Estampilles d'Ordonnancement : Principes & Utilisations

Jean-Philippe LESOT
lesot@src.lip6.fr
Université Pierre et Marie CURIE
Laboratoire d'Informatique de Paris 6
Thème Systèmes Répartis et Coopératifs
4, place Jussieu
F–75252 PARIS CEDEX 05

Jean-Marie RIFFLET
rifflet@src.lip6.fr
Université Denis DIDEROT
2, place Jussieu
F-75251 PARIS CEDEX 05

Résumé

Dans les systèmes distribués, la technique de réplication d'objets apporte deux bénéfices principaux : la sûreté de fonctionnement (disponibilité et fiabilité des données en multipliant les copies sur des sites différents) et une amélioration des performances des applications (en rapprochant les copies des applications qui les utilisent). Dans ces systèmes nous nous intéressons au maintien de la cohérence de manière efficace. Ce document introduit le principe d'un mécanisme générique appelé *estampilles d'ordonnancement* et montre leur utilisation pour supporter simultanément, de manière simple et efficace, des cohérences et synchronisations diverses dans un système distribué à objet avec des invocations asynchrones.

Mots clés : système distribué, système à objet, réplication, performance, cohérence, synchronisation.

Abstract

In distributed systems, the objects replication technique provide two main advantages: dependability (data availability and reliability by increasing the number of object copies on different hosts) and applications performance improvement (by moving object copies closer to applications that use it). In these systems, we are trying to keep the objects coherence in an efficient way. This paper introduces the principle of a generic mechanism named *Invocation Scheduling Stamps*. We show its use while providing, in a simple and efficient way, various coherence and synchronisation mechanisms simultaneously in a distributed objects system with asynchronous invocations.

Keywords: distributed system, distributed objects, replication, performance, coherence, synchronisation.

1 Introduction

Dans ce document, nous nous intéressons à la réplication des applications dans un système distribué afin d'améliorer les performances des applications. En effet, la dissémination des applications provoque un pourcentage élevé d'invocations distantes très pénalisantes pour les performances. Une application souvent exécutée surcharge son serveur ainsi que le site qui l'héberge, et diminue le parallélisme par un effet de sérialisation des requêtes. Les trois supports suivants sont un complément indispensable à la réplication :

- La *tolérance aux fautes* : Avec la distribution, la probabilité pour qu'une activité soit bloquée par la faute d'une composante du système est beaucoup plus forte que dans un système centralisé. Il faut donc garantir à l'utilisateur une grande disponibilité des applications et une fiabilité des informations ;
- Le *passage à l'échelle* : Pour bénéficier des clients ou serveurs disséminés géographiquement et de la puissance disponible sur les réseaux, les applications peuvent être réparties dans un environnement mondial. Ce passage à une grande échelle doit se faire avec des pertes de performance raisonnables ;
- L'*hétérogénéité* : Pour profiter au maximum des machines présentes sur un réseau, le support de l'hétérogénéité permet de répliquer les objets sur des machines d'architecture et/ou de système d'exploitation différents.

Le procédé de réplication adopté s'appuie sur la propagation des invocations des méthodes dans un système à objet plutôt que sur de la mémoire virtuellement partagée. Ainsi l'hétérogénéité est plus simple à mettre en œuvre. La connaissance de la sémantique des opérations est conservée sur la totalité des copies de l'objet, ce qui permet d'augmenter la concurrence [Bernstein et al., 1978] [Spector et Schawartz, 1983] et donc les performances.

La réplication permet d'augmenter la proportion d'accès locaux et de migrer les objets en créant et détruisant des copies. Ainsi, en regroupant les objets fortement couplés sur un même site, ou sur un même réseau local, nous diminuons les communications inter-machines ou inter-réseaux. La réplication permet aussi d'augmenter la sûreté de fonctionnement (aussi bien la disponibilité que la fiabilité). Il se pose alors le problème du maintien de la cohérence des copies qui ne doit pas annuler les bénéfices de la réplication en terme de performance.

Nous avons étudié ce problème dans le domaine de la gestion des fichiers [Chevochot et al., 1995], puis nous l'avons étendu à celui des objets. Dans ce document, nous discutons des *estampilles d'ordonnancement* qui sont le résultat de ce travail. Il s'agit d'un mécanisme générique, à faible coût, fournissant aux objets les moyens de maintenir simultanément des cohérences variées et des synchronisations. Par exemple, elles permettent de mettre en œuvre les cohérences séquentielle [Lamport, 1979], causale [Ahamad et al., 1991], PRAM [Lipton et Sandberg, 1988], de session [Terry et al., 1994], légère (*weak*) [Dubois et al., 1986], de relâchement (*release consistency*) [Gharachorloo et al., 1990], d'entrée (*entry consistency*) [Bershad et Zekauskas, 1991], ainsi que les exclusions mutuelles, sémaphores et barrières de synchronisation. Elles permettent également aux applications de construire des cohérences spécifiques.

Dans ce document, nous donnons quelques définitions générales sur la programmation orientée objet, puis nous exposons les objectifs et le cadre de notre travail. Nous expliquons le principe de base des estampilles d'ordonnancement, et nous examinons les optimisations possibles de ce mécanisme. Puis, nous récapitulons l'intégralité des principes en un seul et unique, et nous en examinons les propriétés de vivacité et de sûreté. Enfin, nous examinons brièvement quelques travaux en relation avec les nôtres.

2 Définitions et terminologies

2.1 Objets

La programmation orientée objet, utilise le concept d'abstraction de données (les *classes*). Cette abstraction associe un ensemble d'opérations (les *méthodes*) à une structure de données. Les méthodes caractérisent le comportement de la classe, alors que la structure de données représente la classe. Un *objet* est une instance de cette classe. Les objets ne peuvent être accédés que par l'intermédiaire de leurs méthodes. Une invocation d'une méthode représente son exécution. 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.

Généralement, l'*état* d'un objet représente l'ensemble des variables définies par sa structure de données. 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.

2.2 Lecture et écriture

Nous devons maintenant définir les actions qui modifient ou non l'état d'un objet.

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

Notation 1
Nous notons une invocation en lecture par L et en écriture par E. Une invocation (en lecture ou en écriture) est notée I.

