Convergence des copies dans les environnements collaboratifs répartis

Nicolas Vidot

Numéro d'identification :

Académie de Montpellier

Université Montpellier II
— **Sciences et Techniques du Languedoc** —

Thèse

présentée à l'Université des Sciences et Techniques du Languedoc
pour obtenir le diplôme de DOCTORAT

Spécialité	: **Informatique**
Formation Doctorale	: **Informatique**
École Doctorale	: **Information, Structures, Systèmes**

Convergence des Copies dans les Environnements Collaboratifs Répartis

par

Nicolas Vidot

Soutenue le 20 Septembre 2002 devant le Jury composé de :

Michelle Cart, Maître de conférence, Université Montpellier II, ... Examinatrice
Jean Ferrié, Professeur, Université Montpellier II, ... Directeur de thèse
Rachid Guerraoui, Professeur, Ecole Polytechnique Fédérale de Lausanne, ... Rapporteur
Alain Jean-Marie, Professeur, Université Montpellier II, ... Examinateur
Pascal Molli, Maître de conférence, Université Henri Poincaré, ... Examinateur
Philippe Pucheral, Professeur, Université de Versailles, ... Président du Jury et Rapporteur
Marc Shapiro, Senior researcher, Microsoft Research, Cambridge, ... Examinateur

Numéro d'identification :

Académie de Montpellier

Université Montpellier II
— **Sciences et Techniques du Languedoc** —

Thèse

présentée à l'Université des Sciences et Techniques du Languedoc
pour obtenir le diplôme de DOCTORAT

Spécialité	: **Informatique**
Formation Doctorale	: **Informatique**
École Doctorale	: **Information, Structures, Systèmes**

Convergence des Copies dans les Environnements Collaboratifs Répartis

par

Nicolas Vidot

Soutenue le 20 Septembre 2002 devant le Jury composé de :

Michelle Cart, Maître de conférence, Université Montpellier II, ... Examinatrice
Jean Ferrié, Professeur, Université Montpellier II, ... Directeur de thèse
Rachid Guerraoui, Professeur, Ecole Polytechnique Fédérale de Lausanne, ... Rapporteur
Alain Jean-Marie, Professeur, Université Montpellier II, ... Examinateur
Pascal Molli, Maître de conférence, Université Henri Poincaré, ... Examinateur
Philippe Pucheral, Professeur, Université de Versailles, ... Président du Jury et Rapporteur
Marc Shapiro, Senior researcher, Microsoft Research, Cambridge, ... Examinateur

Remerciements

Cette thèse a été préparée au Laboratoire d'Informatique, de Robotique et Microélectronique (LIRMM) de l'Université Montpellier II. Je tiens à en remercier tout les membres pour leur acceuil.

Jean Ferrié, a dirigé mes travaux durant mes quatre années de thèse. Je le remercie pour sa grande disponibilité ainsi que les commentaires et suggestions pertinentes dont il m'a fait bénéficier.

C'est pour moi un grand honneur que *Phillipe Pucheral* et *Rachid Guerraoui* aient accepté d'être rapporteurs de ma thèse et de les compter parmi les membres de mon jury.

Je remercie sincèrement *Pascal Molli* pour son intérêt pour mes travaux et pour avoir accepté de faire partie de mon jury.

Je remercie vivement *Marc Shapiro* d'avoir accepté de participer à ce jury. Ses remarques profondes et pertinentes ont largement contribué à améliorer ce document.

Je remercie *Michelle Cart* pour l'intérêt qu'elle a accordé à l'évolution de ce travail dont la progression doit beaucoup à ses interrogations et critiques.

Je remercie *Alain Jean-Marie*, d'avoir accepté de participer au jury en qualité d'examinateur.

Je souhaite remercier également le personnel technique et administratif du LIRMM et tout particulièrement *Josette Durante* pour son acceuil et son aide.

Je remercie sincérement les occupants du "Hall ARC", et en particulier *Xavier*, *Olivier* et *Pierre-Alain* pour leur amitié et leur aide.

Je m'en voudrais d'oublier *Dany* et *Etienne* a qui j'adresse un salut tout particulier.

Enfin, pour son soutien et ses encouragements durant mes longues années d'études je voudrais témoigner à ma famille toute ma reconaissance.

A mes parents.

Sommaire

Introduction

Le cadre des recherches présentées dans ce mémoire est celui des applications collaboratives distribuées. Une application collaborative est une application qui assiste un groupe d'utilisateurs dans la réalisation d'une tâche commune en fournissant une interface vers un environnement partagé. Cette définition a été donnée par [EGR91] sous le nom de *groupware* :

> *"computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment."*

Une traduction généralement adoptée de ce terme est : *collecticiel*. Cette définition plutôt vague traduit la très grande hétérogénéité des applications permettant le travail collaboratif. Il est courant de distinguer différentes catégories de collecticiels en se référant aux notions de *temps* et d'*espace*.

Le *temps* permet de distinguer les environnements selon la synchronicité des interactions. Lorsque la synchronicité est élevé, les effets des actions de chaque utilisateur sont visibles en temps réel aux autres utilisateurs. On parle alors d'interactions synchrones. Au contraire, lorsque la synchronicité est faible, un temps non négligeable peut s'écouler entre le moment où un utilisateur effectue une action et le moment où les effets de celle-ci sont visibles aux autres utilisateurs. On parle alors d'interactions asynchrone.

La notion d'*espace* permet de distinguer les environnements selon la répartition géographique des utilisateurs. Lorsque les utilisateurs sont physiquement distants, les environnements sont qualifiés de répartis.

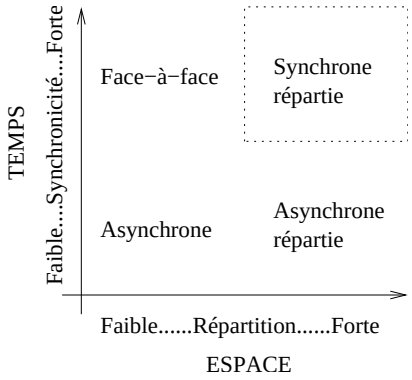


FIG. 1 – Classification des collecticiels en fonction du type d'interaction

La considération de ces deux critères permet d'établir une matrice (fig. 1), qui définit quatre grandes catégories d'applications suivant le type d'interaction mis en œuvre. Cette

étude concerne les applications collaboratives appartenant à la catégorie des applications synchrones réparties

Un des objectif, lors de la conception d'une application collaborative, est de rendre son utilisation aussi agréable qu'une application classique. Pour cela, il est en particulier nécessaire de garantir un temps de réponse aussi court que possible. Dans le contexte, d'une application répartie, cela suppose donc que l'accès aux objets de l'environnement partagé ne nécessite pas de communication avec un site distant. Pour parvenir à cet objectif, il faut que les objets de l'environnement partagés soient répliqués sur les différents sites. Ainsi l'accès à un objet de l'environnement partagé peut continuer à se faire pour un site de façon locale en accédant à la copie qu'il détient.

Cependant, les accès concurrents aux copies d'un même objet peuvent entraîner une divergence entre les états de ces copies. Il faut donc mettre en place une procédure de contrôle de la cohérence. Les algorithmes auxquels nous nous intéressons dans cette thèse permettent précisément d'assurer ce contrôle. Ils sont basés sur la diffusion, entre les sites participants à la collaboration, des opérations qui ont permis de faire évoluer leurs copies locales. La ré-exécution de ces opérations sur les autres sites devrait permet de faire évoluer les différentes copies vers un même état, c.-à-d. d'assurer la convergence des copies. Sur la figure 2 par exemple, les copies des objets du site 1 et du site 5 ont convergé car le site 1 a reçu et exécuté l'opération op_3 générée et exécutée sur le site 5. Cette opération colore en gris le rectangle. Sur les sites 2 et 4 où l'opération est en cours de réception et sur le site 3 vers lequel l'opération est en cours de diffusion, les copies sont encore divergentes car l'opération op_3 n'a pas encore été exécutée.

Cet exemple est un cas simple dans la mesure où lors de l'exécution de op_3 sur les sites 1 à 4, les copies de l'objet sont dans le même état que celui où se trouvait la copie de l'objet sur le site 5 au moment de la génération de op_3. L'opération peut donc être exécutée sans problème lorsqu'elle est reçue. Le même état est obtenu sur les cinq sites.

La problématique de la convergence des copies dans le cadre d'applications collaboratives tient justement au fait que ce cas de figure n'est pas toujours assuré. En effet, dans le cas général, les utilisateurs peuvent générer des opérations en concurrence et de ce fait l'état d'un objet sur un site peut ne pas être le même que celui sur lequel ont été définies les opérations en provenance des autres sites. Une approche intéressante introduite par [EG89] consiste à transformer les opérations reçues pour tenir compte de celles qui ont pu entre-temps modifier l'état de l'objet. Cette approche utilisant les transformées opérationnelles, a donné lieu à de nombreuses publications. Cependant, les solutions proposées soit mettent en œuvre des algorithmes [RNRG96, SCF97, SE98] soumis à la vérification d'une condition très contraignante [Sul98], soit ne parviennent pas toujours à assurer la convergence des copies [EG89, SJZY97]. Plus récemment, l'extension de ces algorithmes pour permettre à un utilisateur d'annuler une opération dans le cadre d'une application collaborative a été envisagée. Il est apparu que ce traitement est loin d'être trivial et que les propositions qui ont été faites dans la littérature soit étaient relativement restrictives sur les conditions dans lesquelles l'annulation pouvait avoir lieu [RG99], soit se sont avérées, après analyse approfondies, fausses et incapables d'assurer la convergence des copies

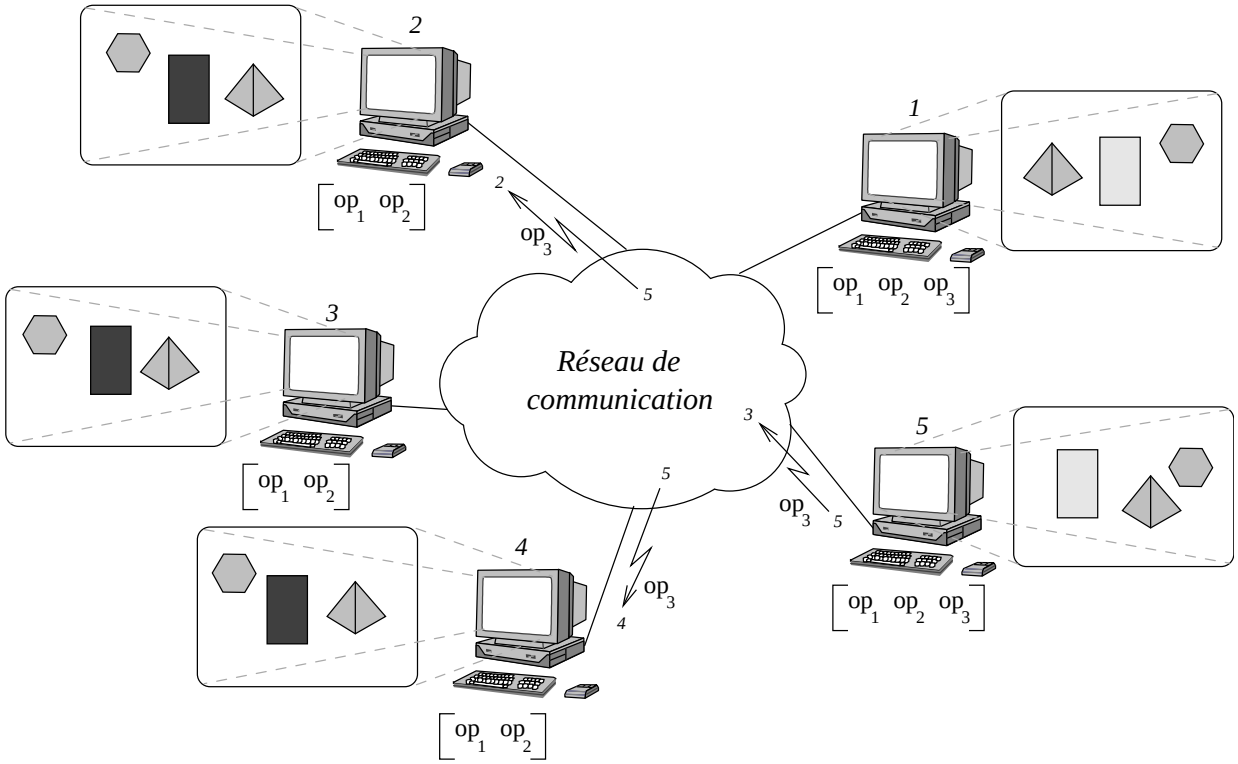


FIG. 2 – Évolution des copies par diffusion des opérations

en cas d'annulation [Sun00].

Notre objectif dans cette thèse est donc double. D'une part, il est de proposer une approche nouvelle [VCFS00] permettant de garantir effectivement la convergence des copies et ne présentant pas les inconvénients des approches existantes. En particulier, les deux algorithmes que nous proposons permettent de s'affranchir des conditions contraignantes qu'imposent les algorithmes existants. D'autre part, nous présentons une étude détaillée des exigences relatives au traitement de l'annulation. Nous montrons à cette occasion en quoi les solutions existantes posent problème et enfin nous proposons une extension aux algorithmes existants [SCF98, SE98] permettant de prendre en compte l'annulation des opérations.

Plan de la thèse

Ce document est organisé en deux parties :

1. La première partie concerne la proposition de nouveaux algorithmes, pour les systèmes collaboratifs, qui ne nécessitent plus la vérification d'une condition particulièrement contraignante, appelée condition C2. Cette partie comporte quatre chapitres. Le chapitre 1 présente la problématique générale des systèmes collaboratifs utilisant des algorithmes basés sur les transformées opérationnelles (c.-à-d. transposition en avant et aussi transposition en arrière). Le chapitre 2, après avoir introduit un cadre

de référence, présente les approches existantes et précise leurs points faibles. Le chapitre 3 présente les deux nouveaux algorithmes que nous avons conçus et qui n'imposent pas la vérification de la condition C2 ; il les compare ensuite aux différents algorithmes proposés dans la littérature. Le chapitre 4 propose une extension de ces nouveaux algorithmes pour supporter la collaboration en présence de sites mobiles.

2. La deuxième partie traite du problème de l'annulation d'opération dans les systèmes collaboratifs. Le chapitre 5 expose la problématique générale de l'annulation dans les systèmes utilisant un algorithme basé sur les transformées opérationnelles (c.-à-d. transposition en avant et aussi transposition en arrière). Le chapitre 6 introduit deux nouvelles conditions nécessaires à la réalisation de l'annulation. Le chapitre 7 présente les limites des algorithmes existants basés sur les transpositions et qui autorisent l'annulation d'opération. Enfin, le chapitre 8 propose une solution basée sur des algorithmes de type SOCT2 [SCF97, SE98], permettant d'obtenir l'annulation des opérations sans que la vérification des deux nouvelles conditions introduites ne constitue une contrainte.

Première partie

Algorithmes assurant la convergence des copies

1

Problématique

Sommaire

1.1 Introduction

Un système collaboratif réparti est constitué d'un ensemble de sites usagers (un usager par site) interconnectés par un réseau que nous supposerons fiable. Chaque objet (c.-à-d. texte, graphique, ...) partagé par les usagers est répliqué de sorte qu'une copie de l'objet existe sur chaque site.

Chaque objet peut être manipulé au moyen d'opérations spécifiques. Afin de maintenir la cohérence des copies, toute opération générée et exécutée par un usager sur sa copie, doit être exécutée sur toutes les autres copies.

Cela suppose que toute opération *générée* sur un site soit *diffusée* à tous les autres sites ; après avoir été *reçue* sur un site, l'opération est *exécutée* sur la copie locale de l'objet. Pour un site donné, une opération *locale* est une opération générée sur ce site, tandis qu'une opération *distante* est une opération générée sur un autre site.

Afin de garantir aux usagers un temps de réponse minimum, toute opération générée sur un site (c.-à-d. toute opération locale à ce site) doit être exécutée immédiatement après avoir été générée sur ce site.

Cette section présente les trois principaux problèmes posés par le maintien de la cohérence des copies d'un objet, à savoir (1) la préservation de la causalité, (2) la préservation des intentions des utilisateurs et (3) la convergence des copies.

Un éditeur de texte collaboratif servira d'exemple. Soit un texte composé d'un ensemble de phrases ordonnées. Chaque phrase est un objet représenté par une chaîne de caractères. Les opérations définies sur cet objet sont :

> `insérer(`p, c`)` **:** insère le caractère c à la position p dans la chaîne,
>
> `effacer(`p`)` **:** efface le caractère à la position p.

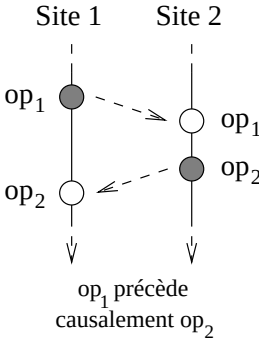
Dans la suite, on supposera que plusieurs usagers travaillent simultanément sur la même phrase et la modifient.

1.2 Respect de la causalité

Le premier problème rencontré concerne l'exécution d'opérations qui sont reliées par une relation de précédence causale.

On dit que l'opération op_1 *précède causalement* l'opération op_2 (noté op_1 *précède*$_c$ op_2) ssi op_2 a été générée sur un site après l'exécution de op_1 sur ce site.

Le fait que op_2 ait été générée après l'exécution de op_1 implique que op_2 a vu les éventuelles modifications effectuées par op_1 et en tient compte implicitement. En conséquence, op_2 est supposée être dépendante de l'effet produit par op_1. Le problème consiste alors à exécuter les opérations diffusées à tous les sites, tout en respectant la causalité.



Par exemple, considérons la figure 1.1 dans laquelle toutes les copies sont dans le même état initial “x”. Le site 2 génère et exécute l'opération `effacer`(1) après avoir reçu (du site 1) et exécuté l'opération `insérer`(1, ‘y’).

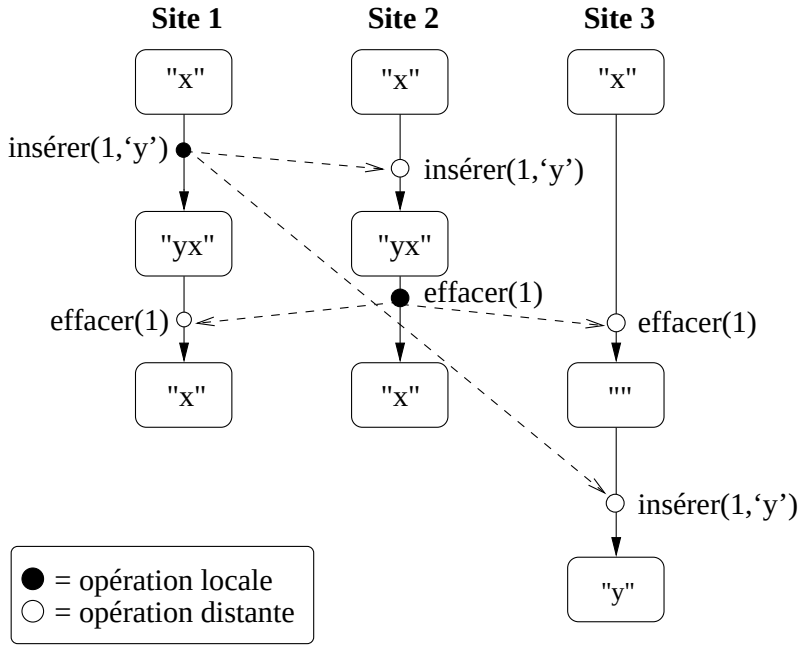


FIG. 1.1 – Résultat erroné sur le site 3 dû à l'absence de livraison causale

Dans ce cas `insérer`(1,‘y’) *précède*$_C$ `effacer`(1). Si ces opérations sont délivrées et exécutées dans un ordre différent de l'ordre causal sur un autre site (par exemple le site 3), alors leurs effets produiront une copie incohérente.

Pour cette raison, la relation de précédence causale doit être respectée sur tous les sites. En d'autres termes, si op_1 *précède*$_C$ op_2, alors op_1 doit être exécutée avant op_2 sur tous les sites.

Le respect de la causalité est résolu dans toutes les méthodes connues, dOPT [EG89], adOPTed [RNRG96], GOT [SJZY97], SOCT2 [SCF97], GOTO [SE98], par une technique permettant de :

1. déterminer si une opération en précède causalement une autre,
2. garantir une livraison causale, c'est à dire de déduire, lorsqu'un site reçoit une opération distante, s'il existe une opération qui la précède causalement et qui n'a pas encore été délivrée.

Cette technique repose sur l'utilisation d'un *vecteur d'état* SV_S associé à chaque site S et à chaque objet.

Un vecteur d'état est une variante de l'horloge vectorielle [Mat89]. Plus précisément, la signification de la composante $SV_S[j]$ est la suivante (avec $1 \leq j \leq N$, N étant le nombre de sites) : $SV_S[j]$ = nombre d'opérations générées par le site j, reçues par le site S et exécutées sur la copie de l'objet.

La composante $SV_S[j]$ est incrémentée sur le site S lorsqu'une opération qui a été générée sur le site j est exécutée. Il peut s'agir, si $j = S$, de l'exécution d'une opération locale ou, si $j \neq S$, de l'exécution, après qu'elle ait été délivrée, d'une opération distante.

Ainsi, toute opération *op* générée sur un objet par le site S_{op} est estampillée par la valeur du vecteur de son site, noté SV_{op}, avant d'être diffusée. À sa réception sur un autre site S', elle n'est délivrée que lorsque toutes les opérations qui la précèdent causalement, ont été délivrées, c'est à dire lorsque le vecteur d'état $SV_{S'}$ du site S' est tel que : $SV_{S'}[i] \geq SV_{op}[i]$, $(\forall i : 1 \leq i \leq N)$. Cette technique est mise en œuvre dans la procédure dite de *livraison causale* qui est définie et prouvée dans [SCF97].

Sur la figure 1.2, on a mentionné les vecteurs d'états. Les vecteurs associés aux différents états de l'objet sont représentés sous ces états et évoluent avec eux au fur et à mesure de l'exécution des opérations. Les vecteurs associés aux opérations sont représentés sur les arcs matérialisant la transmission des opérations.

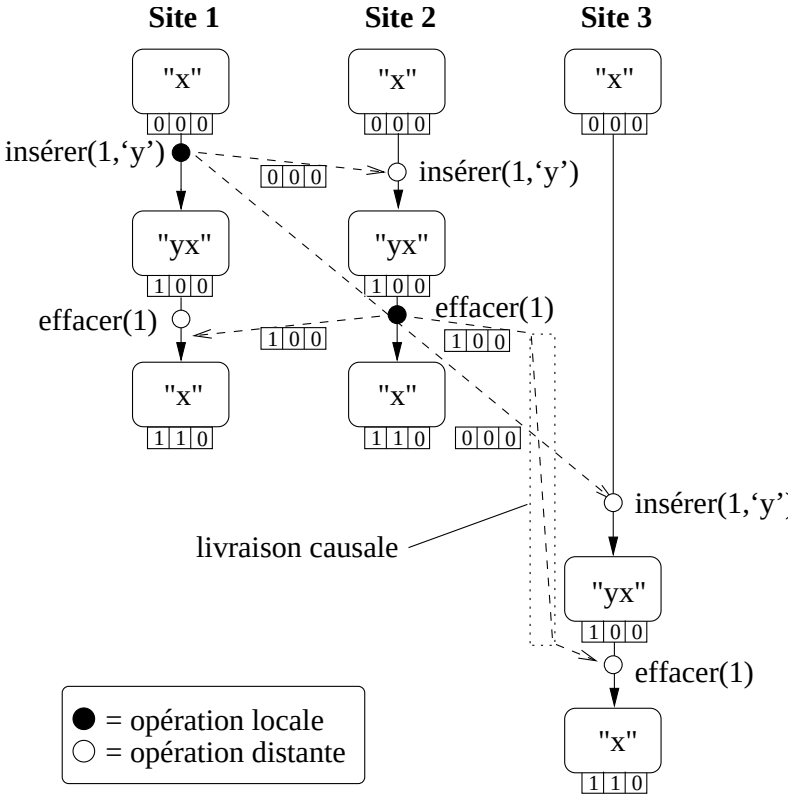


FIG. 1.2 – Résultat correct grâce à l'utilisation de la livraison causale

L'opération `insérer`$(1, \text{'y'})$ est générée sur le site 1 alors que l'objet est dans l'état `"x"` auquel est associé le vecteur d'état $\boxed{0|0|0}$. Ce dernier est donc utilisé pour estampiller l'opération `insérer`$(1, \text{'y'})$ lorsqu'elle est diffusée. Cette opération, une fois exécutée sur le site 2, fait passer l'objet dans l'état `"yx"` et provoque l'évolution du vecteur d'état qui devient $\boxed{1|0|0}$. L'opération `effacer`(1) est générée, sur le site 2, à partir de cet état. Elle est donc diffusée, associée au vecteur d'état $\boxed{1|0|0}$. Ainsi, lorsqu'elle est reçue sur le site 3,

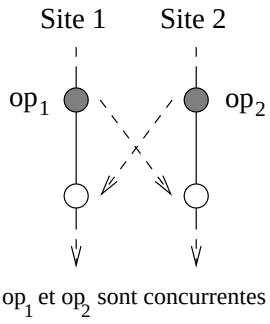
la procédure de livraison causale, décrite plus haut, permet, en exploitant les vecteurs d'état, de retarder la livraison de l'opération `effacer`(1) jusqu'à ce que l'opération `insérer`(1, 'y') soit reçue et livrée elle-même, ceci afin d'assurer une livraison suivant l'ordre causal.

Lorsque les opérations ne sont pas causalement reliées, elles sont concurrentes. Plus précisément op_1 et op_2 sont concurrentes (noté $op_1 \,//\, op_2$) ssi non (op_1 *précède*$_C$ op_2) et non (op_2 *précède*$_C$ op_1).

Enfin, pour deux opérations quelconques op_i et op_j, il est possible de déduire que :
- op_i *précède*$_C$ $op_j \Leftrightarrow SV_{op_i}[S_{op_i}] < SV_{op_j}[S_{op_i}]$.
- op_j *précède*$_C$ $op_i \Leftrightarrow SV_{op_j}[S_{op_j}] < SV_{op_i}[S_{op_j}]$.
- $op_j \,//\, op_i \Leftrightarrow SV_{op_j}[S_{op_i}] \leq SV_{op_i}[S_{op_i}] \wedge SV_{op_i}[S_{op_j}] \leq SV_{op_j}[S_{op_j}]$.

1.3 Intentions de l'usager

Dans le cas où op_1 et op_2 sont concurrentes, aucune n'a vu les effets de l'autre : elles sont indépendantes. Elles peuvent de ce fait être exécutées sur les différents sites dans des ordres quelconques. Cependant, si l'on exécute op_1 avant op_2, il faudra, lors de l'exécution de op_2, tenir compte des effets de op_1 de façon à respecter l'*intention* de l'usager 2.



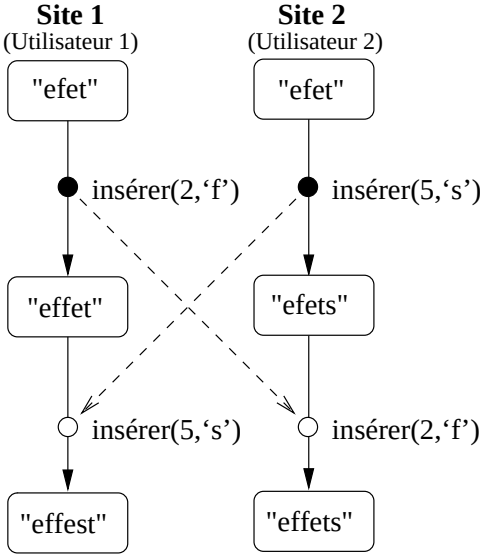


FIG. 1.3 – Non-respect des intentions de l'utilisateur 2, sur le site 1.

Dans l'exemple de la figure 1.3, les deux usagers travaillent simultanément sur le même objet dont l'état initial est "efet". L'intention de l'usager 1 est d'ajouter un 'f' en position 2

pour obtenir "effet" ; son intention est réalisée par l'opération `insérer`(2, 'f'). L'intention de l'usager 2 est d'ajouter 's' à la fin du mot ; son intention est réalisée par l'opération `insérer`(5, 's'). Quand cette opération est délivrée et exécutée sur le site 1, le nouvel état résultant "effest" ne traduit pas l'intention de l'usager 2.

Le problème vient du fait que l'opération réalisant l'intention de l'usager est définie relativement à un certain état de l'objet. Si sur un site distant, l'exécution d'une opération concurrente modifie cet état, l'opération qui réalisait initialement l'intention de l'usager peut réaliser une intention différente. La solution consiste donc à transformer l'opération à exécuter pour tenir compte des opérations concurrentes qui ont été sérialisées avant elle.

Cette transformation est possible si l'on a défini au préalable une fonction, spécifique de la sémantique des opérations, qui donne pour chaque couple d'opérations concurrentes prises dans cet ordre (op_1, op_2) une opération, notée $op_2^{op_1}$, réalisant la même intention que op_2 mais à partir de l'état résultant de l'exécution de op_1.

Cette fonction de transformation qui a été introduite dans dOPT [EG89] est également utilisée dans les autres systèmes adOPTed [RNRG96], GOT [SJZY97], SOCT2 [SCF97] et GOTO [SE98] sous diverses appellations. Nous utilisons l'appelation donnée dans [SCF97, SCF98] à savoir *transposition en avant*. En notant O_i l'état de l'objet, $O_i.op$ l'état obtenu après l'exécution de op et $Intention(op, O_i)$ l'intention de l'usager lorsque son opération op est exécutée à partir de l'état O_i, la transposition en avant s'écrit formellement :

$$\text{Transpose\_av}(op_1, op_2) = op_2^{op_1}, \text{ avec :}$$
$$\forall\ O_i,\ Intention(op_2^{op_1}, O_i.op_1) = Intention(op_2, O_i).$$

L'exemple 1.1 illustre la transposition en avant de l'opération `insérer`(p_2, c_2) par rapport à l'opération `insérer`(p_1, c_1) pour un objet chaîne de caractère. Appliquée à l'exemple de la figure (Fig. 1.3), elle permet de respecter l'intention de l'usager 2, en transformant son opération pour exécuter `insérer`(6, 's') au lieu de `insérer`(5, ''s') (Fig. 1.4).

Exemple 1.1 – Transposition en avant.

```
Transpose_av(insérer(p1, c1), insérer(p2, c2)) =
   cas
      (p1 < p2) retour insérer(p2 + 1, c2) ;
      (p1 ≥ p2) retour insérer(p2, c2) ;
   fincas
```

Plus généralement, la transposée en avant d'une opération op par rapport à une séquence d'opérations seq_n, notée op^{seq_n}, est l'opération obtenue après les transpositions en avant successives de op par rapport aux opérations de la séquence. op^{seq_n} est définie formellement de façon récursive par $op^{seq_n} =$ `Transpose_av`$(op_n, op^{seq_{n-1}})$ avec $seq_n = op_1.op_2.\ldots.op_n = seq_{n-1}.op_n$ et $op^{seq_0} = op$, où la notation $op_i.op_j$ signifie l'exécution dans cet ordre de op_i suivie de op_j.

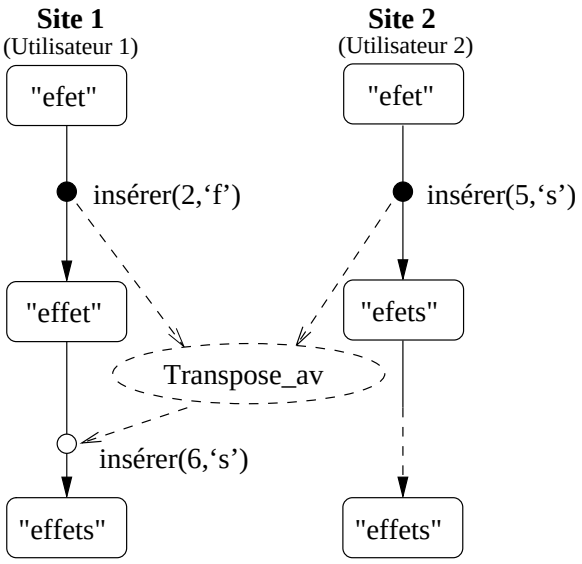


FIG. 1.4 – Utilisation de la transposition en avant permettant le respect des intentions de l'utilisateur 2, sur le site 1.

Il est important de noter que la transposition en avant de l'opération op_2, pour tenir compte de l'opération concurrente op_1, nécessite que les deux opérations soient générées à partir du même état de l'objet. S'il n'en est pas ainsi, alors l'intention de l'usager peut être violée.

Ce problème dit de la *concurrence partielle* peut être illustré par une situation dans laquelle une opération, disons op_3, est concurrente à une séquence d'opérations $op_1.op_2$. Sur la figure 1.5, nous observons que op_1 précède causalement op_2, que op_3 et op_1 sont concurrentes et générées à partir du même état "comédy", mais que op_3 est partiellement concurrente à op_2 dans la mesure où op_2 et op_3 ne sont pas générées à partir du même état.

Sur le site 1, quand $op_3 =$ `effacer`(6) est reçue, elle est transposée en avant successivement avec op_1 puis avec op_2. L'opération résultante, notée $op_3^{op_1.op_2}$, est `effacer`(8) qui conduit au mot "comédie". Sur le site 2, la transposition en avant de op_1 avec op_3 donne $op_1^{op_3} =$ `insérer`$(6, \text{'i'})$; mais la transposition en avant de op_2 avec op_3 donne $op_2^{op_3}$ = `insérer`$(6, \text{'e'})$ qui conduit à l'état "comédei" violant par là l'intention de l'usager 1.

Pour résoudre le problème de la concurrence partielle mis en évidence dans [All89] différentes solutions ont été proposées de façon à ce que la transposition en avant puisse être appliquée correctement.

Dans GOT [SJZY97] l'opération op_2 est transformée en utilisant la fonction inverse (appelée Exclusion-transformation) de la transposition en avant, de façon à ce qu'elle soit définie sur le même état que op_3 et qu'on puisse appliquer la transposition en avant.

Dans adOPTed [RNRG96] les opérations reçues sont conservées sous leur forme originale ainsi que certains résultats obtenus de transpositions précédentes. Cela rend possible le calcul de la transposée en avant de op_3 définie sur le même état que op_2. On obtient ainsi une opération op_3' par rapport à laquelle on peut alors transposer en avant op_1.

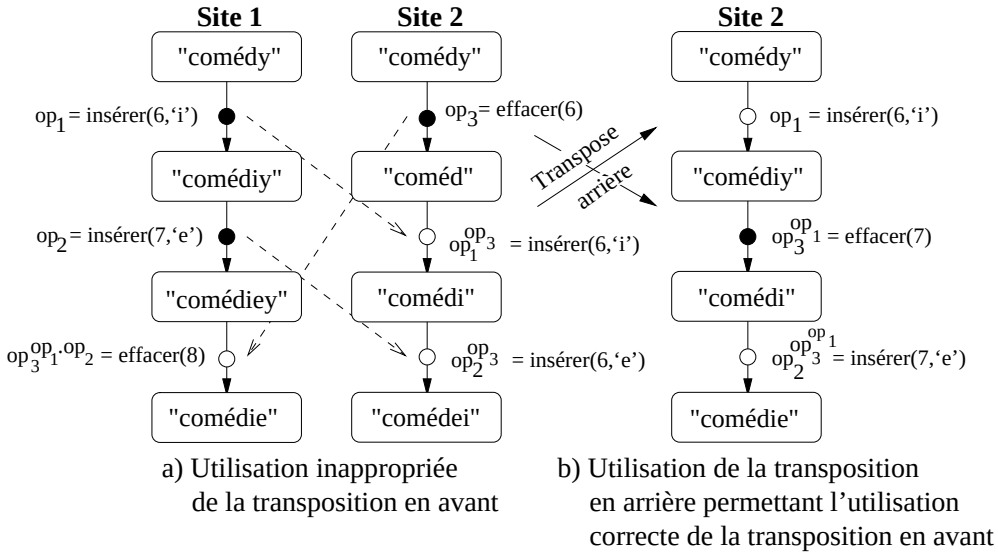
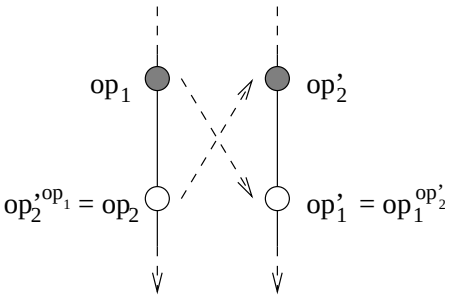


FIG. 1.5 – Exemple d'opérations en concurrence partielle

Dans SOCT2 [SCF97] une nouvelle transformation appelée *transposition en arrière*, est définie. Cette transformation a également été reprise sous l'appellation `transpose` dans GOTO [SE98]. **Cette fonction permet de changer l'ordre d'exécution de deux opérations consécutives sans que leur intention soit violée.** Plus précisément, la transposition en arrière d'un couple d'opérations (op_1, op_2) exécutées dans cet ordre, rend en résultat le couple d'opérations (op_2', op_1') correspondant à leur exécution dans l'ordre inverse, conduisant au même état et compatible avec la transposition en avant. Soit formellement :

$$\text{Transpose\_ar}(op_1, op_2) = (op'_2, op'_1)$$
$$\text{avec : } op_2 = \text{Transpose\_av}(op_1, op'_2)$$
$$\text{et } op'_1 = \text{Transpose\_av}(op'_2, op_1).$$



La transposition en arrière n'est définie que pour une séquence d'opérations (op_1, op_2) obtenue à partir d'opérations concurrentes (op_1, op_2'). On peut donc caractériser plus simplement cette transformation par :

$$\text{Transpose\_ar}(op_i, op_j^{op_i}) = (op_j, op_i^{op_j})$$

L'exemple suivant précise la transposition en arrière du couple d'opérations (`insérer`(p_1, c_1), `insérer`(p_2)) considérées dans cet ordre.

Exemple 1.2 – Transposition en arrière.

```
Transpose_ar(insérer(p1, c1), effacer(p2)) =
    cas (p1 < p2) retour(effacer(p2 − 1),insérer(p1, c1)) ;
        (p1 > p2) retour(effacer(p2), insérer(p1 − 1, c1)) ;
    fincas
```

Dans l'exemple de la figure 1.5, en appliquant, sur le site 2, la transposition en arrière au couple (op_3, $op_1^{op_3}$) on obtient le couple (op'_1, op'_3), c'est à dire les opérations (`insérer`$(6, \text{'i'})$, `effacer`(7)). Les opérations op_2 et op'_3 sont maintenant définies sur le même état ; en transposant en avant op_2 avec op'_3 nous obtenons l'opération `insérer`$(7, \text{'e'})$ dont l'exécution conduit au bon résultat.

1.4 Convergence des copies

La prise en compte de la causalité ainsi que des intentions des usagers ne suffit pas à obtenir dans tous les cas des exécutions qui assurent la convergence des copies sur les différents sites. Considérons deux opérations concurrentes op_1 et op_2, générées à partir du même état et exécutées, après avoir été transposées en avant, dans des ordres différents sur deux sites (Fig. 1.6).

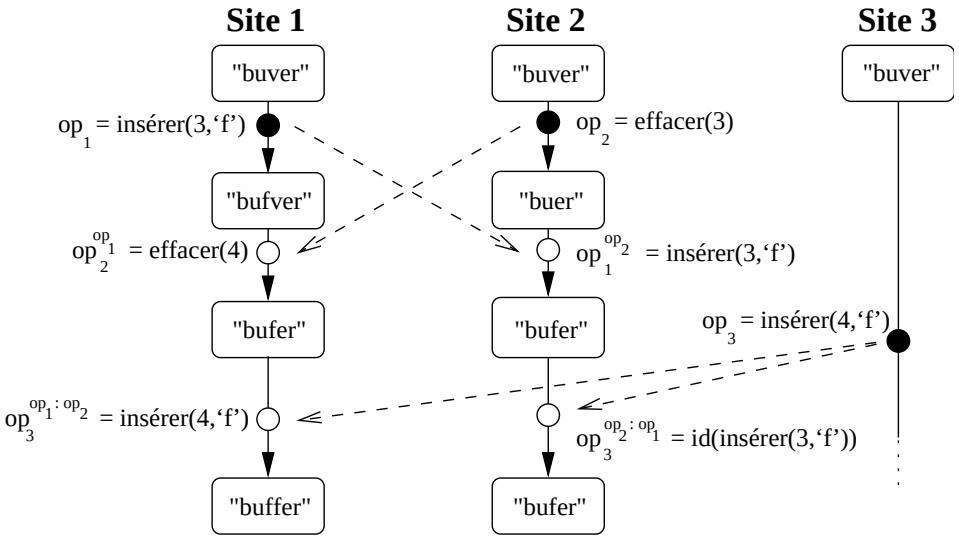


FIG. 1.6 – Convergence des copies

Pour garantir la cohérence des copies, la transposition en avant doit vérifier deux conditions. Tout d'abord, il est nécessaire que l'exécution (sur le site 1) de op_1 suivie de celle de $op_2^{op_1}$ produise le même état que l'exécution (sur le site 2) de op_2 suivie de celle de $op_1^{op_2}$. Cette condition C1 s'exprime formellement par :

Définition 1.1 (Condition C1)

Soit op_1 et op_2 deux opérations concurrentes définies sur le même état. La transposée en avant vérifie la condition C1 ssi :

$$op_1.op_2^{op_1} \equiv op_2.op_1^{op_2}$$

où $\equiv$ dénote l'équivalence d'état après avoir exécuté chacune des séquences à partir du même état.

