser l'utilisation de plusieurs processeurs plutôt que d'un seul et unique cumulant la puissances de chacun d'entre eux.

1.2.3 Disponibilité et fiabilité

Dans l'exécution d'un service fourni par un système distribué, le nombre d'intervenants (au sens matériel et logiciel) augmentant, la probabilité d'être tributaire d'un élément défaillant augmente. Néanmoins, la redondance des processeurs et des canaux de communication peut être utilisée pour traiter ces fautes, et ainsi obtenir un service d'une fiabilité supérieure à celle des systèmes centralisés.

1.2.4 Extensibilité

Un système distribué peut être facilement étendu en ajoutant de nouvelles ressources matérielles ou logicielles, sans remplacer ou modifier les ressources existantes. Ce critère est important pour évoluer facilement en terme de puissance de calcul, de capacité de stockage ou d'intégration de nouvelles technologies.

1.3 Modèles du système

Un système distribué peut être représenté par plusieurs modèles. Dans cette section, nous définissons le modèle synchrone (§ 1.3.1), puis le modèle asynchrone (§ 1.3.2) et enfin le modèle partiellement synchrone (§ 1.3.3).

1.3.1 Synchrone

Un système distribué synchrone est un système dans lequel des suppositions fortes sur les bornes temporelles des calculs, des canaux de communication et de la dérive des horloges sont faites. Ainsi, le délai entre l'envoi et la réception d'un message est borné, et est connu de tous les sites.

1.3.2 Asynchrone

Un système distribué asynchrone est, par opposition à un système synchrone, un système dans lequel aucune hypothèse n'est faite sur les bornes temporelles des calculs et des canaux de communication, ni sur la dérive des horloges. Concrètement, on ne peut pas effectuer de supposition sur le temps que mettra un processeur pour faire un calcul, ou sur le temps que mettra un canal de communication pour acheminer un message à destination.

Malheureusement, Fisher, Lynch et Paterson ont prouvé dans [Fisher et al., 1985] que le consensus est impossible à atteindre dans un système asynchrone soumis à des défaillances (même unique). L'idée intuitive est qu'il est impossible de distinguer un processus ou un canal de communication défaillant, d'un processus ou d'un canal de communication lent.

1.3.3 Partiellement synchrone

Pour contourner ce résultat d'impossibilité, plusieurs modèles partiellement synchrones ont été étudiés en relâchant certaines contraintes du modèle synchrone (par exemple existence d'une borne temporelle sur les messages mais de valeur inconnue). Ces différents modèles ont été classés en fonction de la possibilité d'y résoudre le consensus.

1.4 Modèles de programmation

Dans un système centralisé, les différents processus communiquent par tube (*pipe*) ou par des systèmes de communication inter-processus (*Inter Process Communication*, ipc). Par exemple, sous Unix il existe principalement les queues de messages, les sémaphores et la mémoire partagée [Rifflet, 1993]. Dans un système distribué, l'absence de mémoire commune oblige les processus à communiquer par messages au travers de canaux de communication. Néanmoins, il existe différents modèles de programmation d'un système distribué que nous décrivons dans cette section, à savoir le passage de messages (§ 1.4.1), l'appel de procédures à distance (§ 1.4.2), l'invocation de méthodes à distance (§ 1.4.3) et la mémoire virtuellement partagée (§ 1.4.4).

1.4.1 Passage de messages

Le passage de message est le plus simple des modèles, dans le sens où il n'est fondé que sur deux primitives *envoyer* (*send*) et *recevoir* (*receive*). Ainsi, un processus peut envoyer un message à un autre processus ou à un groupe de processus, ou peut recevoir un message. Un message est constitué de données ou d'instructions. Le passage de messages peut être utilisé pour réaliser une synchronisation.

1.4.1.1 De un vers un

Tout d'abord, intéressons nous au message « de un vers un » (ou point à point). Dans ce cas, le destinataire du message est un seul processus. L'interface la plus répandue pour programmer cette communication est sûrement les *sockets* [Rifflet, 1995] initialement conçue sous Unix.

a) Synchrone *vs* asynchrone

Un passage de message est synchrone[1] quand le processus est bloqué sur la primitive *send* tant que le destinataire n'a pas récupéré le message par un *receive*, et inversement, un processus est bloqué sur un *receive* tant qu'un émetteur n'a pas envoyé un message par un *send*. Tous les autres cas sont des communications asynchrones.

Dans le mode asynchrone, la conception et la mise au point des applications est plus délicate, car elle fait intervenir des évènements asynchrones (notification de messages, déroutement de l'exécution du programme, etc). Néanmoins les processus peuvent continuer leurs exécutions sans blocage, augmentant le degré du parallélisme des exécutions. Cette méthode est donc globalement plus performante que l'utilisation d'un mode synchrone.

Dans le mode synchrone, l'utilisation d'activité (*thread*) est utilisée pour obtenir les même résultat, mais des synchronisations complexes inter-activités sont alors nécessaires.

b) Fiabilité

La fiabilité d'un passage de message est un critère important pour le concepteur d'applications. Dans le cas non fiable, le système ne fournit aucune garantie sur le bon acheminement du message. Si la fiabilité est indispensable, c'est à la charge de l'utilisateur de gérer les pertes de messages. Nous pouvons citer comme exemple de ce type les *sockets* avec le protocole UDP (*User Datagram Protocol*) [Postel, 1980], ou bien les primitives *ipcSend* et *ipcReceive* du μ-noyau Chorus [Rozier et al., 1988].

Dans le cas fiable, le système garantit la livraison du message. Dans le cas contraire, il signal une erreur. Cette solution est assez simplement mise en œuvre. Des protocoles comme RDP (*Reliable Datagram Protocol*) [Partridge et Hinden, 1990] ont été définis à cet effet.

c) Ordre

Pour les communications point à point, il existe plusieurs types d'ordre de livraison des messages. Le plus trivial est l'absence d'ordre, puis l'ordre FIFO (*First In, First Out*) et enfin l'ordre causal. Sans ordre, un message m_1 émis avant un message m_2 par un processus P_1 peut être reçu par un processus P_2 après le message m_2.

L'ordre FIFO conserve l'ordre d'émission depuis un même émetteur, puisque le premier message envoyé est le premier reçu. Néanmoins, l'ordre n'étant pas transitif, il n'y a aucune garantie sur l'ordre des messages transitant par un troisième processus. L'ordre est seulement garantit entre chaque paire d'émetteurs et de récepteurs.

L'ordre causal permet d'ordonner la livraison des messages en fonction d'une relation de causalité. Informellement, le principe de la causalité stipule que si un évènement E' est causé ou influencé par un évènement E, alors tout le système doit observer l'évènement E avant l'évènement E'. Plus formellement, la définition dit qu'un évènement E précède causalement un évènement E' s'il respecte une des trois propriétés suivantes [Lamport, 1978] :

- E et E' se déroulent sur le même processus et E précède E' dans l'ordre programme ;
- E est l'envoi d'un message par un processus et E' est la réception de ce message par un autre processus ;
- Il existe un évènement E'' tel que E précède E'' et E'' précède E' (transitivité).

1. À ne pas confondre avec un système distribué synchrone ou asynchrone (§ 1.3 page 6).

L'ordre causal ajoute une condition supplémentaire par rapport à la précédence causale. Si m_1 et m_2 sont deux messages ayant le même destinataire, alors :

– Si l'émission d'un message m_1 précède l'émission d'un message m_2 alors la réception du message m_1 précède la réception du message m_2.

La figure 1.1 page suivante résume ces différents ordres en fournissant des exemples de scénarii possibles entre plusieurs processus.

1.4.1.2 De un vers plusieurs

Intéressons nous maintenant aux communications « de un vers plusieurs », plus communément appelées les communications de groupe. Un processus peut alors envoyer un message à un groupe constitué de n destinataires, les membres. Un certain nombre de critères caractérisent une communication de groupe.

a) Fiabilité

La fiabilité est un critère décrivant la façon dont le système réagit en cas de défaillance d'un canal de communication ou d'un processeur (tampons pleins, perte de messages, etc). Une description informelle d'une diffusion fiable est la suivante : les destinataires corrects reçoivent le même ensemble de messages (propriété d'*accord*), cet ensemble de messages contient tous les messages provenant d'émetteurs corrects (propriété de *validité*) et enfin il n'existe pas dans cet ensemble des messages non émis (propriété d'*intégrité*). Des variantes *uniformes* des propriétés d'accord et d'intégrité définissent le comportement des processus quand certains messages sont délivrés à des processus fautifs. Pour plus de détails, nous renvoyons le lecteur intéressé à [Hadzilacos et Toueg, 1993].

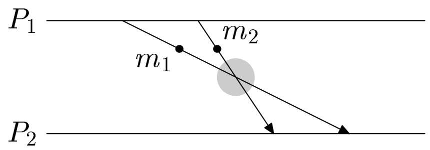
La fiabilité étant relativement difficile à réaliser, beaucoup de systèmes fournissent des primitives des communications de groupe non fiables comme le système V [Cheriton et Zwaenepoel, 1985] ou le μ-noyau Chorus [Rozier et al., 1988]. Dans ce cas, aucune hypothèse sur la livraison du message à tous les destinataires, ni sur la remonté d'erreur à l'émetteur n'est faite. Par contre, des systèmes comme Amoeba [Kaashoek et Tanenbaum, 1994] ou Isis [Birman et Joseph, 1987] fournissent des primitives fiables.

Certaines applications spécifiques considèrent qu'une communication fiable a un coût trop élevé à supporter pour ce qu'elles en attendent. Malgré cela, la majorité des applications requièrent une fiabilité de leurs communications, et les primitives non fiables offertes par certains systèmes sont utilisées principalement par des librairies de communication fournissant des sémantiques plus fortes.

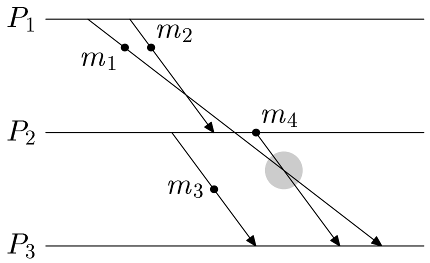
b) Ordre

La communication de groupe ajoute une dimension à l'ordre de livraison des messages. Il s'agit de l'ordre de livraison des messages sur un membre du groupe par rapport à l'ordre de livraison sur chaque membre.

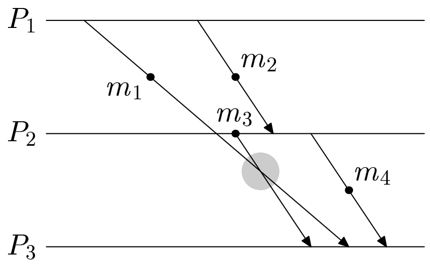
Ainsi, l'ordre total est défini comme un ordre où tous les destinataires reçoivent les messages dans le même ordre. Un exemple de diffusion respectant, ou non, l'ordre total est fourni figure 1.2 page 11.



(a) Sans ordre



(b) Ordre FIFO



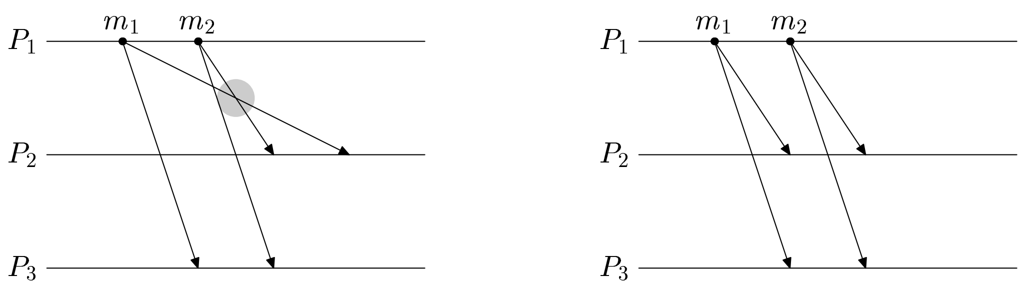
(c) Ordre causal

Sans ordre : l'ordre de livraison des messages m_1 et m_2 se fait sans aucune contrainte ;

Ordre FIFO : la livraison des messages m_3 et m_4 se fait obligatoirement dans cet ordre, car provenant du même processus P_2 à destination du même processus P_3. Par contre, les messages m_1 et m_4 n'ont aucune contrainte (voir le repère grisé) car en provenance de deux processus différents ;

Ordre causal : les messages m_1 et m_4 sont liés causalement, car la réception de m_1 précède causalement son émission, l'émission de $m1$ précède causalement l'émission de m_2 (ordre programme), la réception de m_2 précède causalement l'émission de m_4 (ordre programme), l'émission de m_4 précède causalement sa réception, et donc par transitivité la réception de m_1 précède causalement celle de $m4$. Par contre, aucune contrainte entre la réception de $m1$ et $m3$ n'existe (voir repère grisé) car causalement indépendant.

FIG. 1.1 – Ordre de livraison des messages « de un vers un »



(a) Sans ordre total

(b) Ordre total

Une diffusion ne respectant pas un ordre total autorise les destinataires à recevoir les messages dans un ordre différent (voir repère grisé).
Par contre, une diffusion respectant un ordre total, garantit que tous les destinataires observent le même ordre de livraison des messages.

FIG. 1.2 – Ordre de livraison des messages de « un vers plusieurs »

C'est une dimension supplémentaire, car un ordre total ne garantit pas forcément l'ordre FIFO ou l'ordre causal. C'est pour cela qu'il existe des diffusions FIFO totales et des diffusions causales totales. Pour plus de détails, nous renvoyons de nouveau le lecteur à [Hadzilacos et Toueg, 1993].

Par exemple, la boite à outils Isis [Birman et van Renesse, 1994] fournit trois primitives de diffusion. L'ABCAST (*Atomic BroadCAST*) fournit une diffusion qui respecte un ordre total. Le CBCAST (*Causal BroadCAST*) respecte un ordre causal dans la livraison des messages. La diffusion ABCAST et le CBCAST sont indépendantes, c'est-à-dire que l'ordre des messages diffusés entre ces deux protocoles n'est pas garanti. Par contre, le GBCAST (*Global BroadCAST*) assure un ordre total global, c'est-à-dire que si x est un GBCAST et y une diffusion de n'importe quel autre type, alors x et y seront délivrés dans le même ordre sur l'intersection de tous les destinataires de x et de tous les destinataires de y.

c) Livraison

La sémantique de livraison détermine quand la diffusion d'un message sera considérée comme réussie pour l'émetteur. Il en existe principalement trois types. Le premier, appelé *k-delivery*, définit une diffusion réussie quand k destinataires l'ont reçue. Le deuxième, basé sur des quorums, la considère réussie quand une majorité des membres du groupe l'ont reçue. Enfin, la dernière considère la diffusion réussie quand tous les membres non fautifs ont reçu le message. Dans le cas contraire, aucun d'entre eux ne doit l'avoir reçu. Elle concerne donc une diffusion fiable. Une diffusion fiable respectant un ordre total est appelée une diffusion atomique (par exemple la diffusion ABCAST d'Isis).

1.4.1.3 Conclusion

Bien que le programmeur d'applications par passage de messages puisse faire exactement ce qu'il souhaite, la multiplicité des solutions, des sémantiques et des implantations ne facilite pas le développement d'applications distribuées sur plusieurs types de machines et/ou de systèmes. Des outils de communication comme PVM (*Parallel Virtual Machine*) [Geist et al., 1994], Isis [Birman et Joseph, 1987] ou bien son successeur Horus [van Renesse et al., 1995], largement diffusés et utilisés dans différents environnements, permettent de résoudre ce type de problème.

1.4.2 Appel de procédures à distance

Un des buts des concepteurs de systèmes distribués est de fournir une manière de programmer proche de celle d'un système centralisé. Pour cela, dans [Birrell et Nelson, 1984], les auteurs introduisent le concept élégant d'appel de procédures à distance (*Remote Procedure Call*, RPC). Le but est de cacher à l'utilisateur le fait qu'une procédure s'exécute sur une machine distante, et donc de lui fournir la même interface qu'une procédure appartenant au même processus.

1.4.2.1 Passage de paramètres

L'utilisateur appelle une procédure comme dans un système centralisé. Celle-ci est constitué d'un talon, le plus souvent généré automatiquement à partir du prototype de la procédure. Ce talon emballe (*marshaling*) le nom de la procédure et ses paramètres dans un message. Ensuite, il détermine la machine sur laquelle il va exécuter la procédure. Enfin, il envoie ce message par des mécanismes identiques au passage de messages décrit dans la section précédente (§ 1.4.1.1 page 7).

L'appelé reçoit le message, déballe (*unmarshaling*) les paramètres de la procédure, et d'après le nom de la procédure, appelle celle-ci. Un fois la procédure finie, les paramètres de retours subissent le même traitement dans le sens inverse. L'appelant reçoit alors les réponses, et continue son exécution.

Le point délicat est le traitement du passage de paramètres par référence. On peut tout simplement interdire le passage de paramètres par référence, mais cela introduit une restriction supplémentaire par rapport à une programmation centralisée. Sinon on peut recopier la valeur de la structure pointée. Il peut être nécessaire de la recopier dans l'autre sens parce que la valeur est susceptible d'avoir été modifiée. Certaines optimisations permettent de lever l'ambiguïté sur le mot « susceptible » en demandant au programmeur de spécifier si son pointeur sert à passer un paramètre, à en recevoir un, ou bien à le passer puis à le recevoir modifié.

1.4.2.2 Hétérogénéité

L'hétérogénéité concerne la représentation des paramètres de la procédure. En effet, la représentation peut être différent sur la machine appelante et la machine appelée. Par exemple, un entier de 32 bits ne sera pas codé de la même façon sur une machine petit-boutiste (*little-endian*) et gros-boutiste (*big-endian*). Une solution simple est que la machine appelée convertisse le format s'il est différent du sien. Mais cela requière une mise à jour de chaque machine, quand une machine avec un nouveau format rejoint le système.

Une autre solution est de fournir une représentation commune. La conversion est alors faite à chaque fois aussi bien sur la machine appelante que sur la machine appelée. Mais cette méthode gaspille des ressources si les deux machines ont la même représentation. Il existe par exemple la norme XDR (*eXternal Data Representation*) [Sun Microsystems, 1987], décrivant la représentation des types de base et des types composés.

Il est à noter que ce problème d'hétérogénéité se pose aussi pour le passage de messages. Mais dans la plupart des cas, seule l'application peut connaître le type du message, et donc l'emballer correctement. Ici, le talon connaît le type des paramètres de la procédure, et peut donc emballer les paramètres sans que l'application s'en préoccupe.

1.4.2.3 Sémantique

Le point délicat de la méthode par appel de procédures à distance se situe en cas de défaillance. En effet, un RPC peut échouer pour deux raisons principales. La première est la défaillance d'une machine, la seconde est la défaillance du réseau. En cas de défaillance, l'appelant peut réémettre son appel. Or, nous avons vu que dans un système distribué asynchrone, il n'était pas possible de différencier une machine lente d'une défaillance. Si l'appelant réémet son appel en pensant à une défaillance, alors que la machine est seulement lente, la procédure peut être exécutée deux fois. Or pour les procédures non idempotentes[2], ceci est un comportement inadapté.

C'est pour cela que l'on classe un appel de procédures à distance parmi les trois sémantiques suivantes [Nelson, 1981] :

a) Au moins une fois (*at least once*)

Si la procédure a été exécutée correctement vis-à-vis du client, alors elle a été exécutée au moins une fois sur la machine distante. Sinon elle a été exécutée zéro, une ou plusieurs fois, voire même partiellement exécutée.

b) Au plus une fois (*at most once*)

Si la procédure a été exécutée correctement vis-à-vis du client, alors elle a été exécutée exactement une fois sur la machine distante. Sinon elle a été exécutée zéro, une fois ou partiellement.

c) Exactement une fois (*exactly once*)

Si la procédure a été exécutée correctement vis-à-vis du client, alors elle a été exécutée exactement une fois sur la machine distante. Sinon, elle n'a pas été exécutée une seule fois.

1.4.2.4 Synchrone *vs* asynchrone

Des implantations de RPC fournissent des appels asynchrones, c'est-à-dire que l'appelant n'est pas bloqué après son appel. Il lui est même possible de récupérer les paramètres de retour dans un ordre différent de celui des appels [Anada et al., 1991]. Ces méthodes ont les

2. Nous rappelons qu'une procédure idempotente est une procédure qui peut être exécutée plusieurs fois avec le même résultat qu'une exécution unique.

même inconvénients et les même avantages que pour le passage de messages synchrones ou asynchrones (§ 1.4.1.1.a page 8).

1.4.2.5 Appel de procédures parallèles à distance

Tout comme il existe des communications de groupe, il existe des appels de procédures à distance qui provoquent n exécutions sur différentes machines. L'appelant est alors bloqué jusqu'à ce que les n réponses lui parviennent (comme dans MultiRPC [Satyanarayanan et Siegel, 1990]), ou avec une possibilité d'exécuter du code entre chaque réponse (comme dans PARPC [Martin et al., 1989]).

1.4.2.6 Conclusion

L'appel de procédures à distance est intéressant d'un point de vue programmation. Il offre une similitude par rapport à la programmation centralisée, et est donc bien maîtrisé par le programmeur. C'est essentiellement pour cette raison qu'il est fréquemment employé. Malheureusement, la sémantique en cas de défaillance pose problème. Les implantations gèrent généralement cela en armant des temporisateurs et en réémettant l'appel. Les partitions réseaux (défaillance partageant les machines en groupes telle que seules les machines d'un même groupe peuvent communiquer entre elles) ne sont pas distinguées des défaillances de machines ou de processus, et cette détection se fait indépendamment pour chaque appel [Birman et Glade, 1995]. Il est alors difficile de programmer des applications dans un tel contexte, même avec une sémantique forte de type « au plus une fois ».

1.4.3 Invocation de méthodes à distance

L'invocation de méthodes à distance (*Remote Method Invocation*, RMI) est en fait un appel de procédures à distance (RPC) dans un contexte objet.

La programmation orientée objet, utilise comme modèle d'abstraction de données (les *types*). Un type définit une interface fonctionnelle. Une *classe* spécifie une mise en œuvre particulière d'un type. Un *objet* est une instance d'une classe. Un objet est constitué d'un ensemble de données, son *état*, et d'un ensemble de procédures, ses *méthodes*. Les données ne sont utilisables de l'extérieur que par l'intermédiaire des méthodes. Elles caractérisent le comportement de la classe. L'invocation d'une méthode représente l'appel de la procédure. À un instant donné, le *client* est l'objet qui invoque, et le *serveur* celui qui est invoqué. Un objet peut donc être alternativement client et serveur.

Par rapport aux appels de procédures à distance, les mêmes caractéristiques, remarques, avantages et inconvénients s'appliquent aux invocations de méthodes à distance. Néanmoins, pour introduire un minimum de vocabulaire, et quelques notions sur les courtiers d'objets (*Object Request Brocker*, ORB) nous décrivons succinctement CORBA. La normalisation exhaustive de CORBA (*Common Object Request Brocker Architecture*) par l'OMG (*Object Management Group*) est décrite dans [OMG, 1996]. De nombreuses introductions existent, comme [Vinoski, 1997]. Cette architecture est constituée de huit parties distinctes :

1. L'ORB *core*, qui doit cacher au client la localisation de l'objet, son implantation (langage de programmation, script, etc), son état d'exécution (si l'objet est en train de s'exécuter ou non) et enfin les mécanismes de communication (appel de méthode locale, mémoire virtuellement partagée, TCP / IP, etc) ;

2. L'IDL (*Interface Definition Language*), qui est un langage pour décrire les interfaces des objets. Il fournit l'indépendance au langage de programmation. Il spécifie chaque interface en décrivant pour chacune de ses méthodes, ses paramètres — types élémentaires (`boolean`, `char`, `float`, etc), types composés (`struct`, `union`, etc), patron de types (`string`, `sequence`, etc) —, la « direction » du paramètre — en entrée (`in`), en sortie (`out`), dans les deux sens (`inout`) — et les exceptions possibles. Il supporte l'héritage d'interface, la notion de type universel `any`, les références d'objets, etc ;
3. La projection de langage, définit la correspondance entre l'IDL et un langage cible choisi pour implanter l'objet. Des correspondances pour C, C++, Smalltalk et Ada95 ont été normalisées. D'autres sont en cours de normalisation comme le Cobol ou le Bourne *shell*. Par exemple, il définit qu'un type `octet` de l'IDL correspond à un type `unsigned char` du C++ ;
4. Le dépositaire d'interface, fournit une base de donnée qui contient les interfaces des objets. La plupart des applications ont un besoin statique de ces informations, elles n'utilisent donc pas la base. Par contre, d'autres applications peuvent y trouver à l'exécution l'interface d'un objet, ou y enregistrer leurs propres interfaces dynamiquement. Cette base est elle-même un objet CORBA décrite en IDL ;
5. Les talons (*stubs*) et squelettes (*skeletons*), sont des objets générés à partir de l'IDL. Le talon s'occupe de gérer la requête du client, et le squelette de la délivrer réellement au serveur. Ce mécanisme est appelé invocation statique, car il est intégré dans le client et le serveur, et requière la connaissance de l'IDL du serveur à la compilation ;
6. L'invocation dynamique, constituée de l'interface d'invocation dynamique (*Dynamic Invocation Interface*, DII) et de l'interface de squelette dynamique (*Dynamic Skeleton Interface*, DSI), est par contre l'aspect dynamique du point précédent. Elle permet à un client d'invoquer un objet dont il ne connaissait pas l'IDL lors de sa compilation. De plus, elle permet une plus grande souplesse car l'invocation peut être émise d'une manière synchrone, asynchrone déférée (les réponses sont collectées plus tard) et asynchrone total (*oneway*). Des variantes avec plusieurs destinataires existent aussi ;
7. L'adaptateur d'objets (*Object Adapters*), est un objet qui sert d'interface entre l'implantation de l'objet et l'ORB lui-même. Il doit fournir des services d'enregistrement, de génération de référence, d'activation de processus, d'activation d'objet, de démultiplexage de requête, etc ;
8. Le protocole Inter-ORB (IIOP), est un protocole facilitant l'interopérabilité d'ORB issus de fabricants différents. C'est en fait une spécification de GIOP (*General Inter-*ORB *Protocol*) au dessus de TCP / IP. Une référence à un objet est aussi normalisée et représentée par un IOR (*Interoperable Object Reference*).

1.4.4 Mémoire virtuellement partagée

Une autre méthode pour fournir une façon de programmer proche de celle d'un système centralisée est de fournir une mémoire virtuellement partagée (*Distributed Shared Memory*, DSM). Bien que n'ayant pas de mémoire commune, chaque machine partage une portion mémoire. Un protocole se charge alors de simuler une mémoire partagée en s'échangeant des pages ou des modifications. L'avantage de cette solution est une programmation relativement naturelle. De nombreux systèmes fournissent une mémoire virtuellement partagée, comme par exemple Ivy [Li et Hudak, 1989], Midway [Bershad et Zekauskas, 1991], Munin [Carte, 1995], Chorus [Ortega, 1993], TreadMarks [Amza et al., 1996].

1.4.4.1 Structure et granularité

La structure et la granularité du système de partage de mémoire est un choix important car il conditionne le rapport entre le surcoût et la performance.

La structuration en pages est un des mécanismes utilisé. Il faut alors choisir la granularité de la page. Plus la page est grosse, moins il y aura de surcoût dû à la pagination, mais plus de problèmes de faux partages existeront. Ce problème survient quand deux applications accédant à la même page mémoire, mais sur des zones différentes, rentrent en compétition pour l'acquérir, alors qu'il n'y a aucune concurrence réelle dans les accès. À l'inverse, plus la page est petite, plus les structures d'annuaires sont lourdes. Ainsi, les implantations logicielles de la mémoire distribuée font varier la taille des pages entre 1 Ko et 8 Ko. Les implantations matérielles permettent d'avoir des granularités de l'ordre 16 ou 32 octets.

La structuration par type est une autre méthode. On ne partage plus de la mémoire non typée mais des variables et des structures. Cette méthode est obligatoirement implantée de façon logicielle. C'est la technique adoptée par Munin [Carter et al., 1991] et Midway [Bershad et Zekauskas, 1991].

Enfin, on peut structurer la mémoire par objet. La granularité de la mémoire s'adapte alors à l'application. Les variables ne sont accédées que par les méthodes de cet objet pour améliorer les performances de la mémoire virtuellement partagée. C'est la cas de Linda [Carriero et Gelernter, 1986] et Orca [Bal et al., 1992].

Malheureusement, ces deux dernières méthodes ne résolvent pas les problèmes de faux partages qui peuvent toujours survenir à l'intérieur d'un objet.

1.4.4.2 Cohérence

Pour des questions de performance, les concepteurs ont autorisé qu'une page mémoire puisse être présente physiquement sur plusieurs machines. Il y a alors un mécanisme pour maintenir la cohérence des différentes copies de cette page. Nous décrivons rapidement quelques cohérences, des plus fortes aux plus faibles. Nous reviendrons plus en détail sur ce point au § 2.3 page 25.

Cohérence atomique (*Atomic consistency*) : Une opération de lecture retourne la valeur de l'écriture la plus récente ;

Cohérence séquentielle (*Sequential consistency*) : Tous les accès à la mémoire sont perçus dans le même ordre pour tous les processus où qu'ils se trouvent ;

Cohérence causale (*Causal consistency*) : Seules les opérations d'écriture liées causalement sont perçues dans le même ordre par tous les processus ;

Cohérence PRAM (PRAM *consistency*) : Les opérations d'écriture d'un même processus sont perçues dans le même ordre par tous les autres processus ;

Cohérence faible (*Weak consistency*) : Le programmeur contrôle la cohérence au travers de variables de synchronisation ;

Cohérence au relâchement (*Release consistency*) : Au travers d'opérations de synchronisation — acquérir (*acquire*) et relâcher (*release*) — le programmeur contrôle encore plus finement la cohérence ;

Cohérence à l'entrée (*Entry consistency*) : La donnée partagée est seulement consistante à l'acquisition de la variable de synchronisation qui lui est associée.

1.4.4.3 Hétérogénéité

Le problème de l'hétérogénéité de la représentation des données dans une mémoire virtuellement partagée est essentiellement identique à celui recontré dans le contexte des RPC. Néanmoins, le système de mémoire ne connaît pas le type des données indispensable pour les convertir.

Dans les systèmes de gestion de mémoire virtuellement partagée utilisant les mécanismes de la mémoire virtuelle, il existe, en plus, une hétérogénéité au niveau de la taille physique des pages, obligeant dans la plupart des cas à utiliser la plus importante.

Le système Mermaid [Zhou et al., 1992] utilise un paramètre dans la procédure d'allocation de pages pour spécifier le type de la donnée qui y sera mémorisée. La conversion de type est faite sur le site destinataire, épargnant ainsi des conversions inutiles. La procédure d'allocation peut être obtenue automatiquement par une analyse des symboles des fichiers objets générés par le compilateur [Wortman et al., 1994].

Le système Stardust [Cabillic, 1997] utilise un typage de Pinkerton, qui décrit le type de la donnée sous la forme d'une séquence de types élémentaires. Il est donc possible de créer une région de mémoire virtuellement partagée en indiquant un type de Pinkerton décrivant la donnée qui y sera mémorisée.

1.4.4.4 Tolérance aux fautes

La description de mémoire virtuellement partagée tolérante aux fautes sort largement du cadre de cette introduction. Elle est basée sur des points de reprise ou des espions. Nous renvoyons le lecteur intéressé à l'étude [Morin et Puaut, 1995].

1.4.4.5 Conclusion

La programmation dans un système distribué est simplifiée par l'utilisation de mémoire virtuellement partagée. Néanmoins, dans un milieu hétérogène, la transparence à l'utilisation de cette méthode est diminuée. Mais surtout, avec l'utilisation de cohérence forte, la perte de performance engendrée par ce type de communication n'est pas négligeable. L'utilisation de cohérence plus faible permet de remédier à ce problème, mais en perdant la simplicité de programmation. En effet, le développeur d'application doit prendre en compte cette cohérence, et programmer en fonction de celle-ci.

1.5 Conclusion

Nous avons, dans ce chapitre, introduit différents modèles de programmation des systèmes distribués. Parmi les trois principaux modèles de programmation — le passage de messages, l'appel de procédures ou de méthodes à distance et la mémoire virtuellement partagée — il est parfois difficile de choisir. Historiquement, le passage de messages a été extrêmement employé, et continue à l'être, mais il requiert de la part du programmeur beaucoup plus d'efforts que les deux autres modèles. De plus, la grande diversité des sémantiques et comportements pousse le programmeur vers l'utilisation de boites à outils génériques et portables en lui faisant perdre certainement le contrôle et les performances qui font l'attrait de cette méthode. La mémoire virtuellement partagée simplifie cette programmation, mais souffre d'un rapport difficile à trouver entre la performance et la simplicité de programmation. C'est-à-dire, qu'à moins d'utiliser une cohérence atomique ou séquentielle fort coûteuse, l'application doit être construite en fonction d'un modèle de cohérence précis et compris du programmeur. Enfin, l'appel de procédures ou de méthodes à distance est très intéressant, et fréquemment utilisée, notamment par les courtiers d'objets. Le seul problème se situe dans la sémantique, en cas de défaillances, qui est difficilement exploitable sans autres mécanismes.

Chapitre 2

La réplication

La réplication de services dans un système distribué apporte deux bénéfices principaux : une meilleur sûreté de fonctionnement et une amélioration des performances des applications. Ces bénéfices sont des motivations importantes des systèmes distribués. C'est pourquoi la réplication est une caractéristique indispensable dans leur conception.

Un moyen d'assurer la sûreté de fonctionnement, c'est-à-dire la disponibilité et la fiabilité des traitements des requêtes, est de multiplier les copies sur des sites différents avec des canaux de communication différents.

Un service souvent sollicité surcharge son serveur ainsi que la machine qui l'exécute, et diminue le parallélisme par un effet de sérialisation des requêtes. En multipliant les machines sur lesquelles un service répliqué peut s'exécuter, il y a multiplication des processeurs qui exécutent ce service. De plus la réplication permet d'augmenter la proportion d'accès locaux en fournissant les moyens de rapprocher les copies d'un service des clients qui les utilisent. Il y a alors diminution des communications inter-machines ou inter-réseaux.

Le principal problème de la réplication est le maintien de la cohérence des copies. En effet, chaque copie d'un serveur étant indépendante, c'est-à-dire sans mémoire commune, il faut que des protocoles de cohérence échangent des informations sur les changements d'état d'une copie particulière pour que toutes les autres soient mises à jour.

Dans ce chapitre, nous détaillons d'abord les principaux types de réplication (§ 2.1), à savoir la réplication passive, la réplication active et la réplication semi-active. Nous les comparons (§ 2.2) par rapport à leurs coûts respectifs, leurs propriétés de tolérance aux fautes, mais surtout par rapport à leurs répercussions sur la performance des serveurs. Nous étudions ensuite les principaux modèles de cohérence utilisés (§ 2.3), c'est-à-dire ceux avec ou sans synchronisation, ceux de session et enfin ceux gérés « de bout en bout ». Enfin, nous terminons par l'étude des systèmes que nous pensons représentatifs des techniques de réplication (§ 2.4).

2.1 Types de réplication

Il existe différentes méthodes pour répliquer un service, chacune ayant des caractéristiques différentes en terme de ressources consommées, de performance et de tolérance aux fautes. Nous décrivons trois types largement répandus, à savoir la réplication passive (§ 2.1.1), la réplication active (§ 2.1.2) et la réplication semi-active (§ 2.1.3).

2.1.1 Réplication passive

La réplication passive consiste à ce qu'une seule copie reçoive la requête d'un client et l'exécute. Cette copie est désignée sous le nom de primaire (*primary*) ou coordinateur (*coordinator*).

Les autres copies sont appelées des secondaires, des sauvegardes (*backups*) ou des cohortes. Pour en assurer la cohérence, la copie primaire diffuse régulièrement son nouvel état à tous les secondaires, assurant ainsi un point de reprise (*checkpoint*).

Cette méthode permet de répliquer des serveurs non déterministes, c'est-à-dire que deux exécutions d'une requête sur un serveur dans le même état initial ne le laissent pas forcément dans le même état final. En effet, une seule copie, la primaire, exécute la commande. Les autres sont mises à jour en diffusant le nouvel état.

Quand une copie primaire défaille, un protocole d'élection a lieu pour désigner une nouvelle copie primaire parmi toutes les copies secondaires. Ces dernières sont soit identiques à l'ancienne copie primaire et alors aucun autre mécanisme n'intervient, soit représentent un état antérieur et alors des requêtes doivent être réexécutées (*rollback*) pour obtenir l'état adéquat.

Avec cette méthode, il n'est pas possible d'obtenir plusieurs réponses en provenance de plusieurs copies, il n'y a donc aucun mécanisme pour collecter et choisir ces réponses. Dans la pratique, le client envoie une requête au groupe. Le protocole de réplication lui garantit que, d'une part, une seule copie lui fournira la réponse et que, d'autre part, le nouvel état du serveur sera répercuté sur un nombre suffisant de copies secondaires.

Si une copie secondaire manque une mise à jour, cette copie sera périmée, elle ne pourra tout simplement plus devenir une copie primaire avant qu'elle récupère le dernier état du serveur.

Il existe tout de même des précautions. Une requête peut-être exécutée plusieurs fois si la copie primaire l'exécute, diffuse le nouvel état à ses copies secondaires et défaille avant de répondre au client. Pour résoudre ce problème, les copies secondaires doivent garder la réponse qu'aurait faite la copie primaire [Birman et al., 1985].

2.1.2 Réplication active

La réplication active consiste à ce que toutes les copies reçoivent les requêtes des clients, les exécutent, et y répondent. Quand une copie défaille, aucun mécanisme supplémentaire n'est à mettre en œuvre.

Il n'y a donc pas de point de reprise coûteux, mais l'exécution des requêtes doit être déterministe. Dans le cas contraire des mécanismes supplémentaires doivent être utilisés pour masquer cet indéterminisme sur toutes les copies. Enfin, des mécanismes de collecte des réponses des différentes copies, ainsi que de vote parmi les différents résultats possibles, sont nécessaires.

Le principe de la réplication active a été étudié en utilisant le modèle de machine à état. Une telle machine est constituée d'un ensemble de variables d'état et d'un ensemble de commandes. Les commandes transforment d'une façon déterministe l'état en un nouvel état. Cette machine sert les requêtes une par une. Un client est assuré que :

- O_1 : Les requêtes issues d'un même client sur une machine sont exécutées par celle-ci dans le même ordre qu'elles sont émises ;
- O_2 : Si une requête r émise par un client sur une machine a été provoquée par une requête r' émise par un autre client sur la même machine, alors celle-ci exécutera r' avant r.

Une machine à état tolérant les fautes [Schneider, 1990] [Schneider, 1993] peut être implantée par réplication active. Toutes les k copies de la machine étant dans un état initial identique, et exécutant les mêmes commandes dans le même ordre, produisent les mêmes résultats. Si une copie d'une machine fautive ne contamine pas les autres copies, alors en utilisant les réponses de toutes les copies de la machine, nous obtenons une machine tolérant k fautes.

La condition pour que toutes les copies de la machine reçoivent et exécutent toutes les requêtes dans le même ordre peut être décomposée en deux conditions distinctes :

- C_1 : Toutes les copies non fautives de la machine reçoivent les mêmes requêtes (propriété d'*accord*) ;
- C_2 : Toutes les copies non fautives de la machine exécutent les requêtes dans le même ordre relatif (propriété d'*ordre*).

Nous verrons qu'il est possible, pour des raisons de performances, de relâcher ces deux conditions.

2.1.3 Réplication semi-active

La réplication semi-active est une réplication active dans le sens où chaque copie exécute la requête. Par contre, toutes les sources d'indéterminisme sont résolues par le choix d'une copie primaire qui diffuse aux autres copies ses synchronisations. Enfin, seule la copie primaire renvoie le résultat au client.

2.2 Comparaison des types de réplication

Dans cette section, nous comparons essentiellement la réplication active et la réplication passive. Tout d'abord nous le faisons succinctement par rapport à la tolérance aux fautes (§ 2.2.1). Puis, nous comparons leur coût réseaux (§ 2.2.2), leur coût processeurs (§ 2.2.3), et surtout leurs répercussions vis-à-vis de la performance des serveurs (§ 2.2.4).

2.2.1 Tolérance au fautes

Pour que le service soit k-fiable, c'est-à-dire supporte k défaillances, il faut répliquer un certain nombre de fois le serveur. Ce nombre varie suivant le type de fautes que l'on souhaite supporter, et le type de la réplication.

Ainsi, avec la réplication passive, pour des fautes par arrêt silencieux (*fail-silent*), en cas de défaillance d'une copie secondaire, la copie primaire continue à servir les clients. Il suffit alors de générer une nouvelle copie du serveur pour remplacer la défectueuse. En cas de défaillance de la copie primaire, il doit y avoir un protocole d'élection qui désigne une nouvelle copie primaire. Il est donc clair qu'il suffit de k copies secondaires pour être k-fiable (et donc $k+1$ copies en comptant la copie primaire). Dans un contexte devant supporter les partitions réseaux le nombre de copies secondaires doit être égal à $2k+1$. Enfin, les fautes Byzantines (fautes perçues différemment par les divers intervenants) et les fautes par valeur cohérentes ne sont pas supportées, puisque seule la copie primaire exécute la requête, il ne peut donc pas y avoir de vote sur la valeur de la réponse.

La réplication active possède les mêmes caractéristiques, mais supporte plus de types de fautes. Avec des fautes par valeur, un nombre de $2k+1$ copies est nécessaire. Pour les fautes

Byzantines, il suffit de $2k+1$ copies avec des mécanismes d'authentification, et $3k+1$ copies sans ces mécanismes [Lamport et al., 1982].

2.2.2 Coûts réseaux

Pour la réplication passive dans un mode de communication synchrone, le canal entre le client et le serveur primaire doit être fiable. Entre les serveurs, les messages doivent être délivrés atomiquement. Dans un mode de communication asynchrone, les messages doivent respecter au moins un ordre causal. Dans tous les cas, le coût réseau est proportionnel au volume de données modifiées par les requêtes.

Avec la réplication active, une façon de garantir les conditions C_1 et C_2 de la machine à état est d'utiliser tout simplement un mécanisme de diffusion atomique. Mais ce type de protocole est coûteux. Il est possible de garantir séparément les deux conditions. Une diffusion fiable peut garantir la condition C_1, alors que la condition C_2 peut être fournie par des mécanismes de transaction, laissant plus de souplesse pour relâcher les conditions. Si la communication est asynchrone entre les clients et les copies du serveur, un protocole garantissant l'ordre causal doit être ajouté.

Enfin, pour la réplication semi-active, les protocoles d'accord, coûteux en réseaux, sont évités par rapport à la réplication active, puisque la copie primaire règle les problèmes d'indéterminisme.

2.2.3 Coûts processeurs

Avec la réplication passive, le coût processeur est faible, puisque, quand il n'y a pas de faute, une seule copie exécute la requête. Néanmoins, en cas d'absence de mécanisme pour détecter les zones de données modifiées, le coût processeur augmente considérablement.

En ce qui concerne la réplication active, l'exécution d'une requête se fait sur toutes les copies, le coût processeur global est donc proportionnel au nombre de copies, et donc plus élevé que pour la réplication passive.

2.2.4 Performance

2.2.4.1 Replication passive

Avec la réplication passive de base, la sauvegarde de l'état est un mécanisme coûteux, et doit être effectuée avant de répondre au client. Elle pénalise donc ce dernier. La copie primaire ne pourra pas répondre au client tant que la dernière, et donc la plus lente, des copies secondaires n'aura pas enregistré le nouvel état. Pour cela, des points de reprise périodiques minimisent ce problème, mais la copie primaire devra quand même diffuser les choix faits aux points d'indéterminisme. En effet, une copie secondaire doit être capable de refaire les mêmes choix en cas d'élection en tant que copie primaire. Enfin, si la copie primaire défaille, le service est bloqué durant le protocole d'élection puisqu'aucune copie primaire ne peut satisfaire les requêtes des clients.

De plus, ce type de réplication ne permet pas généralement de tirer partie de la réplication pour améliorer la performance d'un serveur. En effet, la copie primaire devient un goulot d'étranglement, puisque toutes les requêtes lui parviennent. Les copies secondaires ne sont que des sauvegardes prêtes à prendre le relais en cas de défaillance.

Néanmoins, il est possible de fournir plusieurs copies primaires pour un même service, soit en fonction des requêtes (chaque requête peut avoir une copie primaire différente), soit en fonction des mèthodes (chaque mèthode manipulant des données différentes utilise une copie primaire différente). Dans les deux cas, les données manipulées par ces différentes copies primaires sont indépendantes les unes des autres, ou bien des mécanismes d'exclusions mutuelles sont prévus. Avec cette méthode, le service est amélioré, car des requêtes concurrentes bénéficient de plusieurs processeurs pour s'exécuter.

2.2.4.2 Réplication active

En cas de défaillance, la réplication active n'ayant pas de mécanisme d'élection bloquant, elle fournit au client un temps de réponse plus faible et indépendant des défaillances.

De plus, un client peut attendre la première réponse, et donc continuer son exécution en se synchronisant sur la plus rapide des copies. La réplication active peut donc être utilisée pour améliorer les performances vis-à-vis d'un client, malgré le surcoût processeur qu'elle engendre.

Enfin, en prenant en compte la sémantique des commandes, on peut relâcher la condition C_1 de la machine à état. Par exemple, les commandes qui ne changent pas l'état d'une machine, c'est-à-dire les commandes de lecture seule, ne sont pas forcément diffusées à toutes les copies.

2.2.4.3 Requêtes concurrentes

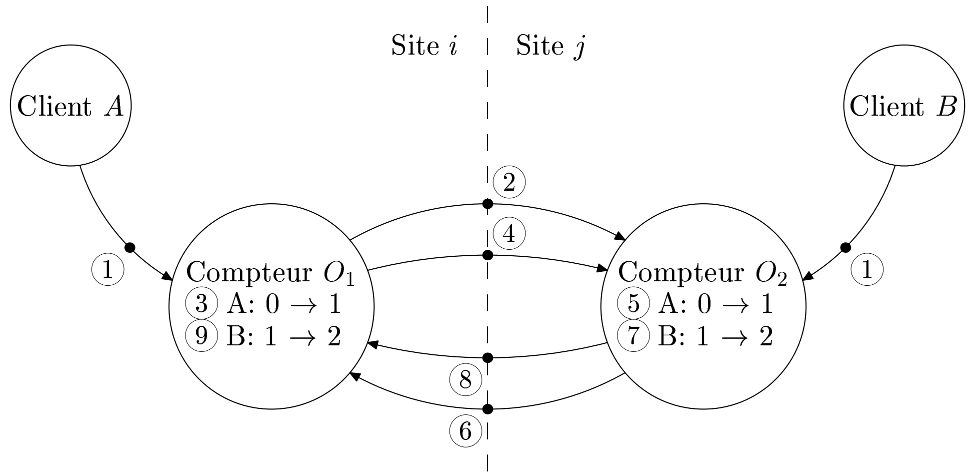
Pour comprendre les différences de ces deux méthodes, nous prendrons un exemple très simple : un compteur que nous pouvons incrémenter d'une unité à l'aide d'une seule requête. Ce compteur est répliqué sur deux sites. Deux clients, A et B, invoquent quasi simultanément la requête d'incrémentation du compteur dans les deux méthodes considérées.

Pour la réplication passive, le mécanisme de maintien de la cohérence ne connaît pas la sémantique des opérations effectuées sur les objets. Dans la plupart des cas, ce gestionnaire n'a que la connaissance des pages modifiées. Donc, le système est obligé de verrouiller totalement ou partiellement le serveur. C'est ce verrouillage qui génère la majorité du trafic réseau et qui diminue la possibilité d'exécution simultanée sur une autre copie du serveur.

Le scénario de la figure 2.1 page suivante montre le déroulement possible des opérations quand la réplication est passive. Il y a, à un instant donné, un des compteurs qui doit verrouiller la valeur et faire les modifications dûes à la requête (ici, l'incrémentation du compteur). Une fois le compteur déverrouillé et les modifications propagées, l'autre copie doit refaire la même démarche.

Toutefois, ce problème de sérialisation des requêtes sans conflit est résolu en utilisant des gestionnaires de verrouillage plus évolués (par exemple [Brun-Cottan et Makpangou, 1995]) qui permettent de verrouiller des parties des données en fonction des opérations qui y sont effectuées, ou bien des modèles de cohérence avec synchronisation (que nous verrons au § 2.3.2 page 34). Néanmoins, la mise à jour de l'état sur une autre copie ne permet pas de le résoudre complètement. Dans notre exemple, nous aurions eu un résultat identique avec la précédente méthode. En effet, si le verrouillage n'avait pas été effectué, à l'arrivée de la mise à jour de l'autre copie, nous aurions été dans l'impossibilité de « fusionner » l'état du compteur modifié par le client A avec l'état du compteur modifié par le client B.

Avec une technique basée sur la réplication active, nous obtenons alors le scénario décrit à la figure 2.2 page 26. Quand les deux clients invoquent l'incrémentation sur les deux copies du compteur, les deux copies traitent cette requête de façon indépendante. Après un certain



① Les deux clients invoquent quasi simultanément l'incrémentation du compteur sur une copie différente ;

② La copie O_1 verrouille le compteur pour travailler dessus ;

③ Cette copie incrémente le compteur suite à la requête du client A ;

④ Elle envoie une mise à jour et déverrouille le compteur ;

⑤ La copie O_2 met à jour son compteur ;

⑥ Elle verrouille le compteur à son tour pour travailler dessus ;

⑦ Elle incrémente le compteur suite à la requête du client B ;

⑧ Elle envoie une mise à jour et déverrouille le compteur ;

⑨ La copie O_1 met à jour son compteur.

Fig. 2.1 – Compteur avec réplication passive

temps, chaque copie propage l'invocation qu'elle a reçue d'un client vers l'autre copie du compteur, chaque copie traitant alors de façon tout aussi indépendante cette mise à jour.

En connaissant la sémantique des invocations, nous pouvons alléger cette synchronisation en relâchant la condition C_2 de la machine à état. En effet, nous savons par la sémantique des requêtes d'incrémentation de notre compteur, que les deux requêtes peuvent être reçues dans un ordre différent sur les deux copies sans que cela empêche l'état des deux copies de converger. De cette propriété de la méthode, nous allégeons les synchronisations entre les deux copies au point de les supprimer complètement. Nous propageons alors simplement la requête à l'autre copie. Si le client B avait demandé une multiplication du compteur (et si, bien sûr, le compteur avait une telle méthode), nous n'aurions pas pu nous passer de synchronisation, car la sémantique de ces deux méthodes nous aurait informé que nos deux requêtes ne sont pas permutables dans le temps. Nous risquions alors de faire diverger l'état des deux copies du compteur.

En résumé, la réplication passive ne permet pas l'exécution de requêtes concurrentes sur les mêmes données. Par contre, avec la réplication active, et grâce à la connaissance de la sémantique de la requête, il est possible de relâcher la condition C_2 de la machine à état en autorisant l'exécution de certaines requêtes concurrentes travaillant sur les mêmes données. Cela augmente le degré de concurrence des requêtes, et donc améliore les performances du serveur.

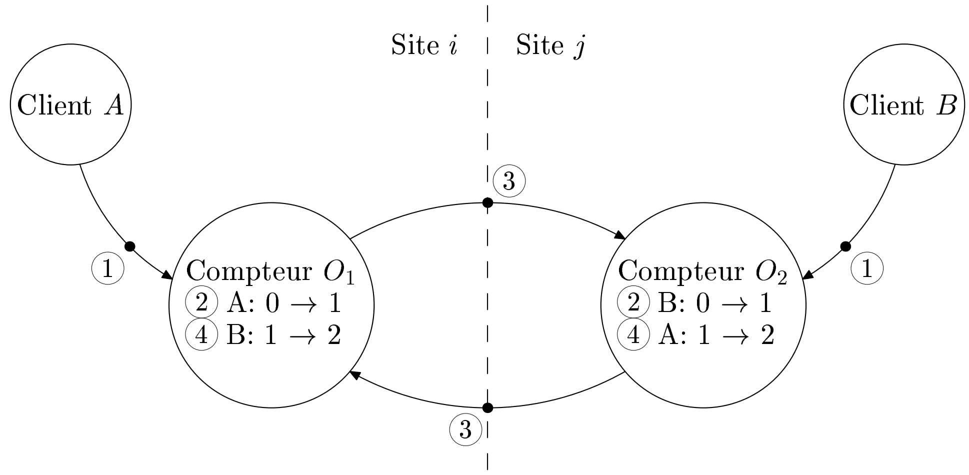
2.3 Modèles de cohérence

Les modèles de cohérence ont été introduis avec les machines multiprocesseurs partageant une mémoire commune dans le but d'offrir aux programmeurs d'applications un modèle plus ou moins proche de la programmation sur des machines monoprocesseur. Dans un tel système, le programmeur considère ses opérations d'écriture comme étant immédiates et atomiques. Dans un système distribué, les communications ne sont pas instantanées, et donc les opérations d'écriture non plus. Ce modèle est une sorte de contrat entre l'application et la mémoire qui spécifie formellement les accès à la mémoire partagée et leur perception vis à vis du programmeur. Un protocole de cohérence implante un modèle de cohérence particulier. Dans un système de réplication, un protocole de cohérence sert à imposer un certain degré de synchronisation entre les différentes copies, en imposant un certain ordre sur les mises à jour.