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

3 Objectifs et cadre

Nous exposons dans ce paragraphe nos objectifs en indiquant les moyens que nous retenons pour les remplir :

- *Disponibilité* : l'invocation de n'importe quelle copie d'un serveur par le client permet d'atteindre cet objectif. Ainsi, même au cours de sa vie, le client peut changer de copie, impliquant une disponibilité accrue. Le système Grapevine [Birrell et al., 1982] fut le premier à offrir cette méthode appelée *read-any/write-any* ;

- *Fiabilité* : l'invocation en écriture de façon k-synchrone (c'est-à-dire que l'invocation est effective quand k copies d'un serveur l'ont réellement exécutée) permet de fiabiliser l'invocation en cas de panne d'au plus $k-1$ copies du serveur avec des hypothèses de défaillance par arrêt ;
- *Performance* : de nombreuses méthodes influencent les performances :
  - Le changement de copie joue un rôle non négligeable, comme pour la disponibilité des données, puisqu'un client peut choisir une autre copie d'un serveur, s'il trouve la sienne trop lente ;
  - L'invocation en lecture de plusieurs copies d'un serveur simultanément pour obtenir la réponse de la plus rapide est un autre moyen, mais pénalise les autres clients ;
  - Le choix du nombre et de la provenance des réponses à une invocation permet à un client de ne pas attendre la réponse d'une copie d'un serveur trop lent, mais seulement le strict minimum en fonction de sa politique de fiabilité ;
  - Fournir la cohérence exacte, et non plus forte, que l'application requière permet d'éviter des dépendances inutiles entre les copies et de diminuer le coût réseau ;
  - Fournir le même mécanisme pour le maintien de la cohérence et pour assurer les synchronisations permet d'optimiser les messages réseau, les mécanismes de synchronisation étant intimement liés à ceux de la cohérence.
- *Passage à l'échelle* : nous décomposons cet objectif en deux sous objectifs distincts concernant d'une part les algorithmes, et d'autres part les réseaux :
  - Cet objectif n'impose pas de choix en particulier, mais doit influencer tous nos algorithmes, solutions, et implantations. Par exemple, nous faisons rarement des hypothèses sur le nombre de copies d'un serveur, et nous fournissons, quand cela est possible, des mécanismes supportant cette contrainte ;
  - Dans les multi-réseaux, la topologie et les caractéristiques des réseaux sont trop variées pour qu'il soit facile d'en tirer parti sans se restreindre à un sous-ensemble. Nous ne faisons donc aucune hypothèse sur les caractéristiques des réseaux, sauf en ce qui concerne leur fiabilité. C'est-à-dire que la seule supposition que nous faisons sur le médium de communication est que les invocations propagées arriveront au destinataire. Nous pouvons donc utiliser des réseaux non FIFO (sans garantie sur l'ordre de livraison des messages), avec déconnections fréquentes, supportant la diffusion, etc.
- *Hétérogénéité* : le support de l'hétérogénéité est simplement fourni par les invocations de méthodes d'un objet. Cela permet à moindre coût — celui de l'encodage/décodage (*marshaling*/*unmarshaling*) — de passer d'une architecture ou d'un système à un autre. De plus, ce mécanisme est fort souvent implanté dans la plate-forme objet sous-jacente.

Enfin, dans ce document, nous restreignons le schéma de relation entre les clients et les serveurs. Deux clients répliqués invoquent chacun des copies différentes d'un serveur. C'est-à-dire que l'ensemble des copies d'un serveur qu'invoque n'importe quel client a une intersection vide avec l'ensemble des copies du serveur invoquées par les autres répliques de ce client. En effet, dans le cas contraire, nous devons prendre en compte la duplication des invocations, et donc mettre en place des filtrages [Mazouni et al., 1995]. De tels mécanismes sortent du cadre de ce document.

4 Estampilles d'ordonnancement

4.1 Relation d'ordre strict

Généralement, une manière de voir 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 exécutée 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 :

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

Il est clair que la relation $\rightharpoonup$ est irréflexive, transitive et asymétrique (donc anti-symétrique puisqu'irréflexive). 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. Ainsi, 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 relativement dans n'importe quel ordre.

4.2 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 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).

4.3 Estampilles d'ordonnancement

4.3.1 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.

Notation 2
Nous noterons 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 différencions 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 « dépendant de » l'invocation I_e. $I'_{i'}$ doit donc être délivrée à l'objet après que I_e ait été complètement exécutée par l'instance de l'objet. Plus formellement, nous posons le premier principe spécifiant l'ordonnancement des invocations sur un objet. Par la suite, il sera complété ou remplacé par d'autres principes.

Principe 1 (Estampille simple)
Quand une invocation I estampillée par e arrive sur un objet o :

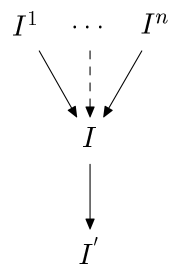
- Attendre que I'_e ai été exécutée ;
- Délivrer I à o.

4.3.2 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 1 page suivante.

4.4 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 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.



(a) Graphe

– $I^1 \cdots\langle i\rangle\cdots$, ..., $I^n \cdots\langle i\rangle\cdots$;

– $I_i\langle j\rangle$;

– $I'_j\cdots$.

(b) Estampille

Fig. 1 – Graphe de dépendance et estampilles

4.4.1 Principe

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

Notation 3
Nous notons une invocation quelconque d'identificateur i et comportant une liste d'estampilles de e_1 à e_n par $I_i\langle e_1\rangle\langle e_2\rangle\cdots\langle e_n\rangle$.

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 :

Principe 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 $1 \leq j \leq n$) ;

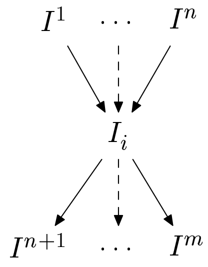
– Délivrer I à o.

4.4.2 Représentation du graphe

La figure 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 $\rightharpoonup$.

4.5 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 2, 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''



(a) Graphe

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

(b) Estampille

Fig. 2 – Graphe de dépendance et listes d'estampilles

l'aura été, et cela sans faire intervenir $I \rightharpoonup I''$ dû 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 en 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 graphes appartenant à 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 page suivante, 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 performances, les estampilles d'ordonnancement générées doivent représenter ce graphe de dépendance direct.

4.6 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

(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 – Exemple de graphe de dépendance

serveur. Ces explications sont des exemples, d'autres moyens d'utiliser ces estampilles sont possibles, y compris pour garantir une cohérence similaire.

4.6.1 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 identificateur d'invocation) notée tout simplement i. La figure 4 page suivante fournit les différentes représentations graphiques des messages.

4.6.2 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.

Pour invoquer une méthode sur une instance d'un objet, un client invoque la méthode sur une des copies du serveur. 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,

$\xrightarrow{I_i\langle e_1\rangle\cdots\langle e_n\rangle}$	Invocation $I_i\langle e_1\rangle\cdots\langle e_n\rangle$ délivrée au gestionnaire d'estampilles,
$\xrightarrow{I_i\langle e_1\rangle\cdots\langle e_n\rangle}$	Invocation $I_i\langle e_1\rangle\cdots\langle e_n\rangle$ délivrée à l'application,
$\xrightarrow{I_i\langle e_1\rangle\cdots\langle e_n\rangle}$	Retard de l'invocation $I_i\langle e_1\rangle\cdots\langle e_n\rangle$ pour ordonnancement,
$\xrightarrow{I_i\langle e_1\rangle\cdots\langle e_n\rangle}$	Invocation $I_i\langle e_1\rangle\cdots\langle e_n\rangle$ 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. 4 – Convention graphique utilisée

- 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, partiellement ou totalement, aux autres copies les invocations. Il fournit donc, suivant sa politique, une propagation à son initiative ou sur demande, immédiate ou périodique, etc.