Dans l'exemple de la figure 1.6, on suppose que les fonctions utilisées pour transposer en avant op_1 par rapport à op_2 et op_2 par rapport à op_1 satisfont à la condition C1. De ce fait, les exécutions de $op_1.op_2^{op_1}$ et de $op_2.op_1^{op_2}$ fournissent bien toutes deux le même état, `"bufer"`.

Par contre, la fonction de transposition en avant de l'opération `insérer`(p_2, c_2) par rapport à l'opération `insérer`(p_1, c_1) donnée dans l'exemple 1.1 ne satisfait pas à la condition C1. En effet, considérons les deux opérations $op_i =$ `insérer`$(1,$ 'a') et $op_j =$ `insérer`$(1,$ 'b') définies sur l'objet dans l'état initial `""`. D'après la définition, on a `Transpose_av(`op_i, op_j`)` = `insérer`(1,'b') et `Transpose_av(`op_j, op_i`)` = `insérer`(1,'a'). Par conséquent, l'exécution de la séquence $op_i.op_j^{op_i}$ fournit l'état `"ba"` et l'exécution de la séquence $op_j.op_i^{op_j}$ fournit l'état `"ab"`. Les deux états ne sont pas équivalents ce qui prouve bien la violation de la condition C1.

Pour que celle-ci soit vérifiée, il faut tenir compte du cas particulier où les deux insertions sont faites à la même position. Dans ce cas, il existe plusieurs solutions correctes qui fournissent des résultats divergents, on doit donc faire un choix arbitraire (Dans certains systèmes [MSMO02], ce choix arbitraire est défini comme un conflit qui appellera plus tard une réconciliation de la part des différents utilisateurs). Cependant, **il faut garantir que le même choix sera fait sur tous les sites**. En se reportant à la transposition en avant de l'exemple 1.1, on peut garantir l'unicité de ce choix en utilisant un ordre total sur les caractères. La fonction `code` associe à chaque caractère un entier différent définissant ainsi un tel ordre, basé par exemple sur l'ordre alphabétique. La définition de la transposition en avant devient alors :

Exemple 1.3 – Transposition en avant vérifiant la condition C1.

```
Transpose_av(insérer(p1, c1), insérer(p2, c2)) =
   cas
      (p1 < p2) retour insérer(p2 + 1, c2) ;
      (p1 > p2) retour insérer(p2, c2) ;
      (p1 = p2)
         cas
            (code(c1) < code(c2)) retour insérer(p2, c2) ;
            (code(c2) < code(c1)) retour insérer(p2 + 1, c2) ;
            (code(c1) = code(c2)) retour identité(insérer(p2, c2)) ;
```

```
        fincas
    fincas
```

L'opération `identité`, dont l'exécution n'a aucun effet sur l'objet, résulte de la transposition d'une opération par rapport à une opération concurrente réalisant la même intention.

En utilisant cette nouvelle définition, on a `Transpose_av`(op_i, op_j) = `insérer`(2,'b') et `Transpose_av`(op_j, op_i) = `insérer`(1,'a'). Par conséquent, l'exécution de la séquence $op_i.op_j^{op_i}$ fournit l'état `"ab"` et l'exécution de la séquence $op_j.op_i^{op_j}$ fournit l'état `"ab"`. Les deux états sont équivalents ce qui grâce à la vérification de la condition C1.

Supposons maintenant qu'une opération op_3, concurrente à op_1 et op_2, soit générée sur le site 3. Lorsqu'elle est reçue par chacun des sites, elle est transposée en avant à son tour. Pour garantir la convergence des copies, la transposition en avant d'une opération par rapport à une séquence de deux (ou plusieurs) opérations concurrentes ne doit pas dépendre de l'ordre dans lequel les opérations de la séquence sont transposées. Pour cela, en plus de vérifier la condition C1, la transposition en avant doit aussi vérifier la condition C2.

Définition 1.2 (Condition 2)

Quelles que soit les opérations op_1, op_2 et op_3, la transposée en avant vérifie la condition C2 ssi :

$$op_3^{op_1:op_2} = op_3^{op_2:op_1}$$

où $op_i : op_j$ dénote $op_i.op_j^{op_i}$.

L'exemple de la figure 1.6, illustre une situation dans laquelle la transposition en avant de op_3 par rapport aux séquences $op_1.op_2^{op_1}$ et $op_2.op_1^{op_2}$ ne vérifie pas la condition C2 dans la mesure où l'opération obtenue est différente sur le site 1 et sur le site 2. En effet, considérons la transposition en avant de `insérer` par rapport à `effacer` vérifiant la condition C1 :

Exemple 1.4 – Transposition en avant vérifiant la condition C1.

```
Transpose_av(effacer(p1, c1), insérer(p2, c2)) =
    cas
        (p1 < p2) retour insérer(p2 − 1, c2) ;
        (p1 ≥ p2) retour insérer(p2, c2) ;
    fincas
```

Il en résulte que la transposée en avant de op_3 par rapport à op_1 est l'opération $op_3^{op_1}$ = `Transpose_av`(`insérer`(3, 'f'), `insérer`(4, 'f')) = `insérer`(5, 'f') et la transposée en avant de $op_3^{op_1}$ par rapport à $op_2^{op_1}$ est l'opération $op_3^{op_1\, op_2^{op_1}}$ = `Transpose_av`(`effacer`(4), `insérer`(5, 'f')) = `insérer`(4, 'f'). C'est cette dernière opération qui sera exécutée sur le site 1.

De la même manière, la transposée en avant de op_3 par rapport à op_2 est l'opération $op_3^{op_2}$ = `Transpose_av`(`effacer`(3), `insérer`(4, 'f')) = `insérer`(3, 'f') et la transposée en avant de $op_3^{op_2}$ par rapport à $op_1^{op_2}$ est l'opération $op_3^{op_2\, op_1^{op_2}}$ = `Transpose_av`(`insérer`(3, 'f'), `insérer`(3, 'f')) = `id`(`insérer`(3, 'f')). C'est cette dernière opération qui est exécutée sur le site 2.

L'exécution de op_3 se fera donc sous la forme de deux transposées différentes, chacune ayant des effets différents sur la copie de l'objet. Or à l'instant où elle intervient, les deux copies de l'objet sont dans le même état sur les deux sites. Il en résultera donc une divergence des copies.

Dans [Sul98] il est montré que satisfaire à la condition C2 nécessite dans ce cas particulier l'ajout de paramètres supplémentaires dans les opérations qui réalisent l'insertion. Ces paramètres, *av* et *ap*, servent à conserver les identifiants des opérations d'effacement, concurrentes à l'opération d'insertion, qui ont effacé un caractère à une position placée respectivement ***av**ant* ou ***ap**rès* la position où l'opération d'insertion a inséré le sien.

La définition de la transposition en avant, de l'opération `insérer` par rapport à l'opération `insérer`, respectant les conditions C1 et C2 est alors la suivante :

Exemple 1.5 – Transposition en avant vérifiant la condition C1 et la condition C2.

```
Transpose_av(insérer(p1, c1, av1, ap1), insérer(p2, c2, av2, ap2)) =
   cas
      (p1 < p2) retour insérer(p2 + 1, c2, av2, ap2) ;
      (p1 > p2) retour insérer(p2, c2, av2, ap2) ;
      (p1 = p2)
         cas
            (ap1 ∩ av2 ≠ ∅) retour insérer (p2 + 1, c2, av2, ap2) ;
            (av1 ∩ ap2 ≠ ∅) retour insérer(p2, c2, av2, ap2) ;
            (code(c1) < code(c2)) retour insérer(p2, c2, av2, ap2) ;
            (code(c2) < code(c1)) retour insérer(p2 + 1, c2, av2, ap2) ;
            (code(c1) = code(c2)) retour identité(insérer(p2, c2, av2, ap2))
         fincas
   fincas
```

On peut noter à ce propos que l'écriture de transposée en avant vérifiant la condition C2 peut, dans certains cas, être problématique. Ce point est détaillé au chapitre 3.

Les systèmes adOPTed [RNRG96], SOCT2 [SCF97] et GOTO [SE98] reposent sur la satisfaction des conditions C1 et C2. Seul GOT [SJZY97] ne les utilise pas mais en contrepartie impose sur tous les sites un ordre de sérialisation unique (compatible avec l'ordre causal) pour les opérations, ce qui oblige à Défaire / Refaire des opérations pour se

conformer à cet ordre. Dans dOPT [EG89] la condition C2 n'est pas requise au détriment de la convergence des copies.

Dans ce contexte, notre objectif est de proposer dans cette thèse une nouvelle alternative à la convergence des copies permettant de s'affranchir de la contrainte liée à la vérification de la condition C2, et garantissant qu'aucune opération ne soit défaite ni refaite.

2

État de l'art

Sommaire

2.1 Introduction

Le chapitre précédent a présenté séparément les différents problèmes à résoudre dans un système collaboratif temps réel à savoir : (1) respect de la causalité, (2) respect de l'intention et (3) convergence des copies, et a fourni les bases théoriques de leur solution. Ce chapitre présente, dans un premier temps, un cadre de référence, permettant de décrire les algorithmes. Puis, dans un deuxième temps, les différents algorithmes proposés dans la littérature sont discutés mettant ainsi en lumière les différentes mises en œuvre des solutions théoriques à ces problèmes.

2.2 Cadre de référence

Le traitement des opérations en environnement réparti se décompose sur chaque site en trois grandes étapes. Nous les nommons **Exécution locale**, **Livraison causale** et **Intégration**.

Exécution locale est l'étape consécutive à la génération d'une opération sur un site. C'est l'étape au cours de laquelle l'opération générée est exécutée localement et où elle est diffusée, sous la forme d'un message $\langle op, S_{op}, SV_{op} \rangle$, à tous les autres sites.

Livraison causale est l'étape qui commence lorsqu'une opération, locale ou distante, est reçue sur un site. Elle a pour but d'assurer la transmission de l'opération suivant l'ordre causal à la procédure **Intégration**.

Intégration est l'étape au cours de laquelle est traitée avant d'être exécutée l'opération transmise par la livraison causale. Le traitement consiste éventuellement à transposer l'opération pour permettre son exécution sur l'état courant de l'objet. Il s'agit de l'étape qui recèle le plus de difficultés et par laquelle les algorithmes existants se différencient.

2.3 Problématique de l'intégration

L'intégration sur le site S d'une opération distante a pour but de permettre son exécution à partir de l'état courant de l'objet. Elle consiste à obtenir, l'opération op' qui est définie sur l'état courant de l'objet et qui réalise la même intention que l'opération op générée sur le site distant. Cela suppose que op soit transposée en avant par rapport aux opérations, déjà exécutées sur le site S, qui lui sont concurrentes. Cela suppose donc qu'on ait conservé la trace des opérations exécutées sur le site S. Cette exigence est réalisée grâce à l'utilisation d'une structure linéaire appelée *Histoire*. Les informations conservées dans cette structure varient suivant les algorithmes. Nous les préciserons dans la suite.

Néanmoins, la transposition en avant d'une opération op, par rapport aux opérations du site qui lui sont concurrentes n'est pas une étape triviale. En effet, nous avons vu que dans des situations de concurrence partielle (Fig. 1.5), il peut arriver que des opérations concurrentes à op soient mêlées à des opérations qui précèdent causalement op. Le problème est alors de réussir à séparer ces opérations pour pouvoir appliquer correctement les transpositions en avant.

2.4 Algorithme SOCT2

Dans l'algorithme SOCT2[1] [SCF97], le respect de la causalité est obtenue comme décrit plus haut grâce à l'utilisation de vecteurs d'états. Ces derniers permettent la vérification de l'ordre causal. Le respect de l'intention des utilisateurs est réalisée par l'utilisation de fonctions de transposition en avant et en arrière définies spécifiquement en fonction des opérations utilisées pour manipuler les objets. La convergence des copies est quand à elle garantie par la conformité des fonctions de transformations à l'égard des conditions C1 et C2.

À chacune des étapes citées précédemment correspond dans SOCT2 une procédure. Elles sont appelées `Exécution_Locale`, `Livraison_Causale` et `Intégration`. La mise en œuvre de la procédure `Intégration` nécessite la maintenance d'une histoire permettant de réaliser les transpositions. Dans SOCT2, la définition formelle d'une histoire est la suivante :

Définition 2.1 (Histoire d'un objet dans SOCT2)

L'histoire d'un objet O sur un site S, notée $H_{S,O}(n)$, est une séquence de n opérations exécutées sur la copie de O sur le site S, respectant l'ordre causal, et qui transforme l'objet O depuis son état initial vers l'état courant.

Dans la suite, pour simplifier la présentation et sans perte de généralité, on considère l'existence d'un seul objet. En conséquence, $H_{S,O}(n)$ est désignée par $H_S(n)$, avec $H_S(n) = op_1.op_2.\ldots.op_i.\ldots.op_n$. On appelle *ordre de sérialisation* sur le site S, l'ordre des opérations dans $H_S(n)$. Lors de l'intégration d'une opération op, on peut-être amené à séparer les opérations concurrentes à op de celles qui précèdent causalement op et modifier ainsi cet ordre. Si l'ordre de sérialisation correspond exactement à l'ordre d'exécution des opérations sur le site, l'histoire est alors qualifiée d'histoire *réelle*.

2.4.1 Exécution locale

La procédure `Exécution_Locale` est exécutée consécutivement à la génération d'une opération sur le site. Elle réalise l'exécution de l'opération locale (pour satisfaire la contrainte d'exécution immédiate) et sa diffusion à tous les sites, y compris au site générateur lui-même.

L'exécution d'une opération op, locale au site S, a aussi pour effet d'ajouter l'opération op à l'histoire du site S (Fig. 2.1). On a donc : $H_S(n+1) = H_S(n).op_{n+1} = H_S(n).op$. On remarquera que dans ce cas, toutes les opérations de l'histoire $(\forall op_i \in H_S(n))$ précèdent causalement op. Le message diffusé pour une opération est représenté par un triplet $\langle op, S_{op}, SV_{op}\rangle$ où op désigne l'opération, S_{op} le site générateur de op et SV_{op} le vecteur d'état associé à op.

[1] Sérialisation d'Opérations Concurrentes par Transposition

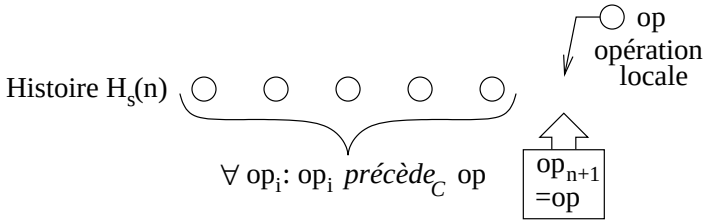


FIG. 2.1 – Exécution d'une opération locale

2.4.2 Livraison causale

La procédure `Livraison_Causale` est exécutée lors de la réception par le site S d'une opération *op*, locale ou distante, diffusée par le site générateur, S_{op}. C'est elle qui assure la livraison des opérations, dans un ordre respectant la précédence causale, à la procédure `Intégration`. Elle correspond à l'algorithme suivant :

```
procédure Livraison_Causale (⟨op, S_op, SV_op⟩) ;
début
  attendre jusqu'à SV_S[i] ≥ SV_op[i], (∀i : 1 ≤ i ≤ N) ;
  livrer (⟨op, S_op, SV_op⟩) ;
  SV_S[S_op] := SV_S[S_op] + 1 ;
fin
```

Cette procédure garantit que si op_1 *précède*$_C$ op_2, alors op_1 sera délivrée avant op_2. Sa preuve est donnée dans [SCF97]. La procédure `livrer` réalise la livraison de l'opération à la procédure `Intégration`. On notera qu'il est possible que les opérations concurrentes soient délivrées dans des ordres différents sur des sites distincts.

2.4.3 Intégration

La procédure `Intégration` est exécutée à la livraison d'une opération locale ou distante. Elle réalise l'exécution locale de l'opération distante (s'il s'agit d'une opération locale, celle-ci a déjà été exécutée). Pour prendre en compte les problèmes évoqués précédemment (respect de l'intention et convergence des copies), l'opération doit subir des transformations avant de pouvoir être exécutée sur l'état courant de l'objet.

Dans SOCT2, l'intégration consiste donc à obtenir, en se servant de l'histoire de l'objet sur le site S, l'opération qui est définie sur l'état courant de l'objet et qui réalise la même intention que l'opération générée sur le site distant.

La problématique de l'intégration d'une opération distante *op* est illustrée sur la figure 2.2-a. La livraison causale assure que, lorsque *op* est délivrée, $H_S(n)$ contient *toutes* les opérations qui précèdent causalement *op*. Mais $H_S(n)$ contient également des opérations concurrentes à *op*. Il s'agit donc d'obtenir à partir de l'opération reçue *op* et de $H_S(n)$ une opération op_{n+1}, exécutable sur l'état courant de l'objet et réalisant la même intention

que *op*, sachant que parmi les opérations de $H_S(n)$, certaines précèdent causalement *op* et d'autres sont concurrentes à *op*.

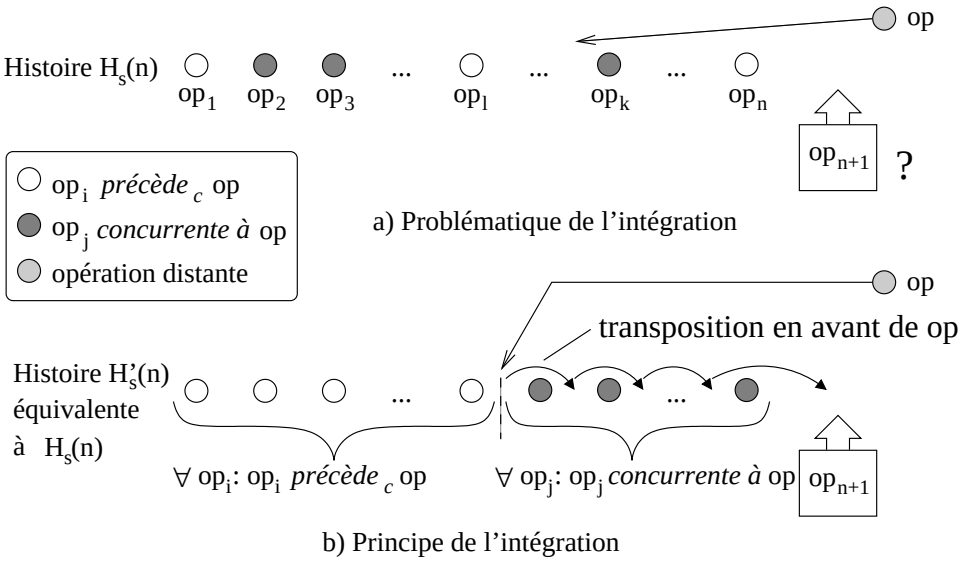


FIG. 2.2 – Intégration d'une opération distante dans SOCT2

Dans son principe, l'intégration se décompose en deux étapes (Fig. 2.2-b) :

Étape 1) Elle consiste à réordonner l'histoire $H_S(n)$, en une histoire équivalente, $H'_S(n)$, dans laquelle toutes les opérations qui précèdent causalement *op*, précèdent les opérations concurrentes à *op*. L'opération à intégrer étant définie sur l'état qui marque la frontière entre les opérations qui la précèdent et celles qui lui sont concurrentes, le problème de la concurrence partielle se trouve donc résolu. La réorganisation de l'histoire en deux sous-histoires (la sous-histoire des opérations qui précèdent causalement et la sous-histoire des opérations concurrentes) est obtenue grâce à la transposition en arrière.

Étape 2) Elle consiste à transformer l'opération *op* pour tenir compte des opérations qui lui sont concurrentes. Ceci est réalisé en transposant en avant l'opération *op* successivement par rapport aux opérations concurrentes. C'est l'opération transposée, notée op_{n+1} qui est insérée dans l'histoire $H_S(n)$ et on obtient alors $H_S(n+1) = H_S(n).op_{n+1}$.

Comme les opérations concurrentes peuvent être délivrées dans des ordres différents sur les sites, chaque site construit un ordre de sérialisation qui lui est propre et qui est déterminé par l'ordre de livraison des opérations, tout en garantissant que les ordres de sérialisation des différents sites sont équivalents.

La nécessité de satisfaire à la condition C2 est la principale contrainte imposée par l'algorithme SOCT2. Nous verrons au chapitre suivant que cette condition est un obstacle

lors de la définition des fonctions de transposition en avant et en arrière. Nous verrons aussi que l'utilisation de la seule transposition en arrière reste néanmoins une contrainte même si elle est moins contraignante que la satisfaction de la condition C2.

2.5 Algorithme GOT

Le respect de la causalité dans GOT[SJZY97], comme dans SOCT2, est obtenu en utilisant des vecteurs d'états. Le respect de l'intention repose sur l'utilisation de deux types de fonctions de transformation. La première fonction appelée `Inclusion-transformation` est similaire à la transposée en avant ; l'autre appelée `Exclusion-transformation` correspond à l'inverse de la transposée en avant. La conformité de ces fonctions de transformations à l'égard des conditions C1 et C2 n'est pas requise ; elle est remplacée par l'adoption d'un ordre de sérialisation unique. La conséquence est qu'il sera parfois nécessaire de défaire puis de refaire certaines opérations.

L'histoire utilisée dans GOT est similaire à celle utilisée dans SOCT2 à la différence près que les opérations y sont ordonnées suivant un ordre de sérialisation unique, $précède_T$. Le passage d'un ordre partiel (ici l'ordre causal) à un ordre total (ici l'ordre de sérialisation unique) est obtenue en définissant un ordre total sur les sites et en ordonnant, conformément à cet ordre, les opérations incomparables dans l'ordre causal. Ainsi, l'ordre $précède_T$ est déterminé par la somme des composantes du vecteur d'état et, en cas d'égalité de cette somme, par une priorité définie entre les sites. Plus précisément, étant données deux opérations $\langle op_1, S_1, SV_1\rangle$ et $\langle op_2, S_2, SV_2\rangle$, op_1 $précède_T$ op_2 ssi (somme(SV_1) < somme(SV_2)) ou (somme(SV_1) = somme(SV_2) et $S_1 < S_2$). Étant basé sur l'ordre causal, l'ordre de sérialisation unique est compatible avec celui-ci. Ainsi, quelles que soient les opérations op_i et op_j, op_i $précède_C$ $op_j \Rightarrow op_i$ $précède_T$ op_j. On notera toutefois que les estampilles ainsi définies ne sont pas continues, c.-à-d. qu'en considérant deux estampilles il n'est pas toujours possible de déterminer si elles sont strictement consécutives ou si il existe des opérations qui les séparent dans l'ordre $précède_T$.

2.5.1 Exécution locale

L'étape d'exécution locale comporte l'exécution de l'opération (pour satisfaire la contrainte d'exécution immédiate) et sa diffusion à tous les sites, y compris au site générateur lui-même.

L'exécution d'une opération op, locale au site S, a pour effet d'ajouter l'opération op à l'histoire du site S (Fig. 2.3). On a donc : $H_S(n+1) = H_S(n).op_{n+1} = H_S(n).op$.

Le message diffusé pour une opération est un triplet $\langle op, S_{op}, SV_{op}\rangle$[2] où op désigne l'opération, S_{op} le site générateur de op et SV_{op} le vecteur d'état associé à op.

[2]Dans [SJZY97] il est dit que l'opération est diffusée accompagnée de son estampille. Il semble toutefois indispensable qu'elle soit également accompagnée du numéro du site générateur puisque celui-ci est nécessaire à la vérification de l'ordre $précède_T$ et qu'il ne peut être déduit du vecteur d'état.

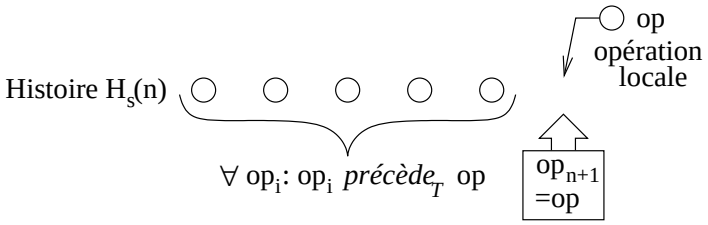


FIG. 2.3 – Exécution d'une opération locale

2.5.2 Livraison causale

L'utilisation des vecteurs d'états pour garantir l'intégration suivant l'ordre causal est similaire à celle qui en est faite dans SOCT2. L'ordre total $précède_T$ ne peut pas être utilisé pour assurer la livraison causale. En effet, cet ordre bien qu'étant compatible avec l'ordre causal, comme il n'est pas continu, ne permet pas de savoir lorsqu'un site reçoit une opération distante s'il existe une opération qui la précède causalement et qui n'a pas encore été reçue par le site (cf. 1.2).

Pour deux opérations op_i et op_j, si op_i $précède_C$ op_j, la livraison causale assure que op_i sera intégrée avant op_j. Au contraire, si op_i est concurrente à op_j alors il peut arriver que op_j soit livrée et intégrée avant op_i même si op_i $précède_T$ op_j.

2.5.3 Intégration

Pour illustrer l'étape d'intégration de GOT, on utilisera l'exemple donné en figure 2.4. On procède à l'intégration d'une opération distante op_4. Sur le site où se déroule l'intégration, ont déjà été exécutées les opérations op_1, op_2, $op_3^{op_2}$, op_5 et op_6. Les opérations op_1 et $op_3^{op_2}$ sont des opérations dont dépend causalement op_4. Les indices des opérations sont attribués en accord avec l'ordre $précède_T$, cela veut dire que op_i $précède_T$ op_j si et seulement si $i < j$.

L'intégration de op_4 se déroule de la manière suivante. Comme les fonctions de transpositions utilisées dans GOT ne satisfont pas nécessairement à la condition C1 et que la livraison des opérations ne suit pas toujours l'ordre $précède_T$, il arrive que des opérations soient intégrées dans l'histoire avant des opérations qui les précèdent dans cet ordre. C'est le cas ici des opérations op_5 et op_6 qui sont concurrentes à op_4 et qui ont été intégrées avant que op_4 qui les précède dans l'ordre $précède_T$ ne l'ait été. De ce fait, il est nécessaire, avant de pouvoir intégrer op_4, de défaire op_5 et op_6 (fig. 2.4-1) qui devront une fois op_4 intégrée être réintégrées (refaites) à leur tour.

Une fois ces deux opérations défaites (fig. 2.4-2), il n'est toujours pas possible de transposer en avant op_4 par rapport à l'opération concurrente op_2. En effet, comme `Transpose_av`, la fonction de transposition en avant utilisée dans GOT, `Inclusion-transformation`, ne peut s'appliquer qu'a deux opérations définie sur le même état. Or op_4 et op_2 ne sont pas définies sur le même état puisque op_4 tient compte des effets des opérations op_1 et op_3 qui la précèdent causalement. Ce n'est pas le cas de op_2. Contraire-

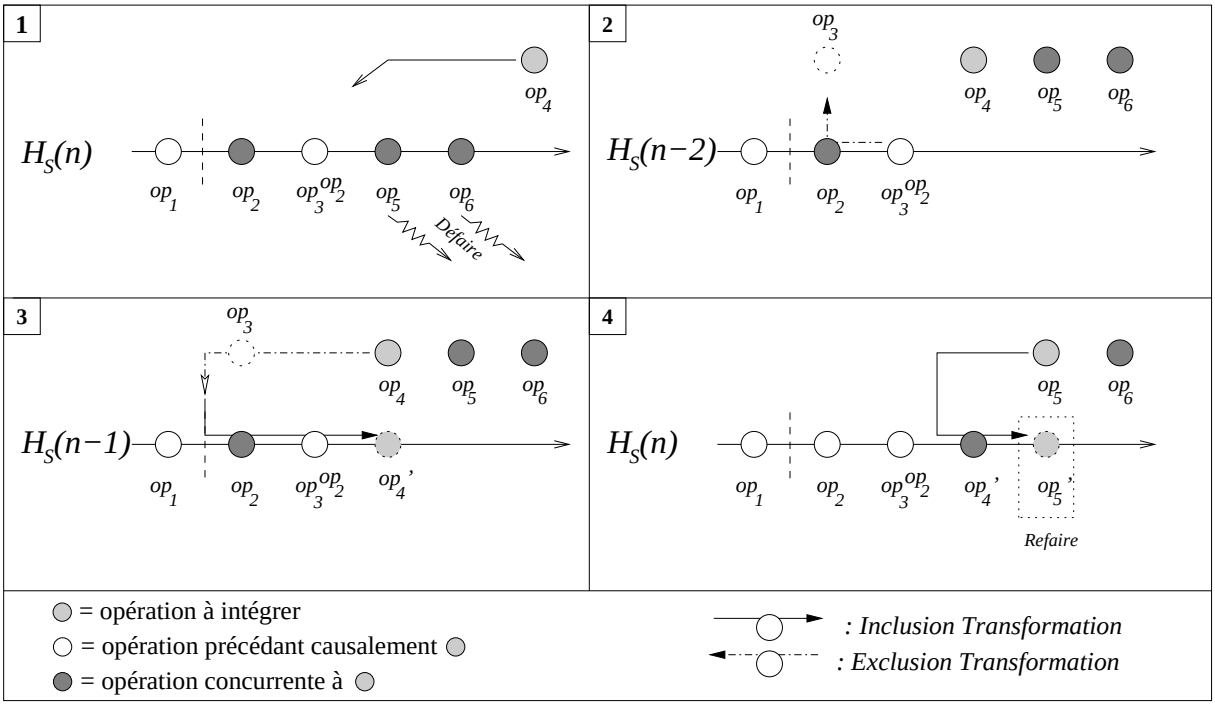


FIG. 2.4 – Intégration d'une opération distante dans GOT

ment à ce qui est fait dans SOCT2 où on cherche justement à faire en sorte que op_2 tienne compte des effets de op_3, dans GOT c'est op_4 qui va être transformée pour ne plus tenir compte des effets de op_3.

Comme on ne dispose pas de l'opération op_3 sur le site où a lieu l'intégration, il va falloir la recalculer à partir de l'histoire du site. Intervient alors la deuxième fonction, l'`Exclusion-transformation`, équivalente à l'inverse de la transposée en avant qui va permettre à partir de $op_3^{op_2}$ et op_2 d'obtenir op_3 (fig. 2.4-2). Ensuite, l'utilisation conjointe de l'`Exclusion-transformation` et de l'`Inclusion-transformation` permet d'obtenir à partir de op_3 et op_4 une opération op_4'' qui ne tienne plus compte des effets de op_3. Puis en utilisant l'`Inclusion-transformation` de op_4 par rapport à op_2 puis par rapport à $op_3^{op_2}$, on obtient l'opération op_4' qui réalise la même intention que op_4 (fig. 2.4-3). C'est cette opération qui est insérée dans l'histoire du site et qui est exécutée sur la copie locale de l'objet.

Une fois l'intégration de op_4 réalisée, il reste à réintégrer les opérations qui avaient été défaites. Cette intégration se déroule de façon similaire à l'intégration de op_4. Ainsi l'intégration de op_5 est relativement immédiate ; elle consiste à la transposer en avant par rapport à op_4 –au moyen de l'`Inclusion-transformation`– (fig. 2.4-4), puisqu'aucune opération dont elle dépend causalement n'est mêlée aux opérations avec lesquelles elle est concurrente (ici par exemple op_4). Il n'en est pas de même pour l'intégration op_6. En effet, lors de son intégration, op_4 avec laquelle elle est concurrente est mêlée à op_5' dont elle dépend causalement. Il faut donc en utilisant l'`Exclusion-transformation` obtenir l'opération op_6'' qui ne tient plus compte des effets de op_5. Il faut ensuite transposer en

avant op''_6 par rapport à op'_4 puis par rapport à op'_5 pour obtenir l'opération op'_6 qui réalise la même intention que op_6.

On notera que la nécessité de défaire puis de refaire certaines opérations constitue également une contrainte importante de l'algorithme GOT. Cela impose de procéder, pour les opérations défaites, à une nouvelle intégration.

En outre, les conditions d'utilisations de l'`Exclusion-transformation`, notamment sur des opérations liées par une dépendance causale pose problème dans la mesure où l'`Exclusion-transformation` considère que les deux opérations mises en jeu sont concurrentes.

Pour toutes ces raisons, un nouvel algorithme, fortement inspiré de SOCT2 [SCF97] et utilisant des fonctions de transpositions en avant et en arrière qui satisfont aux conditions C1 et C2, a été proposé ultérieurement sous le nom de GOTO [SE98] par le même auteur.

2.6 Algorithme adOPTed

L'algorithme adOPTed [RNRG96] exploite une histoire multidimensionnelle. Le respect de la causalité est obtenu comme pour les algorithmes précédents grâce à l'utilisation de vecteurs d'états. Le respect de l'intention ne fait appel qu'à la seule fonction de transposition en avant qui doit satisfaire aux conditions C1 et C2. Elle est nommée `L-transformation` dans adOPTed.

L'histoire multidimensionnelle utilisée par l'algorithme adOPTed est représentée par un graphe. Ce graphe est constitué d'un ensemble de sommets N et d'un ensemble d'arcs A. Les sommets représentent les différents états pris par l'objet dans le système ; les arcs, les opérations dont l'exécution a provoqué la transition d'un état à un autre. Dans ce graphe, on conserve, au contraire de ce qui est fait couramment dans les histoires linéaires, les opérations sous leur forme originales. On peut donc toujours retrouver l'état (le sommet) sur lequel est définie une opération. Lors de l'intégration d'une opération, le chemin (dans le graphe) qui mène de l'état où elle a été définie jusqu'à l'état courant représente la séquence d'opérations concurrentes par rapport à laquelle elle doit être transposée pour obtenir l'opération qui sera exécutée sur l'état courant de l'objet.

Par commodité, on considère la valeur du vecteur d'état comme une coordonnée dans un repère orthonormé. Chacune des dimensions de ce repère est alors associée à un utilisateur et toutes les opérations qu'il a générées sont représentées par des vecteurs représentants du vecteur unité de cette dimension. Chacun des sites où est implantée une copie de l'objet, gère un graphe représentant l'histoire multidimensionnelle des opérations locales et distantes exécutées sur le site.

2.6.1 Exécution locale

L'exécution locale d'une opération est immédiate après sa génération et elle est suivie par la diffusion à tous les sites, du message contenant l'opération et son vecteur d'état. Lors de l'exécution d'une opération, celle-ci est ajoutée au graphe représentant l'histoire du site. Pour cela, un nouvel arc est créé ayant pour origine le sommet représentant l'état

courant et pour vecteur directeur le vecteur unité associé à l'utilisateur ayant généré l'opération. Comme dans les autres algorithmes, le message diffusé pour une opération *op* générée sur un site S_{op} est un triplet $\langle op, S_{op}, SV_{op} \rangle$ où SV_{op} est le vecteur d'état associé à *op*.

2.6.2 Livraison causale

L'utilisation des vecteurs d'états pour garantir l'intégration suivant l'ordre causal est similaire à celle qui en est faite dans SOCT2. Le même type de procédure permet de garantir la livraison et l'intégration des opérations suivant l'ordre causal.

2.6.3 Intégration

On rappelle que l'intégration d'une opération *op* consiste à trouver une opération *op*′, définie sur l'état courant de l'objet, qui réalise la même intention que *op*. La transposition en avant d'une opération op_i par rapport à une opération concurrente op_j peut être modélisée dans adOPTed comme la translation du vecteur représentant op_i par le vecteur représentant l'opération op_j.

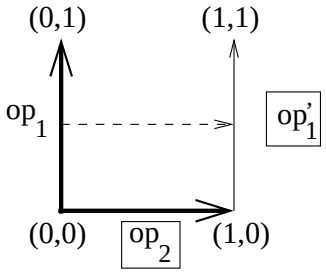


FIG. 2.5 – Graphe des exécutions pour l'utilisateur 2

Considérons un système dans lequel deux utilisateurs ont généré en concurrence une opération chacun, op_1 pour l'utilisateur 1 et op_2 pour l'utilisateur 2. La figure 2.5 représente l'état du graphe correspondant à l'histoire des exécutions associée au site de l'utilisateur 2. En gras sont représentés les vecteurs correspondants aux opérations générées, en pointillés les translations représentant les transpositions en avant, quant aux opérations réellement exécutées sur le site de la copie, elles sont encadrées. Ainsi, ici l'opération locale op_2 a été exécutée sur l'état `(0,0)`. Puis l'opération concurrente op_1 (générée sur le même état que op_2) est reçue. Elle est transposée en avant par rapport à op_2 pour obtenir l'opération op'_1 qui, exécutée sur l'état `(1,0)`, réalisera la même intention que op_1. À chaque copie d'objet est associée une histoire multidimensionnelle représentée par un graphe.

Dans adOPTed la fonction `translateRequest`, équivalente de la procédure `Intégrer` de SOCT2, délivre, pour une opération *op* et un état *O* donnés, l'opération *op*′ définie sur *O* et dont l'exécution réalise la même intention que celle que réalise *op* lorsqu'elle est exécutée à partir de l'état sur lequel elle est définie. L'appel de `translateRequest` avec pour paramètres une opération et l'état courant, réalise donc l'intégration de celle-ci. On notera que dans adOPTed la fonction `translateRequest` délivre une nouvelle opération

dans l'histoire en lui associant, à la différence de ce qui a été fait dans les algorithmes précédents à histoire dite linéaire, le vecteur d'état correspondant à l'état sur lequel cette opération est définie. En outre, certaines opérations, résultats intermédiaires des calculs effectués par la fonction `translateRequest` sont également placées dans l'histoire. Cette opération que délivre `translateRequest` est finalement exécutée.

Dans le cas de l'exemple illustré par la figure 2.5, lorsque l'opération op_1 est reçue, l'appel de `translateRequest`(op_1,`(1,0)`) délivre l'opération $op'_1 = op_1^{op_2}$, qui est exécutée sur l'objet à partir de l'état `(1,0)`.

Définition

Un état O_j est dit *atteignable* à partir d'un état O_i *par l' opération op* si les 3 points suivants sont vérifiés.

1. O_i est un état sur lequel toutes les opérations qui ont été générées *avant op* sur S_{op} ont déjà été exécutées et sur lequel n'a été exécutée aucune opération ayant été générée *après op* sur S_{op} ;
2. O_i est un état sur lequel les opérations qui ont été générées sur un autre site que S_{op} mais qui ont été exécutées *avant op* sur S_{op}, ont toutes été exécutées ;
3. Le vecteur d'état de O_j est égal à celui de O_i où la $(S_{op})^{\text{e}}$composante a été incrémenté de 1.

Ces 3 points peuvent être exprimés plus formellement de la façon suivante :

1. $SV_{O_i}[S_{op}] = SV_{op}[S_{op}]$ et,
2. $\forall\ i \neq S_{op},\ SV_{O_i}[i] \geq SV_{op}[i]$ et,
3. $SV_{O_j}[S_{op}] = SV_{O_i}[S_{op}] + 1$ et $\forall\ k$ t.q. $k \neq S_{op},\ SV_{O_j}[k] = SV_{O_i}[k]$

Remarque Les deux premiers points expriment en fait que O_i est un état sur lequel *op* peut-être exécutée, c.-à-d. un état pour lequel *op* est causalement prête.

Un état O_j est dit *atteignable par transitivité* à partir d'un état O_i si :
- $O_j = O_i$ ou
- s'il existe une opération op_i telle que O_{i+1} soit un état *atteignable par l'opération* op_i et O_j est *atteignable* à partir de O_{i+1}.

États atteignables et états atteignables par transitivité. Prenons pour illustrer ces définitions l'exemple de la figure 2.6. Sur ce graphe, on peut voir que les opérations op_1 et op_3 ont été exécutées sur le site lorsque l'opération distante op_2 est reçue. Cette dernière opération est concurrente aux précédentes. On voit en outre que l'état `(1,2)` est *atteignable* à partir de l'état `(1,1)` *par l'opération* op_3. De plus ce même état `(1,2)` est *atteignable par transitivité* à partir de l'état `(1,0)` car l'état `(1,1)` est *atteignable* depuis l'état `(1,0)` *par l'opération* op_1. On notera donc que dire qu'un état O_j est *atteignable* à partir d'un état O_i *par une opération op* ne signifie pas que cette dernière est définie sur l'état O_i, ni que son exécution donne pour résultat l'état O_j. Cela traduit le fait que l'exécution à partir de l'état O_i de l'opération réalisant la même intention que l'opération

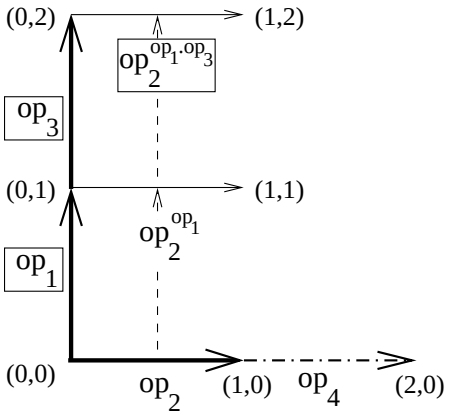


FIG. 2.6 – Exemple de graphe d'exécution dans adOPTed

op donnera pour résultat l'état O_j.