Par exemple, le modèle de cohérence atomique garantit un comportement similaire à de la mémoire centralisée, mais au prix de problèmes d'extensibilité et de nombreux messages réseaux (et donc d'une efficacité moindre dûe principalement à la latence). D'autres modèles ont été proposés, en relâchant la synchronisation entre les copies et l'atomicité des écritures. Il en résulte une efficacité plus grande de ces modèles, mais ils ne garantissent plus un comportement identique à de la mémoire centralisée.

Ces modèles de cohérence ont été principalement étudiés pour de la mémoire virtuellement partagée, dont les primitives d'accès sont des lectures et des écritures. Nous verrons que certains d'entre eux ont été spécifiquement adaptés à des invocations de méthode sur des objets. Pour cela, il suffit de considérer qu'une invocation de méthode modifie ou non l'état de l'objet, et donc peut être considérée comme une opération de lecture ou d'écriture.

Dans cette section, nous décrivons d'abord les modèles de cohérence sans synchronisation (§ 2.3.1), puis les modèles de cohérence faisant intervenir des synchronisations (§ 2.3.2). Nous poursuivons en présentant les garanties de session (§ 2.3.3), et nous terminons en discutant de



① Les deux clients invoquent quasi simultanément l'incrémentation du compteur sur une copie différente ;

② Chacune des deux copies de l'objet incrémente leur copie du compteur suite à la requête de clients différents ;

③ Chaque copie envoie une propagation de l'invocation indépendemment l'une de l'autre ;

④ Chaque copie incrémente sa copie du compteur suite à la requête de l'autre copie.

Fig. 2.2 – Compteur avec réplication active

la cohérence « de bout en bout » (§ 2.3.4).

2.3.1 Cohérences sans synchronisation

Les modèles de cohérence sans synchronisation utilisent les seules primitives de lecture et d'écriture. Nous présentons des exemples d'utilisation, pour illustrer concrètement les différences entre ces modèles. Pour cela, le scénario de la figure 2.3 page suivante est commun à tous les exemples de cette section.

2.3.1.1 Cohérence atomique

Nous avons déjà évoqué le modèle de cohérence atomique, il garantit un comportement similaire à de la mémoire centralisée [Censier et Feautrier, 1978]. Nous pouvons le définir ainsi :

Définition 2.1 (Modèle de cohérence atomique)
Un modèle de cohérence est atomique si une opération de lecture sur une donnée retourne la valeur affectée par la dernière opération d'écriture sur cette donnée.

Dans [Herlihy et Wing, 1990], les auteurs généralisent théoriquement le modèle de la cohérence atomique à celui des objets.

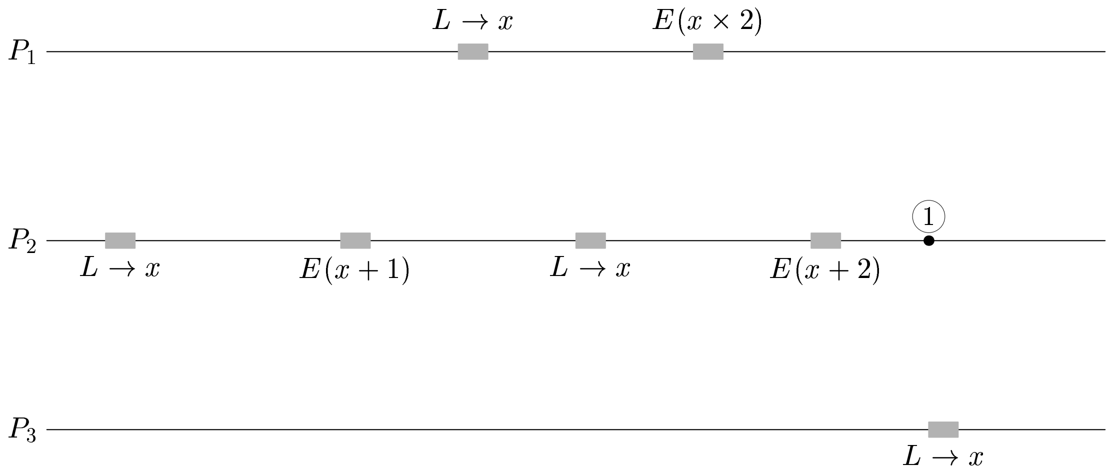
Un des protocoles de cohérence implantant ce modèle pour une mémoire virtuellement partagée est dû aux auteurs de Ivy [Li et Hudak, 1989]. Il se base sur une méthode d'invalidation de pages. Chaque page a un propriétaire. Un processus voulant effectuer une opération lecture sur une page qu'il ne possède pas demande au propriétaire une copie de la page. Le propriétaire se retire les droits d'écriture sur cette page, et lui donne une copie. Quand un processus veut effectuer une opération d'écriture sur une page qu'il ne possède pas, ou qui est protégée en écriture, il envoie une demande au propriétaire. Le propriétaire invalide d'abord toutes les copies de cette page présentes dans le système, y compris la sienne, et renvoie la page. À un instant t, une page peut donc être partagée en lecture par plusieurs processus, ou bien être exclusivement détenue en écriture par un seul processus. Cela garantit l'exclusion mutuelle des paires d'opérations lectures / écritures et écritures / écritures. Par contre, les paires d'opérations lectures / lectures peuvent être concurrentes. La difficulté essentielle est de trouver des mécanismes pour gérer efficacement le propriétaire d'une page. Plusieurs variantes sont proposées parmi des gestionnaires centralisés ou distribués, avec propriétaires fixes ou dynamiques.

D'autres protocoles existent, entre autres pour assurer la tolérance aux fautes dans les bases de données. Par exemple des algorithmes à base de vote [Thomas, 1979] [Gifford, 1979] [van Renesse et Tanenbaum, 1988] ou de quorum [Garcia-Molina et Barbara, 1985]. Un exemple de déroulement du scénario commun est présenté à la figure 2.4 page 29.

2.3.1.2 Cohérence séquentielle et objet

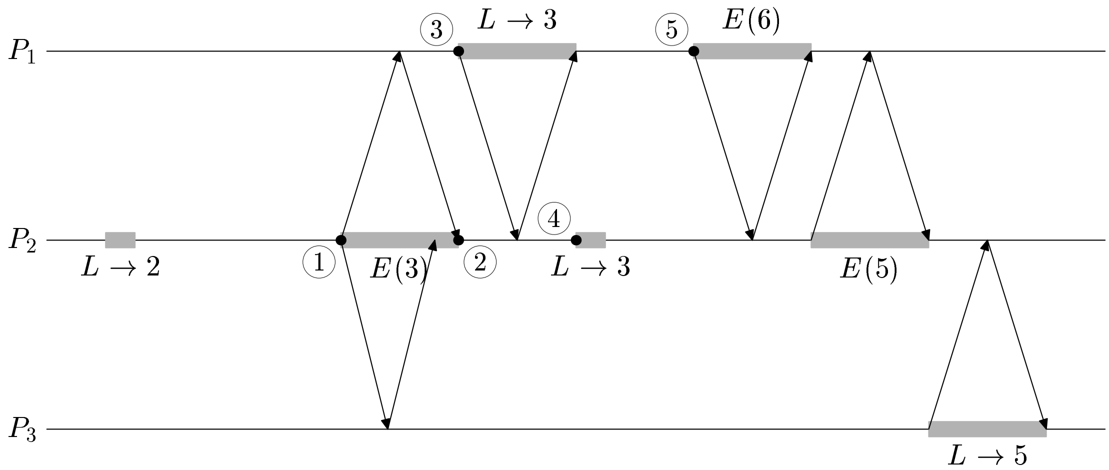
Ce modèle de cohérence a été pour la première fois proposé pour des machines multiprocesseurs dans [Lamport, 1979]. L'auteur définit ce modèle comme suit :

Définition 2.2 (Modèle de cohérence séquentielle)
Le résultat de toutes les exécutions est le même que celui où les opérations de tous les processus seraient exécutées dans un ordre séquentiel donné, et dans cette séquence, les opérations de chaque processus individuel apparaissent dans l'ordre spécifié par leur programme.



Les exemples que nous proposons sur les différents modèles de cohérence sont tous basés sur le même schéma. Il s'agit de trois processus, P_1, P_2 et $P3$ s'exécutant concurremment. Ils partagent une variable, de valeur initiale 2 sur tous les processus. Le processus P_1 exécute l'instruction $x = x \times 2$, le processus P_2 exécute les deux instructions $x = x + 1$ puis $x = x + 2$, et enfin P_3 récupère la valeur de x.
Les abscisses représentent le « temps réel ». Les opérations des processus sont représentées par un rectangle grisé. $L \rightarrow x$ représente une opération de lecture qui retourne x, alors que $E(x)$ représente une opération d'écriture de la valeur x. Enfin, le texte explicatif référence des numéros entourés d'un cercle (①, ②, etc) qui font office de repères dans la figure.
Les messages sont représentés par des flèches entre les processus, mais ne sont ni exhaustifs, ni même forcément représentatifs d'une implantation réelle particulière. Ils sont là pour faire comprendre, d'une façon simple, les interactions entre les processus, les synchronisations éventuelles et le modèle de cohérence.

Fig. 2.3 – Exemple de cohérence : scénario et légende



Le modèle de cohérence atomique retourne la valeur de la dernière écriture, aux délais de communication près. Le protocole ici présenté est celui tiré de Ivy à base d'invalidation. Dans l'état initial, tous les processus possèdent la valeur 2 avec des droits en lecture. Ainsi, nous avons aux repères :

① L'opération d'écriture déclenche l'invalidation de la page sur les processus la possédant en lecture, c'est-à-dire P_1 et P_2 ;

② Tous les processus autres que P_2 ont invalidé leur page, P_1 en est donc le seul détenteur, il peut écrire la valeur 3 ;

③ Puisque sa copie est invalidée, l'opération de lecture de P_1 déclenche une demande de la page, avec un droit de lecture ;

④ Le processus P_2 ayant encore une copie de la page en lecture, son accès en lecture ne déclenche pas de message ;

⑤ L'opération d'écriture de P_1 déclenche l'invalidation de la page sur les processus la possédant en lecture (P_2).

Fig. 2.4 – Exemple de cohérence atomique (*atomic consistency*)

Un ordre séquentiel est un ensemble d'opérations atomiques contenant l'ensemble des opérations faites par tous les processus. Il est obtenu en entrelaçant toutes les opérations de tous les processus et en conservant l'ordre des opérations effectuées par chaque processus. C'est-à-dire que la restriction de cet ensemble aux opérations faites par un processus particulier respecte l'ordre dans lequel ce processus a émis ses opérations.

Tous les processus doivent percevoir le même ordre séquentiel, c'est-à-dire le même entrelacement. Ce modèle garantit donc que tous les processus percevront les opérations dans le même ordre. Néanmoins, ce modèle ne garantit pas qu'un processus effectuant une opération de lecture obtiendra la dernière valeur affectée par une opération d'écriture émise par un autre processus. Ce comportement est le même que celui de plusieurs processus s'exécutant sur le même processeur dans un système à temps partagé. Un exemple de déroulement du scénario commun est donné à la figure 2.5 page suivante.

Un protocole de cohérence pour ce modèle doit essentiellement garantir l'ordre total des opérations. Ainsi, dans [Afek et al., 1993] et [Attiya et Welch, 1994], cet ordre total est garanti en utilisant un protocole de diffusion atomique (§ 1.4.1.2.b page 9).

On rencontre dans la littérature le terme de cohérence objet. C'est en fait la restriction de la cohérence séquentielle à un objet. Ainsi, la cohérence séquentielle n'est pas garantie sur la totalité de la mémoire, mais seulement sur celle de l'objet.

2.3.1.3 Cohérence causale

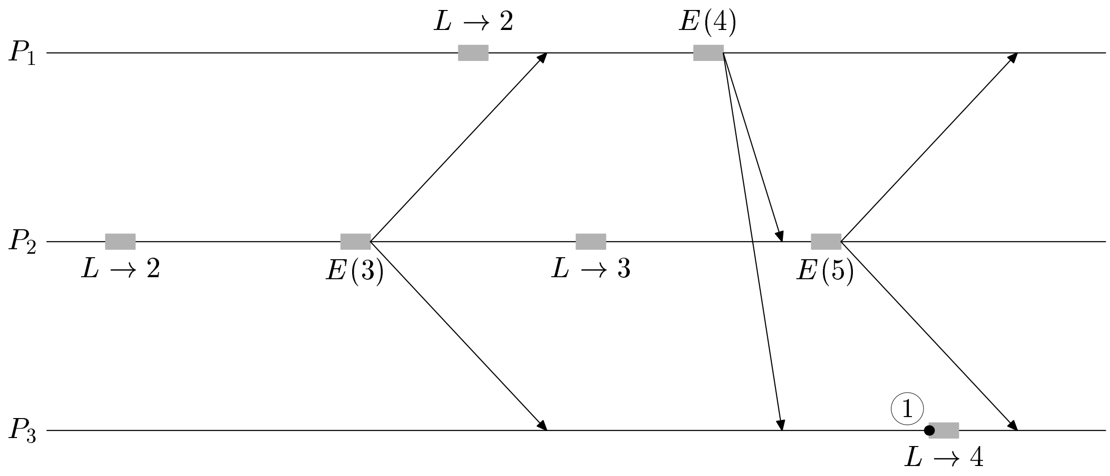
Le modèle de cohérence causale est basé sur la causalité potentielle décrite dans [Lamport, 1978]. La cohérence causale [Hutto et Ahamad, 1990] [Ahamad et al., 1991] n'est que le relâchement de la cohérence séquentielle en fonction de la causalité entre les opérations. Nous rappelons le principe de causalité qui stipule que si un événement E est causé ou influencé par un événement E', alors tout le système doit observer l'évènement E' avant l'évènement E (voir le § 1.4.1.1.c page 8). Il est alors possible de contraindre l'ordre des opérations d'écriture causalement liées, et de ne mettre aucune contrainte sur l'ordre des opérations d'écriture concurrentes, c'est-à-dire sans lien de causalité. Ceci est résumé par la définition suivante :

Définition 2.3 (Modèle de cohérence causale)
Un modèle de cohérence est causal s'il garantit les deux conditions suivantes :

1. Les opérations d'écriture potentiellement causalement liées doivent être perçues par tous les processus dans le même ordre ;
2. Les opérations d'écriture non causalement liées peuvent être perçues dans des ordres différents, sur des processus différents.

Il est à noter que tous les processus perçoivent les opérations d'écriture causalement liées dans le même ordre. Il s'agit donc d'un ordre partiel sur les opérations d'écriture. Les opérations d'écriture non comparables sont des opérations concurrentes. Il y a donc moins de synchronisation que dans le modèle de cohérence séquentielle. Le scénario commun peut se dérouler de la manière décrite à la figure 2.6 page 32.

Une implantation d'une mémoire virtuellement partagée fournissant un modèle de cohérence causale, ainsi que sa programmation, est proposée dans [Ahamad et al., 1995]. Dans cet article, ainsi que dans [Raynal et Schiper, 1995], il est démontré que certaines propriétés entre les opérations de lecture et d'écriture de programmes conçus pour une mémoire à cohérence séquentielle permettaient de les exécuter correctement sur une mémoire à cohérence causale.



Le modèle de cohérence séquentielle peut être implanté par une diffusion respectant l'ordre total. Nous notons que d'une part, tous les processus observent la même séquence sur les opérations d'écriture, et donc les mêmes valeurs dans le même ordre (2, 3, 4 puis 5), mais d'autre part qu'une opération de lecture ne renvoie pas forcément la valeur de la dernière opération d'écriture (voir repère ①).

Fig. 2.5 – Exemple de cohérence séquentielle (*sequential consistency*)

2.3.1.4 Cohérence PRAM et processeur

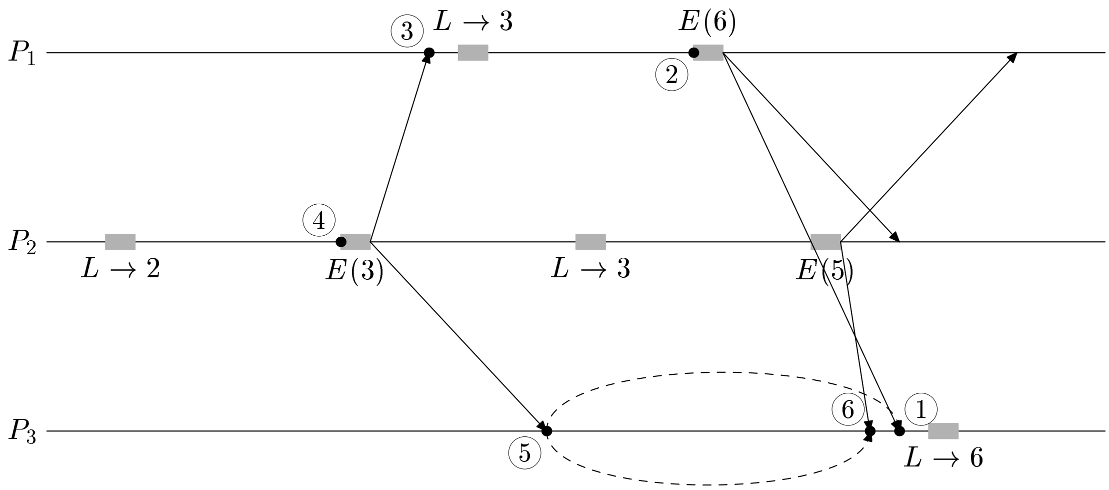
Le modèle de cohérence PRAM (*Pipelined Randomize Access Memory*) décrit dans [Lipton et Sandberg, 1988] relâche encore le modèle de cohérence causale. Voici sa définition :

Définition 2.4 (Modèle de cohérence PRAM)
Un modèle de cohérence est PRAM s'il garantit que les opérations d'écriture effectuées par un même processus sont perçues par tous les processus dans l'ordre où ces opérations ont été effectuées. Les opérations d'écriture effectuées par différents processus peuvent être perçues par chaque processus dans un ordre différent.

Autrement dit, dans la séquence d'opérations observées par un processus, les opérations d'écriture d'un processus particulier doivent apparaître dans le même ordre que celui du programme. Il s'agit d'un relâchement du modèle de cohérence causale, dans le sens où certaines transitivités dûes à la relation de causalité ne sont pas considérées dans la cohérence PRAM. La figure 2.7 page 33 exhibe une séquence de déroulement possible à partir de notre scénario commun.

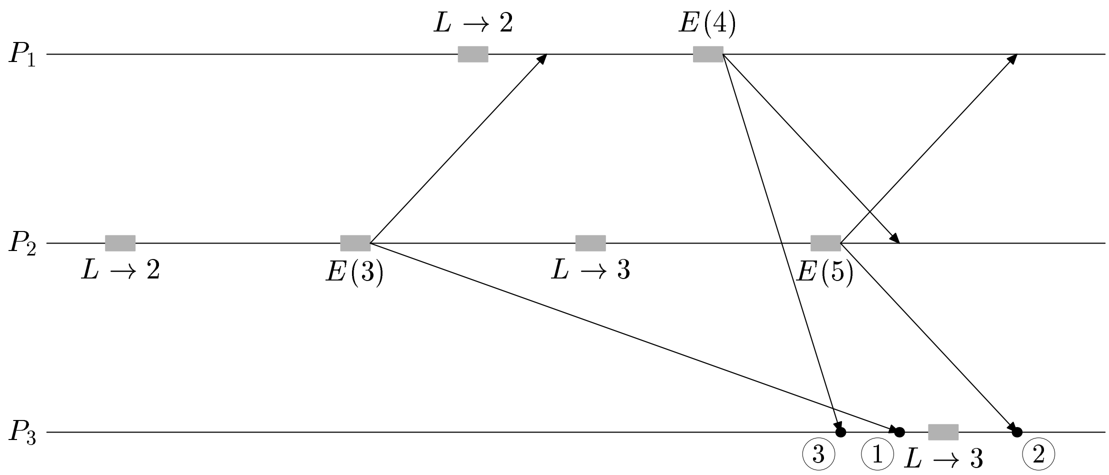
Un protocole de cohérence PRAM est facilement implantable sous la forme de files d'opérations d'écriture. Néanmoins, à cause de la difficulté à programmer avec un tel modèle de cohérence elle n'est pas extrêmement employée.

Dans [Goodman, 1989], l'auteur propose une variante de cette cohérence appelée une cohérence processeur. Il s'agit en fait d'un modèle de cohérence où une contrainte supplémentaire a été rajoutée : toutes les opérations d'écriture sur une variable particulière sont observées dans le même ordre par tous les processus. Pour résumer, les opérations d'écriture sur une variable



Dans le modèle de cohérence causale, la seule contrainte d'ordre se trouve entre des opérations d'écriture causalement liées. Les flèches en pointillés représentent des contraintes causales. Par exemple, la valeur 6 sur P_3 (repère ①), est bien sûr causalement liée à l'opération d'écriture sur P_1 (repère ②), qui elle même est causalement liée à la lecture précédente. Mais cette lecture renvoie une valeur qui a été affectée par le message de P_2 (repère ③), et donc causalement liée à l'opération d'écriture sur P_2 (repère ④) qui a générée un message sur P_3 (repère ⑤). Puisqu'ils sont causalement liés, le message arrivé au repère ① doit être délivré après le message arrivé au repère ⑤. De même le message arrivé au repère ⑥ est causalement lié au message arrivé au repère ⑤ et doit donc être délivré après lui. Par contre, il n'y a aucune contrainte causale entre les messages arrivés au repère ① et au repère ⑥, ils peuvent donc être délivrés dans un ordre arbitraire, éventuellement différent sur chaque processus.

Fig. 2.6 – Exemple de cohérence causale (*causal consistency*)



Dans le modèle de cohérence PRAM, deux opérations d'écriture issues d'un même processus doivent être délivrées sur tous les processus dans leur ordre d'émission. Ainsi, le message arrivé au repère ① et celui arrivé au repère ② sont nécessairement dans cet ordre, car tous les deux sont issus dans ce même ordre du même processus P_2.
Par contre, le message arrivé au repère ③ n'a aucune contrainte vis-à-vis des messages issus d'un processus différent.

Fig. 2.7 – Exemple de cohérence PRAM (PRAM *consistency*)

donnée sont régies par une cohérence objet et donc observées dans le même ordre sur tous les processus. Par contre, les opérations d'écriture issues d'un même processus sont observées dans l'ordre du programme par tous les processus.

2.3.1.5 Cohérence hybride

Le modèle de cohérence hybride [Attiya et Friedman, 1992] étiquette les opérations avec le critère fort (*strong*) ou faible (*weak*). L'idée est que le programmeur de l'application peut ainsi ajuster, opération par opération, la « force » de la cohérence, et donc de la synchronisation. Une définition plus formelle suit.

Définition 2.5 (Modèle de cohérence hybride)
Un modèle de cohérence est hybride s'il garantit les deux conditions suivantes :

1. Les opérations fortes (*strong*) doivent être perçues dans un ordre séquentiel ;
2. Si deux opérations sont émises par un même processus et si l'une d'entre elles au moins est forte (*strong*), alors elles doivent être perçues dans l'ordre du programme par tous les processus.

Ainsi, toutes les paires d'opérations faisant au moins intervenir une opération forte sont observées dans le même ordre par tous les processus. En étiquetant toutes les opérations comme fortes, le modèle est similaire à celui de la cohérence séquentielle. Nous pouvons trouver dans [Attiya et al., 1993] des propriétés intéressantes concernant l'étiquetage. Ainsi, si toutes les opérations de lecture sont faibles, et toutes celles d'écriture sont fortes (écritures fortes ou *Strong Writes*), ou bien l'inverse (lectures fortes ou *Strong Reads*), alors l'exécution est identique à celle d'une cohérence séquentielle.

2.3.1.6 Cohérence « à la longue »

Ce modèle de cohérence est le plus faible, aucun ordre entre les opérations n'étant imposé.

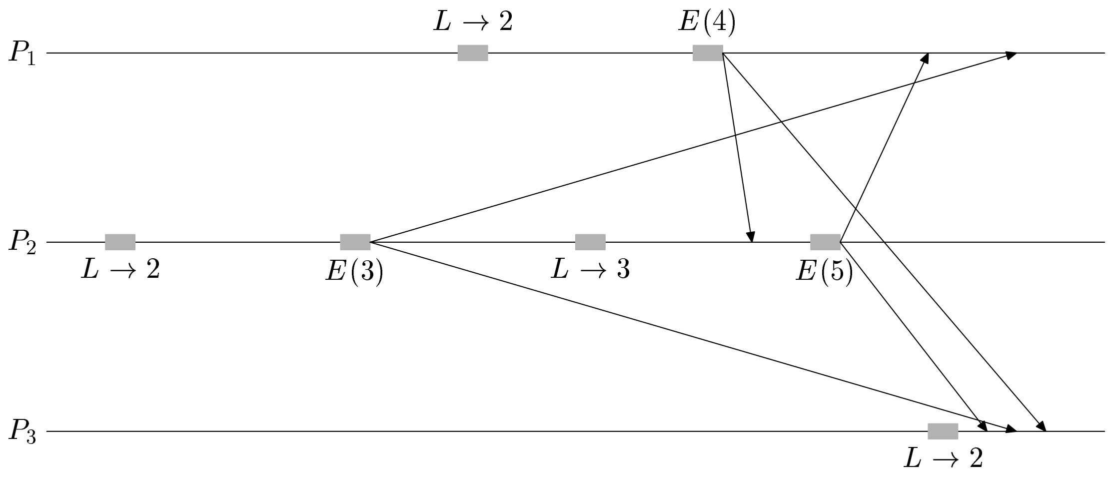
Définition 2.6 (Modèle de cohérence « à la longue »)
Un modèle de cohérence est « à la longue » s'il garantit que les opérations d'écriture seront propagées tôt ou tard.

Ce modèle n'impose aucune contrainte sur l'ordre des opérations d'écriture. Son seul rôle est de propager les modifications à toutes les copies. Un exemple trivial de scénario possible est proposé à la figure 2.8 page ci-contre.

2.3.2 Cohérences avec synchronisation

Les modèles de cohérence avec synchronisations utilisent, en plus des primitives de lecture et d'écriture, des primitives manipulant des variables de synchronisation. En effet, avec les modèles de cohérence sans synchronisation, les opérations pour synchroniser les processus se font par des mécanismes indépendants. Or, en intégrant ces synchronisations avec les modèles de cohérence, il est possible de relâcher des contraintes, et donc d'obtenir un protocole de cohérence moins coûteux et plus performant.

Par exemple, il est fréquent de protéger des variables partagées par une exclusion mutuelle entre les processus. Le processus possédant la section critique est donc le seul à lire et à



Dans le modèle de cohérence « à la longue », aucune contrainte sur l'ordre n'est imposée. La seule règle est qu'un message doit nécessairement arrivé tôt ou tard.

Fig. 2.8 – Exemple de cohérence « à la longue » (*eventual consistency*)

modifier ces variables. Il est donc raisonnable d'alléger la cohérence, puisque ces variables ne sont plus partagées à cet instant précis.

L'exemple de scénario que nous avons utilisé jusqu'alors doit être modifié, puisque les synchronisations imposent un ordre « temps réel » différent de celui choisi. Néanmoins, les exemples sont assez similaires pour en comprendre les différences.

2.3.2.1 Cohérence faible

Le modèle de cohérence faible (*weak consistency*) [Dubois et al., 1986] fait intervenir des variables de synchronisation, que le programmeur doit accéder de façon implicite pour gérer la cohérence. La définition de ce modèle est alors la suivante :

Définition 2.7 (Modèle de cohérence faible)
Un modèle de cohérence est faible s'il garantit les trois propriétés suivantes :

1. Les opérations sur les variables de synchronisation se font selon un modèle de cohérence séquentielle ;
2. L'accès aux variables de synchronisation ne peut se terminer que si toutes les opérations d'écriture et de lecture sont terminées sur tous les sites ;
3. Les opérations de lecture et d'écriture ne peuvent se faire que si toutes les opérations sur les variables de synchronisation sont terminées sur tous les sites.

Les opérations sur les variables de synchronisation sont régies par une cohérence séquentielle, c'est-à-dire sont observées dans le même ordre par tous les processus. Quand un processus effectue une opération sur une variable de synchronisation, il a alors la garantie que, d'une part, toutes les modifications qu'il a effectuées sont reçues par tous les autres processus, et, d'autre part, qu'il a reçu toutes les modifications faites par tous les autres processus.

Il est clair que si un processus modifie plusieurs fois de suite la même variable entre deux accès à une variable de synchronisation, seule la dernière valeur sera propagée aux autres processus. Un scénario possible basé sur des synchronisations de type *rendez-vous* est présenté à la figure 2.9 page suivante.

Ce modèle permet à un protocole de cohérence faible, de regrouper et de propager dans un seul message toutes les modifications qu'a effectuées un processus. Néanmoins, les accès aux variables de synchronisation génèrent des messages supplémentaires. De plus, le protocole doit savoir quelles sont les modifications effectuées par un processus entre deux accès à une variable de synchronisation. Étant essentiellement basés sur des comparaisons de pages mémoires, ces mécanismes sont plus ou moins coûteux en ressource processeur.

2.3.2.2 Cohérence au relâchement

Dans le précédent modèle, on ne distingue pas le commencement d'un accès de sa fin. Cela s'apparente plus à des barrières de synchronisation. Dans leur architecture Dash, les auteurs de [Gharachorloo et al., 1990], définissent le modèle de cohérence au relâchement (*release consistency*). Deux opérations sur les variables de synchronisation existent : acquérir (*acquire*) et relâcher (*release*). La définition est alors la suivante :

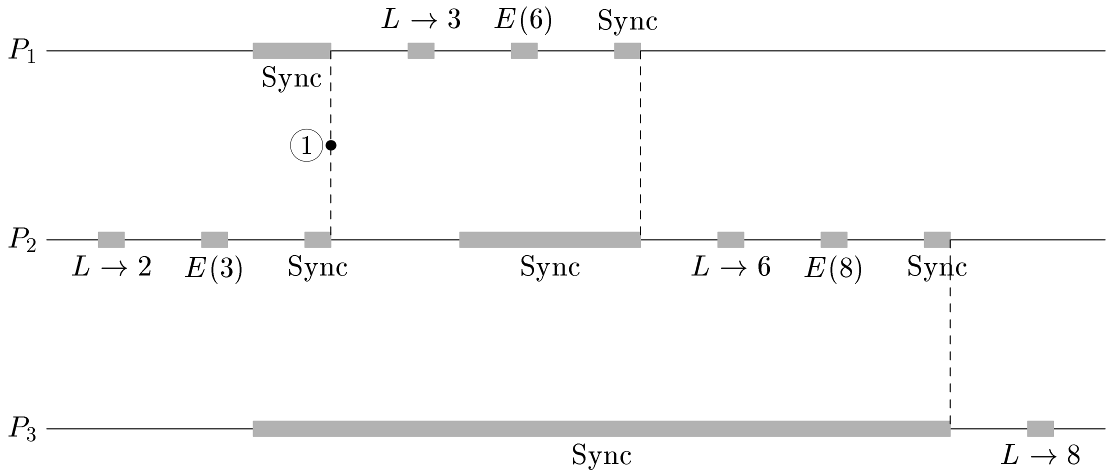
Définition 2.8 (Modèle de cohérence au relâchement)
Un modèle de cohérence est au relâchement s'il garantit les trois propriétés suivantes :

1. Les opérations de synchronisation (*acquire* et *release*) se font selon un modèle de cohérence processeur ;
2. Les opérations de lecture et d'écriture ne peuvent se faire que si toutes les opérations d'acquisition (*acquire*) précédentes sont terminées sur tous les sites ;
3. L'opération de relâchement (*release*) ne peut se terminer que si toutes les opérations d'écriture et de lecture sont terminées sur tous les sites.

Les opérations sur les variables de synchronisation sont régies par une cohérence processeur, c'est-à-dire que celles issues d'un même processus sont observées dans le même ordre par tous les processus. Quand un processus effectue une opération d'acquisition, il observera toutes les modifications effectuées par les autres processus avant les relâchements précédents. Quand un processus effectue une opération de relâchement, les modifications qu'il aura effectuées seront observées par tous les processus faisant une acquisition ultérieure.

Il existe deux protocoles mettant en œuvre ce modèle. Il s'agit de la cohérence au relâchement impatiente (*eager release consistency*) et la cohérence au relâchement paresseuse (*lazy release consistency*). Dans Munin [Carter et al., 1991], qui implante le protocole impatient, toutes les modifications entre les deux opérations de synchronisation sont propagées au moment du relâchement à tous les autres processus, même ceux qui n'effectueront pas d'acquisition ultérieure. Un scénario possible d'un tel protocole est indiqué à la figure 2.10 page 38.

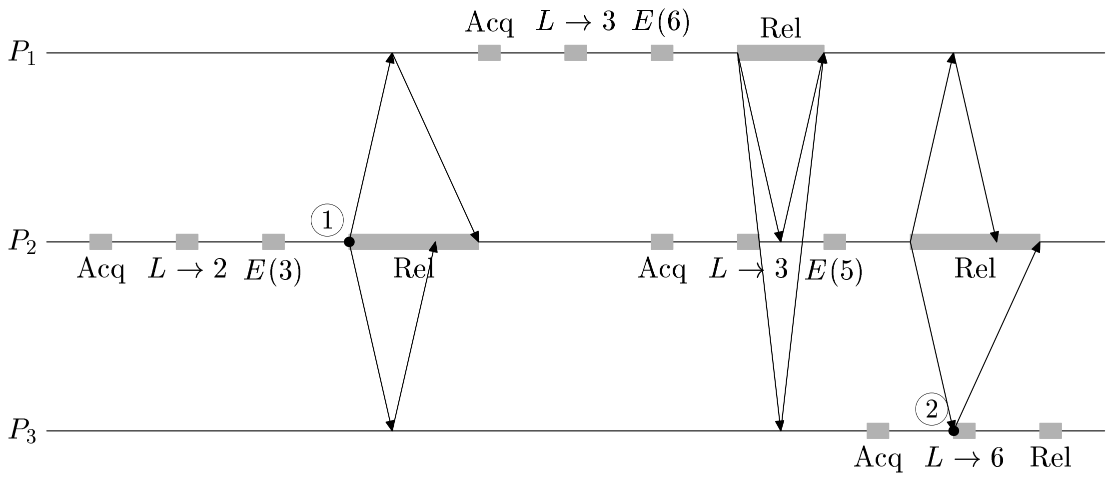
Par contre, dans le protocole paresseux [Keleher, 1995], toutes les modifications entre les deux opérations de synchronisation sont propagées au moment d'une acquisition par un autre processus. Le surcoût dû à ce protocole est compensé par le nombre et la taille des messages échangés. Cette variante est illustrée par l'exemple de la figure 2.11 page 39. L'implantation de cette variante par TreadMarks [Amza et al., 1996] propage les modifications encore plus paresseusement puisqu'au moment du premier accès après l'acquisition.



Dans le modèle de la cohérence faible, l'ordre est géré par un ordre séquentiel sur des variables de synchronisation. Ces variables doivent être utilisées par un mécanisme de synchronisation. Ici, nous avons représenté l'accès aux variables de synchronisation par *Sync*. Le mécanisme de synchronisation est fourni sous forme de *rendez-vous* (repère ①).

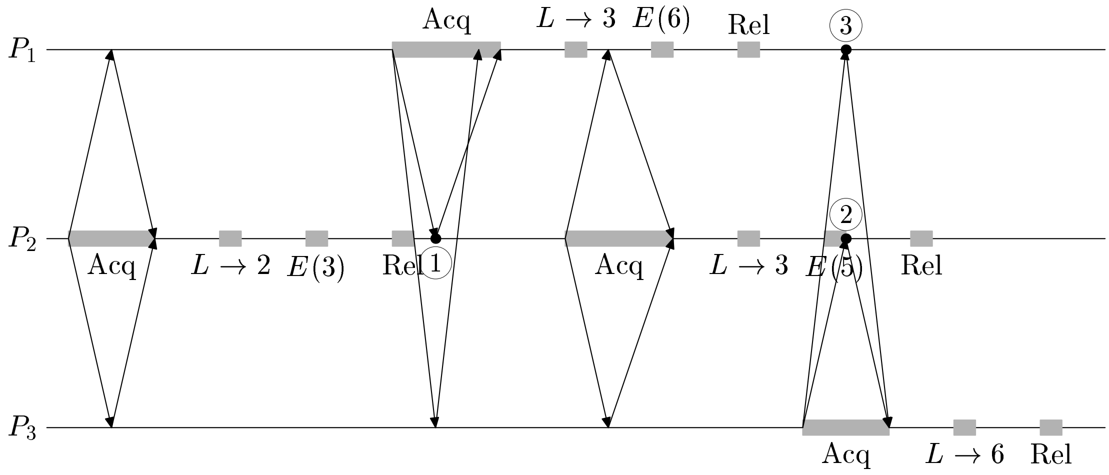
Les messages assurant ce mécanisme de synchronisation ne sont pas représentés. Par contre, ces accès garantissent que les modifications effectuées par un processus seront observables par les autres processus.

Fig. 2.9 – Exemple de cohérence faible (*weak consistency*)



Dans le modèle de cohérence au relâchement, les accès aux variables de synchronisation sont observables dans un ordre processeur. Il existe deux types d'accès à ces variables, l'acquisition (*Acq*) et le relâchement (*Rel*).
Dans la variante impatiente, les modifications sont propagées à tous les autres processus lors d'un relâchement et seront observables lors d'une acquisition ultérieure. Ainsi, au repère ①, il y a une diffusion de la nouvelle valeur 3.
Par contre, au repère ②, l'opération de lecture renvoie la valeur 6 car le dernier relâchement observé par P_3 avant l'acquisition provient de P_1 et propage la valeur 6. Le relâchement en provenance de P_2, et donc la connaissance de la valeur 5, ne sera observable sur P_3 qu'à une acquisition ultérieure.

FIG. 2.10 – Exemple de cohérence au relâchement impatiente (*eager release consistency*)



Dans le modèle de cohérence au relâchement, les accès aux variables de synchronisation sont observables dans un ordre processeur. Il existe deux types d'accès à ces variables, l'acquisition (*Acq*) et le relâchement (*Rel*).
Dans la variante paresseuse, les modifications sont indiquées sur la demande d'une acquisition. Ainsi, au repère ①, l'acquisition du processus P_1 provoque une demande des dernières valeurs modifiées sur les processus P_2 et P_3. Bien sûr, seul P_2 a modifié la valeur, et la communique à P_1.
L'acquisition du processus P_3 se déroule de même, c'est-à-dire demande les modifications de la valeur avant le précédent relâchement. C'est pour cette raison qu'au repère ②, le processus P_2 ne renvoie pas la valeur 5, contrairement au processus P_1, qui au repère ③ renvoie sa dernière modification, c'est-à-dire 6.

FIG. 2.11 – Exemple de cohérence au relâchement paresseuse (*lazy release consistency*)

2.3.2.3 Cohérence à l'entrée

Le modèle de cohérence à l'entrée (*entry consistency*) [Bershad et Zekauskas, 1991] affaiblit encore celui au relâchement. En effet, ce dernier ne distingue pas les variables partagées. La cohérence à l'entrée se distingue par l'association d'une variable de synchronisation s pour une ou plusieurs variables partagées D_s. Ainsi, le modèle de cohérence à l'entrée peut se définir ainsi :

Définition 2.9 (Modèle de cohérence à l'entrée)
Un modèle de cohérence est à l'entrée s'il garantit les trois propriétés suivantes :

1. Une acquisition (*acquire*) d'une variable de synchronisation s ne peut se terminer que si toutes les modifications de D_s ont été effectuées ;
2. Avant qu'un accès exclusif à une variable de synchronisation soit susceptible de se terminer, plus aucun processus ne doit détenir cette variable de synchronisation dans un mode non-exclusif ;
3. Après qu'un accès exclusif par un processus p à une variable de synchronisation s soit terminé, tous les accès en mode non-exclusif à cette variable s ne pourront se terminer que quand toutes les modifications de D_s par p seront mises à jour.

Aucun processus ne peut obtenir des valeurs de D_s plus vieilles que la dernière modification si tous les accès à D_s sur tous les processus se situent dans une section critique représentée par la variable de synchronisation s. Une variable de synchronisation s ne peut pas être détenue par plus d'un processus. Il s'agit du propriétaire de s qui est le dernier processus ayant fait un accès exclusif à s. Dans un mode non-exclusif, plusieurs processus peuvent avoir une copie de la variable de synchronisation. Enfin, la dernière propriété assure que les processus peuvent faire des accès en mode non-exclusif à une variable de synchronisation s sans communiquer avec le propriétaire de s tant qu'un accès à s en mode exclusif n'a pas été effectué.

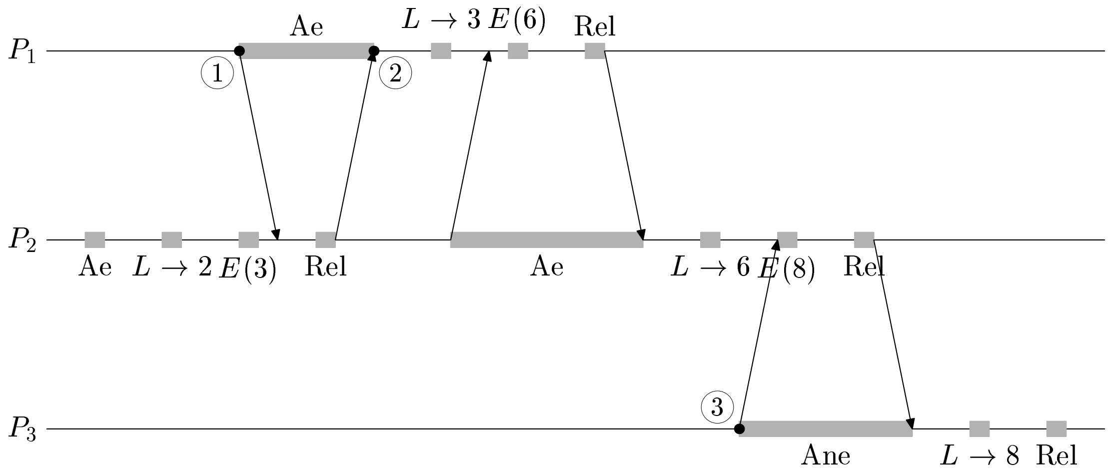
Concrètement, une variable de synchronisation s sert à protéger des données partagées D_s comme le ferait une section critique. Si durant cette section critique, D_s sont modifiées, il faut acquérir s en mode exclusif. Par contre, si D_s sont seulement consultées, il faut acquérir s en mode non-exclusif. Cela s'apparente donc à un verrou écrivain / lecteurs. La figure 2.12 page ci-contre exhibe un exemple d'utilisation d'un tel modèle de cohérence.

Le système Midway [Bershad et al., 1993] fournit une mémoire virtuellement partagée implantant un protocole de cohérence à l'entrée, ainsi que des outils pour écrire les programmes (compilateur modifié). Il s'agit de définir à la compilation les variables de synchronisation et les variables partagées. Il faut, dynamiquement durant l'exécution, associer les données partagées D_s à la variable de synchronisation s par des appels explicites.

Ce modèle permet des accès concurrents aux différentes sections critiques protégeant des variables indépendantes. Malgré cela, l'association des variables partagées aux variables de synchronisation étant manuelle, elle introduit un risque d'erreurs non négligeable.

2.3.3 Cohérences de session

Les cohérences de session ne sont pas exactement des modèles de cohérence comme ceux décrits précédemment qui assurent la cohérence des données partagées entre des copies ou des processus coopératifs. Les cohérences de session sont des modèles basés sur les clients. Leurs auteurs [Terry et al., 1994] les appellent des garanties de session. Dans un système



Dans le modèle de la cohérence à l'entrée, les accès aux variables de synchronisation sont gérés par trois opérations spéciales. L'acquisition exclusive (Ae) garantit qu'un seul processus à la fois peut exécuter le code jusqu'au prochain relâchement. L'acquisition non-exclusive (Ane) garantit que seuls les processus demandant cet accès peuvent exécuter le code en même temps jusqu'au prochain relâchement. Et enfin le relâchement (*Rel*) termine l'acquisition et rend observable les modifications.

Ainsi, au repère ①, le processus P_1 demande une acquisition exclusive, il est donc bloqué jusqu'à ce que le processus P_2 exécute un relâchement (repère ②). Il en est de même pour les autres acquisitions et relâchements. Enfin, au repère ③, l'acquisition est non-exclusive car le processus P_3 ne modifie pas la donnée protégée.

FIG. 2.12 – Exemple de cohérence à l'entrée (*entry consistency*)

avec des copies d'un serveur maintenues par une cohérence faible, et avec des clients pouvant adresser à tout moment une copie quelconque (*read-any / write-any*), ces derniers, ainsi que leurs utilisateurs, peuvent être surpris des résultats obtenus.

Une session est une séquence d'opérations de lecture et d'écriture effectuées durant l'exécution d'une application. Des garanties de certaines propriétés sont proposées aux applications sur les résultats de leurs opérations.

2.3.3.1 « Lire ses écritures »

La première d'entre elles, « lire ses écritures » (*Read Your Writes*), garantit à une application qu'une opération d'écriture d'une session sera visible par toutes les opérations de lecture ultérieures dans la même session :

Définition 2.10 (Modèle de cohérence « lire ses écritures »)
Si une opération d'écriture E précède une opération de lecture L de la même session, alors L sera exécutée sur une copie où E aura été exécutée.

Cette condition ne garantit pas qu'une opération de lecture retournera forcément la valeur de sa dernière écriture sur la même donnée, puisqu'une opération d'écriture d'une autre session aura très bien pu changer cette valeur.

Nous pouvons comparer ce modèle de cohérence à un modèle qui préserve l'ordre programme, c'est-à-dire l'ordre dans lequel les opérations sont émises par un client.

2.3.3.2 « Lectures monotones »

La garantie des « lectures monotones » (*Monotonic Reads*) permet à un utilisateur d'observer le serveur d'une façon continue, c'est-à-dire sans régression dans les mises à jour. En voici la définition :

Définition 2.11 (Modèle de cohérence « lectures monotones »)
Si une opération de lecture L précède une opération de lecture L' durant la même session, alors L' sera exécutée sur une copie qui aura exécutée toutes les opérations d'écriture conditionnant L.

Les opérations d'écriture conditionnant une opération de lecture sont toutes les opérations d'écriture qui ont participé à la valeur renvoyée par l'opération de lecture.

2.3.3.3 « Les écritures suivent les lectures »

La garantie suivante, « les écritures suivent les lectures » (*Writes Follow Reads*), se définit ainsi.

Définition 2.12 (Modèle de cohérence « les écritures suivent les lectures »)
Si une opération de lecture L précède une opération d'écriture E dans une session, alors toutes les copies ayant exécuté E auront exécuté avant E toutes les opérations d'écriture conditionnant L.

Cette garantie est légèrement différente des précédentes, car elle influe sur l'ordre des opérations d'écriture d'autres clients qui n'ont pas forcément demandé une telle garantie. Cette garantie s'apparente à un ordre causal.

2.3.3.4 « Écritures monotones »

Enfin, la dernière garantie, nommée « écritures monotones » (*Monotonic Writes*), permet à un client d'être sûr que les opérations d'écriture seront exécutées sur toutes les copies dans leur ordre d'émission. La définition de cette garantie est la suivante :

Définition 2.13 (Modèle de cohérence « écritures monotones »)
Si une opération d'écriture E précède une opération d'écriture E' dans la même session, alors toutes les copies ayant exécuté E' auront exécuté E avant.

Nous pouvons remarquer que c'est un modèle cohérence PRAM, mais basé sur un unique client.

2.3.4 Cohérence « de bout en bout »

Ce modèle de cohérence ne définit pas un modèle explicite. Il s'agit en fait de cohérence spécifique, pilotée par l'application elle-même, pouvant prendre en compte la sémantique des opérations. Nous voyons au travers du nombre de modèles de cohérence existants, qu'il y a un compromis entre le degré de synchronisation, et par conséquent le nombre de messages réseaux ainsi que la latence de ces messages, et la facilité de développement de l'application.

En partant du principe que chaque application a ses besoins spécifiques, des auteurs, comme dans [Cheriton, 1988], proposent des solutions où l'application fournit elle-même la cohérence à un degré de synchronisation dont elle a besoin. Le même auteur, dans un autre article [Cheriton et Skeen, 1993], montre les limites de CATOCS (*Causally And Totally Ordered Communication Support*). Avec la même philosophie, les auteurs exhibent des classes d'applications et des exemples où l'utilisation de supports de communications génériques ajoutent des surcoûts facilement évitables par l'utilisation de techniques spécifiques à l'application. Le lecteur intéressé peut se référer aux réponses que cet article a engendrées [Birman, 1993], [van Renesse, 1994] et [Cooper, 1994], aux commentaires suscités [Shrivastava, 1994] et enfin à la réponse finale [Cheriton et Skeen, 1994].

CHERITON s'appuie sur l'argument « de bout en bout » [Saltzer et al., 1984]. Ce dernier article décrit le principe suivant : un service placé à un niveau inférieur peut être redondant ou peu utile en comparaison avec le coût qu'il engendre à un tel niveau. En effet, ce service peut ne pas être optimum ou totalement implanté sans l'aide et la connaissance des applications l'utilisant à chaque bout. Par exemple, un transfert de fichier ne peut être considéré comme réussi d'une façon sûre et certaine que si, au niveau applicatif, un accusé de bonne exécution est transmis. En effet, des erreurs, comme un disque plein, peuvent survenir. Dans ces conditions, un protocole de communication fiable peut alors sembler superflu.

Reprenons l'exemple de CATOCS, qui peut être facilement transposable à un modèle de cohérence causale. Des communications cachées peuvent provoquer des relations de causalité non détectées par le système. Ainsi, un champ d'une base de données, ou bien des événements extérieurs comme des capteurs de feux, ou tout simplement l'utilisateur peuvent transporter des relations causales qui seront donc violées par le système. À l'inverse, une communication entre deux processus ne provoque pas forcément une relation causale, mais un système bas niveau ne pourra pas s'en rendre compte, et la considérera comme pertinente. Cela engendre alors des dépendances causales inutiles, et donc une dégradation des performances.

D'un autre côté, les concepteurs de ces systèmes et supports argumentent que leurs outils permettent une programmation plus simple, et donc plus facile à maintenir et à corriger. De plus, cela évite pour des applications similaires de concevoir les mêmes mécanismes plusieurs fois, pour assurer les mêmes propriétés. Pourtant, cet argument est affaibli par l'utilisation de librairies.

Pour conclure, nous voyons que l'on rejoint le compromis entre facilité de conception, programmation, maintenance, et l'efficacité des solutions proposées. Il nous semble néanmoins important de laisser à l'application la possibilité de contrôler la cohérence comme elle le souhaite. C'est ce que nous appelons le modèle de cohérence « de bout en bout ».

2.3.5 Résumé

De nombreux modèles de cohérence ont été inventés. Il en existe d'autres que nous n'avons pas présentés dans cette section, comme le modèle de cohérence mixte [Agrawal et al., 1994] ou le modèle de cohérence faible du système Dream [Dumoulin, 1997] — différent du modèle de cohérence faible (*weak consistency*) présenté précédemment. Une étude plus formelle que la nôtre sur les principaux modèles de cohérence sans synchronisation est proposée dans [Thia-Kime, 1997].

Les modèles de cohérence forte ont des comportements plus ou moins similaires à de la mémoire centralisée, et donc facilitent grandement la programmation et la mise au point des applications. En contrepartie, les protocoles de cohérence forte sont extrêmement coûteux (nombre et taille des messages), et réduisent les performances des applications par des synchronisations trop fortes et souvent inutiles.