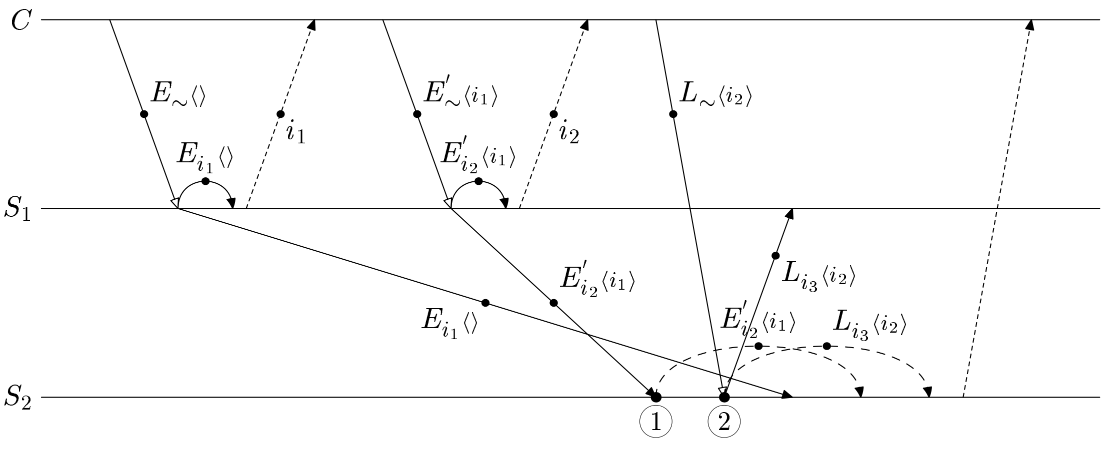
Enfin, un gestionnaire d'ordonnancement se charge de délivrer les invocations de la queue d'exécution à l'application. C'est donc lui qui respecte les différents principes relatifs aux estampilles.

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, dans le but de simplifier les exemples. Nous verrons au § 5.1 page 11 les optimisations possibles pour éviter cette diffusion.

4.6.3 Maintien de l'ordre programme

Nous considérons le cas d'un seul client qui attend que l'ordre programme de ces invocations soit respecté sur un même serveur répliqué. 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 5 page suivante.



Une écriture E parvient à S_1, qui lui assigne un identificateur i_1, la diffuse à l'autre copie S_2, l'exécute, et renvoie le résultat avec l'indication d'estampiller la prochaine invocation sur S avec i_1. Il en est de même pour l'écriture suivante E'. Pour l'invocation en lecture L, C s'adresse à S_2. Pour conserver l'ordre local d'exécution, 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. 5 – Ordre programme avec les estampilles

4.6.4 Maintien de la cohérence séquentielle objet

Nous présentons brièvement une méthode de maintien de la cohérence séquentielle. 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. 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.

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 séquentiel, 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 6 page suivante montre un exemple de scénario possible avec un tel algorithme.

5 Optimisation

La première optimisation possible, nous l'avons faite au § 4.5 page 6, en introduisant le graphe de dépendance direct. Nous décrivons dans cette section d'autres optimisations essentielles pour que les estampilles d'ordonnancement soient efficaces.

5.1 Diffusion des lectures

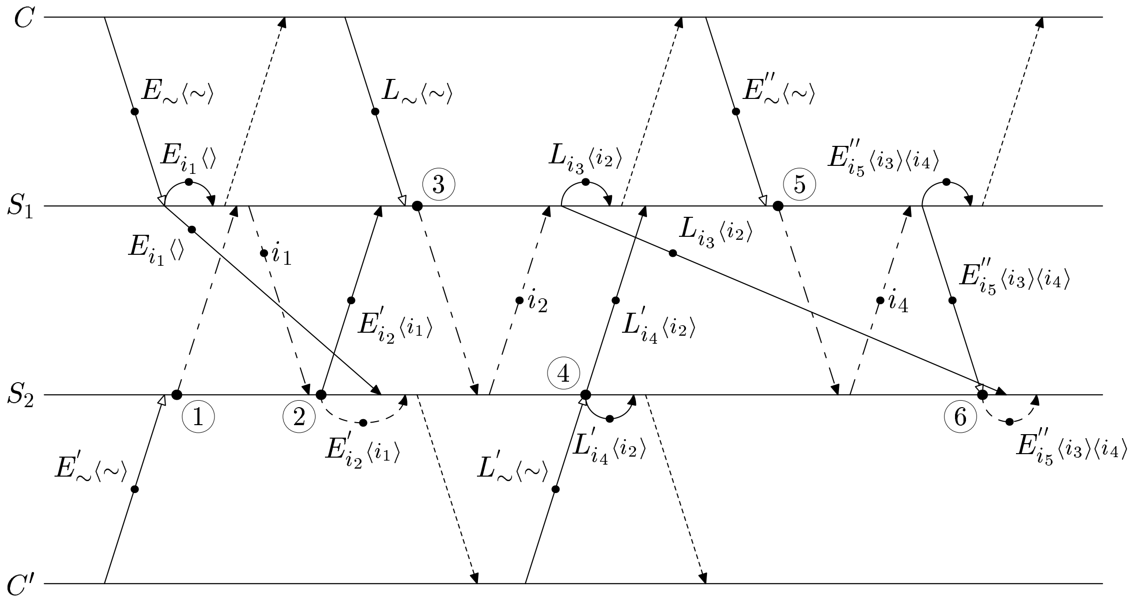
5.1.1 Problème

Des définitions du paragraphe § 2.2 page 2, 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.

Seulement la diffusion des invocations en lecture peut permettre d'améliorer les performances ou la tolérance aux fautes. 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



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 estampille $E^{''}$ avec ;
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. 6 – Cohérence séquentielle objet avec les listes d'estampilles

d'un serveur répliqué. Néanmoins, même si nous voulons de la performance ou de la tolérance aux fautes, 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 censé 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 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 7 page suivante.

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 l'*estampille* de L pour les autres copies.

Regardons l'exemple de la figure 8 page 15. 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 délais 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.