La transposition en avant est invoquée de manière indirecte par l'algorithme adOPTed au travers d'un appel de la fonction `tf`(op_1,op_2)=(op'_1,op'_2). Cette fonction délivre pour un couple d'opérations concurrentes (op_1,op_2) définies sur le même état, le couple d'opérations (op'_1,op'_2) où op'_1 = `Transpose_av`(op_2,op_1) et op'_2 = `Transpose_av`(op_2,op_1). Pour des raisons de clarté, nous avons remplacé l'appel à `tf` par les deux appels de `Transpose_av` correspondants dans l'algorithme associé à la fonction `translateRequest`(op,O_j)=op' qui suit :

Soit *op* l'opération définie sur un état O_{op},
O_j l'état sur lequel doit être exécutée l'opération *op*.
Si $SV_{op} = SV_{O_j}$ [3]
Alors *op* est définie sur O_j, *op* est insérée dans l'histoire et $op' = op$.
Sinon, choisir un état O_i dans l'histoire tel que :
- O_j est *atteignable* à partir de O_i *par l'opération* op_i et
- O_i est *atteignable* à partir de O_{op},

op_a = `translateRequest`(op, O_i)
op_b = `translateRequest`(op_i, O_i)
op'_a = `Transpose_av`(op_b, op_a)
op'_b = `Transpose_av`(op_a, op_b)
op'_a et op'_b sont placées dans l'histoire et $op' = op'_a$.

Utilisation de la fonction `translateRequest` Dans l'exemple de la figure 2.6, l'état courant est `(1,2)`. Cet état a été obtenu en intégrant l'opération op_2. Cette intégration est obtenue en transposant en avant op_2 par rapport à op_1 puis op_3 ; ce qui a donné l'opération $op_2^{op_1.op_3}$. Supposons que l'on doive maintenant intégrer l'opération op_4 définie sur l'état

[3] Le vecteur d'état de l'opération *op* est égal à celui de l'état O_n

(1,0). L'intégration de l'opération op_4 se traduit par l'appel de la fonction `translateRequest` avec comme paramètres l'opération op_4 et l'état courant (1,2). L'exécution sera donc la suivante :

- $SV_{op_4} \neq SV_{(1,2)}$
- (1,1) est un état tel que (1,2) est *atteignable* à partir de (1,1) *par l'opération* op_3 et (1,1) est *atteignable* à partir de (1,0). Donc `translateRequest`(op_4, (1,2)) délivre l'opération obtenue par `Transpose_av`(`translateRequest`(op_3, (1,1)), `translateRequest`(op_4, (1,1))). Il s'agit donc d'obtenir :
  - `translateRequest`(op_3, (1,1)) ; il apparait que :
    - $SV_{op_3} \neq SV_{(1,1)}$,
    - (0,1) est un état tel que (1,1) est *atteignable* à partir de (0,1) *par l'opération* op_2 et (1,1) est *atteignable* à partir de (0,1)[4]. Donc `translateRequest`(op_3, (1,1)) délivre l'opération obtenue par `Transpose_av`(`translateRequest`(op_3, (0,1)), `translateRequest`(op_2, (0,1))).

En poursuivant les appels récursifs, on obtient l'exécution présentée à la figure 2.7. Sous les appels à la fonction `translateRequest` sont placées les valeurs de retour de celles-ci. La jonction de deux appels à la fonction `translateRequest` matérialise l'utilisation de la fonction de transposition en avant. Le résultat de l'appel à `translateRequest` placé au dessus est transposé en avant par rapport au résultat de l'appel placé au dessous. Ainsi, `translateRequest`(op_3,(1,1)) fournit une opération qui exécutée à partir de l'état (1,1) réalise la même intention que op_3 lorsqu'elle a été exécutée à partir de l'état sur lequel elle a été générée. C'est la transposée en avant de op_3 par rapport à $op_2^{op_1}$, c.-à-d. $op_3^{op_2^{op_1}}$. Finalement, l'exécution de l'opération délivrée par la fonction `translateRequest`(op_4, (1,2)) réalise l'intention de l'opération op_4 sur l'état courant (1,2), état obtenu à partir de l'exécution, dans cet ordre, des opérations op_1, op_2 et op_3.

L'algorithme adOPTed impose la vérification de la condition C2. Il est donc soumis aux mêmes contraintes que les autres algorithmes qui imposent également cette condition. Nous exposerons, au chapitre suivant, les difficultés que pose la satisfaction de la condition C2. L'autre contrainte importante d'adOPTed est la gestion et l'occupation mémoire du graphe multidimensionnel qui tient lieu d'histoire.

2.7 Algorithme SOCT3

Il apparaît que la suppression de la condition C2 repose sur la définition d'un ordre global unique, $précède_S$, compatible avec l'ordre causal $précède_C$. Par ailleurs, pour éviter de défaire et refaire des opérations, les opérations distantes doivent être reçues dans l'ordre $précède_S$. Afin de satisfaire cette double contrainte, [Sul98] a proposé d'utiliser un séquenceur pour construire un ordre global continu. Dans l'algorithme SOCT3 [VCFS00] c'est cet ordre global qui est utilisé pour intégrer les opérations. Les nouveaux algorithmes présentés au chapitre suivant sont inspirés de SOCT3, c'est pourquoi la présentation de celui-ci sera un peu plus détaillée que pour les algorithmes précédents. Nous présenterons

[4] qui est l'état sur lequel op_3 est définie

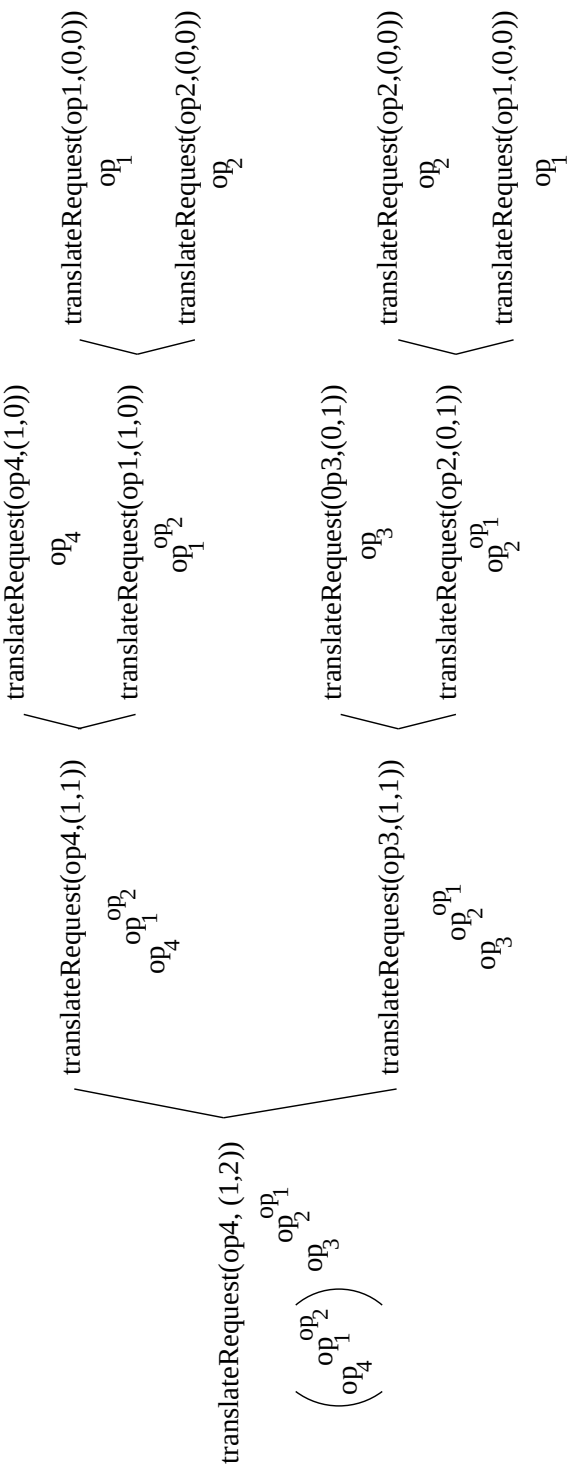


FIG. 2.7 – Déroulement de l'appel récursif dans adOPTed

en particulier les algorithmes associés aux fonction utilisées. La section 2.7.1 présente la construction de l'ordre global continu $précède_S$ et la section 2.7.2 montre son utilisation dans le processus d'intégration d'une opération.

2.7.1 Exécution locale, diffusion et livraison des opérations dans SOCT3

Un séquenceur [RK79] est un objet qui délivre des estampilles de valeur entière, croissante et continue. L'opération qui permet d'obtenir une estampille est désignée par la fonction `Ticket`. Nous ne détaillons pas les différentes techniques pour mettre en œuvre un séquenceur dans un système réparti, à savoir le séquenceur circulant ("à la demande", sur un chemin prédéfini ou un anneau virtuel) [LL78], ou le séquenceur répliqué [BKZ79].

Grâce à la fonction `Ticket` du séquenceur, une estampille est associée à chaque opération générée dans le système collaboratif. L'ordre $précède_S$ correspond à l'ordre des estampilles et nous montrons plus loin qu'il est compatible avec l'ordre causal $précède_C$.

Pour respecter la contrainte de temps de réponse (c.-à-d. l'exécution immédiate), une opération locale est exécutée avant de faire appel à l'opération `Ticket`. Plus précisément, quand une opération est générée sur un site S, elle est exécutée immédiatement et le quadruplet $\langle op, S, SV_{op}, -1\rangle$ est ajouté à l'histoire H du site. La valeur (-1) signifie que l'opération n'a pas encore reçu l'estampille délivrée par le séquenceur. La fonction `Ticket` qui est ensuite exécutée fournit l'estampille N_{op} qui est associée à l'opération op.

Le message diffusé pour une opération est maintenant un quadruplet : $\langle op, S_{op}, SV_{op}, N_{op}\rangle$ où op est l'opération générée, S_{op} le site générateur, SV_{op} le vecteur d'état associé à l'opération et N_{op} l'estampille de l'opération. La procédure de livraison n'est plus ici seulement causale ; elle assure sur chaque site une livraison séquentielle, dans l'ordre des estampilles croissantes, des opérations. Le vecteur d'état reste nécessaire, non plus pour assurer la livraison des opérations, mais pour déterminer quelles sont les opérations concurrentes à l'opération op (cf. algorithme d'intégration). Quand un site S reçoit une opération op, la procédure de livraison la met en attente tant que toutes les opérations, qui ont une estampille inférieure à la sienne, n'ont pas encore été reçues et délivrées. A cette fin, le site conserve dans N_S l'estampille de la dernière opération qui lui a été délivrée.

La procédure de livraison, appelée procédure `Livraison_Séquentielle`, ainsi que la procédure `Exécution_Locale` sont décrites ci-après.

```
procédure Exécution_Locale(op) ;
{Exécution et diffusion d'une opération générée localement}
début
  exécuter(op) ;
  {exécution sur l'état courant de l'objet}
  V_op = V_S ;
  n := n + 1 ;
  H[n] := ⟨op, S, SV_op, −1⟩ ;
  V_S[S] = V_S[S] + 1 ;
  N_op = Ticket() ;
  H[n].estampille := N_op ;
  diffuser(⟨op, S, SV_op, N_op⟩) ;
  {diffusion immédiate à tous les sites, le site S compris}
fin
```

```
procédure Livraison_Séquentielle(⟨op, S_op, SV_op, N_op⟩) ;
début
  attendre jusqu'à (N_S = N_op − 1) ;
  livrer (⟨op, S_op, V_op, N_op⟩) ;
  N_S := N_S + 1 ;
fin
```

L'ordre $précède_S$ (c.-à-d. l'ordre des estampilles) est compatible avec l'ordre causal $précède_C$ car, si op_1 $précède_C$ op_2, alors $N_{op_1} < N_{op_2}$. En effet, si op_1 $précède_C$ op_2, cela signifie que l'opération op_1 a été exécutée avant la génération de op_2. L'exécution de la fonction `Ticket` pour op_1 précède l'exécution de la fonction `Ticket` pour op_2. L'opération op_1 a donc obtenu une estampille inférieure à celle de l'opération op_2 ; ainsi op_1 sera délivrée avant op_2 sur tous les sites.

2.7.2 Intégration d'une opération dans SOCT3

Principe de la solution

L'intégration d'une opération distante suit le principe exposé dans la section 2. La spécificité de l'algorithme est liée au fait que sur chaque site on mémorise une histoire (équivalente à l'histoire réelle) notée $H_S(n)$ dans laquelle les opérations apparaissent dans l'ordre des estampilles.

Plus formellement, soit $H_S(n) = op_1.op_2.\ldots.op_i.op_{L_1}.op_{L_2}.\ldots.op_{L_m}$ l'histoire de l'objet sur le site S. Elle est telle que :

i) pour $(1 \leq j \leq i)$, op_j est une opération, locale ou distante, qui a été délivrée par la procédure de livraison séquentielle ; les opérations $op_1.op_2.\ldots.op_i$ ont donc des

estampilles continues ;

ii) pour $(1 \leq k \leq m)$, op_{L_k} est une opération locale qui a été exécutée mais n'a pas encore été délivrée par la procédure de livraison séquentielle ; les opérations $op_{L_1}.op_{L_2}.\ldots.op_{L_m}$ ont des estampilles discontinues ;

iii) $n = m + i$.

Les estampilles étant associées aux opérations en respectant leur ordre de génération, les opérations locales $(op_{L_1}, op_{L_2}, \ldots op_{L_m})$ ont des estampilles vérifiant : $i < L_1 < L_2 < ...L_m$.

Lorsque l'opération op_{i+1} d'estampille $(i+1)$ est délivrée par la procédure de livraison, il y a deux cas possibles :

1) soit op_{i+1} est l'opération locale op_{L_1} qui a déjà été exécutée : elle ne nécessite aucun traitement supplémentaire ;

2) soit op_{i+1} est une opération distante reçue en provenance du site S'. Il s'agit alors d'intégrer op_{i+1} dans $H_S(n)$, ce qui veut dire :
   - trouver et exécuter, sur l'état courant, l'opération qui réalise la même intention que op_{i+1},
   - réordonner l'histoire obtenue dans l'ordre des estampilles croissantes des opérations.

Pour trouver l'opération à exécuter qui réalise la même intention que op_{i+1}, nous suivons le principe présenté pour SOCT2 au chapitre 2.4. Les opérations de l'histoire $H_S(n)$ sont d'abord réordonnées, par transposition en arrière, en deux séquences seq_1 et seq_2 telles que :
- $H_S(n)$ est équivalente [5] par transposition avec $seq_1.seq_2$, notée $H_S(n) \equiv_T seq_1.seq_2$ où :
  - seq_1 = la séquence des opérations qui précèdent causalement op_{i+1} et
  - seq_2 = la séquence des opérations concurrentes à op_{i+1}.

Ainsi, l'opération à exécuter pour réaliser l'intention de op_{i+1} est $op_{i+1}^{seq_2}$, c'est à dire la transposée en avant de op_{i+1} par rapport à seq_2. On a donc :

$$H_S(n+1) \equiv_T H_S(n).op_{i+1}^{seq_2} = op_1.op_2.\ldots.op_i.op_{L_1}.op_{L_2}.\ldots.op_{L_m}.op_{i+1}^{seq_2}$$

Pour réordonner l'histoire $H_S(n+1)$ dans l'ordre des estampilles croissantes des opérations, on doit amener, dans $H_S(n+1)$, $op_{i+1}^{seq_2}$ à la place correspondant à son estampille $(i+1)$. Pour cela, $op_{i+1}^{seq_2}$ est transposée en arrière par rapport à la séquence des opérations locales[6] $op_{L_1}.op_{L_2}.\ldots.op_{L_m}$, ce qui donne l'histoire suivante (dans laquelle les opérations sont ordonnées suivant leurs estampilles) : $H_S(n+1) = op_1.op_2.\ldots.op_i.op'_{i+1}.op'_{L_1}.op'_{L_2}.\ldots.op'_{L_m}$.

Les différentes étapes de l'intégration d'une opération distante sont représentées sur la figure 2.8. L'histoire $H_S(n+1)$, ainsi obtenue, est identique à celle qu'on obtiendrait

[5] On dit que deux histoires H_1 et H_2 sont équivalentes par transposition [Sul98], noté $H_1 \equiv_T H_2$, si H_2 peut être obtenue à partir de H_1, en transposant en arrière certaines opérations.

[6] Ce qui consiste, d'après [Sul98], à transposer $op_{i+1}^{seq_2}$ en arrière avec op_{L_m}, puis à transposer en arrière la transposée résultante obtenue en arrière avec $op_{L_{m-1}}$ et ainsi de suite.

en intégrant les opérations dans l'ordre des estampilles croissantes. La preuve est présentée dans [Sul98] ainsi que le détail des opérations $op'_{L_1}.op'_{L_2}.\ldots.op'_{L_m}$, obtenues par transposition en arrière.

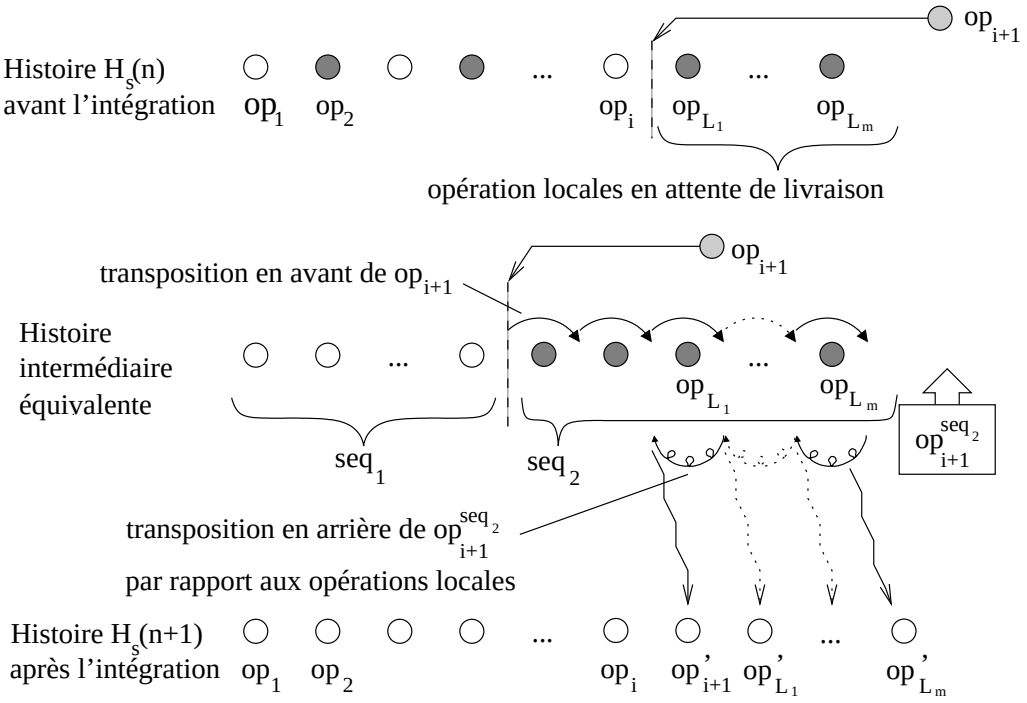


FIG. 2.8 – Intégration d'une opération distante dans SOCT3

Algorithme d'intégration

La procédure `Intégration` est exécutée lorsqu'une opération est délivrée par la procédure de livraison séquentielle définie au paragraphe précédant. Cette procédure intègre une opération dans l'histoire suivant son estampille. Elle utilise pour cela les deux procédures suivantes (directement issues de l'algorithme SOCT2) :

- la procédure `Transpose_Arrière` qui redéfinit la fonction `Transpose_ar` sous forme d'une procédure réalisant la transposition en arrière de deux opérations de l'histoire $H_S(n)$;
- la fonction `Séparer` qui réordonne l'histoire $H_S(n)$ en deux séquences seq_1 et seq_2, et retourne la longueur n_1 de seq_1. La fonction `Séparer` est appliquée sur une copie de l'histoire $H_S(n)$ pour ne pas changer l'ordre des opérations dans l'histoire $H_S(n)$, dans la mesure où elles doivent rester ordonnées suivant leurs estampilles.

Les notations sont les mêmes que précédemment.

```
procédure Transpose_Arrière(j) ;
{Transpose en arrière la j^e opération et la (j − 1)^e opération}
début
  ⟨op_j, S_{op_j}, SV_{op_j}, N_{op_j}⟩ := H[j] ;
  ⟨op_{j−1}, S_{op_{j−1}}, SV_{op_{j−1}}, N_{op_{j−1}}⟩ := H[j − 1]
  (op_j , op_{j−1}) = Transpose_ar(op_{j−1} , op_j) ;
  H[j] := ⟨op_{j−1}, S_{op_{j−1}}, SV_{op_{j−1}}, N_{op_{j−1}}⟩ ;
  H[j − 1] := ⟨op_j, S_{op_j}, SV_{op_j}, N_{op_j}⟩ ;
fin
```

```
fonction Séparer(H , ⟨op, S_op, SV_op, N_op⟩) : entier ;
{Réordonne l'histoire H et retourne n_1 tel que :
 pour 1 ≤ i ≤ n_1 , H[i] précède_C ⟨op, S_op, SV_op, N_op⟩ et
 pour n_1 < i ≤ n , H[i] concurrente à ⟨op, S_op, SV_op, N_op⟩}
début
  n_1 := 0 ;
  pour i := 1 haut n faire
    ⟨op_i, S_{op_i}, SV_{op_i}, N_{op_i}⟩ := H[i] ;
    si SV_{op_i}[S_{op_i}] < SV_op[S_{op_i}] alors
    {op_i précède_C op}
      pour j := i bas n_1 + 2 faire
      {Transposer op_i en arrière jusqu'à seq_1}
        Transpose_Arrière(j) ;
      finpour ;
      n_1 := n_1 + 1 ;
    finsi ;
  finpour
fin
```

```
procédure Intégration(⟨op, S_op, SV_op, N_op⟩) ;
début
  si S ≠ S_op alors
  {op est une opération distante}
    H' := H ;
    {copier l'histoire H dans H'}
    n_1 := Séparer(H', ⟨op, S_op, SV_op, N_op⟩) ;
    pour i := n_1 + 1 haut n faire
    {transposer op en avant par rapport à seq_2}
      op := Transpose_av(H'[i].opération, op) ;
    finpour ;
    exécuter (op) ;
    SV_S[S_op] := SV_S[S_op] + 1 ;
    H[n + 1] := ⟨op, S_op, SV_op, N_op⟩ ;
    n := n+1 ;
    pour j := n bas N_op + 1 faire
    {transposer op en arrière dans H jusqu'à l'élément N_op}
      Transpose_Arrière(j) ;
    finpour ;
  finsi ;
fin
```

Pour des raisons de clarté, les algorithmes présentés n'incluent pas la gestion du parallélisme potentiel entre ***Exécution_Locale*** *et* ***Intégration****.*

L'algorithme SOCT3 n'impose pas la condition C2. Toutefois, nous montrons (cf. Annexe A) qu'il existe des cas où la définition des fonctions de transpositions en arrière qui respectent la cohérence des copies n'est possible que si la condition C2 est satisfaite. Il paraît alors intéressant d'obtenir des algorithmes ne nécessitant pas l'utilisation de la transposée en arrière. Cela représente le double avantage de ne pas avoir à définir les fonctions de transpositions en arrière et de ne pas avoir à vérifier que celles-ci satisfont à la condition C2.

3

Nouveaux Algorithmes

Sommaire

3.1 Limites des algorithmes existants

3.1.1 Contraintes liées à la condition C2

La condition C2 (définie p. 17) permet de garantir que la transposition en avant d'une opération par rapport à une séquence d'opérations concurrentes ne dépend pas de l'ordre dans lequel les opérations de la séquence sont elles-mêmes transposées en avant. La satisfaction de cette condition pose une contrainte importante. En effet, elle ne peut pas toujours être vérifiée à moins que les opérations soient modifiées en conséquence ou que des paramètres supplémentaires leur soient ajoutés comme nous l'avons vu pour la définition de la transposition en avant relative aux opérations de l'ensemble {`insérer`,`effacer`} (cf. exemple 1.5).

D'une façon générale, la vérification de la condition C2 pose un problème dans la mesure où elle fait intervenir un triplet d'opérations, ce qui génère un très grand nombre de cas à prendre en compte. Dans l'exemple illustré figure 1.6, les opérations considérées sont op_1 = `insérer`(3,'f'), op_2 = `effacer`(3) et op_3 = `insérer`(4,'f'). Dans ce cas, vérifier C2 revient à prouver que quel que soit l'ordre dans lequel les opérations op_1 et op_2 auront été transposées en avant, la transposée en avant de op_3 par rapport à la séquence qu'elles formeront alors est unique. La vérification concerne donc le triplet {`insérer`,`effacer`,`insérer`}, les deux premières opérations représentant les opérations op_1 et op_2 et la dernière représentant l'opération op_3. La vérification de la satisfaction de la condition C2 pour cet unique triplet comporte à lui seul 19 cas. Si on tient compte de tout les triplets possibles (cf. [Sul98]), on obtient un total de 129 cas, pour, rappelons-le, le simple ensemble d'opérations {`insérer`,`effacer`}. En outre, il peut exister des cas où elle ne peut tout simplement pas être satisfaite ce qui remet alors en question la convergence des copies.

Partant de ce constat, il apparaît que l'emploi d'algorithmes nécessitant la vérification de la condition C2 n'est pas envisageable dans le cadre d'applications faisant intervenir des ensembles d'opérations plus conséquents. C'est pourquoi il nous est apparu nécessaire de proposer des algorithmes qui n'imposent pas la vérification de la condition C2. Se passer de la condition C2 suppose de pouvoir ordonner les opérations concurrentes de la même façon sur tous les sites. Cela est possible grâce à la définition d'un ordre global de sérialisation compatible avec l'ordre causal. Nous le nommons $précède_S$.

3.1.2 Nécessité de Défaire/Refaire des opérations

L'utilisation d'un ordre global de sérialisation n'est pas exempt de contraintes. En effet, le problème est que l'ordre global de livraison (respectant la causalité) peut dans le cas d'opérations concurrentes différer de l'ordre de sérialisation. Une solution à ce problème nécessite de Défaire/Refaire des opérations. Plus précisément celle-ci consiste, lors de la livraison d'une opération, à défaire les opérations qui ont été délivrées avant elle mais qui lui sont postérieures dans l'ordre de sérialisation, ensuite à intégrer l'opération, puis à réintégrer les opérations défaites.

Être obligé de défaire des opérations pour les refaire ultérieurement n'est pas satisfaisant. En plus de la difficulté liée au Défaire/Refaire, il faut ajouter celle liée à l'intégration des opérations défaites. En effet, entre temps l'état de l'objet aura été modifié et on ne pourra alors pas envisager de les réexécuter sans au préalable avoir procédé à nouveau à leur intégration.

Pour éviter de devoir défaire puis refaire des opérations, SOCT3 [Sul98] met en œuvre un ordre de sérialisation continu, compatible avec l'ordre causal, qui permet de délivrer les opérations dans cet ordre. De cette façon, la condition C2 peut être éliminée, sans toutefois nécessiter de défaire puis refaire des opérations.

3.1.3 Utilisation de la transposée en arrière

L'algorithme SOCT3 utilise une procédure de livraison séquentielle qui en exploitant l'ordre de sérialisation global continu permet de garantir l'intégration des opérations distantes suivant le *même* ordre sur tous les sites. Cela permet d'utiliser des transpositions en avant et en arrière qui ne satisfont pas nécessairement à la condition C2. Cependant nous avons montré (cf. Annexe A) qu'il existe des cas où les fonctions de transposition en arrière ne sont plus à même de garantir la convergence des copies lorsque la condition C2 n'est plus satisfaite.

La convergence des copies étant un des objectifs principaux des algorithmes que nous étudions, il ne peut être question d'utiliser la transposée en arrière si elle ne permet pas de garantir que les différentes copies convergent. Il faut donc envisager la conception d'algorithme ne faisant pas appel à la transposition en arrière.

3.1.4 Solutions proposées

Face à ces limitations, notre objectif est donc de proposer des solutions nouvelles qui satisfont aux trois exigences suivantes : (i) ne pas utiliser de transposition en arrière, (ii) ne pas imposer la vérification de la condition C2 pour la transposition en avant et (iii) ne pas Défaire/Refaire des opérations. Pour répondre à ces exigences, nous proposons dans la suite deux nouveaux algorithmes, appelés SOCT4 et SOCT5, pour lesquels l'utilisation de la transposition en arrière n'est pas requise. Ils utilisent la fonction de transposition en avant sans requérir qu'elle satisfasse à la condition C2 et ne nécessitent pas de Défaire/Refaire des opérations.

Le premier, SOCT4, fait appel à la diffusion différée des opérations. Cette technique permet une simplification importante du processus d'intégration. Le deuxième algorithme, SOCT5, est une évolution de SOCT4 qui utilise la diffusion immédiate des opérations. Cette technique rend l'algorithme un peu plus complexe mais a l'avantage non négligeable de rendre plus performante la diffusion des opérations en autorisant les diffusions concurrentes entre les sites.

3.2 Ordre global continu et diffusion différée : algorithme SOCT4

3.2.1 Principe

Dans SOCT4 [VCFS00], l'idée consiste, comme dans SOCT3, à ordonner globalement les opérations suivant la relation $précède_S$, en les estampillant à l'aide d'un séquenceur, et à les délivrer sur chaque site suivant ce même ordre au moyen de la livraison séquentielle. La différence essentielle avec les autres algorithmes réside dans le fait de faire effectuer le travail de transposition en avant de l'opération à émettre, par le site émetteur plutôt que par les sites récepteurs, pour tenir compte des opérations concurrentes. Ceci présente plusieurs avantages importants :

a) cela évite d'avoir à séparer l'histoire sur les sites récepteurs pour distinguer les opérations concurrentes, rendant inutile l'utilisation de la transposition en arrière ;

b) l'opération reçue peut être rangée telle quelle dans l'histoire, sans avoir besoin d'être transposée en arrière.

Pour cela, la diffusion d'une opération doit être différée. Plus précisément, une opération *op* qui est générée sur un site est exécutée immédiatement pour satisfaire la contrainte de temps réel. Elle n'est diffusée qu'après avoir reçu une estampille et qu'après que toutes les opérations qui la précèdent dans l'ordre des estampilles (c.-à-d. $précède_S$), aient été reçues par le site émetteur. En outre, elle n'est diffusée qu'après avoir été transposée en avant par rapport aux opérations concurrentes reçues après sa génération et la précédant dans cet ordre global.

Dans cet algorithme, lorsqu'une opération distante, disons op_{i+1}, d'estampille $(i+1)$ est délivrée au site S par la procédure de livraison séquentielle, toutes les opérations op_j $(\forall j : 1 \leq j \leq i)$ qui la précèdent dans l'ordre global ont déjà été reçues et exécutées sur ce site. Elle peut donc être exécutée telle quelle s'il n'existe pas d'opérations locales, notées op_L, en attente de diffusion. Dans le cas contraire, s'il y a m opérations locales $op_{L_1}, op_{L_2}, \ldots op_{L_m}$, exécutées dans cet ordre et en attente de diffusion, son intégration dans l'histoire $H_S(n)$, où $H_S(n) = op_1.op_2.\ldots.op_i.op_{L_1}.op_{L_2}.\ldots.op_{L_m}$ (avec $i+m=n$) consiste à :

1) déterminer l'opération à exécuter sur l'état courant. L'opération op_{i+1} est transposée en avant pour tenir compte de l'exécution locale des opérations concurrentes $op_{L_1}, op_{L_2}, \ldots op_{L_m}$, et ensuite l'opération obtenue est exécutée sur l'état courant ;

2) réordonner l'histoire dans l'ordre des estampilles. Les opérations locales en attente de diffusion $op_{L_1}, op_{L_2}, \ldots op_{L_m}$, sont transposées en avant, les unes après les autres, pour tenir compte de l'exécution de l'opération concurrente op_{i+1}. De cette façon, elles n'auront pas à être transposées à la livraison. L'opération op_{i+1} est rangée telle quelle dans l'histoire, à son rang $(i+1)$.

En ce qui concerne le point 1), la transposée en avant de op_{i+1} correspond à op_{i+1}^{seq} , où $seq = op_{L_1}.op_{L_2}.\ldots.op_{L_m}$ est la séquence des opérations locales en attente de diffusion.

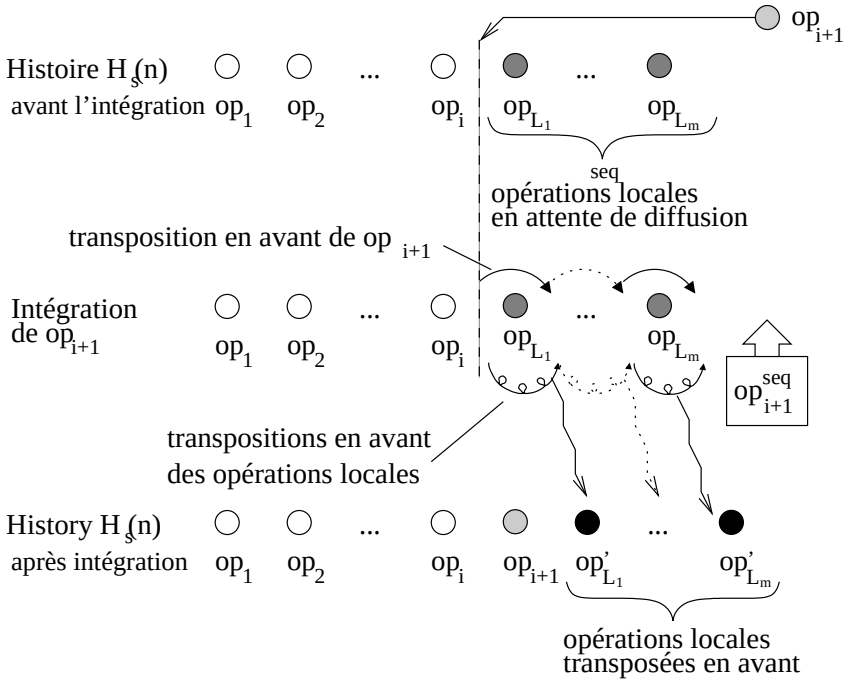


FIG. 3.1 – Principe de l'intégration dans l'algorithme SOCT4

En ce qui concerne le point 2), chaque opération locale op_{L_k} doit être transposée en avant par rapport à la transposée de op_{i+1}, notée $op_{i+1}^{seq'_k}$, où $seq'_k = op_{L_1}.op_{L_2}\ldots.op_{L_{k-1}}$ est la sous-séquence des $(k-1)$ opérations locales qui précèdent op_{L_k}. En procédant ainsi, les opérations $op_{i+1}^{seq'_k}$ et op_{L_k} sont définies sur le même état de l'objet. Les transposées en avant des m opérations locales sont respectivement les suivantes :

$$op_{L_1}^{op_{i+1}},\ op_{L_2}^{op_{i+1}^{op_{L_1}}},\ \ldots op_{L_m}^{op_{i+1}^{op_{L_1}.op_{L_2}\ldots.op_{L_{m-1}}}}.$$

Nous précisons en outre que les transpositions décrites au point 1) peuvent être réalisées en même temps que celle décrites en 2). Ainsi, dans un premier temps, la transposition en avant de l'opération distante op_{i+1} par rapport à la 1re opération locale, op_{L_1}, fournit l'opération $op_{i+1}^{op_{L_1}}$. En même temps, la transposition en avant de l'opération op_{L_1} par rapport à l'opération op_{i+1} fournit l'opération $op_{L_1}^{op_{i+1}}$. Puis, dans un second temps, la transposition de l'opération $op_{i+1}^{op_{L_1}}$ par rapport à la 2e opération locale, op_{L_2}, fournit l'opération $op_{i+1}^{op_{L_1}.op_{L_2}}$. En même temps, la transposition en avant de l'opération op_{L_2} par rapport à l'opération $op_{i+1}^{op_{L_1}}$ fournit l'opération $op_{L_2}^{op_{i+1}^{op_{L_1}}}$. Le processus se poursuit ainsi de suite avec le reste des opérations locales, $op_{L_3}\ldots.op_{L_m}$.

Nous avons envisagé jusqu'à maintenant le cas où l'opération op_{i+1} délivrée au site S était une opération distante. Lorsque l'opération délivrée est locale à S, comme elle a déjà été exécutée et rangée dans l'histoire, elle est ignorée. On notera que sa transposition pour tenir compte des éventuelles opérations concurrentes arrivées après sa génération, a déjà été faite avant sa diffusion.

Contrairement à SOCT3, dans SOCT4 les opérations délivrées ne sont plus nécessaires au fonctionnement de l'algorithme car elles n'interviennent plus dans les transpositions.

3.2.2 Algorithme SOCT4

L'algorithme SOCT4 est constitué des procédures : `Exécution_Locale`, `Diffusion_Différée` et `Intégration` ainsi que de la procédure `Livraison_Séquentielle` définie pour SOCT3 au chapitre 2.7. On utilise les mêmes notations que précédemment. On rappelle que N_S est l'estampille de la dernière opération délivrée sur le site S et est incrémentée dans la `Livraison_Séquentielle`. n est le nombre total d'opérations (locales et distantes) exécutées et rangées dans l'histoire (initialement $N_S = n = 0$).

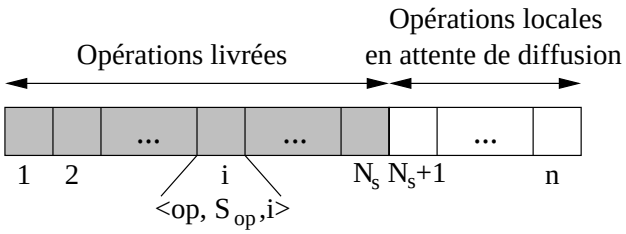


FIG. 3.2 – Représentation de l'histoire $H_S(n)$

Le tableau $H[]$ est utilisé pour mémoriser les opérations délivrées au site S, ainsi que les opérations locales. Les opérations délivrées sur S sont rangées dans le tableau, dans l'ordre des estampilles. Sans perte de généralité, on supposera qu'il y a identité entre l'estampille d'une opération et le rang de l'emplacement qu'elle occupe dans H. Ainsi, une opération d'estampille i (avec $i \leq N_S$) délivrée sur le site S a été rangée à l'emplacement $H[i]$. Seules les opérations locales en attente de diffusion peuvent temporairement occuper des emplacements $H[j]$ (avec $N_S < j \leq n$) qui ne correspondent pas à leurs estampilles. En réalité, les opérations une fois délivrées n'ont pas besoin d'être mémorisées.

La procédure `Exécution_Locale` est lancée à la génération d'une opération locale *op*. Après avoir été exécutée, *op* est rangée à la fin de l'histoire, c'est à dire à l'emplacement $(n+1)$. Ce n'est qu'après qu'elle ait obtenu une estampille de la fonction `Ticket` que l'on contrôle si elle peut être diffusée.

```
procédure Exécution_Locale(op) ;
début
  exécuter(op) ;
  n := n + 1 ;
  H[n] := ⟨op, S_op, −1⟩ ;
  H[n].estampille := Ticket() ;
  Diffusion_Différée() ;
fin
```

La procédure `Diffusion_Différée` diffuse à tous les sites (y compris S) la première opération locale si son estampille est égale à $N_S + 1$, puisqu'on est alors assuré que toutes les opérations qui portent des estampilles plus petites ont été délivrées.

```
procédure Diffusion_Différée() ;
début
  ⟨op, S_op, N_op⟩ := H[N_S + 1] ;
  si N_op = N_S + 1 alors
    diffuser (⟨op, S_op, N_op⟩) ;
  finsi ;
fin
```

La procédure `Intégration` est appelée lors de la livraison d'une opération par la procédure `Livraison_Séquentielle`. S'il s'agit d'une opération locale, elle l'ignore car l'opération a déjà été exécutée et elle figure dans l'histoire. S'il s'agit d'une opération distante, elle range le triplet $\langle op, S_{op}, N_{op}\rangle$ à l'emplacement $N_S + 1$ (c.-à-d. N_{op}) après avoir décalé d'un emplacement vers la droite les opérations locales en attente. Ensuite elle la transpose en avant par rapport aux opérations locales op_L et exécute la transposée résultante obtenue. Dans le même temps, chaque opération locale op_L est à son tour transposée en avant pour tenir compte de cette nouvelle opération concurrente. La procédure termine en contrôlant si une opération locale en attente peut être diffusée.

```
procédure Intégration(⟨op, S_op, N_op⟩) ;
  début
    si S ≠ S_op alors
      pour j := n bas N_S faire
        H[j + 1] := H[j]
      finpour ;
    H[N_S] := ⟨op, S_op, N_op⟩ ;
    n := n + 1 ;
    pour j := N_S + 1 haut n faire
      op_L := H[j].opération ;
      H[j].opération := Transpose_av (op, op_L) ;
      op := Transpose_av (op_L, op) ;
      finpour ;
      exécuter(op) ;
    finsi ;
  Diffusion_Différée() ;
fin
```

Pour des raisons de clarté, les algorithmes présentés n'incluent pas la gestion du parallélisme potentiel entre ***`Exécution_Locale`*** *et* ***`Intégration`****. La preuve de correction de SOCT4 est donnée en annexe B.*

3.3 Ordre global continu et diffusion immédiate : algorithme SOCT5

3.3.1 Principe

On rappelle que le principe de SOCT4 repose sur la diffusion différée des opérations. Cela autorise la transposition d'une opération *op* par rapport à toutes les opérations concurrentes qui la précèdent dans l'ordre total $précède_S$. Ainsi, ce traitement n'est effectué qu'une seule fois sur le site générateur de *op* et lors de l'intégration, il ne reste plus qu'a la transposer par rapport aux opérations concurrentes locales qui la suivent dans l'ordre $précède_S$. Ce principe de fonctionnement a pour avantage de simplifier grandement la procédure d'intégration et de rendre inutile la transposition en arrière. Cependant, le fait que les opérations soient diffusées de manière différée selon l'ordre global va en quelque sorte sérialiser les communications entre les différents sites. Ceci constitue un désavantage de SOCT4 par rapport à SOCT3 qui lui n'utilise pas de diffusion différée et profite donc du parallélisme dans la communication entre les sites.