Les modèles de cohérence faible, avec ou sans synchronisation, tendent à diminuer la taille et le nombre des messages, et donc à diminuer les temps d'accès. En contrepartie, ces modèles sont plus difficiles à comprendre, et donc à programmer. Dans certains d'entre eux, l'ajout de synchronisation augmente le risque d'erreur.

Le choix d'un modèle est un compromis entre le degré de synchronisation des copies, et les performances. C'est essentiellement un choix que seul le programmeur d'applications peut faire. Il connaît le besoin de son application, et connaît le rapport entre le surcoût de développement avec un modèle de cohérence faible, et les gains que cela procure en terme de performance et/ou de ressources consommées.

2.4 Systèmes

Cette section présente quelques systèmes supportant la réplication, ainsi que quelques systèmes plus éloignés de ce domaine, mais introduisant des concepts intéressants pour le support de cette technique. Nous avons essayé de présenter des systèmes parmi les plus représentatifs, à notre connaissance, des différentes méthodes existant à ce jour. Peu d'entre eux ont comme objectif principal d'améliorer les performances des applications, mais tous ont un souci de performance dans leurs propositions de réplication.

Ainsi, nous décrivons Grapevine (§ 2.4.1) pour son schéma « *read-any / write-any* », la *lazy replication* (§ 2.4.2) pour le support de plusieurs cohérences dont une se rapproche de celle « de bout en bout », Psync (§ 2.4.3) pour son utilisation de la sémantique des opérations afin de relâcher des contraintes d'ordre, Bayou (§ 2.4.4) pour son protocole d'« anti-entropie » et ses garanties de session variées, le système coordinateur / cohortes d'Isis (§ 2.4.5) pour son objectif principal d'amélioration de performance et enfin Arjuna (§ 2.4.6) pour la gestion de la cohérence « de bout en bout ». Nous présentons aussi Globe (§ 2.4.7) et CORBA (§ 2.4.8) pour l'intégration de la réplication dans un système à objets.

2.4.1 Grapevine

Le système Grapevine [Birrell et al., 1982] offre une réplication de serveur. Son but est de fournir essentiellement des services de courrier, mais aussi des services de nommage (de personnes, de machines et de services), des services d'authentification (de personnes et machines) et enfin des services de localisation (de services).

Il fut un des premiers systèmes à expérimenter la réplication suivant un schéma « *read-any / write-any* », c'est-à-dire où un client est susceptible d'adresser n'importe quelle copie d'un serveur, y compris d'en changer pour une raison ou une autre (défaillance, performance, etc). La cohérence entre les serveurs est extrêmement faible, occasionnant certains troubles pour les utilisateurs (par exemple, le changement d'un mot de passe sur une copie d'un serveur, mais l'utilisation de l'ancien sur une autre).

2.4.2 *Lazy replication*

La *lazy replication* (réplication paresseuse) [Ladin et al., 1990] a été développée essentiellement pour fournir une haute disponibilité des données. Pour cela, la réplication est active, et les contraintes de cohérence sont affaiblies, ce qui fournit, en plus d'une disponibilité accrue, un temps de réponse meilleur.

Le système supporte des défaillances par arrêt silencieux, mais ne supporte pas les fautes Byzantines, les partitions réseaux, les pertes, les duplications et les retards de messages. Ce système utilise obligatoirement des canaux de communication FIFO. Les copies d'un serveur ont accès à une mémoire stable (c'est-à-dire non volatile) pour stocker leur état. Enfin, le nombre et la localisation des copies sont fixes.

Trois types d'ordres de livraison des messages sont possibles, (appelés différemment dans [Ladin et al., 1992]) :

1. Ordre client (*client-order*) qui correspond à un ordre que le client détermine. Il est possible de concevoir des intercepteurs, pour utiliser une cohérence causale d'une façon transparente vis-à-vis du client. Néanmoins, des clients sophistiqués peuvent utiliser directement les différents mécanismes pour assurer leur propre cohérence, se rapprochant ainsi du modèle de cohérence « de bout en bout » ;
2. Ordre serveur (*server-order*), ou mise à jour forcée (*forced updates*) qui correspond à un ordre total. Les opérations « forcées » sont donc reçues sur toutes les copies dans le même ordre. Il n'y a pas d'ordre entre ces opérations là, et celles vues précédemment ;
3. Ordre global (*global-order*), ou mise à jour immédiate (*immediate update*) qui fournit un ordre total, vis-à-vis de toutes les opérations, c'est-à-dire des opérations régies par l'ordre client et l'ordre serveur.

Un client adresse ses requêtes par un schéma « *read-any / write-any* ». S'il considère que la réponse tarde à venir, il peut changer de copie, ou même adresser sa requête à plusieurs copies en parallèle. Le système garantit une sémantique « au plus une fois » sur les requêtes. Des échanges paresseux de messages (*gossip messages*) entre les copies d'un serveur transportent les requêtes pour mettre à jour l'état des autres copies.

Le principe de base est le suivant. Un identificateur unique est assigné à une opération d'écriture par le serveur qui reçoit cette opération du client. Le client est informé de cet identificateur. Une opération d'écriture ou de lecture prend en paramètre une étiquette, qui est un ensemble d'identificateurs uniques décrivant après quelles opérations cette opération a le droit de s'exécuter. Enfin, une opération de lecture renvoie, en plus de la valeur de retour, une nouvelle étiquette représentant l'état de la copie qui a répondu.

Ainsi, un client, en manipulant les étiquetages, peut contrôler l'ordre des opérations. Pour des questions de représentations compactes des étiquetages, l'identificateur unique, ainsi que l'étiquetage, sont représentés par une estampille temporelle (*timestamp*). L'ordre client le plus faible est alors un ordre similaire à l'ordre causal. Un client sophistiqué peut en fait contrôler les dépendances (notamment enlever les liens causals superflus, ou ajouter des liens causals dûs à des canaux de communications cachés).

Pour l'ordre serveur, une composante supplémentaire est ajoutée à l'estampille temporelle. Cette composante représente la copie du serveur imposant l'ordre total. Néanmoins, pour des raisons de tolérance aux fautes, il n'est pas possible de laisser gérer cette composante à une seule copie du serveur. C'est pourquoi, les auteurs utilisent un mécanisme de serveurs primaire / secondaires avec des changements de vue pour rendre la copie responsable de l'ordre totale tolérantes aux fautes.

Pour l'ordre global, un mécanisme en trois phases (*three phase commit*) (non bloquant) permet de générer cette composante supplémentaire de l'estampillage temporel en accord avec toutes les copies du serveur. Ces dernières ne peuvent plus exécuter des opérations d'écriture durant la première phase du protocole.

Enfin, il est possible qu'une copie du serveur défaille après avoir répondu au client, mais avant d'avoir envoyé un message *gossip*. Si ce client émet des nouvelles opérations, il se peut que celles-ci soient bloquées, car attendant des opérations qui sont perdues (dû à l'absence du *gossip* message). Ce problème, indiqué par les auteurs, ne peut s'affaiblir qu'en envoyant les *gossip* messages plus souvent. Il est aussi possible d'introduire des accusés de réception que la copie du serveur doit attendre avant de répondre au client, mais cela rajoute des contraintes de synchronisation entre les copies.

2.4.3 Psync

Psync [Peterson, 1987] (*pseudosynchronous*) est un système de communication fournissant un diffusion fiable entre des objets. Il ne supporte pas les partitions réseaux. Les processus s'échangent des messages au travers d'une « conversation ». Il utilise le concept de « graphe de contexte », qui est un graphe direct et acyclique. Un nœud représente un message, et un arc la relation « envoyer dans le contexte de ». Ainsi, si un message m_2 est envoyé alors que son émetteur connaît le message m_1, il existe un arc de m_1 vers m_2. S'il n'existe pas d'arc entre m_2 et un autre message, par exemple m_3, m_2 et m_3 sont dit concurrents, car étant envoyés en même temps (relativement à un temps logique).

Chaque participant dans la conversation possède une image du « graphe de contexte ». Si un processus envoie un message m, tous les autres processus participant à la conversation le reçoivent avec des informations supplémentaires permettant à tous les processus de mettre à jour leur « graphe de contexte ». Des primitives utilisateurs permettent de balayer le « graphe de contexte ».

Dans [Mishra et al., 1989], Psync est utilisé pour construire un système de réplication d'objets tolérant les fautes. Les auteurs définissent des groupes d'opérations commutatives pour améliorer les performances du système, en les laissant s'exécuter dans un ordre quelconque sur les différentes copies.

À chaque copie, est associé un gestionnaire qui reçoit les requêtes des clients. Le gestionnaire primaire, celui qui reçoit la requête, diffuse la requête à toutes les copies, y compris lui-même, en utilisant Psync. Il est chargé de répondre à ce client.

Les auteurs définissent une vague (*wave*) comme étant l'ensemble maximum des messages dans lequel il n'existe aucune paire qui vérifie la relation « envoyer dans le contexte de ». Une vague est complète quand plus aucun message à venir ne pourra s'y rajouter. En fait, une vague est complète quand il existe une opération stable, c'est-à-dire connue de toutes les copies. Ainsi, un algorithme de tri déterministe sur la vague détermine l'ordre total requis, puisque chaque participant a la même connaissance de la vague.

Pour affaiblir l'ordre total, ce système utilise la propriété de commutativité entre certaines opérations. C'est-à-dire que l'algorithme de tri déterministe ordonne totalement toutes les opérations non commutatives entre elles, mais permet aux opérations commutatives de s'exécuter dans un ordre différent sur les différentes copies du système. Plus formellement, un groupe d'opérations O est défini comme étant un ensemble de messages du « graphe de contexte » tel que :

- O contient exactement une opération non commutative,
- ou bien O est le plus grand ensemble d'opérations commutatives tel qu'entre toutes les paires d'opérations de O il n'existe pas de chemin dans le « graphe de contexte » passant par une opération non commutative.

Un algorithme plus général est décrit. Il s'applique à un serveur ayant n opérations classifiables en k ensembles disjoints d'opérations commutatives. Par exemple, pour les quatre opérations arithmétiques de base, il s'agit d'un groupe contenant l'addition et la soustraction et d'un autre avec la multiplication et la division. Ainsi cette technique permet de relâcher l'ordre des opérations commutatives entre elles, tout en garantissant un ordre total sur les opérations non commutatives.

2.4.4 Bayou

Bayou [Demers et al., 1994] [Petersen et al., 1996] est un système pour construire des applications collaboratives, hautement disponibles, dans un environnement mobile, en utilisant la réplication et des cohérences variables. Le contexte est extrêmement varié. En plus des traditionnels machines et réseaux, il est constitué de machines portables et d'assistants numériques (ayant des ressources limitées) reliés par des liaisons sans fils, infrarouges ou par modem (déconnections, bande passante faible et irrégulière). Pour cela, les trois principaux critères de Bayou sont les suivants :

- *Disponibilité* : le schéma de réplication « *read-any / write-any* », ainsi qu'une cohérence « à la longue », assurent une disponibilité accrue entre les copies. Le client a le choix entre quatre garanties de session (§ 2.3.3 page 40) pour adapter à son gré le rapport entre la disponibilité et la cohérence. Cette dernière caractéristique permet d'améliorer les performances vis-à-vis du client ;
- *Passage à l'échelle* : le coût d'exécution des opérations, ainsi que la taille requise pour stocker ces opérations sont le plus indépendant possible du nombre de copies. Les opérations de lecture, d'écriture, de création et de destruction de copie ne mettent en jeux que deux machines à la fois. Le seul point où le système est dépendant du nombre de copies est le vecteur de version qu'il utilise ;
- *Adaptable* : il est possible de changer dynamiquement le rôle des serveurs primaires, ainsi que d'ajouter et d'enlever des copies. L'architecture client / serveur ainsi fournie est souple, et permet de s'adapter aux machines possédant peu de ressources.

Pour ménager la bande passante, le système Bayou se base sur la propagation des mises à jour par un protocole d'« anti-entropie » [Petersen et al., 1997]. Celui ci utilise une propagation des opérations d'écriture entre deux machines dans un seul sens. Chaque site choisit quand, avec qui et quelles données propager, s'adaptant ainsi à des topologies réseaux variées, ou des médias transportables (disquettes, cartes PCMCIA, etc). Ce protocole assure une cohérence « à la longue », c'est-à-dire que les opérations d'écriture arriveront, à terme, sur toutes les copies, ainsi qu'une cohérence causale. Il est basé sur des vecteurs de versions et des journaux permettant de garder les opérations d'écriture pour une mise à jour ultérieure d'une autre copie. Ces journaux peuvent être tronqués sur décision unilatérale du propriétaire en cas de consommation excessive de ressource. Une copie ne pouvant plus être mise à jour à cause d'un journal tronqué, est alors mise à jour par transfert de l'état d'une autre copie.

2.4.5 Isis

Le système Isis fournit un outil de réplication passive, supportant le partage de charge, appelé coordinateur / cohortes [Birman et al., 1985].

Chaque client adresse sa requête à un groupe de processus répliqués fournissant le même service. Toutes les copies reçoivent cette requête, mais une seule d'entre elle, le processus coordinateur, l'exécute, propage un point de reprise à toutes les autres copies, les cohortes, et répond au client.

Le partage de charge est fourni par un choix différent du coordinateur à chaque requête. Donc, des requêtes concurrentes ont différents coordinateurs, et donc différentes copies les exécutant. Par rapport à une approche traditionnelle de serveurs primaire / secondaires, où la copie primaire exécute toutes les requêtes, il est clair que la charge est répartie sur plusieurs processeurs, et donc que la performance des applications est augmentée.

Toutes les copies maintiennent la connaissance de toutes les autres copies non fautives. À chaque requête, l'ensemble des copies est trié avec un algorithme déterministe. Cet algorithme peut prendre en compte la provenance de la requête et des informations sur la charge comme paramètre. Quoi qu'il en soit, chaque copie obtient le même ensemble de copies trié dans le même ordre, sans l'utilisation de protocoles d'accord coûteux. La copie de plus haut rang est désignée comme processus coordinateur pour cette requête. Si un coordinateur défaille, la copie suivante dans la liste est élue nouveau processus coordinateur, sans devoir envoyer un seul message. Quand le coordinateur a fini d'exécuter la requête, il effectue une diffusion d'un point de reprise, ainsi que du résultat de la requête, à tous les cohortes par l'intermédiaire d'un CBCAST (diffusion respectant l'ordre causal). Cet ordre causal permet de relâcher les contraintes sur les points de reprise, et donc d'améliorer les performances du service.

Un suspecteur de fautes surveille tous les membres du groupe, et notifie la défaillance probable d'une copie par l'envoi d'un message au groupe avec GBCAST (diffusion respectant un ordre total sur tous les groupes). Ainsi, toutes les copies observent toutes les défaillances et réintégrations des autres copies de manière équivalente. C'est cette propriété, ainsi que le déterminisme de l'algorithme de tri, qui assurent que toutes les copies élisent le même processus coordinateur par rapport à une même requête.

2.4.6 Arjuna

Le système Arjuna [Little, 1991] offre des mécanismes de réplication active d'objets, dans un but de tolérance aux fautes et d'amélioration des performances. L'auteur se base donc sur le modèle de machine à état dont nous rappelons brièvement les deux conditions nécessaires à la consistance des copies :

- C_1 : Toutes les copies non fautives de la machine reçoivent les mêmes requêtes (propriété d'*accord*) ;
- C_2 : Toutes les copies non fautives de la machine exécutent les requêtes dans le même ordre relatif (propriété d'*ordre*).

La condition C_1 est obtenue par l'utilisation d'un protocole de diffusion fiable sans propriété d'ordre. La condition C_2 est garantie au niveau applicatif par l'emploi d'actions atomiques. L'ordre des actions est donc de type « de bout en bout », qui permet d'améliorer les performances en imposant seulement les dépendances nécessaires à l'application.

La technique s'appuie essentiellement sur une base de données (*Group-View Database*) répertoriant le nom de l'objet répliqué, une liste de noms des différentes copies et de leur adresse, une liste de noms d'objets utilisateurs, et une liste de noms de copies exclues.

Un client voulant faire des requêtes sur un groupe d'objets répliqués, demande d'abord la liste des objets disponibles faisant partie de ce groupe (c'est-à-dire la liste des objets de ce groupe privée des objets exclus), et il se rajoute dans la liste des utilisateurs. Il initie la connexion avec les copies, et invoque ses requêtes. Si la défaillance probable d'une copie est détectée, il l'ajoute à sa propre liste de copies exclues. Quand il valide la transaction distribuée, il y a en plus mise à jour atomique de la base de données, c'est-à-dire le retrait de son nom des utilisateurs, ainsi que la fusion de sa liste d'exclusion avec celle gérée dans la base de données. Informellement, la liste des copies disponibles correspond donc à la liste des objets dont on est sûr de la cohérence.

2.4.7 Globe

Globe [van Steen et al., 1997] est une architecture pour la construction d'applications objets à large échelle. Il traite de domaines comme la communication, le nommage, la réplication, la migration, la persistance, la tolérance aux fautes et la sécurité. Cette description sera uniquement basée sur l'architecture de Globe pour comprendre où et comment est gérée la réplication. Pour d'autres aspects, le lecteur intéressé pourra se référer à [Homburg et al., 1995], [van Steen et al., 1995] et [Ballintijn et al., 1997].

Le modèle objet de Globe décrit un objet comme ayant une ou plusieurs interfaces utilisables par d'autres objets. Son état peut être fragmenté ou distribué, mais de façon transparente. C'est à dire que l'état et les opérations qui agissent dessus sont complètement encapsulés. Il existe les objets partagés et distribués, physiquement distribués, intégrant leur politique de réplication et migration. Ces objets sont eux-mêmes constitués de plusieurs objets locaux propres. Un objet local, est un objet résidant dans un seul espace d'adressage. Il peut être composé ou non de plusieurs objets locaux. Par exemple, un objet local peut tout simplement être un objet mandataire (*proxy*).

Chaque objet local formant une partie d'un objet partagé et distribué est composé des objets locaux suivants :

- *Objet sémantique* (*semantic object*) est l'objet qui fournit la sémantique de l'objet, c'est-à-dire qui fournit réellement le service. C'est le seul objet que le développeur doit programmer en fournissant les diverses interfaces et une implantation dans un langage. Les autres objets peuvent être tirés de librairies ;

– *Objet communication* (*communication object*) est responsable des communications entre les différents objets locaux constituant l'objet partagé et distribué. C'est généralement un objet fourni par la machine. Il peut implanter des communications point à point, des diffusions ou même les deux ;
– *Objet réplication* (*replication objet*) doit assurer la cohérence de l'état global de l'objet partagé et distribué. L'état global d'un objet partagé et distribué est constitué d'états de plusieurs objets locaux qu'il faut maintenir cohérents par un protocole particulier dépendant de la sémantique de l'objet ;
– *Objet contrôle* (*control object*) prend en charge les requêtes des clients (emballage, déballage, etc), et contrôle les interactions entre l'objet sémantique et l'objet réplication.

Dans [Kermarrec et al., 1998], il est décrit un cadre de travail pour des objets Web consistants et répliqués en utilisant l'architecture Globe. Ainsi, les objets sont des dépositaires de pages. Il en existe trois catégories, les permanents (équivalent à un serveur Web), les initiés par les objets (équivalent à des serveurs miroirs), et les initiés par les clients (équivalent aux caches Web). Différentes cohérences entre ces trois types d'objets peuvent être utilisées. Il existe des cohérences demandées par les objets (séquentielle, PRAM, causale et « à la longue ») ainsi que des cohérences demandées par les clients (PRAM, causale, « lire ses lectures » et « lectures monotones »). Un ensemble de paramètres permet de configurer l'implantation, comme par exemple l'initiative du transfert — pousser (*push*) ou tirer (*pull*) —, le moment du transfert — immédiate ou paresseuse (dans ce dernier cas, périodique ou autres) —, etc.

Cet exemple montre la grande diversité dans l'implantation d'un service au dessus de Globe, grâce à l'encapsulation dans l'objet de la politique de replication et de migration.

2.4.8 CORBA

Le succès grandissant de CORBA (*Common Object Request Brocker Architecture*) [OMG, 1996] a généré des travaux sur l'intégration de la réplication au sein de cette architecture. Cette section ne détaillera pas les mécanismes de la réplication, qui sont les mêmes que ceux décrit précédemment, mais plus spécialement les méthodes d'intégration.

2.4.8.1 Par intégration

La toute première méthode d'intégration de groupes de communication et de réplication fut par intégration, c'est-à-dire en modifiant l'ORB (*Object Request Brocker*). Par exemple il existe le système Orbix+Isis [II, 1994], basée sur Orbix [ORBIX, 1995] pour le coté ORB et sur Isis [Birman et van Renesse, 1994] pour le coté groupe de communication. Il existe aussi Electra [Maffeis, 1995] qui a une approche plus hiérarchisée permettant d'utiliser aussi bien Isis, Horus [van Renesse et al., 1995] que d'autres systèmes.

L'idée de cette approche est que l'ORB distingue une référence à un objet d'une référence à un groupe. Dans ce dernier cas, il transmet la requête à un système de communication de groupe. Du côté serveur, l'invocation est de nouveau transformée en un appel standard sur chaque membre du groupe, et les réponses sont acheminées par le système de communication de groupe jusqu'au site du client, qui les rassemble, et en retourne une seule. Ce service est transparent au client. Un client plus sophistiqué peut, au moyen d'une IDL (*Interface Definition Language*) étendue, demander l'ensemble des réponses (par exemple comme l'extension de l'IDL proposée dans Cobra [Beaugendre et al., 1997]).

Le principal avantage de cette méthode, est la transparence des clients vis-à-vis de la réplication puisqu'ils manipulent une référence qu'ils croient correspondre à un seul objet. Il est donc très simple de réutiliser des clients. Mais le modèle objet de CORBA spécifie explicitement qu'une référence doit correspondre à un seul objet, et qu'elle doit désigner le même objet à chaque utilisation. L'approche par intégration viole donc la spécification CORBA. De plus, cette approche est dépendante de l'ORB utilisé, c'est-à-dire non interopérable. Par exemple, un client d'Orbix+Isis ne pourra pas communiquer avec un groupe de serveurs d'Electra et vice et versa. Malgré tout, cette solution est la plus efficace, car implantée directement par les ORB.

2.4.8.2 Par service

Une autre possibilité est de fournir ces techniques comme un service CORBA [OMG, 1997]. Il n'existe pas pour l'instant de normalisation de tels services. Il existe tout de même des travaux sur des services de communication de groupes (par exemple [Felber et al., 1996]), ou des services de réplication d'objets (par exemple [Sousa et Oliveira, 1997]).

Le service présente alors une interface, et le client doit l'utiliser de façon explicite. Le service est alors libre d'utiliser un système de communication spécifique, ou bien les primitives de communications normalisées par CORBA (beaucoup plus interopérable).

Avec cette approche, la portabilité et l'interopérabilité sont maximisées. Le client référence explicitement un objet de service, mais ce dernier est indépendamment implanté. Il est possible de normaliser l'interface d'un service de communication de groupe ou de replication d'objets. Si le service utilise les primitives CORBA de communication, un service de réplication peut coopérer avec un autre service de réplication issu d'un constructeur différent. Dans tous les cas, le service n'est pas associé à une mise en œuvre particulière de CORBA. Par contre, le client doit utiliser explicitement le service, ce n'est donc pas transparent pour lui. L'utilisation de mandataires intelligents (*smart proxy*) [Felber et al., 1997], permet d'assurer cette transparence, mais au détriment de la portabilité. Enfin, cette solution est moins efficace, car la requête passe du client à l'ORB local, puis au service, de nouveau sur l'ORB local pour être propagée à un objet service distant, qui va faire suivre la requête au serveur par l'intermédiaire de l'ORB distant.

2.4.8.3 Par interception

Enfin, la dernière méthode permet d'intercepter les requêtes IIOP (*Internet Inter-Object Protocol*) d'un ORB quelconque [Narasimhan et al., 1997b]. Les requêtes IIOP [Minton, 1997] sont une standardisation des messages échangés entre les ORB pour en assurer l'interopérabilité.

Dans [Narasimhan et al., 1997a], les auteurs détaillent l'architecture. Ils interceptent les communications entre l'ORB et le système de communication (*socket*) du système d'exploitation. Ils font suivre cette requête à un gestionnaire spécifique, et un système de communication de groupes. Côté serveur, le chemin inverse est effectué, c'est-à-dire la communication de groupes, le gestionnaire, l'intercepteur puis l'ORB.

L'intérêt de cette méthode est à la fois la transparence vis-à-vis des clients et de l'ORB, et de bonnes performances. De plus, le service peut être associé à n'importe quelle implantation d'un ORB respectant IIOP, ou même avoir différentes implantations coté client et serveur.

2.5 Conclusion

Le concepteur d'un service de réplication a devant lui un nombre important de choix à faire, conditionnant entre autres la performance, la tolérance aux fautes, la transparence ainsi que la difficulté de programmation. On peut regrouper ces choix en deux grandes catégories :

1. Le type de la réplication, c'est-à-dire active, passive ou semi-active, qui influe sur les performances et la tolérance aux fautes ;
2. Les modèles de cohérence supportés, qui ont principalement des répercussions sur la performance, la transparence, la difficulté de programmation des serveurs et/ou des clients ;

De nombreuses solutions ont été proposées au travers de systèmes divers, chacun ayant fait ses propres choix, le plus souvent d'une façon définitive. Certains sont orientés vers une catégorie d'applications précise, mais peu d'entre eux fournissent des mécanismes adaptables au besoin des serveurs répliqués et aux clients qui les utilisent. D'où, finalement, le nombre et la variété des systèmes disponibles.

À notre connaissance, aucun système de réplication pour améliorer les performances ne regroupe les différentes techniques utilisées, à savoir aussi bien le support de la cohérence « de bout en bout » que des modèles de cohérence générique des plus faible au plus forte, l'utilisation d'un schéma « *read-any / write-any* », la séparation de la politique de propagation des mises à jour et les requêtes parallèles à différentes copies.

Chapitre 3

Estampilles d'Ordonnancement : principes et utilisations

Le modèle de cohérence qui garantit la cohérence de l'état des copies d'un serveur est un paramètre important dans un système de réplication d'objets ayant comme objectif l'amélioration de la performance des serveurs. Un tel système doit permettre aussi bien l'utilisation de modèles de cohérence variés, des plus fortes aux plus faibles, que l'utilisation des modèles « de bout en bout ».

Le protocole des estampilles d'ordonnancement permet à chaque invocation de transporter avec elle toutes les informations nécessaires pour déterminer l'ordre de sa livraison au serveur destinataire. Ainsi, l'ordre de livraison peut être déterminé par un gestionnaire séparé, facilement interchangeable suivant le modèle de cohérence souhaitée.

Dans ce chapitre, nous donnons d'abord quelques définitions générales sur les objets (§ 3.1). Nous expliquons ensuite le principe de base des estampilles d'ordonnancement (§ 3.2), et nous examinons les optimisations nécessaires à ce protocole (§ 3.3) pour qu'il soit performant. Puis, nous récapitulons l'intégralité des principes en un seul et unique, et nous en examinons les propriétés de vivacité et de sûreté (§ 3.4). Enfin, nous examinons brièvement quelques travaux en relation avec les nôtres (§ 3.5).

3.1 Définitions et terminologies

Dans cette section, quelques définitions supplémentaires sur les objets (§ 3.1.1) et sur les invocations de méthodes (§ 3.1.2) sont données.

3.1.1 Objets

Nous avons déjà introduit le vocabulaire relatif aux objets au § 1.4.3 page 14. Nous complétons ces définitions avec la *copie d'un objet*, qui est une autre instance de la classe de cet objet mais qui représente, par son état, le même objet. Ils doivent donc avoir un état cohérent entre eux.

Généralement, l'*état* d'un objet représente l'ensemble des variables définies par sa structure de données, comme dans le cas de la machine à état (§ 2.1.2 page 20). Cette définition trop stricte ne nous convient pas car ne faisant pas intervenir l'état des serveurs invoqués par cet objet. Nous définissons l'état d'un objet comme les données qu'il peut fournir à ses clients, c'est-à-dire la vue qu'il leur présente. C'est pour nous une notion que seule la classe peut définir, pouvant pour cela faire intervenir un sous-ensemble de sa structure de données ou même l'état d'autres objets qu'elle utilise. Par exemple, avec cette définition, l'état d'un objet mandataire est identique à l'état de l'objet qu'il représente.

3.1.2 Invocations

Une invocation d'une méthode d'un objet représente l'appel de la méthode. Cette invocation peut être transmise par le réseau, arrivée sur un objet, délivrée à un objet, et par abus de langage, exécutée par un objet (en fait, c'est la méthode invoquée qui est exécutée).

Nous définissons les invocations qui modifient ou non l'état d'un objet ainsi :

Définition 3.1 (Type d'une invocation)
Par abus de langage relatif aux systèmes de gestion de fichiers, une invocation d'une méthode sur un objet peut être de type :

- *lecture* si elle provoque l'exécution d'une méthode qui ne modifie pas l'état de l'objet,
- *écriture* si elle provoque l'exécution d'une méthode qui modifie l'état précédent de l'objet.

Notation 3.1
Nous notons une invocation en lecture par L et en écriture par E. Une invocation de type quelconque (lecture ou écriture) est notée I.

Par la suite, nous différencons deux invocations de même type par l'ajout d'un exposant. Ainsi, L, $L^{'}$ et L^1 représentent trois invocations en lecture différentes.

3.2 Estampilles d'ordonnancement

Cette section décrit le principe général des estampilles d'ordonnancement. Nous commençons par l'introduction d'une relation d'ordre strict (§ 3.2.1) représentant une relation de cohérence. Nous décrivons comment représenter cette relation avec un graphe de dépendance (§ 3.2.2). Nous exposons alors le principe des estampilles d'ordonnancement (§ 3.2.3) et de la liste d'estampilles (§ 3.2.4) utilisées pour représenter ce graphe. Nous ferons une première transformation du graphe en graphe de dépendance direct (§ 3.2.5) pour faciliter la description d'exemples d'utilisations (§ 3.2.6) de ces estampilles d'ordonnancement.

3.2.1 Relation d'ordre strict

Généralement, une façon de représenter une relation de cohérence particulière est de la considérer comme une relation binaire sur les invocations. Nous appelons cette relation binaire $\rightharpoonup$ « dépendant de ». Ainsi, une invocation I qui doit être traitée après l'exécution d'une invocation $I^{'}$, est dépendante de $I^{'}$. Nous pouvons donc représenter cette dépendance par : $I \rightharpoonup I^{'}$.

Nous rappelons quelques propriétés des relations binaires sur un domaine E :

$$\begin{aligned} \textit{Irréflexive} : &\quad \text{si } x \in E,\ x \not\rightharpoonup x\,; \\ \textit{Transitive} : &\quad \text{si } x, y, z \in E,\ x \rightharpoonup y,\ y \rightharpoonup z \ \Rightarrow\ x \rightharpoonup z\,; \\ \textit{Anti-symétrique} : &\quad \text{si } x, y \in E,\ x \rightharpoonup y,\ y \rightharpoonup x \ \Rightarrow\ x = y. \end{aligned}$$

Il est clair que la relation $\rightharpoonup$ est irréflexive, transitive et anti-symétrique. Ces trois propriétés confèrent à cette relation binaire les caractéristiques d'un ordre strict.

Une relation de cohérence peut donc être considérée comme un ordre strict. Ce dernier peut être total ou partiel suivant les caractéristiques de la relation de cohérence. Contrairement à un ordre partiel, un ordre total sur le domaine E vérifie la propriété suivante :

$$\forall (x,y) \in E \times E : \; x \rightharpoonup y \text{ ou } y \rightharpoonup x$$

Ainsi, il existe dans un ordre partiel des éléments non comparables. En considérant l'ordre strict « dépendant de », cela signifie qu'il existe des invocations non comparables donc indépendantes. Nous les appelons des invocations concurrentes, qui peuvent donc être exécutées dans n'importe quel ordre les unes par rapport aux autres.

3.2.2 Graphe de dépendance

Un ordre peut être représenté par un graphe orienté $G = (X, U)$, où X est l'ensemble des nœuds du graphe, et U l'ensemble des couples ordonnés $(x, y) \in X \times X$ appelés arcs orientés. Ainsi, le graphe $G = (X, U)$ associé à la relation d'ordre strict $\rightharpoonup$ sur le domaine E est définie par :

$$x, y \in E, \; x \rightharpoonup y \; \Leftrightarrow (x, y) \in U$$

Dans le cas de la relation d'ordre strict « dépendant de », nous appelons ce graphe le graphe de dépendance. Il hérite de la relation d'ordre strict les propriétés d'irréflexivité, de transitivité et d'anti-symétrie. Ces trois propriétés confèrent au graphe la particularité d'être sans circuit [Berge, 1963, chap. 2].

Pour résumer, une relation de cohérence peut être considérée généralement comme une relation d'ordre strict partiel ou total sur un ensemble d'invocations (relation « dépendant de »). Cette relation est représentable par un graphe orienté sans circuit (graphe de dépendance).

3.2.3 Estampilles d'ordonnancement

3.2.3.1 Principe

Nous proposons de définir un mécanisme pour représenter cette relation d'ordre strict « dépendant de » de manière performante. Pour cela, il ne doit pas fournir une cohérence plus forte que celle voulue, et il doit être le plus simple possible pour une implantation efficace. Nous représentons le graphe de dépendance de la relation d'ordre strict voulue avec les estampilles d'ordonnancement.

Une estampille d'ordonnancement est le moyen de représenter un arc du graphe de dépendance. Tout d'abord, chaque invocation possède un identificateur i, et éventuellement une estampille d'ordonnancement.

Définition 3.2 (Estampille d'ordonnancement)
Une estampille d'ordonnancement est un identificateur d'une invocation.

Notation 3.2
Nous notons une invocation quelconque d'identificateur i et d'estampille e par $I_i\langle e\rangle$ (Nous omettons, quand l'information est inutile pour la compréhension, l'identificateur et/ou l'estampille).

Nous différencons deux identificateurs ou estampilles par l'ajout d'exposants ou d'indices. Ainsi, i', i_x et i_x^y représentent trois identificateurs différents.

Une estampille est un identificateur d'une autre invocation. Une invocation *estampillée* par x fait référence à une invocation d'*identificateur* x. Ainsi, s'il existe dans le graphe de dépendance l'arc $I_i \rightharpoonup I'_{i'}$ alors I_i est estampillée par i' et elle est notée $I_i\langle i' \rangle$. Enfin, s'il n'existe aucun arc partant de I_i, alors I_i n'est pas estampillée et notée $I_i\langle\rangle$.

Venons en alors au principe régissant ces estampilles d'ordonnancement. Une invocation $I'_{i'}$ estampillée par e, est « dépendante de » l'invocation I_e. $I'_{i'}$ doit donc être délivrée à l'objet après que I_e ait été complètement exécutée sur celui-ci. Plus formellement, nous posons le premier principe spécifiant l'ordonnancement des invocations sur un objet. Par la suite, il est complété ou remplacé par d'autres principes.

Principe 3.1 (Estampille d'ordonnancement simple)
Quand une invocation I estampillée par e arrive sur un objet o :

- Attendre que I'_e ait été exécutée ;
- Délivrer I à o.

3.2.3.2 Représentation du graphe

Ainsi, une partie du graphe de dépendance est représentée à l'aide des estampilles d'ordonnancement. Nous avons résumé cette représentation relativement à une invocation de référence I dans la figure 3.1 page ci-contre.

3.2.4 Liste d'estampilles

Il est évident que cette méthode ne peut pas représenter un graphe de dépendance quelconque. En effet, nous voyons sur la figure 3.1 page suivante, que le nœud I ne peut avoir qu'un seul arc sortant, et donc qu'une invocation ne peut dépendre que d'une seule autre invocation.

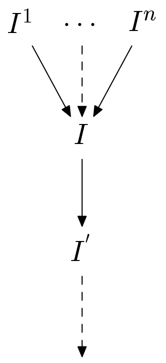
3.2.4.1 Principe

Pour pallier ce problème, nous introduisons la possibilité d'estampiller une invocation avec une *liste* d'estampilles d'ordonnancement, et non plus une seule.

Définition 3.3 (Liste d'estampilles d'ordonnancement)
Une liste d'estampilles d'ordonnancement est une liste d'identificateurs d'invocations.

Notation 3.3
Nous notons une invocation quelconque d'identificateur i et comportant une liste d'estampilles de e_1 à e_n par $I_i\langle e_1, e_2, \ldots, e_n \rangle$.

Le principe est alors similaire, mais une invocation I estampillée par une liste d'estampilles est « dépendante de » *toutes* les invocations dont l'identificateur est présent dans sa liste. Donc, I doit être délivrée à l'objet après que *toutes* ces invocations aient été complètement exécutées par l'objet. Nous pouvons formuler ce principe ainsi :



(a) Graphe

- $I^1\langle i\rangle, \dots, I^n\langle i\rangle$;
- $I_i\langle j\rangle$;
- $I^{'}_j\langle \dots\rangle$.

(b) Estampille

FIG. 3.1 – Graphe de dépendance et estampilles

Principe 3.2 (Liste d'estampilles d'ordonnancement)
Quand une invocation I estampillée par e_1 à e_n arrive sur un objet o :

- Attendre que les invocations d'identificateur e_j aient été exécutées (avec $1 \leq j \leq n$) ;
- Délivrer I à o.

3.2.4.2 Représentation du graphe

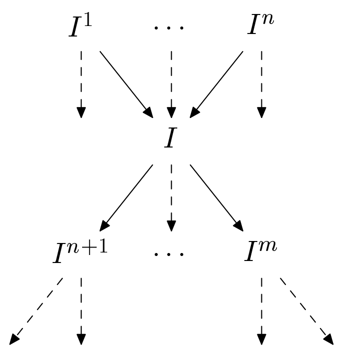
La figure 3.2 page suivante complète alors les différentes possibilités de représentation du graphe de dépendance avec les estampilles et les listes. Maintenant, nous pouvons représenter n'importe quel graphe avec ce mécanisme, et nous pouvons ordonnancer les invocations en respectant le graphe de dépendance associé à l'ordre strict $\rightharpoonup$.

3.2.5 Graphe de dépendance direct

Il est clair qu'il est redondant de véhiculer et traiter les estampilles d'ordonnancement nécessaires à représenter la transitivité de la relation d'ordre strict. En effet, si nous avons $I \rightharpoonup I^{'} \rightharpoonup I^{''}$, d'après le principe 3.2 page précédente, I est exécutée seulement quand $I^{'}$ l'aura été. De même, $I^{'}$ est exécutée seulement quand $I^{''}$ l'aura été. Il est donc clair que I est exécutée quand $I^{''}$ l'aura été, et cela sans faire intervenir $I \rightharpoonup I^{''}$ dûe à la transitivité. Le principe des estampilles d'ordonnancement respectant intrinsèquement la propriété de transitivité, nous pouvons sans problème supprimer ces arcs du graphe de dépendance.

Pour cela, nous devons utiliser la fermeture transitive τ d'un graphe. Nous rappelons que si $G = (X, U)$ alors $\tau(G) = G(X, \tau(U))$ et que $(x, y) \in \tau(U)$ si et seulement si il existe un chemin allant de x à y dans G (ou plus simplement, on peut, en suivant un chemin de G, aller de x à y). Cette fermeture transitive définit une classe d'équivalence, appelée τ-équivalente, telle que deux graphes G et G' sont dans cette classe si et seulement si $\tau(G) = \tau(G')$. Concrètement, G et G' possèdent les mêmes possibilités de cheminement dans le graphe.

Il existe dans cette classe un ensemble de graphes, appelé τ-minimaux. Ces graphes ont la propriété de ne plus faire partie de leur classe τ-équivalente si on leur retire un arc. Donc, les



(a) Graphe

- $I^1\langle ...,i,...\rangle$, ..., $I^n\langle ...,i,...\rangle$;
- $I_i\langle j_{n+1},...,j_m\rangle$;
- $I^{n+1}_{j_{n+1}}\langle ...\rangle$, ..., $I^m_{j_m}\langle ...\rangle$.

(b) Estampille

FIG. 3.2 – Graphe de dépendance et listes d'estampilles

graphes appartenant à l'ensemble τ-minimaux, sont des graphes dont les arcs engendrés par la transitivité de la relation d'ordre n'existent plus. Or, si le graphe considéré est sans circuit, il existe un seul graphe τ-minimal, qui lui soit τ-équivalent et ce graphe est un graphe partiel du graphe considéré [Roy, 1969, § IV.B.2]. Ce graphe τ-minimal est obtenu par la relation de couverture associée à la relation d'ordre.

Or, notre graphe de dépendance est sans circuit, donc il existe un seul graphe τ-minimal, que nous appelons le graphe de dépendance direct, qui offre les mêmes possibilités de cheminement que le graphe de dépendance, mais dont les arcs dûs à la transitivité de la relation d'ordre strict ont été supprimés. Ce graphe étant un graphe partiel du graphe de dépendance, il est clair qu'il est sans circuit et conserve les propriétés d'anti-symétrie et d'irréflexivité. Nous donnons, dans la figure 3.3 page ci-contre, un exemple de différents types de graphes que nous venons de décrire.

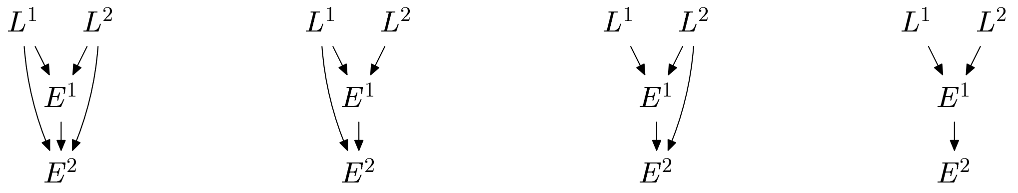
Nous avons donc un graphe de dépendance direct, sans aucun arc redondant, et utilisable en lieu et place du graphe de dépendance, sans modification de l'ordonnancement des invocations et sans surcoût dû à la transitivité. Il est alors clair que pour des raisons de performance, les estampilles d'ordonnancement générées doivent représenter ce graphe de dépendance direct.

3.2.6 Utilisations

Dans cette section, nous montrons comment les clients et les serveurs peuvent, grâce aux estampilles d'ordonnancement, maintenir une cohérence voulue entre les diverses copies d'un serveur. Ces explications sont des exemples. D'autres moyens d'utiliser ces estampilles sont possibles, y compris pour garantir les mêmes cohérences que celles présentées ici.

3.2.6.1 Notation

Dans les figures suivantes, nous représentons les clients par C et les serveurs par S. L'éventuel indice S_x (resp. C_x) représente une copie d'un serveur (resp. un client). L'identificateur (resp. estampille) spécial $\sim$ est associé à une invocation qui n'a pas encore reçu d'identificateur (resp. d'estampille). Les retours des invocations pourront véhiculer une estampille (donc un



(a) G (b) G' (c) G'' (d) G'''

Nous décrivons quelques définitions et propriétés sur ces exemples :

- G est le graphe de dépendance associé à une relation d'ordre strict ;
- $G''' \subset G' \subset G$, $G''' \subset G'' \subset G$;
- G, G', G'' et G''' sont dans la même classe τ-équivalente ;
- G''' est l'élément τ-minimal de cette classe, il correspond donc au graphe de dépendance direct.

Fig. 3.3 – Exemple de graphe de dépendance

identificateur d'invocation) notée tout simplement i. La figure 3.4 page suivante fournit les différentes représentations graphiques des messages.

3.2.6.2 Architecture

Dans cette section, nous ne discutons pas des différentes implantations possibles sur une plate-forme objet. Néanmoins, pour une compréhension accrue, nous décrivons sommairement l'architecture utilisée pour présenter ces exemples d'utilisation. Nous détaillerons un cadre de travail plus complet et plus réaliste au chapitre 4 page 85.

Pour invoquer une méthode sur un serveur, un client invoque la méthode d'une de ses copies. Cette copie n'est pas fixe tout au long de la vie du client. Ce dernier peut décider d'en changer à tout moment, par exemple parce que les temps de réponse ne lui conviennent pas, ou parce que la copie n'existe plus.

La réception de cette invocation se fait par un gestionnaire d'estampilles et non directement par le serveur. Son rôle est :

- d'identifier et d'estampiller cette invocation en fonction de sa politique de cohérence et de synchronisation,
- de l'introduire dans la queue d'exécution locale,
- et de la transmettre au gestionnaire de propagation pour une éventuelle diffusion.

Les invocations en provenance d'une autre copie du serveur, et donc d'un gestionnaire de propagation, sont directement introduites dans la queue d'exécution sans passer par le gestionnaire local, puisqu'elles sont déjà identifiées et estampillées.

Le gestionnaire de propagation est chargé de diffuser les invocations, partiellement ou totalement, aux autres copies. Il fournit donc, suivant sa politique, une propagation à son initiative ou sur demande, immédiate ou périodique, etc.

$\xrightarrow{I_i\langle e_1,\ldots,e_n\rangle}$ Invocation $I_i\langle e_1,\ldots,e_n\rangle$ délivrée au gestionnaire d'estampilles,

$\xrightarrow{I_i\langle e_1,\ldots,e_n\rangle}$ Invocation $I_i\langle e_1,\ldots,e_n\rangle$ délivrée à l'application,

$\xrightarrow{I_i\langle e_1,\ldots,e_n\rangle}$ Retard de l'invocation $I_i\langle e_1,\ldots,e_n\rangle$ pour ordonnancement,

$\xrightarrow{I_i\langle e_1,\ldots,e_n\rangle}$ Invocation $I_i\langle e_1,\ldots,e_n\rangle$ ignorée,

$\xrightarrow{i}$ Retour de l'invocation (véhiculant l'éventuelle estampille i),

$\xrightarrow{i}$ Message entre copies (véhiculant l'éventuelle estampille i),

1 2 ... Repères se rapportant aux explications.

FIG. 3.4 – Convention graphique utilisée

Enfin, un gestionnaire d'ordonnancement se charge de délivrer les invocations de la queue d'exécution à l'application. C'est donc lui qui respecte les différents principes relatifs aux estampilles.

Le gestionnaire d'ordonnancement est local aux différentes copies du serveur, mais le gestionnaire d'estampilles peut être situé sur le site du client dans un mandataire (*proxy* [Shapiro, 1986]) du serveur. Ceci est un détail d'implantation du gestionnaire, il est de la responsabilité du concepteur du gestionnaire d'optimiser les communications entre le(s) mandataire(s) et le serveur.

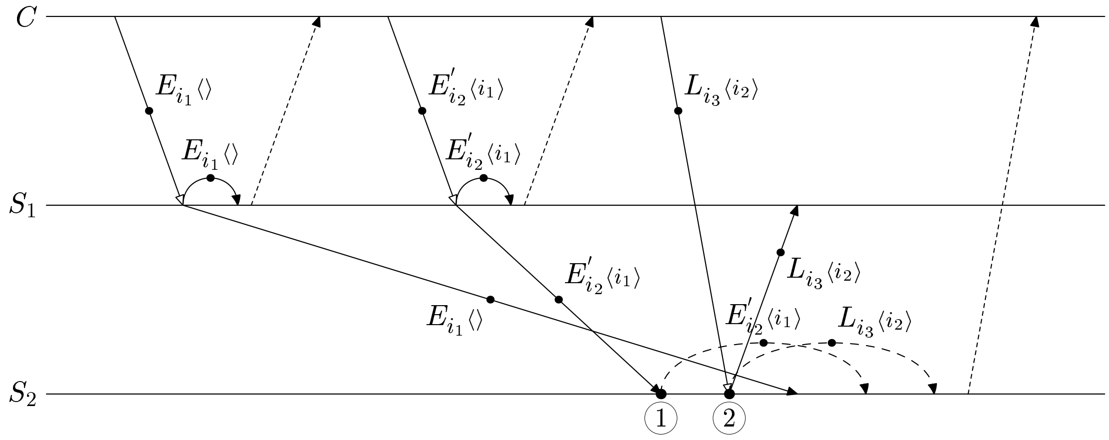
Dans un premier temps, nous diffusons les invocations en lecture à toutes les copies du serveur, afin de simplifier les exemples. Nous verrons au § 3.3.1 page 62 les optimisations possibles pour éviter cette diffusion.

3.2.6.3 Maintien de l'ordre programme

Nous considérons le cas d'un seul client qui attend que l'ordre programme de ses invocations soit respecté sur un même serveur répliqué. C'est un comportement identique à la garantie de session « lire ses écritures » décrite § 2.3.3.1 page 42. L'objectif est de garantir cet ordre, même si le client utilise une copie différente en cours d'exécution. On peut alors voir le scénario de la figure 3.5 page suivante.

3.2.6.4 Maintien de la cohérence atomique objet

Nous présentons brièvement une méthode de maintien de la cohérence atomique. Nous utilisons des communications d'estampilles et de droits d'estampiller entre les copies d'un serveur. Nous avons adapté des algorithmes utilisés dans [Li et Hudak, 1989] pour la mémoire virtuellement partagée dont nous avons parlé au § 2.3.1.1 page 27. Le principe général est alors : à un instant donné, plusieurs gestionnaires peuvent estampiller des invocations en lecture, ou un seul d'entre eux peut estampiller des invocations en écriture.



Le client C invoque une écriture E. Le mandataire lui assigne un identificateur i_1 et la transmet à S_1. Ce dernier va la diffuser à l'autre copie S_2, puis l'exécuter et renvoyer le résultat à C. Pour l'écriture suivante E', le mandataire, en plus de lui assigner un identificateur i_2, l'estampille avec l'identificateur de la précédente invocation, c'est-à-dire i_1. Et ainsi de suite.
Pour l'invocation en lecture L, C s'adresse à S_2. Pour conserver l'ordre programme des exécutions, nous avons aux repères :

① le retard de l'exécution de E'_{i_2} car estampillée par i_1, elle ne peut pas être exécutée avant E_{i_1} (elle attendra donc son exécution),

② le retard de l'exécution de L car estampillée par i_2, elle ne peut pas être exécutée avant E'_{i_2}.

Fig. 3.5 – Ordre programme avec les estampilles

Initialement, un seul gestionnaire possède le droit d'estampiller une invocation en écriture, ou bien plusieurs possèdent le droit d'en estampiller en lecture. Si un client veut émettre une invocation en écriture, le gestionnaire d'estampilles demande le droit d'estampiller cette écriture au(x) serveur(s) le possédant actuellement, et leur retire. Ces derniers communiquent les identificateurs des invocations qu'ils ont émises. Cela permet d'estampiller l'invocation en écriture avec cette liste. Le gestionnaire a alors la garantie que son invocation en écriture est exécutée immédiatement après les invocations précédentes par rapport à l'ordre atomique, et qu'il est le seul à pouvoir le faire.

De même, si un client veut émettre une invocation en lecture, le gestionnaire d'estampilles demande le droit d'estampiller cette lecture au(x) serveur(s) le possédant actuellement, mais en leur retirant les droits d'estampiller en écriture.

La figure 3.6 page suivante montre un exemple de scénario possible avec un tel algorithme.

3.3 Optimisations

Nous avons fait une première optimisation au § 3.2.5 page 57, en introduisant le graphe de dépendance direct. Nous décrivons dans cette section d'autres optimisations essentielles à l'efficacité des estampilles d'ordonnancement. Ainsi nous présentons les estampilles surchargées (§ 3.3.1) pour limiter la diffusion des invocations en lecture. Puis, nous décrivons les groupes d'invocations (§ 3.3.2) pour alléger la gestion de certains schémas de relation courants. Enfin, nous terminons en présentant des exemples d'utilisation (§ 3.3.3).