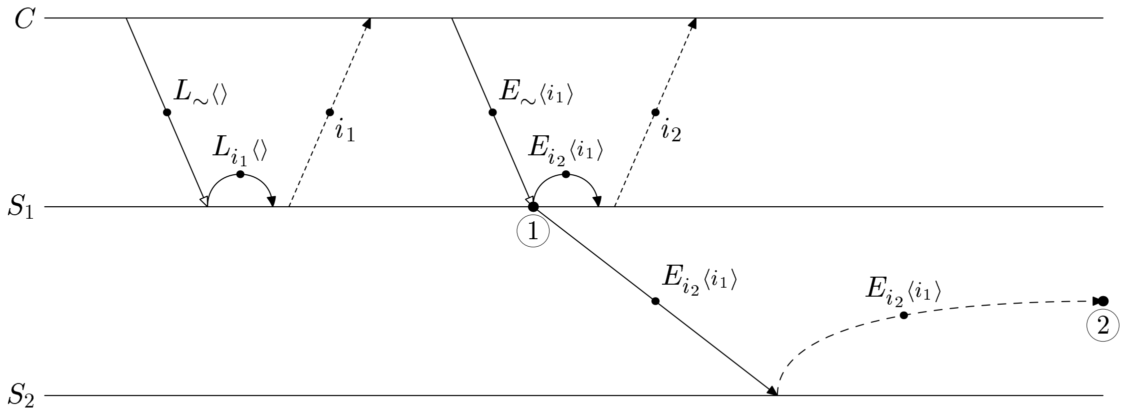
5.1.2 Première approche

Au lieu de transmettre deux fois une invocation en écriture référençant une invocation en lecture, avec deux estampilles différentes, la solution adoptée est tout simplement de la transmettre une seule fois, avec les deux estampilles possibles. Cela est à la charge du gestionnaire d'ordonnancement d'utiliser la bonne. Nous devons donc modifier le principe des estampilles, pour qu'une estampille référençant une invocation en lecture soit complétée par d'autres estampilles, appelées estampilles secondaires, de même valeur que les estampilles de L. Ainsi, lorsque $I \rightharpoonup L_{i_0}\langle i_1\rangle\cdots\langle i_n\rangle$, nous avons $I\langle i_0, i_1, \ldots, i_n\rangle$.

Ce principe est malheureusement récursif, c'est-à-dire que si l'invocation référencée, par exemple par i_1, est $L_{i_1}\langle i_1^1\rangle\cdots\langle i_n^1\rangle$, nous devons remplacer i_1 dans les estampilles secondaires de I par les estampilles de L_{i_1}. Il est donc assez inimaginable de fournir un mécanisme simple et efficace avec un nombre d'estampilles secondaires qui peut très vite devenir conséquent.

5.1.3 Graphe de dépendance simplifié

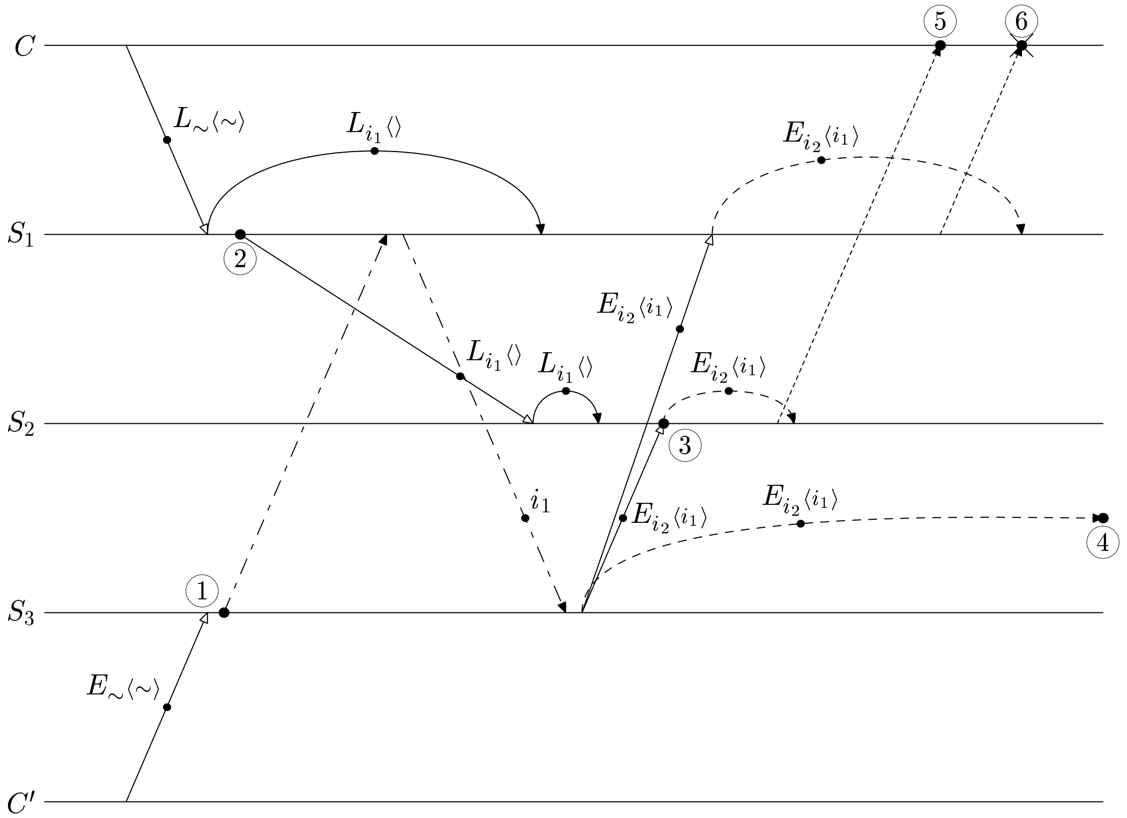
Si nous posons l'hypothèse qu'une invocation I qui ne dépend d'aucune autre invocation, dépend de l'invocation vide $\emptyset$, une invocation en lecture L a donc comme schéma de dépendance un des trois cas suivants :



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



Dans cet exemple, nous voulons fournir une cohérence séquentielle 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. 8 – Estampilles référençant une lecture (problème 2/2)

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

Or, par la définition 1 page 2, 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 :

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

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 simplifié, où une invocation en lecture est soit indépendante, soit dépendante d'une invocation en écriture. Elle supprime un arc, et en ajoute un qui est dû à la transitivité de la relation d'ordre strict $\rightharpoonup$. Il est donc clair que le graphe de dépendance simplifié est un graphe partiel du graphe de dépendance. Par conséquent, il est sans circuit et conserve les propriétés d'anti-symétrie et d'irréflexivité.

Par la suite, nous travaillerons exclusivement sur ce graphe de dépendance simplifié. Ce graphe résout donc le problème dû à la récursivité des estampilles secondaires engendrée par les dépendances $L \rightharpoonup L^{'}$.

5.1.4 Estampilles secondaires

Nous venons de voir qu'une invocation en lecture ne pouvait qu'être référencée par une estampille associée à une invocation en écriture. Les estampilles secondaires concernent donc exclusivement les invocations en écriture.

Définition 2 (Estampilles secondaires)
Une invocation en écriture possédant une estampille d'ordonnancement référençant une invocation en lecture L, possède des autres estampilles, appelées estampilles secondaires, de même valeurs que les estampilles de L.

Notation 4
Nous notons une invocation en écriture E d'identificateur i, ayant une estampille e, et des estampilles secondaires e_1 à e_n par $E_i\langle e,e_1,\ldots,e_n\rangle$.

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

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

De plus, pour qu'une invocation en lecture diffusée, qui arrive en retard par rapport à l'éventuelle écriture dont elle dépend, ne renvoie pas une valeur incohérente, nous devons ajouter le principe suivant :

Principe 3 (Identificateur 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.