Le but de l'algorithme SOCT5 est de rétablir l'équilibre sur ce point avec SOCT3. Pour cela, il faut envisager la mise en œuvre d'une diffusion immédiate des opérations aux autres sites. Ceci implique évidemment que les traitements effectués dans SOCT4 avant la diffusion de l'opération ne sont plus envisageables dans SOCT5. Ces traitements doivent donc être effectués par chaque site récepteur, le défi étant de ne pas nécessiter de transposition en arrière.

L'idée est en fait de simuler sur chaque site le comportement de tout le système collaboratif. Cela est possible en répliquant, sur chacun des sites, les parties d'histoire utilisées dans SOCT4 pour mémoriser les opérations en attente de diffusion (cf 3.3). Ainsi, sur chaque site S, l'histoire des opérations locales en attente de diffusion devient donc l'histoire des opérations du site S en attente de sérialisation, notée H_{Sd_S} et l'histoire des opérations délivrées devient l'histoire des opérations sérialisées, H_{Ss}. À ces deux histoires s'ajoutent les histoires des opérations des sites distants en attente de sérialisation, notées H_{Sd_x} où x est le nom du site distant correspondant à chaque histoire. Elles correspondent dans SOCT4 aux histoires des opérations en attente de diffusion des sites distants.

Malgré la multiplicité des histoires mises en jeu, nous verrons plus loin que le supplément d'informations à conserver sur chaque site reste limité par rapport à SOCT4 et plus encore si on le compare à SOCT3. Comme SOCT3 et SOCT4, SOCT5 exploite un ordre global défini par la relation $précède_S$. A la différence de ces deux algorithmes cependant SOCT5 utilise la livraison causale des opérations.

Une opération *op* qui est générée sur un site S y est exécutée immédiatement pour satisfaire la contrainte de temps réel. Au contraire de SOCT4 mais de manière similaire à SOCT3, elle est diffusée aussitôt qu'une estampille a pu lui être attribuée. Elle ne subit donc aucune transposition en avant et est donc diffusée telle quelle. L'opération est placée dans l'histoire des opérations du site S en attente de sérialisation sur le site S, H_{Sd_S}.

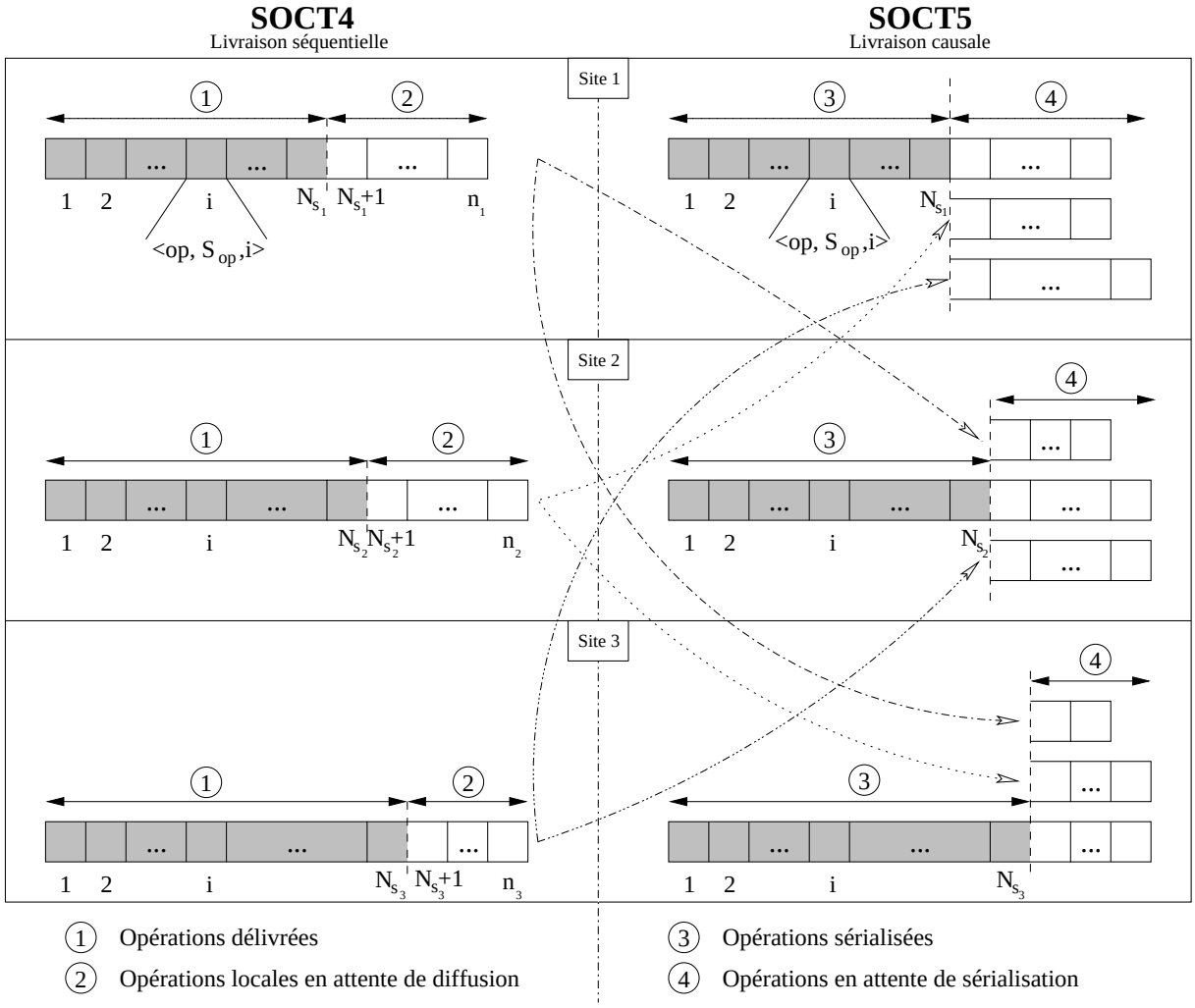


FIG. 3.3 – Passage de SOCT4 à SOCT5

Dans SOCT5, lorsqu'une opération distante, op_{i+1} par exemple, d'estampille $(i+1)$ est délivrée au site S par la procédure de livraison causale, toutes les opérations op_j qui la précèdent dans l'ordre causal ont déja été reçues sur ce site. Ces opération op_j ont leur estampilles telles que $j \leq i$. Elles n'ont en revanche pas forcément déjà été exécutées, l'exécution se faisant suivant l'ordre global défini par $précède_S$.

Une opération op_{i+1} est prête à être sérialisée sur un site S lorsque toutes les opérations d'estampille inférieure y ont déjà été sérialisées. Pour faire le parallèle avec SOCT4, on peut dire que cela correspond en fait au moment où l'opération sur son site de génération est prête à être diffusée.

Dans SOCT4, lorsqu'une opération est prête à être diffusée, elle a déjà été transposée en avant par rapport à toutes les opérations concurrentes sérialisées avant elle. Nous devons donc dans SOCT5, assurer cette même propriété, c.-à-d. que lorsqu'une opération op_{i+1} est prête à être exécutée sur un site S, elle doit déjà avoir été transposée en avant par rapport à toutes les opérations concurrentes sérialisées avant elle.

Pour cela, il faut, lorsqu'on sérialise une opération op_{i+1} sur un site S, réaliser non seulement son intégration sur le site local S mais simuler également son intégration sur les sites distants. Certaines opérations, concurrentes à op_{i+1}, en provenance de sites distants n'ont peut-être pas encore été reçues. Il faut donc conserver la transposée en avant de op_{i+1} par rapport à celles qui ont déjà été reçues de ces sites. De cette façon, cela permettra la transposition en avant de celles qui arriveront ultérieurement. La sérialisation d'une opération op_{i+1} consiste donc à :

1) déterminer l'opération à exécuter sur l'état courant. Si l'opération op_{i+1} est une opération distante, elle est transposée en avant pour tenir compte de l'exécution des opérations locales concurrentes $op_{L_1}, op_{L_2}, \ldots op_{L_m}$ contenu dans H_{Sd_S}. L'opération obtenue est ensuite exécutée sur l'état courant. Dans le cas contraire où op_{i+1} est locale, elle a déjà été exécutée et il n'y a rien à faire. Dans tous les cas, op_{i+1} est placée à la fin de l'histoire des opérations sérialisées $H_{Ss}(i+1) := H_{Ss}(i).op_{i+1}$;
2) déterminer les opérations qui seront placées dans ce que nous nommerons des *filtres* et qui serviront à transposer les opérations, concurrentes à des opérations sérialisées, qui pourraient ne pas avoir été encore reçues. Pour chaque histoire $H_{Sd_{S'}}$ d'opérations en attente de sérialisation d'un site S' $(S' \neq S)$ op_{i+1} est transposée en avant par rapport aux opérations $op_{d_{S',1}}, op_{d_{S',2}}, \ldots op_{d_{S',m_{S'}}}$ de $H_{Sd_{S'}}$. L'opération résultante obtenue est placée à la fin du filtre $H_{Sf_{S'}}$. Cela permettra la transposition en avant de celles qui arriveront ultérieurement ;
3) transposer en avant, les unes après les autres, les opérations en attente de sérialisation $op_{d_{x,1}}, op_{d_{x,2}}, \ldots op_{d_{x,m_x}}$ de chaque histoire H_{Sd_x} pour tenir compte de l'exécution de l'opération concurrente op_{i+1}.

Nous précisons que les transpositions décrites aux point 1) et 2) concernent des histoires indépendantes et peuvent donc être réalisées en exploitant au maximum la concurrence. Les transpositions décrites en 3) concernent les histoires utilisées aux points 1) et 2). Cependant, comme pour SOCT4, les transpositions décrites en 3) peuvent être réalisées en même temps que les transpositions décrites en 1) et 2). On utilisera pour cela la méthode décrite pour SOCT4 (p. 45).

Insertion d'une opération dans le filtre d'un site. Supposons qu'on sérialise, sur le site S, l'opération op_{i+1} (fig. 3.4) . Le traitement au point 2) consiste à transposer en avant l'opération op_{i+1} pour tenir compte des opérations concurrentes en attente de sérialisation $op_{d_{S'},1}$, $op_{d_{S'},2}$, ..., $op_{d_{S'},m_{S'}}$ en provenance du site S'. On obtient ainsi l'opération $op_{c_{S'},k+1}$ qui est placé dans le filtre $H_{Sf_{S'}}$ suivant l'inverse de l'ordre $précède_S$. On répète ce traitement pour chacune des histoires $H_{Sd_{S'}}$ ($\forall S' \neq S$).

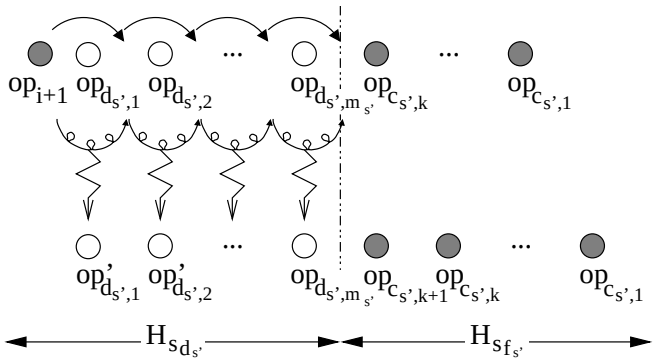


FIG. 3.4 – Insertion d'une opération dans le filtre associé à un site, lors de sa sérialisation

Ce traitement ne concerne pas les opérations du site où a été générée op_{i+1} puisque ces dernières ne sont pas concurrentes à op_{i+1}. De plus, au cas où, lors du traitement, on rencontre une opération distante en attente de sérialisation qui suit causalement (fig. 3.5) op_{i+1} alors :

- on peut interrompre le traitement, car op_{i+1} précède causalement les opérations suivantes puisque les opérations sont classées suivant l'ordre causal.
- la transformée de op_{i+1} n'a pas besoin d'être conservée puisqu'il n'y a plus de risque qu'une opération concurrente à op_{i+1} soit reçue du site S.

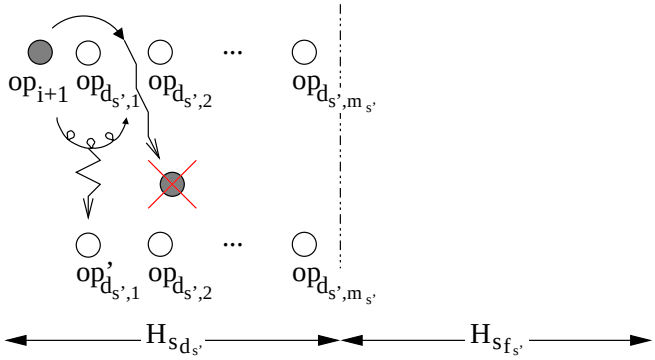


FIG. 3.5 – Interruption de l'insertion d'une opération dans le filtre associé à un site

On notera qu'en cas d'interruption du traitement, le filtre est forcément vide. En effet, les opérations qui sont placées dans le filtre précèdent totalement ($précède_S$) op_{i+1} du fait de la sérialisation. Par conséquent, si une opération $op_{d_{S',j}}$ de l'histoire des opérations en attente de sérialisation suit causalement op_{i+1} alors :

- elle aura interdit la mise en place, dans le filtre, des opérations qui précèdent totalement op_{i+1} si elle était déjà présente dans $H_{Sd_{S'}}$ ou
- elle aura provoqué la purge des opérations qui précèdent totalement op_{i+1} et qui étaient présentes dans le filtre lors de sa livraison (cf. § suivant).

Intégration d'une opération après livraison. Supposons que la procédure de livraison causale délivre une opération op_k (fig. 3.6). Si celle-ci est locale alors aucun traitement n'est nécessaire, car elle est déjà présente dans H_{Sd_S}. Sinon, l'opération op_k, que l'on supposera avoir été générée sur S', délivrée par la procédure de livraison causale est transposée en avant par rapport aux opérations présentes dans le filtre pour tenir compte des opérations distantes concurrentes sérialisées avant elles. L'opération obtenue, $op_{d_{S',m_{s'}+1}}$, est placée dans l'histoire des opérations du site S' en attente de sérialisation, $H_{Sd_{S'}}$ conformément à l'ordre causal. Dans le même temps, les opérations présentes dans le filtre $H_{Sf_{S'}}$ sont transposées en avant par rapport à l'opération délivrée.

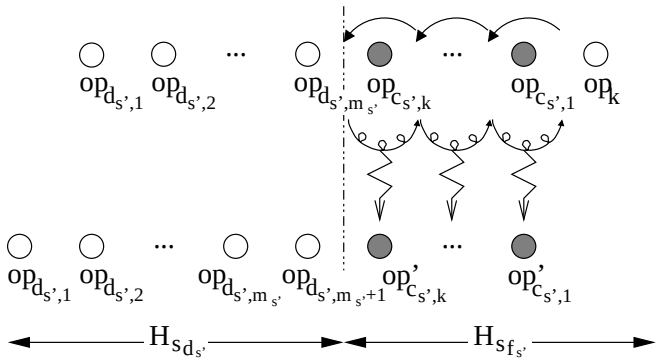


FIG. 3.6 – Insertion d'une opération dans la file d'attente associée à un site

On notera qu'aucun filtre n'est nécessaire pour le site local. En effet, les opérations locales sont placées immédiatement après leur exécution dans H_{Sd_S}. Les filtres sont nécessaires pour les sites distants en provenance desquels certaines opérations risquent d'être reçues après que les opérations qui leur sont concurrentes aient été sérialisées.

Lors de ce traitement, le filtre est purgé des opérations qui sont devenues inutiles. Ce sont les opérations du filtre qui précèdent causalement l'opération qu'on vient de recevoir (fig. 3.7).

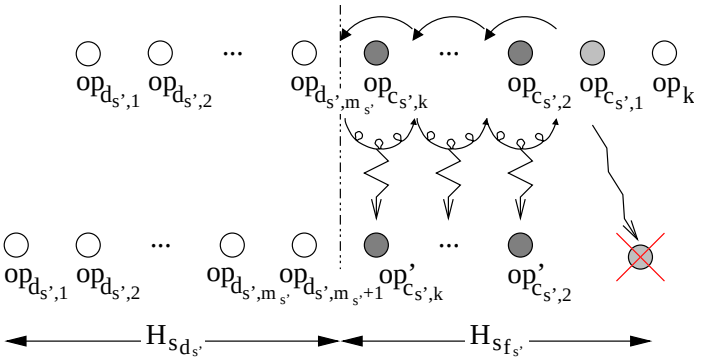


FIG. 3.7 – Suppression d'une opération du filtre lors de l'intégration d'une nouvelle opération

3.3.2 Algorithme SOCT5

L'algorithme SOCT5 est constitué des procédures : `Exécution_Locale`, `Diffuser`, `Intégration`, `Sérialisation` ainsi que de la procédure `Livraison_Causale` définie à la section 2.4.2. On utilise les mêmes notations que précédemment. On notera N_S l'estampille de la dernière opération sérialisée sur le site S. Elle est incrémentée dans la procédure `Sérialisation`.

Le tableau $Hs[]$ est utilisé pour mémoriser les opérations sérialisées sur le site S. Les opérations sérialisées sur S sont rangées dans le tableau, dans l'ordre des estampilles. Il y a donc identité entre l'estampille d'une opération et le rang de l'emplacement qu'elle occupe dans H. Ainsi, une opération d'estampille i (avec $i \leq N_S$) sérialisée sur le site S a été rangée à l'emplacement $Hs[i]$. En toute rigueur, les opérations sérialisées n'auraient pas besoin d'être mémorisées pour les besoins de l'algorithme.

Les opérations en attente de sérialisation sont placées dans un tableau $Hd[][]$. Les opérations en attente de sérialisation en provenance du site S' sont placées dans le tableau $Hd[S'][]$. Elles sont placées dans le tableau suivant l'ordre croissant de la composante S' de leur vecteur d'état (c.-à-d. suivant l'ordre causal).

Les variables n_x et l_x ($1 \leq x \leq$ *nombre de sites*) représentent respectivement le nombre d'opérations reçues du site x et le nombre d'opérations sérialisées du site x. À l'initialisation, $n_x = l_x = 0$.

La procédure `Exécution_Locale` est lancée à la génération d'une opération locale *op*, sur un site S. Après avoir été exécutée, *op* est rangée à la fin de l'histoire $Hd[S][]$, c'est à dire à l'emplacement $(n_x - l_x + 1)$. Ce n'est qu'après qu'elle ait obtenu une estampille de la fonction `Ticket` qu'elle est diffusée. La procédure `Diffuser` diffuse immédiatement à tous les sites (y compris S).

```
procédure Exécution_Locale(op) ;
début
  exécuter(op) ;
  n_x := n_x + 1 ;
  H[n_x − l_x] := ⟨op, S_op, SV_op, −1⟩ ;
  N_op = Ticket() ;
  H[n_x − l_x].estampille := N_op ;
  Diffuser(⟨op, S_op, SV_op, , N_op⟩) ;
fin
```

La procédure `Intégration` est appelée lors de la livraison d'une opération par la procédure `Livraison_Causale`. S'il s'agit d'une opération locale, elle est ignorée car l'opération a déjà été exécutée et figure déjà dans l'histoire $Hd[S][]$. S'il s'agit d'une opération distante, le quadruplet $\langle op, S_{op}, SV_{op}, N_{op}\rangle$ est rangé à l'emplacement $Hd[S_{op}][n_{S_{op}} - l_{S_{op}} + 1]$ après avoir transposé en avant l'opération par rapport aux opérations du filtre $Hf[S_{op}][]$. Si $N_{op} = N_S + 1$, alors on procède à la sérialisation de *op*. La fonction **Retirer** permet d'enlever l'opération la plus récente de l'histoire $Hd[S_i][]$ spécifiée en paramétre. Elle incrémente la valeur de l_{S_i} et décale les éléments du tableau d'un rang vers la gauche. En fait, l'opération *op* est retirée de l'histoire $Hd[S_op]$ dans laquelle elle vient d'être mémorisée sous la forme transposée op'. C'est finalement op' qui va être sérialisée.

```
procédure Intégration(⟨op, S_op, SV_op, N_op⟩) ;
début
  si S ≠ S_op alors
    n_{S_op} := n_{S_op} + 1 ;
    op' := Filtrer(⟨op, S_op, SV_op, N_op⟩) ;
      Hd[S_op][n_{S_op} − l_{S_op}] := ⟨op', S_op, SV_op, N_op⟩ ;
  finsi
  si N_op = N_S + 1 alors
    l_{S_op} := l_{S_op} + 1 ;
    ⟨op', S_op, SV_op, N_op⟩ := Retirer(Hd[S_op][]) ;
    Sérialisation(⟨op', S_op, SV_op, N_op⟩) ;
  finsi
fin
```

La procédure **filtrer** permet de tenir compte des opérations concurrentes qui ont été sérialisées avant que l'opération *op* soit livrées.

```
fonction filtrer(⟨op, S_op, SV_op, N_op⟩) ;
début
  pour j := 1 haut Hf[S_op][].size faire
    op_L := Hf[S_op][j].opération ;
    si op_L précède_C op alors
      retirer(Hf[S_op][]) ;
    sinon
      Hf[S_op][j].opération := Transpose_av (op, op_L) ;
      op := Transpose_av (op_L, op) ;
    finsi ;
  finpour ;
  retour op ;
fin
```

La procédure **Sérialisation** est exécutée, sur le site S', lorsqu'une opération *op* est prête à être intégrée. Elle est appelée soit à la suite de l'intégration de l'opération *op* elle-même, soit à la suite de la sérialisation d'une opération op' qui rend alors possible la sérialisation de *op*. Elle fait appel à la fonction **Transforme** qui effectue d'une part la transposition en avant des opération en attente de sérialisation d'un site S' par rapport à l'opération *op* et d'autre part transpose en avant *op* par rapport aux opérations de cette histoire. Elle retourne en résultat la transposée en avant de *op*. De plus, la fonction **Transforme** précise si toutes les opérations concurrentes à *op* en provenance du site S' ont été reçues. Si la réponse est `non` alors la transformée de *op* sera conservée dans le filtre.

```
fonction Transforme(⟨op, S_op, SV_op, N_op⟩, Hd[S'][]) ;
début
  si S' = S_op alors retour ⟨op, oui⟩
  sinon
    pour j := 1 haut Hd[S'][].size faire
      op_L := Hd[S'][j].opération ;
      si op précède_C op_L alors
        retour ⟨op, oui⟩
      sinon
        H[S'][j].opération := Transpose_av (op, op_L) ;
        op := Transpose_av (op_L, op) ;
      finsi ;
    finpour ;
    retour ⟨op, non⟩
  finsi ;
fin
```

```
procédure Sérialisation(⟨op, S_op, SV_op, N_op⟩) ;
début
  pour tout S' ∈ {ens. des sites} faire
    ⟨op', reponse⟩ := Transforme(⟨op, S_op, SV_op, N_op⟩, Hd[S'][]) ;
    si S' ≠ S alors
      si reponse = non alors
        size_{f_S'} := Hf[S'][].size ;
        Hf[S'][size_{f_S'} + 1] := op' ;
      finsi
    sinon
      executer(op') ;
      N_S := N_S + 1 ;
      Hs[N_op] := ⟨op, S_op, SV_op, N_op⟩ ;
    finsi
  finpour
  pour tout S' ∈ {ens. des sites} faire
    si Hd[S'][1].Nop = N_S + 1 alors
      ⟨op', S_op', SV_op', N_op'⟩ := Retirer(Hd[S_op][]) ;
      Sérialisation(⟨op', S_op', SV_op', N_op'⟩) ;
    finsi
  finpour
fin
```

Pour des raisons de clarté, les algorithmes présentés n'incluent pas la gestion du parallélisme potentiel entre ***Exécution_Locale****,* ***Intégration*** *et* ***Sérialisation****.*

3.4 Discussion sur le séquenceur

Dans les trois algorithmes, SOCT3, SOCT4 et SOCT5, un séquenceur est utilisé pour sérialiser globalement les exécutions. L'utilisation d'un séquenceur a pour conséquence dans SOCT4 que toutes les diffusion sont séquentielles ; il en résulte que la collaboration entre les utilisateurs sera d'autant plus difficile que le délai de propagation d'une opération sur le réseau est élevé (par exemple, de l'ordre d'une minute). Cette caractéristique fait que SOCT4 est plus adapté aux réseaux rapides. SOCT5, en exploitant une diffusion immédiate est moins sensible au délai d'acheminement des messages. Ce gain étant obtenu au prix d'un algorithme plus délicat à implémenter.

Pour des raisons de clarté, les algorithmes donnés ici sont séquentiels. Cependant, lorsqu'on exploite la concurrence potentielle entre les procédures, le fait qu'une opération locale soit en attente d'une estampille n'exclut pas l'exécution d'une autre opération locale ni l'intégration d'un opération distante.

Pour ce qui concerne le manque de robustesse du séquenceur, nous soulignons que la panne du séquenceur (ou la perte d'une estampille) n'entrave pas le fonctionnement local. La collaboration est suspendue mais chaque utilisateur peut continuer le travail séparément. La collaboration pourra être reprise aussitôt que le séquenceur sera redevenu opérationnel. L'effet est identique quand une opération estampillée n'est pas diffusée par un site malveillant.

3.5 Comparaison récapitulative des algorithmes

Dans cette section, nous nous appuierons sur le tableau récapitulatif (Tab. 3.1) pour donner une vision générale des algorithmes SOCT4 et SOCT5 et des différents algorithmes dOPT, adOPTed, GOT, GOTO, SOCT2, SOCT3 proposés dans la littérature pour répondre aux trois impératifs des systèmes collaboratifs à savoir : (1) respect de l'intention, (2) convergence des copies et (3) respect de la causalité. Nous verrons qu'il existe entre ces algorithmes certaines similitudes quand aux techniques employées et nous nous attarderons sur les différences qui font l'originalité de chacun d'eux.

Le respect de l'intention de l'usager est obtenu dans tous les algorithmes en transformant une opération par rapport à celles qui lui sont concurrentes de façon à permettre son exécution. Cette transformation est employée sous diverses appellation (L-transformation, Inclusion transformation, Transposition en avant). En outre pour assurer le respect de l'intention dans les situations de concurrence partielle, certains algorithmes mettent en jeu une transformation supplémentaire, qui permet de changer l'ordre d'exécution de deux opérations séquentielles, sans que leur intention soit violée ; il s'agit de l'Exclusion transformation dans GOT et de la Transposition en arrière dans SOCT2, SOCT3 et GOTO. Dans adOPTed le problème de la concurrence partielle est résolu par la construction et la mémorisation d'un graphe multidimensionnel qui permet de retrouver tous les ordres de sérialisation possibles. Seul SOCT4 et SOCT5 n'utilisent qu'une seule transformation (la Transposition en avant) dans la mesure où le problème de concurrence partielle y est résolu grâce à la diffusion différée dans SOCT4 et grâce à la transposition en avant des opérations livrées par rapport aux opérations des filtres dans SOCT5.

Pour assurer la convergence des copies, la définition des ces transformations doit en règle générale satisfaire deux conditions (C1 et C2). La condition C1 permet de garantir que le résultat de la transformation d'opérations concurrentes ne dépend pas de leur ordre. Tous les algorithmes supposent que les transformations vérifient la condition C1. Seul l'algorithme GOT n'impose pas cette condition mais il fixe un ordre global et contraint les transformations à se faire dans cet ordre ; cela oblige à défaire des opérations qui sont arrivées *en avance*, c.-à-d. avant d'autres opérations qui la précèdent dans l'ordre global. L'usage de la condition C2 vise à rendre le résultat de la transformation d'une opération par rapport à une séquence indépendant de l'ordre des opérations dans la séquence. L'algorithme dOPT, n'utilise pas la condition C2 mais ne permet pas de garantir la convergence des copies. De même, il ne propose pas de solution au problème de la concurrence par-

tielle, ces deux problèmes n'ayant pas encore été mis en évidence lors de son élaboration. La conformité à la condition C2, lorsqu'elle est possible, reste très coûteuse à vérifier, aussi est-elle avantageusement remplacée par un ordre de sérialisation global dans GOT, SOCT3, SOCT4 et SOCT5. Dans GOT, l'ordre global de sérialisation n'étant pas continu, il s'avère parfois nécessaire de défaire puis de refaire certaines opérations pour pouvoir intégrer une opération qui arrive *en retard* à sa bonne place dans l'histoire. Dans SOCT3, SOCT4 et SOCT5, un ordre global continu est obtenu en utilisant un séquenceur qui associe une estampille à chaque opération. La livraison et l'exécution des opérations distantes peut alors se faire suivant l'ordre des estampilles dans SOCT3 et SOCT4. Dans SOCT5, au contraire, la livraison est faite suivant l'ordre causal et seule l'exécution se fait suivant l'ordre des estampilles.

Le respect de la causalité est obtenu dans tous les algorithmes, hormis SOCT3 et SOCT4, par l'utilisation de vecteurs d'états qui permettent la mise en œuvre d'une procédure assurant la livraison des opérations suivant un ordre compatible avec l'ordre causal. Dans SOCT3 et SOCT4, l'utilisation d'un séquenceur pour obtenir des estampilles continues permet la livraison des opérations suivant un ordre global, compatible avec l'ordre causal. Ce même ordre permet, dans SOCT3, SOCT4 et SOCT5 de s'affranchir de la condition C2. SOCT4 ne requiert pas l'utilisation de vecteurs d'état. Cela le distingue de SOCT3 et SOCT5 qui les utilisent pour distinguer les opérations concurrentes.

En ce qui concerne la diffusion d'une opération, elle s'effectue de manière immédiate dans tous les algorithmes sauf dans SOCT4 où elle est différée jusqu'à ce que toutes les opérations qui la précèdent dans l'ordre global aient été reçues. Cela simplifie l'intégration et permet de se passer de la transposition en arrière, nécessaire dans SOCT2, SOCT3, GOT et GOTO. Sans requérir l'utilisation de la transposition en arrière, SOCT5 permet de profiter de la concurrence offerte par la diffusion immédiate.

Tous ces algorithmes sauf SOCT4 sont basés sur la mémorisation par chaque site des opérations qu'il a générées ou reçues. SOCT4 n'a besoin que des opérations qui sont en attente de diffusion, c.-à-d. des opérations générées et pas encore diffusées. La gestion de l'histoire sur chaque site en est ainsi simplifiée. L'algorithme SOCT5 exploite et mémorise une partie des opérations reçues mais n'a pas besoin des opérations qui sont déjà sérialisées.

		dOPT	adOPTed	GOT	GOTO	SOCT2	SOCT3	SOCT4	SOCT5
Contraintes	**Respect de l'intention**	dOP transformation	L−transformation et graphe multidimensionnel	Inclusion transformation et Exclusion transformation	Inclusion transformation et Exclusion transformation	Transposition en avant et Transposition en arrière	Transposition en avant et Transposition en arrière	Transposition en avant	Transposition en avant
	Respect de la causalité	Vecteurs d'état	Vecteurs d'état	Vecteurs d'état	Vecteurs d'état	Vecteurs d'état	Estampilles + Vecteurs d'état	Estampilles	Vecteurs d'état + Estampilles
	Convergence des copies	Condition C1	Condition C1 et Condition C2	Ordre global non continu et Défaire/Refaire	Condition C1 et Condition C2	Condition C1 et Condition C2	Condition C1 et Ordre global continu	Condition C1 et Ordre global continu	Condition C1 et Ordre global continu
Opération distante	**Diffusion**	Immédiate	Immédiate	Immédiate	Immédiate	Immédiate	Immédiate (dès attribution de l'estampille)	Différée (dans l'ordre des estampilles)	Immédiate (dès attribution de l'estampille)
	Livraison	Ordre causal	Ordre causal	Ordre causal	Ordre causal	Ordre causal	Ordre global continu	Ordre global continu	Ordre causal
	Exécution	Ordre causal	Ordre causal	Ordre causal	Ordre causal	Ordre causal	Ordre global continu	Ordre global continu	Ordre global continu
Histoire	**Ordre mémorisé**	Ordre d'exécution	Plusieurs ordres équivalents respectant l'ordre causal	Ordre global (= ordre d'exécution	Ordre causal optimisé	Ordre causal optimisé	Ordre global continu (≠ ordre d'exécution)	Ordre global continu (≠ ordre d'exécution)	Ordre global continu (≠ ordre d'exécution) + qq. op. temporaires
	Opération mémorisée (au moment de son intégration)	Opération exécutée	Opération reçue et certaines opérations transformées	Opération exécutée	Opération exécutée	Opération exécutée	Opération transformée conforme à l'ordre des estampilles	Opération reçue	Opération transformée conforme à l'ordre des estampilles

TAB. 3.1 – Techniques mises en œuvre au sein des différents algorithmes

4

Extensions pour la mobilité

Sommaire

4.1 Introduction

Le but de ce chapitre est de montrer comment les algorithmes SOCT4 et SOCT5 peuvent être utilisés dans le cadre d'une collaboration impliquant des sites mobiles.

Jusqu'à présent, nous avons uniquement considéré un réseau de communication fiable. Dans le cadre d'une extension prenant en compte la mobilité, il nous faut abandonner cette hypothèse. En effet, les sites mobiles sont caractérisés par un accès dégradé au réseau. Ceci peut être dû aux caractéristiques intrinsèques des technologies mises en œuvre ou précisément aux contraintes de mises en œuvre de ces technologies, soumises à des aléas de fonctionnement, dans le cas d'un réseau sans-fil par exemple. Une caractéristique importante d'un site mobile est le risque fréquent de déconnexion temporaire.

Dans un premier, nous présenterons comment la collaboration impliquant des sites mobiles a été traitée dans une approche existante. Nous verrons que les techniques basées sur les transpositions qui sont mises en œuvre dans les algorithmes de type SOCT généralisent celles basées sur la commutativité qui sont utilisées dans cette approche.

4.2 Collaboration dans COACT

4.2.1 Principe

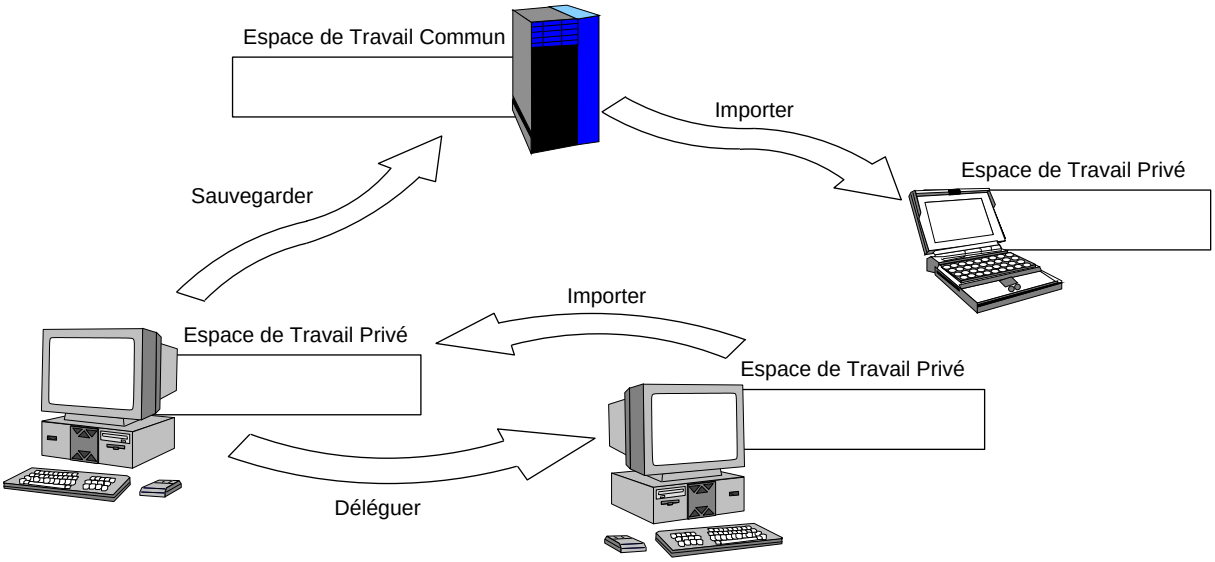


FIG. 4.1 – Principe de la collaboration dans COACT

Le modèle collaboratif de COACT[KTW97] propose un canevas pour gérer les activités collaboratives de longue durée dans le cadre d'environnements multi-utilisateurs. Ce canevas doit permettre le développement d'applications collaboratives asynchrones.

Le principe de fonctionnement de COACT est illustré sur la figure 4.1. Les objets partagés sont représentés dans l'espace de travail commun. Cet espace contient la copie de

référence des objets et assure leur persistance. L'état d'un objet dans l'espace de travail commun représente l'état de l'objet après validation des activités coopératives. Comme dans les algorithmes de type SOCT, chaque utilisateur dispose d'un espace de travail privé. Les objets dont l'utilisateur a besoin pour réaliser sa propre activité sont répliqués dans son espace de travail privé.

Au début d'une activité de collaboration, les utilisateurs acquièrent les objets dont ils auront besoin en les *important* dans leur espace de travail privé depuis l'espace de travail commun. Au cours de la phase de coopération, un utilisateur peut importer les modifications apportées à l'objet par un autre utilisateur depuis l'espace privé de ce dernier. Il peut également *déléguer*, c.-à-d. fournir les modifications qu'il a lui même effectuées à un autre utilisateur, charge à ce dernier de les intégrer dans son activité ou non. Le principe dans les deux cas est le même et nous nous placerons dans la suite dans le cas d'un utilisateur réalisant l'*import* des modifications effectuées par un autre utilisateur.

Fusion d'histoires

Une histoire dans COACT est une séquence totalement ordonnée d'opérations. À chaque opération est associé un identifiant unique, ses paramètres d'appel et le résultat de son exécution avec ces paramètres. Pour chaque espace de travail, une histoire répertorie la séquence d'opérations qui y a été exécutée depuis l'état initial. Cet état initial, O_i, est le même pour tous les espaces de travail privés des utilisateurs participant à une même activité de collaboration. La *fusion d'histoire* est le mécanisme permettant, dans COACT, de réaliser les échanges d'informations nécessaires à la collaboration. Dans la suite, sans perte de généralité, on supposera que toutes les opérations de l'histoire opèrent sur le même objet.

Lorsqu'un utilisateur décide d'importer une partie d'activité, il sélectionne dans l'histoire H de l'espace de travail privé de l'autre utilisateur, l'ensemble A des opérations correspondantes à cette partie d'activité. Puis il demande la fusion de cette histoire avec la sienne.

L'algorithme de fusion de COACT agit en deux étapes.

Etape 1. Construction d'une histoire valide.
Cela consiste à obtenir, à partir de la sous-histoire A, une sous-histoire AE *valide* sur l'état O_i commun. Une histoire est dite *valide* sur un état O si les opérations qu'elle contient peuvent être exécutées séquentiellement, à partir de l'état O, dans l'ordre où elles se trouvent dans l'histoire.

Etape 2. Intégration des opérations
Cela consiste à vérifier la validité de l'exécution des opérations de AE dans l'espace de travail privée en tenant compte des opérations qui y ont déjà été exécutées.