3.3.1 Diffusion des lectures

3.3.1.1 Problème

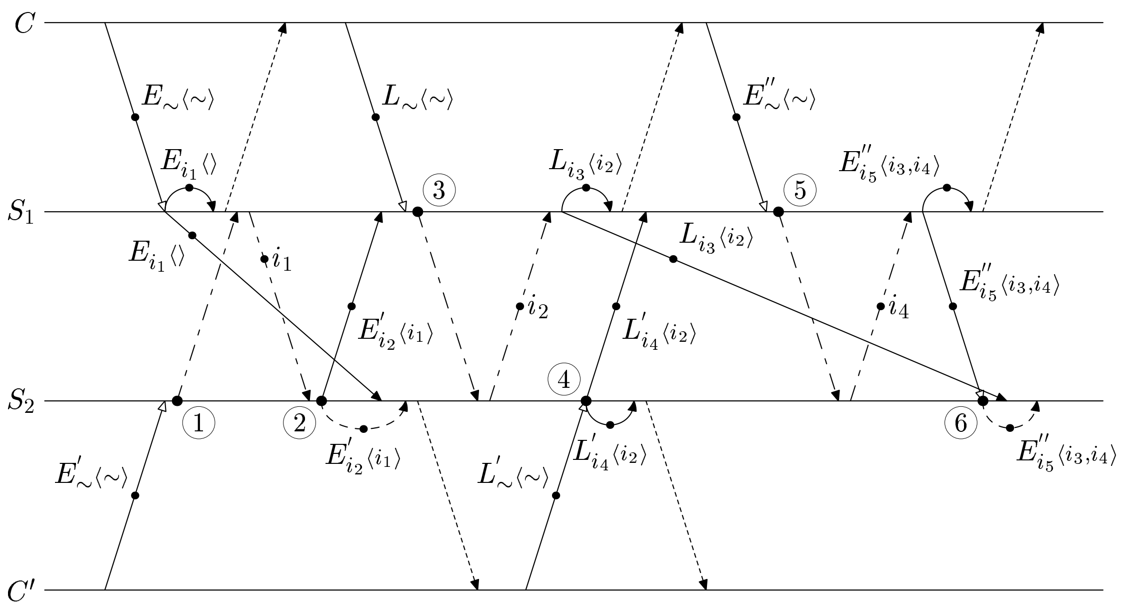
Des définitions du paragraphe § 3.1.2 page 54, nous pouvons déduire le rôle des différents types d'invocations :

- *Lecture* : elle informe le client de l'état ou d'un sous-ensemble de l'état d'un objet. Elle peut donc être délivrée à une seule copie de l'objet ;
- *Écriture* : elle modifie l'état de l'objet et elle informe éventuellement le client de son état ou d'un sous-ensemble. Elle doit donc être délivrée, à terme, à toutes les copies de l'objet présentes dans le système.

Or, jusqu'à maintenant, nous diffusons les invocations en lecture sur toutes les copies du serveur. Mais, une invocation en lecture ne modifie pas l'état du serveur, il est donc inutile, dans la plupart des cas, de la diffuser.

Pourtant, la diffusion des invocations en lecture peut permettre d'améliorer les performances ou la disponibilité. Si nous voulons recevoir le résultat d'une invocation en lecture le plus rapidement possible, nous la diffusons à toutes les copies, et nous utilisons le premier résultat, et ignorons les autres. Le mécanisme est similaire pour assurer la disponibilité d'un serveur répliqué. Néanmoins, même si nous voulons de la performance ou de la disponibilité, il n'est pas toujours imaginable de diffuser la lecture à toutes les copies du système comme, par exemple, dans un réseau à large échelle (*Man* ou *Wan*). Il est beaucoup plus sensé de diffuser partiellement (*multicast*) les lectures, par exemple, sur toutes les copies du réseau local (*Lan*).

Nous devons donc considérer qu'une invocation en lecture n'est généralement pas diffusée à toutes les copies. Elle peut être acheminée vers une seule copie du serveur, ou à un sous



C émet une invocation en écriture E sur S_1, qui lui assigne un identificateur i_1, la diffuse à l'autre copie S_2, l'exécute, et renvoie le résultat. De même, C' envoie une invocation en écriture $E^{'}$ sur S_2, qui lui assigne un identificateur i_2. Nous avons alors aux différents repères :

① S_2 demande le droit d'estampiller une écriture à S_1. S_1 lui renvoie l'identificateur de la dernière écriture : i_1. S_2 estampille alors $E^{'}$ avec cette valeur ;

② S_2 l'envoie à l'autre copie S_1, et retarde l'exécution locale jusqu'à ce que l'écriture d'identificateur i_1 arrive sur et soit exécutée par S_2 ;

③ C invoque une lecture L sur S_1. S_1 demande à S_2 le droit d'estampiller des lectures. S_2 lui renvoie l'identificateur de la dernière écriture : i_2. S_1 estampille la lecture L avec i_2 ;

④ C' invoque une lecture $L^{'}$ sur S_2. Ce dernier pourra alors estampiller directement la lecture avec i_2 puisqu'il possède toujours ce droit ;

⑤ C émet une invocation en écriture $E^{''}$ sur S_1. Il demande alors à S_2 le droit d'estampiller une écriture. S_2 lui renvoie la liste des identificateurs des dernières lectures qu'il a émises. S_1 fusionne cette liste avec la sienne, et les utilise pour estampiller $E^{''}$;

⑥ $E^{''}$ est retardée jusqu'à ce que la lecture L d'identificateur i_3 soit exécutée par S_2, la lecture $L^{'}$ d'identificateur i_4 ayant déjà été exécutée.

FIG. 3.6 – Cohérence atomique objet avec les listes d'estampilles

ensemble. Or, toutes les autres invocations qui sont estampillées avec l'identificateur de cette lecture, et qui arrivent sur les autres copies du serveur, vont attendre indéfiniment cette dernière pour être délivrées à l'application. Nous proposons un scénario de ce type à la figure 3.7 page ci-contre.

Une solution naïve consiste à demander aux gestionnaires d'estampilles, quand ils traitent une dépendance d'une invocation en écriture E sur une invocation en lecture L, d'estampiller E avec l'*identificateur* de L pour la copie locale, et d'estampiller E avec les *estampilles* de L pour les autres copies.

Regardons l'exemple de la figure 3.8 page 66. Il présente un scénario où une autre copie que S_1 répond à la lecture du client. Notre précédente solution nous contraindrait à envoyer l'invocation en écriture deux fois, une sur S_1 et S_2 avec comme estampille l'identificateur de L_{i_1} (donc i_1), et une sur S_3 sans estampille (puisque L_{i_1} n'en possède pas). Nous pouvons facilement trouver des configurations où cette double propagation implique une double transmission sur le même réseau physique. Cette solution n'est donc pas très intéressante.

De plus, il est attractif d'autoriser le client à réémettre son invocation en cas de délai trop longs. Ainsi, la requête peut se faire en deux temps, un premier où le client invoque normalement une lecture, et un second, après un certain temps, où le client réinvoque sa lecture en demandant une diffusion (restreinte ou non). La solution naïve devient alors complètement inadaptée.

3.3.1.2 Estampilles surchargées

Ce problème peut en fait être décomposé en deux sous-problèmes. D'une part, comment une invocation I qui dépend d'une autre invocation en lecture L peut-elle éviter d'attendre cette dernière indéfiniment ? D'autre part, comment I peut elle respecter les dépendances de L, puisque, par transitivité, elle dépend de toutes les invocations dont dépend L ?

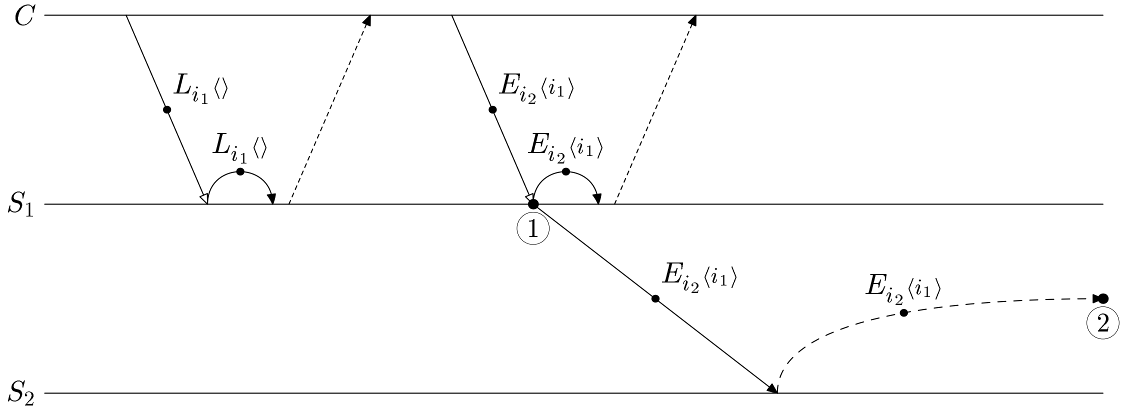
Commençons par ce deuxième problème. En fait, ce sont les dépendances dûes à la transitivité qui nous font défaut. Il est nécessaire que ces dépendances arrivent sur tous les sites. L'idée est alors de diffuser, dans tous les cas, les dépendances de L mais sans l'invocation elle-même, c'est-à-dire sans la fonction et les paramètres associés, puisqu'elle ne sera pas exécutée sur tous les sites.

Néanmoins, pour que cette diffusion ne soit pas coûteuse, il est possible de la véhiculer avec les invocations qui seront estampillées avec l'identificateur de cette invocation en lecture. L'avantage est double. D'abord cela ne nécessite pas de message particulier, coûteux, surtout avec des diffusions. Ensuite, cette information ne sera diffusée que si une invocation dépend de cette invocation en lecture. Nous appelons ces estampilles, des *estampilles surchargées*.

Définition 3.4 (Estampille d'ordonnancement surchargée)
Une invocation en écriture possédant une estampille d'ordonnancement i référençant une invocation en lecture L_i, peut posséder d'autres estampilles, appelées estampilles surchargées, de mêmes valeurs que les estampilles de L_i, et rattachées à i.

Notation 3.4
Nous notons une invocation en écriture E d'identificateur i, ayant une estampille e, et des estampilles surchargées e_1 à e_n rattachées à e, par $E_{i\langle e\langle e_1,\ldots,e_n\rangle\rangle}$.

Autrement dit, nous pouvons écrire la manière d'obtenir les estampilles surchargées ainsi :

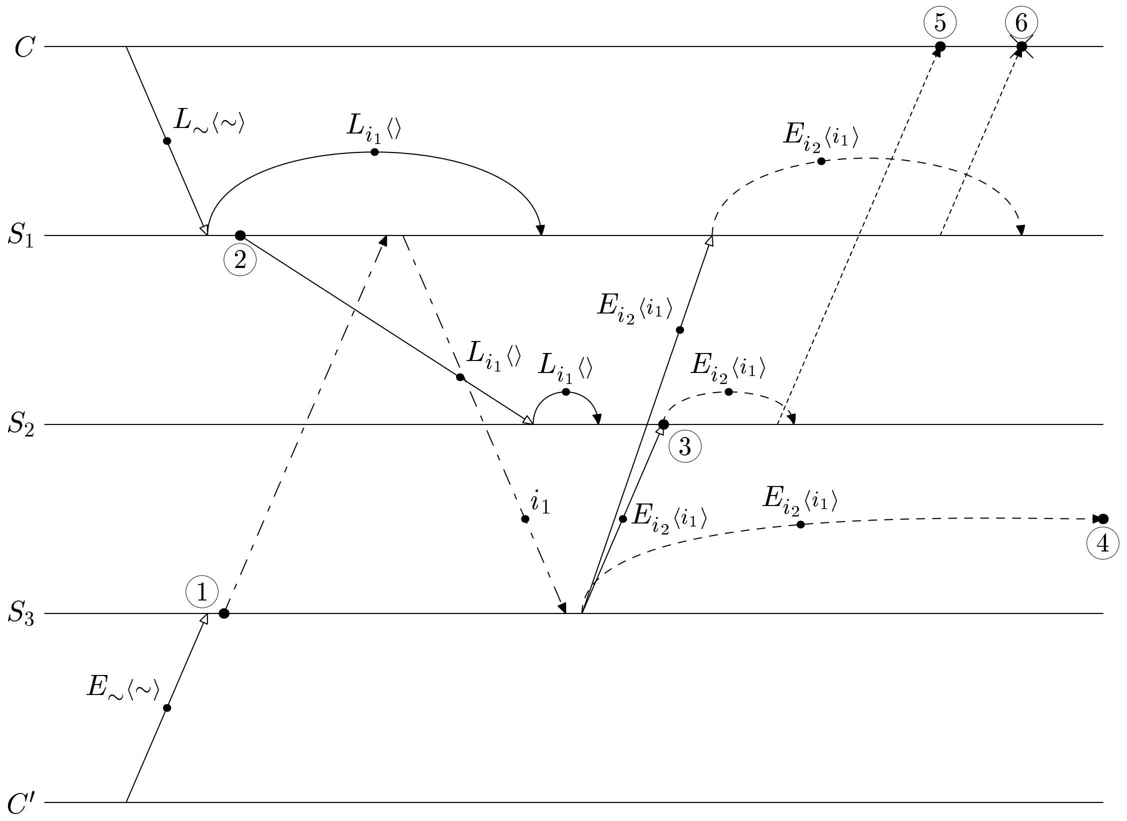


Une lecture L émise de C parvient à la copie S_1 du serveur, et n'est donc pas propagée à l'autre copie. Le client C invoque alors une écriture E estampillée par l'identificateur de L_{i_1}. Nous avons alors en :

① S_1 propage E à l'autre copie, et l'exécute puisque L_{i_1} a bien été exécutée sur cette copie ;

② S_2 reçoit E et attend *indéfiniment* que l'invocation identifiée par i_1 soit exécutée, puisque L_{i_1} n'est exécutée que sur S_1.

FIG. 3.7 – Estampilles référençant une lecture (problème 1/2)



Dans cet exemple, nous voulons fournir une cohérence atomique objet aux clients. Nous voulons donc que l'écriture soit exécutée après la lecture. Nous avons alors :

① S_3 demande à S_1 le droit d'estampiller l'invocation en écriture ;

② S_1 décide de propager la lecture à S_2 (par exemple, parce qu'il se considère surchargé), tout en la conservant en vue de l'exécuter. Ainsi C profitera de la réponse du plus rapide des serveurs entre S_1 et S_2 ;

③ E est retardée, car estampillée avec i_1 et L n'a pas fini d'être exécutée ;

④ S_3 reçoit E et attend *indéfiniment* que la lecture identifiée par i_1 soit exécutée ;

⑤ C reçoit la réponse de S_2, d'où l'importance du retard en 3, pour assurer une réponse valide ;

⑥ Finalement C obtient la réponse de S_1, mais comme il l'a déjà obtenue de S_2, il l'ignore.

FIG. 3.8 – Estampilles référençant une lecture (problème 2/2)

$$L_i\langle i_1,\ldots,i_n\rangle \text{ et } E_j\langle \ldots,i,\ldots\rangle \quad\Longrightarrow\quad L_i\langle i_1,\ldots,i_n\rangle \text{ et } E_j\langle \ldots,i\langle i_1,\ldots,i_n\rangle,\ldots\rangle$$

Pour la suite, la définition suivante est indispensable :

Définition 3.5 (Invocation partielle et totale)
Une invocation est partielle sur un objet o quand ses paramètres sont inconnus alors que son identificateur et ses estampilles sont connus. Dans le cas contraire, elle est totale.

Donc, si $E_j\langle \ldots,i\langle i_1,\ldots,i_n\rangle,\ldots\rangle$ arrive sur un objet o, mais pas $L_i\langle i_1,\ldots,i_n\rangle$, alors L_i est une invocation partielle sur o. Une invocation partielle est dûe au traitement d'une estampille surchargée. Contrairement à une invocation en lecture, une invocation en écriture étant toujours diffusée, elle est toujours totale.

Le principe suivant gère les estampilles surchargés. Par soucis de compréhension, ce principe n'utilise qu'une estampille. La forme générale $E_i\langle i_0^1\langle i_1^1,\ldots,i_n^1\rangle,\ldots,i_0^p\langle i_1^p,\ldots,i_m^p\rangle\rangle$ peut en être facilement déduite.

Principe 3.3 (Estampille d'ordonnancement surchargée)
Quand une invocation en écriture E d'identificateur i ayant comme estampille e et les estampilles surchargées e_1 à e_n rattachée à e arrive sur un objet o :

- Si $L_e\langle e_1,\ldots,e_n\rangle$ n'est pas totale sur o déclarer $L_e\langle e_1,\ldots,e_n\rangle$ comme une invocation partielle ;
- Traiter $E_i\langle e\rangle$.

Maintenant que nous avons répondu au deuxième sous problème, nous pouvons répondre simplement au premier. Comment une invocation I qui dépend d'une autre invocation en lecture L peut-elle éviter d'attendre cette dernière indéfiniment ? Il suffit qu'une invocation I dépendante d'une invocation L partielle sur un objet o puisse être considérée *éventuellement* exécutée quand toutes les dépendances de L l'auront été, mais sans qu'elle le soit réellement. Ainsi, les invocations n'ont pas besoin d'attendre l'exécution d'une invocation partielle (ce qui est impossible, et qui provoque l'attente infinie), mais seulement l'exécution de ses dépendances. Cela peut se décrire par le principe suivant :

Principe 3.4 (Exécution d'une invocation partielle)
Une invocation partielle $L\langle e_1,\ldots,e_n\rangle$ est considérée exécutée quand les n invocations d'identificateur e_i sont exécutées (avec $1 \leq i \leq n$).

3.3.1.3 Estampilles invalides

Pour maximiser les performances, nous avons déjà évoqué le cas intéressant d'un client qui invoque normalement une lecture, et après un certain temps, la réinvoque en demandant une diffusion (restreinte ou non) pour accélérer son traitement. À la première transmission, l'invocation est partielle sur toutes les copies sauf une. À la réception de la seconde transmission, il faut donc passer l'invocation de l'état partiel à l'état total puisque l'objet reçoit la méthode et ses paramètres. Le principe suivant remplit ce rôle.

Principe 3.5 (Passage de l'état partielle à totale)
Si une invocation I arrive sur un objet o et si I est une invocation partielle sur o, alors déclarer I comme une invocation totale.

Il faut alors faire attention à ce qu'une invocation en lecture $L_i\langle ... \rangle$ qui arrive après l'exécution d'une invocation en écriture $E\langle i\langle ... \rangle \rangle$ (c'est-à-dire qui dépend de L et qui crée L à l'état partiel), ne renvoie pas une valeur incohérente. En effet, d'après les dépendances, L doit renvoyer une valeur dépendant de l'état de l'objet avant l'exécution de E. Pour éviter ce mauvais comportement, il faut utiliser les deux principes complémentaires suivants :

Principe 3.6 (Identificateur invalide)
Avant d'exécuter une invocation $I\langle e_1, ..., e_n \rangle$, déclarer les identificateurs e_1 à e_n invalides.

Principe 3.7 (Invocation invalide)
Quand une invocation I possédant un identificateur i arrive sur un objet, et que i appartient à la liste des identificateurs invalides, I est ignorée.

Ainsi, quelque soit l'invocation I, son exécution invalide les identificateurs des invocations dont elle dépend, et donc les empêchera de s'exécuter ultérieurement.

Il ne reste plus qu'a garantir qu'aucune invocation en lecture ne soit perdue, car déclarée invalide sur toutes les copies de l'objet. Pour cela, les gestionnaires d'estampilles respectent le principe suivant :

Principe 3.8 (Condition d'utilisation d'estampilles)
Le gestionnaire d'estampilles doit garantir qu'il n'utilise pas l'identificateur d'une invocation en lecture avant que cette dernière ne soit totale sur au moins une des copies.

Ce principe garantit qu'aucune lecture n'est perdue. Tout d'abord parce qu'aucune autre invocation ne pourra être estampillée par son identificateur avant qu'elle ne soit totale sur au moins un site. Ensuite, parce que si une invocation en lecture est totale, alors elle ne pourra plus être ignorée. Ainsi aucune invocation ne pourra invalider son identificateur sur toutes les copies.

Lorsque le gestionnaire d'estampilles reçoit l'invocation en lecture, il lui assigne éventuellement un identificateur et une estampille, et il la place en attente d'exécution dans sa queue locale, elle devient donc une invocation totale sur ce site. Il peut, à partir de ce moment, utiliser son identificateur comme estampille. Avec ce schéma, ce principe n'introduit aucun délai supplémentaire dû à la propagation dans l'ordonnancement des invocations.

3.3.1.4 Dépendance de propagation

Une optimisation supplémentaire est possible avec les estampilles surchargées. En effet, considérons deux invocations en écriture dépendant d'une même invocation en lecture, par exemple, $L_i\langle e \rangle$, $E\langle i\langle e \rangle \rangle$ et $E^{'}\langle i\langle e \rangle \rangle$. Si E arrive sur o avant $E^{'}$, alors l'estampille surchargée e rattachée à i sur $E^{'}$ est inutile, puisque $L_i\langle e \rangle$ est déjà dans un état partiel sur o à l'arrivée de E. Le schéma est symétrique, il y a donc le même comportement si E arrive après $E^{'}$.

Il est possible de ne pas surcharger les estampilles sur une des invocations en écriture, et donc d'optimiser la taille et le coût de traitement des messages. Il suffit de surcharger l'estampille i uniquement sur la première des invocations en écriture émises. Ainsi, si le client émet L, puis E et termine en émettant $E^{'}$, les invocations seront propagées sous la forme $L_i\langle e \rangle$, $E\langle i\langle e \rangle \rangle$ et $E^{'}\langle i \rangle$.

Néanmoins, l'ordre de réception du côté des serveurs n'est pas garanti par le gestionnaire de propagation, puisque c'est au rôle des estampilles d'ordonnancement de garantir cet ordre.

Donc, il se peut qu'un objet o reçoive $E^{'}\langle i\rangle$ suivie de $E\langle i\langle e\rangle\rangle$. Dans ce cas, $E^{'}$ ne peut pas être exécutée tant que E n'est pas arrivée sur o. Cette dépendance ne fait pas partie de l'ordre représenté par les estampilles. On peut donc croire que ce principe ajoute des dépendances non souhaitées.

Cette dépendance est en fait différente des autres dépendances, nous l'appelons « dépendance de propagation ». En effet, une fois E arrivée sur o, $E^{'}$ pourra être exécutée avant E puisqu'il n'existe pas de dépendance représentée par une estampille. Cette dépendance de propagation retarde donc uniquement une invocation tant qu'une autre n'est pas *arrivée* sur un objet o, et non pas *exécutée* par o.

Finalement, cette optimisation qui consiste à ne surcharger une estampille que sur la première invocation estampillée avec est un compromis entre la taille des messages, et le nombre de dépendances de propagation introduites. Ce n'est pas forcément pénalisant d'introduire quelques dépendances de propagation. Dans tous les cas, c'est le gestionnaire d'estampilles, et non le gestionnaire d'ordonnancement, qui fait ce choix en fonction de la cohérence qu'il implante. Nous pouvons alors énoncer ce principe :

Principe 3.9 (Condition de surcharge d'estampilles)
Le gestionnaire d'estampilles doit surcharger chaque estampille référençant une invocation en lecture avec les estampilles de cette invocation jusqu'à ce qu'au moins une invocation en écriture transporte cette surcharge.

3.3.1.5 Graphe de dépendance transformé

Le principe des estampilles surchargées est récursif, c'est-à-dire que si nous avons $E_{i_3} \rightharpoonup L_{i_2} \rightharpoonup L^{'}_{i_1}$, la première invocation en lecture est émise normalement, la seconde avec une première surcharge, $L_{i_2}\langle i_1\langle ...\rangle\rangle$, et l'invocation en écriture avec deux surcharges imbriquées, $E_{i_3}\langle i_2\langle i1\langle ...\rangle\rangle\rangle$. En effet, le principe précédent indique que l'ont peut arrêter de surcharger une estampille seulement quand elle aura été transportée par une invocation en écriture.

Pour limiter ce problème, nous pouvons limiter les dépendances d'invocations en lecture sur une autre invocation en lecture.

Si nous posons l'hypothèse qu'une invocation I qui ne dépend d'aucune autre invocation, dépend de l'invocation vide $\emptyset$, une invocation en lecture L a donc comme schéma de dépendance un des trois cas suivants :

$$\begin{cases} (1) \quad L \rightharpoonup \emptyset \\ (2) \quad L \rightharpoonup E \\ (3) \quad L \rightharpoonup L^1 \rightharpoonup \ldots \rightharpoonup L^n \rightharpoonup E \quad (\text{avec } n \geq 1) \end{cases}$$

Or, par la définition 3.1 page 54, l'exécution d'une invocation en lecture sur un objet ne change pas l'état de l'objet. L'inversion de l'ordre des dépendances des invocations L^1 à L^n du cas 3 ne modifie donc pas l'état de l'objet. Nous pouvons donc remplacer la dépendance d'une invocation en lecture L sur une autre invocation en lecture $L^{'}$ par une dépendance de L sur la *dépendance* de $L^{'}$. Nous obtenons donc la transformation suivante, x représentant un ensemble d'invocations quelconques (à noter que seules les dépendances transformées sont représentées) :

$$x \rightharpoonup L \rightharpoonup L^{'} \rightharpoonup I \quad \longrightarrow \quad x \rightharpoonup L \rightharpoonup I \text{ et } x \rightharpoonup L^{'} \rightharpoonup I$$

En appliquant récursivement sur n cette transformation au cas 3, nous pouvons ainsi remplacer toutes les dépendances des invocations en lecture entre elles, par des dépendances sur des invocations en écriture. Nous obtenons alors après transformation, les deux cas suivants :

$$\begin{cases} (1') & L \rightharpoonup \emptyset \\ (2') & L \rightharpoonup E \end{cases}$$

Cette transformation, appliquée au graphe de dépendance direct, produit donc un autre graphe, que nous appelons graphe de dépendance transformé, où une invocation en lecture est soit indépendante, soit dépendante d'une invocation en écriture. Elle supprime un arc, et en ajoute d'autres qui sont dûs à la transitivité de la relation d'ordre strict $\rightharpoonup$. Il est donc clair que le graphe de dépendance transformé est un graphe partiel du graphe de dépendance. Par conséquent, il est sans circuit et conserve les propriétés d'anti-symétrie et d'irréflexivité.

Ce graphe minimise donc le problème dû à la récursivité des estampilles surchargées engendrées par les dépendances $L \rightharpoonup L'$, puisqu'il maximise la présence des surcharges sur les invocations en écriture. Le programmeur de gestionnaire d'estampilles doit donc, dans la mesure du possible, appliquer cette transformation.

3.3.1.6 Représentation du graphe

Nous devons maintenant spécialiser la figure 3.2 page 58 en faisant intervenir le type des invocations. Ainsi, la figure 3.9 page ci-contre décrit la représentation d'un motif du graphe de dépendance transformé à l'aide des estampilles surchargées.

3.3.1.7 Utilisation

Nous considérons sur cet exemple les différents scénarii qui peuvent se présenter lors de l'utilisation des estampilles surchargées. Pour se placer dans une situation comportant le plus de cas possibles, l'invocation en lecture est diffusée sur toutes les copies du serveur. Nous pouvons alors avoir le scénario de la figure 3.10 page 72.

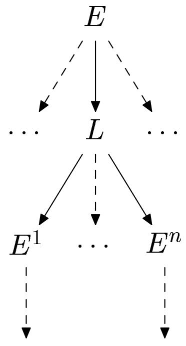
3.3.2 Taille de la liste d'estampilles

3.3.2.1 Problème

La liste d'estampilles présente sur des invocations pose le problème de sa taille. Cette liste étant non bornée, le gestionnaire d'estampilles peut théoriquement constituer une liste infinie, posant alors le problème du surcoût réseau dû à son transport, et du surcoût processeur dû à sa gestion.

Prenons l'exemple du scénario de la figure 3.11 page 73. Le gestionnaire d'estampilles doit fournir une cohérence atomique objet avec l'algorithme rapidement décrit au § 3.2.6.4 page 60. Une liste d'estampilles est utilisée pour estampiller une invocation en écriture avec la liste de tous les identificateurs des invocations en lecture précédentes. Il est alors clair que la taille de la liste n'est pas maîtrisée par le gestionnaire d'estampilles, mais dépend de l'application et de son exécution (de son ordonnancement entre autres).

De plus, ce type de dépendance est extrêmement fréquent. Il s'agit d'un groupe d'invocations qui sont exclusivement référencées en tant que groupe, sans être référencées individuellement par ailleurs. On retrouve un tel schéma pour la cohérence atomique, pour les



(a) Graphe

- $E\langle ...,i_0\langle i_1,...,i_n\rangle,...\rangle$;
- $L_{i_0}\langle i_1,...,i_n\rangle$;
- $E^1_{i_1}\langle ...\rangle, \ldots, E^n_{i_n}\langle ...\rangle$.

(b) Estampille

Fig. 3.9 – Graphe de dépendance et estampilles surchargées

synchronisations, exclusions mutuelles, barrières, etc. Il est donc intéressant d'optimiser ce type de dépendance.

3.3.2.2 Groupe d'invocations

La solution retenue consiste à rassembler les invocations dans un groupe et non plus une liste, et à faire référencer ce groupe par *une seule* estampille. Pour cela, il suffit d'autoriser plusieurs invocations à posséder le même identificateur.

Définition 3.6 (Groupe d'invocations)
Un groupe d'invocations est un ensemble non ordonné d'invocations ayant le même identificateur.

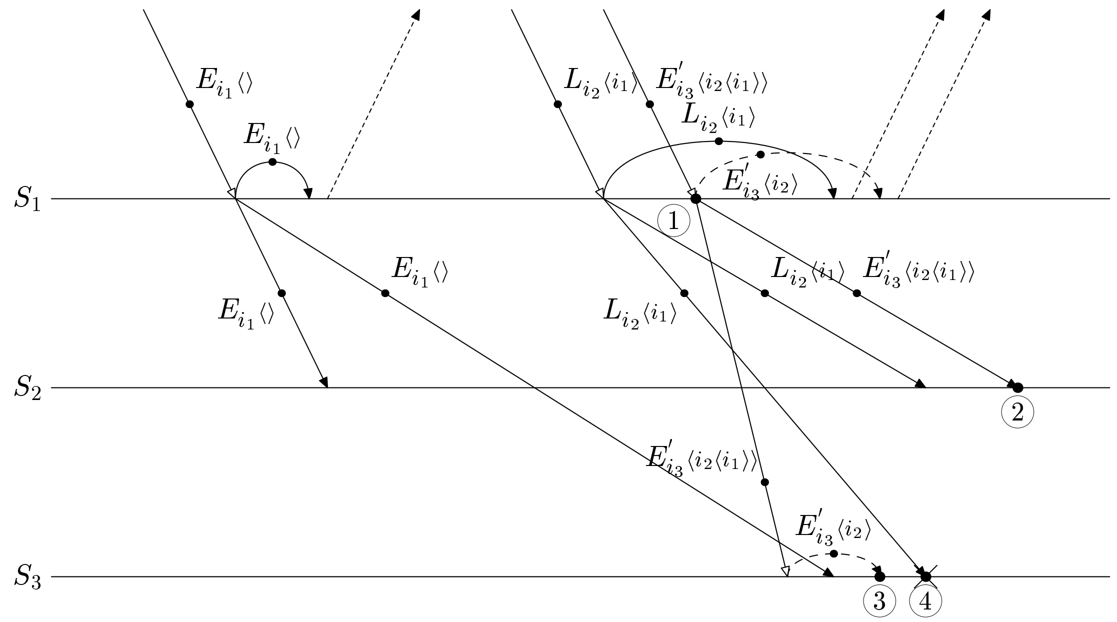
Nous pouvons définir le type du groupe d'invocations, en fonction du type des invocations qu'il contient, comme suit :

Définition 3.7 (Type d'un groupe d'invocations)
Un groupe d'invocations peut être de type :

- *lecture* si toutes les invocations qui le composent sont des invocations en lecture,
- *écriture* si toutes les invocations qui le composent sont des invocations en écriture,
- *lecture/écriture* (ou *mixte*) s'il contient au moins une invocation en lecture et une en écriture.

Les invocations en lecture dans un groupe, tout comme les invocations individuelles en lecture, ne sont pas forcément diffusées. Un groupe d'invocations sur un objet o particulier contient donc, à terme, toutes les invocations en écriture, et un nombre quelconque d'invocations en lecture. Nous définissons le cardinal d'un groupe comme le nombre minimal d'invocations qu'il contient sur n'importe quelle copie :

Définition 3.8 (Cardinal d'un groupe d'invocations)
Le cardinal d'un groupe d'invocations est le nombre d'invocations en écriture dans ce groupe.



Une écriture E d'identificateur i_1 arrive sur S_1. Puis une lecture L d'identificateur i_2 et estampillée avec i_1 parvient à S_1. Enfin, une écriture E' d'identificateur i_3 et estampillée par la lecture i_2 atteint S_1. L'écriture E'_{i_3} est estampillée par une lecture (L_{i_2}). Cette estampille i_2 doit donc avoir été surchargée avec l'estampille de L_{i_2}. Il s'agit ici de i_1. Une fois arrivée sur les serveurs, cette surcharge est traitée comme une invocation. Nous avons alors les cas suivants :

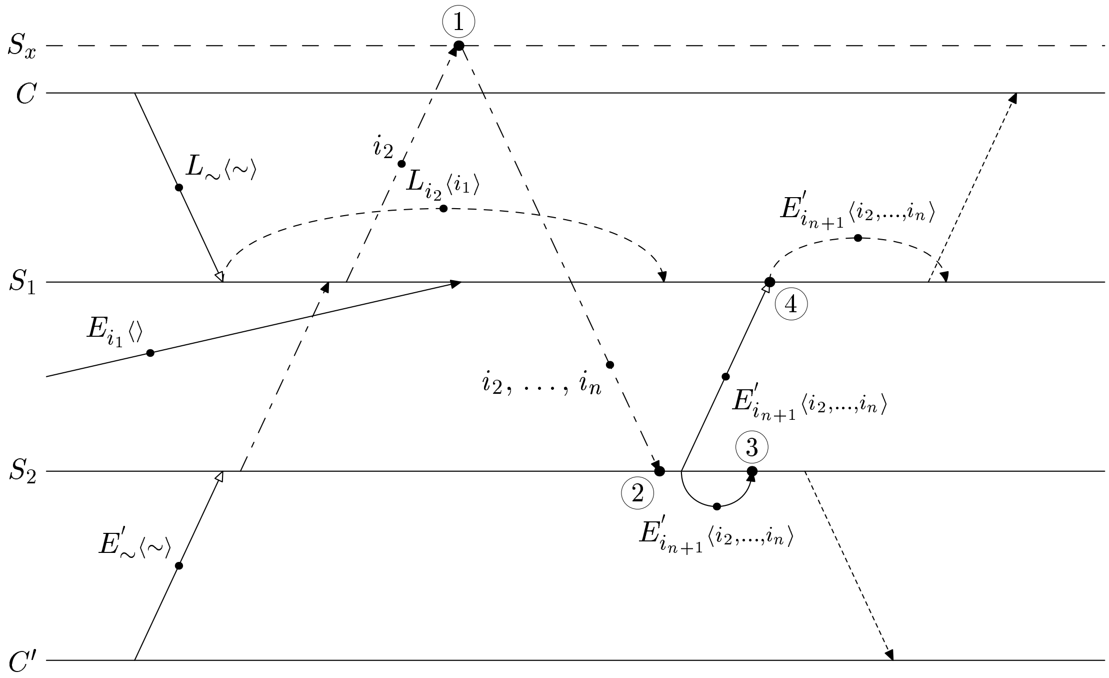
① E'_{i_3} est retardée, puisque L_{i_2}, déjà parvenue sur S_1, est en attente d'exécution ;

② E'_{i_3} est en cours d'exécution, puisque E_{i_1} puis L_{i_2} ont déjà été exécutées sur S_2 ;

③ E'_{i_3} est retardée en attendant E_{i_1}. En effet, S_3 attend tout d'abord que E_{i_1} soit exécutée. Puis, L_{i_2} étant une invocation partielle, il déclare i_2 invalide et exécute E'_{i_3} ;

④ Si L_{i_2} arrive quand même sur S_3 pendant ou après l'exécution de E'_{i_3}, celle ci est tout simplement ignorée car i_2 est maintenant un identificateur invalide.

Fig. 3.10 – Gestion des lectures avec les estampilles surchargées



Nous voulons ici maintenir une cohérence atomique objet. Initialement, S_1 possède les droits « en lecture », ainsi que d'autres serveurs S_x. C a émis L. C' veut émettre une écriture $E^{'}$ qui doit donc être exécutée après toutes les invocations en lecture. S_2 doit donc demander à S_1 la valeur de l'estampille référençant toutes les invocations en lecture.

① S_1 demande à chaque copie ayant les droits en « lecture », de ne plus l'utiliser ;

② S_2 reçoit les estampilles référençant toutes les invocations en lecture émises, elle peut donc les utiliser pour estampiller $E^{'}$;

③ S_2 peut immédiatement délivrer $E^{'}$ à l'application sans attendre que toutes les invocations en lecture soient terminées ;

④ Bien sûr, puisque i_2 appartient à la liste, $E^{'}$ est retardée seulement sur les copies ayant une exécution d'une invocation en lecture en cours.

Fig. 3.11 – Taille importante de la liste d'estampilles (problème)

Définition 3.9 (Cardinal d'une estampille d'ordonnancement)
Par extension, on dira que le cardinal d'une estampille d'ordonnancement est le cardinal du groupe d'invocations qu'elle référence.

Nous devons donc remplacer le principe 3.2 page 56 sur les estampilles d'ordonnancement pour qu'il utilise le cardinal des groupes d'invocations :

Principe 3.10 (Groupe d'invocations)
Quand une invocation I estampillée par e_1 à $e_{n'}$ de cardinaux respectifs n_1 à n'_n arrive sur un objet o :

- Si $n_i = 0$, alors attendre qu'au moins une invocation en lecture d'identificateur e_i ait été exécutée (avec $1 \leq i \leq n'$) ;
- Si $n_i > 0$, alors attendre que les n_i invocations en écriture d'identificateur e_i aient été exécutées (avec $1 \leq i \leq n'$) ;
- Attendre que toutes les invocations en lecture arrivées sur o d'identificateur e_i aient été exécutées (avec $1 \leq i \leq n'$) ;
- Délivrer I à o.

Concrètement, quand une invocation arrive, ce principe la fait attendre que toutes les invocations en écriture des groupes mixtes ou en écriture dont elle dépend soient exécutées. Ou bien il la fait attendre qu'au moins une invocation de chaque groupe en lecture dont elle dépend soit exécutée pour satisfaire la transitivité de la relation représentée par les estampilles de ces groupes en lecture. Enfin, il la fait attendre que toutes les invocations en lecture dont elle dépend, et qui sont déjà arrivées sur l'objet, soient exécutées pour maximiser la proportion d'invocations en lecture réellement exécutées.

3.3.2.3 Connaissance du cardinal

Nous avons donc besoin de connaître le cardinal d'un groupe pour appliquer ce principe d'ordonnancement. Nous pouvons indiquer le cardinal à chaque fois que nous émettons l'estampille. C'est une solution simple, mais qui a un inconvénient : le groupe doit être clos, c'est-à-dire son cardinal connu, au moment où son identificateur est utilisé pour estampiller une invocation.

Seulement, nous aimerions être capable d'estampiller une invocation avec un groupe non encore entièrement constitué. Nous verrons par la suite des exemples d'utilisation d'un tel mécanisme. Nous proposons donc deux moyens pour indiquer le cardinal d'un groupe.

Principe 3.11 (Connaissance du cardinal)
Quand une estampille e référence un groupe d'invocations G en écriture ou mixte, nous devons choisir au moins une des deux façons suivantes de faire connaître le cardinal de G.

- Pour toutes les invocations estampillées avec e, e doit être accompagnée du cardinal de G ;
- Au moins une invocation en écriture d'identificateur e doit être accompagnée du cardinal de G.

En effet, pour que le principe d'ordonnancement 3.10 ne ralentisse pas (et à fortiori ne bloque pas) l'exécution d'une invocation I estampillée par e référençant un groupe d'invocations G en écriture ou mixte, il faut que le serveur connaisse le cardinal de G avant que I ne soit prête à être exécutée. Or :

- Si toutes les invocations estampillées avec e sont accompagnées du cardinal de G, alors il est clair que le serveur à la réception de I connaîtra le cardinal de G avant ou au moment où I est prête à être exécutée ;
- Sinon, s'il existe au moins une écriture E d'identificateur e qui spécifie le cardinal n du groupe G, et puisque l'invocation I estampillée par e n'est exécutée que quand toutes les invocations en écriture de G l'auront été, alors nécessairement, le serveur reçoit E et donc connaît n avant que I ne soit prête à être exécutée.

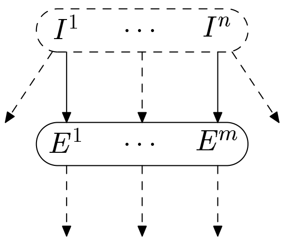
Notation 3.5
Nous notons une estampille e accompagnée du cardinal n du groupe auquel elle fait référence par $e : n$. De même, un identificateur i accompagné du cardinal n du groupe qu'il représente est noté $i : n$.

3.3.2.4 Représentation du graphe

La représentation du graphe de dépendance transformé par les groupes d'invocations se divise en trois cas, suivant que le groupe d'invocations est en lecture, en écriture ou mixte. Dans les figures suivantes, les groupes en traits pleins représentent les groupes d'invocations, alors que ceux en traits pointillés sont une représentation d'un regroupement logique de zéro, une ou plusieurs invocations utilisé pour améliorer la lisibilité des figures.

La figure 3.12 concerne les groupes en écriture. Une invocation en écriture n'impliquant pas d'estampilles surchargées sur les invocations la référençant, sa représentation avec les estampilles est donc simple et sans aucune contrainte.

Par contre, les groupes en lecture font intervenir les estampilles surchargées des invocations les référençant. Une condition sur les estampilles des invocations du groupe existe : elles doivent être toutes identiques dans le groupe. Ainsi, toutes les invocations référençant le groupe auront la même estampille, accompagnée par les mêmes estampilles surchargées. Cette condition est représentée dans la figure 3.13 page suivante.

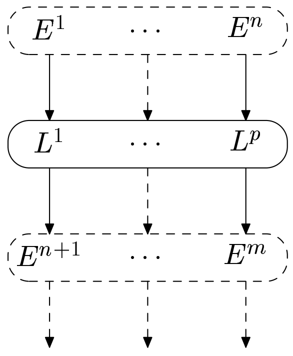


(a) Graphe

- $I^1\langle ...,i_0,... \rangle$, … , $I^n\langle ...,i_0,... \rangle$;
- $E^1_{i_0}\langle ... \rangle$, … , $E^m_{i_0}\langle ... \rangle$;
- $i_0 : m$.

(b) Estampille

FIG. 3.12 – Graphe de dépendance et groupes d'invocations en écriture



(a) Graphe

- $E^1_{\cdot}\langle ...,i_0\langle i_{n+1},...,i_m\rangle\rangle$, ..., $E^n_{\cdot}\langle ...,i_0\langle i_{n+1},...,i_m\rangle\rangle$;
- $L^1_{i_0}\langle i_{n+1},...,i_m\rangle$, ..., $L^p_{i_0}\langle i_{n+1},...,i_m\rangle$;
- $E^{n+1}_{i_{n+1}}\langle ...\rangle$, ..., $E^m_{i_m}\langle ...\rangle$;
- $i_0 : 0$.

(b) Estampille

FIG. 3.13 – Graphe de dépendance et groupes d'invocations en lecture

Enfin, la gestion des groupes mixtes est similaire à celle des groupes en lecture, si ce n'est que les invocations en écriture référençant le groupe ne comportent pas d'estampilles surchargées, puisque les invocations en écriture du groupe garantissent le respect de la dépendance sur les invocations suivantes. La figure 3.14 page suivante présente ces groupes.

3.3.2.5 Groupe d'invocations *vs* liste d'estampilles

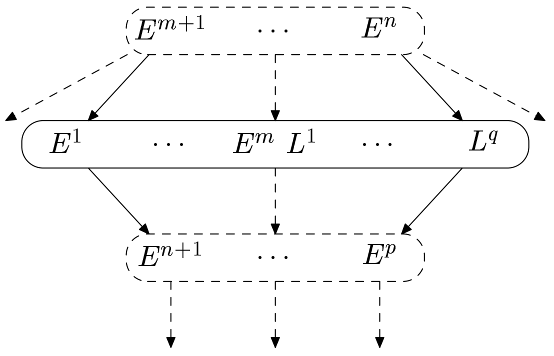
Nous pouvons nous demander s'il est concevable d'abandonner l'utilisation de la liste d'estampilles au profit du groupe d'invocations. Or il est impossible de représenter n'importe quel graphe de dépendance transformé avec le groupe. Par exemple, considérons ce graphe :

$$\begin{tikzcd} I^1 \arrow[r] \arrow[dr] & I^2 \\ I^3 \arrow[r] & I^4 \end{tikzcd}$$

Il est clair que si nous mettons I^2 et I^4 dans un groupe d'invocations pour que I^1 le référence, nous ajoutons une dépendance $I^3 \rightharpoonup I^2$ puisque les membres d'un groupe ne peuvent pas être référencés individuellement.

Par contre, le groupe d'invocations est superflu par rapport à la liste d'estampilles d'un point de vue strictement théorique. Mais les optimisations qu'il introduit sont loin d'être négligeables, surtout par son utilisation extrêmement fréquente. Ainsi, dans les cas où le groupe est utilisable, nous évitons le problème de la taille des listes. Mais aussi, nous introduisons un asynchronisme plus fort entre l'attribution des informations d'ordonnancement et l'exécution puisque nous pouvons émettre des invocations estampillées avec l'identificateur d'un groupe avant que celui-ci soit entièrement constitué. Cela permet, entre autres, d'acquérir l'estampille sur un autre serveur sans attendre la terminaison des invocations dont elle dépend. Cette dernière propriété nous permet des comportements fort intéressants, comme par exemple dans la gestion de l'exclusion mutuelle que nous décrivons au § 3.3.3.2 page ci-contre.

3.3.3 Utilisations



(a) Graphe

- $E^{m+1}\langle ...,i_0,...\rangle$, ..., $E^n\langle ...,i_0,...\rangle$;
- $E^1_{i_0}\langle i_{n+1},...,i_p\rangle$, ..., $E^m_{i_0}\langle i_{n+1},...,i_p\rangle$, $L^1_{i_0}\langle i_{n+1},...,i_p\rangle$, ..., $L^q_{i_0}\langle i_{n+1},...,i_p\rangle$;
- $E^{n+1}_{i_{n+1}}\langle ...\rangle$, ..., $E^p_{i_p}\langle ...\rangle$;
- $i_0 : m$.

(b) Estampille

Fig. 3.14 – Graphe de dépendance et groupes d'invocations mixtes

3.3.3.1 Maintien de la cohérence atomique objet « inverse »

Nous allons reprendre l'exemple de la figure 3.6 page 63, mais en modifiant les rôles des écritures et des lectures. Tout d'abord parce que cela simplifie le chronogramme, mais surtout pour illustrer la puissance et la souplesse de la propagation des invocations alliées aux estampilles d'ordonnancement. En effet, avec une mémoire virtuellement partagée, il serait impossible d'utiliser une telle cohérence de façon performante.

Le serveur fournit une méthode en écriture commutative, c'est-à-dire dont les invocations peuvent être exécutées dans une ordre quelconque. Il possède aussi une méthode en lecture pour consulter la valeur. Si le nombre d'invocations en écriture est bien supérieur au nombre de consultations, le concepteur aimerait pouvoir donner aux clients le moyen d'invoquer l'écriture avec un coût beaucoup plus faible que la lecture.

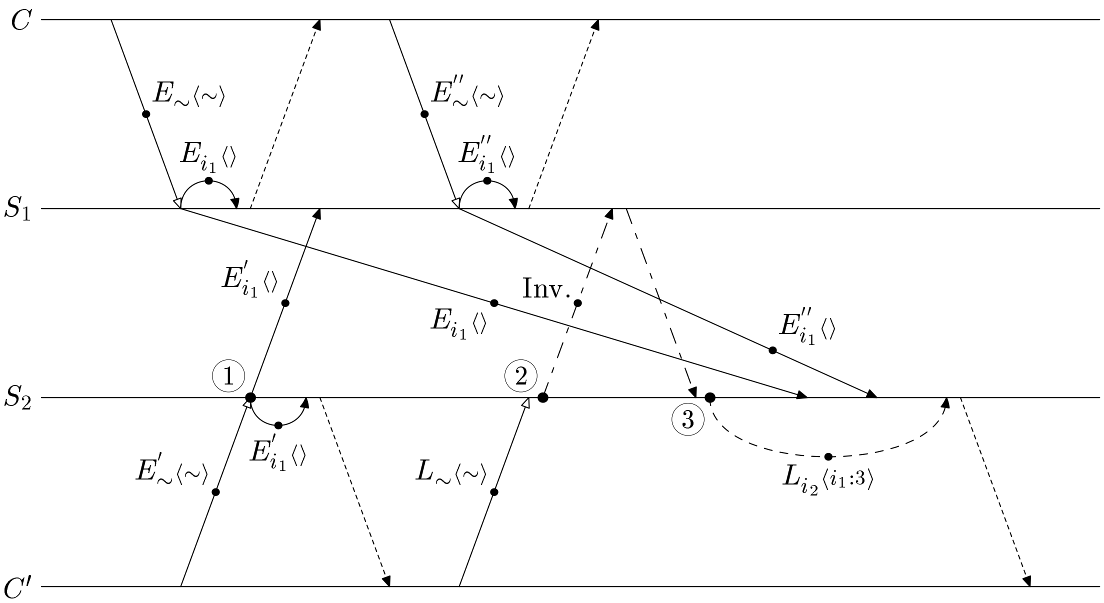
Par exemple, ce serveur peut être un compteur avec une méthode d'incrémentation et une méthode de consultation de la valeur. Son utilisation peut être faite à des fins statistiques pour savoir combien de fois une méthode est invoquée sur un objet. L'incrémentation doit donc être faite de la manière la moins pénalisante pour le serveur, puisqu'elle est invoquée à chaque utilisation du serveur. La consultation de la valeur peut sans problème être beaucoup plus coûteuse, puisque beaucoup moins utilisée.

Pour satisfaire ces critères, nous utiliserons le même protocole qu'au § 3.2.6.4 page 60, mais en inversant le rôle des invocations en lecture avec celles en écriture. Nous montrons un exemple de scénario possible à la figure 3.15 page suivante.

3.3.3.2 Gestion d'exclusion mutuelle

Nous allons fournir un autre exemple d'application pour deux raisons. La première est que le cardinal du groupe d'invocations est communiqué par un identificateur (et non une estampille comme pour l'exemple précédent). La deuxième est qu'il montre l'utilité des estampilles d'ordonnancement pour des mécanismes de synchronisation, en l'occurrence une exclusion mutuelle.

Un client voulant acquérir un verrou sur un objet, doit le demander avec une opération



Initialement, les deux copies S_1 et S_2 connaissent l'estampille (vide) et l'identificateur du groupe. Quand ils reçoivent les invocations en écriture, les serveurs n'ont donc qu'à leurs assigner un identificateur, une estampille, et à les diffuser. Nous pouvons alors voir aux différents repères :

① Les écritures sont commutatives, il n'y a donc aucun inconvénient à exécuter E' avant E sur S_2, bien que l'ordre soit inversé sur S_1 ;

② À la première lecture, S_2 retire le droit d'estampiller à S_1, qui lui retourne le nombre d'invocations en écriture qu'elle a estampillé (soit E et E'', donc 2) ;

③ Connaissant le cardinal du groupe i_1 (2 provenant de S_1 et 1 de S_2), S_2 retarde L jusqu'à ce que les 3 invocations en écriture estampillées avec i_1 soient exécutées.

Fig. 3.15 – Cohérence atomique objet « inverse » avec les groupes d'invocations

P. Il obtient en retour un identificateur et une estampille avec lesquels il doit commencer à identifier et estampiller les invocations. Quand il veut rendre le verrou, il ne le fait pas avec une opération V classique, mais l'indique avec sa dernière invocation en section critique (l'invocation terminale). Le gestionnaire d'estampilles communique alors le cardinal du groupe, qu'il calcule en comptant le nombre d'invocations en écriture effectuées durant la section critique.

Un exemple de scénario possible est présenté figure 3.16 page suivante. Nous pouvons remarquer qu'un client concurrent peut acquérir l'exclusion mutuelle avant qu'elle ne soit rendue, et commencer à invoquer des méthodes dedans. Bien sûr, elles ne seront exécutées qu'à la fin de la section par l'autre client. Ainsi, les messages de gestion du verrou, et la diffusion des invocations peuvent se faire en parallèle avec l'exécution de la section critique par un autre client. Il est évident que cette caractéristique est impossible à obtenir avec les listes d'estampilles, ces dernières nécessitant la connaissance exacte de la liste avant d'estampiller une invocation.