Les principes 1 page 5 et 2 page 6 sont encore valides pour les invocations estampillées par un identificateur d'une invocation en écriture, mais il doit être complété par le principe suivant, que nous avons simplifié pour des raisons de compréhension. En effet, il est clair que la forme générale d'une invocation en écriture devient $E_{i\langle i_0^1, i_1^1, \ldots, i_n^1\rangle \cdots \langle i_0^p, i_1^p, \ldots, i_m^p\rangle}$. Toutefois, la généralisation du principe est trivial.

Principe 4 (Estampilles secondaires)
Quand une invocation en écriture E ayant comme estampille e et comme estampilles secondaires e_1 à e_n arrive sur un objet o :

- Attendre que $E^j_{e_j}$ aient été exécutées (avec $1 \leq j \leq n$) ;
- Attendre que L_e ne soit ni en cours d'exécution ni dans la queue locale ;
- Déclarer e invalide ;
- Délivrer E à o.

Il faut donc, pour garantir qu'aucune invocation en lecture ne soit perdue, que les gestionnaires d'estampilles respectent le principe suivant :

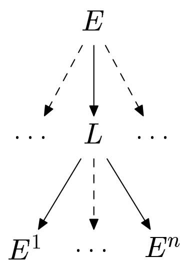
Principe 5
Le gestionnaire d'estampilles devra garantir qu'il n'utilise pas l'identificateur d'une invocation en lecture avant que cette dernière ne soit en attente d'exécution sur une des copies.

Ce principe garantit qu'aucune lecture ne soit perdue. Tout d'abord parce qu'aucune autre invocation ne pourra être estampillée par son identificateur avant qu'elle ne soit en attente d'exécution sur au moins un site. Ensuite, parce que d'après le principe 4, si une invocation en lecture est en attente d'exécution, 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. Il peut, à partir de ce moment, utiliser son identificateur comme estampille. Dans ce cas, ce principe n'introduit aucun délai supplémentaire dans l'ordonnancement des invocations.

5.1.5 Représentation du graphe

Nous devons maintenant spécialiser la figure 2 page 7 en faisant intervenir le type des invocations. Ainsi, la figure 9 page suivante décrit la représentation d'un motif du graphe de dépendance simplifié à l'aide des estampilles secondaires.



(a) Graphe

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

(b) Estampille

Fig. 9 – Graphe de dépendance et estampilles secondaires

5.1.6 Utilisation

Nous considérons sur cet exemple les différents scénarii qui peuvent se présenter lors de l'utilisation des estampilles secondaires. 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 10 page suivante.

5.2 Taille de la liste d'estampilles

5.2.1 Problème

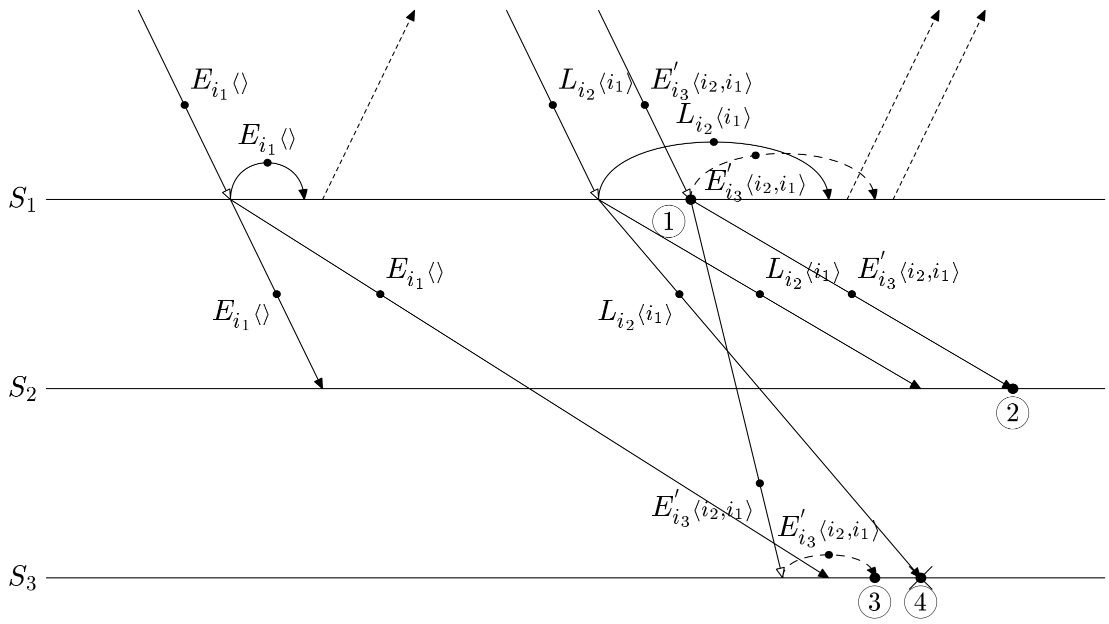
La liste d'estampilles présente sur des invocations pose le problème de sa taille. Cette liste étant non bornée, le gestionnaire d'estampilles peut théoriquement constituer une liste infinie, posant alors le problème du surcoût réseau dû à son transport, et du surcoût processeur dû à sa gestion.

Prenons l'exemple du scénario de la figure 11 page 20. Le gestionnaire d'estampilles doit fournir une cohérence séquentielle objet avec l'algorithme rapidement décrit au § 4.6.4 page 11. 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 séquentielle, pour les synchronisations, exclusions mutuelles, barrières, etc. Il est donc intéressant d'optimiser ce type de dépendance.

5.2.2 Groupe d'invocations

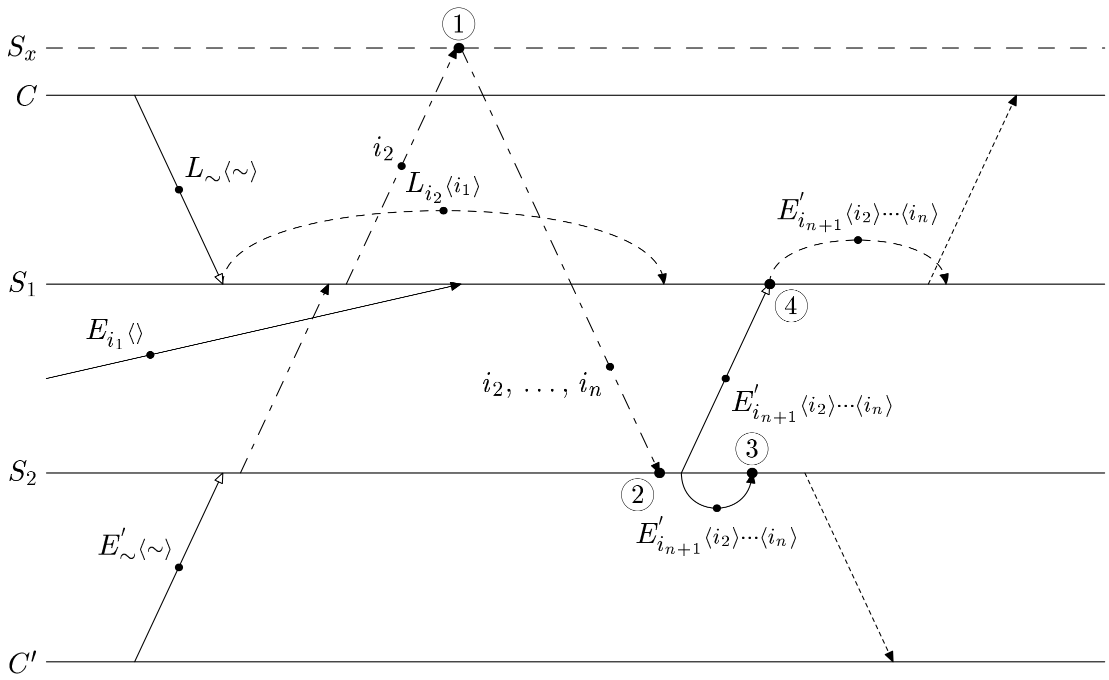
La solution retenue consiste à rassembler les invocations dans un groupe et non plus une liste, et à faire référencer ce groupe par *une seule* estampille. Pour cela, il suffit d'autoriser plusieurs invocations à posséder le même identificateur.