4.2.2 Construction d'une histoire valide

Comme on l'a vu à la section 1.2, une opération op_2 peut-être dépendante de l'effet produit par l'exécution antérieure d'une opération op_1. L'histoire AE sera donc l'histoire valide associée à A, si elle contient toutes les opérations de A et toutes les opérations de H dont dépend une opération de AE. Par analogie, dans les algorithmes de type SOCT, on pourrait dire qu'une *histoire* est *valide* si pour toute opération *op* qu'elle contient, elle contient aussi toutes les opérations qui précèdent causalement *op*. Dans COACT, la dépendance est exprimé au moyen de la *commutativité en arrière* [Wei88]. La commutativité en arrière y est définie de la manière suivante[KTW97] :

Définition 4.1 (Commutativité en arrière dans COACT)

L'opération op_2 commute en arrière avec l'opération op_1, noté ***comm_ar****(op_1, op_2) si quel que soit O_i, à partir duquel la séquence $op_1.op_2$ (resp. $op_2.op_1$)* ***peut être exécutée*** *alors :*

$$\begin{aligned} O_i.op_1.op_2 &= O_i.op_2.op_1, \\ \texttt{resultat}(O_i, op_1) &= \texttt{resultat}(O_i.op_2, op_1) \quad et \\ \texttt{resultat}(O_i, op_2) &= \texttt{resultat}(O_i.op_1, op_2) \end{aligned}$$

où `resultat`*(O_j,op_k) est le résultat rendu par l'exécution de op_k sur l'état O_j.*

On remarquera que dans COACT, la commutativité en arrière est définie improprement comme une relation symétrique alors qu'en fait, la commutativité en arrière dans sa définition initiale est une relation asymétrique [GFP95] :

Définition 4.2 (Commutativité en arrière)

L'opération op_2 commute en arrière avec l'opération op_1, noté ***comm_ar****(op_1, op_2) si quel que soit O_i, à partir duquel la séquence $op_1.op_2$* ***a été exécutée*** *alors :*

$$\begin{aligned} O_i.op_1.op_2 &= O_i.op_2.op_1, \\ \texttt{resultat}(O_i, op_1) &= \texttt{resultat}(O_i.op_2, op_1) \quad et \\ \texttt{resultat}(O_i, op_2) &= \texttt{resultat}(O_i.op_1, op_2) \end{aligned}$$

où `resultat`*(O_j,op_k) est le résultat rendu par l'exécution de op_k sur l'état O_j.*

Remarque. Quelles que soient deux opérations op_1 et op_2, exécutées dans cet ordre :
comm_ar(op_1,op_2) $\Rightarrow$ `Transpose_ar`(op_1,op_2) = (op_2,op_1).
Autrement dit, si deux opérations op_1 et op_2 commutent en arrière dans cet ordre alors leur transposée en arrière correspond à ces mêmes opérations prises dans l'ordre inverse. La transposée en arrière ne modifie donc pas ces opérations.

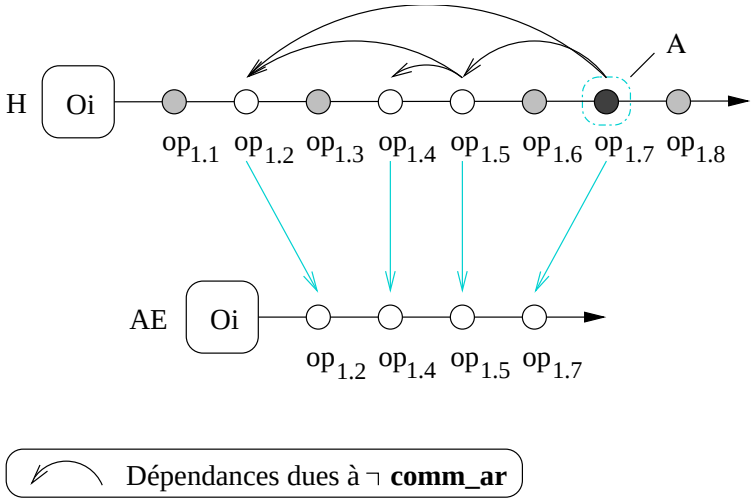


FIG. 4.2 – Construction d'une histoire valide dans COACT

La figure 4.2 donne un exemple du déroulement de cette étape. Pour plus de clarté, on supposera que $op_{1.7}$ représente à elle seule une activité. Un utilisateur a décidé de partager cette activité, soit en l'important, soit en la déléguant. Il a donc sélectionné l'opération $op_{1.7}$. On suppose l'existence de dépendances dues à la non commutativité en arrière de certaines opérations. L'algorithme de fusion en utilisant ces dépendances fournit l'histoire AE qui doit être transmise pour que l'exécution de $op_{1.7}$ soit permise. Il s'agit ici de l'histoire contenant les opérations $op_{1.2}$, $op_{1.4}$, $op_{1.5}$ et bien sûr $op_{1.7}$. Les opérations $op_{1.2}$ et $op_{1.5}$ sont concernées car $op_{1.7}$ ne commute en arrière ni avec $op_{1.5}$ ni avec $op_{1.2}$. L'opération $op_{1.4}$ est concernée car $op_{1.7}$ dépend d'elle de façon indirecte par l'intermédiaire de $op_{1.5}$ qui ne commute pas en arrière avec $op_{1.4}$.

Une fois cette histoire AE construite, elle est transmise au site où se trouve l'utilisateur qui a déclenché l'import. A là réception de celle-ci, la deuxième étape de l'algorithme de fusion peut commencer. Elle consiste à intégrer les opérations de AE dans l'histoire de l'espace de travail de l'utilisateur qui a lancé l'*import*.

4.2.3 Intégration des opérations

Cette étape d'intégration est soumise aux mêmes contraintes que l'étape de construction de l'histoire valide ; c.-à-d. que l'histoire de l'espace de travail privé obtenue après intégration des opérations de AE doit être une histoire valide. Pour cela, il faut tenir compte des effets des opérations qui ont été exécutées en concurrence dans l'espace de travail privé. Ces opérations ont pu avoir des effets sur l'état de l'objet. Il importe donc de s'assurer que ces effets n'empêchent pas l'exécution correcte des opérations de AE. Pour chaque opération op_i de AE, prise dans l'ordre où elle se trouve dans AE, et pour chaque opération op_j de l'histoire H associée à l'espace de travail privé, prise dans l'ordre où elle se trouve dans H, l'algorithme COACT doit contrôler que l'exécution de op_i est valide après l'exécution de op_j.

La validité de l'exécution des opérations concurrentes est basée dans COACT sur la relation de *commutativité en avant* [Wei88]. La commutativité en avant est définie, dans COACT, de la manière suivante :

Définition 4.3 (Commutativité en avant)

Les opérations op_1 et op_2 commutent en avant, noté ***comm_av****(op_1, op_2) si quel que soit O_i, à partir duquel l'opération op_1 (resp. op_2)* ***peut être exécutée*** *alors :*

$$\begin{aligned} O_i.op_1.op_2 &= O_i.op_2.op_1, \\ \texttt{resultat}(O_i, op_1) &= \texttt{resultat}(O_i.op_2, op_1) \quad et \\ \texttt{resultat}(O_i, op_2) &= \texttt{resultat}(O_i.op_1, op_2) \end{aligned}$$

où `resultat`*(O_j,op_k) est le résultat rendu par l'exécution de op_k sur l'état O_j.*

Remarque. Quelles que soient deux opérations concurrentes op_1 et op_2 :
comm_av(op_1,op_2) $\Leftrightarrow$ `Transpose_av`(op_1,op_2) $= op_2$ et `Transpose_av`(op_2,op_1) $= op_1$.
Autrement dit, si deux opérations op_1 et op_2 commutent en avant alors la transposée en avant d'une opération par rapport à l'autre est cette même opération. La transposée en avant d'une opération ne modifie pas cette opération.

Ainsi, l'exécution d'une opération op_i après l'exécution d'une opération op_j est valide, dans COACT, si op_i commute en avant avec op_j. L'intégration d'une opération op_i dans l'histoire H associée à un espace de travail privé ne peut se faire que si son exécution est valide sur l'état courant de l'objet. L'opération op_i doit donc commuter en avant avec toutes les opérations de l'histoire, H pour que son intégration soit autorisée.

L'étape d'intégration, dans COACT, consiste donc à tester la commutativité en avant de chacune des opérations de l'histoire échangée AE avec chacune des opérations de l'histoire H associée à l'espace de travail privé.

Si toutes les opérations commutent en avant, alors les opérations de AE peuvent être exécutées en séquence à partir de l'état courant en suivant l'ordre dans lequel elles se trouvent dans AE (fig. 4.3). Dans le cas contraire, un ou plusieurs conflits existent entre les opérations déjà exécutées et celles qu'on désire importer. L'utilisateur a alors le choix entre abandonner la tentative d'import (fig. 4.4-a) ou défaire les opérations de l'histoire H qui sont en conflit avec les opérations qui doivent être intégrées (fig. 4.4-a).

4.2.4 Conclusion

Le modèle collaboratif COACT permet la collaboration entre des utilisateurs dont certains peuvent se trouver sur des sites mobiles. Il utilise pour cela un algorithme de fusion d'histoires. Celui-ci est basé sur la commutativité en avant et la commutativité

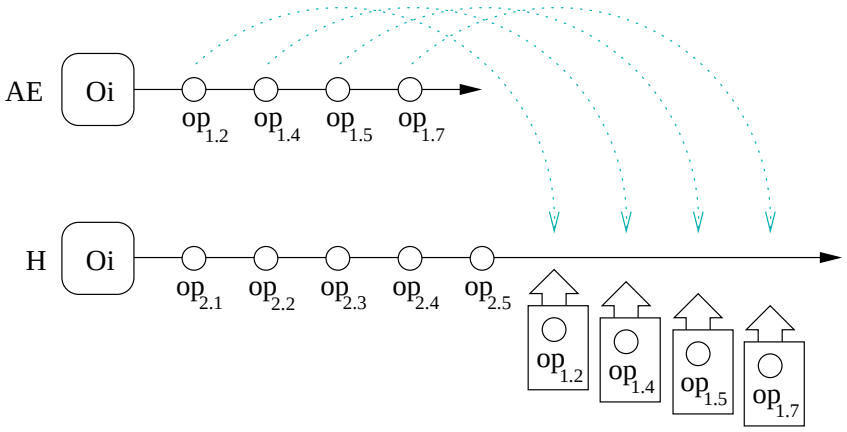


FIG. 4.3 – Intégration en l'absence de conflits dans COACT

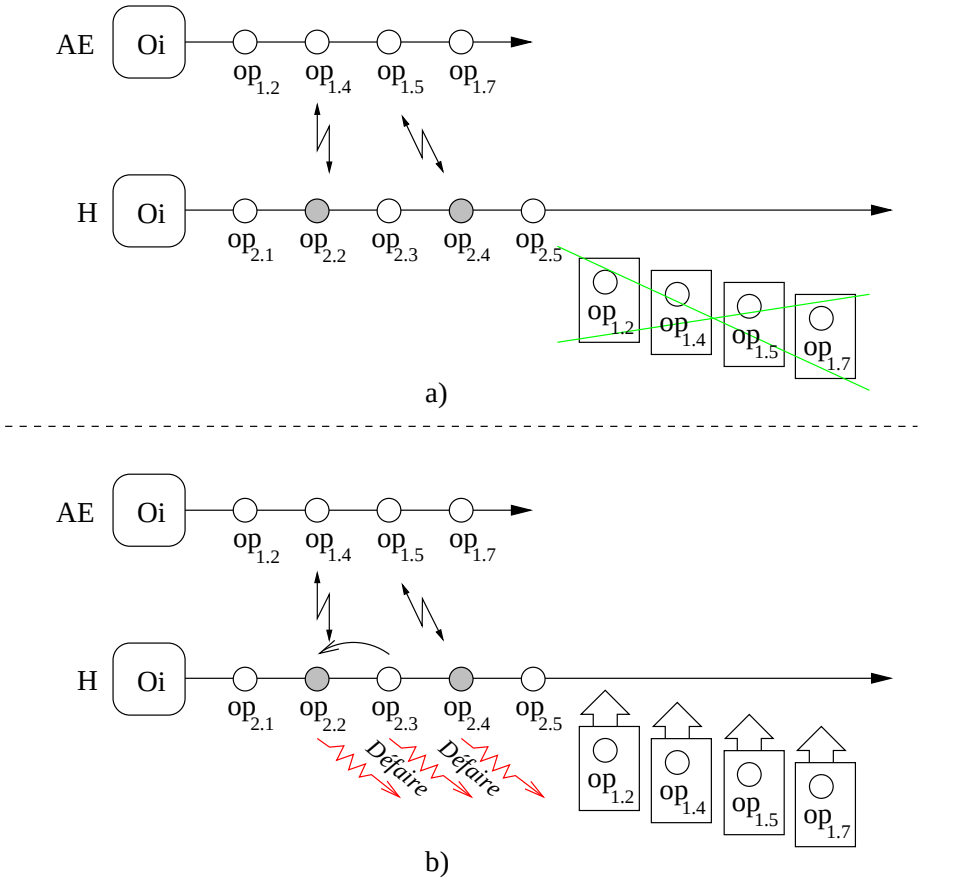


FIG. 4.4 – Intégration en présence de conflits dans COACT

en arrière des opérations. L'intégration d'une opération y est autorisée notamment si elle commute en avant avec celles qui lui sont concurrentes. Or on a vu que, quelles que soient deux opérations concurrentes, si elles commutent en avant alors leurs transposées correspondent à ces mêmes opérations. En d'autres termes, la commutativité correspond à un cas particulier de transposition. On remarquera alors qu'au prix de la définition des transpositions correspondantes, les algorithmes de type SOCT sont beaucoup plus généraux dans la mesure où ils autorisent la fusion d'histoires contenant des opérations qui peuvent ne pas commuter, évitant ainsi aux utilisateurs de perdre une partie de leur travail. Dans la suite, nous exposons brièvement comment les algorithmes SOCT4 et SOCT5 peuvent être adaptés pour supporter la présence de sites mobiles.

4.3 Collaboration à partir d'un site mobile dans SOCT4 et SOCT5

L'objectif de cette section est de proposer, comme cela a été fait pour SOCT2 et SOCT3 [SCF98], une extension aux algorithmes SOCT4 et SOCT5 pour autoriser la présence de sites mobiles.

4.3.1 Position du problème

Comme on l'a précisé plus haut, un site mobile est soumis à un risque élevé de déconnexion. Ces déconnexions empêchent la diffusion des messages vers les autres sites ainsi que la réception des messages en provenance de ces derniers. Toutefois, les algorithmes que nous présentons ont pour exigence de garantir l'exécution immédiate des opérations sur le site où elles ont été générées. Ainsi, le fait qu'un site soit déconnecté n'interdit pas la génération et l'exécution des opérations locales. Un utilisateur présent sur un tel site peut donc continuer à travailler de façon autonome. Du point de vue des sites qui sont restés connectés, le constat est le même ; ils peuvent continuer à collaborer comme si le site déconnecté ne générait aucune opération.

On peut distinguer quatre phases lorsqu'on considère la collaboration d'un site mobile, à savoir : (1) la collaboration en mode connecté, (2) la déconnexion, (3) le travail en mode déconnecté et (4) la reconnexion.

Remarquons que la collaboration en mode connecté se déroule pour un site mobile de manière similaire à la collaboration à partir d'un site fixe. Les opérations diffusées sont reçues normalement et les opérations générées localement peuvent être diffusées, une fois estampillées, sans aucun problème.

En ce qui concerne le travail en mode déconnecté, il a pour conséquences d'une part, qu'un site mobile déconnecté ne peut plus diffuser ses opérations et d'autre part, qu'un site mobile déconnecté ne reçoit plus les opérations qui sont diffusées. Ces deux conséquences ne sont pas des problèmes puisque le fait qu'un site déconnecté ne puisse plus diffuser d'opérations n'empêche ni ce site mobile ni les autres sites de fonctionner. Il en est de même en ce qui concerne l'impossibilité pour le site déconnecté de recevoir les opérations diffusées par les autres sites.

Nous esquissons maintenant les phases de transition que sont la connexion et la déconnexion. La phase de déconnexion doit permettre de s'assurer que les différents sites vont pouvoir continuer à fonctionner de manière satisfaisante : le site déconnecté en permettant à l'utilisateur qui l'exploite de travailler de façon autonome et les sites qui sont connectés en permettant la collaboration des différents utilisateurs. La phase de connexion doit quand à elle assurer que le site qui se reconnecte va pouvoir participer à la collaboration, que le travail effectué lorsqu'il était déconnecté va être pris en compte par les autres sites et que lui même va prendre en compte le travail effectué par ces mêmes sites alors qu'il était déconnecté.

4.3.2 Déconnexion

Elle peut survenir de manière volontaire ou non. Dans le cas d'une déconnexion volontaire, le site mobile a la possibilité de préparer sa déconnexion. Au contraire, dans le cas d'une déconnexion involontaire, le site ne peut que constater sa déconnexion et n'est alors plus en mesure de communiquer. Comme nous allons le voir, la différence entre les mode de diffusion utilisés dans SOCT4 et SOCT5 font que la manière dont survient la déconnexion a des conséquences plus ou moins gênantes selon les cas.

Déconnexion volontaire dans le cas de SOCT4

Le problème dans SOCT4 est relatif à la diffusion différée. En effet, elle suppose qu'à un instant donné il est possible qu'un site détienne des opérations qui bien que porteuses d'une estampille sont encore en attente de diffusion. Si on interrompt la connexion avant que ces opérations n'aient été diffusées, cela provoquera un arrêt de la collaboration des sites connectés. En effet, dans SOCT4, l'intégration d'une opération ne peut se faire que si toutes les opérations d'estampilles inférieures ont déjà été intégrées. Si un site détient des opérations estampillées non encore diffusées quand il se déconnecte, les autres sites ne pourront pas intégrer les opérations portant des estampilles supérieures. La collaboration sera interrompue même si la génération et l'exécution des opérations locales restent possibles.

Un site mettant en œuvre l'algorithme SOCT4 doit donc, lors d'une déconnexion volontaire, appliquer au choix une des trois actions suivantes.

Action 1. Ne plus demander d'estampilles pour ses opérations locales et attendre que toutes celles qui avaient déjà été estampillées soient diffusées.

Action 2. Ne plus demander d'estampilles pour ses opérations locales et révoquer les estampilles attribuées a des opérations en attente de diffusion (ce qui suppose que de nouvelles estampilles leur soient attribuées après la reconnexion).

Action 3. Faire appel à un site *mandataire*. Le site mobile S_b transmet à un site fixe S_f la partie de son histoire locale contenant des opérations déjà estampillées mais non encore diffusées, $op_{L_1}, op_{L_2}, \ldots op_{L_x}$. Il transmet également N_{S_b} l'estampille correspondant à la dernière opération qu'il a intégrée. Le site fixe est alors chargé d'effectuer le travail du site S_b comme si celui-ci était resté

connecté. Il transpose en avant les opérations $op_{L_1}, op_{L_2}, \ldots op_{L_x}$ par rapport aux opérations délivrées par la procédure de livraison causale et lorsque les opérations $op_{L_1}, op_{L_2}, \ldots op_{L_x}$ sont prêtes à être diffusées, il se charge de le faire à la place du site S_b. L'estampille N_{S_b} lui permet de savoir s'il a reçu des opérations $op_{D_1}, op_{D_2}, \ldots op_{D_y}$ que le site S_b n'avait pas encore reçues lorsqu'il lui a transmis ses opérations locales $op_{L_1}, op_{L_2}, \ldots op_{L_x}$. Si c'est le cas, il lui suffit de transformer les opérations transmises par S_b par rapport aux opérations $op_{D_1}, op_{D_2}, \ldots op_{D_y}$ présentes dans son histoire. Il exploite ainsi une caratéristique de SOCT4, à savoir que les opérations diffusées sont rangées telles quelles dans l'histoire. Le site S_b n'a plus besoin de s'occuper du traitement des opérations $op_{L_1}, op_{L_2}, \ldots op_{L_x}$ même s'il se reconnecte avant la diffusion par S_f de ces opérations. À la livraison de chacune d'elles, il se contente de mettre à jour N_{S_b} dans la mesure où l'opération se trouve déjà à son rang dans l'histoire.

Déconnexion involontaire dans SOCT4

En cas de déconnexion involontaire d'un site utilisant l'algorithme SOCT4, on ne peut pas envisager de mettre en place l'une des actions envisagées suite à la déconnexion volontaire dans la mesure où cela nécessite de pouvoir diffuser certaines opérations avant la déconnexion. Une procédure envisageable dans ce cas serait un généralisation de l'action 3. Elle consisterait à ce qu'un site, pour lequel une déconnexion involontaire est possible, délègue de manière permanente l'estampillage et la diffusion des ses opérations locales à un site mandataire fixe. De cette façon, aucune opération ne risquerait de voir sa diffusion impossible du fait d'une déconnexion. Le site mobile continuerait à gérer son histoire de manière habituelle. Il continuerait à transposer en avant les opérations distantes reçues, par rapport aux opérations locales en attente de diffusion et inversement. En effet, cela ne requiert pas que les opérations locales soient estampillées. La seule différence résiderait dans le fait que le site mobile doit diffuser à son site mandataire ses opérations locales dès leur génération. Le site mandataire gèrerait quand à lui l'histoire du site mobile à la manière dont un site gère l'histoire d'un site distant dans l'algorithme SOCT5.

Déconnexion volontaire dans SOCT5

Dans le cas de SOCT5, la déconnexion volontaire pose moins de problème dans la mesure où SOCT5 diffuse immédiatement les opérations. Il peut donc en cas de déconnexion, diffuser toutes ses opérations locales une fois estampillées avant de se deconnecter.

Déconnexion involontaire dans SOCT5

Dans le cas d'une déconnexion involontaire, on se retrouve même avec SOCT5 dans une situation similaire à celle de SOCT4. En effet, il est possible que certaines opérations qui ont déjà été estampillées n'aient pas encore été diffusées au moment où survient la déconnexion. Leur non diffusion bloquera donc sur les autres sites la sérialisation des opérations qui se fait, rappelons-le, suivant l'ordre des estampilles. Une procédure qui peut être mise en place consiste à faire se qui est proposé pour SOCT4, c'est à dire à

déléguer l'estampillage des opérations à un site mandataire fixe. Ainsi, la diffusion des opérations pourra se faire à partir de ce site fixe en toute sécurité.

4.3.3 Reconnexion

La reconnexion est une phase au cours de laquelle le site mobile doit d'une part s'assurer du rétablissement de la connexion aux sites fixes et d'autre part veiller à récupérer les éventuels messages qui auraient été diffusés en son absence et qu'il n'aurait pas reçus. On suppose que le site mobile S_b qui se reconnecte connaît au moins un site fixe S_j. Il n'y a pas de différence dans la procédure de reconnexion entre SOCT4 et SOCT5.

Le principe général en est le suivant, pour la reconnexion du site S_b.

i) Demander à un site connecté S_j la liste LP_j des sites participants.

ii) Envoyer un message `reconnecter` à tous les sites S_i de la liste LP_j, en précisant :
   - l'identifiant S_b du site mobile, permettant ainsi à S_i de mettre à jour sa liste LP_i,
   - la liste LOR_{S_b} des opérations reçues par le site S_b en provenance du site S_i, afin que celui-ci puisse lui transmettre uniquement les opérations qu'il a diffusées mais que S_b n'a pas reçues.

iii) Les sites S_i répondent avec un message `reconnexion` au site S_b en précisant :
   - leur identifiant S_i,
   - la liste LOM_{S_i} des opérations diffusées par S_i que S_b n'a pas reçues.

iv) Si un site S_k de LP_j ne répond pas au message `reconnecter`, dans le cas d'un site mobile par exemple, le site S_b envoie un message `reconnexion-erreur` au site S_j en lui précisant :
   - son identifiant S_b,
   - l'identifiant S_k du site fautif,
   - la liste LOR_{S_b} des opérations reçues par le site S_b en provenance du site S_i, afin que le site S_j puisse lui transmettre uniquement les opérations, diffusées par S_k, que le site S_b n'a pas reçues.

v) Après vérification de l'absence effective du site S_k, le site S_j répond par le message `reconnexion-déléguée` en précisant :
   - son identifiant S_j,
   - l'identifiant S_k du site pour lequel il opère,
   - la liste LOM_{S_k} des opérations diffusées par S_k que S_b n'a pas reçues.

Deuxième partie

Traitement de l'annulation

5

Problématique de l'annulation

Sommaire

5.1 Introduction

La mise en œuvre, dans un logiciel applicatif interactif, d'une interface utilisateur à effet immédiat (c.-à-d. une interface dans laquelle les actions des utilisateurs ont un effet immédiat sur le document en cours d'édition) rend souhaitable la possibilité donnée à l'utilisateur d'annuler ses actions. Cela lui permet de corriger ses éventuelles erreurs et incidemment rend l'utilisation de l'application plus aisée en autorisant les essais.

L'annulation d'une opération *op* peut être réalisée de deux manières différentes [AD92] :
- **implicitement** : l'utilisateur génère de lui-même une opération dont l'exécution annule les effets de l'opération *op*,
- **explicitement** : l'utilisateur fait appel à une fonction du système qui déclenche la génération d'une opération dont l'exécution annule les effets de l'opération *op*.

Dans le cadre d'une application mono-utilisateur, le premier problème sera donc de déterminer quelle sera l'opération qui sera annulée lorsque l'utilisateur invoquera la fonction d'annulation. L'opération qui sera choisie sera fonction des différentes politiques d'annulation disponibles et du choix qu'aura fait l'utilisateur quand à celle qui sera activée. Le deuxième problème sera ensuite de déterminer l'opération dont l'exécution annule les effets de celle qui a été choisie.

Dans le cadre d'une application collaborative, les opérations exécutées n'ont pas toutes forcément été générées par l'utilisateur qui invoque la fonction `annuler`. Un deuxième problème se pose alors : à savoir s'il faut ou non considérer les opérations des autres utilisateurs dans le choix de l'opération à annuler (que ce choix soit fait par l'utilisateur –*annulation implicite*– ou par le logiciel –*annulation explicite*).

Les solutions récemment proposées [Sun00, RG99] pour prendre en compte la problématique de l'annulation dans les applications collaboratives ne tiennent compte que de l'annulation *explicite*. En effet, elles utilisent une procédure ad-hoc qui distingue de manière nette les opérations générées lors d'une annulation des autres opérations. La prise en compte de l'annulation *implicite* nécessiterait alors de distinguer aussi les opérations générées dans ce cadre ce qui semble peu envisageable.

5.2 Annulation dans une application non partagée

On considère une application permettant à un utilisateur de manipuler un objet [7] (texte, image, etc.) au moyen d'opérations prédéfinies. On utilise le modèle adopté pour les algorithmes SOCT. L'objet se trouve à l'origine dans un état initial O_0. On suppose que l'exécution d'une opération op_j, réalisant une intention I_j, sur l'objet dans un état O_i le fait passer dans un nouvel état O_{i+1}. Le déroulement d'une session de travail est

[7] pour des raisons de simplicité et sans perte de généralité, on considère que l'application ne manipule qu'un seul objet même si les opérations agissent sur des entités indépendantes de cet objet qui pourraient elles-mêmes être considérés comme objets.

modélisé par l'histoire $H(n)$ (Fig. 5.1) des opérations exécutées et qui ont amené l'objet de l'état initial O_0 à l'état courant O_n (n étant le nombre d'opérations exécutées sur l'objet depuis l'état initial O_0).

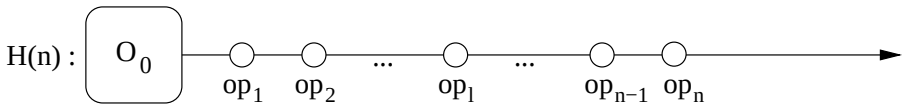


FIG. 5.1 – Histoire des opérations

L'annulation d'une opération op_j, réalisant une intention I_j consiste à exécuter sur l'objet, dans l'état courant O_n, l'opération op_{n+1} qui conduit à un état équivalent à celui qui aurait été obtenu si on avait exécuté sur l'objet, à partir de l'état initial O_0, toutes les opérations op_i nécessaires à la réalisation des intentions I_i ($\forall i, i \neq j : 1 \leq i \leq n$). En reprenant le formalisme présenté précédemment, après l'annulation de op_j, on obtient l'état O_{n+1} qui doit vérifier l'égalité suivante :

$$\begin{aligned} O_{n+1} &= O_0.op_1.op_2 \ldots op_{j-1}.op_j.op_{j+1} \ldots op_n.op_{n+1} \\ &= O_0.op_1.op_2 \ldots op_{j-1}.op'_{j+1} \ldots op'_n \end{aligned}$$

où $op'_{l\ (j+1 \leq l \leq n)}$ est l'opération qui réalise les même intentions que op_l en tenant compte de l'absence de op_j. La problématique se résume donc à trouver quelle doit être l'opération op_{n+1} à exécuter pour obtenir ce résultat (Fig. 5.2).

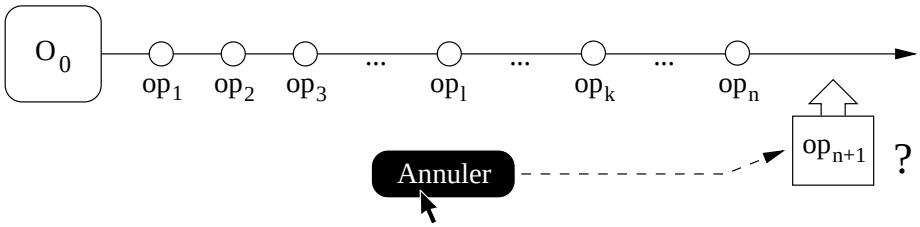


FIG. 5.2 – Problématique de l'annulation

Dans le cadre d'une application non-répartie mono-utilisateur, les différents modes d'annulation qui peuvent être proposés sont les suivants :

1. annulation de la dernière opération uniquement,
2. annulation suivant l'ordre chronologique inverse,
3. annulation libre ou sélective.

5.2.1 Annulation de la dernière opération

Dans ce premier mode de fonctionnement, seule la dernière opération peut être annulée. Après avoir annulé une opération (Fig. 5.3.1), un nouvel appel de la fonction d'annulation,

conduit à refaire l'opération qui avait été annulée (Fig. 5.3.2). L'appel de la fonction d'annulation après une première annulation correspond alors à la fonction `refaire` (`redo`). La fonction d'annulation agit, dans ce mode, comme un interrupteur qui active ou désactive la dernière opération effectuée.

L'utilisateur qui a fait une erreur doit agir sans délai pour la corriger, dans le cas contraire l'exécution d'une nouvelle opération (autre que l'annulation, bien sûr) constitue en quelque sorte une validation implicite de l'opération précédente qui ne peut alors plus être défaite.

Ce mode d'annulation nécessite que, pour chaque opération devant être annulée, il existe une opération inverse qui annule ses effets. Dans le cas où il existe une opération pour laquelle cette hypothèse ne se vérifie pas (opération ayant un effet irréversible), il faut pour pouvoir l'annuler, conserver l'état qui prévaut avant exécution de l'opération. Ce mode est en fait une version simplifiée du mode chronologique 5.2.2. Sa mise en œuvre peut se justifier par un des cas de figure suivants :

– impossibilité de mémoriser plus d'une opération,
– opérations aux effets irréversibles et impossibilité de mémoriser plus d'un état.

Lorsque l'utilisateur demande l'annulation, l'opération op_{n+1} que l'on doit exécuter est l'opération op_n^{-1}, inverse de la dernière opération exécutée, op_n (Fig. 5.3-1). En cas de nouvelle annulation, on exécute une nouvelle fois l'inverse de la dernière opération exécutée, c.-à-d. l'opération op_n (Fig. 5.3-2). On obtient alors la fonction `refaire` (`redo`).

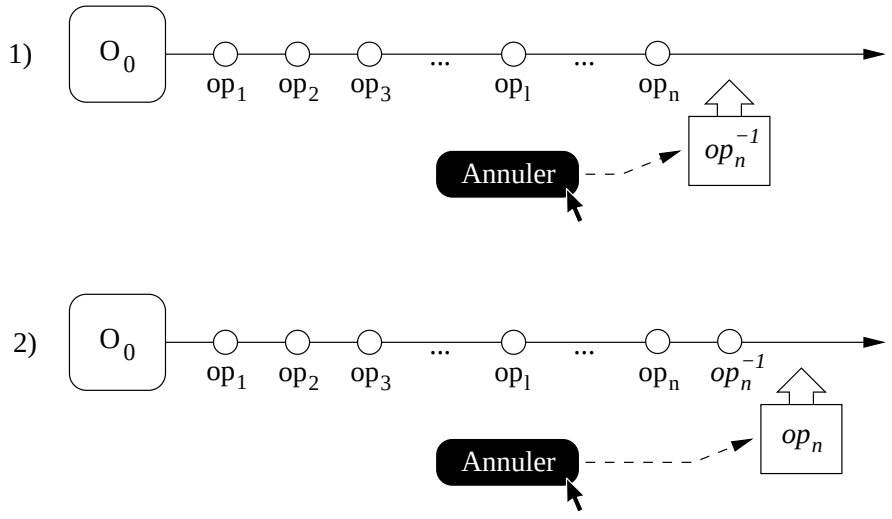


FIG. 5.3 – Annulation de la dernière opération

Ce mode d'annulation que nous désignerons aussi par *annulation simple* par la suite est utilisé par exemple dans le logiciel de dessin vectoriel XFIG.

5.2.2 Annulation dans l'ordre chronologique inverse

Dans le second mode de fonctionnement, l'utilisateur peut annuler consécutivement plusieurs opérations suivant l'ordre chronologique inverse (c.-à-d. en commençant par

la plus récente). L'appel répété de la fonction d'annulation va provoquer l'annulation de la dernière opération, puis de l'avant-dernière, etc. L'utilisateur dispose alors de la possibilité de corriger une erreur antérieure mais cela au prix de la perte de ce qu'il a produit postérieurement à cette erreur. Le cas des opérations à effets irréversibles (cf. § précédent) peut-être pris en compte moyennant le coût de la sauvegarde des états successifs de l'objet.

Ce mode d'annulation est notamment mis en œuvre dans le traitement de texte WORD et dans l'éditeur de texte EMACS. Dans cet éditeur, l'emploi de la commande `annuler` débute une séquence d'annulation au cours de laquelle les appels successifs provoqueront l'annulation d'opérations de plus en plus anciennes. Pour cela, un pointeur marque dans l'histoire des opérations exécutées, la prochaine opération devant être annulée (Fig. 5.4-1 à 5.4-3).

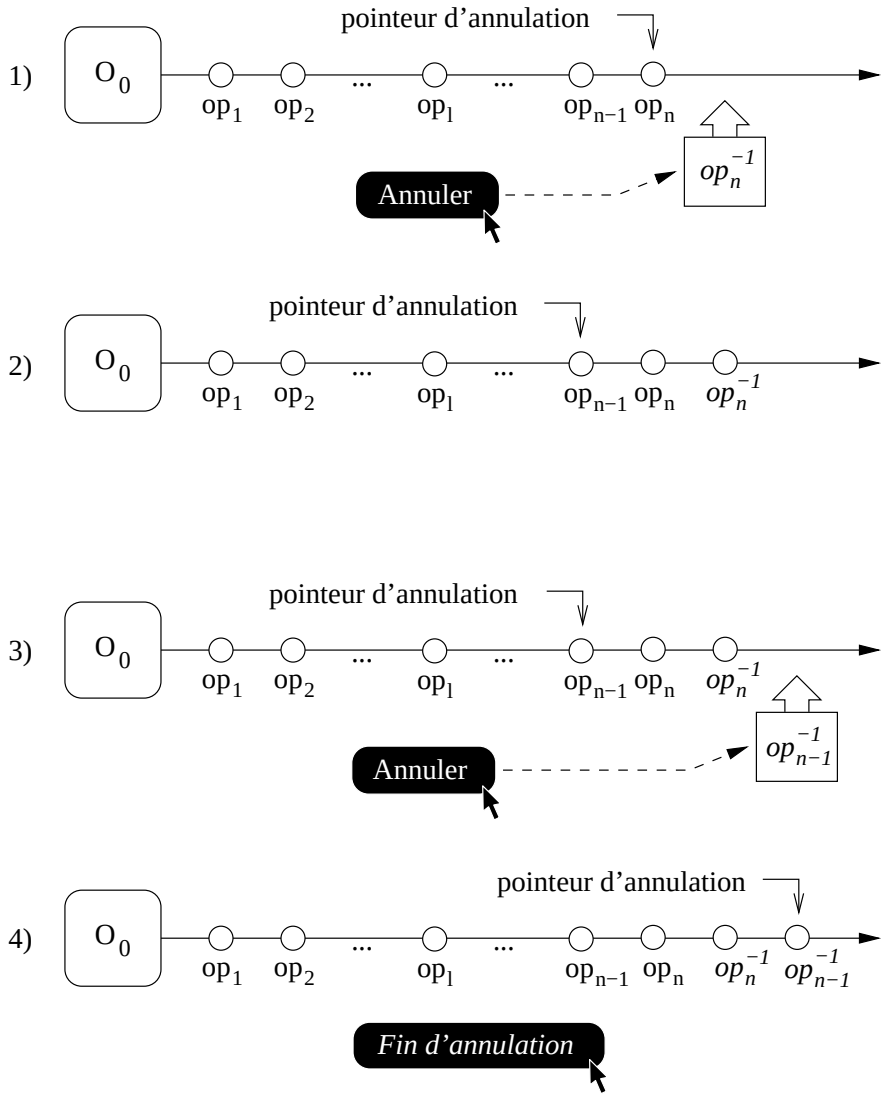


FIG. 5.4 – Annulation chronologique

Cette séquence d'annulation n'est interrompue que par l'utilisation d'une commande

autre que la commande `annuler` (Fig. 5.4-4). L'interruption de la séquence d'annulation provoque alors la ré-initialisation du pointeur d'annulation sur la plus récente opération exécutée. Une opération *sans effet* (`ctrl-f`) permet ainsi de basculer en mode `refaire` en initialisant le pointeur sur la dernière opération d'annulation exécutée lors de la séquence d'annulation. De cette façon, les utilisations ultérieures de la commande `annuler` conduisent à *refaire* les opérations précédemment annulées.

5.2.3 Annulation libre

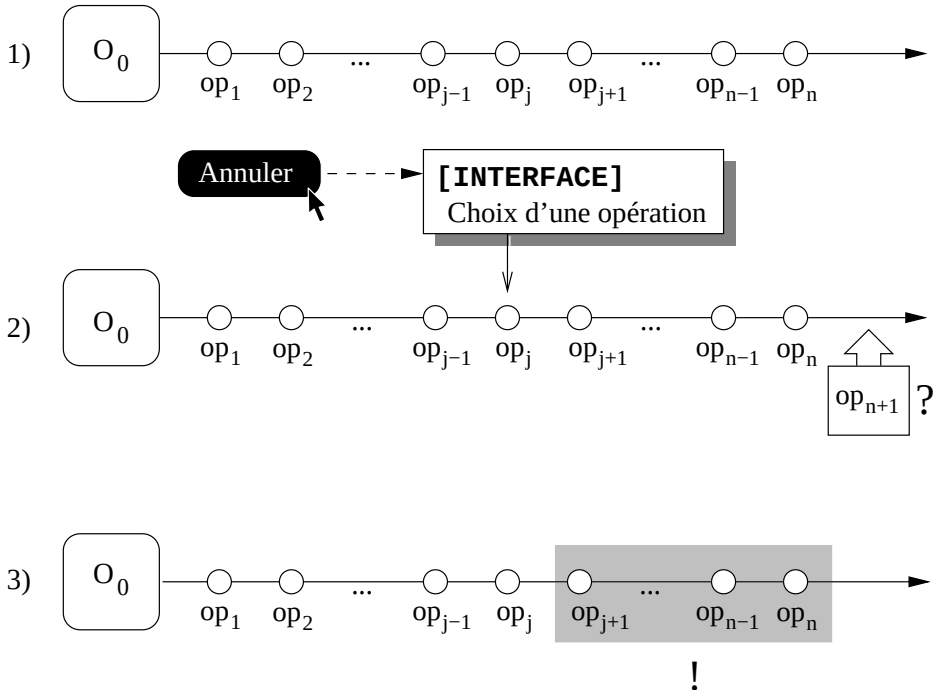


FIG. 5.5 – Annulation libre

Dans le mode d'annulation libre ou sélective, on offre à l'utilisateur la possibilité d'annuler n'importe laquelle des opérations qu'il a générées. Cela lui permet de corriger des erreurs même anciennes sans perdre le travail qu'il aura effectué entre-temps. Cela suppose que l'on fournisse une interface spécifique permettant à l'utilisateur de choisir l'opération qu'il désire annuler. Ce mode est évidemment le plus souple pour l'utilisateur mais aussi le plus délicat à mettre en œuvre. En effet, contrairement aux cas précédents l'opération que l'on va annuler n'est pas forcément la dernière ayant été exécutée sur l'objet (Fig. 5.5-3). Exécuter l'opération inverse sur l'état courant de l'objet ne suffit plus dans ce cas pour obtenir l'annulation d'une opération. Il faut alors envisager la mise en œuvre des fonctions de transposition pour obtenir une opération dont l'exécution donnera le résultat escompté. L'utilisation des fonctions de transpositions peut se faire selon l'une ou l'autre des deux stratégies suivantes. Pour simplifier, dans les schémas suivants on considère que l'opération choisie est op_{n-1}.

Stratégie 1. Générer d'abord l'opération inverse $\overline{op_j}$ de l'opération à annuler op_j.
Transposer en avant l'opération inverse pour tenir compte des opérations qui ont été exécutées après l'opération que l'on souhaite annuler.

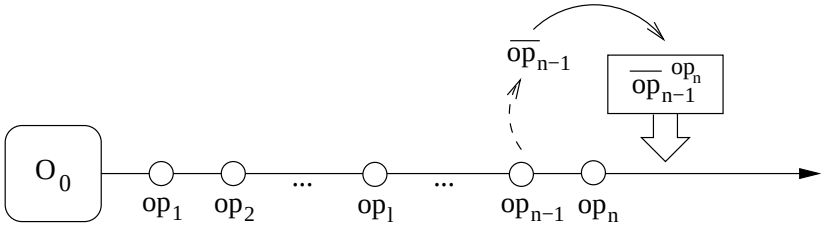


FIG. 5.6 – Annulation de op_{n-1} suivant la stratégie 1

Stratégie 2. Transposer en arrière les opérations qui ont été exécutées postérieurement à l'opération qu'on souhaite annuler, de façon à ce que la transposée op'_j de l'opération à annuler soit la dernière de l'histoire.
Générer l'inverse $\overline{op'_j}$ de l'opération obtenue.