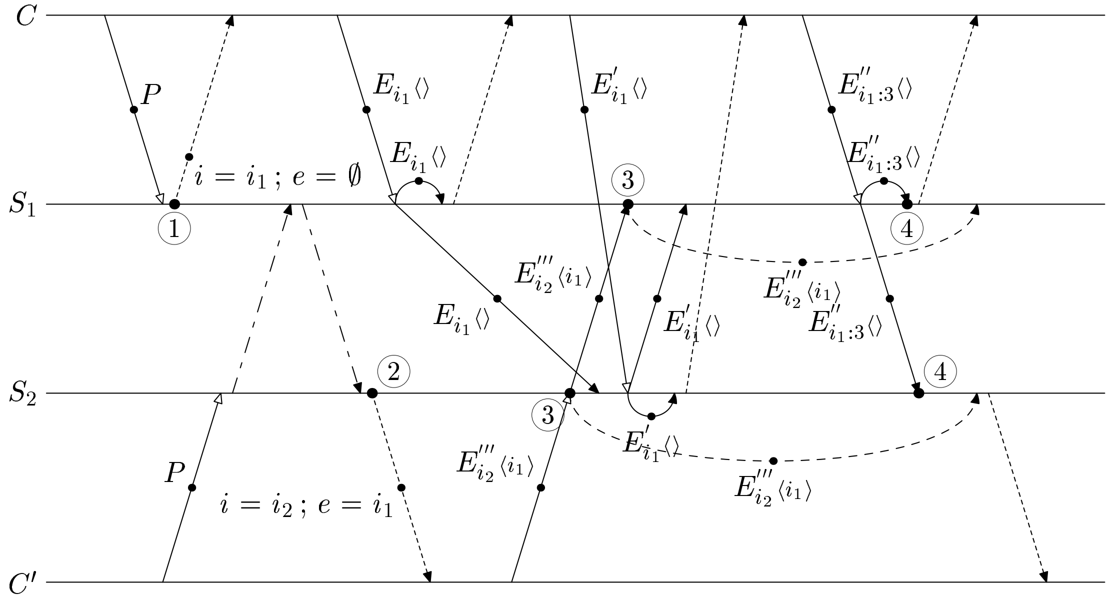
3.4 Principe général et propriétés

Nous venons d'étudier pas à pas notre mécanisme d'estampilles d'ordonnancement, tout d'abord en définissant les bases, puis en ajoutant des optimisations. Dans cette section, nous donnons en un seul principe simplifié une récapitulation des principes des estampilles d'ordonnancement (§ 3.2.3.1 page 55) et de leurs dérivés (listes d'estampilles (§ 3.2.4 page 56), estampilles surchargées (§ 3.3.1.2 page 64), identificateurs invalides (§ 3.3.1.3 page 67) et groupes d'invocations (§ 3.3.2.2 page 71)). Nous rappelons que ce mécanisme permet de représenter le graphe de dépendance transformé (§ 3.3.1.5 page 69) dérivé du graphe de dépendance direct (§ 3.2.5 page 57), lui-même déduit du graphe de dépendance (§ 3.2.2 page 55). Ce dernier est la représentation de la relation d'ordre strict « dépendant de » (§ 3.2.1 page 54) qui caractérise la ou les cohérences et synchronisations voulues. Ce principe général remplace tous les principes qui régissent le gestionnaire d'ordonnancement, sans pour autant remplacer ceux concernant le gestionnaire d'estampilles, c'est-à-dire les principes d'utilisations des estampilles (principe 3.8 page 68), de surcharge des estampilles (principe 3.9 page 69) et celui sur la connaissance du cardinal (principe 3.11 page 74).

Ce principe général ne fait plus de distinction entre les estampilles et les groupes d'invocations. En effet, une invocation en lecture est représentable par un groupe en lecture ne comportant qu'elle, et donc de cardinal 0. De même, une invocation en écriture est représentable par un groupe en écriture de cardinal 1.

Principe 3.12 (Principe général des Estampilles d'Ordonnancement)
Quand une invocation $I_{e\langle e_0^1\langle e_1^1,\ldots,e_{m_1}^1\rangle,\ldots,e_0^p\langle e_1^p,\ldots,e_{m_p}^p\rangle\rangle}$ (avec n_i^j cardinal du groupe d'identificateur e_i^j) arrive sur un objet o :

1. Si $I_{e_0^j}\langle e_1^j,\ldots,e_{m_j}^j\rangle$ est inconnue, alors la déclarer partielle (avec $1 \leq j \leq p$) ;
2. Si e est invalide alors ignorer I_e ;
3. Sinon :
   (a) Si $I_{e\langle e_0^j,\ldots,e_0^p\rangle}$ est partielle, alors la déclarer totale ;
   (b) Si $n_i^j = 0$, alors attendre qu'au moins une invocation en lecture d'identificateur e_i^j ait été exécutée (avec pour chaque $1 \leq j \leq p\colon 0 \leq i \leq m_j$) ;



Initialement, toutes les données concernant le verrou se trouvent sur S_1. Nous avons alors aux repères :

① Suite à l'acquisition du verrou, S_1 renvoie à C un identificateur et une estampille utilisables durant la section critique ;

② S_2, après avoir demandé la prochaine section critique auprès de S_1, renvoie les mêmes types d'informations à C' ;

③ L'invocation de C', bien qu'émise en parallèle avec celles de C, ne sera délivrée que quand le cardinal de i_1 sera connu et que toutes les écritures de i_1 auront été exécutées ;

④ C rend le verrou en communiquant le cardinal de i_1, ainsi E'''_{i_2} pourra être délivrée puisque les trois invocations E_{i_1}, E'_{i_1} et E''_{i_1} l'ont déjà été.

Fig. 3.16 – Exclusion mutuelle avec les groupes d'invocations

(c) Si $n_i^j > 0$, alors attendre que les n_i^j invocations en écriture d'identificateur e_i^j aient été exécutées (avec pour chaque $1 \leq j \leq p : 0 \leq i \leq m_j$) ;

(d) Attendre que toutes les invocations en lecture arrivées sur o d'identificateur e_i^j aient été exécutées (avec pour chaque $1 \leq j \leq p : 0 \leq i \leq m_j$) ;

(e) Déclarer e_i^j invalide (avec pour chaque $1 \leq j \leq p : 0 \leq i \leq m_j$) ;

(f) Si I est totale, délivrer I à o et attendre la fin de son exécution ;

(g) Déclarer I comme exécutée .

3.4.1 Vivacité

La condition de vivacité (*liveness*) du principe des estampilles d'ordonnancement est qu'il n'y ait pas de blocage dû à une attente infinie d'une invocation.

Il existe deux types de blocage possibles. Le premier type est dû à un cycle dans les références représentées par les estampilles ($I \rightharpoonup \ldots \rightharpoonup I$). Nous avons vu que les estampilles représentent le graphe de dépendance transformé et qu'il est sans circuit. Cette propriété du graphe nous garantit donc qu'il ne peut pas y avoir de blocage de ce type.

Enfin, un deuxième type de blocage peut provenir du principe 3.12 page 79. Ici, trois blocages sont possibles, dûs aux points 3b, 3c et 3d du principe. Une invocation en attente sur 3c, va nécessairement être débloquée car d'une part, les invocations en écriture sont toutes diffusées, et donc sont toutes arrivées sur toutes les copies, et d'autre part parce que le cardinal est véhiculé, d'après le principe 3.11 page 74, par une invocation en écriture (et donc de manière fiable), ou bien par l'estampille elle-même (c'est à dire par l'invocation qui est bloquée).

Les invocations en attente sur 3d vont nécessairement être débloquées. D'abord seules les invocations en lecture (partielles ou totales) arrivées sur o sont attendues. Ensuite, les invocations en lecture sont diffusées en surcharge avec au moins une estampille d'une invocation d'écriture (principe 3.9 page 69) et donc de manière fiable. Et enfin, le point 1 du principe transforme les estampilles véhiculées en surcharge en des invocations partielles.

Enfin, les invocations en attente sur 3b sont en attente d'un groupe de cardinal 0, et donc contenant au moins une invocation en lecture. Dans le cas contraire, ce groupe serait vide et son identificateur n'existerait pas. Par le même raisonnement que ci-dessus, ces invocations vont nécessairement être débloquées.

3.4.2 Sûreté

La condition de sûreté (*safety*) du principe des estampilles d'ordonnancement se compose des deux conditions suivantes :

1. Toutes les invocations en écriture sont exécutées, à terme, sur toutes les copies, après les invocations en écriture dont elles dépendent, mais avant celles qui dépendent d'elles (relativement à la relation $\rightharpoonup$). Cette condition garantit le respect de la cohérence décrite par la relation $\rightharpoonup$;
2. Toutes les invocations en lecture sont exécutées sur au moins une des copies, après les invocations en écriture dont elles dépendent, mais avant celles qui dépendent d'elles (relativement à la relation $\rightharpoonup$). Cette condition garantit la légalité des invocations en lecture, c'est-à-dire qu'elles retournent la dernière valeur écrite par rapport à l'ordre imposé par $\rightharpoonup$.

Nous considérons que les communications sont fiables. Donc toutes les invocations diffusées sont reçues sur toutes les copies. Par le point 3c du principe, une invocation quelconque dépendante (entre autres) de x invocations en écriture ne pourra être exécutée qu'après ces dernières. Par le point 3b du principe, une invocation quelconque dépendante (entre autres) de y groupes d'invocations en lecture ne pourra être exécutée qu'après l'exécution (réelle ou non) d'au moins une invocation de chaque groupe. Enfin, une invocation en écriture, forcément totale, ne peut être considérée comme exécutée qu'après son exécution réelle (point 3f). Par récurrence, cette propriété satisfait la première condition de sûreté.

Par le principe 3.8 page 68, une invocation en lecture L est au moins dans une des queues d'exécution d'une des copies, et donc elle est dans un état total sur au moins un objet. Par le point 3d du principe, aucune invocation qui dépend de L ne peut être exécutée avant L, et donc invalider son identificateur sur ce site. Donc, sur au moins cet objet, l'invocation en lecture L est réellement exécutée, puisque dans un état total (point 3f), en respectant l'ordre imposé par $\rightharpoonup$. Si sur d'autres sites, une invocation en écriture qui dépend de L est exécutée avant L, alors par le point 3e du principe, l'identificateur de L est invalide, et L n'est jamais exécutée sur ce site à cause du point 2. Ceci vérifie la deuxième condition de sûreté.

3.5 Travaux similaires

La *lazy replication* (§ 2.4.2 page 45) décrite dans [Ladin et al., 1990] fournit le moyen à un client de contrôler finement le type de cohérence voulue. Des mécanismes additionnels coté client peuvent fournir une cohérence causale transparente aux clients. Cet aspect est similaire à notre travail, car nos clients peuvent aussi contrôler la cohérence en manipulant les estampilles d'ordonnancement directement, mais les mandataires du serveur peuvent jouer ce rôle pour les rendre transparentes aux applications. La *lazy replication* est basée sur la cohérence causale (de par l'utilisation des estampilles temporelles (*timestamps*) sous forme de vecteurs), et ne peut donc pas fournir de cohérence plus faible. En fait, les clients peuvent rajouter des liens causals pour prendre en compte les canaux cachés, ou bien en supprimer, pour masquer des communications sans rapport avec la causalité. Des cohérences plus fortes sont décrites [Ladin et al., 1992], mais elles sont implantées de façon relativement rigide. Toutes les solutions décrites dans ces documents sont transposables en utilisant les estampilles d'ordonnancement, puisqu'utilisant des vecteurs d'estampilles temporelles, que nous pouvons obtenir avec une liste d'estampilles d'ordonnancement.

De même, dans le système de gestion de fichiers distribué Coda [Kistler et Satyanarayanan, 1992] les serveurs exportent l'accès aux vecteurs de versions qu'ils maintiennent. Cela permet donc à des clients de fournir leur propre cohérence. Ainsi, les quatre cohérences de session utilisées dans le système Bayou [Terry et al., 1994] ont été expérimentées sur Coda sans changer le moindre source du serveur. Cela est donc similaire à nos estampilles d'ordonnancement qui peuvent être gérées par le client. Néanmoins, notre mécanisme est plus flexible car il est utilisé dans un cadre plus large (celui des objets) et il supporte un plus large éventail de cohérences et synchronisations.

Enfin, le système de réplication décrit dans [Mishra et al., 1989], utilisant Psync (§ 2.4.3 page 46), est assez proche de nos propositions. Ce système permet d'obtenir un ordre total entre toutes les opérations non commutatives, et un ordre arbitraire dans un groupe d'opérations commutatives. Avec les estampilles d'ordonnancement, les opérations commutatives peuvent être dans un groupe d'invocations, alors que les opérations non commutatives, peuvent être identifiées et estampillées pour représenter un ordre total. Cela est similaire à l'exemple de la cohérence atomique objet, où les opérations d'écriture représentent les opération non commutatives, et les opérations de lecture, celles commutatives. Ici aussi, les estampilles d'ordonnancement s'avèrent plus souples et puissantes que l'utilisation de « graphes de contexte ». En effet, pour que ce système effectue un tri déterministe sur une vague d'opérations, il faut que la vague soit complète, et donc qu'il y ait une opération stable, c'est-à-dire connue de tous les participants. Cela retarde le tri déterministe, et donc la livraison, des opérations. Notre solution détermine l'ordre en amont grâce aux estampilles. À la réception d'une invocation, le gestionnaire d'ordonnancement a toutes les informations, ou les aura à temps, pour livrer l'invocation sans délai supplémentaire.

3.6 Conclusion

Dans ce chapitre, nous avons présenté et optimisé pas à pas notre protocole d'estampilles d'ordonnancement permettant à des objets répliqués de supporter des cohérences et des synchronisations multiples de façon performante. Il permet à chaque invocation de transporter avec elle toutes les informations nécessaires pour déterminer l'ordre de sa livraison au serveur destinataire. Ce protocole est simple, efficace, et assez générique pour s'intégrer dans divers systèmes à objets. Il ne requiert aucune autre propriété de communication que la fiabilité et l'absence de duplication, supportant ainsi des conditions difficiles (déconnexions, variations de débit, etc). Il est constitué d'un principe général, qui doit être respecté par le gestionnaire d'ordonnancement, et de principes stipulant quelques règles, peu contraignantes, au gestionnaire d'estampilles pour implanter les cohérences et synchronisations qu'il souhaite.

La contribution principale des estampilles d'ordonnancement est la séparation de la gestion des cohérences et des synchronisations (le gestionnaire d'estampilles) des mécanismes de stockage, d'ordonnancement et de livraison des invocations à l'application (gestionnaire d'ordonnancement). Cela facilite la conception et l'intégration, à posteriori, de toutes les cohérences et les synchronisations représentables par une relation d'ordre strict. Ainsi, le développeur d'un serveur répliqué à le choix du modèle de cohérence qu'il souhaite, et il peut donc utiliser le plus adéquat, et le plus performant pour son application.

Chapitre 4

Estampilles d'Ordonnancement : cadre de travail

Ce cadre de travail a été conçu autour du protocole des estampilles d'ordonnancement. D'une part, il permet de clarifier leur utilisation dans un environnement concret. D'autre part il fournit une proposition d'architecture partielle d'un système à objets principalement orienté vers la performance. Pour cela, chaque gestionnaire tire partie de son environnement et les interactions entre les objets sont les plus asynchrones et les plus locales possibles.

Dans ce chapitre, nous donnons, pour faciliter la compréhension, une vue d'ensemble de ce cadre de travail (§ 4.1). Nous détaillons alors le rôle de chaque objet et gestionnaire (§ 4.2), puis leurs interactions. Pour cela, nous décrivons d'abord le parcours de l'invocation d'une méthode d'un serveur par un client (§ 4.3). Puis nous examinons le déroulement de la création et la destruction d'une copie ainsi que le recyclage des identificateurs (§ 4.4). Enfin nous terminons en examinant les interactions entre quelques autres gestionnaires (§ 4.5.3).

4.1 Vue d'ensemble

Nous présentons dans cette section, une rapide vue d'ensemble du cadre de travail des estampilles d'ordonnancement, en présentant d'abord les différents gestionnaires (§ 4.1.1) et ensuite le déroulement de l'invocation d'une méthode d'un serveur par un client (§ 4.1.2).

4.1.1 Gestionnaires

Un objet serveur est un objet qui rend un service à un objet client (ou client). Pour supporter la réplication, cet objet serveur est accompagné de quatre autres objets gestionnaires. Nous employons le terme de copie d'un serveur pour désigner l'ensemble de ces cinq objets. Ces cinq objets gestionnaires ont pour but de séparer précisément les fonctionnalités du système d'invocation de méthodes à distance puisque différentes versions d'un gestionnaire implantent différentes politiques. Ces différents gestionnaires doivent être clairement définis pour qu'ils soient interchangeables les uns indépendamment des autres.

L'objet serveur correspond à l'objet qui est en charge d'exécuter les requêtes et donc de rendre le service au client, c'est-à-dire l'objet qui implante la sémantique du serveur ;

Le gestionnaire d'estampilles contrôle la cohérence et les synchronisations des objets serveurs des différentes copies au moyen des estampilles d'ordonnancement. Il fournit donc la politique de cohérence et de synchronisation ;

Le gestionnaire d'ordonnancement se charge d'ordonner les invocations en fonction de leurs estampilles. Il respecte les principes que nous avons détaillés au chapitre 3 page 53. Il fournit en plus une politique de priorité de livraison, pour décider parmi plusieurs invocations prêtes à être exécutées, laquelle le sera en premier ;

Le gestionnaire de propagation permet de propager une invocation à toutes les autres copies d'un serveur. Il doit aussi assurer les communications entre le client et le serveur. Il encapsule la politique de dissémination, c'est-à-dire où, à qui, quand et comment propager une invocation ;

L'objet contrôle assure la politique d'exécution (par la création ou l'utilisation d'activités par exemple), la politique de propagation « à la demande » (invocation transmise sur la demande des copies des serveurs, et non spontanément à toutes les copies) et enfin la politique de retransmission des invocations et des accusés de réception.

Le client s'adresse à un objet serveur par l'intermédiaire d'un objet mandataire (*proxy*). Ce mandataire est constitué lui-même d'un mandataire de l'objet serveur, du gestionnaire d'estampilles et de l'objet contrôle. Nous désignons sous le terme serveur l'ensemble des copies du serveur et de ses sous-objets.

4.1.2 Interactions client / serveur

Un objet client invoque une méthode d'un serveur d'une manière transparente, c'est-à-dire comme une invocation d'une méthode d'un objet local. Cette invocation est prise en charge par le mandataire local du serveur. Ce mandataire peut, par exemple, être résidant dans l'exécutable du client par l'utilisation d'une librairie générée automatiquement par un compilateur d'IDL.

L'objet mandataire prend alors en charge tous les mécanismes jusqu'à l'obtention d'une réponse, ou d'une erreur. Il transmet l'invocation à une copie du serveur que nous appelons une copie primaire pour cette invocation. Une copie est désignée comme primaire à chaque invocation, elle peut changer à tout instant, pour des raisons de performances et de disponibilités.

Quand une copie du serveur reçoit une invocation d'un mandataire, elle la délivre localement à un gestionnaire d'estampilles qui lui applique un modèle de cohérence ou de synchronisation particulier. L'invocation est alors propagée à toutes les copies s'il s'agit d'une en écriture, ou à un sous-ensemble s'il s'agit d'une invocation en lecture. Elle est délivrée à un gestionnaire d'ordonnancement, qui la mettra en ordre en fonction de ses estampilles et d'une politique de priorité.

Il existe un sous-ensemble des copies qui sont dites synchrones, c'est-à-dire qui bloquent le déroulement des opérations. Si l'invocation est en écriture, les copies synchrones bloquent le client, tant qu'elles n'ont pas exécuté l'invocation. Si l'invocation est en lecture, les copies synchrones bloquent le gestionnaire d'estampilles, qui ne peut pas utiliser cet identificateur tant qu'il n'est pas délivré aux copies synchrones. Il s'agit de garantir le principe 3.8 page 68 sur la condition d'utilisation des estampilles.

L'invocation, une fois prête sur une copie, est exécutée par l'objet serveur local, et les résultats renvoyés au mandataire. Celui-ci propage une réponse au client. Les autres copies n'ont pas encore forcément exécuté cette invocation. Seul le strict minimum des copies requises pour assurer la fiabilité sont synchrones avec le client, toutes les autres étant asynchrones.

Nous utiliserons les conventions graphiques décrites à la figure 4.1 page suivante, pour représenter les mandataires, les copies primaires, synchrones et secondaires, ainsi que les différents gestionnaires.

4.2 Rôle des gestionnaires

Cette section décrit le rôle des différents gestionnaires. Ainsi, nous décrivons quelques caractéristiques de la plate-forme sous-jacente (§ 4.2.1) puis l'objet serveur (§ 4.2.2), l'objet contrôle (§ 4.2.3), le gestionnaire d'estampilles et ses structures (§ 4.2.4), le gestionnaire d'ordonnancement (§ 4.2.5) et de propagation (§ 4.2.6). Nous présentons l'interface de chaque objet au travers d'une IDL. Celle-ci nous aidera à comprendre les interactions que nous détaillerons aux sections suivantes.

4.2.1 Plate-forme sous-jacente

Pour décrire les interfaces de différents gestionnaires, nous devons présenter quelques données dépendantes de la plate-forme sous-jacente. Il s'agit principalement d'un type `oid`, c'est-à-dire d'un identificateur unique d'objet. Nous avons besoin d'un identificateur unique, au moins dans l'espace, qui identifie chaque copie d'un objet serveur différemment. L'unicité dans le temps n'est pas indispensable.

Enfin, nous avons besoin d'une part d'un type `timeval` pour décrire un intervalle de temps, et d'autre part, d'une classe `propagMsg`, qui est utilisée (par héritage de classe) pour représenter des classes « sérialisables », c'est-à-dire que l'on peut exporter et importer à partir d'un tampon mémoire.

L'interface 4.1 résume ces quelques structures et classes.

Interface 4.1 Divers

```
interface oid {};

struct timeval {
  long   sec;
  long  usec;  };

const timeval timevalEvent   =  { 0,  0};
const timeval timevalNoDelay =  {-1, -1};

interface propagMsg : serialization
{
  // Externalization or Import/Export methods
};
```

4.2.2 Objet serveur

L'objet serveur est l'objet qui représente la sémantique du service. C'est donc l'objet qui exécute les invocations en fonction des paramètres et qui détermine les réponses. L'interface 4.2

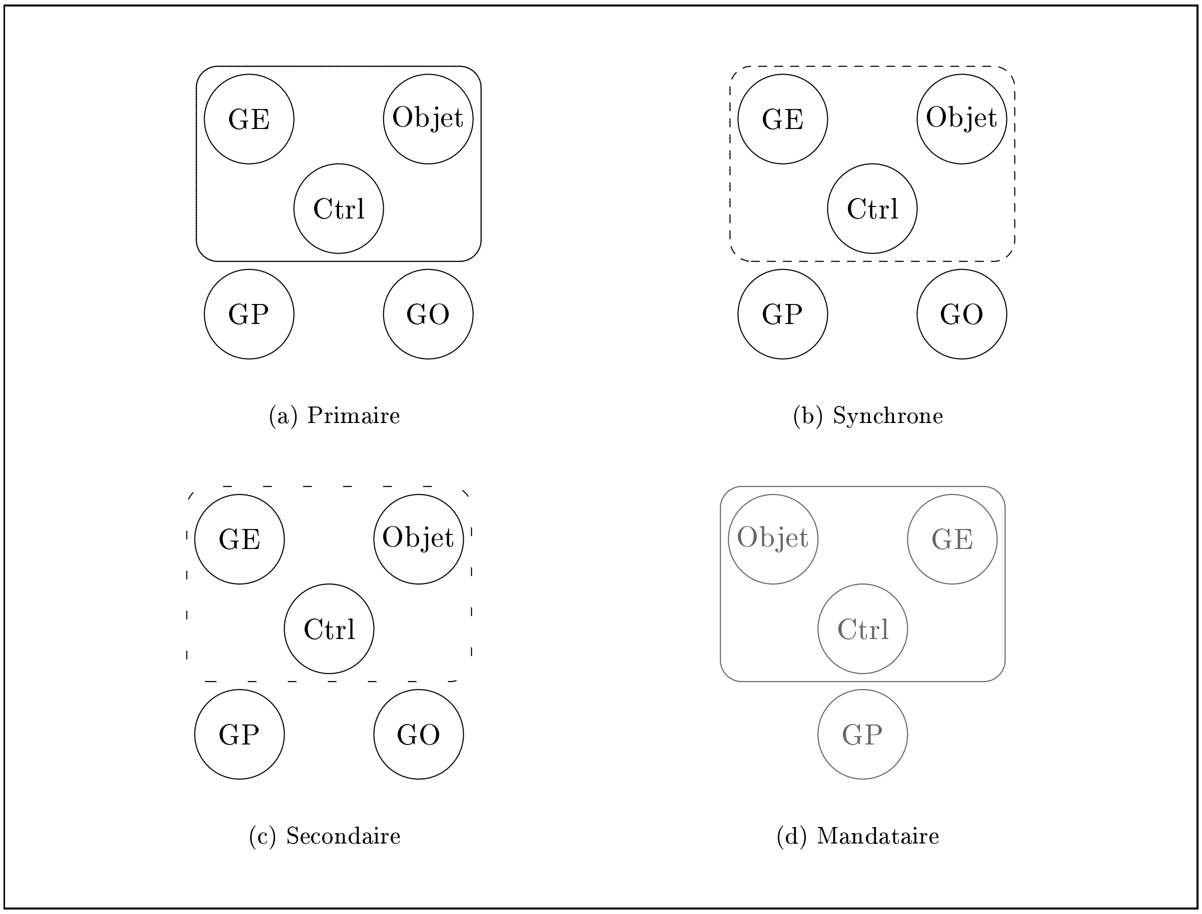


FIG. 4.1 – Type d'une copie d'un serveur

présente cet objet.

Interface 4.2 Objet serveur

```
#include "misc.idl"

enum invType { invRead,
               invWrite,
               invDefered };

interface objectQuery : propagMsg
{
        invType  getType  ();

  unsigned long  nbrIn    ();
        boolean  getIn    (in unsigned long index, out any p);
};

interface objectReply : propagMsg
{
  unsigned long  nbrOut   ();
        boolean  getOut   (in unsigned long index, out any p);
};

interface object : propagMsg
{
  // From Object
     void  tryType   (inout objectQuery query);
  boolean  doCall    (in    objectQuery query, out objectReply reply);

     void  getState  (  out propagMsg   private);
     void  setState  (in    propagMsg   private);
};
```

Dans notre cadre de travail, cet objet, coté serveur, contient en plus un squelette pour exécuter une invocation emballée. Symétriquement, coté mandataire, cet objet possède un talon, pour emballer les paramètres. Il ne s'occupe pas de l'aspect distant, qui lui est complètement caché. Son seul rôle est de fournir l'hétérogénéité des paramètres, le type d'une invocation et des fonctions d'importation et d'exportation de son état. Lui seul peut fournir ces services, car ils sont dépendants du type des paramètres, de la sémantique de ses méthodes et de la structure de ses données internes.

Ainsi, le développeur d'un serveur doit concevoir l'objet en fournissant une implantation, comme il l'aurait fait pour un objet appartenant au client. Il doit donner l'interface qu'il présente au client en fournissant une IDL de son serveur, comme il l'aurait fait pour un service distribué classique. Il doit, en plus, implanter des méthodes permettant d'importer et d'exporter son état et de retourner le type d'une invocation en fonction de ses paramètres.

Un compilateur d'IDL génère alors le talon et le squelette respectivement dans l'objet mandataire et l'objet serveur. Ce compilateur peut essayer de générer automatiquement les méthodes pour déterminer le type des invocations (par l'utilisation de mots clefs dans l'IDL du service par exemple), ainsi que celles important et exportant l'état. Néanmoins, le développeur du service peut facilement concevoir et optimiser ces méthodes.

4.2.3 Objet contrôle

L'objet contrôle fait office de coordinateur. Il contient plusieurs types de politiques dépendantes de l'application, dont certaines doivent être coordonnées du coté serveur et du coté mandataire. L'interface 4.3 décrit l'objet contrôle et son mandataire.

Interface 4.3 Objet contrôle

```
#include "object.idl"
#include "misc.idl"

interface controlProxy
{
  // From Object Proxy
  void  doCall    (in objectQuery query, out objectReply reply);
};

interface controlManager
{
  // From Object
  void  runServer (in timeval timeout);

  // From Delivery Manager
  void  dupObject (out propagMsg private);
};
```

Cet objet est responsable de trois politiques différentes :

1. La politique de retransmission : il peut demander la réémission d'une invocation en attente dont il considère que la réponse tarde à arriver. Pour cela, il pilote le gestionnaire de propagation pour l'aider à assurer la fiabilité. En effet, le temps d'attente est un paramètre dépendant de l'application. Par exemple, un serveur de calcul intensif demande une temporisation plus importante qu'un serveur fournissant l'heure. Nous verrons qu'une réémission d'une invocation est une chose relativement coûteuse. Il faut donc que la temporisation soit choisie d'une manière adéquate à la fois par l'objet contrôle et son mandataire. Ce choix peut éventuellement être dynamique en observant les temps de transmission usuels. Ainsi, ils peuvent minimiser les accusés de réception, et les retransmissions inutiles d'invocations ;
2. La politique de propagation « à la demande » : avec une telle méthode, ce sont les copies des serveurs qui demandent la réception d'une invocation particulière et non le gestionnaire de propagation qui les propage spontanément à toutes les copies. L'objet contrôle décide quelles sont les invocations qui sont transmises par un tel mécanisme, et quand les demander ;
3. La politique d'exécution : il se charge de faire exécuter les invocations prêtes à l'objet serveur. Pour cela, il peut avoir une politique d'exécution concurrente au moyen d'activités (en les créant dynamiquement sur demande ou bien en utilisant un stock) ou une politique mono-activité.

4.2.4 Gestionnaire d'estampilles

Le gestionnaire d'estampilles fournit la politique de cohérence et de synchronisation. Il est constitué d'une partie résidant dans la copie du serveur, et d'un mandataire coté client. Son interface est décrite dans l'IDL 4.4 page suivante.

Nous ne revenons pas sur ce gestionnaire, puisque pour garantir une cohérence il utilise les estampilles d'ordonnancement que nous avons décrites au chapitre 3 page 53.

N'étant pas le seul à manipuler les identificateurs et estampilles, ce gestionnaire doit fournir une interface aux autres gestionnaires pour accéder à ces informations. L'objet de type `deliveryMsg` remplit ce rôle, une instance est associée à chaque invocation. Son interface est constituée d'une méthode pour accéder aux informations concernant son invocation associée (`getIss`). Il fournit aussi les méthodes `firstPiggy` et `nextPiggy` pour accéder à des informations concernant des identificateurs en surcharge, c'est-à-dire ne concernant pas directement l'invocation associée.

Les estampilles d'ordonnancement sont représentées par le type `iss`, décrit par l'interface 4.5 page 93. Il permet de connaître l'identificateur par `getId`, le cardinal et le degré de l'identificateur [1] (`getCard` et `getDegIn`) ainsi que les estampilles d'ordonnancement de cet identificateur (`firstStamp` et `nextStamp`). Ce processus est récursif. Des méthodes permettent de manipuler l'identificateur, le cardinal et le degré.

Un identificateur est représenté par l'objet `issId`. Il est opaque pour tous les gestionnaires sauf celui des estampilles. Il est généré de façon unique par chaque gestionnaire d'estampilles comme bon lui semble. Le gestionnaire peut utiliser l'identificateur d'objet (`oid`) de la copie comme base pour l'unicité dans l'espace. Les autres gestionnaires peuvent seulement tester l'égalité de deux identificateurs ou son appartenance à un ensemble (en utilisant un masque).

4.2.5 Gestionnaire d'ordonnancement

Le gestionnaire d'ordonnancement est un objet qui est logiquement rattaché à un site particulier. Il renferme la politique d'ordonnancement des invocations. L'interface 4.6 page 94 le décrit.

Il ordonne les invocations en fonction des identificateurs et des estampilles d'ordonnancement, et donc respecte les principes que nous avons vus au chapitre 3 page 53. De plus, entre plusieurs invocations prêtes à être exécutées, il peut donner une priorité relative en fonction de la politique qu'il gère.

Ainsi, un gestionnaire d'ordonnancement peut favoriser les invocations de certains clients en fonction de leurs localisations ou de tout autre critère. Par exemple, il peut délivrer parmi toutes les invocations prêtes à être exécutés, celles issues des clients de la même machine, puis du même réseau local et enfin de réseaux précis. Néanmoins, il doit garantir l'absence de famine, c'est-à-dire des invocations prêtes qui ne seront jamais exécutées, et une équité compatible avec sa politique.

Il gère donc l'utilisation de l'objet serveur de la copie locale, et donc indirectement l'utilisation du processeur. C'est pour cela que ce gestionnaire est logiquement géré par le site, et non par l'application. Il est donc situé sur au moins chaque site où une copie d'un serveur existe. Il est exclusivement local, c'est-à-dire sans mandataire ni communication entre sites.

1. Nous détaillons le rôle du degré dans le chapitre 5 page 119.

Interface 4.4 Gestionnaire d'estampilles

```
#include "delivery.idl"
#include "object.idl"
#include "propag.idl"
#include "misc.idl"
#include "iss.idl"

interface deliveryMsg : propagMsg {};
{
         void  getIss      (  out iss id);

      boolean  firstPiggy  (  out iss id);
      boolean  nextPiggy   (inout iss id);
};

enum stampRet { stampNo,
                stampPartial,
                stampDone   };

interface stampProxy
{
  // From Control Proxy
  stampRet  tryStamp (inout deliveryMsg msg,   out propagMsg private,
                      in    objectQuery inv);
      void  update   (in    deliveryMsg msg, in    propagMsg private);

      void  getState (  out propagMsg   private);
      void  setState (in    propagMsg   private);
};

interface stampManager
{
  // From Control Manager
      void  doStamp (inout deliveryMsg msg, inout propagMsg private,
                     in    objectQuery inv, inout boolean  rexmit);

  // From Delivery Manager
      void  freeId (in propagHdl hdl, in sequence<issUid> ids, int nbr);

  // From Propagation Manager
      void  freeId        (in propagMsg ids);
      void  notifSync     (in issUid    id);
      void  notifFlush    (in issUid    id, in issUid mask);
      void  excludeMember (in sequence<oid> members);
};
```

Interface 4.5 Estampilles d'ordonnancement

```
interface issUid
{
        boolean  equal  (in issUid id);
        boolean  equal  (in issUid id, in issUid mask);
};

interface issCard
{
     void  dec     ();
     void  add     (in Card c);

  boolean  isNull  ();
};

interface issDegIn
{
     void  dec     ();
     void  add     (in issDegIn d);

  boolean  isNull  ();
};

interface iss
{
         void  getId       (out issUid id);
      boolean  getCard     (out issCard card);
      boolean  getDegIn    (out issDegIn deg);

      boolean  firstStamp  (  out iss id);
      boolean  nextStamp   (inout iss id);
}
```

Interface 4.6 Gestionnaire d'ordonnancement

```
#include "control.idl"
#include "misc.idl"
#include "iss.idl"

interface deliveryHdl {};

struct deliveryStat {
  unsigned long  nbrExec;
  unsigned long  nbrReady;
  unsigned long  nbrWait;
  unsigned long  nbrAbort; };

interface deliveryManager
{
  // From Propagation Manager
  boolean  queueInv        (in deliveryMsg msg,  in propagHdl phdl,
                            in invType     type, in int       flw,
                            in boolean   sync                    );
     void  notifFlush      (in issUid      id,   in issUid    mask);

  // From Control Manager
  boolean  unqueueReady    (  out deliveryHdl dhdl,   out propagHdl phdl,
                                                    in    timeval   timeout);
  boolean  unqueueWaiting  (  out issUid      id,   in    timeval   timeout);
     void  notifFinish     (in    deliveryHdl dhdl, in    boolean   done);
  boolean  isEmpty         ();

     void  stopNew         ();
     void  getState        (  out propagMsg private);
  boolean  setState        (in    propagMsg private);

  // From Other
     void  getStat (in oid obj, out deliveryStat snap, out delievryStat total);
};
```

4.2.6 Gestionnaire de propagation

Le gestionnaire de propagation doit assurer la diffusion des invocations d'une façon fiable et sans duplication. Presqu'aucune contrainte sur l'ordre de livraison des invocations ne lui est demandée, puisque cet ordre est régi par les estampilles d'ordonnancement. Ce gestionnaire est décrit par l'interface 4.7 page suivante.

Il localise et sélectionne les copies des serveurs à qui il propage « à la longue », de manière k-fiable (gérant $k-1$ sites défaillants), sans duplication et sans ordre, une invocation. Il doit assurer un minimum d'ordre dans la livraison des messages par l'intermédiaire de barrières d'invocations (cf. § 4.4.1). Il gère aussi la communication entre le mandataire du serveur coté client et la copie primaire (cf. § 4.5.1). Il peut gérer des politiques de propagation « à la demande » (cf. § 4.3.4). Enfin, il collecte, d'une manière efficace, le bon nombre de résultats et d'accusés de réception.

Il implante comme bon lui semble ce contrat, et peut exploiter son environnement pour optimiser ces communications. Il est généralement dépendant du site, puisqu'il utilise directement les ressources réseaux de celui-ci, et qu'il peut ainsi exploiter la topologie et les caractéristiques des réseaux auxquels il a accès. Ainsi, sur un réseau local, il peut utiliser le *multicast* IP [Deering, 1989]. Dans un environnement déconnectable, il peut utiliser un protocole communiquant par paires de machines comme l'« anti-entropie » du projet Bayou [Petersen et al., 1997] par exemple. Sur des nœuds d'interconnexion de réseaux, ou de portables, il peut utiliser une journalisation sur disque des invocations comme dans [Holbrook et al., 1995]. La gestion des suspecteurs de fautes est aussi dépendante de l'environnement, les temporisateurs n'ayant pas forcément les mêmes valeurs. Il peut optimiser la collecte des résultats grâce à des caractéristiques de son protocole de communication.

4.3 Parcours d'une invocation

Dans cette section, nous approfondissons la description du déroulement d'une invocation que nous avons brièvement présentée au § 4.1.2. Pour cela, nous décrivons quatre étapes successives : du client à la copie primaire (§ 4.3.1), de la copie primaire aux autres copies (§ 4.3.2), des copies synchrones à la copie primaire (§ 4.3.3), de la politique de propagation « à la demande » (§ 4.3.4) et enfin de l'exécution et du retour au client des résultats (§ 4.3.5). Dans chacune de ces étapes, quand cela est pertinent, nous décrivons les répercussions de ce cadre de travail sur la performance et la tolérance aux fautes, et nous résumons ces interactions par une figure.

4.3.1 Du client à la copie primaire du serveur

Quand un client invoque une méthode du serveur, le mandataire prend en charge cette invocation par l'intermédiaire du mandataire de l'objet serveur. Par l'intermédiaire des méthodes que lui a fournies le développeur du service, il tente de déterminer le type de l'invocation, c'est-à-dire de type lecture (`invRead`), de type écriture (`invWrite`) ou de type inconnu (`invDefered`). Généralement, il peut déterminer le type des méthodes qui ne dépend pas de l'état de l'objet serveur. Puis il emballe les paramètres dans un objet de type `objectQuery` et transmet cette invocation au mandataire de l'objet contrôle avec sa méthode `doCall`.

Ce dernier, en utilisant la méthode `tryStamp` du mandataire du gestionnaire d'estampilles, essaye d'identifier et d'estampiller localement cette invocation. Pour cela, en fonction de la

Interface 4.7 Gestionnaire de propagation

```
#include "object.idl"
#include "misc.idl"

enum propagRet { propagNone,     propagMayBe,
                 propagPartial,  propagSure,
                 propagDone,     propagError,
                 propagReXmit };

struct propagArg {
        boolean  propag;
        boolean  rexmit;
        boolean  pull;
  unsigned long  nbrReply;
  unsigned long  nbrSync;  };

struct propagStatus {
  unsigned long  nbrReply;
  unsigned long  nbrSync;  };

interface propagHdl {};

interface propagManager
{
  // From Control Proxy
  propagRet  remoteCall (inout propagHdl    hdl, in    propagArg arg,
                         inout deliveryMsg  dm,  inout propagMsg msg,
                           out propagStatus s,     out sequence<objectReply> rep,
                         in    timeval      timeout);
       void  cancelCall (in    propagHdl    hdl);

  // From Control Manager
  propagRet  receiveQuery (  out propagHdl hdl,   out deliveryMsg dm,
                             out propagMsg msg, in    timeval     timeout);
       void  propagQuery  (inout propagHdl hdl,   out deliveryMsg dm,
                           in    propagMsg msg, in    propagArg   arg);
       void  pullQuery    (in    issUid    id);
       void  sendAck      (inout propagHdl hdl, in deliveryMsg dm,
                           in    propagMsg msg);
       void  sendReply    (in    propagHdl hdl, in deliveryMsg dm,
                           in    propagMsg msg, in objectReply reply);
    boolean  newObject    (  out propagMsg state);

  // From Stamp Manager or Stamp Proxy
       void  sendFreeId    (in propagHdl hdl, in propagMsg ids);
       void  sendFlush     (in issUid    id,  in issUid    mask);
       void  excludeMember (in sequence<oid> members);

  // From Delivery Manager
       void  abortInv      (in    propagHdl hdl);
};
```

méthode invoquée, des paramètres de celles-ci et de la politique fournie, ce dernier retourne `stampDone` s'il a pu le faire complètement, `stampPartial` s'il n'a pu le faire que partiellement et enfin `stampNo` s'il n'a pas pu le faire du tout.

À ce stade, l'invocation peut être transmise aux copies du serveur par l'intermédiaire de la méthode `remoteCall` du gestionnaire de propagation. Deux cas sont possibles :

1. L'invocation est de type écriture, identifiée et estampillée complètement dans le mandataire : le mandataire peut alors propager directement l'invocation en utilisant le paramètre `propag` de `remoteCall`. Dans ce cas, cela revient à considérer le mandataire comme la copie primaire. Cette situation est décrite à la figure 4.2 page suivante, et le déroulement des interactions au § 4.3.2 page 102 ;
2. Dans tous les autres cas, il doit passer par une copie primaire, et donc invoquer la méthode `remoteCall` avec le paramètre `propag` à faux. La figure 4.3 page 99 résume cette situation.

L'objet contrôle transmet un message, qui est constitué des paramètres de l'invocation (de type `propagMsg`), éventuellement de l'identificateur et des estampilles (de type `deliveryMsg`) ainsi qu'une valeur de temporisation et des paramètres de propagations (de type `propagArg`). Ces derniers sont :

propag : indique si l'invocation doit être propagée à toutes les copies, ou bien délivrée à une seule, la copie primaire ;

rexmit : demande de réémission d'une invocation déjà émise ;

pull : permet de spécifier une politique de propagation « à la demande », c'est-à-dire que le gestionnaire de propagation propage cette invocation sur la demande d'une copie du serveur, et non de manière spontanée à toutes les copies ;

nbrReply : spécifie le nombre de réponses issues de différentes copies que le gestionnaire doit renvoyer. Le nombre de réponses effectivement reçues est renvoyé dans `propagStatus` ;

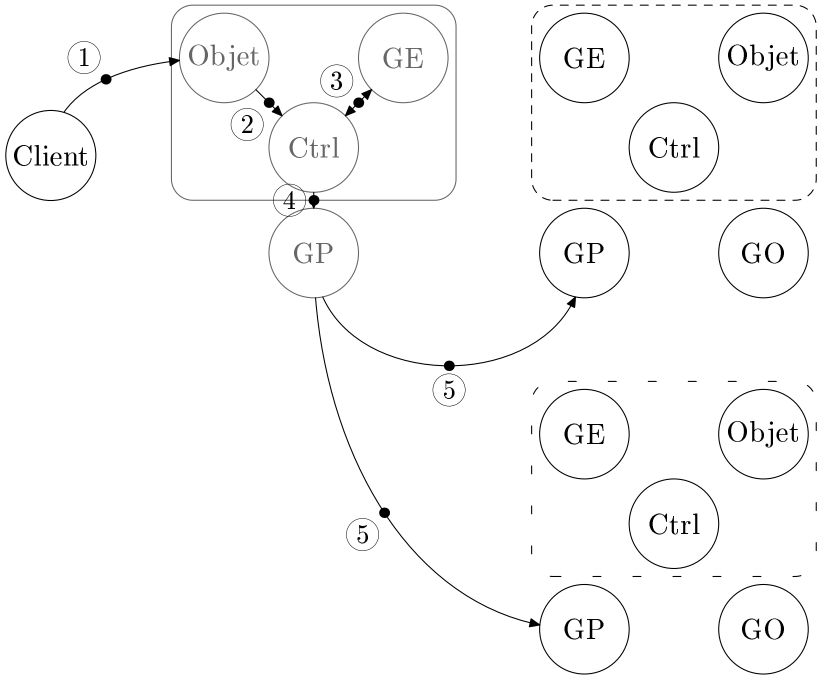
nbrSync : indique le nombre de copies du serveur qui doivent être synchrones. Pour une invocation en écriture, cette valeur indique le nombre minimal de copies devant exécuter cette invocation avant qu'elle soit considérée exécutée. Pour une invocation en lecture, cette valeur indique le nombre minimal de copies devant recevoir l'invocation avant que son identificateur soit réutilisé. Ici aussi, le nombre effectif de copies synchrones est renvoyé dans `propagStatus`.

La valeur du temporisateur est la durée maximal d'exécution de la méthode. Elle peut être infinie avec la valeur prédéfinie `timevalNoDelay` ou bien sur événement avec `timevalEvent`. Dans ce dernier cas, la méthode se termine à chaque changement de statut que renvoie la méthode `remoteCall`. L'invocation peut alors être abandonnée par `abortCall`. Les différents statuts possibles sont :

propagNone : Ce statut indique, d'une façon sûre et certaine, que l'invocation n'est pas arrivée à destination ;

propagMayBe : Dans ce cas, l'invocation est peut-être arrivée, identifiée et estampillée sur la copie primaire du serveur. Aucune garantie n'est fournie ;

propagSure : Ici, il y a la garantie que l'invocation est arrivée, identifiée et estampillées par la copie primaire du serveur. L'objet de type `deliveryMsg` est mis à jour, c'est-à-dire que l'on a, du coté mandataire, la connaissance de son identificateur et de ses estampilles. Néanmoins, aucune réponse n'a été encore reçue ;



① Le client invoque la méthode sur l'objet mandataire ;

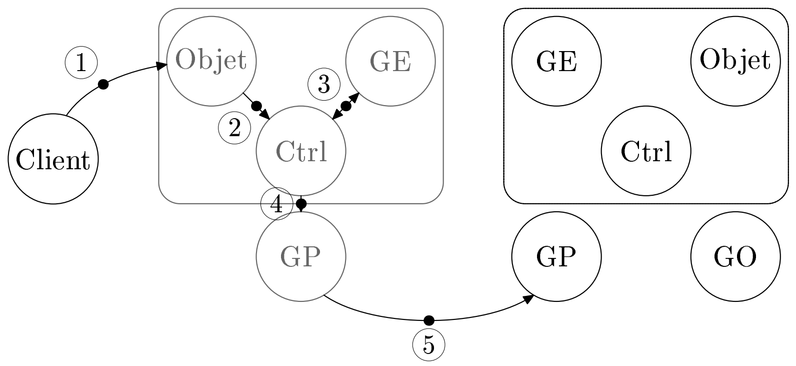
② L'objet mandataire détermine son type (écriture) et emballe les paramètres ;

③ L'objet contrôle la fournit au gestionnaire d'estampilles pour identifications et estampillages (effectués complètement) ;

④ Il appelle le gestionnaire de propagation pour diffusion de l'invocation à toutes les copies ;

⑤ Enfin, l'invocation est transmise à toutes les copies localisées par le gestionnaire de propagation.

Fig. 4.2 – Du client aux copies du serveur (sans primaire)



① Le client invoque la méthode sur l'objet mandataire ;

② L'objet mandataire détermine son type et emballe les paramètres ;

③ L'objet contrôle la fournit au gestionnaire d'estampilles pour identifications et estampillages éventuelles ;

④ Il appelle le gestionnaire de propagation pour transmission de l'invocation à une copie primaire ;

⑤ Enfin, l'invocation est transmise à une copie primaire choisie par le gestionnaire de propagation.

Fig. 4.3 – Du client à la copie primaire du serveur

propagPartial : La situation est identique à la précédente, mais au moins une réponse a été reçue. Par contre le nombre de réponses et le nombre de copies synchrones attendues ne sont pas encore atteints ;

propagError : le nombre attendu de réponses et/ou le nombre voulu de copies synchrones ne peut être atteints. L'appel est terminé ;

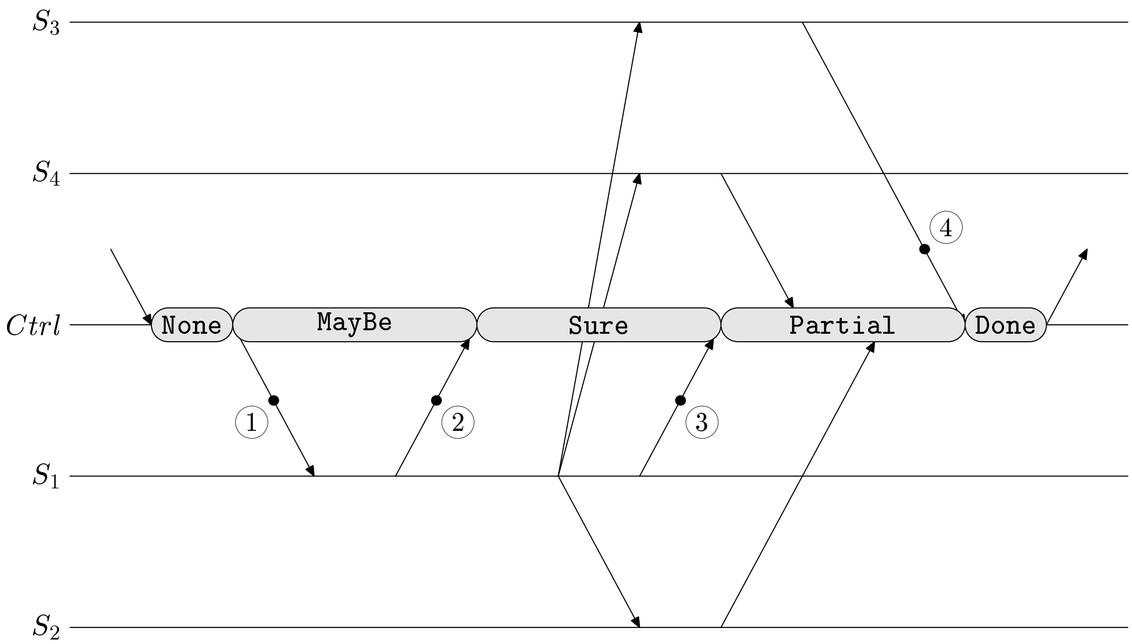
propagDone : Enfin, ce statut indique que l'on a reçu le nombre attendu de réponses et qu'il y a le nombre voulu de copies synchrones, et donc que l'appel est terminé du coté du mandataire.

Pour illustrer ces différents statuts, nous proposons à la figure 4.4 page suivante un exemple de scénario possible d'une invocation de méthode à distance.

C'est le gestionnaire de propagation qui assure la localisation des copies du serveur. C'est lui aussi qui choisit la copie primaire la plus susceptible d'être rapide vis-à-vis du temps de réponse au client. Il enferme donc des mécanismes de localisation et de sélection des copies.

4.3.1.1 Performance

La performance est obtenue par l'utilisation du schéma « *read-any / write-any* ». Il permet au gestionnaire de propagation de changer de copie primaire quand bon lui semble ou de réémettre l'invocation quand la réponse tarde à venir du point de vue de l'objet contrôle (à cause d'une copie ou d'un réseau trop lent ou bien d'une défaillance). Néanmoins, cette réémission ne peut se faire sans problème que si le statut de `remoteCall` n'est pas `propagMayBe`. Dans ce cas, soit l'invocation n'a pas encore été transmise, nous sommes alors à la phase initiale,



L'objet contrôle ($Ctrl$) représente l'utilisateur de la méthode `remoteCall`. Cette méthode est invoquée avec `propag`, `rexmit` et `pull` à faux, c'est-à-dire que c'est une invocation à destination d'une copie primaire, sans réémission et en mode de propagation spontanée. Le nombre de réponses attendues est de 2 et le nombre de copies synchrones est de 4. Il y a aux indices :

① À partir de l'instant où l'invocation est émise, l'émetteur ne peut savoir si elle est bien arrivée, le statut est donc `propagMayBe` ;

② Dès que l'émetteur reçoit un accusé de réception de la copie primaire, le statut passe à `propagSure`, qui indique qu'il est certain qu'elle a été reçue, identifiée et estampillée ;

③ Quand l'émetteur reçoit la première réponse, le statut est `propagPartial`, puisque l'invocation aura été exécutée sur un nombre partiel de copies du serveur ;

④ Enfin, quand l'émetteur a reçu toutes les réponses et tous les accusés de synchronisation demandés, le statut passe à `propagDone` puisque l'invocation de la méthode est terminée du coté mandataire.