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}). Elle doit donc avoir une estampille secondaire égale à celle de L_{i_2}. Il s'agit ici de i_1. 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, n'ayant pas L_{i_2} ni en attente, ni en cours d'exécution, 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. 10 – Gestion des lectures avec les estampilles secondaires



Nous voulons ici maintenir une cohérence séquentielle 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 estampiller E' avec ;
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. 11 – Taille importante de la liste d'estampilles (problème)

Définition 3 (Groupe d'invocations)
Un groupe d'invocations est un ensemble non ordonné d'invocations ayant le même identificateur.

Définition 4 (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.

Définition 5 (Cardinal d'un groupe d'invocations)
Le cardinal d'un groupe d'invocations est le nombre d'invocations en écriture dans ce groupe.

Définition 6 (Cardinal d'une estampille)
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 1 page 5 sur les estampilles d'ordonnancement pour qu'il utilise le cardinal des groupes d'invocations :

Principe 6 (Groupe d'invocations et estampilles)
Quand une invocation I estampillée par e de cardinal n et sans estampille secondaire arrive sur un objet o :

- Attendre que les n invocations en écriture d'identificateur e aient été exécutées,
- Attendre qu'aucune invocation en lecture d'identificateur e ne soit ni en cours d'exécution ni dans la queue locale.
- Déclarer e invalide ;
- Délivrer I à o.

Nous devons aussi remplacer le principe 4 page 17 sur les estampilles secondaires pour utiliser les groupes d'invocations.

Principe 7 (Groupe d'invocations et estampilles secondaires)
Quand une invocation en écriture E estampillée par e de cardinal n et possédant des estampilles secondaires e_1 à e_n, respectivement de cardinal n_1 à $n_{n'}$, arrive sur un objet o :

- Attendre que les n_i invocations en écriture d'identificateur e_i aient été exécutées (avec $1 \leq i \leq n'$) ;
- Attendre qu'aucune invocation en lecture d'identificateur e_i ne soit ni en cours d'exécution ou ni la queue locale (avec $1 \leq i \leq n'$) ;
- Déclarer e_i invalide (avec $1 \leq i \leq n'$);

- Attendre qu'aucune invocation en lecture d'identificateur e ne soit ni en cours d'exécution ni dans la queue locale ;

- Déclarer e invalide ;

- Délivrer E à o.

5.2.3 Connaissance du cardinal

Nous avons donc besoin de connaître le cardinal d'un groupe pour appliquer les principes d'ordonnancement. Nous pouvons indiquer le cardinal à chaque fois que nous émettons l'estampille. C'est une solution simple, mais qui a comme inconvénient que le groupe doit être clos 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 8 (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.

- Quelque soit 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 les principes d'ordonnancement 6 et 7 page précédente ne ralentissent pas (et à fortiori ne bloquent 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 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 notée $i : n$.

5.2.4 Représentation du graphe

La représentation du graphe de dépendance simplifié par les groupes d'invocations se divise en trois cas, suivant que le groupe d'invocations soit 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 pour améliorer la lisibilité des figures.

La figure 12 concerne les groupes en écriture. Une invocation en écriture n'impliquant pas d'estampilles secondaires 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 secondaires des invocations les référant. 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 secondaires. Cette condition est représentée dans la figure 13 page suivante.

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 secondaires, puisque les invocations en écriture du groupe garantissent le respect de la dépendance sur les invocations suivantes. La figure 14 page suivante présente cette transformation.

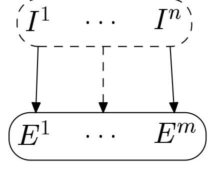
5.2.5 Groupe d'invocations *vs* liste d'estampilles

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 simplifié 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, 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

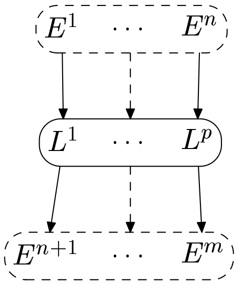


(a) Graphe

– $I^1 \cdots\langle i_0\rangle\cdots, \ldots, I^n \cdots\langle i_0\rangle\cdots$;
– $E^1_{i_0}\cdots, \ldots, E^m_{i_0}\cdots$;
– $i_0 : m$.

(b) Estampille

Fig. 12 – Graphe de dépendance et groupes d'invocations en écriture

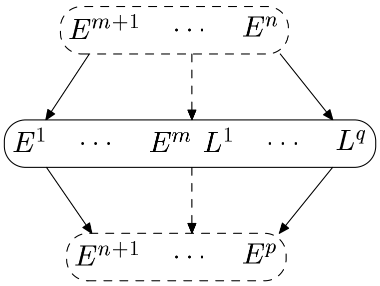


(a) Graphe

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

(b) Estampille

Fig. 13 – Graphe de dépendance et groupes d'invocations en lecture



(a) Graphe

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

(b) Estampille

Fig. 14 – Graphe de dépendance et groupes d'invocations mixtes

de la taille des listes. Mais aussi, nous introduisons un asynchronisme plus fort entre l'acquisition du « droit et ordonnancement d'exécution » 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 § 5.3.2.

5.3 Utilisations

5.3.1 Maintien de la cohérence séquentielle objet « inverse »

Nous allons reprendre l'exemple de la figure 6 page 12, 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 de statistiques pour savoir combien de fois une méthode est invoquée sur un objet. L'incrémentation doit donc être faite de 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.