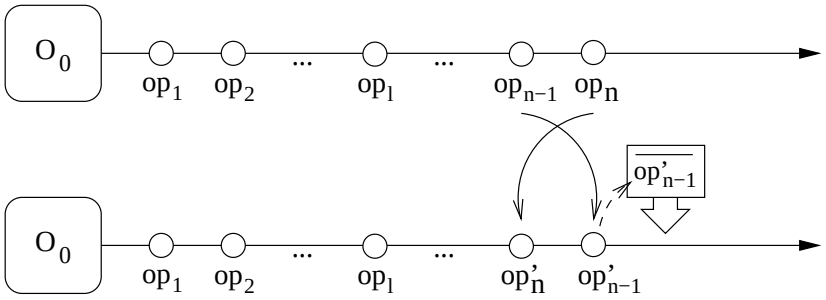


FIG. 5.7 – Annulation de op_{n-1} suivant la stratégie 2

La mise en œuvre de ce mode requiert donc, en plus de la nécessité d'obtenir une opération inverse pour chaque opération disponible, qu'il soit possible de transposer en arrière les opérations les unes par rapport aux autres. La sauvegarde des états avant exécution pour les opérations aux effets irréversibles est sans utilité dans ce mode d'annulation.

5.3 Annulation dans une application partagée

Dans une application partagée, la particularité provient du fait que les opérations qui sont exécutées sur l'objet ne sont pas forcément générées par un unique utilisateur. Contrairement à ce qui se passe pour une application mono–utilisateur, dans laquelle les opérations sont indifférenciées et traitées de façon égale, dans une application partagée, les opérations exécutées peuvent être distinguées suivant l'utilisateur qui les a générées.

L'incidence évidente de cette particularité sur la fonction d'annulation est l'ajout d'une nouvelle dimension au choix de l'opération qui devra être annulée lorsque l'utilisateur appelle la fonction `annuler`.

Le fait que l'application soit aussi répartie ne change pas la nature du problème. La différence supplémentaire est qu'il existe une copie de chaque objet partagé sur chacun des sites. Les opérations d'un utilisateur sont alors les opérations *locales* et les opérations des autres utilisateurs sont les opérations *distantes*.

5.3.1 Différentes portées de l'annulation

Deux choix sont possibles pour la *portée* de l'annulation, suivant le type d'opération considéré lorsqu'on fait appel à la fonction d'annulation.

Portée locale. La fonction de désignation de l'opération à annuler ne permet de désigner que les opérations locales. Lors de la recherche de l'opération à annuler, on ne va retenir comme candidates que les opérations locales. Dans cette configuration, l'utilisateur n'est pas autorisé à annuler les opérations générées par d'autres utilisateurs.

Portée globale. La fonction de désignation de l'opération à annuler permet de choisir toutes les opérations, aussi bien celle générées par l'utilisateur qui demande l'annulation que celles, distantes, qui ont été générées par d'autres utilisateurs.

Dans les deux premiers modes d'annulation (annulation de la dernière opération et annulation dans l'ordre chronologique inverse) ce choix influence directement celui de l'opération qui sera annulée lorsque l'utilisateur fait appel à la fonction `annuler`. Dans le mode d'annulation libre, par contre, ce choix peut être pris en compte pour filtrer dans l'interface de sélection les opérations sur lesquelles l'utilisateur pourra faire porter l'annulation.

5.3.2 Politiques d'annulation

L'association d'un mode d'annulation et d'une portée d'annulation définit une politique d'annulation. Cette dernière détermine entièrement l'opération qui sera annulée lorsque l'utilisateur fera appel à la fonction d'annulation. Il existe plusieurs modes d'annulation et plusieurs portées d'annulation mais toutes les combinaisons de mode et de portée ne sont pas souhaitables si l'on veut fournir à l'utilisateur une méthode d'annulation fiable. En effet, certaines combinaisons peuvent conduire à des politiques qui ne sont pas décidables pour l'utilisateur ; cela veut dire que, lorsque l'utilisateur fera appel à la fonction d'annulation, il ne sera pas en mesure de dire quelle sera l'opération qui sera annulée.

Pour illustrer ce propos, examinons les différentes combinaisons que nous pouvons obtenir à partir des portées et des principaux modes d'annulation : chronologique et libre (ou sélectif). Les modes d'annulation ne sont pas ici examinés de façon exhaustive, le but étant simplement de montrer que toutes les combinaisons ne sont pas souhaitables.

Annulation de portée locale suivant l'ordre chronologique inverse

Dans le cadre de cette politique d'annulation, l'utilisateur a la possibilité d'annuler uniquement les opérations locales (c.-à-d. celles qu'il a lui-même générées). Les opérations

ne peuvent en outre être annulées que suivant l'ordre chronologique inverse de leur exécution[8]. Lorsque l'utilisateur veut annuler une opération, il utilise la fonction d'annulation. La dernière opération exécutée est alors annulée. Chaque nouvel appel de la fonction d'annulation annulera alors l'opération locale qui précède directement, dans l'ordre chronologique d'exécution, celle qui vient d'être annulée.

Le résultat d'un appel à la fonction d'annulation est parfaitement établi et prévisible par l'utilisateur qui en fait usage. Cette politique d'annulation peut donc être proposée, sans risque, à l'utilisateur. Ce dernier ne sera pas confronté à un résultat inattendu puisqu'il sait exactement quelle sera l'opération qui sera annulée. Ce n'est pas le cas avec la combinaison suivante qui de ce fait ne devrait pas être proposée à l'utilisateur.

Annulation de portée globale suivant l'ordre chronologique inverse

Comme précédemment, les opérations sont choisies suivant l'ordre chronologique inverse de leur exécution. Maintenant toutes les opérations, qu'elles soient locales (c.-à-d. générées par l'utilisateur faisant appel à la fonction d'annulation) ou distantes (c.-à-d. générées par les autres utilisateurs), sont prises en compte lors de l'annulation. Cependant, avec cette politique d'annulation l'utilisateur peut ne pas être en mesure de savoir quelle opération sera annulée. Cela est dû à la concurrence ; en effet, pendant que l'annulation est en train d'être effectuée, d'autres utilisateurs peuvent travailler. Cela peut, en certaines circonstances, conduire à la suite d'événements suivants.

Deux utilisateurs travaillent simultanément et après quelques instants, l'utilisateur U_1 décide d'annuler certaines de ses opérations. Pour ce faire, il invoque la fonction d'annulation pour commencer par annuler les dernières opérations. Supposons qu'au même moment l'utilisateur U_2 génère l'opération *op*, qui est immédiatement exécutée et transmise au site où se trouve l'utilisateur U_1. Si cette opération arrive entre l'instant où l'utilisateur décide d'annuler ses dernières opérations et celui où il invoque réellement la fonction d'annulation, alors l'opération *op* générée par l'utilisateur U_2 sera annulée sans qu'aucun des acteurs n'en ait eu réellement l'intention. En fait, en utilisant cette politique d'annulation, le système n'est pas en mesure d'assurer le respect des intentions des utilisateurs durant le processus d'annulation. C'est la raison pour laquelle nous pensons que cette politique ne peut pas être raisonnablement proposée à l'utilisateur. Une telle situation ne peut pas survenir dans le cas où l'on ne considère que les opérations locales. En effet dans ce cas, comme dans le cas d'une application mono-utilisateur, l'utilisateur peut déterminer quelles seront les opérations qui seront annulées lorsqu'il invoquera la fonction d'annulation.

Annulation libre de portée locale

Dans ce mode d'annulation (c.-à-d. libre), les opérations sont choisies par les utilisateurs. Un élément de l'interface utilisateur permet à l'utilisateur faisant appel à la fonction d'annulation de choisir la ou les opérations qu'il désire annuler. La portée de l'annulation étant définie locale, seules les opérations locales peuvent être sélectionnées par l'utilisateur. Comme l'utilisateur désigne directement les opérations devant être annulées, aucune

[8]Les opérations étant locales, cet ordre correspond également à l'ordre inverse de leur génération.

ambiguïté ne peut subsister quant au résultat du processus. Bien évidemment, les autres utilisateurs peuvent continuer à apporter des modifications au document en concurrence. Avec cette politique, on est assuré que si une annulation survient, elle est due à la réalisation d'une intention d'un des utilisateurs.

Annulation libre de portée globale

En utilisant cette politique d'annulation, on se retrouve dans une configuration similaire à la précédente, à la différence près qu'ici on autorise l'annulation par un utilisateur des opérations des autres utilisateurs. Comme dans le cas précédent, l'utilisateur est seul à décider quelles seront les opérations qui seront annulées. Aucun processus automatique n'intervient dans ce choix et l'utilisateur en conserve donc la pleine maîtrise. Il en résulte l'absence totale d'ambiguïté quand au résultat qui sera obtenu.

Discussion sur les politiques d'annulation

Le problème qui apparaît clairement avec le mode d'annulation chronologique inverse lorsqu'il est associé à une portée globale est que l'utilisateur risque de générer une opération (en l'occurrence l'annulation d'une opération) qui ne correspond pas à son intention. Une façon de parer à ce problème pourrait être de proposer une demande de confirmation de l'opération. Cependant, je pense que cela serait alors redondant avec le mode de sélection libre et confirmerait donc le peu d'intérêt d'une telle politique d'annulation. L'annulation simple étant un cas particulier d'annulation chronologique, la même restriction s'applique à celle-là.

Portées de l'annulation	*Modes d'annulation*		
	Simple	Chronologique	Sélection libre
Locale	Oui	Oui	Oui
Globale	Non	Non	Oui

Tab. 5.1 – Conformité des politiques d'annulation avec le respect des intentions

En résumé, nous avons vu que les politiques d'annulation sont obtenues par la combinaison d'un mode d'annulation et d'une portée d'annulation. Nous avons vu que dans le cadre d'une application collaborative il y avait deux portées possibles : celle qui concerne uniquement les opérations locales (Portée locale) et celle qui concerne les opérations locales et distantes (Portée globale). Nous avons examiné les deux principaux modes d'annulation : chronologique inverse et par sélection libre. Nous avons précisé également que l'annulation simple pouvait être vue comme un cas particulier de l'annulation chronologique inverse. Finalement nous avons montré que du fait de la concurrence, certaines combinaisons du mode et de la portée d'annulation n'étaient pas souhaitables. En effet, elles peuvent conduire à des résultats qui ne respectent pas les intentions des utilisateurs. Le tableau 5.1 donne une vue synthétique des différentes combinaisons et de leur conformité avec le respect des intentions des utilisateurs.

5.4 Situations problématiques dues à l'annulation

5.4.1 Introduction

Dans la suite, on se place dans le cadre d'une application répartie dans laquelle il existe une copie de chaque objet partagé sur chacun des sites. Cette section présente uniquement les exemples de situations qui posent problème lorsque l'annulation, utilisée dans un algorithme basé sur les transpositions, n'est pas traité de manière spécifique. Ces situations problématiques (*puzzle*) mises en évidence dans la littérature, ont notamment été identifiées dans [Ber94, PK94, RG99, Sun00]. Ces exemples reposent sur l'utilisation de la politique d'annulation la plus simple. Il s'agit de la politique d'annulation qui met en œuvre la portée locale et le mode d'annulation simple.

Dans la suite, on considère un objet dans l'état O_i sur lequel l'utilisateur a exécuté l'opération locale *op*. Pour annuler cette opération, deux cas sont à considérer :

- soit *op* est la dernière opération exécutée et dans ce cas il suffit de générer et d'exécuter l'opération inverse, notée $\overline{op}$;
- soit une séquence *seq* d'opérations distantes ont été reçues et exécutées après *op*. Dans ce cas, on peut considérer que l'opération inverse $\overline{op}$ générée sur l'état résultant $O_i.op$ est concurrente à la séquence *seq*. Il faut donc la transposer en avant, avant de l'exécuter.

Ces considérations conduisent à l'algorithme suivant que nous appellerons l'algorithme naïf d'annulation (fig. 5.8).

Algorithme naïf d'annulation

1. Générer l'inverse $\overline{op}$ de l'opération à annuler *op*, sur l'état résultant de l'exécution de l'opération qu'on annule, de sorte que $O_i.op.\overline{op} = O_i$.
2. Transposer en avant cette opération par rapport à la séquence *seq* des opérations qui ont été exécutées sur le site postérieurement à celle qu'on annule. On obtient : $\overline{op}^{seq}$.
3. Exécuter la transposée obtenue : $\overline{op}^{seq}$.
4. Inclure cette opération dans l'histoire.
5. Diffuser l'opération aux sites distants.

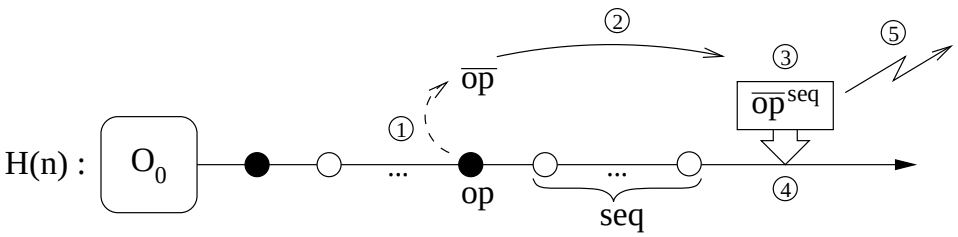


FIG. 5.8 – Modèle naïf d'annulation

Les cas critiques qui sont analysés dans la suite sont basés sur le modèle d'annulation précédent. Nous montrerons dans le chapitre suivant en quoi il est insuffisant.

5.4.2 Situation d'ordre perdu

On se trouve dans une situation d'ordre perdu (*Order Puzzle*) [RG99] (Fig. 5.9) lorsque l'annulation de deux opérations consécutives donne un résultat incorrect parce que l'ordre original des opérations n'a pas pu être maintenu.

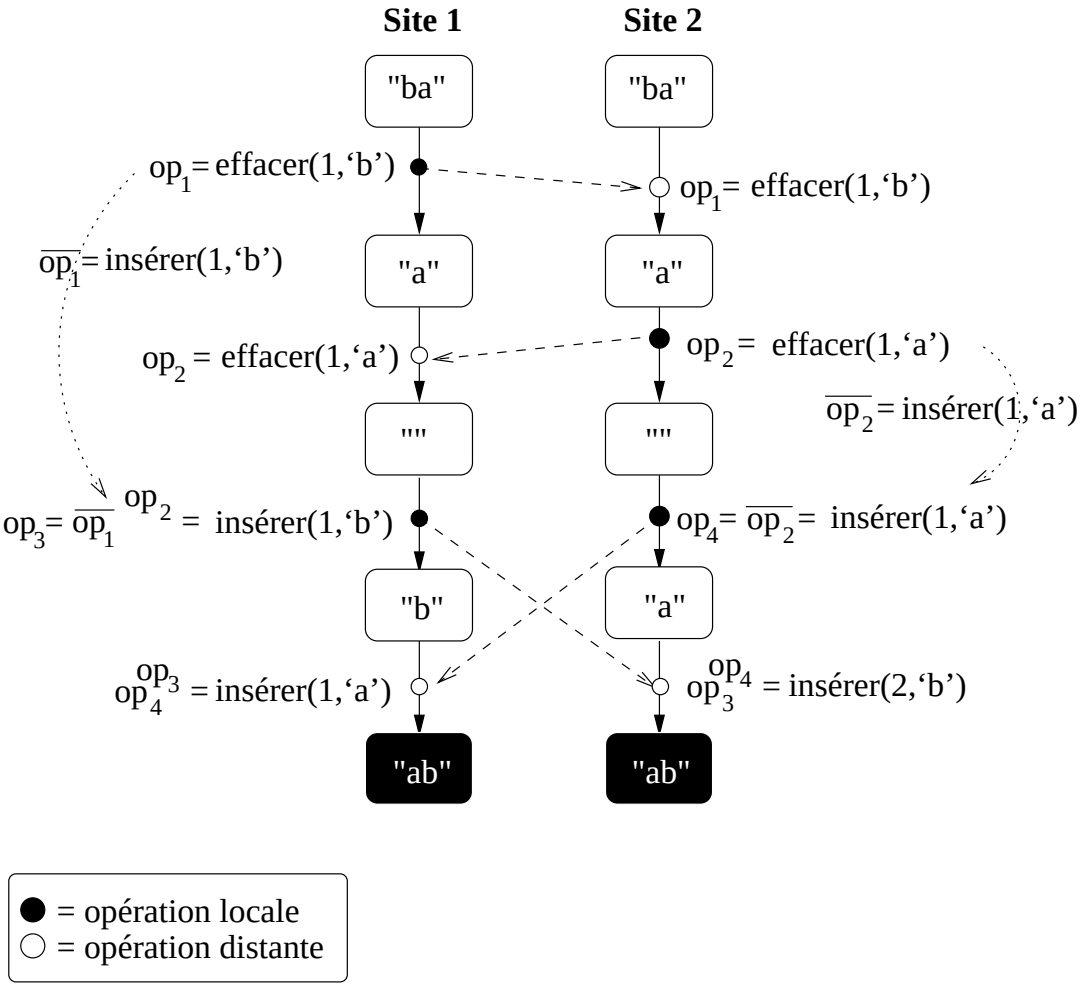


FIG. 5.9 – Situation d'ordre perdu

Considérons la situation suivante. On manipule un objet de type chaîne de caractères ayant la valeur initiale "ba". Cet objet est partagé par deux utilisateurs. L'utilisateur 1, sur le site 1, génère et exécute l'opération suivante : op_1 = `effacer`(1, 'b'). Cette opération est diffusée au site 2, où se trouve le deuxième utilisateur, pour y être exécutée. Ultérieurement, l'utilisateur 2, sur le site 2, génère et exécute op_2 = `effacer`(1, 'a'). Cette opération est alors transmise au site 1 où elle est exécutée. L'état de la chaîne de caractère est maintenant "" (c.-à-d. la chaîne vide). Les deux utilisateurs, décident d'annuler leur dernière opération, et ce de manière concurrente. L'état du document après les deux annulations devrait en toute logique être "ba" (c.-à-d. l'état initial de l'objet avant les deux premières opérations op_1 et op_2). Cependant, on remarquera que l'algorithme naïf

d'annulation conduit ici à restaurer un état incorrect, "ab"[9] dans la mesure où il utilise la transposition en avant dont on on va montrer dans la suite qu'elle n'est pas adaptée aux opérations d'annulation.

Le problème ici est qu'à la fois les intentions de l'utilisateur 1 et de l'utilisateur 2 ne sont pas respectées. L'intention de l'utilisateur 1 n'est pas respectée lorsque son opération op_3 est transposée en avant par rapport à op_4 sur le site 2. Et l'intention de l'utilisateur 2 n'est pas respectée lorsque son opération op_4 est transposée en avant par rapport à op_3 sur le site 1. Cette violation de l'intention dans la transposition en avant met en évidence son inadéquation pour traduire l'annulation d'une opération.

5.4.3 Annulation d'une opération réalisant la même intention qu'une opération concurrente

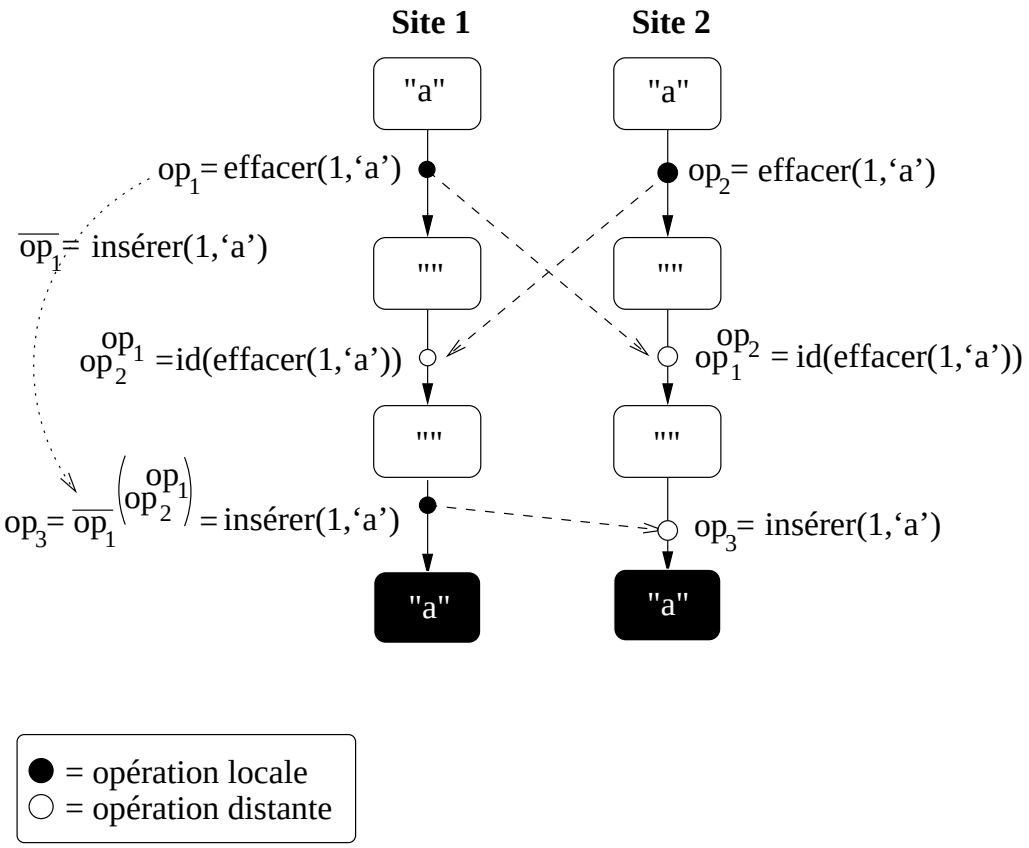


FIG. 5.10 – Annulation d'une opération réalisant la même intention qu'une opération concurrente

L'annulation d'une opération réalisant la même intention qu'une opération concurrente (*Delete-Delete-Undo Puzzle* [RG99]) (Fig. 5.10) est une situation dans laquelle deux opérations traduisant la même intention ont été exécutées en concurrence et où l'une

[9] L'ordre arbitraire pour deux insertions à la même position est basé sur l'ordre alphabétique des caractères insérés. L'ordre étant arbitraire, on peut toujours construire de telles configurations problématiques.

d'elles est ultérieurement annulée. Lors de l'intégration d'une opération, sa transposition en avant par rapport à une autre opération concurrentes réalisant la même intention a pour résultat l'opération identité, `id(...)`. Il en résulte qu'après l'intégration des deux opérations concurrentes, on est dans une situation où deux intentions identiques sont en réalité réalisées par l'exécution d'une seule opération. L'annulation de cette opération risque alors d'entraîner l'annulation de toutes les opérations traduisant la même intention. C'est notamment le cas avec l'algorithme naïf.

En effet, considérons la situation suivante. On dispose d'un objet partagé entre deux utilisateurs. L'état initial de l'objet est "a". Les deux utilisateurs génèrent et exécutent en concurrence la *même* opération `effacer`(1, 'a'). L'annulation d'une quelconque des deux opérations par l'utilisateur qui l'a générée, risque de produire un résultat incorrect, car ne tenant pas compte de l'intention de l'autre utilisateur. C'est le cas ici quand l'utilisateur 1 décide d'annuler l'opération op_1. L'algorithme naïf, à partir de op_1, génère $\overline{op_1}$ qui est ensuite transposée en avant par rapport à l'opération $op_2^{op_1}$. Le résultat est l'opération op_3 dont l'exécution ne respecte pas l'intention de l'utilisateur 1. En effet, bien qu'on puisse considérer le résultat comme celui escompté, il y a bien violation de l'intention. L'intention de l'utilisateur 1 est d'annuler son opération op_1. Or le résultat obtenu après exécution de op_3 est tel que op_1 et op_2 sont toutes deux annulées. Ce n'est manifestement pas l'intention de l'utilisateur 1. Ce problème est encore plus manifeste sur le site 2. En effet, annuler une opération op consiste à exécuter l'opération $\overline{op}$ sur l'état résultant de l'exécution de op. Or sur le site 2, op_3 qui est exécutée pour annuler l'opération $op_1^{op_2}$, n'est pas l'inverse de celle-ci.

On remarquera cependant que l'annulation ultérieure de l'opération op_2 par l'utilisateur 2 produirait sur les sites 1 et 2 un état tel que :
- la convergence des copies est assurée,
- les intentions des deux usagers sont respectées.

En effet, la transposition de l'opération $\overline{op_2}$ par rapport à $op_1^{op_2}$ puis op_3 fournira comme résultat l'opération $op_4 =$ `id`(`insérer`(1, 'a')). Après son exécution, l'état de l'objet aura pour valeur "a" ce qui reflète bien l'annulation des *deux* opérations d'effacement.

5.4.4 Situation de fausse concurrence

La fausse concurrence (*Insert-Insert-Tie-Dilemma* [Sun00]) (Fig. 5.11) est une situation dans laquelle l'annulation, si elle est considérée comme une opération concurrente *classique*, va entraîner un résultat incorrect lors de l'emploi des transpositions.

Considérons l'exemple suivant. Un objet ayant pour valeur initiale "a" est partagé entre deux utilisateurs. Sur le site 1, l'utilisateur 1 génère et exécute l'opération $op_1 =$ `effacer`(1, 'a'). Concurremment, l'utilisateur 2, sur le site 2, génère et exécute $op_2 =$ `inserer`(1, 'b'). Les deux opérations sont diffusées aux sites distants respectifs pour y être exécutées. Après livraison, les opérations sont donc transposées en avant par rapport aux opérations locales puis exécutées. L'état des copies de l'objet sur les deux sites est maintenant "b". Sur le site 1, l'utilisateur 1 décide alors d'annuler son opération $op_1 =$ `effacer`(1, 'a'). L'algorithme naïf génère donc l'opération $\overline{op_1} =$ `insérer`(1, 'a') qui est

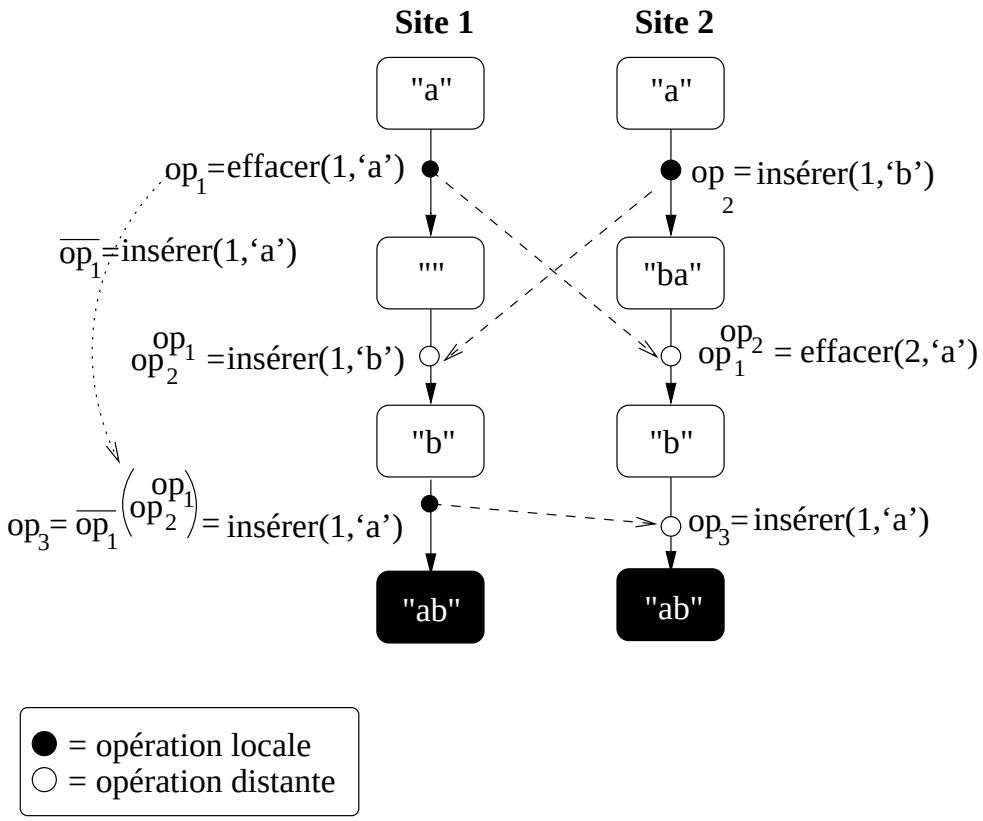


FIG. 5.11 – Situation de fausse concurrence

ensuite transposée par rapport à $op_2^{op_1}$. Cette dernière opération va conduire à un résultat incorrect dans la mesure où l'exécution de $\overline{op_1}^{(op_2^{op_1})}$ conduit à l'état "ab" qui ne correspond pas aux intentions de l'utilisateur 1. En effet, son intention est d'annuler son opération op_1 ce qui devrait avoir pour effet de conduire à l'état "ba". On remarquera cependant que la convergence des copies est assurée malgré tout.

5.4.5 Situation de référence ambiguë

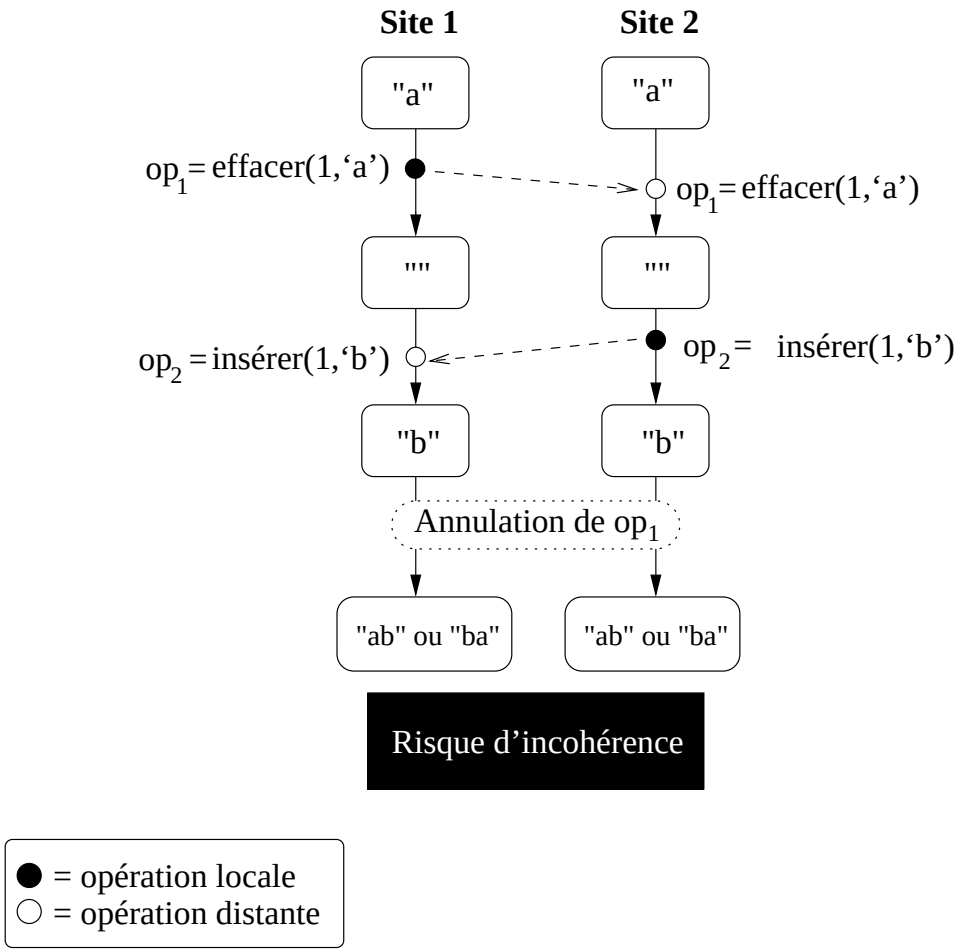


FIG. 5.12 – Situation de référence ambiguë

Une situation de référence ambiguë (*Ambiguous Reference* [Ber94]) est une situation dans laquelle l'annulation d'une opération peut conduire à l'obtention de plusieurs états qui sont tous valides. En fait, elle correspond à l'expression de plusieurs intentions conflictuelles pouvant être résolues de manières différentes et toutes acceptables. Il faut donc faire un choix parmi ces états. Il en résulte qu'il est alors nécessaire pour obtenir la convergence des copies que le même choix soit effectué sur tous les sites.

Considérons l'exemple suivant. Un objet ayant pour valeur initiale "a" est partagé entre deux utilisateurs. Sur le site 1, l'utilisateur 1 génère et exécute l'opération $op_1 =$

`effacer`(1, 'a'). Cette opération est exécutée localement et transmise aux site 2, distant. Après que op_1 ait été reçue et exécutée, sur le site 2, l'état de l'objet est maintenant "" (c.-à-d. la chaîne vide) sur les deux sites. Supposons que l'utilisateur 2, sur le site 2, génère et exécute op_2 = `insérer`(1, 'b'). Cette opération est transmise au site 1 distant pour y être exécutée. Après avoir été exécutée, supposons que l'utilisateur 1, décide d'annuler sa dernière opération. L'état de l'objet après l'annulation peut tout aussi bien être "ba" que "ab". En effet, on ne peut pas dire si l'opération op_2 = `insérer`(1, 'b') correspond à l'insertion avant ou après le 'a' qui avait été au préalable effacé. On se retrouve en fait dans le cas où on a eu deux insertions concurrentes à la même place. Le point délicat ici est donc de s'assurer que le même choix sera fait sur tous les sites qui se trouveront dans cette situation afin d'assurer la cohérence des copies.

6

Conditions nécessaires à l'annulation

Sommaire

6.1 Introduction

Dans cette section, on rappellera dans un premier temps les conditions nécessaires au traitement des opérations d'une application collaborative utilisant un algorithme de type SOCT2 basé sur les transformées opérationnelles. Nous verrons ensuite quelles sont les contraintes nouvelles apportées par la prise en compte de l'annulation. Les conditions nécessaires au traitement correct de l'annulation par les algorithmes utilisant les transformées opérationnelles pourront alors être mise en évidence.

L'analyse des exemples de situations dites problématiques qui ont été présentées à la section précédente permettra de mettre en évidence les conditions qui n'ont pas été respectées dans chacun des cas.

6.2 Propriétés de l'annulation

Le principe de base de l'annulation est le suivant. Pour annuler une opération op il faut exécuter l'opération $\overline{op}$ sur l'état résultant de l'exécution de op. De plus certaines conditions sont nécessaires au bon déroulement des algorithmes utilisant les transformées opérationnelles lorsqu'on veut traiter l'annulation. On peut les définir de la manière suivante.

Soit une séquence $seq = op_1.op_2.\ldots.op_n$ réalisant les intentions $I_1, I_2, \ldots I_n$. Si op_1 et op_n sont telles que l'intention I_n est d'annuler I_1, alors en posant $seq' = op'_2.op'_3.\ldots.op'_{n-1}$, où op'_i réalise l'intention I_i, alors les propriétés suivantes doivent être vérifiées.

1. Les effets sur l'objet de l'exécution de seq doivent être identiques aux effets sur l'objet de l'exécution de seq' : $O_0.seq \equiv O_0.seq'$.
2. Les effets sur une opération op de la séquence seq lors d'une transposition en avant[10] doivent être identiques aux effets sur l'opération op de la séquence seq' lors de la même transposition : $op^{seq} = op^{seq'}$.

Ces propriétés relatives à l'annulation vont conditionner le traitement de l'annulation par les algorithmes basés sur la transformation d'opérations. Deux nouvelles conditions vont donc, à l'instar des conditions C1 et C2 pour les opérations *classiques*, déterminer les règles d'un traitement correct de l'annulation. Ne pas tenir compte de ces nouvelles conditions conduit aux situations problématiques décrites dans la section précédente. C'est ce que nous allons montrer maintenant. Nous verrons également comment le respect de ces conditions permet d'éviter ces situations problématiques.

Nous rappellerons rapidement dans un premier temps les conditions C1 et C2. Cela nous permettra de mettre en évidence le fait que certaines situations problématiques dans le cadre du traitement de l'annulation ont une solution si on tient compte des conditions C1 et C2.

[10] L'emploi de la transposition *en avant* n'est pas restrictif. La transposée *en avant* étant utilisée pour définir la transposée *en arrière*, cette dernière hérite par conséquent de cette propriété.

6.3 Conditions d'exécution correcte

On se place dans le cadre d'applications collaboratives réparties. Il faut assurer que l'exécution d'opérations concurrentes sera compatible avec l'obtention de résultats corrects. Comme on l'a déjà précisé, il est admis qu'un résultat sera correct si (1) la causalité est préservée, (2) l'intention des utilisateurs est respectée et (3) la convergence des copies assurée.

Le premier point (c.-à-d. la préservation de la causalité) vise à prévenir les incohérences dues à la violation de la dépendance d'une opération par rapport à une autre. En effet, s'il existe une dépendance réelle d'une opération op_2 par rapport à une opération op_1 ($op_1 \rightarrow op_2$) alors on peut affirmer que op_2 dépend causalement de op_1 ($op_1 \rightarrow_c op_2$). Ainsi, si on préserve la causalité, on respecte la dépendance causale et par conséquent on respecte aussi une éventuelle dépendance réelle.

6.3.1 Respect de la causalité

Le traitement de l'annulation requiert de revoir l'importance qu'on accorde au problème de la conservation de l'ordre causal. En effet, lorsque $op_1 \rightarrow_c op_2$, on doit s'interdire de transposer en arrière le couple d'opérations (op_1,op_2), ceci afin de préserver la causalité entre op_1 et op_2. En toute logique, on devrait donc s'interdire également d'annuler une opération dont dépendent causalement d'autres opérations. En effet, si on s'interdit de transposer en arrière le couple (op_1,op_2) c'est qu'on considère que l'exécution de op_2 peut ne pas avoir de sens si elle n'est pas précédée de l'exécution de op_1. Il apparaîtrait alors paradoxal d'autoriser l'annulation de op_1.

On remarquera que dans l'algorithme proposé dans [RG99] l'annulation d'une opération ne peut survenir que si toutes les opérations locales causalement dépendantes ont été au préalable annulées. Toutefois, ce comportement semble plus dicté par l'impossibilité de calculer une transposée correcte de ce type d'opérations par rapport à l'opération d'annulation, que par le souci de préserver la causalité. En effet, les opérations distantes causalement dépendantes ne sont pas soumises à cette restriction (cf. 7.1.2).

Imposer le strict respect de la causalité oblige, avant d'annuler une opération *op*, à annuler toute opération causalement dépendante de *op*. Il ne s'agit ici bien évidemment pas seulement des opérations locales causalement dépendantes mais également des opérations distantes causalement dépendantes. Si le temps écoulé sur un site distant entre l'exécution de l'opération et la livraison de son annulation est grand (ce qui n'est pas improbable vu la nature répartie du système considéré et la possibilité d'exécutions déconnectées), les opérations concernées par le processus d'annulation peuvent être nombreuses et par là représenter une grande partie du travail des utilisateurs. Il paraît alors inconcevable de requérir l'annulation de toutes ces opérations, comme préalable à l'annulation de l'opération *op*. Une alternative doit être préconisée ; elle passe par le relâchement des contraintes concernant le respect de la causalité. Ainsi dans le cadre du traitement de l'annulation, il

est indispensable d'autoriser l'utilisations des transpositions pour des opérations qui bien qu'ayant une dépendance causale ne présentent pas de dépendance réelle. C'est en effet le seul moyen de parvenir à traiter l'annulation de manière satisfaisante en environnement collaboratif.

6.3.2 Respect de l'intention de l'usager

Dans les systèmes basés sur les transformées opérationnelles, qui ne traitent pas l'annulation, le respect de l'intention est obtenu en définissant des fonctions de transformation spécifiques à la sémantique des opérations. La principale fonction, appelée transposition en avant, permet, étant données deux opérations op_1 et op_2, d'obtenir une opération $op_2^{op_1}$. L'opération $op_2^{op_1}$ est telle que l'intention qu'elle réalise en étant exécutée à partir de l'état résultant de l'exécution de op_1 est identique à l'intention que réalise op_2 en étant exécutée à partir de l'état où elle a été générée.

Le traitement de l'annulation introduit une nouvelle sémantique qui remet en cause le respect des intentions des utilisateurs lorsque les fonctions de transposition sont utilisées sans modification. Nous verrons un peu plus loin quelles sont les conditions nécessaires au traitement correct de l'annulation. Nous allons pour l'instant rappeler l'origine des conditions C1 et C2.

6.3.3 Convergence des copies

Dans le cadre d'un système collaboratif qui ne permet pas l'annulation, les transpositions en avant doivent satisfaire deux conditions. En effet, considérons deux opérations concurrentes op_1 et op_2, générées à partir du même état et exécutées, après transposition en avant, dans un ordre différent sur deux sites distincts (Fig. 6.1). L'exécution de op_1 suivie de l'exécution de $op_2^{op_1}$ sur le site 1 doit produire le même résultat que l'exécution de op_2 suivie de l'exécution de $op_1^{op_2}$ sur le site 2. Cette première condition appelée condition C1 est formellement définie de la manière suivante.