Fig. 4.4 – Statut d'un appel de méthode à distance

soit elle l'a été, mais le mandataire a connaissance de l'identificateur et des estampilles de l'invocation, et peut donc la réémettre sans problème. Nous rappelons que le gestionnaire de propagation garantit qu'il n'y a pas de duplication d'invocations. Nous discutons de la phase concernant le statut `propagMayBe` dans la section suivante.

Le choix de la copie primaire est faite par le gestionnaire de propagation d'une façon qui lui est propre. Soit il utilise une méthode triviale comme celle qui consiste à utiliser une copie locale si elle existe, sinon une copie distante quelconque. Soit il peut utiliser l'aide d'un gestionnaire de répartition de charge permettant de maximiser les performances en choisissant une copie primaire peu chargée et/ou dont les communications sont rapides.

4.3.1.2 Tolérance aux fautes

Nous pouvons séparer la tolérance aux fautes en deux parties distinctes. La première, vis-à-vis du client, contrôle comment celui-ci peut avoir la garantie que ces invocations ne seront pas perdues à la suite d'une défaillance. La deuxième décrit comment le mandataire, et les gestionnaires d'estampilles, peuvent interagir quand l'on suspecte la défaillance d'une copie primaire.

a) Perte d'invocations

La tolérance aux fautes est essentiellement assurée par le paramètre `nbrSync` de la méthode `remoteCall` et par les propriétés du gestionnaire de propagation. La communication doit être k-fiable, c'est-à-dire résistance à $k-1$ défaillances, avec $k = \texttt{nbrSync}$.

Pour une invocation en écriture, `nbrSync` indique le nombre de serveurs qui doivent avoir reçu et exécuté l'invocation avant que l'appel ne soit terminé. Ainsi, avec un nombre k, le client a la garantie que k copies du serveur ont exécuté son invocation, et donc que même si $k-1$ copies sont défaillantes, ses modifications ne seront pas perdues.

Pour une invocation en lecture, `nbrSync` indique le nombre de serveurs qui doivent avoir reçu (sans exécution, mais transmise au gestionnaire d'ordonnancement) l'invocation avant que le gestionnaire d'estampilles puisse utiliser son identificateur comme estampille. Ainsi, avec un nombre k, le client a la garantie que k copies du serveur auront enregistré son invocation, et qu'au moins sur ces k copies son invocation ne sera pas invalidée (cf. principe 3.8 page 68). Donc même si $k-1$ copies sont défaillantes, il aura une réponse du serveur.

b) Retransmission

Le problème majeur se situe quand `remoteCall` a le statut `propagMayBe`, c'est-à-dire quand le client ne sait pas si son invocation a été reçue par la copie primaire. Nous rappelons que dans un système asynchrone, on ne peut pas distinguer un processus ou un réseau lent, d'une panne (§ 1.3.2 page 7). Pendant cette période, on ne peut donc pas savoir si l'invocation a été reçue, identifiée, estampillée et propagée aux autres copies. Si c'est le cas, et que nous réémettons cette invocation, elle sera identifiée par deux identificateurs différents et estampillés différemment.

En cas de réémission durant la période où le statut est `propagMayBe`, le gestionnaire de propagation en est informé par le paramètre `rexmit`, et essayera de contacter lui même la copie primaire pour obtenir l'identificateur et les estampilles. En cas d'échec, il choisit une nouvelle copie primaire et en informe le gestionnaire d'estampilles par le paramètre `rexmit`. En effet, seul le gestionnaire d'estampilles peut résoudre efficacement ce problème, car la solution dépend de la politique de cohérence et de synchronisation.

Par exemple, dans le cas d'une cohérence « à la longue » (§ 2.3.1.6 page 34), c'est-à-dire sans ordre sur les invocations, il est clair qu'il suffit de réémettre l'invocation sans aucun autre mécanisme. Par contre, pour une cohérence causale (§ 2.3.1.3 page 30), une solution est de réémettre l'invocation en créant une partition entre les copies ayant reçut la première invocation, et les autres. C'est pour cela que la méthode `excludeMember` permet au gestionnaire d'estampilles d'exclure des copies.

Ce choix, le gestionnaire d'estampilles l'indique par l'intermédiaire du paramètre `rexmit`, qui est recopié par l'objet contrôle dans le paramètre du même nom de `propagQuery`. Si `rexmit` est vrai, cela correspond à la demande d'une partition entre les copies, sinon, cette invocation est traitée normalement. La partition des copies entre un groupe ayant reçu celles émise par la première copie primaire, et un autre ayant reçu celles émise par la deuxième copie primaire est à la charge du gestionnaire de propagation.

4.3.2 De la copie primaire aux autres copies du serveur

Dans cette section, nous nous intéressons à la propagation de l'invocation de la copie primaire jusqu'aux autres copies. L'objet contrôle de chaque copie demande une invocation par la méthode `receiveQuery`, durant un laps de temps spécifié par le `timeout`. Il s'agit d'un message de type `propagMsg` et un objet de type `deliveryMsg`. Cette méthode renvoie `propagNone` si aucune invocation n'est reçue, `propagDone` dans le cas contraire ou bien `propagReXmit` si l'invocation reçue a déjà été envoyée, et a donc peut-être déjà été identifiée et estampillée par une autre copie primaire.

Si l'invocation reçue est de type inconnu, l'objet contrôle peut obtenir son type par l'intermédiaire de l'objet serveur et de sa méthode `tryType` qui complète `objectQuery`. Si l'invocation n'est pas entièrement identifiée et estampillée, alors l'objet contrôle demande au gestionnaire d'estampilles, par sa méthode `doStamp`, de compléter ou de déterminer l'identificateur et les estampilles de cette invocation. Ce dernier le fait en fonction de la méthode invoquée, des paramètres de celle ci (`objectQuery`), des paramètres privés du mandataire (`stampMsg`), du pré-estampillage (`deliveryMsg`) par le mandataire et de la politique fournie par le gestionnaire. Par cette même méthode, le gestionnaire d'estampilles est informé des paramètres privés que lui communique son mandataire (`private`). Il peut mettre à jour cet objet pour, à son tour, lui faire parvenir des paramètres privés. Nous reviendrons sur cet échange au § 4.5.1 page 115.

L'objet contrôle de la copie primaire a donc en sa possession une invocation entièrement identifiée et estampillée et de type connu. Il la diffuse alors aux autres copies en utilisant la méthode `propagQuery`. Le gestionnaire de propagation doit garantir que cette invocation arrivera une seule fois aux copies pertinentes, tôt ou tard. Pour cela, il peut utiliser n'importe quel protocole lui convenant.

Chaque gestionnaire de propagation fait suivre les invocations identifiées et estampillées aux gestionnaires d'ordonnancement par la méthode `queueInv`, en indiquant le type de l'invocation et si le site fait partie des copies synchrones (par le paramètre `sync`). Ce denier lui indique en retour si oui ou non l'invocation sera exécutée sur ce site (l'identificateur pouvant être invalide). Cela permet au gestionnaire de propagation de savoir si cette copie peut convenir comme copie synchrone.

La figure 4.5 page suivante résume les interactions qui se passent durant l'acheminement d'une invocation d'une copie primaire vers les autres copies.

4.3.2.1 Performance

Le paramètre `propag` de `propagQuery` indique si cette invocation doit atteindre toutes les copies, ou seulement un nombre correspondant au minimum à `nbrSync`. Concrètement, l'objet contrôle met à jour ce paramètre en fonction du type de l'invocation, c'est-à-dire qu'il demande la propagation à toutes les copies seulement si c'est une invocation en écriture.

Cette caractéristique, couplée aux principes des estampilles d'ordonnancement, permet d'économiser la diffusion des invocations en lecture à toutes les copies, et donc d'économiser des ressources réseaux et processeurs. Cela augmente alors globalement les performances du système, et permet de réduire le temps de réponse vis-à-vis du client.

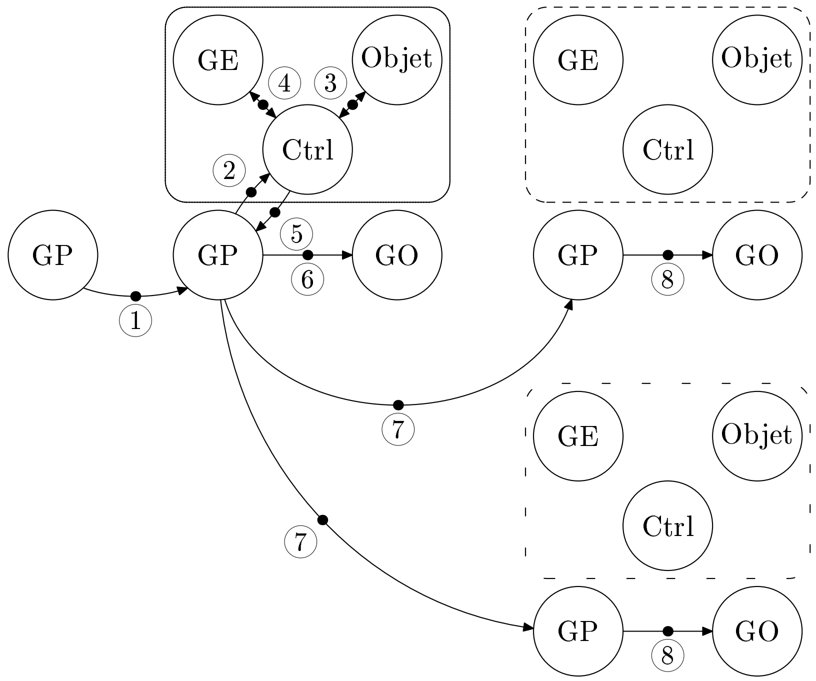
De plus, si `nbrSync` est supérieur à un, même une invocation en lecture, doit être partiellement propagée. Le gestionnaire de propagation peut choisir, grâce a un éventuelle gestionnaire de répartition de charge, les copies qui lui semblent les plus aptes à fournir une réponse rapide vis-à-vis des clients.

4.3.2.2 Tolérance aux fautes

La communication entre les serveurs de propagation n'étant pas synchrone, elle est beaucoup plus simple à mettre en œuvre. Le contrat du gestionnaire de propagation est de faire parvenir, tôt ou tard, l'invocation à toutes ou certaines copies. Il est donc possible pour ce gestionnaire d'attendre le rétablissement d'une liaison défaillante, ou d'attendre réellement qu'une machine lente soit prête à propager les invocations.

Néanmoins, si une copie d'un serveur est défaillante, ou bien si toutes les ressources des gestionnaires de propagation sont utilisées, il doit pouvoir exclure certains membres du groupe des copies du serveur. C'est essentiellement une décision qui concerne seulement les gestionnaires de propagation entre eux. Néanmoins, les gestionnaires d'estampilles doivent être informés d'une telle décision par l'utilisation de leur méthode `excludeMember`. En effet, ces derniers peuvent aussi devoir exclure le gestionnaire d'estampilles de la copie exclue. Par souci de performance, un seul gestionnaire d'estampilles est informé. Par exemple, pour une cohérence locale, c'est-à-dire respectant l'ordre programme (§ 2.3.3.1 page 42), chaque gestionnaire d'estampilles étant complètement indépendant, aucune action particulière n'est à prendre. Dans le cas contraire, le gestionnaire d'estampilles recevant cette notification doit en informer les autres.

L'objet contrôle peut accuser la réception, l'identification et l'estampillage d'une invocation par la méthode `sendAck`. Ceci limite la fenêtre temporelle durant laquelle existe le statut délicat de `propagMayBe` du coté client. Cet accusé de réception contient l'identificateur et les estampilles de cette invocation. La connaissance de l'invocation entièrement identifiée et estampillée coté mandataire permet à celui-ci de la réémettre sans aucun problème de duplication. Le fait que l'objet contrôle et son mandataire soient conçus l'un avec l'autre permet d'ajuster les valeurs des temporisateurs des deux cotés pour ne pas générer des réémissions trop nombreuses durant cette période. Par exemple, si un traitement est de l'ordre de la minute, l'objet contrôle peut n'envoyer un accusé de réception qu'après deux minutes, et le mandataire la réémission de l'invocation qu'après trois minutes. Cela limite l'envoi d'accusés de réception systématiques, et donc économise les ressources réseaux. C'est ici la première politique que



① Un gestionnaire de propagation transmet une invocation à une copie qu'il a choisie comme primaire ;

② L'objet contrôle récupère cette invocation ;

③ Il la transmet à l'objet serveur pour déterminer son type (lecture ou écriture) ;

④ Il la transmet au gestionnaire d'estampilles pour l'identifier et l'estampiller complètement ;

⑤ Enfin, il la transmet au gestionnaire de propagation ;

⑥ Le gestionnaire de propagation la délivre au gestionnaire d'ordonnancement ;

⑦ Et il la propage aux gestionnaires de propagation des autres copies du serveur ;

⑧ Ces derniers la délivrent, à leur tour, au gestionnaire d'ordonnancement local.

Fig. 4.5 – De la copie primaire aux autres copies du serveur

l'objet contrôle fournit. Elle est dépendante du type de l'application, et doit donc être gérée dans le serveur et non par le site (dans le gestionnaire de propagation). L'adéquation entre la temporisation du coté mandataire et la durée entre la réception de l'invocation et l'émission de l'accusé de réception par l'objet contrôle garantit un minimum de retransmission d'invocations et de transmission d'accusés de réception.

4.3.3 Des copies synchrones à la copie primaire du serveur

Nous rappelons qu'il y a deux types de synchronisation dans ce cadre de travail. Le premier, pour une invocation en écriture, correspond à une synchronisation entre le client et l'exécution de l'invocation par un certain nombre de copies. Le deuxième, pour une invocation en lecture, correspond à une synchronisation entre le gestionnaire d'estampilles et l'enregistrement de l'invocation sur un certain nombre de copies. Le premier est géré par la collecte des réponses des exécutions, et est décrit au § 4.3.5 page 107. Cette section décrit les interactions du deuxième type.

Une fois que le gestionnaire de propagation a délivré l'invocation au gestionnaire d'ordonnancement, celui-ci sait si cette copie peut devenir une copie synchrone. Si le nombre de copie synchrone indiqué dans le paramètre `nbrSync` de `remoteCall` est supérieur à un, le gestionnaire de propagation doit faire remonter l'information que sa copie est synchrone.

En fait le gestionnaire de propagation est libre de fournir le moyen qui convient pour faire remonter cette information. Quand le gestionnaire de propagation de la copie primaire a obtenu le nombre suffisant de notifications de copies synchrones, il le signale au gestionnaire d'estampilles par sa méthode `notifSync`. Ce dernier pourra alors utiliser librement cet identificateur, en étant sûr que sur les copies synchrones, l'invocation en lecture ne sera pas invalidée.

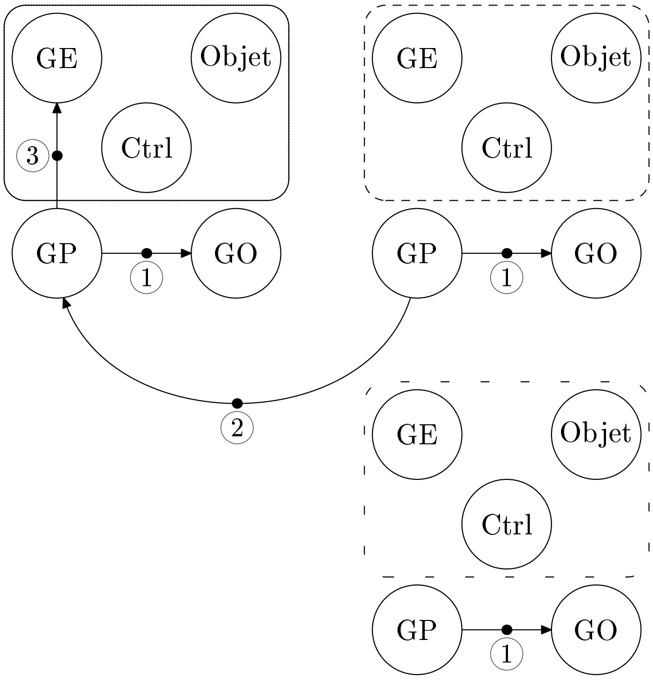
Nous résumons brièvement cette situation à la figure 4.6 page suivante.

4.3.4 Gestion de la propagation « à la demande »

Le deuxième rôle de l'objet contrôle est de piloter si nécessaire une propagation « à la demande ». Cette méthode consiste à ce que les copies demandent une invocation qui leur manque, et non à qu'il y ait une propagation spontanée de toutes les invocations vers les copies.

Pour cela, l'objet contrôle utilise la méthode `unqueueWaiting` du gestionnaire d'ordonnancement pour récupérer les identificateurs des invocations manquantes (qui ne sont pas encore arrivées sur cette copie) et qui empêchent l'exécution d'autres invocations. Une fois récupérés ainsi, ces identificateurs ne peuvent plus réapparaître dans cette liste. L'objet contrôle en informe alors le gestionnaire de propagation par `pullQuery`. Ce dernier peut, ou non, faire le nécessaire pour que cette invocation arrive sur cette copie. La fréquence de ces demandes est une politique qui dépend de l'objet contrôle. Cela peut être immédiatement, c'est-à-dire à chaque fois qu'une invocation fait défaut, ou bien à des périodes régulières dont la durée est aussi déterminée par l'objet contrôle.

Le mandataire de l'objet contrôle coté client doit, par l'intermédiaire du paramètre `pull` de `remoteCall`, spécifier quelles sont les invocations qui sont propagées de cette façon. En effet, il n'est pas très efficace d'imposer ce genre de politique à une invocation qui doit rendre des résultats au client. Donc, l'objet contrôle et son mandataire renferment la politique de



① Une fois que l'invocation est transmise au gestionnaire d'ordonnancement, la copie est considérée comme synchrone ;

② Le gestionnaire de propagation peut, ou non, transmettre cette information suivant le nombre de copies synchrones demandées. Ici, deux copies sont requises : la primaire, la synchrone, mais pas la troisième ;

③ Une fois le nombre de copies synchrones requis atteint, le gestionnaire de propagation notifie le gestionnaire d'estampilles de cet état.

Fig. 4.6 – Des copies synchrones à la copie primaire du serveur

propagation « à la demande » en désignant quelles invocations sont concernées, comment et quand elles seront demandées. Ces interactions sont résumées à la figure 4.7 page suivante.

Il faut noter que, même si l'objet contrôle met en place une politique de propagation « à la demande », le gestionnaire de propagation n'implante pas forcément ce mécanisme. Dans ce cas, il propage normalement toutes les invocations nonobstant le paramètre `pull`, et ignore les invocations de type `pullQuery`.

4.3.5 Exécution et retour des paramètres au client

Cette section décrit les interactions à partir du moment où l'invocation est prête à être exécutée jusqu'à ce que les résultats de son exécution soient connus du client.

L'objet contrôle de chaque copie demande régulièrement au gestionnaire d'ordonnancement les invocations prêtes à être exécutées par la méthode `unqueueReady`. Le gestionnaire d'ordonnancement retourne les invocations prêtes à être exécutées en fonction du principe des estampilles d'ordonnancement et de sa politique locale de priorité d'exécution.

Le troisième rôle de l'objet contrôle, la politique d'exécution, se trouve dans cette phase. Il doit décider comment exécuter cette invocation prête. Il peut créer ou non une activité par invocation, avec ou non une borne sur le nombre d'activités, ou bien utiliser un stock d'activités qu'il attribue au fur et à mesure des invocations prêtes, etc. Dans tout les cas, il fait exécuter à l'objet serveur cette invocation prête, en utilisant `doCall` par un moyen approprié. Il utilise un objet de type `objectQuery` pour passer les paramètres entrants et un objet de type `objectReply` pour recevoir les paramètres sortants.

Une fois l'exécution de cette invocation terminée, l'objet contrôle en informe le gestionnaire d'ordonnancement par `notifFinish` pour que ce dernier gère les invocations dépendantes d'elle, ainsi que ses structures de données.

Enfin, l'objet contrôle renvoie alors les résultats de l'exécution au gestionnaire de propagation à l'aide de la méthode `sendReply`. Il est à noter qu'au cas où la méthode `sendAck` n'a pas encore été appelée, cette méthode envoie aussi l'identificateur, les estampilles et les paramètres privés du gestionnaire d'estampilles. Le gestionnaire de propagation fait suivre les réponses au mandataire coté client, d'une façon qui lui est propre, et au moins en nombre suffisant par rapport à la quantité demandée (`nbrReply`).

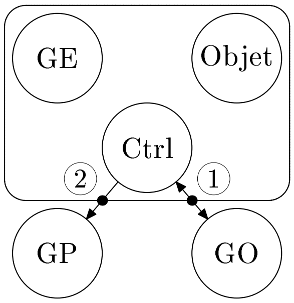
Le mandataire de l'objet contrôle reçoit les différentes réponses. Il informe, par la méthode `update`, le mandataire du gestionnaire d'estampilles des paramètres privés que le gestionnaire d'estampilles qu'il représente lui fait parvenir. Le mandataire de l'objet contrôle transmet éventuellement à un gestionnaire de vote les différents résultats. Enfin, il fournit une réponse au mandataire de l'objet serveur, qui déballe les paramètres, et les retourne au client.

L'ensemble de cette démarche est résumé à la figure 4.8 page 109.

4.3.5.1 Performance

La collecte du nombre de réponses attendues, comme du nombre de copies synchrones, est faite par le gestionnaire de propagation. Dans tous les cas l'objet contrôle envoie les réponses à ce gestionnaire, mais lui ne renvoie pas forcément ces indications à la copie primaire, ou au mandataire. Lui seul connaît la configuration du réseau, et les protocoles qu'il utilise. Il peut donc optimiser leur acheminement.

Par exemple, si le gestionnaire de propagation utilise un canal *multicast* IP, il n'est pas beaucoup plus coûteux pour lui d'émettre la réponse d'une copie à toutes les copies. Toutes



① L'objet contrôle demande les identificateurs des invocations qui bloquent d'autres invocations ;

② Il fait alors une demande de réception de ces invocations au gestionnaire de propagation.

Fig. 4.7 – Politique de propagation « à la demande » par l'objet contrôle

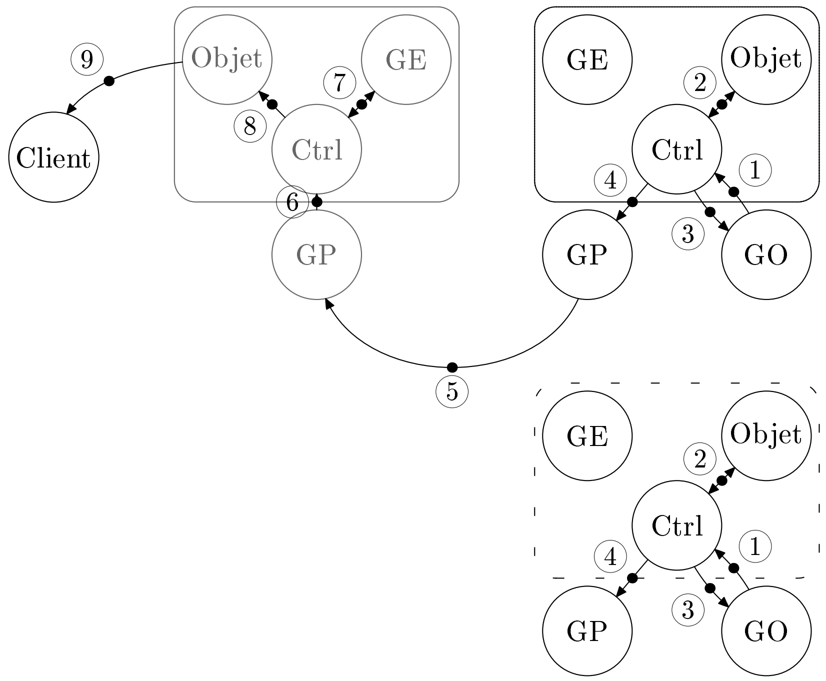
les copies connaissent alors le nombre de réponses actuellement collectées, ainsi que le nombre requis. Si le client demande deux réponses, les deux premières copies diffusent leur réponse respective. La troisième sait alors qu'il est inutile pour elle de renvoyer sa propre réponse. Pour d'autres protocoles, comme l'utilisation de paires de processus communiquant, ce genre d'optimisation n'est pas économique, voire impossible. Dans ce cas, le gestionnaire de propagation peut désigner, à l'avance, quelles seront les copies qui fourniront les réponses attendues, par exemple à l'aide d'un gestionnaire de répartition de charge.

Enfin, la collecte des réponses par le gestionnaire de propagation utilise les premières arrivées. Ainsi, une invocation en écriture par exemple, sera synchrone par rapport à la plus rapide des copies du serveur. La propagation de l'invocation aux autres copies est strictement asynchrone. Cette asynchronisme permet une amélioration des performances, sans toutefois basculer dans un mode de programmation asynchrone, qui, nous l'avons vu, est plus difficile à utiliser.

4.3.5.2 Erreur d'exécution locale

Nous appelons erreur d'exécution locale, les erreurs d'exécution se produisant que sur certaines copies. Il s'agit par exemple d'erreurs d'allocation mémoire (toutes les copies n'ayant pas forcément les mêmes ressources sur tous les sites), ou d'entrées / sorties disques (problèmes de place libre ou de permission), etc. Dans le cas contraire, une erreur est dite globale, car se produisant sur toutes les copies.

Une telle erreur peut donc se produire sur une copie et pas une autre, il y a alors une divergence d'état entre ces deux copies. Néanmoins, cette différence d'état ne condamne pas forcément cette copie à s'arrêter. Elle peut très bien continuer d'exécuter des invocations qui ne dépendent pas de cet état. C'est le modèle de cohérence qui indique cette dépendance. Par exemple, si toutes les invocations sont indépendantes les unes par rapport aux autres, une erreur d'exécution locale n'empêche pas les autres invocations de s'exécuter. En fait, si une invocation a échoué à cause d'une erreur locale, seule les invocations qui dépendent d'elle ne doivent pas être exécutées, puisque dépendant d'un état qui n'a pas été atteint.



① L'objet contrôle demande au gestionnaire d'ordonnancement une invocation prête à être exécutée ;

② Il la transmet à l'objet pour exécution et récupère les résultats ;

③ Il notifie le gestionnaire d'ordonnancement de la fin de son exécution ;

④ Enfin, il indique le résultat au gestionnaire de propagation ;

⑤ Le gestionnaire de propagation transmet, ou non, suivant le nombre demandé, la réponse. Ici, une seule réponse est demandée, la seconde copie n'a donc pas besoin de transmettre son résultat ;

⑥ Le gestionnaire de propagation collecte les résultats, et les transmet au mandataire de l'objet contrôle ;

⑦ Le mandataire de l'objet contrôle met à jour le mandataire du gestionnaire d'estampilles avec les paramètres privés transmis par la copie primaire ;

⑧ Il fournit une seule réponse au mandataire de l'objet ;

⑨ Enfin, ce dernier déballe le résultat, et répond à l'invocation du client.

FIG. 4.8 – Exécution et retour des paramètres au client

Pour cela, la méthode `doCall` de l'objet serveur renvoie un paramètre informant si l'exécution de l'invocation a provoqué, ou non, une erreur locale. Il faut bien comprendre qu'une erreur globale, comme par exemple détruire un élément d'une liste vide, c'est-à-dire une erreur qui va se produire sur toutes les copies, est considérée à ce niveau comme une exécution sans erreur. Quand `doCall` renvoie une erreur locale, alors l'objet contrôle le signale au gestionnaire d'ordonnancement par un paramètre de la méthode `notifFinish`.

Ainsi, le gestionnaire d'ordonnancement peut abandonner toutes les invocations qui dépendent de cette invocation et cela de manière récursive. Elles ne sont donc pas exécutées sur cette copie. Chaque abandon d'invocation est notifié au gestionnaire de propagation par la méthode `abortInv` pour qu'il libère les ressources éventuellement occupées par celle-ci. Bien sûr, toutes les invocations arrivant ultérieurement, et qui dépendent d'une invocation abandonnée sont aussi abandonnées.

Cette méthode a l'avantage de ne pas systématiquement arrêter les copies provoquant des erreurs locales. Néanmoins, cette copie peut avoir une forte proportion d'invocations abandonnées. Cela consomme inutilement des ressources (essentiellement réseau). Nous verrons au § 4.5.2 page 116 comment un gestionnaire de répartition de charge peut prendre des décisions en fonction de ce critère.

4.4 Cycle de vie

Cette section décrit le déroulement des opérations de cycle de vie. Nous commençons par décrire le mécanisme des barrières d'invocations (§ 4.4.1) indispensable aux opérations suivantes. Nous décrivons alors la création d'une copie (§ 4.4.2) et sa destruction (§ 4.4.3). Enfin, nous examinons le recyclage des identificateurs (§ 4.4.4).

4.4.1 Barrières d'invocations

Nous présentons dans ce paragraphe un mécanisme de barrière d'invocations qui nous sert à assainir certaines situations décrites par la suite. La méthode `sendFlush` du gestionnaire de propagation prend comme argument un identificateur et un masque, définissant alors un ensemble d'identificateurs. Il doit garantir la séparation de la livraison de certaines invocations (celles dont l'identificateur ou une de ces estampilles appartiennent à cet ensemble) émises avant le `sendFlush` de celles émises après. C'est une barrière entre les invocations transportant un identificateur appartenant à un ensemble donné.

Le gestionnaire garantit cette propriété comme il le veut, puisque cela dépend essentiellement de son protocole de communication. Par contre, quand il passe cette barrière — c'est-à-dire après avoir délivré toutes les invocations émises avant le `sendFlush` et avant de délivrer celles émises après — il doit le notifier au gestionnaire d'ordonnancement et au gestionnaire d'estampilles grâce à leur méthode `notifFlush`. Il peut ensuite délivrer les autres invocations dès que ces deux méthodes `notifFlush` sont terminées.

La méthode `notifFlush` du gestionnaire d'ordonnancement a pour but d'abandonner toutes les invocations qui dépendent d'une invocation dont l'identificateur fait partie de l'ensemble décrit par la barrière. En effet, le gestionnaire de propagation garantit, qu'après la barrière, plus aucune invocation dont l'identificateur appartient à cet ensemble ne peut arriver. Plus exactement, si une telle invocation arrive, elle n'a aucun rapport avec l'invocation réellement attendue. Ces invocations ne peuvent donc plus jamais être prêtes pour exécution. Elles sont donc abandonnées, en le signalant au gestionnaire de propagation par sa méthode `abortInv` pour qu'il libère les éventuelles ressources consommées par cette invocation. Cette procédure est récursive, c'est-à-dire qu'il faut aussi abandonner toutes les invocations qui dépendent d'une invocation abandonnée.

C'est en fait la seule garantie sur l'ordre de livraison que le gestionnaire de propagation doit observer. Elle génère des dépendances entre les invocations concernées avant et après la barrière. Elle rajoute des dépendances superflues, c'est donc un mécanisme exceptionnel à n'utiliser qu'en cas de besoin. Néanmoins l'utilisation d'ensembles d'identificateurs permet de cibler précisément les invocations concernées, et de ne pas influencer les autres. Cette méthode est utilisée pour remettre à zéro des situations compliquées ou instables dûes principalement à des défaillances.

4.4.2 Création d'une copie

Dans ce cadre de travail, nous décrivons les mécanismes pour que l'état de la copie créée soit cohérent avec les autres copies. Nous ne décrivons pas le mécanisme de prise de décision de la création d'une nouvelle copie. Il peut s'agir d'une décision par une commande lancée manuellement par un utilisateur ou un administrateur. Cela peut tout aussi bien être une demande de création émanant d'un gestionnaire de répartition de charge.

La création d'une copie d'un serveur commence par la création de l'objet contrôle qui va prendre en charge le transfert d'état au moyen de la méthode `newObject` du gestionnaire de propagation. Ce dernier a la charge de trouver une copie du serveur que nous appellons la copie source. Si aucune copie n'existe, le gestionnaire de propagation renvoie cette information.

Quand le gestionnaire de propagation de l'objet source reçoit une demande de copie de l'état, il ne délivre plus aucune invocation au gestionnaire d'ordonnancement jusqu'à ce que la méthode `dupObject` de l'objet contrôle soit appelée et terminée. Le `dupObject` de l'objet contrôle bloque la livraison des invocations prêtes à l'objet serveur local, et invoque `getState` sur celui-ci. L'objet serveur recopie son état, après avoir éventuellement attendu la fin des diverses activités en cours d'exécution. L'objet contrôle invoque alors le `getState` du gestionnaire d'estampilles et du gestionnaire d'ordonnancement. Ce premier renvoie des informations opaques, comme un état ou une adresse de groupe de communication. Ce dernier recopie l'état de ses structures de données concernant cette copie de l'objet. Il s'agit aussi d'informations opaques. Enfin, l'objet contrôle constitue un message de ces trois états, et ajoute d'éventuelles informations pour l'objet contrôle de la nouvelle copie. Il renvoie ce message comme paramètre de retour de sa méthode `dupObject`. À ce moment là, l'objet contrôle peut reprendre la livraison des invocations prêtes à l'objet serveur et le gestionnaire de propagation peut reprendre la livraison des invocations au gestionnaire d'ordonnancement. Mais ce dernier doit maintenant les propager à la nouvelle copie.

L'objet contrôle de la nouvelle copie reçoit donc les états cohérents de chaque sous objet. Il peut alors appeler les `setState` du gestionnaire d'ordonnancement, du gestionnaire d'estampilles et de l'objet serveur, ainsi que mettre à jour son propre état.

Les états de tous ces objets sont cohérents, puisqu'au niveau de la copie source, plus aucune invocation n'est délivrée ni à l'objet contrôle, ni à l'objet serveur, pendant la récupération des états. Par contre, dès que cette phase est terminée, le gestionnaire de propagation fait suivre à la nouvelle copie les invocations qu'il délivre au gestionnaire d'ordonnancement de la copie source.

4.4.2.1 Performance

Le critère de performance pour la création d'une copie n'est pas crucial. Néanmoins, le gestionnaire de propagation peut choisir une copie source pertinente, c'est-à-dire peu utilisée. En effet, la création d'une copie bloque l'exécution de la copie source durant la recopie des états. Il faut donc encourager l'utilisation de copies peu sollicitées.

4.4.2.2 Tolérance aux fautes

Le gestionnaire de propagation doit garantir une communication fiable. Entre autres, il doit garantir que l'état de la copie source arrive bien à la copie cible, et il doit propager les invocations qui arrivent après la recopie des états.

Par contre, l'identificateur unique est construit à partir d'un identificateur d'objet. Le choix de l'identificateur d'objet, est un choix du système à objets sous-jacent. Nous ne voulons pas imposer une quelconque vue sur celui-ci. Nous considérons donc qu'il est possible que l'identificateur d'objet ne soit pas unique dans le temps. Dans ce cas, la nouvelle copie peut se voir attribuer un identificateur d'objet qui a déjà servi, et qui peut donc faire partie d'identificateurs d'invocations encore présents dans le système. Cela risque de créer des dépendances superflues ou, plus important, d'en violer des requises.

Pour régler ce problème, le gestionnaire d'estampilles utilise la méthode `sendFlush` du gestionnaire de propagation avec les paramètres adéquats, pour qu'il y ait une séparation entre les invocations émises par le précédent objet possédant cet identificateur, et ce nouvel objet. Cette méthode n'est pas très coûteuse si aucun identificateur de l'ancien objet n'était encore présent dans le système, et donc si l'identificateur d'objet été unique dans le temps. Dans le cas contraire, elle induit des surcoûts, mais elle est alors indispensable.

4.4.3 Destruction d'une copie

Tout comme la création d'une copie, nous nous intéressons seulement au mécanisme de destruction d'une copie et non à la prise de décision de celle ci. Nous pouvons considérer deux sortes de destruction d'une copie, l'une qui est « douce » qui correspond à un ordre de destruction, et une « violente » qui correspond essentiellement à une défaillance.

4.4.3.1 Destruction « douce »

La destruction « douce » correspond à une destruction que nous voulons concertée. Il s'agit d'un arrêt progressif, c'est-à-dire permettant aux invocations déjà délivrées d'être exécutées, mais en refusant toutes les nouvelles.

Pour cela, quand l'objet contrôle veut détruire sa copie, il invoque `stopNew` du gestionnaire d'ordonnancement. Ce dernier n'accepte plus que les invocations qui pourront débloquer des invocations déjà arrivées avant la demande d'arrêt. Toutes les autres invocations sont refusées. Cela permet d'exécuter toutes les invocations déjà présentes au moment de l'arrêt, et donc de fournir les paramètres de retour de ces invocations au client.

Une fois que le gestionnaire d'ordonnancement n'a plus aucune invocation ni prête, ni en attente (il peut vérifier cette condition par la méthode `isEmpty`), il s'arrête. Les destructeurs des différents objets peuvent alors effectuer des actions pour se retirer de groupes de communication ou informer partiellement de leur état de manière adéquate.

4.4.3.2 Destruction « violente »

La destruction « violente » est dûe à un arrêt forcé de la copie, à une erreur, à une déconnexion, ou à toute autre situation où la copie ne peut ni s'arrêter doucement, ni même prévenir les autres copies.

Dans ce cas, ce sont les autres copies qui s'aperçoivent de cette situation. Nous plaçons ce cadre de travail dans un système asynchrone, il ne peut donc pas y avoir de certitude sur la défaillance d'une machine. Néanmoins, par des temporisateurs, ou des suspecteurs de fautes, le gestionnaire de propagation peut considérer, à tort ou à raison, qu'une copie est défaillante. Dans ce cas, il va tout simplement l'exclure des copies du serveur.

Les mécanismes pour suspecter cette défaillance, les valeurs des temporisateurs, tout comme les actions prises pour exclure une copie sont internes au gestionnaire de propagation. Il peut très bien être conçu pour supporter des portables déconnectés, et donc avoir des valeurs de temporisateurs de l'ordre de journées, ou bien supporter du travail collaboratif, et des valeurs plus proches de la minute. Il peut utiliser un protocole de propagation entre deux machines sur une topologie fixe, et donc ne signaler qu'a lui même la suspicion de défaillance de son partenaire, ou bien utiliser des groupes de communication avec changement de vue, et donc répercuter l'exclusion aux autres membres.

Dans tous les cas, il avertit le gestionnaire d'estampilles, par l'intermédiaire de sa méthode `excludeMember`, de l'exclusion de copies. Ce dernier, ayant éventuellement un groupe de communication avec les autres copies, peut avoir besoin d'exclure aussi cette copie. De plus, suivant le modèle de cohérence qu'il implante, et étant donné qu'il est le seul à connaître la structure de l'identificateur, il peut provoquer une barrière d'invocations à l'aide de `sendFlush`. Il provoque ainsi l'abandon de toutes les invocations bloquées par d'autres invocations qui n'arriveront jamais à cause de cette exclusion.

Un seul gestionnaire d'estampilles est notifié de l'exclusion, puisque suivant sa politique de cohérence, il se peut qu'il n'y ait pas de communication entre eux. Dans l'exemple trivial de la cohérence « à la longue » (§ 2.3.1.6 page 34), les gestionnaires d'estampilles sont complètement indépendants, puisque ne faisant rien. Il est clairement inutile de prévenir tous les gestionnaires d'estampilles de l'exclusion d'une copie.

4.4.4 Recyclage des identificateurs

L'espace des identificateurs n'est pas infini, ce qui veut dire qu'il faut un mécanisme pour assurer la réutilisation des identificateurs qui ne servent plus. Une technique simple consiste à prendre un espace suffisamment large pour que l'on puisse faire l'hypothèse qu'une nouvelle utilisation d'un identificateur soit suffisamment espacée dans le temps pour qu'il soit considéré comme étant réutilisable. Cette solution, optimiste, a le mérite d'être très économique en gestion. Néanmoins, pour des utilisations particulières, il n'est pas possible de prendre ce risque. Nous avons opté pour un mécanisme qui génère un faible surcoût s'il n'est pas utilisé, laissant le libre choix au développeur des gestionnaires d'estampilles de l'utiliser ou non.

Quand un identificateur ne sert plus sur une copie précise, le gestionnaire d'ordonnancement[2] en informe le gestionnaire d'estampilles local par sa méthode `freeId`. Le gestionnaire d'estampilles est responsable de la collecte des notifications d'identificateurs pour la copie locale et pour toutes les copies à qui le gestionnaire de propagation local a fait suivre l'invoca-

2. Nous verrons dans le chapitre 5 page 119 comment le gestionnaire d'ordonnancement sait qu'un identificateur ne sert plus.

tion. Pour cela, le paramètre `nbr` de la méthode `freeId` indique combien il lui reste à attendre de `freeId` sur cet identificateur de la part des autres gestionnaires d'estampilles.

Quand ce paramètre est à zéro, cela veut dire que l'identificateur est libre localement et sur toutes les copies qui ont reçu cet identificateur de cet objet. S'il s'agit d'un identificateur qui a été émis par ce gestionnaire d'estampilles, l'identificateur est donc libre sur toutes les copies, puisque cet identificateur a été propagé initialement par cet objet. Si il a été émis par un autre gestionnaire, alors le gestionnaire d'estampilles doit en informer le gestionnaire de l'objet qui lui a transmis avec la méthode `freeId` du gestionnaire de propagation.

Cette dernière méthode ne prend plus en paramètres un ensemble d'identificateurs, mais un type opaque. Grâce à cela, le gestionnaire d'estampilles est capable d'effectuer des compressions d'identificateurs, puisqu'il a la connaissance de la structure de l'identificateur et de son utilisation. Par exemple, si les identificateurs sont émis dans un ordre croissant, il est possible d'informer de leur disponibilité en n'indiquant que le plus grand d'un ensemble d'identificateurs consécutifs libres, ou bien de fournir des intervalles d'identificateurs libres. Pour cela, il peut retarder la notification en attendant que d'autres identificateurs soient libres. De plus, il connaît l'espace des identificateurs, il peut donc savoir localement si le gestionnaire qui a émis cet identificateur est plutôt en pénurie ou non. Il peut donc adapter les périodes de notifications en fonction de ce critère.

4.4.4.1 Performance

Ce gestionnaire d'estampilles peut tout simplement oublier la notification du gestionnaire d'ordonnancement si il implante la solution optimiste décrite plus haut. Le surcoût est donc pratiquement nul, et reste confiné sur le site.

Par contre, les gestionnaires d'estampilles voulant gérer de façon pessimiste le recyclage des identificateurs peuvent le faire de manière performante notamment grâce à des mécanismes de compression. Ils sont les seuls à avoir suffisamment de connaissance sur la structure d'un identificateur pour réaliser cela.

4.4.4.2 Tolérance aux fautes

La tolérance aux fautes d'un tel mécanisme est complexe. Nous n'avons pas introduit de protocoles particuliers pour la gérer. En effet, le surcoût ne serait plus tolérable, essentiellement pour les protocoles optimistes qui ne font rien de particulier.

Néanmoins, pour gérer ces problèmes de tolérance aux fautes, les gestionnaires d'estampilles à court d'identificateurs libres peuvent utiliser la méthode `sendFlush` pour garantir que les émissions d'identificateurs après cet appel ne se mélangeront pas avec les identificateurs émis avant cet appel. Comme nous l'avons vu précédemment, cela provoque une barrière entre les invocations utilisant l'ensemble d'estampilles désignées avant et après l'appel. Il y a donc des synchronisations superflues entre les invocations, et la cohérence est donc plus forte que celle voulue par le gestionnaire d'estampilles. C'est donc une méthode à utiliser en dernier ressort.

Quand un gestionnaire d'estampilles n'a plus assez de ressources pour gérer le recyclage des identificateurs, il peut alors tout simplement abandonner ce recyclage. Cela provoque tôt ou tard un `sendFlush` du gestionnaires d'estampilles émetteurs, dont il est averti par sa méthode `notifFlush`. Il peut alors reprendre le recyclage d'une manière plus fine.

4.4.4.3 Déconnexion importante

Un autre cas se produit quand il existe des sites déconnectés relativement longtemps. Prenons l'exemple d'un portable où une copie d'un serveur réside. Tant que celui ci n'a pas été connecté, et n'a pas exécuté les invocations, alors les identificateurs ne peuvent pas être réutilisés. Or le gestionnaire de propagation, qui, rappelons le, est indépendant, peut autoriser de longues déconnexions. Dans ce cas, le portable provoque des pénuries d'identificateurs. Avec l'utilisation de barrières d'invocations, les identificateurs peuvent être réutilisés avant que le portable se reconnecte. Le gestionnaire de propagation aura alors enregistré la barrière dans ses journaux, et la transmet au moment voulu quand le portable se reconnecte.

4.5 Autres interactions

Cette section décrit les dernières interactions de notre cadre de travail. Nous commençons par décrire les interactions entre les gestionnaires d'estampilles entre eux (§ 4.5.1), puis celles entre le gestionnaire d'ordonnancement et un gestionnaire de répartition de charge (§ 4.5.2) et enfin les différentes interactions qui peuvent exister dans un système complet (§ 4.5.3) c'est-à-dire en utilisant les mécanismes de la plate-forme objet sous-jacente.

4.5.1 Interactions entre les gestionnaires d'estampilles

Nous avons déjà vu qu'un gestionnaire d'estampilles échange des informations privées avec ses mandataires du coté client. Plus exactement, le mandataire transmet un message privé au gestionnaire d'estampilles de la copie primaire. Réciproquement, un message privé est envoyé du gestionnaire d'estampilles vers le mandataire lors de l'acquittement ou du retour d'une invocation. Ce premier échange permet de communiquer avec le mandataire sans que celui-ci fasse partie d'un groupe de communication. En effet, pour avoir un système qui supporte facilement le passage à l'échelle par rapport au nombre de clients, il n'est guère possible de mettre tous les mandataires dans un groupe de communications. De plus, cet échange permet au gestionnaire d'estampilles de stocker les données concernant un client particulier, dans le mandataire sur le site de ce client. Il n'a donc pas besoin d'une structure de données dont la taille est proportionnelle au nombre de clients. Le gestionnaire d'estampilles passe donc plus facilement l'échelle, et peut simplifier les mécanismes de tolérance aux fautes.

Un deuxième type d'interaction peut exister entre les gestionnaires d'estampilles, suivant le modèle de cohérence ou de synchronisation qu'ils fournissent. Ce n'est pas le cas dans, par exemple, les modèles de cohérence comme celui « à la longue » (§ 2.3.1.6 page 34) ou « lire ses écritures » (§ 2.3.3.1 page 42). Par contre, une cohérence atomique (§ 2.3.1.1 page 27), doit communiquer avec les gestionnaires d'estampilles des autres copies.

Cette communication n'est pas spécifiée dans notre cadre de travail, laissant entièrement la responsabilité au développeur des gestionnaires d'estampilles d'utiliser les moyens de communication qui lui semblent nécessaires. De même, il lui laisse la liberté de choisir son moyen de détecter les défaillances. Ainsi, il peut utiliser aussi bien une gestion de groupe comme Isis avec des suspecteurs de fautes, que des mécanismes plus bas niveaux comme des RPC avec des temporisations.

À première vue, cela peut sembler redondant avec les gestionnaires de propagation, puisque eux aussi maintiennent un groupe de copies. D'ailleurs, les méthodes `excludeMember` répercutent de l'un à l'autre les changements de membres. Néanmoins, les deux types de communications sont fort différents. Le gestionnaire de propagation assure entre les membres du groupe une diffusion « à la longue » asynchrone, alors que l'essentiel des besoins d'un gestionnaire d'estampilles est plutôt de type synchrone avec des suspicions de défaillances rapides, pour ne pas bloquer des clients trop longtemps.

Or, nous voulons un gestionnaire de propagation par site, qui s'adapte à son environnement. Par exemple, s'il est situé sur un réseau local, il peut utiliser le *multicast* IP avec des temporisateurs de faible durée, s'il est situé sur une machine qui accepte des connexions de portable, il préférera des communications par paire de machines, avec une journalisation des invocations sur disques, et des temporisateurs de durée beaucoup plus importante. Il peut très bien être dans les deux cas, et fournir une passerelle entre le réseau local et le portable. Bref, le gestionnaire de propagation doit s'adapter à la topologie du réseau pour être efficace, et doit pouvoir être géré par l'administrateur du site, et non le développeur de l'application. Le gestionnaire d'estampilles n'aurait donc aucune garantie sur les caractéristiques du gestionnaire de propagations, il pourrait même avoir des garanties différentes sur différents sites.

Inversement, le gestionnaire d'estampilles peut vouloir utiliser les outils de communication qu'il souhaite, avec les valeurs de temporisation qu'il trouve adéquates. Il peut vouloir être extrêmement performant, et par exemple surveiller la défaillance que de certaines copies précises.

De plus, il est possible que l'exclusion d'un membre du groupe des gestionnaires d'estampilles ne soit pas pris en compte par le gestionnaire de propagation. Dans ce cas, une copie n'ayant pas de gestionnaire d'estampilles, car exclue, est une copie secondaire (elle ne peut plus accepter directement d'invocations de la part d'un client). Par exemple, même si une copie a des temps de réponses provisoirement très lents, et même si le gestionnaire d'estampilles s'est fait exclure du groupe, en prévision d'une amélioration il peut être utile de garder l'état de cette copie cohérent en lui propageant les invocations. Ainsi, la réintégration du gestionnaire d'estampilles peut se faire simplement sans transfert d'état. Ce cas est adapté au portable, ou l'on peut imaginer une copie qui a un gestionnaire d'estampilles dans le groupe quand il est connecté, et exclu quand il est déconnecté. Dans les deux cas, le gestionnaire de propagation journalise les invocations à lui transmettre. La copie évite donc ainsi des transferts d'état par destruction et création alternative d'une copie.

Pour toutes ces raisons nous laissons le développeur de gestionnaire d'estampilles utiliser les technologies qu'il souhaite, et nous autorisons les gestionnaires de propagation à optimiser leur protocole en fonction de la topologie et des caractéristiques des réseaux.

4.5.2 Interactions avec un répartiteur de charge

Nous avons évoqué, à plusieurs reprises, l'existence d'un gestionnaire de répartition de charge. Il peut sélectionner les copies primaires, les copies synchrones, ou bien décider de la création ou de la destruction d'une copie tout en maintenant un nombre de copies minimum d'un serveur, pour assurer une tolérance aux fautes.

Un gestionnaire de répartition de charge se base sur des informations de charge, comme le taux d'utilisation des processeurs, des réseaux, de la mémoire. Nous pouvons indiquer à un tel gestionnaire des informations supplémentaires sur l'état d'une copie d'un serveur. Ainsi, le gestionnaire d'ordonnancement peut fournir, par l'intermédiaire de sa méthode `getStat`, une structure décrivant des valeurs à un instant donné (`snap`) ou cumulées depuis la dernière consultation (`total`). Cette structure, `deliveryStat`, contient les champs suivants :

nbrExec : Il indique le nombre d'invocations qui ont été exécutées depuis la dernière fois (`total`), ou qui sont en cours d'exécution (`snap`) ;

nbrReady : Il donne le nombre d'invocations qui sont prêtes a être exécutées ;

nbrWaiting : Il fournit le nombre d'invocations en attente d'autres invocations qui ne sont pas encore arrivées ;

nbrAbort : Il indique le nombre d'invocations qui ont été abandonnées.

Cette méthode ne fournit pas de charge amortie, car le temps d'amortissement est un paramètre important dans la prise de décision. Ainsi, grâce au paramètres `total`, et à des requêtes régulières, l'utilisateur peut appliquer l'amortissement qu'il considère pertinent.

4.5.3 Interactions avec d'autres gestionnaires

Ce cadre de travail n'est qu'une vue partielle d'un système distribué à objets. Il décrit une solution pour supporter le protocole des estampilles d'ordonnancement. Par exemple, le gestionnaire de propagation doit localiser les différentes copies d'un serveur. Néanmoins, la plate forme objet sous-jacente a, dans la plupart des cas, un service de nommage et de localisation similaire. Le gestionnaire de propagation doit donc s'adapter à un tel environnement. Pour cela nous avons décrit un cadre de travail le plus indépendant possible de ces interactions.

Ce cadre de travail est donc une architecture partielle, chaque gestionnaire pouvant être affiné pour intégrer un système particulier, mais sans ajouter d'interactions supplémentaires entre les gestionnaires que nous avons décrits.