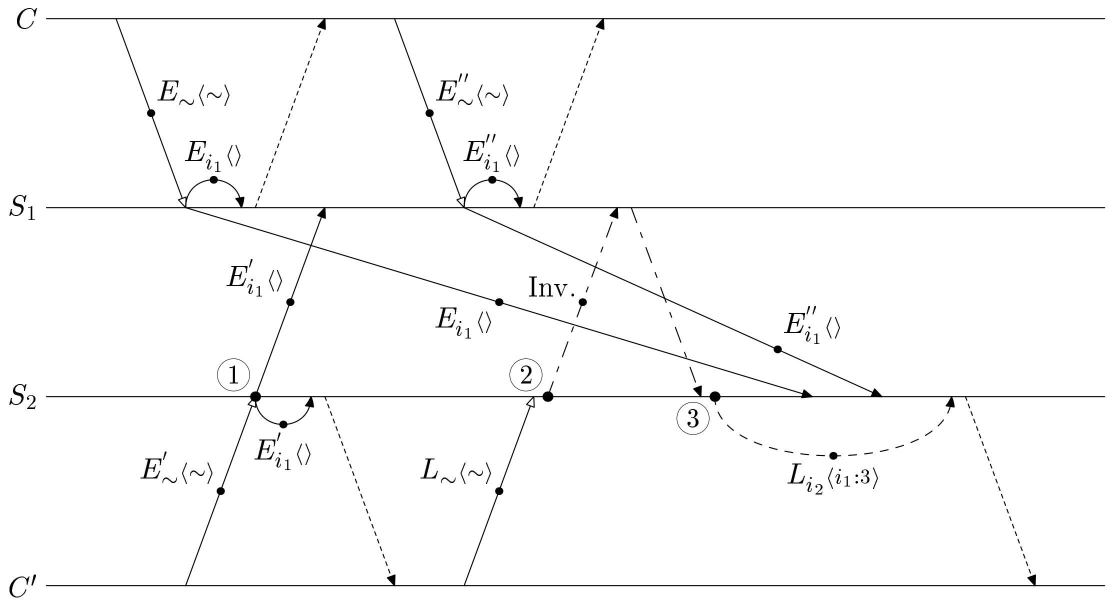
Pour satisfaire ces critères, nous utiliserons le même protocole qu'au § 4.6.4 page 11, mais en inversant le rôle des invocations en lecture avec celles en écriture. Nous montrons un exemple de scénario possible à la figure 15 page suivante.

5.3.2 Gestion d'exclusion mutuelle

Nous allons fournir un autre exemple d'application. D'abord, le cardinal du groupe d'invocations est communiqué par un identificateur (et non une estampille comme pour l'exemple précédent). Ensuite, il s'agit de montrer 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 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 16 page 28. Nous pouvons remarquer qu'un client concurrent peut acquérir le verrou avant qu'il ne soit rendu, et commencer à invoquer des méthodes avec. Bien sûr, elles ne seront exécutées qu'à la fin de la section critique



Initialement, les deux copies S_1 et S_2 connaissent l'estampille (vide) et l'identificateur du groupe. Lors des 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 les droits 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. 15 – Cohérence séquentielle objet « inverse » avec les groupes d'invocations

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.

6 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 (§ 4.3.1 page 5) et de leurs dérivés (listes d'estampilles (§ 4.4 page 5), estampilles secondaires (§ 5.1.4 page 16) et groupes d'invocations (§ 5.2.2 page 18)). Nous rappelons que ce mécanisme permet de représenter le graphe de dépendance simplifié (§ 5.1.3 page 13) dérivé du graphe de dépendance direct (§ 4.5 page 6), lui-même déduit du graphe de dépendance (§ 4.2 page 4). Ce dernier est la représentation de la relation d'ordre strict « dépendant de » (§ 4.1 page 4) qui caractérise la ou les cohérences et synchronisations voulues.

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.

Il ne distingue pas non plus les estampilles des estampilles secondaires, et considère tous les identificateurs d'invocations équivalents. En fait, cette simplification est possible grâce au cardinal des groupes. Par exemple, les estampilles secondaires sont des estampilles ordinaires, mais référençant obligatoirement un groupe en écriture ou mixte. Nous obtenons donc $n_i^j \geq 1$ (avec pour chaque $1 \leq j \leq p$: $1 \leq i \leq m_j$). Nous pouvons aussi observer que les estampilles secondaires existent seulement si l'estampille associée référence un groupe en lecture, et donc que son cardinal est égal à zéro. Par conséquent, si $n_0^j \geq 1$ alors e_i^j n'existent pas (avec $1 \leq i \leq m_j$).

En considérant le type de l'invocation, d'autres propriétés existent. Cela nous permet d'écrire le principe général régissant les estampilles d'ordonnancement d'une façon simplifiée :

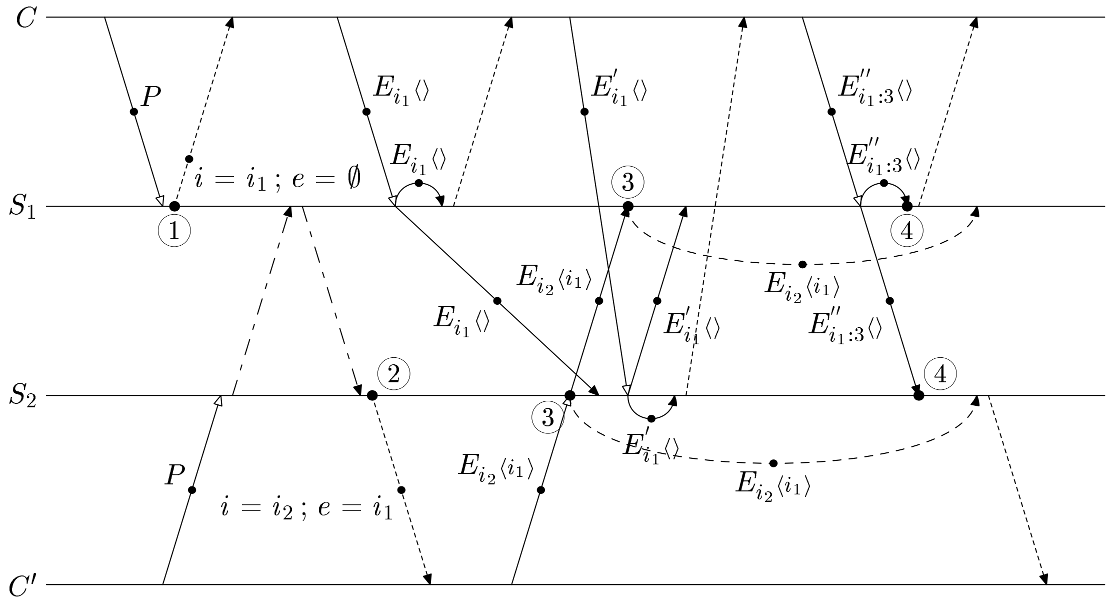
Principe 9 (Estampilles d'ordonnancement)
Quand une invocation $I_{e\langle e_0^1, e_1^1, \ldots, e_{m_1}^1\rangle \cdots \langle e_0^p, e_1^p, \ldots, e_{m_p}^p\rangle}$ (avec n_i^j cardinal du groupe d'identificateur e_i^j) arrive sur un objet o :

1. Attendre que les n_i^j invocations en écriture d'identificateur e_i^j aient été exécutées (avec pour chaque $1 \leq j \leq p$: $0 \leq i \leq m_j$) ;

2. Attendre qu'aucune invocation en lecture d'identificateur e_i^j ne soit ni en cours d'exécution ou ni la queue locale (avec pour chaque $1 \leq j \leq p$: $0 \leq i \leq m_j$) ;

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

4. Délivrer I à o .



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 3 invocations E_{i_1}, E'_{i_1} et E''_{i_1} l'ont déjà été.