Définition 6.1 (Condition C1)

Soit op_1 et op_2 deux opérations concurrentes définies sur le même état. La transposée en avant vérifie la condition C1 ssi :

$$op_1.op_2^{op_1} \equiv op_2.op_1^{op_2}$$

où $\equiv$ dénote l'équivalence d'état après avoir exécuté chacune des séquences à partir du même état.

La condition C1 assure que le résultat obtenu après l'exécution de deux opérations ne dépend pas de l'ordre dans lequel ces deux opérations ont été exécutées. Si on considère la situation représentée à la figure 6.1, la condition C1 permet d'assurer que l'exécution de op_1 = `insérer(3, 'f')` suivie de l'exécution de $op_2^{op_1}$ = `effacer(4)` donnera le même résultat (c.-à-d. "bufer") sur le site 1 que sur le site 2 l'exécution de op_2 = `effacer(3)`

suivie de $op_1^{op_2}$ = `insérer(3, 'f')`.

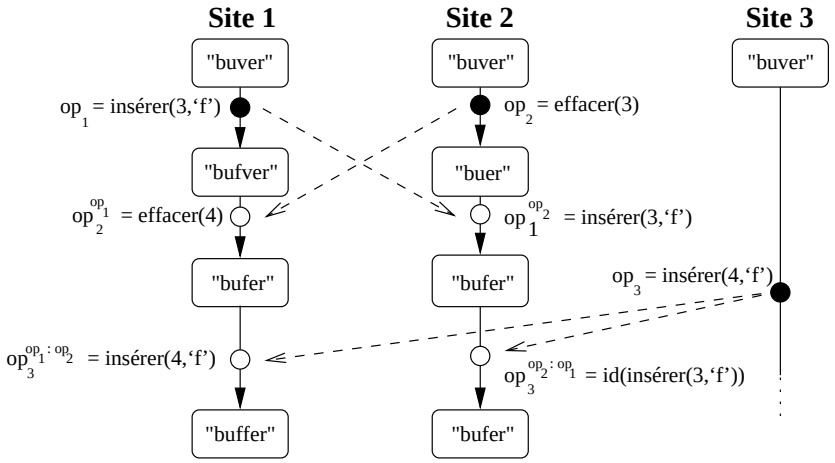


FIG. 6.1 – Convergence des copies

Considérons à présent une opération op_3, générée sur un troisième site – le site 3 – et concurrente à op_1 et op_2. Elle sera à son tour transposée en avant une fois reçue sur les sites 1 et 2. Pour garantir la convergence de toutes les copies, la transposition en avant d'une opération par rapport à une séquence de deux (ou plus) opérations concurrentes ne doit pas dépendre de l'ordre utilisé pour sérialiser ces opérations. Pour cela, en plus de la condition C1, la transposition en avant doit vérifier la condition C2.

Définition 6.2 (Condition C2)

Quelles que soit les opérations op_1, op_2 et op_3, la transposée en avant vérifie la condition C2 ssi :

$$op_3^{op_1:op_2} = op_3^{op_2:op_1}$$

où $op_i : op_j$ dénote $op_i.op_j^{op_i}$.

La condition C2 assure que le résultat d'une transposition d'une opération par rapport à une séquence de deux opérations ne dépend pas de l'ordre suivant lequel ces opérations sont ordonnées dans la séquence. Si on considère la situation représentée à la figure 6.1, on constate que la condition C2 n'est pas respectée. En effet, suite aux transpositions, $op_3^{op_1:op_2}$ = `insérer(4, 'f')` diffère de $op_3^{op_2:op_1}$ = `id(insérer(3, 'f'))` et les copies des sites 1 et 2 se retrouvent dans des états divergents une fois toutes les opérations exécutées. Si la condition C2 avait été respectée cela n'aurait pas était le cas car $op_3^{op_1:op_2}$ et $op_3^{op_2:op_1}$ auraient été identiques.

6.3.4 Respect de l'intention en cas d'annulation

Les systèmes existants qui traitent l'annulation et qui utilisent les transpositions d'opérations [RG99, Sun00] ont fait le choix de considérer l'annulation d'une opération *op* comme l'intention d'exécuter l'opération inverse, $\overline{op}$. Ce mode opératoire permet de se passer de fonctions spécifiques pour la transposition en avant d'une opération d'annulation puisque celle-ci est considérée comme une opération *classique*. Malheureusement, l'utilisation de cette opération inverse $\overline{op}$ entraîne une perte de sémantique. On perd notamment l'information qui précise que cette opération est *l'annulation* de l'opération *op*. Examinons l'exemple suivant pour bien comprendre. Il représente le cas d'une situation de fausse concurrence.

Condition C3 : Neutralité du couple opération/annulation pour la transposition en avant

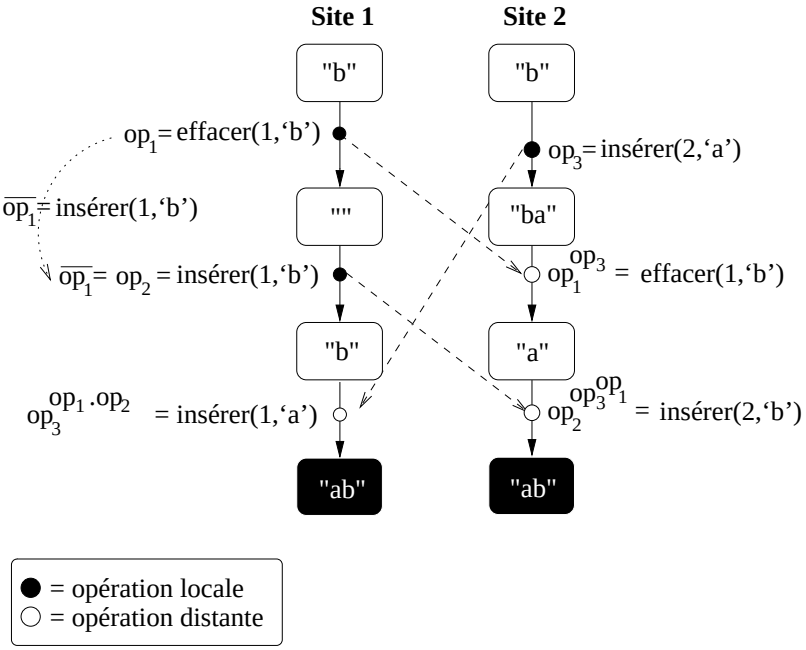


FIG. 6.2 – Situation de non respect de la condition C3

Considérons la figure 6.2. Un objet de type chaîne de caractères est partagé par réplication sur deux sites. L'état initial de l'objet est "b". Sur le site 1, l'utilisateur U_1 génère et exécute l'opération op_1 = `effacer(1, 'b')`. L'état de l'objet est alors "". Puis il décide d'annuler son opération[11] ; pour cela, op_2 = `insérer(1, 'b')` est générée et exécutée ce qui ramène l'état de l'objet à "b". Supposons que concurremment l'utilisateur U_2 sur le site 2 a généré et exécuté l'opération op_3 = `insérer(2, 'a')`. Il l'a ensuite transmise au site 1 pour y être exécutée. Intéressons-nous à ce qui se passe sur le site 1. Y ont été exécutées les opérations op_1 et op_2. L'histoire du site est donc $H_{(2,0)}$[12]= `{effacer(1, 'b'), insérer(1, 'b')}`. Pour réaliser l'intégration de op_3 on va donc devoir la transposer en

[11] L'algorithme utilisé étant l'algorithme de type naïf décrit en 5.4.1

avant par rapport à ces deux opérations. On transpose donc tout d'abord op_3 en avant par rapport à op_1 :

`Transpose_av(effacer(1,'b'),insérer(2,'a')) = insérer(1,'a')`

$op_3^{op_1}$ est ensuite transposée en avant par rapport à op_2 :

`Transpose_av(insérer(1,'b'),insérer(1,'a')) = insérer(1,'a')`[13]

Réaliser l'intention de l'utilisateur U_2 est donc traduit sur le site 1 par l'exécution de l'opération $op_3^{op_1.op_2}$ = `insérer(1,'a')`. L'état final de la copie de l'objet sur ce site est donc "ab". Si l'intention de l'utilisateur U_1 est effectivement respectée puisque la lettre 'b' n'a pas été effacée, il n'en est pas de même en ce qui concerne l'utilisateur U_2. L'intention de celui-ci était de placer un 'a' *après* le 'b'. C'est le contraire qui se produit ici ; le 'a' est placé *avant* le 'b'. Il est manifeste que l'intention de l'utilisateur U_2 n'a pas été respectée lors de cette exécution.

La séquence d'opérations $op_1.op_2$ a entraîné la modification de l'opération op_3 lorsque cette dernière a été transposée en avant par rapport à cette séquence. Or op_2 est l'opération qui réalise l'intention I_2, `annuler l'opération` op_1. La séquence $op_1.op_2$ aurait donc dû agir comme élément neutre pour la transposition en avant de op_3. La transposition en avant aurait donc dû être telle que $op_3^{op_1.op_2} = op_3$ et vérifier ainsi la condition suivante.

Définition 6.3 (Condition C3)

Soit une séquence $seq = op_i.op_{i+1}.\ldots.op_{j-1}.op_j$ *et une séquence* $seq' = op'_{i+1}.\ldots.op'_{j-1}$ *telles que :*
– $\forall k \in [i\,..\,j]$, op_k *réalise l'intention* I_k,
– $\forall l \in [i{+}1\,..\,j{-}1]$, op'_l *réalise l'intention* I_l,
– op_j *soit l'opération qui annule l'opération* op_i,
alors la transposition en avant vérifie la Condition C3 si et seulement si quel que soit op_k *:*

$$op_k^{seq} = op_k^{seq'}$$

Cette condition établit que la transposée en avant d'une opération op par rapport à une séquence seq contenant une opération op_i et son annulation op_j doit être égale à la transposée de op par rapport à la séquence seq modifiée de telle sorte que l'opération op_i et son annulation, op_j n'y apparaissent plus.

La vérification stricte de cette condition permettrait de prendre en compte les deux types d'annulation (c.-à-d. l'annulation explicite et l'annulation implicite). Cependant, en se basant sur l'expérience acquise en démontrant la vérification de la condition C2 pour de petits ensembles d'opérations tels que `{insérer, effacer}`, on peut penser que la vérification de cette condition sera très difficile dans le cas général.

[12] $(2, 0)$ indique ici que les deux premières opérations générées sur le site 1 ont été intégrées et qu'aucune opération du site 2 n'a été intégrée. Cet indice correspond en fait à la valeur du vecteur d'état du site au moment où on observe cette histoire

[13] L'ordre arbitraire utilisé est l'ordre alphabétique. Changer d'ordre ne résoud pas le problème car un exemple ad-hoc peut toujours être construit.

Condition C4 : Validité de la transposition en avant pour les inverses d'opérations

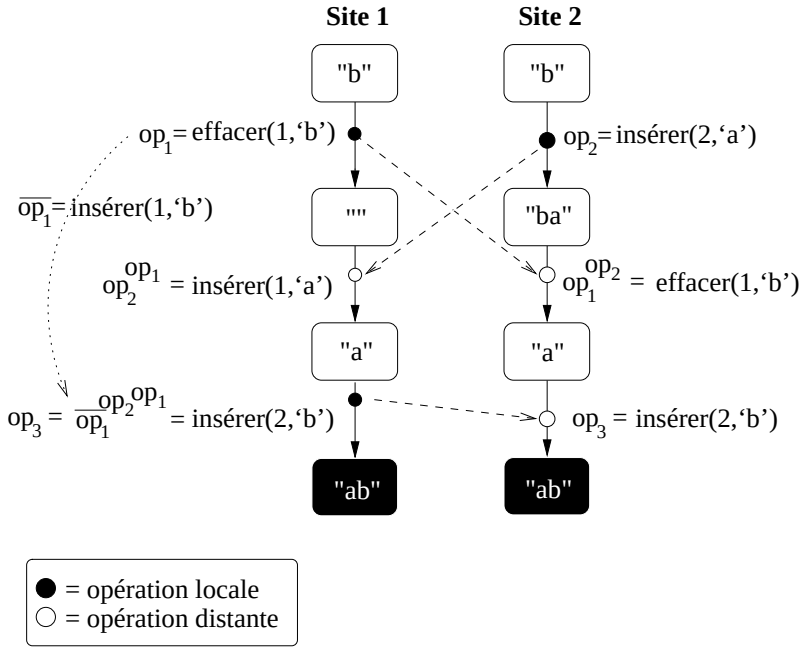


FIG. 6.3 – Une situation de fausse concurrence

Considérons la figure 6.3. Un objet de type chaîne de caractères est partagé par réplication sur deux sites. L'état initial de l'objet est "b". Sur le site 1, l'utilisateur U_1 génère et exécute l'opération op_1 = `effacer(1, 'b')`. L'état de la copie sur le site 1 est alors "" et l'opération op_1 est transmise au site 2. Supposons que sur le site 2 au même moment, l'utilisateur U_2 génère et exécute l'opération op_2 = `insérer(2, 'a')`. L'état de la copie sur le site 2 est donc maintenant "ba". L'opération op_2 est transmise au site 1. À sa livraison sur le site 1, op_2 est transposée en avant par rapport à l'opération op_1, ce qui donne $op_2^{op_1}$ = `insérer(1, 'a')`. L'exécution de cette dernière amène alors l'objet dans l'état "a". Sur le site 2, l'opération op_1 une fois reçue est transposée en avant par rapport à l'opération op_2, ce qui donne $op_1^{op_2}$ = `effacer(1, 'b')`. L'exécution de celle-ci amène l'objet sur le site 2 dans l'état "a". Après avoir exécuté les opérations op_1 et op_2 sur les deux sites, on a donc des états qui ont convergé. De plus, les intentions des utilisateurs ont été respectées. L'utilisateur U_1 voulait effacer le caractère 'b' et l'utilisateur U_2 voulait insérer le caractère 'a'. Seule la condition C1 a été nécessaire pour garantir ce résultat.

Supposons que l'utilisateur décide maintenant d'annuler son opération op_1. Conformément à l'algorithme naïf décrit au chapitre précédent, le système génère l'opération $\overline{op_1}$. Cette dernière est alors transposée en avant par rapport à l'opération $op_2^{op_1}$ ce qui est légitime pusiqu'elles sont en effet toutes deux définies sur le même état. On obtient alors $\overline{op_1}^{op_2^{op_1}}$ = `insérer(2, 'b')`. L'exécution de cette opération ne réalise malheureusement pas l'intention de l'utilisateur U_1. En effet, elle ne permet pas de retourner à l'état correct qui aurait été l'état où seule l'intention de l'utilisateur U_2 aurait été réalisée, c.-à-d. l'état "ba".

L'exécution sur le site 2 apporte une réponse claire au problème. On a dit qu'annuler une opération *op* exécutée sur un objet O dans l'état O_i, revenait à exécuter l'*inverse*[14] de *op* sur l'objet O dans l'état $O_i.op$. Cela veut donc dire que si l'exécution de l'annulation survient immédiatement après l'exécution de *op* alors l'opération qui réalisera cette annulation sera $\overline{op}$. Remarquons que sur le site 2, après avoir exécuté op_1 sous la forme $op_1^{op_2}$, on a exécuté immédiatement après une opération qui réalise l'annulation de op_1. Cette opération aurait donc du être égale à $\overline{op_1^{op_2}}$ = `insérer(1, 'b')`. Ce n'est pas le cas ici ; ce résultat erroné s'explique par le fait que les fonctions de transposition en avant ne sont pas adaptées à l'annulation. Une condition supplémentaire doit être introduite pour garantir un résultat correct.

Définition 6.4 (Condition C4)

Soit une opération op, une séquence seq et une séquence seq′ telle que `Transpose_av`*(op, seq′) = seq alors la transposée en avant vérifie la Condition C4 ssi :*

$$\overline{op}^{seq} = \overline{op^{seq'}}$$

Cette condition exprime le fait que la transposée en avant de l'inverse $\overline{op}$ d'une opération *op*, par rapport à une séquence d'opération *seq* doit être égale à l'inverse de la transposée en avant de *op* par rapport à la séquence *seq′* où la séquence *seq′* est telle que la transposée en avant de *seq′* par rapport à *op* est égale à *seq*.

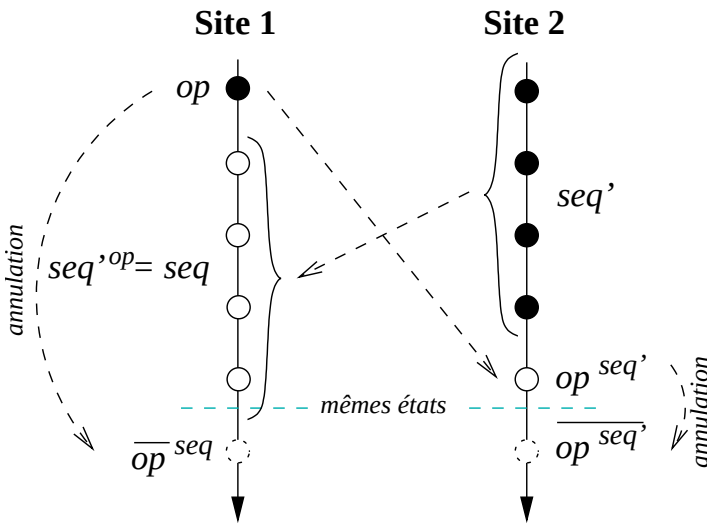


FIG. 6.4 – Illustration de la condition C4

[14] Il va de soi que la possibilité d'annuler dépend de l'existence de cette inverse.

6.4 Analyse critique des exemples de situations problématiques

6.4.1 Situation d'ordre perdu

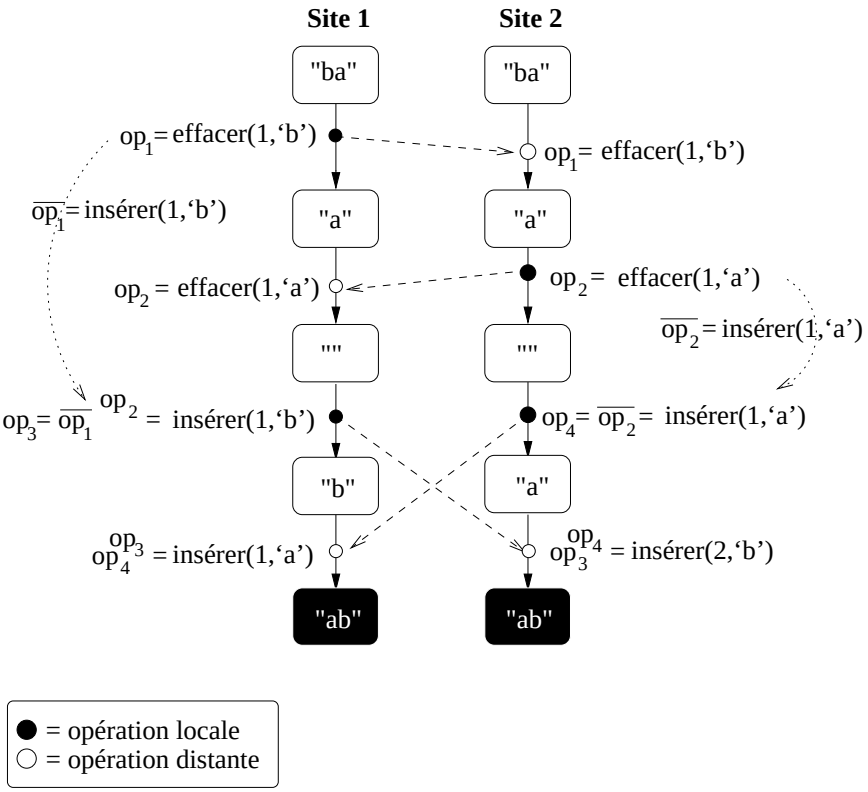


FIG. 6.5 – Situation d'ordre perdu

Plaçons nous dans le cas d'une situation d'ordre perdu (voir Fig. 6.5). Un des utilisateurs, appelons-le l'utilisateur U_1, génère une opération op_1 qui est exécutée localement avant d'être transmise aux sites distants pour y être exécutée. Après exécution sur un des sites distants, le site 2 dans notre exemple, l'utilisateur U_2, génère une opération op_2 qui dépend alors causalement de op_1. Cette opération op_2 est exécutée localement avant d'être transmise aux autres sites, et notamment au site 1, pour y être exécutée également. On a donc $op_1 \rightarrow_c op_2$ (c.-à-d. op_1 précède causalement op_2). À partir de là, on aboutit à une situation d'ordre perdu si les deux utilisateurs décident d'annuler leur opération en concurrence. Remarquons que l'exécution locale de chaque annulation réalise effectivement l'annulation de l'opération correspondante op_1 (resp. op_2) sur le site 1 (resp. 2).

La violation des intentions des utilisateurs se manifeste lorsque op_3 (resp. op_4) est intégrée puis exécutée sur le site 2 (resp. 1). En effet, alors que les résultats issus des exécutions locales des opérations op_3 et op_4 étaient satisfaisants, les exécutions distantes fournissent un résultat qui n'est pas celui attendu. Sur le site 2, on a intégré op_3 en la

transposant en avant par rapport à $\overline{op_2}$ avant de l'exécuter. Comme $op_3 = \overline{op_1}^{op_2}$ et que $op_4 = \overline{op_2}$ alors la transposée en avant de op_3 par rapport à op_4, $op_3^{op_4}$, correspond à $\overline{op_1}^{op_2.\overline{op_2}}$. En notant $seq = op_2.\overline{op_2}$, l'application de la condition C3 impose que $\overline{op_1}^{seq} = \overline{op_1}$. On constate que ce n'est pas le cas ici et c'est justement ce qui provoque la violation de l'intention de l'utilisateur U_1 sur le site 2. **Si la condition C3 avait été vérifiée**, alors $op_3^{op_4} = \overline{op_1}$ aurait été égale à `insérer(1, 'b')` et son exécution aurait conduit à un résultat correct. **Les intentions des deux utilisateurs auraient** ainsi **été préservées sur le site 2**.

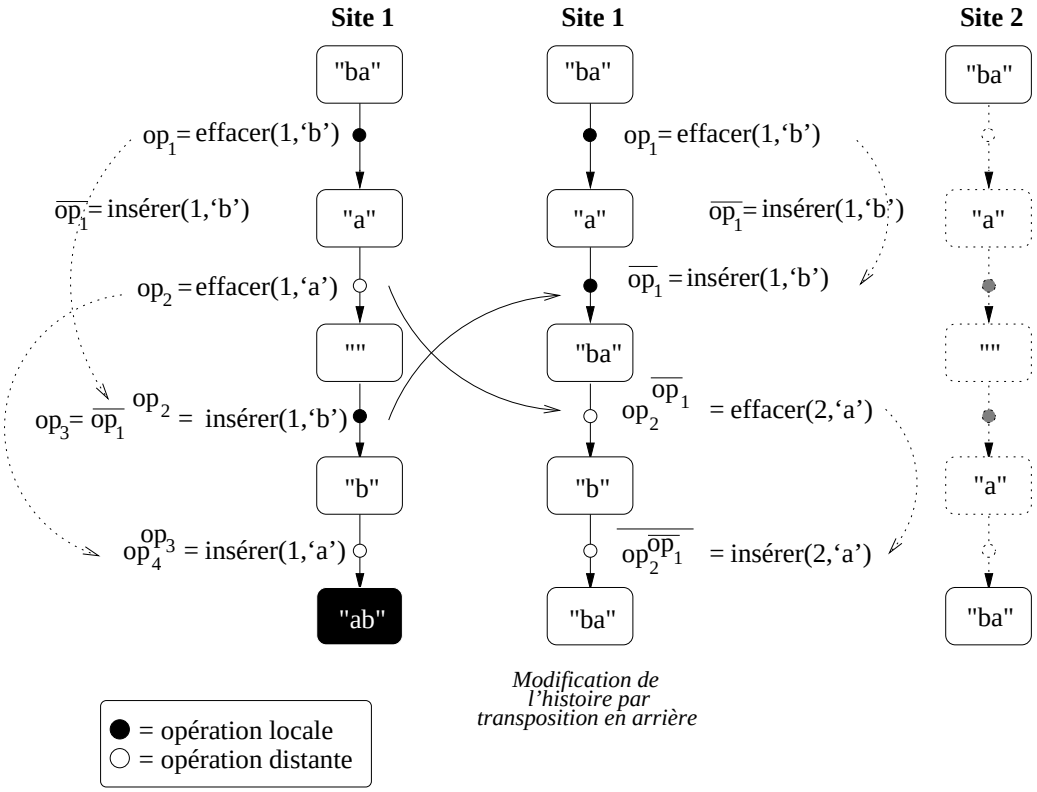


FIG. 6.6 – Mise en évidence de la condition C4 dans la situation d'ordre perdu

Sur le site 1, on obtient le même résultat que sur le site 2. La convergence des copies est donc assurée. Malheureusement cette convergence se fait sur un résultat qui, on vient de le dire n'est pas correct. Pourtant ici, on n'a pas eu à faire de transposition dans laquelle intervient la condition C3. En effet, on a intégré $op_4 = \overline{op_2}$ en la transposant en avant par rapport à $op_3 = \overline{op_1}^{op_2}$. On a donc calculé $\overline{op_2}^{(\overline{op_1}^{op_2})}$ pour obtenir `insérer(1, 'a')`. Puisque U_2 avait décidé d'annuler l'opération op_2, on aurait pu envisager de transposer en arrière op_2 et op_3 (Fig. 6.6). On aurait obtenu comme résultat le couple ($\overline{op_1}$ = `insérer(1, 'b')`, $op_2^{\overline{op_1}}$ = `effacer(2, 'a')`). Dans ces conditions, il apparaît alors évident que l'annulation de op_2 aurait été obtenue en exécutant $\overline{op_2^{\overline{op_1}}}$ = `insérer(2, 'a')`. Donc, dans ce cas, on peut affirmer que la transposition en avant qui a permis d'obtenir $op_4^{op_3}$ = `insérer(1, 'a')` ne respecte pas la condition C4, sinon on aurait du effectivement avoir égalité entre $op_4^{op_3} = \overline{op_2}^{(\overline{op_1}^{op_2})}$ et $\overline{op_2^{\overline{op_1}}}$. Et le résultat obtenu aurait alors été le même que

celui que l'on aurait obtenu sur le site 2 si la condition C3 avait été respectée.

6.4.2 Annulation d'une opération réalisant la même intention qu'une opération concurrente

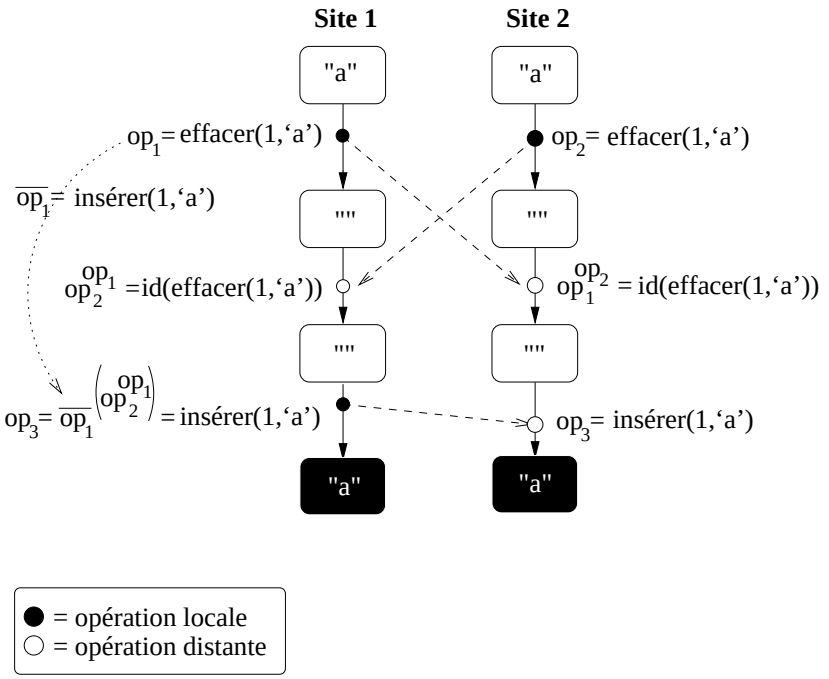


FIG. 6.7 – Annulation d'une opération réalisant la même intention qu'une opération concurrente

Dans cette situation (*Delete-Delete-Undo puzzle*), deux utilisateurs expriment à un moment donné des intentions identiques, c.-à-d. qu'ils génèrent deux opérations identiques, op_1 et op_2 (Fig. 6.7). Lors de l'intégration, la transposition en avant d'une de ces opérations par rapport à l'autre aura pour résultat une opération identité (c.-à-d. de la forme *id(...)*). À partir de là, on aboutit à une situation critique lorsque sur un site, un des utilisateurs demande l'annulation de son opération. Cette dernière, du fait de l'exécution immédiate, a été exécutée sans modification ; c'est le cas de op_1 sur le site 1 et de op_2 sur le site 2. Seule l'opération distante est transformée en une opération identité ; c'est le cas de op_2 sur le site 1 et de op_1 sur le site 2. Supposons que ce soit l'utilisateur sur le site 1 qui réclame l'annulation de son opération, op_1.

L'algorithme naïf, impose de transposer l'inverse de l'opération op_1 par rapport à $op_2^{op_1}$. Cette dernière est une opération identité. Or la transposition en avant par rapport une opération identité ne modifie pas l'opération. On va donc exécuter, pour annuler op_1, l'opération $\overline{op_1}^{op_2^{op_1}}$ = `insérer(1, 'a')` ce qui annule du même coup l'opération op_2. En effet, l'effet de cette dernière ne se reflète plus sur l'état de l'objet obtenu après l'annulation de op_1. Ce faisant, on ne respecte pas la première propriété énoncée en début de chapitre, à

savoir qu'une opération annulée ne doit pas avoir d'effet sur les actions des autres utilisateurs. L'anomalie constatée provient du fait que la transposition en avant ne respecte pas la condition C4. Cet exemple est intéressant de ce point de vue car en observant ce qui se passe sur le site 2, on peut retrouver l'opération qui aurait du être exécutée sur le site 1 pour annuler op_1. En effet, sur le site 2, op_1 est transposée en avant par rapport à op_2 et exécutée après celle-ci. La dernière opération exécutée sur ce site est donc celle qui réalise l'intention associée à l'opération op_1 sur le site 1. Il suffit donc pour annuler op_1 sur le site 2 d'exécuter l'inverse de cette opération, c.-à-d. $\overline{op_1^{op_2}}$ = `id(insérer(1, 'a'))`. **Si la transposition en avant** utilisée pour obtenir $\overline{op_1}^{op_1}_{op_2}$ **avait respecté la condition C4 alors** $\overline{op_1}^{op_1}_{op_2}$ aurait été égale à $\overline{op_1^{op_2}}$ et **le résultat obtenu**, "", **aurait été correct sur les deux sites**.

Un raisonnement similaire peut être tenu pour ce qui concerne l'annulation de l'opération op_2 par l'utilisateur opérant sur le site 2. La vérification de la condition C4 par la fonction de transposition en avant assurera un retour à l'état initial de l'objet après que op_2 ait à son tour été annulée, en l'absence de tout autre génération et exécution d'opération.

6.4.3 Situation de fausse concurrence

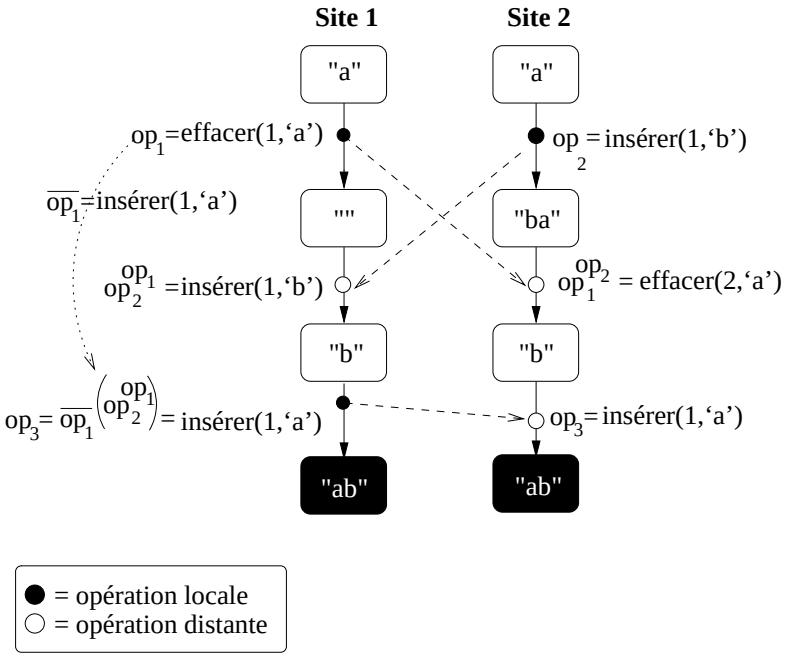


FIG. 6.8 – Situation de fausse concurrence

Une situation de fausse concurrence (*Insert-Insert-Tie Dilemma*) constitue également une situation où l'ordre initial est perdu à la suite des transformations d'opérations. On considère deux opérations concurrentes, op_1 et op_2 qui ont été générées respectivement

sur les sites 1 et 2. Après avoir été exécutées localement, elles sont transmises aux sites distants pour y être exécutées. Sur les deux sites, la situation est la suivante.

- Sur le site 1, l'utilisateur a généré op_1 qui a été immédiatement exécutée. Ce dernier a ensuite reçu l'opération op_2 du site 2, distant. Elle a été intégrée à l'histoire du site, c.-à-d. transposée en avant par rapport à op_1 avant d'être exécutée.
- Sur le site 2, l'utilisateur a généré op_2 qui a été immédiatement exécutée. Ce dernier a ensuite reçu l'opération op_1 du site 1, distant. Elle a été intégrée à l'histoire du site, c.-à-d. transposée en avant par rapport à op_2 avant d'être exécutée.

La situation critique apparaît lorsque l'utilisateur qui a généré op_1 va requérir l'annulation de cette dernière. En effet, comme pour le cas précédent et c'est de là que vient la similitude, l'opération op_1 qui doit être annulée n'est pas la dernière à avoir été exécutée sur le site. Générer et exécuter son inverse directement ne suffit donc pas. Il faut aussi tenir compte de l'exécution ultérieure de $op_2^{op_1}$.

Nous avons déjà analysé cette situation, lors de la présentation de la condition C4, en 6.3.4. L'algorithme naïf propose de transposer en avant $\overline{op_1}$ par rapport à $op_2^{op_1}$. Ce mode opératoire nécessite cependant, comme on l'a vu dans le cas précédent que la transposition en avant respecte la condition C4. Ce n'est pas le cas ici, ce qui explique le résultat incorrect obtenu. Assurer **la vérification de la condition C4, permet d'obtenir** l'opération $\overline{op_1}^{op_2^{op_1}} = \overline{op_1^{op_2}} = \texttt{insérer}(2, \text{'a'})$ dont l'exécution donne **le résultat correct**. C'est d'ailleurs l'opération qui réalise l'annulation de op_1 sur le site 2.

6.4.4 Situation de référence ambiguë

Le cas de référence ambiguë (*ambiguous reference*) est présenté dans [Sun00] comme faisant partie des cas où l'algorithme d'annulation ne parvient pas à rétablir l'ordre correct de caractères adjacents à la suite d'une annulation[15]. Cette classification paraît tout à fait inexacte. En effet, il n'existe précisément pas d'*ordre correct* dans ce cas précis. Il n'y a donc pas de raison de craindre un quelconque problème d'ordre. Le seul risque est celui d'une possible divergence des copies sur les différents sites.

Cependant, on peut remarquer que cette situation s'apparente à l'expression de deux intentions conflictuelles, l'annulation de op_1 traduite par $\texttt{insérer}(1, \text{'a'})$ et op_2 traduite par $\texttt{insérer}(1, \text{'b'})$. L'annulation de op_1 considérée comme concurrente à op_2 sera intégrée dans l'histoire de chacun des sites en utilisant la transposition en avant. Si cette dernière est bien définie pour assurer la convergence des copies, et en particulier si elle vérifie les conditions C1 et C2 alors le problème qui est évoqué ici n'a pas de raison d'apparaître. On notera que dans les situations précédentes, il n'y a pas de divergence car les conditions C1 et C2 sont bien respectées ; le résultat incorrect obtenu était dû à la non vérification des conditions C3 et C4 par la transposition en avant employée.

[15] "[...] the undo system fails to restore the correct order of adjacent characters in performing undos."

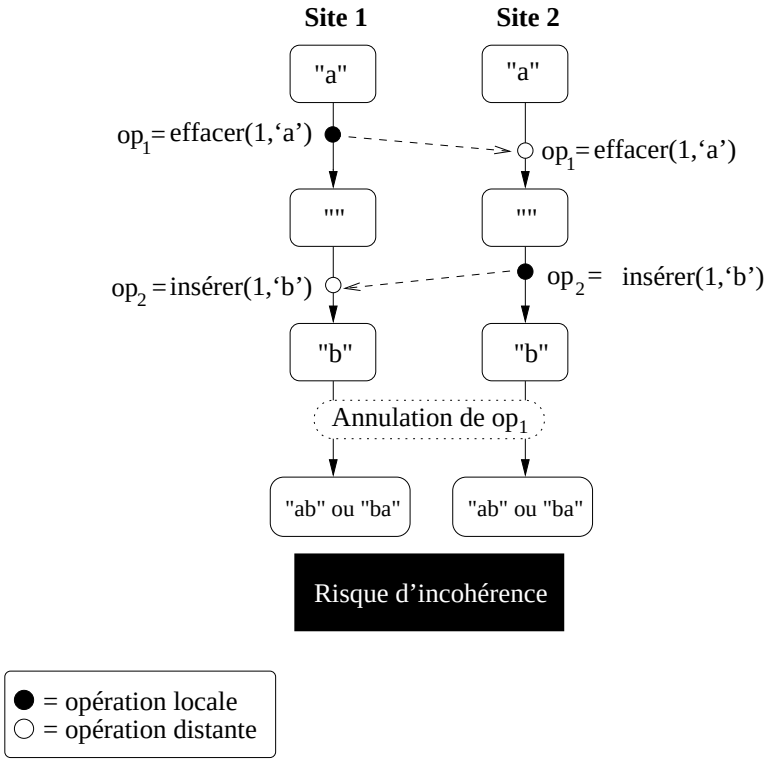


FIG. 6.9 – Référence ambiguë

7

État de l'art

Sommaire

7.1 Solution basée sur l'algorithme adOPTed

7.1.1 Principle de l'annulation dans l'algorithme adOPTed

Dans adOPTed [RG99], le principe du traitement de l'annulation est basé sur la neutralité du couple opération/annulation par rapport à la transposition en avant. Ainsi, le résultat de la transposition en avant d'une opération *op* par rapport à une séquence $op'.\overline{op'}$ est l'opération *op* elle-même.

L'annulation se passe de manière chronologique et concerne uniquement les opérations locales. Lorsqu'un utilisateur demande l'annulation d'une opération, le système détermine quelle est l'opération *op* la plus récente n'ayant pas encore été annulée. Puis en utilisant la fonction auxiliaire `mirror`, étudiée dans la suite, le système fournit une opération réalisant l'annulation de *op* sur l'état courant de la copie. L'opération est ensuite diffusée à tous les sites pour y être exécutée. Le processus d'exécution, similaire pour tous les sites, est le suivant.

Étape a) Intégration de l'opération qui réalise l'annulation
Celle-ci est réalisée grâce à la fonction `translateRequest` déjà utilisée dans l'algorithme adOPTed. La seule différence vient de l'ajout d'un cas particulier. Celui-ci concerne le cas où, dans l'algorithme, l'opération op_i qui permet d'atteindre l'état O_j à partir de l'état O_i est une opération d'annulation $\overline{op}$. Dans ce cas, on sait que $\overline{op}$ est la dernière opération d'une séquence d'annulation *seq* c.-à-d. que $seq = op.seq_1.\overline{op}$ où seq_1 est aussi une séquence d'annulation. La fonction `fold`, étudiée plus loin, permet alors d'obtenir une transposition correcte par rapport à la séquence d'annulation *seq*. Le résultat de la transposition en avant d'une opération *op* par rapport à une séquence d'annulation *seq* devant être égale à l'opération *op* elle-même.

Étape b) Exécution de l'opération
Une fois l'intégration réalisée, l'opération obtenue est exécutée de façon classique.

Fonctions auxiliaires pour gérer l'annulation L'essentiel du traitement relatif à l'annulation se passe dans la fonction `translateRequest` présentée en annexe de [RG99] et qui sous cette forme semble d'ailleurs incomplète (cf. 7.1.2). La fonction `translateRequest` fait appel à deux fonctions auxiliaires :

- `mirror`(*op*, *j*) : *op* est l'opération que l'on désire annuler et *j* est un entier qui représente le nombre d'annulations successives qu'on a effectuées (y compris celle qu'on traite actuellement). Le résultat de l'appel de mirror est simplement l'opération inverse de *op* à laquelle est associé un nouveau vecteur d'état. Ce dernier est obtenu en ajoutant $(2j-1)$ à la composante du vecteur d'état de *op* représentant l'utilisateur qui a généré *op* et qui demande son annulation. Dans l'exemple de la figure 7.1 si on considère l'annulation de `effacer(1,'a')`, on voit que l'on doit auparavant annuler `insérer(1,'b')`. Lorsqu'on appelle `mirror`, *j* vaut donc 2. L'appel à `mirror` retourne donc l'opération `insérer(1,'a')` à laquelle est associé le vecteur d'état `(0,3) = (0,{0+2×2-1})`.