4.6 Conclusion

Ce chapitre propose un cadre de travail pour les estampilles d'ordonnancement. Les interactions entre ces gestionnaires ont été conçues avec pour critère principal la performance, en privilégiant les trois caractéristiques suivantes :

1. Les gestionnaires peuvent tirer parti de l'environnement pour optimiser leurs actions : l'objet contrôle de la sémantique du serveur, les gestionnaires d'estampilles des caractéristiques du modèle de cohérence ou de synchronisation, le gestionnaire de propagation de la topologie et des caractéristiques des réseaux ;
2. Les communications entre les différents objets sont les plus asynchrones possible en accord avec les paramètres de tolérance aux fautes ;
3. Les interactions sont au maximum locales, pour ne pas pénaliser les gestionnaires qui n'utilisent pas telle ou telle fonctionnalité (recyclage des identificateurs, exclusion de copies, etc).

Pour cela, nous avons défini chaque gestionnaire comme encapsulant des fonctions précises, et des interactions claires avec les autres gestionnaires. Ils sont donc interchangeables les uns indépendamment des autres, facilitant les changements de politique en changeant la version d'un gestionnaire.

Chapitre 5

Estampilles d'Ordonnancement : implantation

L'IMPLANTATION D'UN PROTOTYPE pour vérifier concrètement certaines hypothèses nous semble important. En effet, à partir d'un protocole simple, l'ajout de mécanismes supplémentaires essentiels pour assurer de bonnes performances (groupe d'invocations, cardinal, surcharge, etc), peut sembler compliquer le protocole des estampilles d'ordonnancement et le cadre de travail. Le prototype a donc pour but essentiel de montrer que les solutions adoptées restent peu coûteuses et simples d'utilisation.

Dans ce chapitre, nous énumérons les objectifs qui ont guidé l'implantation du prototype et nous décrivons brièvement une vue d'ensemble de celui-ci (§ 5.1). Nous introduisons alors un mécanisme supplémentaire, le degré entrant, indispensable à la réalisation du protocole des estampilles d'ordonnancement (§ 5.2). Nous examinons ensuite l'implantation d'un gestionnaire d'ordonnancement en se concentrant sur ses coûts (§ 5.3). Nous décrivons alors une librairie d'identificateurs ainsi que des exemples de gestionnaires d'estampilles (§ 5.4). Enfin, nous présentons quelques évaluations de ce prototype (§ 5.5).

5.1 Implantation d'un prototype

Nous avons implanté un prototype intégrant le protocole des estampilles d'ordonnancement, ainsi que le cadre de travail. Les objectifs principaux de ce prototype sont :

- Vérifier concrètement le bien fondé du cadre de travail décrit dans le chapitre précédent ;
- Vérifier que le gestionnaire d'ordonnancement engendre des coûts faibles ;
- Vérifier que le protocole des estampilles d'ordonnancement permet de programmer simplement des modèles de cohérence et de synchronisation.

Pour cela, nous avons implanté un ORB réduit, sans nommage ni localisation et sans gestionnaire de répartition de charge. Il est écrit en C++. Chaque gestionnaire est alors une classe, le client et les copies des serveurs étant des exécutables renfermant tous les gestionnaires qui leurs sont nécessaires. La tolérance aux fautes n'a pas été prise en compte dans ce prototype.

Le gestionnaire de propagation est implanté en utilisant les services de la boite à outils RMF (*Reliable Multicast Framework*) [DeCleene et al., 1997] qui fournit, entre autres, une diffusion de paquets sans ordre, sans duplication mais de façon fiable en utilisant le *multicast* IP.

Un serveur utilisé par ce prototype a été repris d'une implantation d'un B-Arbre existant. Aucune modification de ce source n'a été introduite. Seuls un talon, un squelette et une méthode pour déterminer le type d'une invocation ont été ajoutés. Le client utilise alors le mandataire comme l'objet serveur, sans être informé de l'aspect distant de ses requêtes.

Nous nous sommes concentrés sur l'implantation d'un gestionnaire d'ordonnancement efficace (en terme de ressources mémoires et de coût processeur), d'une librairie d'utilisation des estampilles d'ordonnancement ainsi que de plusieurs gestionnaires d'estampilles fournissant des modèles de cohérence différents, et un mécanisme d'exclusion mutuelle.

5.2 De la théorie à la pratique

Pour ne pas compliquer le principe des estampilles d'ordonnancement décrit au chapitre 3 page 53, nous avons délibérément passé sous silence les problèmes se rapportant à l'implantation d'un gestionnaire d'ordonnancement. Dans cette section, nous exposons le problème principal à résoudre pour l'implantation de ces principes (§ 5.2.1), et pour le résoudre nous complétons le protocole des estampilles d'ordonnancement avec l'introduction du degré rentrant d'un identificateur (§ 5.2.2) et nous finissons sur le moyen de distribuer ce degré sur plusieurs machines (§ 5.2.3).

5.2.1 Finitude des ressources et de l'espace des identificateurs

Le principe des estampilles d'ordonnancement, principe 3.12 page 79, repose essentiellement sur l'attente de l'exécution d'invocations d'identificateurs donnés. Une solution naïve est de se souvenir de tous les identificateurs des invocations qui ont été exécutées. Ainsi, si une copie reçoit $I\langle e\rangle$, et si e est inconnue, elle est mise en attente, sinon, elle est prête. Cette solution pose deux problèmes. Le premier concerne les ressources (qui ne sont pas infinies) et le temps processeur consommé pour stocker tous ces identificateurs. Le second est qu'elle interdit la réutilisation des identificateurs avec d'autres invocations. Comme l'espace des identificateurs est fini, le nombre de groupes d'invocations gérés par ce système est alors égal au nombre d'identificateurs possibles. Ces deux limitations sont donc inacceptables.

On peut imaginer résoudre ce problème en oubliant tous les identificateurs des groupes d'invocations qui ont été exécutés. Avec le mécanisme décrit au § 4.4.4 page 113, le problème de la réutilisation des identificateurs est clairement résolu. Mais l'absence d'un identificateur dans le système a deux interprétations radicalement différentes. En effet, si une copie reçoit $I\langle e\rangle$, et si e est inconnue, alors soit I'_e a déjà été exécutée, et dans ce cas, $I\langle e\rangle$ est prête, soit I'_e n'a pas encore été délivrée, et dans ce cas, $I\langle e\rangle$ doit être mise en attente. Il est donc impossible d'adopter un tel comportement.

En fait, il faut un mélange de ces deux propositions. Il faut oublier un identificateur pour pouvoir le réutiliser et libérer les ressources qu'il consomme. Mais il faut aussi garder cet identificateur tant qu'il a une utilité, c'est-à-dire tant qu'il existe une invocation identifiée ou estampillée avec mais pas encore arrivée sur la copie. Pour cela, une information supplémentaire, le degré entrant d'un identificateur, indique combien de fois il est utilisé, et donc quand il peut être libéré.

5.2.2 Degré entrant

Les estampilles d'ordonnancement représentent un graphe de dépendance. Dans un graphe orienté, un nœud possède deux degrés. Le degré négatif correspond au nombre d'arcs qui partent du nœud. Le degré positif correspond au nombre d'arcs qui arrivent sur le nœud. Le degré positif d'un nœud, que nous appelons le degré entrant, correspond donc au nombre d'utilisations de l'identificateur pour estampiller une invocation.

La connaissance du degré entrant d'un identificateur, permet donc de connaître le nombre de fois qu'il est utilisé. Ainsi, le gestionnaire d'ordonnancement peut prendre deux décisions sur ces identificateurs. La première est d'oublier cet identificateur seulement quand le gestionnaire d'ordonnancement aura observé le nombre exact d'utilisations de cet identificateur, et qu'il ne s'en servira plus. Quand une invocation $I\langle e\rangle$ est délivrée au gestionnaire d'estampilles, alors soit il connaît e, et donc il sait si cet identificateur bloque ou non I, soit il ne connaît pas e et, dans ce cas, il doit attendre la livraison de $I^{'}_{e}$. La deuxième décision qu'il peut prendre est d'informer le gestionnaire d'estampilles qu'un identificateur est libre localement, et donc de participer à la gestion du recyclage des identificateurs.

Définition 5.1 (Degré entrant d'un identificateur)
Le degré entrant d'un groupe G d'invocations est le nombre de groupes mixtes ou en écriture qui sont estampillés avec l'identificateur de G.

Dans cette définition, seules les estampilles des groupes mixtes ou en écriture sont comptabilisées dans le degré entrant. En effet, un groupe en lecture n'est pas forcément diffusé à toutes les copies. Rappelons nous du principe 3.9 page 69, qui stipule que les estampilles d'une invocation en lecture sont surchargées sur les invocations la référençant. Cela veut dire que si aucune invocation ne dépend d'une invocation en lecture L, alors les estampilles de L ne sont pas propagées, et donc connues, sur toutes les copies. Il faut alors seulement compter dans le degré entrant les estampilles qui sont diffusées sur tous les sites, c'est-à-dire les groupes comportant au moins une invocation en écriture : les groupes mixtes ou en écriture.

Contrairement au cardinal d'un groupe, qui influence la livraison des invocations à l'objet serveur, le degré entrant ne régit que le recyclage des identificateurs et la gestion des ressources du gestionnaire d'ordonnancement. La connaissance du degré entrant d'un identificateur a donc beaucoup moins de contraintes.

Principe 5.1 (Connaissance du degré entrant)
Le degré entrant d'un identificateur doit être, à terme, connu de toutes les copies.

Ainsi, le degré entrant peut être véhiculé avec l'identificateur, avec l'estampille ou bien en surcharge sur n'importe qu'elle invocation en écriture.

5.2.3 Degré entrant distribué

Les gestionnaires d'estampilles, pour implanter certains modèles de cohérence, se communiquent des identificateurs dans le but d'estampiller des invocations avec. Dans l'exemple de la cohérence atomique objet « inverse » que nous avons présenté au § 3.3.3.1 page 77, chaque gestionnaire d'estampilles est capable d'estampiller une invocation en écriture avec un identificateur identique, chacun de leur coté. Ces gestionnaires d'estampilles augmentent donc le degré entrant. D'une manière générale, il est intéressant de laisser plusieurs gestionnaires d'estampilles manipuler le même identificateur, et donc d'avoir un degré entrant distribué, qui ne générera pas de synchronisation supplémentaire au niveau des gestionnaires d'estampilles.

L'idée est de faire du degré entrant une somme partielle. Chacun des gestionnaires maintient un nombre local d'utilisations, un degré partiel, que chacun aura la responsabilité de diffuser à toutes les copies. Le gestionnaire d'ordonnancement les collectera pour obtenir le degré total. Pour être le plus général possible, nous avons adopté la solution suivante.

Chaque degré entrant partiel contient la valeur locale, et se situe à un niveau donné (qui représente une génération). Le niveau 0 est celui du gestionnaire qui a créé l'identificateur. Ce degré partiel contient aussi le nombre de duplications dont il a été la source. Ainsi, quand un gestionnaire veut communiquer un identificateur qu'il connaît, et pas forcément qu'il a créé, il lui communique une copie avec un degré partiel dont la valeur est à 0 (pas encore utilisée), dont le niveau est supérieur au sien de 1 (génération suivante), et dont le nombre de duplications est 0. En plus, il incrémente le nombre de duplications de son degré partiel source. Il peut aussi communiquer un identificateur, et ne plus s'en servir localement. Dans ce cas, il communique son propre degré partiel sans aucune modification. La notation d'un degré entrant partiel est la suivante :

Notation 5.1
Nous notons un identificateur ou une estampille i accompagnée de sont degré entrant partiel de niveau n, ayant été la source de d duplications et de valeur v par $e(n, d, v)$.

Du coté gestionnaire d'ordonnancement, celui ci additionne les valeurs de tous les degrés partiels. Il découvre le nombre de degrés partiels, en sachant que le nombre de degrés partiels d'un niveau est égal à la somme des duplications des degrés partiels de niveau inférieur. En sachant qu'il existe un seul degré partiel de niveau 0, et en appliquant ce principe récursivement, il connaît le nombre de degrés partiels, et donc la valeur totale du degré entrant. La figure 5.1 page ci-contre montre un exemple d'un scénario possible entre quatre gestionnaires d'estampilles et un gestionnaire d'ordonnancement.

5.3 Gestionnaire d'ordonnancement

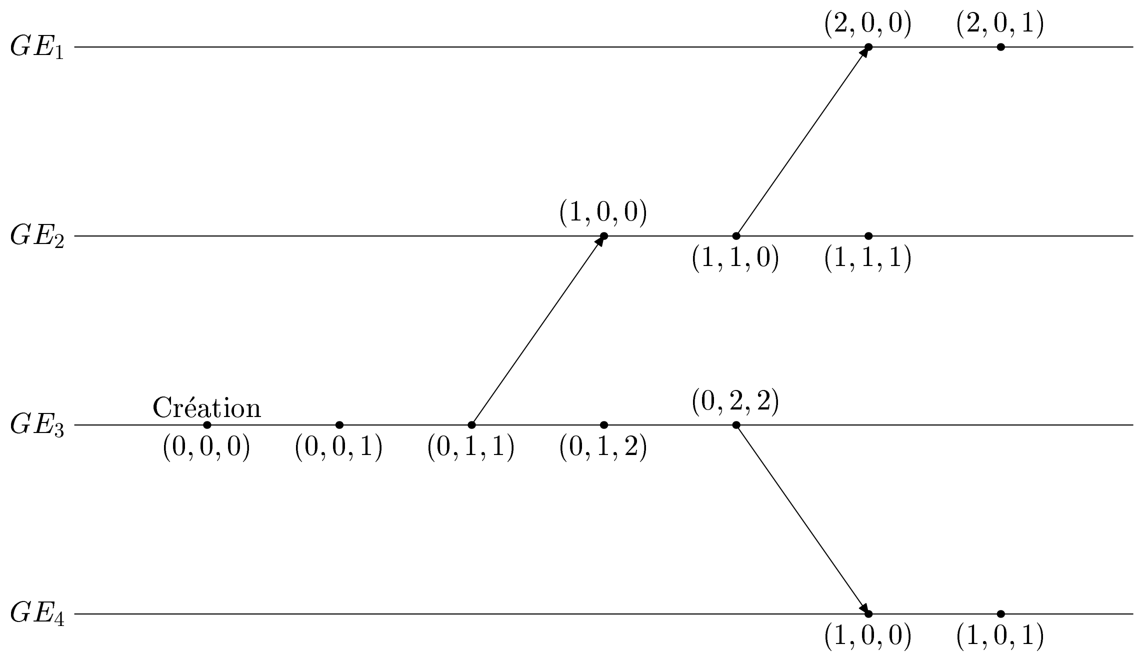
Cette section décrit notre l'implantation du gestionnaire d'ordonnancement. Néanmoins, seules les parties nécessaires à la compréhension des coûts de fonctionnement sont décrites. Nous présentons donc les différents états d'un identificateur (§ 5.3.1) ainsi que les transitions possibles entre ces différents états (§ 5.3.2). Nous décrivons les différentes structures de données employées dans notre implantation (§ 5.3.3), ainsi que les algorithmes des principales opérations (§ 5.3.4). Enfin, nous finissons par l'analyse des coûts d'une invocation dans ce gestionnaire (§ 5.3.5).

5.3.1 État d'un identificateur

Un identificateur représente un groupe, et donc une ou plusieurs invocations. Huit états décrivent un identificateur, ils sont présentés dans le tableau 5.1 page 124.

5.3.2 Transition d'état

Les transitions entre les différents états d'un identificateur sont présentées à la figure 5.2 page 124. Ces 14 transitions sont décrites dans les paragraphes suivants.



L'utilisation de l'identificateur pour estampiller une invocation est représentée par des points. Sa duplication pour transmission à un autre gestionnaire d'estampilles est représentée par des flèches. Nous résumons dans le tableau ci-dessous, la collecte des degrés partiels par un gestionnaire d'ordonnancement. Il reçoit le degré partiel du gestionnaire 4, puis 1, puis 2 et enfin 3.

Niveau	(n,d,v)	$(1,0,1)$	$(2,0,1)$	$(1,1,1)$	$(0,2,2)$
0	1	1	1	1	0
1		-1	-1	-2	0
2			1	0	0
Somme	0	1	2	3	5

Quand tous les niveaux sont à zéro, le gestionnaire sait que la collecte est finie, et donc que la somme est totale. Dans ce cas, le nombre d'utilisations est 5.

Fig. 5.1 – Exemple d'utilisation du degré entrant distribué

Tab. 5.1 – *État d'un identificateur*

État	Description
`Free`	L'identificateur n'est pas utilisé. En fait, c'est un pseudo-état, les identificateurs ayant cet état n'étant pas représentés en mémoire.
`Pending`	Les invocations de cet identificateur ne sont pas encore connues, ses estampilles non plus.
`Waiting`	Les invocations de cet identificateur sont en attente de l'exécution d'autres invocations.
`Ready`	Toutes les invocations de cet identificateur sont prêtes à être exécutées.
`Running`	Au moins une invocation de cet identificateur a été exécutée, ou est en cours d'exécution.
`Executed`	Toutes les invocations en écriture de cet identificateur ont été exécutées.
`Invalid`	Toutes les invocations en lecture non encore exécutées ne peuvent plus s'exécuter.
`Abort`	Toutes les invocations estampillées par cet identificateur ne peuvent plus s'exécuter.

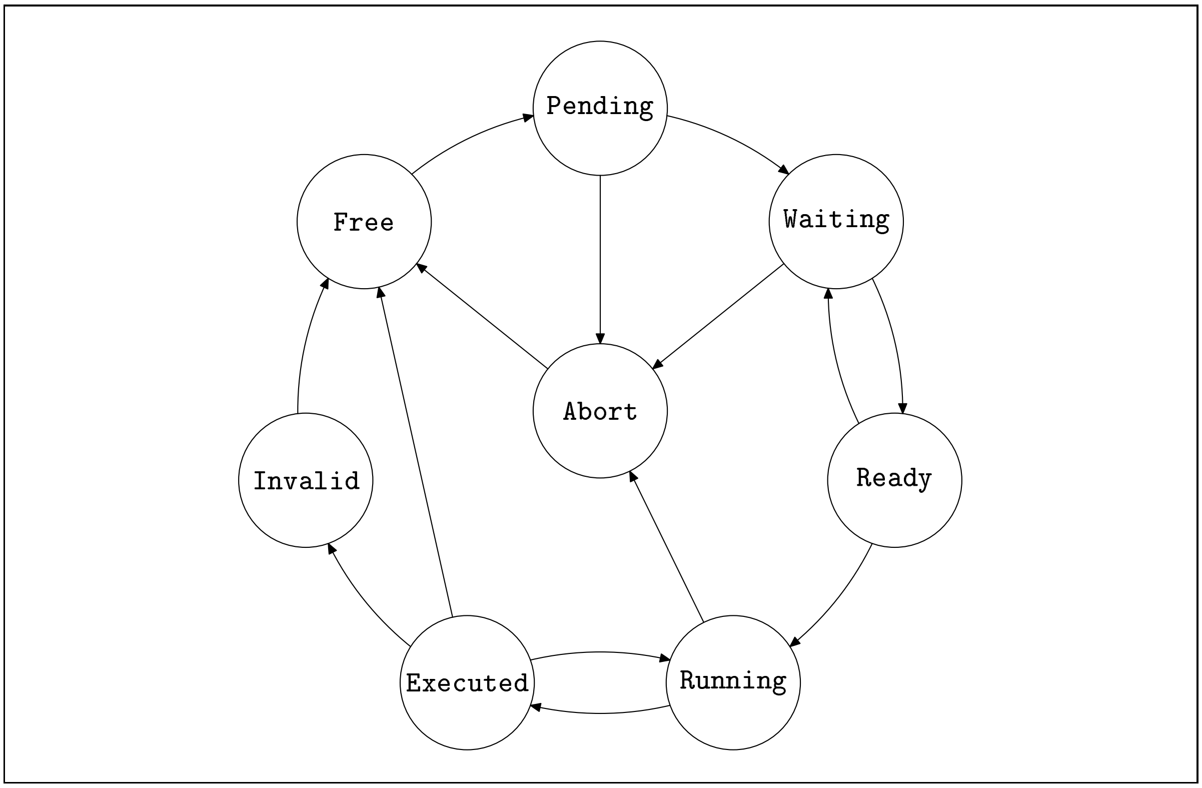


Fig. 5.2 – Transition d'état d'un identificateur d'invocations

5.3.2.1 Transitions du cycle

Les transitions du cycle correspondent aux transitions d'un déroulement normal des opérations.

a) De `Free` à `Pending`

Un identificateur est initialement dans l'état `Free`. Il passe dans l'état `Pending` quand on a la connaissance de son existence, c'est-à-dire quand on reçoit une invocation identifiée ou estampillée avec cet identificateur.

b) De `Pending` à `Waiting`

Le passage de l'état `Pending` à `Waiting` se produit quand on a la connaissance des estampilles de cet identificateur, c'est-à-dire quand on a reçu au moins une invocation identifiée avec cet identificateur.

c) De `Waiting` à `Ready`

Le passage de l'état `Waiting` à `Ready` se produit quand toutes les invocations dont dépendent les invocations de cet identificateur ont été exécutées, c'est-à-dire quand ses estampilles sont toutes dans un état `Executed`.

d) De `Ready` à `Running`

Le passage de l'état `Ready` à `Running` s'effectue quand une invocation avec cet identificateur est en cours d'exécution. Les invocations dont dépend cet identificateur sont alors dans l'état `Invalid`.

e) De `Running` à `Executed`

Le passage de l'état `Running` à `Executed` se produit quand la dernière invocation en lecture connue et toutes les invocations en écriture du groupe identifié avec cet identificateur viennent d'être exécutées avec succès. Le cardinal de cet identificateur est donc connu.

f) De `Executed` à `Invalid`

Le passage de l'état `Executed` à `Invalid` s'effectue quand une invocation dépendante de cet identificateur vient d'être livrée pour exécution à l'objet serveur.

g) De `Executed` ou `Invalid` à `Free`

Enfin, le passage de l'état `Running` ou de l'état `Executed` à `Free` se produit quand le gestionnaire d'ordonnancement n'a plus besoin de cet identificateur. Le degré total entrant est alors connu, puisqu'il sert à déterminer que cet identificateur n'est plus utile. L'utilisation future de cet identificateur représentera donc d'autres invocations.

5.3.2.2 Transitions arrières

Les transitions arrières, sont les transitions de l'état `Ready` à `Waiting`, et de l'état `Executed` à `Running`. Ces deux transitions sont liées. Quand un identificateur est dans l'état `Ready`, cela veut dire que toutes les invocations en écriture ont été exécutées avec succès, et que toutes les invocations en lecture connues aussi. Or, il peut encore exister des invocations en lecture identifiées avec cet identificateur. Si une telle invocation arrive, alors que l'identificateur est dans un état `Executed`, il faut passer cet état à `Running` pour indiquer que maintenant il y a une nouvelle invocation en lecture à exécuter.

Quelques identificateurs estampillés avec cet identificateur étaient peut être dans l'état `Ready`. Par conséquent, il faut repasser ces identificateurs dans l'état `Waiting` pour refléter le fait qu'il y a une nouvelle invocation à attendre.

5.3.2.3 Transitions pour l'abandon

Les transitions pour l'abandon d'invocation font intervenir l'état `Abort`. Elles sont au nombre de quatre.

a) De `Pending` à `Abort`

On passe un identificateur de l'état `Pending` à `Abort`, quand on reçoit une barrière d'invocations dont l'identificateur appartient à l'ensemble (cf. § 4.4.1 page 110). En effet, plus aucune invocation avec cet identificateur n'existe, donc la transition de l'état `Pending` à `Waiting` est impossible. Au moins une invocation s'est perdue, ou n'a pas été émise par une copie défaillante.

b) De `Running` à `Abort`

Cette transition de l'état `Running` à `Abort` se produit à la place de la transition `Running` à `Executed` dans deux cas. Le premier, quand au moins une exécution d'une invocation identifiée par cet identificateur a produit une erreur locale (cf. § 4.3.5.2 page 108). Le deuxième, quand une barrière d'invocation dont l'identificateur appartient à l'ensemble (cf. § 4.4.1 page 110) a été reçue, et qu'il manque donc soit des invocations en écriture identifiées par cet identificateur, soit la connaissance du cardinal du groupe.

c) De `Waiting` à `Abort`

Cette transition de l'état de `Waiting` à `Abort` se passe quand au moins une estampille de l'identificateur est dans un état `Abort`. Cela permet d'abandonner récursivement toutes les invocations qui dépendent d'invocations abandonnées.

d) De `Abort` à `Free`

Enfin, cette transition de l'état `Abort` à `Free` se produit quand le gestionnaire d'ordonnancement n'a plus besoin de cet identificateur, comme toutes les transitions qui amènent vers l'état `Free`.

5.3.3 Structures de données

Cette section décrit les structures de données utilisées dans notre implantation d'un gestionnaire d'ordonnancement. Cette description a pour but de comprendre l'utilisation des ressources, mais surtout de décrire plus facilement les algorithmes et leur coût dans la section suivante. Les principales interactions entre les structures de données sont décrites dans la figure 5.3 page suivante.

Tout d'abord, une table de hachage (`HashTab`) permet de chercher un identificateur donné. Nous avons choisi cette structure, mais toute autre structure de recherche est possible.

Une queue (`ReadyQueue`) permet de lier les identificateurs qui ont des invocations prêtes à être exécutées. L'état des identificateurs appartenant à cette liste est donc soit `Ready`, soit `Running`.

Une autre queue (`WaitQueue`) permet de lier les identificateurs qui attendent qu'une invocation arrive, c'est-à-dire qui dépendent d'un identificateur dont l'état est `Pending`. Cette queue est utilisée pour gérer la politique de propagation « à la demande ».

Une structure d'invocation (`Invoc`) contient des données opaques que lui a fournies le gestionnaire de propagation. Elles sont simplement liées entre elles pour représenter un groupe d'invocations.

Enfin, il existe une structure représentant un identificateur (`Ident`), qui contient son état, son cardinal, son degré entrant total, une liste d'invocations ayant cet identificateur, et une liste de références sur des autres identificateurs. Cette dernière liste fournit la relation inverse représentée par les estampilles. C'est-à-dire quand il existe $I' \rightharpoonup I$, alors il y a une invocation I_i, et une autre $I'_{i'}\langle i \rangle$. La structure de l'identificateur i a alors une référence sur la structure de l'identificateur i'.

Une structure d'identificateur ne référence pas forcément une liste d'autres identificateurs, si aucune autre invocation ne dépend d'elle. Dans ce cas, cette structure n'est pas forcément liée dans la table de hachage si elle n'a pas d'identificateur.

Il n'est pas obligatoire d'avoir une structure de donnée pour fournir la liste des dépendances dans le sens de la relation d'ordre représentée par les estampilles. Ceci allège considérablement les structures de données, les ressources consommées, et le coût des algorithmes.

5.3.4 Algorithmes

Nous considérons uniquement les coûts qui ne sont pas constants. Toutes les listes, et la table de hachage, sont organisées pour supporter des insertions et retraits en temps constant, en choisissant entre un chaînage simple ou double suivant le contexte. Les opérations qui ne sont pas en temps constant sont donc la recherche d'un identificateur dans la table de hachage (R_{Id}) et le balayage de la liste inverse de dépendances (B_{Dep}). La description des algorithmes se borne à faire ressortir ces opérations coûteuses, et non tous les détails de la mise en œuvre.

5.3.4.1 Ajouter une nouvelle invocation

L'ajout d'une nouvelle invocation dans le système par le gestionnaire de propagation conduit aux principales actions suivantes :

1. Si l'invocation a un identificateur, le chercher dans la table de hachage. Si l'invocation n'a pas d'identificateur ou bien si il n'a pas été trouvé, alors créer sa structure ;

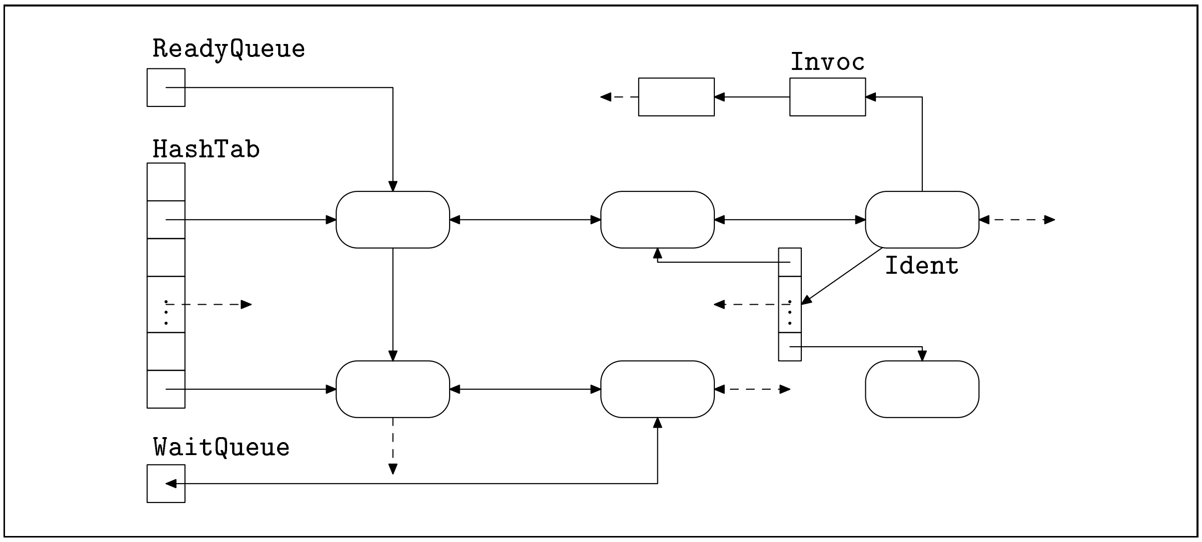


Fig. 5.3 – Structures de données d'un gestionnaire d'ordonnancement

2. Insérer l'invocation dans la liste des invocations de cet identificateur et mettre à jour l'identificateur avec l'éventuel cardinal ou degré entrant partiel que l'invocation transporte ;
3. Si l'invocation est une lecture, vérifier qu'elle n'est pas invalide en balayant la liste inverse des dépendances. Si elle est valide, remettre en attente les éventuels identificateurs en balayant de nouveau la liste (retour arrière dans le cycle d'état) ;
4. Traiter les estampilles de l'invocation. Soit l'état est `Pending` et alors il faut exécuter cette séquence pour chacune des estampilles. Soit l'état est différent, et il faut l'exécuter seulement pour les estampilles transportant une information supplémentaire (degré entrant partiel, cardinal ou surcharge) :
   (a) Chercher l'estampille dans la table de hachage (si elle n'est pas trouvée, la créer) ;
   (b) Créer un lien inverse, c'est-à-dire de la structure de l'estampille jusqu'à la structure de l'identificateur et/ou mettre à jour les informations transportées par cette estampille ;
   (c) Mettre à jour les différents paramètres (décrémentation du degré entrant total, incrémentation du nombre de références, etc).

Le point 1 introduit une recherche d'un identificateur dans la table de hachage (R_{Id}), tandis que le point 2 s'effectue en temps constant. Le point 3, se produit seulement pour des invocations en lecture dans des cas particuliers, et provoque au moins un balayage de la liste inverse des dépendances (B_{Dep}), et est éventuellement suivi d'un autre balayage si cette invocation est valide. Le point 4a provoque une recherche dans la table de hachage, dans le pire des cas, pour chaque estampille de l'invocation. Enfin, les points 4b et 4c se font en temps constant.

5.3.4.2 Délivrer une invocation prête

Pour une politique de type « première prête, première exécutée », délivrer une invocation se résume à trouver immédiatement un identificateur qui possède encore des invocations prêtes grâce à la liste `ReadyQueue`, et dans cet identificateur à prendre une invocation. Éventuellement, il faut retirer l'identificateur de cette liste si on délivre sa dernière invocation. Toutes ces opérations sont effectuées en temps constant.

5.3.4.3 Délivrer un identificateur attendu

Quand l'objet contrôle demande un identificateur qui bloque des invocations, mais qui n'est pas encore délivré à l'objet, il suffit de prendre un identificateur dans la liste `WaitQueue`, et de le retirer de cette liste. La mise à jour de la liste se fait par des opérations similaires, aucun balayage n'est utile. Ces actions sont donc effectuées en temps constant.

5.3.4.4 Notifier la fin de l'exécution d'une invocation

La notification de la fin de l'exécution d'une invocation par l'objet contrôle provoque les actions suivantes :

1. Mettre à jour les différents paramètres (décrémentation du cardinal, de l'état, etc) ;
2. Si c'était la dernière invocation de l'identificateur, balayer la liste des dépendances pour indiquer que cet identificateur ne les bloque plus, en passant éventuellement certains d'entre eux dans l'état prêt.

Le point 1 se fait en temps constant, alors que le point 2 provoque, quelquefois, un balayage de la liste inverse des dépendances.

5.3.5 Analyse du coût d'une invocation

Le coût d'une invocation émise pour le gestionnaire d'ordonnancement dépend de nombreux paramètres. Nous les explicitons sans considérer l'abandon d'invocations.

Par mesure de simplicité, nous examinons d'abord le cas où un groupe est toujours constitué d'une seule invocation. Son cardinal est donc propagé par son identificateur. Cet identificateur possède e estampilles, l'invocation dépend donc de e autres invocations. Enfin, cet identificateur est utilisé pour estampiller d autres identificateurs, c'est-à-dire qu'il existe d invocations qui dépendent d'elle. Le coût d'une invocation en écriture est alors :

$$(1 + e) \times R_{Id} + B_{Dep}$$

En effet, quand l'invocation est délivrée, son identificateur est recherché (R_{Id}). Puis, les estampilles de cet identificateur sont recherchées ($e \times R_{Id}$). Enfin, l'identificateur passe à l'état exécuté, et donc il y a balayage de ses dépendances pour débloquer les éventuelles invocations qui dépendent de lui (B_{Dep}). Le paramètre d spécifie la taille de la liste inverse des dépendances, le coût du balayage de la liste (B_{Dep}) est donc proportionnel à d.

Le coût d'une invocation en lecture est légèrement différent. D'abord, il existe en plus le mécanisme de surcharge. Mais en prenant le cas où le gestionnaire d'estampilles se base sur le graphe de dépendances transformé, il n'y a pas de surcoût entraîné par cette surcharge. Par contre, une invocation en lecture peut arriver en retard, et générer un retour arrière dans le cycle normal d'états des identificateurs. Son coût maximum est donc :

$$(1 + e) \times R_{Id} + 4 \times B_{Dep}$$

En effet, un retour arrière entraîne un balayage de la liste inverse des dépendances pour vérifier que cet identificateur est valide. Dans ce cas, il génère un balayage supplémentaire pour passer les invocations qui étaient prêtes en attente, et un autre balayage pour les repasser prêtes. C'est un coût maximum car le retour arrière est un cas rare, il se passe sur des copies non synchrones, qui reçoivent quand même l'invocation. Il faut donc que l'invocation en lecture soit diffusée plus largement que le nombre de copies synchrones. De plus, un gestionnaire d'ordonnancement peut décider de ne pas implanter ce mécanisme.

Nous détaillons maintenant le coût d'un groupe d'invocations. Dans ce cas, un identificateur désigne un groupe de i invocations, dont l sont des lectures. Sur toutes les invocations en lecture de ce groupe, r d'entre elles ($r \leq l$) arrivent en retard par rapport à l'exécution du groupe, et parmi elles, v sont quand même valides ($v \leq r \leq l$). En fait, v correspond au nombre de retour arrière dans le cycle normal d'états. Il reste un paramètre s, que nous expliquons après. Le coût par invocation d'un tel groupe est :

$$\left(1 + \frac{e+s}{i}\right) \times R_{Id} + \left(\frac{1+r+2v}{i}\right) \times B_{Dep}$$

L'explication du coût du groupe entier est le suivant. À chaque fois qu'une invocation arrive, son identificateur est recherché ($i \times R_{Id}$). Une seule fois, les estampilles de cet identificateur sont recherchées ($e \times R_{Id}$). Enfin, à chaque fois qu'une estampille transporte une information supplémentaire, elle est recherchée aussi ($s \times R_{Id}$). Chaque identificateur passe au moins une fois à l'état exécuté, et donc il y a balayage de ses dépendances pour débloquer les éventuelles invocations qui dépendent de lui (B_{Dep}). À chaque fois qu'une invocation en lecture arrive après que toutes les invocations en écriture du même groupe soient exécutées, il faut rechercher si cet identificateur est valide ou non ($r \times B_{Dep}$). Et si il est valide, il faut faire un retour arrière dans le cycle d'états, c'est-à-dire qu'il faut balayer une fois les dépendances pour les bloquer de nouveau, et une autre fois pour les débloquer quand elles auront été exécutées ($2v \times B_{Dep}$). Tout ceci doit être divisé par i pour trouver le coût par invocation du groupe.

Le paramètre s dépend du nombre de fois que l'identificateur, utilisé en estampille, transporte des informations supplémentaires (cardinal ou degré entrant partiel propagé par les estampilles, surcharge d'estampilles). Dans le cas le plus défavorable, toutes les d estampilles contiennent une information (par exemple si chaque estampille a été émise par des copies différentes). Mais une recherche de ces identificateurs est déjà effectuée pour mettre à jour les dépendances du groupe. Nous obtenons donc $s \leq d - 1$.

La comparaison du coût entre un groupe de i invocations, et i groupes d'une invocation est la suivante. Dans le cas peu probable où $v = r = l = i$, c'est-à-dire que chaque invocation d'un groupe en lecture arrive en retard individuellement (et donc provoque i retours en arrière dans le cycle normal d'états), la différence de coût est en faveur des groupes pour le facteur de B_{Dep} puisqu'il provoque $i - 1$ fois moins de balayage. Mais le coût de B_{Dep} est plus faible que celui de R_{Id}. En analysant ces coûts, on peut déduire que le groupe est plus coûteux en recherche dans la table de hachage quand $s > (i-1) \times e$, et donc dans le cas extrêmement défavorable $d > (i-1) \times e + 1$. Mais ces coûts sont des coûts maximum, et il n'est pas inconcevable de considérer l'utilisation de groupes comme moins coûteuse en général.

Nous pouvons alors déduire le coût maximum d'une invocation dans un groupe en considérant le coût maximum d'une invocation. Dans la grande majorité des cas, une invocation génère dans le gestionnaire d'ordonnancement sur chaque copie, le coût maximum suivant :

$$(1 + e) \times R_{Id} + B_{Dep}$$

Trois remarques importantes peuvent être faites :

1. Le coût $(1+e)\times R_{Id}$ peut être transformé en $\alpha\times R_{Id}+\beta\times B_{Tmp}$ en balayant moins de fois la table de hachage, et en constituant une liste temporaire à balayer. Une implantation future peut très bien utiliser un tel mécanisme pour réduire les coûts ;
2. Une invocation qui n'est pas identifiée, et donc qui n'est pas dans la table de hachage, voit son coût se réduire à zéro. Il s'agit d'une propriété intéressante : si le protocole des estampilles d'ordonnancement n'est pas utilisé, son coût est réduit au minimum ;
3. À aucun moment le nombre de copies d'un objet n'intervient, il s'agit toujours du nombre d'invocations ou de dépendances. Le coût dans les gestionnaires d'ordonnancement est donc proportionnel à l'activité du serveur (le nombre d'invocations), pondéré par le modèle de cohérence (le nombre d'identificateurs et d'estampilles).

5.4 Gestionnaire d'estampilles

L'apparente complexité des estampilles d'ordonnancement peut limiter le bénéfice d'une séparation claire entre les gestionnaires. Or, cette complexité relative de la gestion de des estampilles peut être grandement limitée par l'utilisation d'une librairie qui automatise et cache certaines tâches. Nous décrivons une telle libraire (§ 5.4.1) que nous avons utilisée pour implanter les différents exemples de gestionnaires d'estampilles assurant un modèle de cohérence « à la longue » (§ 5.4.2.1), « lire ses écritures » (§ 5.4.2.2), atomique objet (§ 5.4.2.3) et un mécanisme d'exclusion mutuelle (§ 5.4.2.4).

5.4.1 Librairie d'identificateurs

L'apparente complexité de la gestion des estampilles d'ordonnancement peut limiter la conception et l'utilisation de gestionnaire d'estampilles spécifiques, ou même de cohérences de base. En effet, un développeur doit comprendre non seulement les bases des différents principes, mais les détails régissant le cardinal et sa propagation, le degré entrant partiel et sa propagation, les estampilles surchargées et ses règles, etc.

Or la plupart de ces mécanismes sont automatisables. Par exemple, la librairie que nous avons implantée pour développer des gestionnaires d'estampilles est basée sur sept méthodes. Les classes C++ partielles sont décrites dans le source 5.1 page suivante.

La classe `iss` représente un identificateur. L'utilisateur ne la manipule pas directement, mais par une classe pointeur, `ptrIss`. Cette classe fournit un compteur transparent de références à un identificateur, et peut donc détecter quand un identificateur n'est plus utilisé localement, et ainsi le détruire. Cela permet d'automatiser la propagation du cardinal et des degrés entrant partiels.

Un nouvel identificateur est généré par une classe `issFact` et sa méthode `newIss`. La classe d'identificateur `iss` fournit les méthodes suivantes :

Source 5.1 Librairie d'identificateurs

```
typedef enum { issIdent,
               issStamp,
               issIdentStamp } issWhat;

typedef enum { issClone,
               issGive       } issMode;

typedef enum { issNet,
               issDep,
               issAuto       } issOpti;

class iss
{
public:

    void  addStamp (ptrIss& id);
    void  close    (void);
  card_t  card     (void);
    void  join     (ptrIdd& id);

  size_t  size     (issWhat w, issMode m, issOpti o)            const;
    void* export   (issWhat w, issMode m, issOpti o, void* adr) const;
    void* import   (void* adr);
};

typedef ptrVar<iss> ptrIss;

class issFact
{
public:

  void  newIss     (ptrIss& id);
  ...
};

class deliveryMsg
{
public:

  void  setId      (ptrIss& id);
  ...
};
```

Tab. 5.2 – *Méthode de la classe d'identificateur*

	`class iss`
addStamp	Ajoute une estampille
close	Ferme le groupe
card	Renvoie le cardinal local
join	Fusionne avec un identificateur identique
size export	 Transmet l'identificateur
import	Récupère un identificateur

Pour transmettre un identificateur à un autre gestionnaire d'estampilles, le développeur utilise la méthode `export` avec plusieurs paramètres qui lui permettent de décrire dans quel but il veut transmettre cet identificateur. Les tableaux suivants indiquent les différents choix.

Tab. 5.3 – *But de la transmission de l'identificateur*

	`enum issWhat`
issIdent	Pour identifier
issStamp	Pour estampiller
issIdentStamp	Pour identifier et estampiller

Tab. 5.4 – *Méthode de transmission de l'identificateur*

	`enum issMode`
issGive	Donne l'identificateur (plus de copie local)
issClone	Duplique l'identificateur

Tab. 5.5 – *Optimisation de la transmission de l'identificateur*

	`enum issOpti`
issNet	Optimise le coût réseau
issDep	Optimise les dépendances de propagation
issAuto	Optimisation laissée à la librairie

Enfin, la méthode `setId` de la classe `deliveryMsg` permet d'affecter un identificateur à une invocation. La seule règle générale pour le développeur de gestionnaires d'estampilles est d'effectuer les opérations le plus tôt possible (fermeture d'un groupe, déréférencement d'un identificateur, etc). Avec cette unique règle, la librairie d'estampilles se chargera de manipuler et propager le cardinal au bon moment, de manipuler les degrés entrant partiels et de les propager le plus tôt possible, de surcharger les estampilles jusqu'à ce qu'il ne soit plus nécessaire de le faire, etc.

Cette librairie est une implantation possible, d'autres peuvent exister pour des langages différents, pour des utilisations différentes, plus simples, ou plus complexes, pour assurer de la tolérance aux fautes, etc.

5.4.2 Exemple de gestionnaires d'estampilles

Ces exemples de gestionnaire d'estampilles sont décrits pour deux raisons. D'une part ils montrent comment certains modèles de cohérence ou de synchronisation peuvent être concrètement implantés par les estampilles d'ordonnancement. D'autre part ils montrent la simplicité de conception de tels gestionnaires, grâce aux protocoles des estampilles d'ordonnancement, et grâce à la librairie d'utilisation de ces estampilles. De plus ils sont identiques à ceux que nous avons décrits au chapitre 3 page 53, fournissant ainsi des scénarii possibles. Ces exemples sont tirés de notre prototype, ils sont donc programmés en C++ et ne traitent ni de la tolérance aux fautes, ni du recyclage des identificateurs.

5.4.2.1 Modèle de cohérence « à la longue »

Le modèle de cohérence « à la longue » (§ 2.3.1.6 page 34) est extrêmement simple à implanter, puisque le gestionnaire de propagation fournit déjà cette caractéristique. Le gestionnaire d'estampilles est donc trivial, puisqu'il est vide. Néanmoins, le source 5.2 qui décrit ce gestionnaire permet de voir les méthodes C++ dérivées de l'interface 4.4 page 92 du gestionnaire d'estampilles.

Source 5.2 Gestionnaires d'estampilles : cohérence « à la longue »

```
stampRet stampProxy::tryStamp (deliveryMsg&, stampPrivate&, const objectQuery*&)
{
  return (stampDone);
}

void stampProxy::update (const deliveryMsg&, const stampPrivate&)
{
}

void stamp::doStamp (deliveryMsg&, stampPrivate&, const objectQuery&, Bool&)
{
}
```

5.4.2.2 Modèle de cohérence « lire ses écritures »

Avec le modèle de cohérence « lire ses écritures » (§ 2.3.3.1 page 42) les invocations sont délivrées dans le même ordre que celui de leur émission par le client. Un exemple d'utilisation avec des estampilles d'ordonnancement a été présenté au § 3.2.6.3 page 60. Ce gestionnaire, purement local, prend place dans le mandataire. Concrètement, il s'agit d'estampiller une invocation avec l'identificateur de la précédente. Deux variantes existent, une sans utilisation de groupe d'invocations, et l'autre avec.

a) Sans groupe

Le source 5.3 page ci-contre décrit son implantation. Une variable est nécessaire, celle de l'identificateur courant, `currId`. Dans le mandataire, quand une invocation est traitée, on sauvegarde l'identificateur courant puisqu'il devient le précédent (ligne 14), et on génère un nouvel identificateur pour l'invocation courante (ligne 15). Si le précédent identificateur existe (c'est-à-dire si ce n'est pas la première invocation du client), on estampille l'identificateur courant avec l'identificateur précédent (ligne 17). Il ne reste plus qu'à identifier réellement l'invocation (ligne 19), et fermer le groupe (ligne 20) puisque ce dernier ne contient qu'une seule invocation à chaque fois. Enfin, la valeur de retour (ligne 22) indique que le mandataire a entièrement identifié et estampillé cette invocation.

Source 5.3 Gestionnaires d'estampilles : « lire ses écritures » (sans groupe)

```
1   class stampProxy
2   {
3   private:
4
5     issFact  factId;
6      ptrIss  currId;
7   };
8
9   stampRet stampProxy::tryStamp (deliveryMsg& deliv, stampPrivate&,
10                                             const objectQuery*&)
11  {
12    ptrIss  prevId;
13
14    prevId = currId;
15    factId.newIss(currId);
16    if ( prevId != NULL )
17      currId->addStamp(prevId);
18
19    deliv->setId(currId);
20    currId->close();
21
22    return (stampDone);
23  }
24
25  void stampProxy::update (const deliveryMsg&, const stampPrivate&)
26  {
27  }
28
29  void stamp::doStamp (deliveryMsg&, stampPrivate&, const objectQuery&, Bool&)
30  {
31  }
```

La figure 5.4 page suivante présente d'une part le graphe de dépendance direct d'une série de deux invocations en lecture suivies d'une invocation en écriture, et d'autre part, les invocations identifiées, estampillées avec le transport éventuel du cardinal, du degré entrant et des surcharges générées par ce gestionnaire.

b) Avec groupe

Cet exemple a pour but de montrer que l'utilisation de groupes n'est pas beaucoup plus compliqué. Il regroupe les invocations en lecture consécutives dans un groupe. Les gains en

$$\begin{tikzcd}
L^1 \\
L^2 \arrow[u] \\
E^3 \arrow[u] \\
L^4 \arrow[u] \\
L^5 \arrow[u] \\
E^6 \arrow[u]
\end{tikzcd}$$

(a) Graphe

1. L_1
2. $L_2\langle 1{:}0\rangle$
3. $E_{3:1}\langle 2{:}0(0,0,1)\langle 1{:}0(0,0,1)\rangle\rangle$
4. $L_4\langle 3\rangle$
5. $L_5\langle 4{:}0\langle 3\rangle\rangle$
6. $E_{6:1}\langle 5{:}0(0,0,1)\langle 4{:}0(0,0,1)\langle 3(0,0,1)\rangle\rangle\rangle$

(b) Estampille

FIG. 5.4 – Exemple d'exécution : « Lire ses écritures » sans groupe

terme de nombre d'identificateurs générés et en nombre d'estampilles surchargées ne sont pourtant pas négligeables.

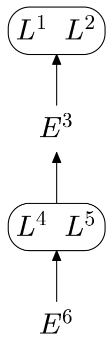
Le source 5.4 page ci-contre est différent du précédent par l'ajout d'une condition (ligne 12). Cette condition indique quand créer un groupe. Il faut que l'invocation soit en écriture, ou bien que ce soit la première invocation du client, ou bien que le cardinal du groupe soit supérieur à un (donc groupe mixte ou en écriture). On peut expliquer cette condition dans l'autre sens, c'est-à-dire qu'on ne crée pas de groupe si l'invocation est une lecture, si ce n'est pas la première émise par le client, et si le groupe actuel est en lecture. Cela regroupe donc bien les invocations en lecture consécutives dans un même groupe.

La création du groupe se fait en créant un nouvel identificateur, et en l'estampillant avec le précédent (ligne 14 à 19). Dans tous les cas, on identifie l'invocation avec l'identificateur courant. On ferme le groupe immédiatement en vertu de la règle « au plus tôt ». En effet, les groupes sont soit constitués d'une seule invocation en écriture, soit de une ou plusieurs invocations en lecture. Dans le premier cas, on peut fermer le groupe, puisqu'il n'y aura plus d'autres invocations dedans. Dans le second cas, puisque les invocations en lecture n'incrémentent pas le cardinal du groupe, celui ci peut être fermé.

La figure 5.5 page 138 présente le graphe de dépendance transformé d'une série de deux invocations en lecture suivies d'une invocation en écriture. Il est accompagné des invocations identifiées, estampillées et des éventuels cardinaux, degrés entrants et surcharges générés par ce gestionnaire.

Source 5.4 Gestionnaires d'estampilles : « lire ses écritures » (avec groupe)

```
class stampProxy
{
private:

  issFact  factId;
   ptrIss  currId;
};

stampRet stampProxy::tryStamp (deliveryMsg& deliv, stampPrivate&,
                                          const objectQuery*& query)
{
  if ( query->getType() == invWrite || currId == NULL || currId->card() > 0 )
  {
    ptrIss  prevId;

    prevId = currId;
    factId.newIss(currId);
    if ( prevId != NULL )
      currId->addStamp(prevId);
  }

  deliv->setId(currId);
  currId->close();

  return (stampDone);
}

void stampProxy::update (const deliveryMsg&, const stampPrivate&)
{
}

void stamp::doStamp (deliveryMsg&, stampPrivate&, const objectQuery&, Bool&)
{
}
```



(a) Graphe

1. L_1
2. L_1
3. $E_{2:1}\langle 1{:}0(0,0,1)\rangle$
4. $L_3\langle 2\rangle$
5. $L_3\langle 2\rangle$
6. $E_{4:1}\langle 3{:}0(0,0,1)\langle 2(0,0,1)\rangle\rangle$

(b) Estampille

FIG. 5.5 – Exemple d'exécution : « Lire ses écritures » avec groupe

5.4.2.3 Modèle de cohérence atomique objet

Le modèle de cohérence atomique objet a été décrit au § 2.3.1.1 page 27 ainsi qu'un exemple d'utilisation avec des estampilles d'ordonnancement au § 3.2.6.4 page 60. Le gestionnaire de cohérence atomique objet est plus difficile à décrire, puisqu'il fait intervenir des interactions entre différents gestionnaires. C'est donc un source partiel qui est présenté, et nous le décrivons tout aussi partiellement.

Le premier source, 5.5 page ci-contre, ne décrit pas les communications, mais uniquement les opérations locales. C'est un gestionnaire qui se place exclusivement du coté des copies, le mandataire indique donc qu'il ne peut ni identifier ni estampiller (ligne 15). Le protocole se base sur des diffusions de demandes et d'invalidations d'un jeton distribué (`token`) qui représente le droit d'émettre des invocations. Plusieurs copies peuvent le posséder, pour estampiller des invocations en lecture, mais une seule copie le possède pour estampiller des invocations en écriture. Dans tous les cas, une seule copie maître gère ce jeton (`master`).