Fig. 16 – Exclusion mutuelle avec les groupes d'invocations

6.1 Vivacité

La condition de vivacité (*liveness*) du principe des estampilles d'ordonnancement est qu'il n'y ai 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 ($I \rightharpoonup \cdots \rightharpoonup I$). Nous avons vu que les estampilles représentent le graphe de dépendance simplifié 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, une deuxième type de blocage peut provenir d'une invocation qui n'arrive jamais sur un site, bloquant ainsi les invocations estampillées avec son identificateur. Or, nous considérons que les communications sont fiables. Donc, toutes les invocations diffusées (c'est-à-dire tous les groupes d'invocations en écriture ou mixte ainsi que certains groupes d'invocations en lecture) sont reçues sur toutes les copies, libérant les invocations en attente de celles ci (point 1 du principe). Enfin, les invocations diffusées partiellement (c'est-à-dire les groupes d'invocations en lecture non diffusés) ne sont pas forcément reçues sur toutes les copies, mais ne sont pas bloquantes (point 2 du principe). Ces propriétés nous garantissent l'absence de ce type de blocage.

6.2 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'elle retourne 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 1 du principe, une invocation quelconque dépendante (entre autres) de x invocations en écriture ne pourra être exécutée qu'après ces dernières. Par récurrence immédiate, cette propriété satisfait la première condition de sûreté.

Par le principe 3 page 17, une invocation en lecture L est au moins dans une des queues d'exécution d'une des copies. Par le point 2 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 ce site, l'invocation en lecture L est exécutée 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 3 du principe, son identificateur est invalide, et L n'est jamais exécutée sur ce site. Ceci vérifie la deuxième condition de sûreté.

7 Travaux similaires

La *lazy replication* 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, dans le sens où 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 part l'utilisation des estampilles temporelles (*timestamps*) sous forme de vecteurs), et ne peut donc pas fournir de cohérence plus faible. Des cohérences plus fortes sont décrites [Ladin et al., 1992], mais elles sont implantées de façons relativement rigides. Toutes les solutions décrites dans ces documents sont transposables en utilisant les 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 a 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, dans le sens où il est utilisé dans un cadre plus large (celui des objets) et où il supporte un plus large éventail de cohérences et synchronisations.

Le projet Bayou [Petersen et al., 1996] consiste à spécifier et implanter un système de stockage distribué supportant les besoins des utilisateurs mobiles d'ordinateurs portables. La propagation des mise-à-jours entre deux copies du serveur de stockage est implantée par leur protocole d'« anti-entropie » [Petersen et al., 1997]. Il permet de faire converger les copies en propageant les mise-à-jours entre deux systèmes de stockage. Il supporte une variété de politiques (quand et où propager les modifications). Il s'adapte à des conditions réseau extrêmement variées et difficiles (faible bande passante, déconnections fréquentes, support transportable, etc). C'est donc une caractéristique qui peut être similaire aux nôtres, mais leur travail est complémentaire à nos estampilles d'ordonnancement, puisque nous ne faisons aucune hypothèse sur le mode de propagation des invocations.

Enfin, le système Globe [van Steen et al., 1997] est une plate forme basée sur des objets pour le développement d'applications à large échelle. Il fournit aux applications, entre autres, une transparence à la réplication en encapsulant dans un objet distribué le protocole (point-à-point, diffusion, etc) de communication (objet communication) et sa politique de réplication (objet de réplication). De ce que nous pouvons déduire de [Kermarrec et al., 1997], notre gestionnaire d'estampilles est un très bon candidat pour les différents objets de réplication de Globe. En effet, la philosophie d'encapsuler les différentes politiques de cohérence à l'intérieur des objets nous est commune. Néanmoins, notre système fournit une séparation supplémentaire entre la partie qui gère la politique de cohérence (gestionnaire d'estampilles) et la partie qui fournit le stockage, l'ordonnancement et la livraison des invocations (gestionnaire d'ordonnancement). Cette séparation permet d'écrire plus facilement et d'intégrer plus rapidement des politiques de cohérences et de synchronisations.

8 Conclusion

Dans ce document, nous avons présenté et optimisé pas à pas notre mécanisme d'estampilles d'ordonnancement permettant à des objets répliqués de supporter des cohérences et des synchronisations multiples de façon performante. Ce mécanisme est simple et assez générique pour s'intégrer dans divers systèmes à objets. Il ne requière aucune autre caractéristique de communication que la fiabilité, supportant ainsi des conditions difficiles (déconnections, variations de débit, etc). L'architecture a été conçue pour supporter le passage à l'échelle et exploiter au maximum l'asynchronisme possible dans les communications entre objets répliqués.

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 stockages, 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.

Références

[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.

[Berge, 1963] BERGE, C. (1963). *Théorie des graphes et ses applications.* Dunod, Paris, deuxième edition.

[Bernstein et al., 1978] BERNSTEIN, P., ROTHNIE, J. J., GOODMAN, N., et PAPADIMITRIOU, C. (1978). « The Concurrency Control Machanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case) ». *IEEE Transactions on Software Engineering*, 4(3):154–168.

[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.

[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.

[Chevochot et al., 1995] CHEVOCHOT, P., LESOT, J.-P., ROGER, F., et RIFFLET, J.-M. (1995). « Système de Gestion de Fichiers Distribué : migration et réplication dynamique comme approche à l'amélioration des performances ». Rapport de DEA, Université Pierre et Marie Curie, IBP, Laboratoire MASI, Paris, France. Aussi à l'URL : `http://gloups.home.ml.org/Dea/report-fr.html`.

[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.

[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.

[Kermarrec et al., 1997] KERMARREC, A.-M., KUZ, T., van STEEN, M., et TANENBAUM, A. (1997). « A Framework for Consistent, Replicated Web Objects ». Rapport Technique IR-431, Vrije Universiteit, Department of Math and Computer Science, Amsterdam, The Netherlands.

[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., 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.

[Ladin et al., 1990] Ladin, R., Liskow, 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.

[Lamport, 1979] Lamport, L. (1979). « How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs ». *IEEE Transactions on Computers*, 28(9):690–691.

[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.

[Mazouni et al., 1995] Mazouni, K., Garbinato, B., et Guerraoui, R. (1995). « Filtering Duplicated Invocations Using Symmetric Proxies ». Dans *1995 Proceeding of the International Workshop on Object-Orientation in Operating Systems*, Lund, Sweden. IEEE Computer Society.

[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.

[Roy, 1969] Roy, B. (1969). *Algébre moderne et théorie des graphes*, volume 1. Dunod, Paris.

[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.

[Spector et Schawartz, 1983] Spector, A. et Schawartz, P. (1983). « Transactions: A Construct for Reliable Distributed Computing ». *Operating Systems Review*, 17(2):18–35.

[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.

[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.