La demande d'identification est simple, si on a le jeton, on obtient un identificateur (`getId`), sinon on le demande aux autres copies en fonction du type de l'invocation (`askId`). Dans le premier cas, cela ressemble à l'exemple précédent, puisqu'il s'agit de créer un groupe dans les mêmes conditions, mais en invalidant tous les jetons (`invToken` ligne 40), et en devenant maître du jeton (ligne 26 et 45) si l'on en crée un. Concrètement, on crée un groupe quand on émet une invocation en écriture ou bien quand on émet une invocation en lecture après une invocation en écriture. Dans les deux cas, il faut donc invalider le jeton pour qu'aucune invocation concurrente ne soit émise par une autre copie.

Le source 5.6 page 140 décrit les demandes effectuées aux autres copies sur le réseau. La méthode `askId` demande un identificateur, en fonction du type de l'invocation (ligne 5). En retour, cet identificateur est fourni, ainsi que l'information pour savoir si on est le maître (ligne 6). Dans tous les cas, on possède alors le jeton, et on n'en a pas encore fait de copie (ligne 7 et 8).

Pour invalider un jeton (`invTokebn`), si on est le maître, on diffuse un message d'invalidation (ligne 16) et on collecte le nombre d'accusés de réception égal au nombre de copies qui existent (ligne 25 à 27). On est alors sûr qu'aucune autre copie possède le jeton, et donc le

Source 5.5 Gestionnaires d'estampilles : cohérence atomique

```
class stamp
{
private:

            Bool  master;
            Bool  token;
   unsigned char  nbrCpy;
       issIdFact  factId;
           ptrId  currId;
  ...
};

stampRet stampProxy::tryStamp (deliveryMsg&, stampPrivate&, const objectQuery*&)
{
  return (stampNo);
}

void stampProxy::update (const deliveryMsg&, const stampPrivate&)
{
}

void stamp::doStamp (deliveryMsg& deliv, stampPrivate&, const objectQuery& query,
                            Bool&)
{
  if ( token )
    master = getId(qury->type()) || master;
  else
    master = askId(query->type());

  deliv->setId(currId);
  currId->close();
}

Bool stamp::getId (invType t)
{
  if ( t == invWrite || currId == NULL || currId->card() > 0 )
  {
    ptrId  prevId;

    invToken();
    prevId = currId;
    factId.newIss(currId);
    if ( prevId != NULL )
      currId->addStamp(prevId);
    return (True);
  }
  return (False);
}
```

droit d'émettre une invocation concurrente. Si on n'est pas le maître, on diffuse à son intention une demande d'invalidation (ligne 16) puisque il est le seul à connaître le nombre de copies qui existent. On attend alors un accusé de réception du maître (ligne 22), qui transmet par la même occasion le droit d'estampiller.

Source 5.6 Gestionnaires d'estampilles : cohérence atomique (émission)

```
1   Bool stamp::askId (invType t)
    {
      Bool  master;

5     // Broadcast (True,   t == invRead)
      // Receive   (master, currId)
      token  = True;
      nbrCpy = 0;
      return (master);
10  }

    void stamp::invToken (void)
    {
      if ( nbrCpy != 0 || !master )
15    {
        // Broadcast (False, master);
        do
        {
          if ( !master )
20        {
            currId = new issId;
            // Receive (currId)
          }
          else
25          // Receive ()
          nbrCpy--;
        } while ( master && nbrCpy != 0 );
      }
      nbrCpy = 0;
30  }
```

Enfin, le source 5.7 page 142 décrit les opérations sur la réception d'un message en provenance d'un autre gestionnaire. Si on ne possède pas le jeton, cela ne nous concerne pas (ligne 6). Dans le cas contraire, si c'est une demande d'identificateur, et si on est le maître (ligne 10 à 14), on demande localement un identificateur pour une invocation de type fourni par le message (ligne 16). S'il y a création d'un groupe, alors on donne l'identificateur pour identification et estampillage (ligne 19), on transmet la responsabilité du maître (ligne 20) et le jeton (ligne 21). En effet, s'il y a création de groupes, aucune autre copie que le demandeur ne doit posséder le droit d'émettre une invocation concurrente. Enfin, s'il n'y a pas création de groupes, on duplique l'identificateur pour identification (ligne 25) et on augmente de 1 le nombre de copies qu'on a faite (ligne 26). En effet, cela correspond au partage du droit d'identifier un invocation en lecture.

Enfin, sur la réception d'une demande d'invalidation, le maître invalide les jetons des autres (`invToken` ligne 37) et donne le droit d'estampiller au demandeur (ligne 38). En effet, le demandeur n'est pas le maître, il n'a donc, dans le meilleur des cas, que le droit d'identifier une invocation. S'il demande l'invalidation, c'est qu'il crée un nouveau groupe, qu'il va estampiller. Il lui faut donc les droits d'estampiller. Enfin, dans tous les cas, il renvoie uniquement un accusé de réception, et se retire le jeton (ligne 42 et 43).

La complexité de ce gestionnaire est due au modèle de cohérence. L'implantation avec une autre méthode que les estampilles d'ordonnancement requiert à peu prés les mêmes communications. Néanmoins, par soucis de simplicité, nous avons adopté une communication par diffusion. Des gestionnaires plus évolués peuvent utiliser, par exemple, des méthodes par indication.

5.4.2.4 Exclusion mutuelle

Ce dernier exemple de gestionnaire implante une exclusion mutuelle comme nous l'avons décrit en exemple au § 3.3.3.2 page 77. Il est différent des précédents, d'une part car il est à la fois dans le mandataire et le gestionnaire, et d'autre part, car il fait intervenir des communications privées entre ce mandataire et le gestionnaire. Par souci de concision, et de compréhension, il ne gère pas la distribution de l'exclusion mutuelle, c'est-à-dire que toutes les invocations doivent arriver sur la même copie primaire.

Le source 5.8 page 143 décrit l'implantation coté mandataire. Une exclusion est demandée en positionnant `toggleMutex` à vrai avant la première invocation, et elle est relâchée en faisant de même avant la dernière invocation.

Si une exclusion mutuelle est demandée (ligne 34 à 38), on en informe le gestionnaire par les données privées. Au retour de l'invocation, `update` est chargée de noter un identificateur que lui renvoie le gestionnaire (ligne 48). Cet identificateur est utilisé pour identifier toutes les invocations à l'intérieur de l'exclusion mutuelle. C'est ce qui se passe quand on est dans l'exclusion mutuelle, et qu'on ne demande pas d'en sortir (ligne 21).

Si on demande à sortir de l'exclusion mutuelle, le mandataire ferme le groupe (ligne 25) puisqu'il ne contiendra plus aucune autre invocation. Il oublie cet identificateur (ligne 26) puisqu'il n'en a plus besoin.

Le gestionnaire est décrit par le source 5.9 page 144. Il gère uniquement le début d'une nouvelle exclusion mutuelle. Il crée un nouvel identificateur (ligne 22) pour identifier cette nouvelle exclusion. Il affecte cet identificateur à la première invocation (ligne 23), les autres seront identifiées par le mandataire. Et enfin, il estampille ce nouvel identificateur avec le précédent (ligne 26), c'est-à-dire qu'il indique que l'exclusion mutuelle précédente doit s'exécuter avant l'exclusion mutuelle courante.

Quand il communique l'identificateur de l'exclusion mutuelle courante au mandataire (ligne 35), il lui donne le droit d'identifier, puisque lui ne s'en servira plus. Le droit d'estampiller n'est pas nécessaire, puisque le gestionnaire l'a déjà effectué à la ligne 26.

Cet exemple montre l'utilisation des données privées entre le gestionnaire et le mandataire. Sans ce mécanisme, à chaque invocation le gestionnaire devrait déterminer le client qui la lui a émise. Il faudrait alors rechercher l'identificateur de l'exclusion mutuelle de ce client dans une table. Donc, cette table est une structure de données dont la taille dépend du nombre de clients demandant une exclusion mutuelle « en même temps ». Le gestionnaire est donc beaucoup moins extensible, c'est-à-dire qu'il supporte beaucoup moins un passage à l'échelle. Dans la solution avec les données privées, chaque mandataire retient l'identificateur de l'exclusion mutuelle qui le concerne. Le gestionnaire n'a à se souvenir que d'un seul identificateur, celui

Source 5.7 Gestionnaires d'estampilles : cohérence atomique (réception)

```
void stamp::recvHandler (void)
{
  Bool  ask, arg;

  // Receive (ask, arg)
  if ( token )
  {
    void* adr = NULL;

    if ( ask ) // Ask Id
    {
      if ( master )
      {
        Bool  remoteMaster;

        remoteMaster = getId((arg) ? invRead : invWrite);
        if ( remoteMaster )
        {
          currId->export(issIdentStamp, issGive, issAuto, adr);
          master = False;
          token  = False;
        }
        else
        {
          currId->export(issIdent, issClone, issAuto, adr);
          nbrCpy++;
        }
        // Reply (remoteMaster, adr);
      }
    }
    else       // Invalidation
    {
      if ( master )
      {
        if ( nbrCpy > 0 )
          nbrCpy--;
        invToken();
        currId->userExport(issStamp, issGive, issAuto, adr);
      }
      if ( arg || master )
      {
        // Reply (adr)
        token = False;
      }
    }
  }
}
```

Source 5.8 Gestionnaires d'estampilles : exclusion mutuelle (coté mandataire)

```
class proxyStamp
{
private:

  ptrIss  currId;
    Bool  inMutex;
    Bool  toggleMutex;
};

class stampPrivate
{
  union { Bool  newSection;
        ptrIss  currId;   }
};

stampRet stampProxy::tryStamp (deliveryMsg& deliv, stampPrivate& private,
                                             const objectQuery*&)
{
  if ( inMutex )
  {
    deliv->setId(currId);

    if ( toggleMutex )
    {
      currId->close();
      currId = (issId*)NULL;

      inMutex     = False;
      toggleMutex = False;
    }
  }
  else if ( toggleMutex )
  {
    inMutex            = True;
    toggleMutex        = False;
    private->newSection = True;

    return (stampNo);
  }

  private->newSection = False;
  return (stampDone);
}

void stampProxy::update (const deliveryMsg&, const stampPrivate& private)
{
  if ( private->currId != NULL )
    currId = private->currId;
}
```

Source 5.9 Gestionnaires d'estampilles : exclusion mutuelle (coté serveur)

```
class stamp
{
private:

  issFact  factId;
   ptrIss  prevSection;
   ptrIss  currSection;
};

class stampPrivate
{
  union { Bool  newSection;
        ptrIss  currId;   }
};

void stamp::doStamp (deliveryMsg& deliv, stampPrivate& private,
                     const objectQuery&, Bool&)
{
  if ( private->newSection )
  {
    prevSection = currSection;
    factId.newIss(currSection);
    deliv->setId(currSection);

    if ( prevSection != NULL )
      currSection->addStamp(prevSection);

    private->currId = currSection;
  }
}

// Serialization stampPrivate::currId
{
  ...
  adr = currId->export(issIdent, issGive, issAuto, adr);
}
```

de la dernière exclusion mutuelle qu'il a générée.

5.5 Évaluation

Bien que les solutions que nous proposons pour améliorer les performances soient multiples, et quelquefois spécifiques à l'application, nous pouvons évaluer des gains dans un contexte précis.

Le test que nous avons effectué utilise un serveur qui possède deux méthodes, une en lecture et une en écriture. Ces deux méthodes prennent chacune un paramètre de même type (en entrée / sortie) pour que leurs emballages n'influencent pas les résultats. Elles simulent toutes les deux un calcul identique.

Nous mesurons le temps cumulé que mettent deux clients à invoquer les méthodes chacun 2000 fois. La proportion d'invocations en lecture par rapport aux invocations en écriture est variable. Le gain de temps est mesuré par rapport à deux configurations différentes. Dans celle de référence, les deux clients sont sur deux machines différentes, et il existe un seul serveur sur l'une d'entre elle. L'autre configuration est l'ajout d'une copie du serveur sur la machine qui n'en possède pas.

La politique du gestionnaire d'ordonnancement est celle décrite dans ce chapitre, c'est-à-dire « première prête, première arrivée ». La politique de choix de la copie pour une invocation en lecture est simplement l'utilisation de la copie locale si elle existe, sinon de la copie distante. Une invocation en écriture est propagée aux copies à partir du client. Chaque copie du serveur transmet alors le résultat au client qui utilise le premier reçu. Enfin, le modèle de cohérence est « lire ses écritures ».

Les deux machines utilisées sont d'une puissance légèrement différente. L'utilisation de machines de puissance strictement identique ne permet pas d'exhiber des situations pénalisantes, et est finalement moins réaliste dans un contexte mondial. Nous avons donc deux courbes, une décrivant le gain obtenu quand la deuxième copie du serveur est placée sur la machine la plus rapide, et l'autre quand la copie ajoutée est placée sur la plus lente. Enfin, nous indiquons une courbe optimale, représentant le gain théorique d'une situation avec deux copies sur deux machines de puissances identiques par rapport à une seule copie sur une des deux machines. Ce gain optimal est tout simplement calculé en fonction du nombre d'invocations exécutées par chacune des copies du serveur. Les résultats sont présentés à la figure 5.6 page suivante.

Nous voyons que l'ajout d'une copie sur une machine plus rapide est toujours bénéfique. La courbe est sur-optimale, puisque la machine est plus rapide, et donc exécute plus d'invocations. En fait, ceci est possible car les clients se synchronisent avec la machine la plus rapide. Le gain varie entre 17 % quand il n'y a que des invocations en écriture, jusqu'à 50 % quand ce ne sont que des invocations en lecture. Dans cette dernière configuration, chaque copie est indépendante et traite les requêtes de leur client local.

L'ajout d'une copie sur une machine lente pénalise l'ensemble des deux clients quand ils font exclusivement des invocations en écriture (-3,78 %). Ceci est dû au traitement des réponses multiples. En effet, les deux clients se synchronisent avec la plus rapide des copies, mais traitent quand même la réponse de la plus lente en l'ignorant. Malgré cela, dès qu'il existe quelques invocations en lecture, un gain se fait sentir (3,74 % avec un rapport de 1 lecture pour 9 écritures). Quand il n'y a que des invocations en lecture, le gain est de 35,5 %.

La proportion d'invocations en lecture par rapport à celles en écriture n'est pas le seul facteur intervenant. La complexité des invocations en terme de temps d'exécution est aussi

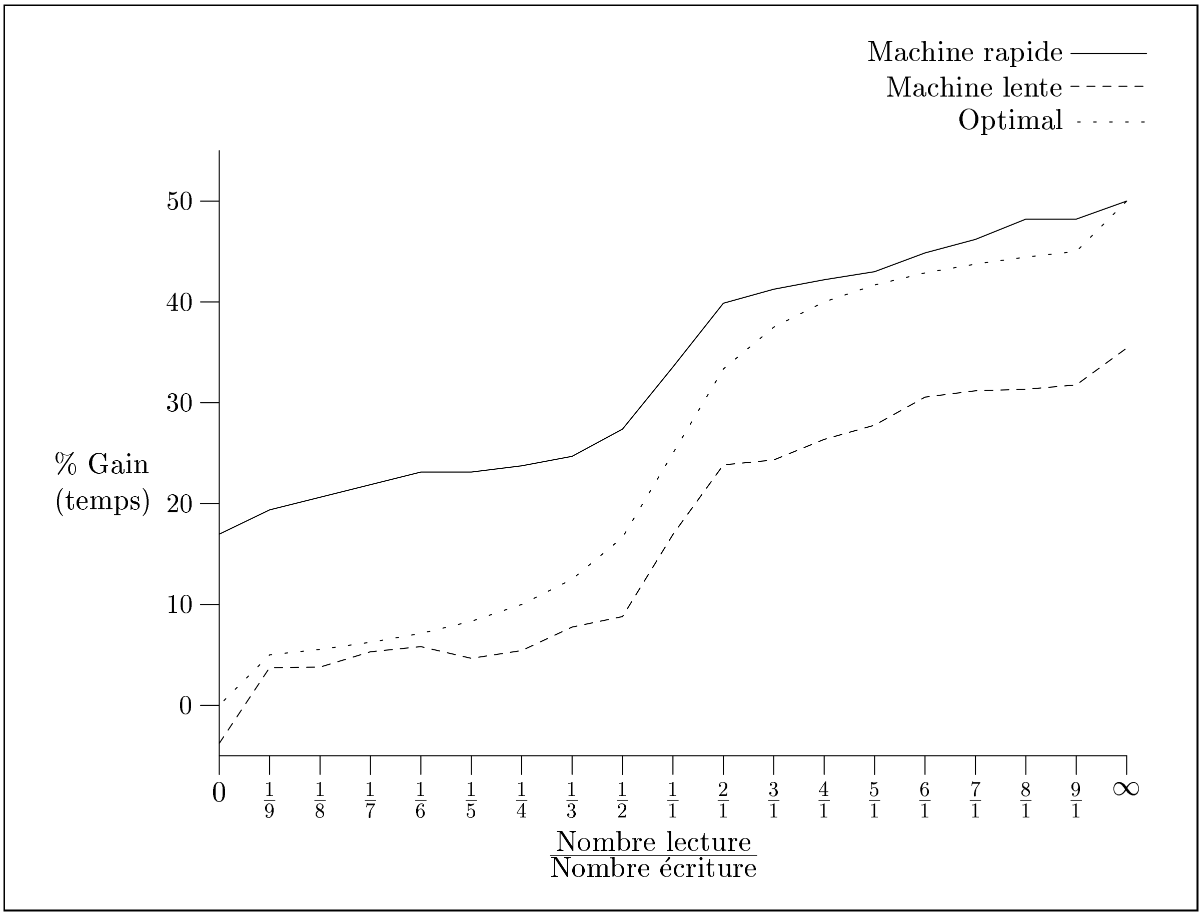


Fig. 5.6 – Gain avec une cohérence « lire ses écritures »

un facteur important. Le tableau 5.6 montre, dans des conditions identiques au précédent test et pour un nombre d'invocations en lecture égal à celui en écriture, comment le gain évolue en fonction de la durée d'exécution des invocations. La durée de l'invocation en écriture est notée t_E et celle en lecture t_L. Le gain est logiquement moins important quand la durée de l'invocation en écriture est plus importante que celle en lecture, puisque cette invocation est exécutée par toutes les copies (donc, dans ce test, deux fois).

Tab. 5.6 – *Gain avec une invocation en lecture pour une en écriture*

Relation de durée	% Gain
$t_E \times 1000 \;=\; t_L$	21,5 %
$t_E \;=\; t_L \times 1000$	16,9 %

D'autres facteurs interviennent, comme le modèle de cohérence ou les canaux de communication. Néanmoins, ces tests montrent qu'il est possible d'obtenir des gains de performance, sans mécanisme supplémentaire de répartition de charge ou de sélection de copie primaire avancée, même dans les cas où une forte proportion d'invocations en écriture est émise. Ce résultat est principalement dû à l'asynchronisme possible entre le client et certaines copies du serveur.

5.6 Conclusion

Ce chapitre décrit notre prototype : un ORB réduit conforme à notre cadre de travail, et utilisant le protocole des estampilles d'ordonnancement. Il nous a permis principalement de montrer les deux points suivants.

Tout d'abord, le gestionnaire d'ordonnancement, gestionnaire local qui doit respecter les principes des estampilles d'ordonnancement, est relativement peu coûteux en ressource mémoire, et en temps processeur. Les coûts engendrés ne dépendent pas du nombre de copies du serveur, mais de l'activité de ce dernier et du modèle de cohérence et de synchronisation que les gestionnaires d'estampilles implantent. De plus, l'utilisateur ne paye pas le coût de ces mécanismes si celui ci ne les utilise pas.

Enfin, les principes des estampilles d'ordonnancement sont facilement simplifiables par l'utilisation de librairies d'identificateurs. Les mécanismes d'emballage, de propagation (partielle ou totale), de stockage des invocations en attente, d'ordonnancement suivant l'ordre décrit, de livraison à l'objet, de synchronisme des copies, de collecte des résultats, etc, sont complètement séparés et dissimulés au développeur d'un gestionnaire d'estampilles. Le développement d'un modèle de cohérence se réduit alors principalement à la conception des mécanismes pour déterminer un ordre sur les invocations, et des éventuelles actions pour assurer la tolérance aux fautes. Néanmoins, la compréhension de quelques notions des estampilles d'ordonnancement est indispensable à ce concepteur, à savoir le rôle des identificateurs, des estampilles, et la notion de cardinal.

Conclusion

Bilan

Notre travail a consisté à étudier et à proposer un cadre de travail et un protocole pour améliorer les performances des systèmes à objets vis-à-vis du temps de réponse aux clients.

Nous avons tout d'abord étudié les techniques de réplication et leurs applications. À notre connaissance, peu de systèmes considèrent la réplication active comme une technique pour améliorer la performance des applications. Elle est essentiellement utilisée pour la tolérance aux fautes. L'étude des principaux modèles de cohérence nous a permis de comprendre leurs avantages et inconvénients respectifs, et d'imaginer une solution pour les implanter avec les mêmes mécanismes de base. Nous avons aussi étudié quelques systèmes à objets, nous permettant d'en isoler les fonctionnalités et de proposer un cadre de travail s'adaptant facilement à des systèmes existants.

Mais pour satisfaire notre objectif d'amélioration des performances, notre travail a été principalement inspiré et guidé par les trois arguments suivants :

1. L'argument « de bout en bout » préconise de laisser à l'application certaines tâches, qu'elle exécute de manière plus performante qu'un mécanisme de niveau inférieur, car ayant plus de connaissances sur son rôle. Appliqué au maintien de la cohérence d'un serveur répliqué, cet argument est un point des plus importants pour améliorer les performances. Néanmoins, nous sommes bien conscients que cette solution n'est pas universelle, et que des considérations de coûts financiers et humains, ne permettent pas son application systématique ;
2. L'adaptation des fonctionnalités permet à un mécanisme de tirer parti de son environnement. Il n'est pas question d'imposer une conception particulière pour chaque environnement possible, mais de laisser la possibilité de le faire ;
3. Supporter les coûts des seules fonctionnalités utilisées, et non de toutes celles présentes dans le système permet de proposer des fonctionnalités plus spécifiques, mais sans pénaliser les situations générales.

Ces trois arguments nous ont conduits à proposer un protocole d'estampilles d'ordonnancement et un cadre de travail qui permet à un serveur d'être répliqué pour en améliorer les performances.

Ainsi, l'argument « de bout en bout » est utilisé afin de proposer une méthode simple pour concevoir des modèles de cohérences spécifiques à l'application. Le protocole des estampilles d'ordonnancement permet à une invocation de transporter avec elle toutes les informations nécessaires à sa livraison dans un ordre imposé par un gestionnaire d'estampilles. L'implantation de ce gestionnaire se résume alors à déterminer un ordre sur les invocations. Cela dissimule les mécanismes de propagation, de stockage, de remise en ordre et de livraison des requêtes à l'application. Nous espérons ainsi augmenter l'utilisation de cohérences spécifiques à l'application en proposant une solution simple et efficace.

L'adaptation des fonctionnalités à leur environnement a influencé le découpage d'un serveur en gestionnaires fournissant des fonctionnalités précises. L'exemple le plus représentatif est sans doute le gestionnaire de propagation qui permet de s'adapter à son environnement réseau sans répercutions sur les autres gestionnaires.

Enfin, tous les mécanismes ont des coûts négligeables s'ils ne sont pas utilisés. Le protocole des estampilles d'ordonnancement respecte lui-même cette règle, tout comme le recyclage des identificateurs, le support de la tolérance aux fautes, etc.

Perspectives

Les perspectives pouvant être envisagées à ce travail sont nombreuses, car elles concernent d'une part les systèmes distribués à objets, et d'autre part l'amélioration des performances. Ainsi, la conception d'un gestionnaire de répartition de charge utilisant les statistiques du gestionnaire d'ordonnancement est une voie possible. L'intégration de ce travail dans un ORB complet en est une autre. Néanmoins, les deux perspectives suivantes nous paraissent les plus intéressantes d'un point de vue recherche.

a) Gestionnaire de propagations

Le gestionnaire de propagation est une partie importante pour obtenir un système opérationnel. La conception d'un gestionnaire de propagation est très ouverte. Il peut être dédié à un certain type de réseau, avec une journalisation sur des disques pour gérer des machines déconnectées, ou pour atténuer des pics d'utilisation. Dans tous les cas, c'est une partie qui a des répercutions non négligeables sur les performances.

Pour assurer la tolérance aux fautes entre le client et le serveur sans que le gestionnaire d'estampilles n'intervienne, la conception d'un gestionnaire de propagation effectuant des transactions entre le mandataire et la copie primaire est une autre voie à explorer.

Dans tous les cas, grâce au cadre de travail, ces voies n'interfèrent pas avec les autres gestionnaires, et peuvent donc être envisagées séparément.

b) Composition des gestionnaires d'estampilles

L'utilisation d'un seul gestionnaire d'estampilles est actuellement possible. La composition de plusieurs d'entre eux est une perspective intéressante. L'idée est de les structurer, par exemple en pile, pour qu'une invocation soit identifiée et estampillée par plusieurs d'entre eux. Par exemple, avec la composition d'un gestionnaire assurant l'exclusion mutuelle avec des autres gestionnaires assurant un modèle de cohérence particulier, on peut imaginer un modèle de cohérence « lire ses écritures » en dehors d'une exclusion mutuelle, et un modèle de cohérence « à la longue » à l'intérieur.

Si techniquement cette solution est relativement facile, le problème majeur est de maintenir un graphe de dépendance sans circuit, pour satisfaire le principe de vivacité des estampilles d'ordonnancement.

Bibliographie

[Afek et al., 1993] AFEK, Y., BROWN, G., et MERRITT, M. (1993). « Lazy caching ». *ACM Transaction on Programming Languages and Systems*, 15(1):182–205.

[Agrawal et al., 1994] AGRAWAL, D., CHOY, M., LEONG, H., et SINGH, A. (1994). « Mixed consistency: A model for parallel programming ». Dans *Proceeding of the 13th Symposium on Principles of Distributed Computing*, pages 101–110, Los Angeles, USA.

[Ahamad et al., 1991] AHAMAD, M., BURNS, J., HUTTO, P., et NEIGER, G. (1991). « Causal Memory ». Dans *Proceeding of the 5th International Workshop in Distributed Algorithms*, numéro 579 dans Lecture Notes in Computer Sciences, pages 9–30, Delphi, Greece. Springer-Verlag.

[Ahamad et al., 1995] AHAMAD, M., HUTTO, P., NEIGER, G., BURNS, J., et KOHLI, P. (1995). « Causal Memory: Definitions, Implementations and Programming ». *Distributed Computing*, 9(1):37–49.

[Amza et al., 1996] AMZA, C., COX, L., DWARKADAS, S., KELEHER, P., LU, H., RAJAMONY, R., YU, W., et ZWAENEPOEL, W. (1996). « TreadMarks: Shared Memory Computing on Networks of Workstations ». *IEEE Transactions on Computers*, 29(2):18–28.

[Anada et al., 1991] ANADA, A., TAY, B., et KOTH, E. (1991). « ASTRA—An Asynchronous Remote Procedure Call Facility ». Dans *Proceeding of the 11th International Conference on Distributed Computing Systems*, pages 172–179, Arlignton, Texas, USA. IEEE Computer Society.

[Attiya et al., 1993] ATTIYA, H., CHAUDHURI, S., FRIEDMAN, R., et WELCH, J. (1993). « Shared memory consistency conditions for non sequential executions: definitions and programming strategies ». Dans *Prooceeding of the 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 241–250, Vale, Germany.

[Attiya et Friedman, 1992] ATTIYA, H. et FRIEDMAN, R. (1992). « A Correctness Condition for High-Performance Multiprocessors ». Dans *Proceedings of the 24th Annual Symposium on the Theory of Computing*, pages 679–690, Victoria, British Columbia, Canada.

[Attiya et Welch, 1994] ATTIYA, H. et WELCH, J. (1994). « Sequential consistency versus linearisability ». *ACM Transaction on Computer Systems*, 12(2):91–122.

[Bal et al., 1992] BAL, H., KAASHOEK, M., et TANENBAUM, A. (1992). « Experience with distributed programming in Orca ». *IEEE Transactions on Software Engineering*, 18(4):190–205.

[Ballintijn et al., 1997] BALLINTIJN, G., SANDBERG, M., et van STEEN, M. (1997). « Scheduling Concurrent RPCs in the Globe Location Service ». Dans *Proceeding of the 3rd annual ASCI Conference*, pages 28–33, Heijen, The Netherlands.

[Beaugendre et al., 1997] BEAUGENDRE, P., PRIOL, T., et RENÉ, C. (1997). « Cobra: a CORBA-compliant Programming Environment for High-Performance Computing ». Rapport Technique PI-1141, IRISA, Rennes, France.

[Berge, 1963] BERGE, C. (1963). *Théorie des graphes et ses applications.* Dunod, Paris, deuxième edition.

[Bershad et Zekauskas, 1991] BERSHAD, B. et ZEKAUSKAS, M. (1991). « Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors ». Rapport Technique CMU-CS-91-170, Carnegie Mellon University, Department of Computer Science, Pittsburgh, PA, USA.

[Bershad et al., 1993] BERSHAD, B., ZEKAUSKAS, M., et SAWDON, W. (1993). « The Midway Distributed Shared Memory System ». Rapport Technique CMU-CS-93-119, Carnegie Mellon University, Department of Computer Science, Pittsburgh, PA, USA.

[Birman, 1993] BIRMAN, K. (1993). « A Response to Cheriton and Skeen's Criticism of Causal and Totally Ordered Communication ». Rapport Technique, Cornell University, Department of Computer Science Research, Ithaca, New York, USA.

[Birman et Glade, 1995] BIRMAN, K. et GLADE, B. (1995). « Reliability Through Consistency ». *IEEE Software*, 12(3):29–41.

[Birman et Joseph, 1987] BIRMAN, K. et JOSEPH, T. (1987). « Reliable Communication in the Presence of Failures ». *ACM Transaction on Computer Systems*, 5(1):47–76.

[Birman et al., 1985] BIRMAN, K., JOSEPH, T., RAEUCHLE, T., et ABADI, A. (1985). « Implementing Fault-Tolerant Distributed Objects ». *IEEE Transactions on Software Engineering*, 11(6):502–508.

[Birman et van Renesse, 1994] BIRMAN, K. et van RENESSE, R. (1994). *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press.

[Birrell et al., 1982] BIRRELL, A., LEVIN, R., NEEDHAM, R., et SHROEDER, M. (1982). « Grapevine: An exercise in distributed computing ». *Communications of the ACM*, 25(4):260–274.

[Birrell et Nelson, 1984] BIRRELL, A. et NELSON, B. (1984). « Implementing Remote Procedure Calls ». *ACM Transaction on Computer Systems*, 2(1):39–59.

[Brun-Cottan et Makpangou, 1995] BRUN-COTTAN, G. et MAKPANGOU, M. (1995). « Adaptable Replicated Objects in Distributed Environments ». Rapport Technique RR-2593, INRIA, Rocquencourt, France.

[Cabillic, 1997] CABILLIC, G. (1997). « *Exécution d'applications parallèles sur architecture distribuées hétérogènes : propositions et mise en oeuvre* ». PhD thesis, IRISA, Rennes, France.

[Carriero et Gelernter, 1986] CARRIERO, N. et GELERNTER, D. (1986). « The S/Net's Linda kernel ». *ACM Transaction on Computer Systems*, 4(2):110–129.

[Carte, 1995] CARTE, J. (1995). « Design of the Munin Distributed Shared Memory System ». *Journal of Parallel and Distributed Computing*, 29(2):219–227.

[Carter et al., 1991] CARTER, J., BENNETT, J., et ZWAENEPOEL, W. (1991). « Implementation and performance of Munin ». Dans *Proceeding of the 13th Symposium on Operating Systems Principles*, volume 25(5) de *Operating Systems Review*, pages 152–164, Pacific Grove, California, USA.

[Censier et Feautrier, 1978] Censier, L. et Feautrier, P. (1978). « A New Solution to Coherence Problems in Multicache Systems ». *IEEE Transactions on Computers*, 27(12):1112–1118.

[Cheriton, 1988] Cheriton, D. (1988). « Problem-Oriented Shared Memory: A Decentralized Approach to Distributed System Design ». Dans [ICDCS, 1988], pages 190–197.

[Cheriton et Skeen, 1993] Cheriton, D. et Skeen, D. (1993). « Understanding the Limitations of Causally and Totally Ordered Communication ». Dans *Proceeding of the 14th Symposium on Operating Systems Principles*, Operating Systems Review, Asheville, North Carolina, USA.

[Cheriton et Skeen, 1994] Cheriton, D. et Skeen, D. (1994). « Comments on the Responses by Birman, Van Renesse, and Cooper ». *Operating Systems Review*, 28(1):32.

[Cheriton et Zwaenepoel, 1985] Cheriton, D. et Zwaenepoel, W. (1985). « Distributed Process Groups in the V Kernel ». *ACM Transaction on Computer Systems*, 3(2):77–107.

[Cooper, 1994] Cooper, R. (1994). « Experience with Causally and Totally Ordered Communication Support ». *Operating Systems Review*, 28(1):28–32.

[COOTS, 1997] COOTS (1997). *Proceedings of the 3rd USENIX 1997 Conference on Object-Oriented Technologies*, Portland, Oregon, USA. USENIX Association.

[DeCleene et al., 1997] DeCleene, B., Bhattacharaya, S., Friedman, T., Keaton, M., Kurose, J., Rubenstein, D., et Towsley, D. (1997). « Reliable Multicast Framework (RMF) ». White paper, University of Massachusetts, Department of Computer Science, Amherst, Massachusetts, USA. Aussi à l'Url : `http://www.tascnets.com/mist/RMF/RMFWP.htm`.

[Deering, 1989] Deering, S. (1989). « Host Extensions for IP Multicasting ». Request For Comments 1112.

[Demers et al., 1994] Demers, A., Petersen, K., Spreitzer, M., Terry, D., Theimer, M., et Welch, B. (1994). « The Bayou Architecture: Support for Data Sharing among Mobile Users ». Dans *Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 2–7, Santa Cruz, California, USA. IEEE Computer Society.

[Dubois et al., 1986] Dubois, M., Scheurich, C., et Briggs, F. (1986). « Memory Access Buffering in Multiprocessors ». Dans *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, Tokyo, Japan. IEEE Computer Society.

[Dumoulin, 1997] Dumoulin, C. (1997). « *Dream: Une mémoire partagée répartie à cohérence programmable* ». PhD thesis, Université des Sciences et Technologies de Lille, LIFL, Villeneuve d'Ascq, France.

[Felber et al., 1996] Felber, P., Garbinato, B., et Guerraoui, R. (1996). « The Design of a CORBA Group Communication Service ». Dans *Proceedings of the 15th Symposium on Reliable Distributed Systems*, pages 150–159, Niagara on the lake, Ontario, Canada. IEEE Computer Society.

[Felber et al., 1997] Felber, P., Guarraoui, R., et Schiper, A. (1997). « A CORBA Object Group Service ». Accepted for the ECOOP'97 Workshop on Corba, Aussi à l'Url : `ftp://ftp-lse.epfl.ch/pub/felber/papers/ECOOP-97.ps`.

[Fisher et al., 1985] Fisher, M., Lynch, N., et Paterson, M. (1985). « Impossibility of Distributed Consensus with One Faulty Process ». *Journal of the ACM*, 32(2):374–382.

[Garcia-Molina et Barbara, 1985] Garcia-Molina, H. et Barbara, D. (1985). « How to design votes in a distributed system ». *Journal of the ACM*, 32(4):841–850.

[Geist et al., 1994] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., et Sunderam, V. (1994). *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press.

[Gharachorloo et al., 1990] Gharachorloo, K., Lenoski, D., Maudon, J., Gibbons, P., Gupta, A., et Hennessy, J. (1990). « Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors ». Dans *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, Seattle, Washington, USA. IEEE Computer Society.

[Gifford, 1979] Gifford, D. (1979). « Weighted Voting for Replicated Data ». Dans *Proceeding of the 7th Symposium on Operating Systems Principles*, pages 150–162, Pacific Grove, California, USA.

[Goodman, 1989] Goodman, J. (1989). « Cache Consistency and Sequential Consistency ». Rapport Technique 1006, Computer Science Department, University of Wisconsin-Madison, Madison, Wisconsin, USA. IEEE Scalable Coherence Interface Working Group.

[Hadzilacos et Toueg, 1993] Hadzilacos, V. et Toueg, S. (1993). « *Fault-Tolerant Broadcasts and Related Problems* », Chapitre 5, pages 97–145. Dans [Mullender, 1993], second edition.

[Herlihy et Wing, 1990] Herlihy, M. et Wing, J. (1990). « Linearizability: A Correctness Condition for Concurrent Objects ». *ACM Transaction on Programming Languages and Systems*, 12(3):463–492.

[Holbrook et al., 1995] Holbrook, H., Singhal, S., et Cheriton, D. (1995). « Log-Based Receiver-Reliable Multicast for Distributed Interactive Simulation ». Dans *Proceeding of the 1995 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Cambridge, Massachusetts, USA. ACM.

[Homburg et al., 1995] Homburg, P., van Doorn, L., van Steen, M., Tananbaum, A., et de Jonge, W. (1995). « An Object Model for Flexible Distributed Systems ». Dans *Proceeding of the 1st annual ASCI Conference*, pages 69–78, Heijen, Netherlands.

[Hutto et Ahamad, 1990] Hutto, P. et Ahamad, M. (1990). « Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories ». Dans *Proceeding of the 10th International Conference on Distributed Computing Systems*, pages 302–311, Paris, France. IEEE Computer Society.

[ICDCS, 1988] ICDCS (1988). *Proceeding of the 8th International Conference on Distributed Computing Systems*, San José, California, USA. IEEE Computer Society.

[II, 1994] II (1994). « *An Introduction to Orbix + ISIS* ». IONA Techonogies Ltd. and Isis Distributed Systems, Inc.

[Kaashoek et Tanenbaum, 1994] Kaashoek, M. et Tanenbaum, A. (1994). « Efficient Reliable Group Communication for Distributed Systems ». Rapport Technique, Vrije Universiteit, Department of Math and Computer Science, Amsterdam, The Netherlands.

[Keleher, 1995] Keleher, P. (1995). « *Lazy Release Consistency for Distributed Shared Memory* ». PhD thesis, Rice University, Departement of Computer Science, Houston, Texas, USA.

[Kermarrec et al., 1998] Kermarrec, A.-M., Kuz, T., van Steen, M., et Tanenbaum, A. (1998). « A Framework for Consistent, Replicated Web Objects ». Dans *Proceeding of the 18th International Conference on Distributed Computing Systems*, pages 276–284, Amsterdam, The Netherlands. IEEE Computer Society.

[Kistler et Satyanarayanan, 1992] KISTLER, J. et SATYANARAYANAN, M. (1992). « Disconnected Operation in the Coda File System ». *ACM Transaction on Computer Systems*, 10(1):3–24.

[Ladin et al., 1990] LADIN, R., LISKOV, B., et SHRIRA, L. (1990). « Lazy Replication: Exploiting the Semantics of Ditributed Services ». Dans *Proceeding of the 9th Symposium on Principles of Distributed Computing*, pages 43–57, New York, USA.

[Ladin et al., 1992] LADIN, R., LISKOV, B., SHRIRA, L., et GHEMAWAT, S. (1992). « Providing High Availability Using Lazy Replication ». *ACM Transaction on Computer Systems*, 10(4):360–391.

[Lamport, 1978] LAMPORT, L. (1978). « Time, Clocks and the Ordering of Events in a Distributed System ». *Communications of the ACM*, 21(7):558–565.

[Lamport, 1979] LAMPORT, L. (1979). « How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs ». *IEEE Transactions on Computers*, 28(9):690–691.

[Lamport et al., 1982] LAMPORT, L., SHOSTAK, R., et PEASE, M. (1982). « The Byzantine Generals Problem ». *ACM Transaction on Programming Languages and Systems*, 4(3):382–401.

[Li et Hudak, 1989] LI, K. et HUDAK, P. (1989). « Memory Coherence in Shared Virtual Memory Systems ». *ACM Transaction on Computer Systems*, 7(4):321–359.

[Lipton et Sandberg, 1988] LIPTON, L. et SANDBERG, J. (1988). « PRAM: A scalable shared memory ». Rapport Technique CS-TR-180-88, Princeton University, Departement of Computer Science.

[Little, 1991] LITTLE, M. (1991). « *Object Replication in a Distributed System* ». PhD thesis, University of Newcastle.

[Maffeis, 1995] MAFFEIS, S. (1995). « Adding Group Communication and Fault-Tolerance to CORBA ». Dans *Proceedings of the 1st USENIX 1995 Conference on Object-Oriented Technologies*, Monterey, California, USA. USENIX Association.

[Martin et al., 1989] MARTIN, B., BERGAN, C., BURKHARD, W., et PARIS, J. (1989). « Exprerience with PARPC ». Dans *Proceeding of the 1989 USENIX Winter Conference*, pages 1–12. USENIX Association.

[Minton, 1997] MINTON, G. (1997). « IIOP Specification: A Closer Look ». *Unix Review*, pages 41–50.

[Mishra et al., 1989] MISHRA, S., PETERSON, L., et SCHLICHTING, R. (1989). « Implementing Fault-Tolerant Replicated Objects Using Psync ». Dans *Proceedings of the 8th Symposium on Reliable Distributed Systems*, pages 42–52, Seattle, Washington, USA. IEEE Computer Society.

[Morin et Puaut, 1995] MORIN, C. et PUAUT, I. (1995). « A Survey of Recoverable Distributed Shared Memory Systems ». Rapport Technique PI-975, IRISA, Rennes, France.

[Mullender, 1993] MULLENDER, S., éditeur (1993). *Distributed Systems*. Addison–Wesley / ACM Press, second edition.

[Narasimhan et al., 1997a] NARASIMHAN, P., MOSER, L. E., et MELLIAR-SMITH, P. M. (1997a). « Exploiting the Internet Inter-ORB Protocol Interface to Provide CORBA with Fault Tolerance ». Dans [COOTS, 1997].

[Narasimhan et al., 1997b] NARASIMHAN, P., MOSER, L. E., et MELLIAR-SMITH, P. M. (1997b). « The Interception Approach to Reliable Distributed CORBA Objects ». Dans [COOTS, 1997].

[Nelson, 1981] Nelson, B. (1981). « *Remote Procedure Call* ». PhD thesis, Carnegie Mellon University, Department of Computer Science, Pittsburgh, PA, USA. Available as Technical Report CMU-CS-81-119.

[OMG, 1996] OMG (1996). « *The Common Object Request Broker: Architecture and Specification* ». Object Management Group, Framingham, MA, USA, 2.0 ed. edition.

[OMG, 1997] OMG (1997). « *CORBAservices: Common Object Services Specifications* ». Object Management Group, Framingham, MA, USA.

[ORBIX, 1995] ORBIX (1995). « *The Orbix Architecture* ». IONA Techonogies Ltd., Dublin, Ireland.

[Ortega, 1993] Ortega, M. (1993). « *La mémoire virtuelle partagée répartie du système Chorus* ». PhD thesis, Université Pierre et Marie Curie, IBP, Paris, France.

[Partridge et Hinden, 1990] Partridge, C. et Hinden, R. (1990). « Version 2 of the Reliable Data Protocol (RDP) ». Request For Comments 1151.

[Petersen et al., 1996] Petersen, K., Spreitzer, M., Terry, D., et Theimer, M. (1996). « Bayou: Replicated Database Services for World-wide Applications ». Dans *Proceeding of the 7th SIGOPS European Workshop*, Connemara, Ireland. ACM.

[Petersen et al., 1997] Petersen, K., Spreitzer, M., Terry, D., Theimer, M., et Demers, A. (1997). « Flexible Update Propagation for Weakly Consistent Replication ». Dans *Proceeding of the 16th Symposium on Operating Systems Principles*, pages 288–301, Saint Malo, France.

[Peterson, 1987] Peterson, L. (1987). « Preserving Context Information in an IPC Abstraction ». Dans *Proceedings of the 6th Symposium on Reliable Distributed Systems*, Williamsburg, USA. IEEE Computer Society.

[Postel, 1980] Postel, J. (1980). « User Datagram Protocol ». Request For Comments 768.

[Raynal et Schiper, 1995] Raynal, M. et Schiper, A. (1995). « From Causal Consistency to Sequential Consistency in Shared Memory Systems ». Dans Thiagarajan, P., éditeur, *Proceedings of the 15th International Conference on Foundations of Software Technology and Theorical Computer Science*, numéro 1026 dans Lecture Notes in Computer Sciences, pages 180–194, Bangalore, India. Springer-Verlag.

[Rifflet, 1993] Rifflet, J.-M. (1993). *La Programmation sous Unix*. Ediscience.

[Rifflet, 1995] Rifflet, J.-M. (1995). *La Communication sous Unix*. Ediscience.

[Roy, 1969] Roy, B. (1969). *Algébre moderne et théorie des graphes*, volume 1. Dunod, Paris.

[Rozier et al., 1988] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Leonard, P., et Neuhauser, W. (1988). « Chorus Distributed Operating Systems ». *USENIX Computing Systems*, 1(4):305–370.

[Saltzer et al., 1984] Saltzer, J., Reed, D., et Clark, D. (1984). « End-to-End Arguments in System Design ». *ACM Transaction on Computer Systems*, 2(4):277–288.

[Satyanarayanan et Siegel, 1990] Satyanarayanan, M. et Siegel, E. (1990). « Parallel Communication in a Large Distributed Environment ». *IEEE Transactions on Computers*, 39(3):328–348.

[Schneider, 1990] Schneider, F. (1990). « Implementing Fault-Tolerant services using the State-Machine Approach: A tutorial ». *ACM Computing Surveys*, 22(4):299–319.

[Schneider, 1993] Schneider, F. (1993). « *Replication Management using the State-Machine Approach* », Chapitre 7, pages 169–197. Dans [Mullender, 1993], second edition.

[Shapiro, 1986] SHAPIRO, M. (1986). « Structure and encapsulation in distributed systems: The proxy principle ». Dans *Proceeding of the 6th International Conference on Distributed Computing Systems*, pages 198–204. IEEE Computer Society.

[Shrivastava, 1994] SHRIVASTAVA, S. (1994). « To CATOCS or not to CATOCS, that is the ... ». *Operating Systems Review*, 28(4):11–14.

[Sousa et Oliveira, 1997] SOUSA, J. et OLIVEIRA, R. (1997). « A CORBA Object Replication Service ». Accepted for the ECOOP'97 Workshop on Corba, Aussi à l'URL : `http://gsd.di.uminho.pt/~als/ecoop97.ps`.

[Sun Microsystems, 1987] SUN MICROSYSTEMS, I. (1987). « XDR: External Data Representation Standard ». Request For Comments 1014.

[Tanenbaum, 1995] TANENBAUM, A. (1995). *Distributed Operating Systems*. Prentice Hall.

[Terry et al., 1994] TERRY, D., DEMERS, A., PETERSEN, K., SPREITZER, M., THEIMER, M., et WELCH, B. (1994). « Session Guarantees for Weakly Consistent Replicated Data ». Dans *Prodeedings of the third International Conference on Parallel and Distributed Information Systems*, pages 140–149, Austin, Texas, USA. IEEE Computer Society.

[Thia-Kime, 1997] THIA-KIME, G. (1997). « *Critères de cohérence pour données partagées à support réparti* ». PhD thesis, IRISA, Rennes, France.

[Thomas, 1979] THOMAS, R. (1979). « A Majority Consensus Approach to Concurrency Control for Multiple Copies Databases ». *ACM Transaction on Database Systems*, 4(2):180–209.

[van Renesse, 1994] van RENESSE, R. (1994). « Why Bother with CATOCS? ». *Operating Systems Review*, 28(1):22–27.

[van Renesse et al., 1995] van RENESSE, R., BIRMAN, K., GLADE, B., GUO, K., HAYDEN, M., HICKEY, T., MALKI, D., VAYSBURD, A., et VOGELS, W. (1995). « Horus : A Flexible Group Communications System ». Rapport Technique, Cornell University, Department of Computer Science Research, Ithaca, New York, USA. TR95-1500.

[van Renesse et Tanenbaum, 1988] van RENESSE, R. et TANENBAUM, A. (1988). « Voting with Ghosts ». Dans [ICDCS, 1988], pages 456–462.

[van Steen et al., 1997] van STEEN, M., HOMBURG, P., et TANENBAUM, A. (1997). « The Architectural Design of Globe: A Wide-Area Distributed System ». Rapport Technique IR-422, Vrije Universiteit, Department of Math and Computer Science, Amsterdam, The Netherlands.

[van Steen et al., 1995] van STEEN, M., HOMBURG, P., van DOORN, L., TANANBAUM, A., et de JONGE, W. (1995). « Towards Object-based Wide Area Distributed Systems ». Dans *1995 Proceeding of the International Workshop on Object-Orientation in Operating Systems*, pages 224–227, Lund, Sweden. IEEE Computer Society.

[Vinoski, 1997] VINOSKI, S. (1997). « CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments ». *IEEE Communications Magazine*, 14(2).

[Wortman et al., 1994] WORTMAN, D., ZHOU, S., et FINK, S. (1994). « Automating Data Conversion for Heterogeneous Distributed Shared Memory ». *Software: Practice & Experience*, 24(1):111–125.

[Zhou et al., 1992] ZHOU, S., STUMM, M., LI, K., et WORTMAN, D. (1992). « Heterogeneous Distributed Shared Memory ». *IEEE Transactions on Parallel and Distributed Systems*, 3(5):540–554.

Résumé : L'amélioration des performances d'un service distribué vis-à-vis de ses utilisateurs est un critère important de satisfaction. Or, les canaux de communication introduisent des temps de propagation et de latence qui pénalisent le client. L'utilisation de portables connectés par intermittence ou avec des débits faibles et irréguliers ajoute des contraintes supplémentaires. Notre travail a comme objectif principal d'améliorer les performances des clients dans un système distribué à objets. Notre contribution principale est d'une part un cadre de travail (architecture partielle d'un système à objets utilisant la réplication active) et d'autre part un protocole d'estampilles d'ordonnancement. Tout d'abord, le protocole des estampilles d'ordonnancement permet à une requête de transporter avec elle toutes les informations nécessaires à sa livraison dans un ordre imposé par un modèle de cohérence. L'implantation de ce modèle se résume alors à utiliser ce protocole pour déterminer un ordre sur les requêtes en les identifiant et les estampillant. Cela dissimule aux concepteurs de serveurs les problèmes de propagation, de stockage, de remise en ordre et de livraison des requêtes à l'application. Nous espérons ainsi augmenter l'utilisation d'un modèle de cohérence strictement nécessaire, voire d'un modèle spécifique, à l'application en proposant une solution simple et efficace. Ensuite, dans notre cadre de travail, chaque objet gestionnaire fournit, indépendamment des autres, des fonctionnalités précises permettant à chacun de tirer partie de son environnement, et d'être facilement interchangeable. Entre ces différents objets, les communications sont les plus asynchrones possibles et les interactions sont au maximum locales. Enfin, l'implantation d'un prototype montre le coût limité du protocole des estampilles d'ordonnancement, ainsi que sa facilité d'utilisation pour implanter des modèles de cohérence et des mécanismes de synchronisation.

Mot clés : systèmes distribués, systèmes à objets, performance, cohérence, réplication active, synchronisation.

Abtract: Speeding up distributed application is a significant quality criteria for users. Communication channels have associated propagation time as well as latency which slow clients down. The use of wireless appliances connected on a discontinuous basis or with a low and/or irregular bandwidth add even more constraints to the system. Our main goal is to improve server performance in an distributed objects' system. Our main contributions are (i) a framework, a partial architecture design of an objects' system using active replication, and (ii) an invocation scheduling stamp protocol. Firstly, the invocation scheduling stamp protocol allows a query to carry all the necessary information required to respect a delivery order fixed by a coherency model. The implementation of this model is simply done using our protocol to establish an order between queries by identifying and stamping them. This method hides the query propagation, spooling, reordering and delivery problems to application. Thus, we hope to increase the use of a strictly necessary coherency model, or even the use of application specific model, by suggesting a simple and efficient way. Secondly, in our framework, each manager provides a focused functionality, independently of other managers, allowing each one to take advantage of its environment and to be easily interchangeable. Asynchronous communications between those different managers are favoured as well as local interactions. Finally, we have built a prototype of the framework. Its shows the low cost of the invocation scheduling stamp protocol and how easy it is to implement coherency models and synchronization mechanisms with it.

Keywords: distributed systems, objects' systems, performance, coherency, active replication, synchronization.