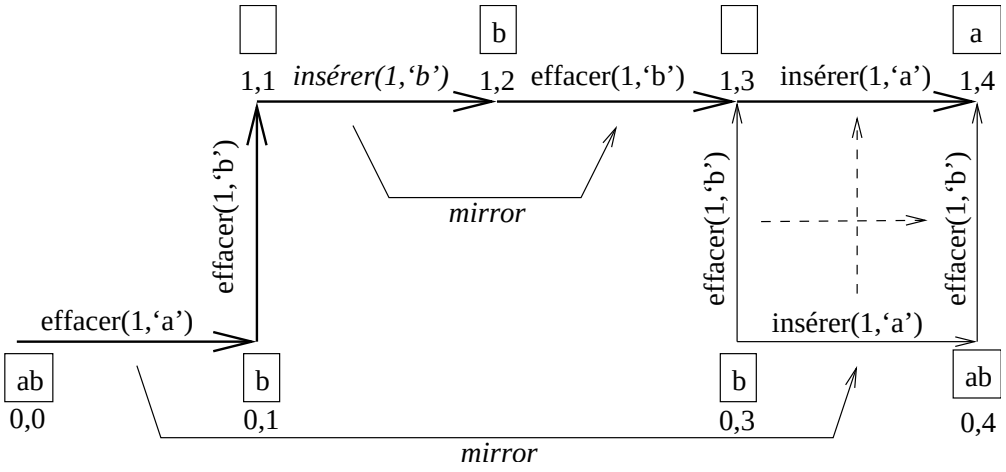


FIG. 7.1 – Opérateur mirror

– `fold`(op, u, j) : op est l'opération que l'on désire transposer par rapport à la séquence d'annulation, u l'index de la composante du vecteur d'état associé à l'utilisateur qui a généré la séquence d'annulation et j le nombre d'annulation dans la séquence. `fold` a pour unique fonction de retourner une copie de l'opération op à laquelle est associé un nouveau vecteur d'état. Ce dernier est une copie du vecteur d'état de op dont la composante $V[u]$ est augmenté de $(2j)$. Dans l'exemple de la figure 7.2, on applique `fold` à l'opération `effacer(1, 'b')` générée par l'utilisateur 2, sur l'état `(0,1)`. On obtient une copie de cette opération avec comme vecteur d'état `(0,3) = (0,{1+2×1})`. La partie droite de la figure permet de comprendre le choix du terme `fold` (plier). En effet, lorsqu'on applique cette fonction tout se passe comme si le graphe était plié de sorte que les vecteurs représentant l'opération et son annulation soient superposés. Les états "avant" et "après" l'exécution du couple opération/annulation se retrouvent également superposés. Il en va de même pour tous les vecteurs ayant pour origine ou pour destination l'un de ces deux états et représentant une opération générée par un utilisateur autre que celui qui a généré l'annulation.

7.1.2 Critique de l'algorithme

Si on regarde en détail le fonctionnement de l'algorithme adOPTed dans [RG99], on s'aperçoit qu'il ne peut fonctionner tel qu'il est décrit.

En effet, considérons l'exemple de la figure 7.3. On se trouve au départ dans l'état `(1,2)`. On veut obtenir l'annulation de l'opération `effacer(1, 'a')` définie sur l'état `(0,0)`. Pour cela, on annule d'abord `insérer(2, 'c')` définie sur l'état `(0,1)`. Puis la fonction `mirror` nous fournit l'opération `insérer(1, 'a')` définie sur l'état `(0,3)`.

On veut maintenant obtenir une opération qui réalise la même intention mais qui est définie sur l'état courant `(1,3)`. Il faut donc transposer `insérer(1, 'a')` par rapport à une opération op qui, définie sur l'état `(0,3)`, amène l'objet dans l'état `(1,3)`. L'état `(1,3)` étant obtenu par l'exécution d'une opération d'annulation, un appel à la fonction `fold` sera effectué. Il devrait permettre d'obtenir une telle opération op. La fonction `fold`

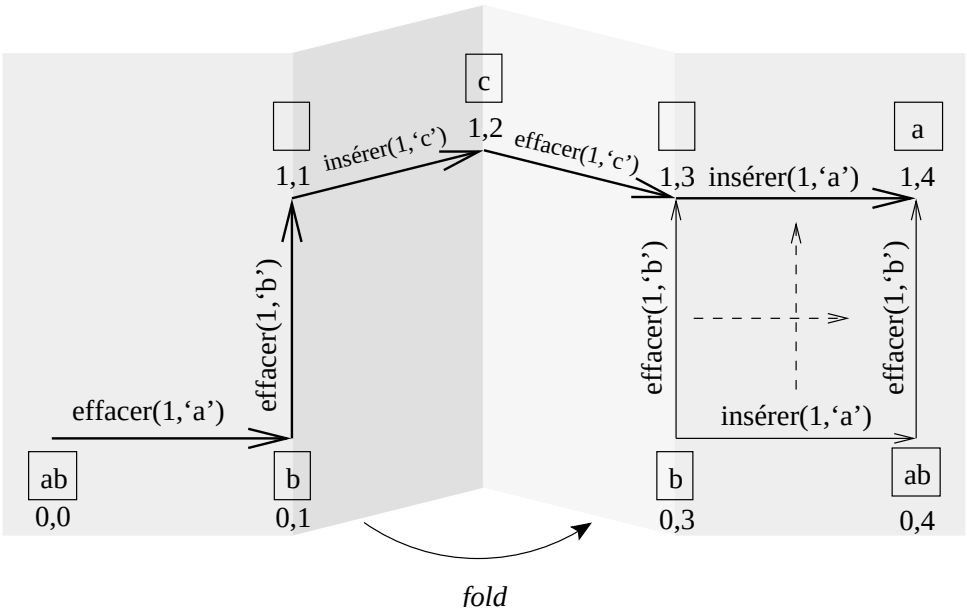


FIG. 7.2 – Opérateur fold

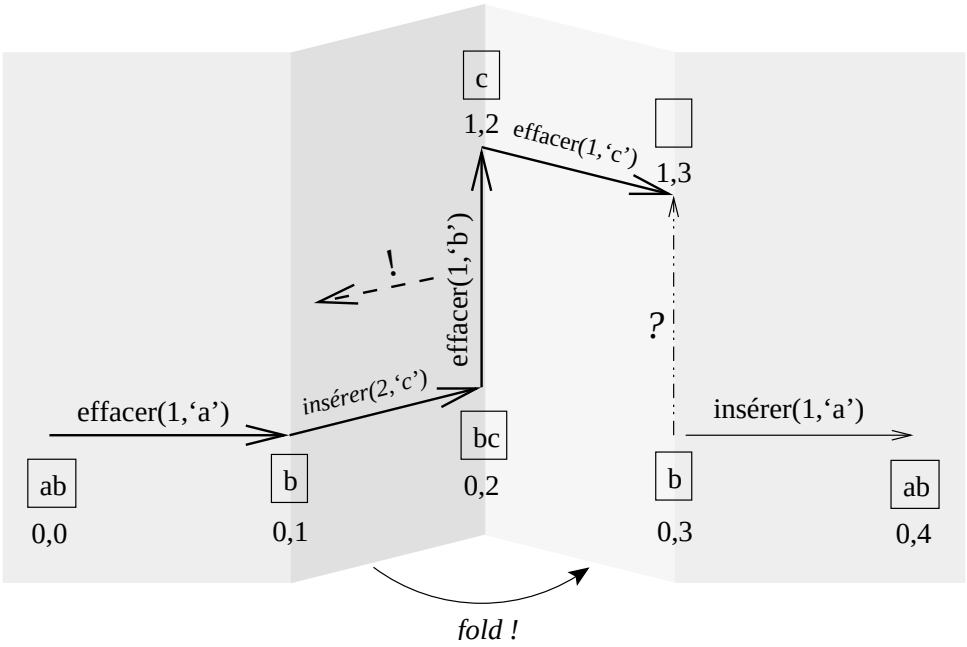


FIG. 7.3 – Impossibilité d'utiliser l'opérateur `fold`

construit *op* en faisant une copie de l'opération qui, définie sur l'état `(0,1)`[16], fait passer l'objet dans l'état `(1,1)`.

Malheureusement, l'opération qui fait passer l'objet d'un état `(0,x)` à un état `(1,x)` est l'opération `effacer(1, 'b')`. Or cette opération est définie sur l'état `(0,2)`. En conséquence, la fonction `fold` ne sera pas en mesure d'obtenir une opération *op* définie sur l'état `(0,1)`. Un contrôle supplémentaire est donc nécessaire lors de l'utilisation de la fonction `fold` en rapport avec une séquence $op.seq.\overline{op}$. On rappelle que dans ce cas, *seq* est aussi une séquence d'annulation puisque l'annulation est chronologique dans adOPTed. Il s'agirait de vérifier, avant son utilisation, que l'opération que l'on recherche ne soit pas la transposée d'une opération dont la génération suit causalement l'opération *op* ; c.-à-d. une opération dont la génération est survenu *dans le pli*.

7.1.3 Conclusion

L'algorithme d'annulation décrit dans [RG99] fournit une solution au problème de l'annulation dans un environnement réparti. La solution est obtenue au prix d'importante restrictions sur les politiques d'annulation permises puisque seule l'annulation chronologique à portée locale est autorisée. Cependant, le problème posé par les opérations distantes causalement dépendantes d'une opération qu'on annule n'y est pas abordé. Ce problème pourrait, dans l'exemple qu'on vient d'évoquer, concerner l'opération `effacer(1, 'b')` qui dépend causalement de l'opération annulée, `insérer(2, 'c')`.

7.2 Solution basée sur l'algorithme GOTO

7.2.1 Principe de l'algorithme ANYUNDO

Le principe de l'algorithme ANYUNDO [Sun00] consiste à *marquer* (symbolisé par la présence de * après le nom de l'opération) une opération annulée et à modifier la fonction de transposition en avant de telle sorte qu'une opération marquée soit considérée comme élément neutre, c.-à-d. :

$$\forall op_i, op_j : Transpose\_av(op_j*, op_i) = op_i$$

Lorsqu'un utilisateur demande l'annulation d'une opération *op*, une opération d'annulation `annuler`(*op*) est générée qui identifie l'opération à annuler. Cette opération est exécutée localement et diffusée sous cette forme aux sites distants. Lors de l'exécution d'une telle opération (p.ex. `annuler`(op_3)) sur son site local ou suite à sa livraison sur un site distant, les traitements suivants sont effectués (cf. Fig. 7.4).

Etape a) Exécution de l'opération qui réalise l'annulation

(1) Génération de l'opération inverse : $\overline{op_3}$.

(2) Transposition en avant par rapport aux opérations qui *suivent* dans l'histoire et avec lesquelles l'opération inverse est concurrente puisqu'elle est générée sur l'état résultant de l'exécution de op_3.

[16] $(0, \{3 - 2 * 1\})$

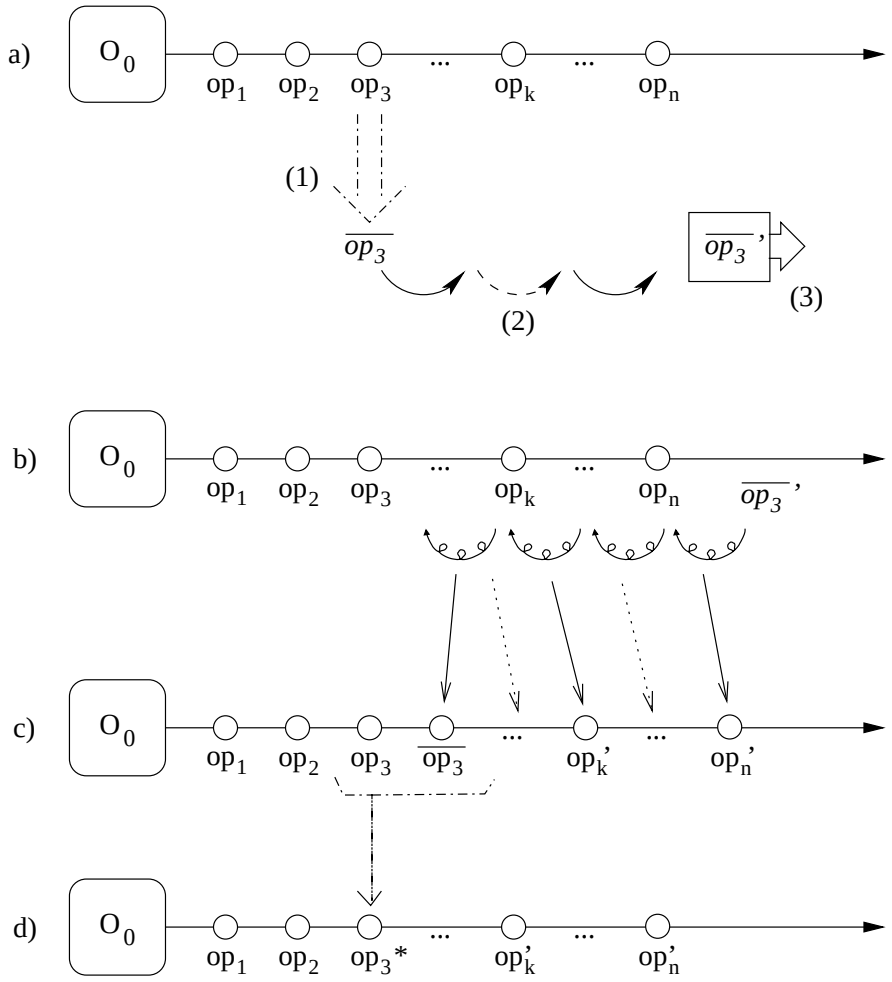


FIG. 7.4 – Principe de l'algorithme ANYUNDO

(3) Exécution de la transposée en avant $\overline{op_3}'$ sur l'état courant de l'objet.

Etape b) Transposition en arrière de $\overline{op_3}'$ par rapport à toutes les opérations qui la séparent, dans l'histoire, de op_3 pour que celles-ci prennent en compte les modifications que $\overline{op_3}$ apporte.

Etape c) Le couple (op_3, $\overline{op_3}$) qui est maintenant adjacent dans l'histoire [17] est contracté pour donner op_3*.

Etape d) L'histoire contient maintenant une opération marquée annulée qui se comportera comme élément neutre pour les transpositions.

7.2.2 Critique de l'algorithme

Il y a deux raisons pour lesquelles l'algorithme ANYUNDO n'est pas correct. La première est due à une perte d'information lors du stockage d'une opération d'annulation dans l'histoire d'un site. La deuxième est due à l'utilisation impropre des fonctions de transposition définies pour d'autres opérations que celles pour lesquelles on les utilise. Ces erreurs dans la conception de l'algorithme ANYUNDO peuvent avoir des conséquences sur l'ordre de caractères adjacents au cours de l'annulation, voire entrainer des divergences entre les différentes copies. Ces problèmes sont présentés dans les deux sections suivantes.

Problème 1. Conséquence de la non datation de l'annulation

Dans cet algorithme, l'annulation d'une opération n'est pas datée, c.-à-d. qu'aucune estampille ne lui est attribuée. Dans l'histoire, la seule information conservée concerne l'annulation de l'opération. Ceci permet lorsqu'on doit transposer une opération par rapport à une séquence d'opération de ne pas tenir compte des opérations de la séquence qui sont marquées comme annulées. Ce mécanisme est cohérent avec le principe qui veut qu'un couple d'opérations *faire/défaire* ne doit pas avoir de répercussion sur les opérations concurrentes (c.-à-d. tout se passe comme si rien ne s'était passé).

Le problème vient du fait qu'une fois réalisée l'annulation d'une opération op_i, si on reçoit une opération op_j qui dépend causalement de op_i on ne peut pas dire si op_j dépend causalement de l'annulation de op_i ou non. Ceci est directement dû au fait que l'annulation est ignorée en tant qu'opération à part entière et que seul est retenu dans l'histoire son effet. La comparaison des vecteurs d'état de op_i et de op_j exprime que op_i précède causalement op_j. L'absence de vecteur d'état associé à l'annulation empêche toute comparaison et laisse non précisée la relation de causalité entre l'annulation de op_i et op_j. L'exemple suivant permet d'illustrer de manière concrète ce problème.

Exemple 7.1 – Non datation de l'annulation.

Considérons un objet de type chaîne de caractères. Cet objet est partagé entre deux sites. Son état initial est "bd". La figure 7.5 illustre l'évolution des

[17] en réalité $\overline{op_3}$ n'est pas réellement présente dans la structure de données qui représente l'histoire

copies de l'objet sur les deux sites en fonction des opérations générées localement et des opérations reçues. À chaque état est associée l'histoire du site. L'état final sur le site 1, noté en noir, indique un problème de cohérence. Il provient de l'intégration incorrecte de l'opération op_2. Cette dernière, dépendant causalement de op_1, sera considérée comme causalement prête à être exécutée par l'algorithme ANYUNDO lorsqu'elle sera reçue. En effet, l'annulation de op_1 n'étant pas estampillée, on ne peut pas savoir si op_2 lui est concurrente. Ainsi op_2 ne sera pas transposée en avant par rapport à l'annulation de op_1. Il en résulte que l'intention de l'utilisateur 2 qui a généré op_2 ne sera pas respectée sur le site 1, provoquant du même coup une divergence entre les copies de l'objet sur les deux sites. Voyons de manière détaillée le déroulement de l'exécution.

Sur le site 1, l'utilisateur 1 génère l'opération op_1 = *insérer(1, 'a')* ; l'opération est exécutée localement et transmise au site 2 où elle est également exécutée. L'état de la copie est maintenant "abd" sur les deux sites. L'utilisateur 2 présent sur le site 2 génère à son tour une opération op_2 = *insérer(3, 'c')* qui dépend donc causalement de op_1. Elle est exécutée localement et transmise au site 1 pour y être exécutée. Dans le même temps, l'utilisateur 1 génère l'opération op_3 qui consiste en l'annulation de op_1. Elle est donc causalement dépendante de op_1 et concurrente à op_2. Elle est exécutée localement et transmise au site 2 pour y être exécutée.

Reçue sur le site 2, l'opération op_3 est traitée par l'algorithme ANYUNDO de la manière suivante.

1. Génération de $\overline{op_1}$ à partir de op_1, $\overline{op_1}$ = *effacer(1, 'a')*. Transposition en avant par rapport aux opérations qui suivent op_1 dans l'histoire du site (en l'occurrence op_2). Exécution de $\overline{op_1}^{op_2}$ = *effacer(1, 'a')* sur l'état courant "abcd" pour obtenir le nouvel état "bcd".
2. Construction de la paire *faire/défaire*. Pour ce faire l'opération exécutée est transposée en arrière par rapport à toutes les opérations qui suivent dans l'histoire l'opération qui est annulée (ces opérations sont ici op_2). La transposition en arrière du couple (op_2,$\overline{op_1}^{op_2}$), Transpose_ar(*insérer(3, 'c')*, *effacer(1, 'a')*), donne le couple ($\overline{op_1}$, op_2') = (*effacer(1, 'a')*, *insérer(2, 'c')*). $\overline{op_1}$ forme avec op_1 la paire *faire/défaire* matérialisée dans l'histoire par op_1^*. L'opération op_2' remplace quand à elle op_2 dans l'histoire afin de refléter les conséquences de l'annulation de op_1 sur la suite de l'histoire.

Sur le site 2, le résultat est conforme à ce qu'on attendait.

Le problème se présente sur le site 1. Lorsque l'opération op_2 est reçue sur le site 1, supposons que l'opération locale op_3 ait déjà été exécutée. L'algorithme ANYUNDO aura été appliqué de la manière suivante :

1. Génération de $\overline{op_1}$ à partir de op_1, $\overline{op_1}$ = *effacer(1, 'a')*. Transposition en avant par rapport aux opérations qui suivent op_1 dans l'histoire du site

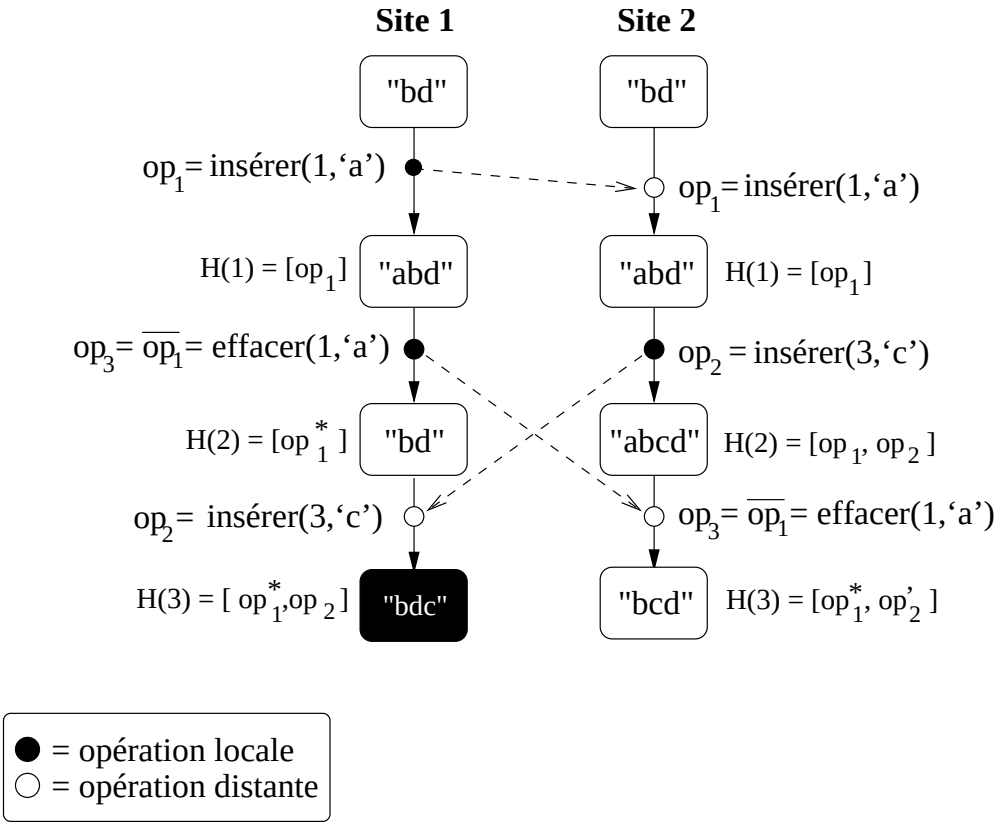


FIG. 7.5 – Exemple d'incohérence dans l'algorithme ANYUNDO

(il n'en existe pas à cet instant). Exécution de $\overline{op_1}$ = *effacer(1, 'a')* sur l'état courant "abd" pour obtenir le nouvel état "bd".

2. Construction de la paire *faire/défaire*. Pour ce faire l'opération exécutée est transposée en arrière par rapport à toutes les opérations qui suivent, dans l'histoire, l'opération qui a été annulée (il n'en existe aucune dans notre cas). $\overline{op_1}$ forme donc directement avec op_1 la paire *faire/défaire* matérialisée dans l'histoire par op_1^*. L'histoire ne contient que cette opération.

Lors de l'intégration de op_2 sur le site 1, on sait (en exploitant les estampilles) que op_1 la précède causalement. Cependant, l'annulation n'ayant pas été estampillée, on ne peut savoir si op_2 a été générée *avant* ou *après* que op_1 ait été annulée. Or on a vu en traitant ce qui se passe sur le site 2 que si op_2 tient compte de l'annulation de op_1, son expression peut en être modifiée. Il est donc indispensable de pouvoir différencier ces deux cas pour pouvoir soit transformer op_2 avant son exécution si elle ne tient pas compte de l'annulation de op_1, soit l'exécuter telle quelle dans le cas contraire. Sur la figure 7.5 nous avons inséré op_2 telle quelle sans tenir compte qu'elle est concurrente à l'annulation de op_3. C'est ce qui conduit à la divergence des copies.

Problème 2. Conséquences de l'inadaptation des fonctions de transpositions

Si on considère qu'on a une solution au premier problème, un deuxième reste à résoudre. L'annulation d'une opération op est obtenue en exécutant sur l'état résultant de l'exécution de op l'opération inverse, $\overline{op}$. Cependant, il est possible que l'objet ne soit plus dans l'état résultant de l'exécution de op. En effet, d'autres opérations dépendantes causalement ou concurrentes à op ont pu être exécutées entre-temps. Supposons dans un premier temps que les changements d'état ne soient dûs qu'à des opérations concurrentes. Pour simplifier, on supposera que op est générée à partir de l'état initial de l'objet. Au moment où l'annulation de op est demandée, on a donc l'histoire suivante :

$$H = op.op_1^{op}.op_2^{op.(op_1^{op})} \dots op_n^{op.(op_1^{op})\dots(op_{n-1}^{op\dots})}$$

En posant,

$$seq = op_1.op_2^{op_1}.op_3^{op_1.(op_2^{op_1})} \dots op_n^{op_1.(op_2^{op_1})\dots(op_{n-1}^{op_1\cdots})},$$

on peut écrire [Cor95, SCF98, Sul98] : $H = op.seq^{op}$

Le fonctionnement de l'algorithme ANYUNDO dans le cas qui nous intéresse ici sera alors le suivant. L'opération $\overline{op}$ est générée sur l'état résultant de l'exécution de op. De ce fait, $\overline{op}$ peut-être considérée comme étant concurrente aux opérations qui composent seq. Ensuite, $\overline{op}$ est transposée en avant par rapport à cette séquence. On obtient donc $\overline{op}^{(seq^{op})}$ qui exécutée sur l'état résultant de l'exécution de $op.seq^{op}$ devrait annuler les effets de op. Cependant, cette méthode ne peut fonctionner qu'à la condition que la définition des fonctions de transpositions tiennent effectivement compte des opérations d'annulation. Dans le cas contraire, cette façon de procéder peut ne pas donner le résultat correct. Nous allons illustrer cela par le contre-exemple suivant.

Exemple 7.2 – Inadaptation des fonctions de transpositions.

On considère comme objet partagé une chaîne de caractères avec comme opérations l'insertion, l'effacement et l'annulation (Fig. 7.6).

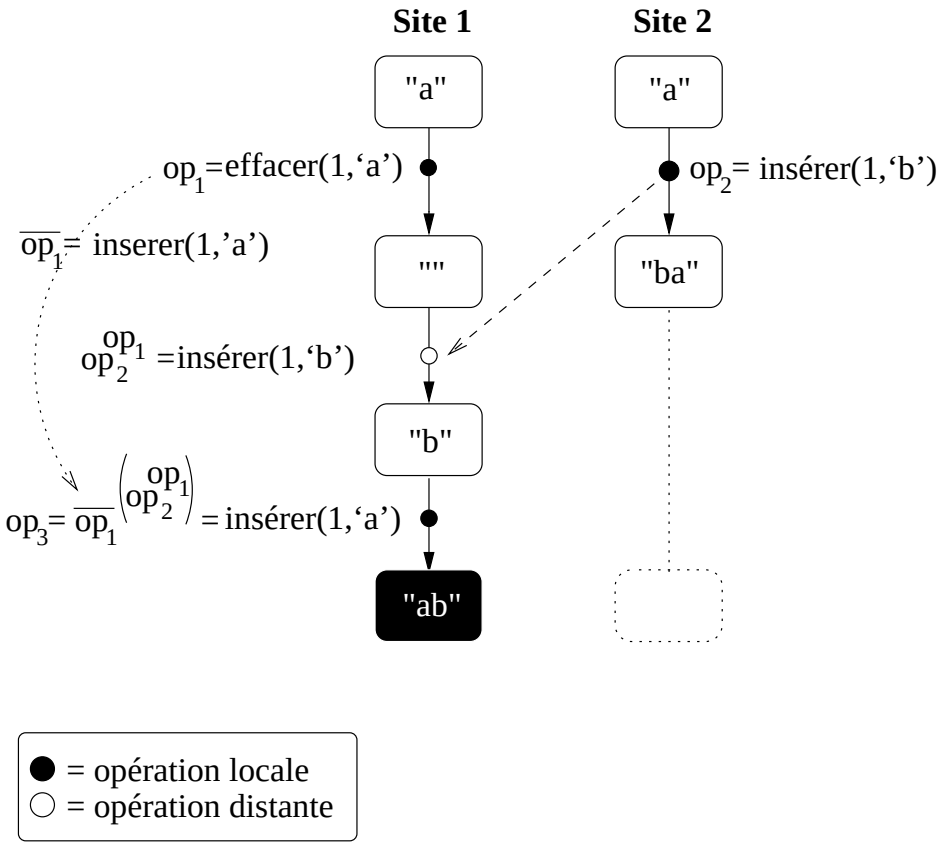


FIG. 7.6 – Exemple d'incohérence dans l'algorithme ANYUNDO

On pose op_1 = *effacer(1, 'a')* et *seq* = op_2 = *insérer(1, 'b')*. Du fait de l'opération op_1, on prendra comme état initial O_0 = *"a"*. L'opération op_1 réalise l'intention *"effacer le caractère 'a' en position 1"* et l'opération op_2, l'intention *"insérer le caractère 'b' avant le premier caractère"*. Si on se place sur le site sur lequel a été générée l'opération op_1, alors l'histoire est la suivante : $H = op_1.op_2^{op_1}$. Lorsque l'usager 1 demande l'annulation de op_1, il y a génération de $\overline{op_1}$ = *insérer(1, 'a')*, opération définie sur l'état résultant de l'exécution de op_1. On doit ensuite effectuer la transposition en avant de $\overline{op_1}$ par rapport à $op_2^{op_1}$ = *insérer(1, 'b')*. On obtient : $op_3 = \overline{op_1}^{(op_2^{op_1})}$ = *insérer(1, 'a')* [18]. L'exécution de cette opération op_3, après qu'ait été exécutée $op_1.op_2^{op_1}$, ne permet pas d'obtenir un résultat correct. En effet, l'exécution de $op_1.op_2^{op_1}$ sur l'état

[18] On suppose que l'ordre pour deux insertions concurrentes à la même position est donné par l'ordre alphabétique. Inverser l'ordre ne résout pas le problème ; un contre-exemple similaire peut toujours être construit en inversant les caractères 'a' et 'b'

initial "a" donne l'état "b". Enfin l'exécution de $\overline{op_1}^{op_2^{op_1}}$ à partir de cet état donne l'état "ab" qui ne correspond pas au résultat attendu "ba".

La nature particulière de l'annulation, à savoir une opération dont l'expression dépend d'une opération générée antérieurement, ne permet pas d'utiliser les transpositions définies pour les opérations *classiques*. En fait, comme pour toute nouvelle opération, les fonctions de transpositions doivent être adaptées.

Cela n'a pas été pris en compte lors de la conception de l'algorithme ANYUNDO. Cette lacune entraîne dans certaines situations un résultat incorrect lorsque l'annulation est utilisée. Dans [Sun00], l'auteur donne un moyen d'éviter ce problème lorsqu'on utilise le couple d'opérations {*insérer, effacer*}. La principe de la solution appelée *Concurrent Insert and Delete Reordering* est le suivant. Lorsqu'une opération vient d'être exécutée, une vérification est faite pour voir si une opération d'effacement concurrente à cette insertion existe dans l'histoire du site. Si c'est le cas, l'histoire est réordonnée pour que l'insertion soit placée avant l'effacement dans l'histoire du site. Cette solution, même si elle donne des résultats corrects, n'est appuyée par aucune justification. C'est une solution *ad-hoc* qui n'apporte aucune garantie sérieuse quand à l'adaptation possible de cet algorithme à d'autres types d'opérations. Cette solution n'est donc absolument pas adaptée au cas général que l'algorithme ANYUNDO prétend traiter.

7.2.3 Conclusion

L'algorithme d'annulation donné dans [Sun00] ne permet pas d'apporter une réponse au problème de l'annulation dans un environnement collaboratif réparti. L'objectif annoncé, à savoir la réussite de l'annulation de toute opération à tout moment[19], n'est en aucune manière atteint. Certaines difficultés sont traitées par des méthodes douteuses, qu'aucune justification sérieuse ne vient soutenir.

[19] [...] undoing any operation at any time with guaranteed success.

8

Principe d'une solution basée sur SOCT2

Sommaire

8.1 Introduction

Ce chapitre propose une solution au problème de l'annulation d'opérations quelconques dans le cadre d'un environnement collaboratif réparti utilisant un algorithme de type SOCT2. Il présente dans un premier temps les règles de base qui doivent être appliquées pour obtenir un résultat correct. Ces règles serviront à définir l'algorithme permettant le traitement de l'annulation.

8.2 Règles de base pour l'annulation

L'obtention d'un résultat correct pour le traitement de l'annulation en utilisant un algorithme à base de transpositions est soumise aux trois règles suivantes :

1. datation des opérations d'annulation,
2. utilisation d'une opération spécifique, `annuler(...)`,
3. utilisation de transpositions spécifiques à l'annulation.

La première règle conditionne la possibilité de déterminer la relation de dépendance entre les opérations et par conséquent la possibilité d'utiliser un algorithme à base de transpositions. En effet, les transpositions sont appliquées en fonction des dépendances existant entre les opérations.

La deuxième règle permet d'éviter la manipulation d'opérations dont la sémantique n'est pas conforme aux intentions qu'elles sont censées représenter. L'intention exprimée est l'annulation ; il faut donc une nouvelle opération, `annuler(...)`.

La troisième règle, enfin, conséquence de la deuxième exige l'utilisation de transpositions spécifiques à la sémantique de cette nouvelle opération dans la mesure où les transpositions sont des opérations de transformations qui dépendent de *la sémantique des opérations*. La suite de cette section détaille la mise en œuvre de ces trois règles.

8.2.1 Datation des opérations d'annulation

L'annulation d'une opération est un événement important. À ce titre, il doit être conservé dans l'histoire du site. En effet, cette histoire est la pierre angulaire des algorithmes utilisant l'intégration d'opération. Comme le montre le problème mis en évidence dans l'algorithme ANYUNDO (cf. 7.2.2), négliger d'y consigner correctement certains événements peut conduire à un résultat incorrect. Cela est d'autant plus vrai quand cet événement modifie l'objet manipulé. Signaler l'annulation d'une opération uniquement par une marque associée à celle-ci est insuffisant.

L'annulation d'une opération doit être stockée dans l'histoire au même titre que tout autre opération. En particulier, on doit veiller à lui adjoindre une estampille permettant de dater le moment auquel elle est survenue. Cette estampille est attribuée selon les mêmes règles que pour les autres opérations. Cette estampille est en fait la valeur du vecteur d'état de l'objet au moment où l'opération d'annulation a été générée.

8.2.2 Utilisation d'opération spécifique à l'annulation

Comme nous l'avons vu dans la section 6.4.2, utiliser l'inverse d'une opération et la transposer en avant par rapport à d'autres opérations de l'histoire nécessite que soit vérifiée la condition C4 (Déf. 6.4, p. 101). Cette vérification peut s'avérer difficile du fait de la présence du terme *seq* qui représente une séquence quelconque d'opérations. Les transpositions en avant que l'on fait subir à l'inverse de l'opération visent à ce qu'elle soit définie sur l'état courant de l'objet. Il paraÎt donc plus judicieux de transformer *l'histoire du site* pour faire en sorte que l'opération que l'on désire annuler se retrouve à la dernière place dans cette histoire. Ainsi en générant son inverse, on aura bien une opération qui, exécutée sur l'état courant de l'objet, réalise son annulation. L'intérêt de cette méthode tient au fait que les manipulations permettant d'amener l'opération à annuler en dernière position dans l'histoire sont déjà définies et correctes. L'annulation conduit donc simplement à transformer l'histoire à l'aide de fonctions dont la correction a été prouvée. De ce fait, on peut affirmer que l'annulation est elle même correcte, dans la mesure où elle respecte les intentions des utilisateurs et n'entraine pas de divergence entre les copies d'un objet.

Agir de cette façon nécessite que l'on puisse distinguer une opération d'annulation d'une opération *classique*. On introduit donc une nouvelle opération, `annuler(`op`)`, qui exprimera l'opération d'annulation de l'opération op. C'est cette opération `annuler` qui sera utilisée lors de la génération de l'opération et lors des transpositions nécessaires au processus d'intégration. L'utilisation d'une opération spécifique à l'annulation rend inutile la vérification de la condition C4 dans la mesure où cette condition ne concerne que l'opération inverse. De plus, les transpositions spécifiques à l'annulation, présentées dans la section suivante, sont telles que si l'exécution de l'opération op est séparée de celle de son annulation par la séquence seq alors l'opération d'annulation sera `annuler`(op'). L'opération op' est issue du couple (seq', op') obtenu en transposant en arrière le couple (op, seq). Ainsi la partie de l'histoire comprenant l'opération op et son annulation sera $op.seq.$`annuler`(op'), qui est également équivalente à $seq'.op'.$`annuler`(op'). Transposer en avant une opération op_i par rapport à la séquence $op.seq.$`annuler`(op') revient donc à transposer en avant op_i successivement par rapport à $op.seq$ puis par rapport à `annuler`(op'). Effectuer cette dernière transposition en avant consiste, comme nous le verrons à la section suivante, à effectuer l'inverse de la transposition en avant de op_i par rapport à op'. Compte tenu de l'équivalence entre les séquences $op.seq$ et $seq'.op'$ cela revient à dire que la transposition en avant de op_i par rapport à $op.seq.$`annuler`(op') consiste à transposer en avant op_i par rapport à seq'. La vérification de la condition C3 est donc naturellement assurée par ces transpositions.

8.2.3 Transpositions spécifiques à l'annulation

On a vu que si les fonctions de transpositions sont définies pour les opérations ordinaires, elles ne sont en revanche pas définies pour l'opération d'annulation qui est une opération particulière pour laquelle un traitement spécifique doit être employé. Pour être intégrée dans l'histoire d'un site, une opération doit être transposée en avant par rapport

aux opérations qui ont déjà été exécutées et qui lui sont concurrentes. Il faut donc enrichir les définitions des fonctions de transpositions en avant pour tenir compte du cas particulier de l'opération d'annulation.

La procédure d'intégration nécessite la définition des deux fonctions de transposition, transposition en avant et transposition en arrière. En ce qui concerne la transposition en arrière (`Transpose_ar`), on propose, d'utiliser la définition générique suivante :

$$\begin{aligned}\texttt{Transpose\_ar}(op_1,\, op_2) &= (op'_2, op'_1), \text{ avec :}\\ op'_2 &= \texttt{Transpose\_av}^{-1}(op_1, op_2) \text{ et}\\ op'_1 &= \texttt{Transpose\_av}(op'_2, op_1).\end{aligned}$$

Il suffit donc pour définir la transposition en arrière de donner la définition de l'inverse de la transposition en avant, $\texttt{Transpose\_av}^{-1}$.

Transposition en avant par rapport à une opération `annuler(...)`

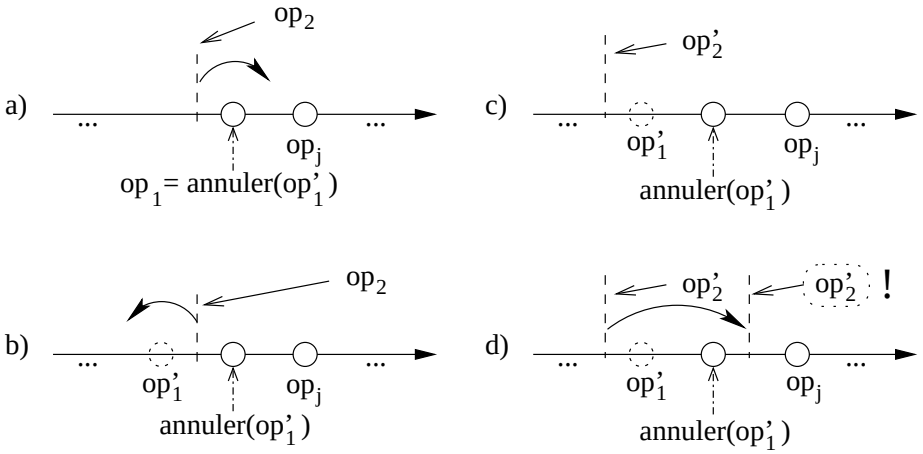


FIG. 8.1 – Transposition en avant par rapport à une opération d'annulation

Supposons que l'on se trouve dans la situation de la figure 8.1–a. On désire transposer en avant une opération op_2 par rapport à une opération $op_1 = \texttt{annuler}(op'_1)$. Cela suppose donc que op_2 et $\texttt{annuler}(op'_1)$ soient toutes deux définies sur le même état. On peut donc considérer que op'_1 est virtuellement l'opération qui a été exécutée avant $\texttt{annuler}(op'_1)$ (Fig. 8.1-b). C'est plausible puisque $\texttt{annuler}(op'_1)$ est l'opération qui annule l'effet de op'_1.

Annuler op'_1 c'est défaire l'effet de op'_1. De même, transposer en avant par rapport à $\texttt{annuler}(op'_1)$ c'est défaire l'effet qu'a eu la transposition en avant par rapport à op'_1. On va donc utiliser une fonction qui pour une opération op'_j et une opération op_i fournit l'opération op_j telle que $\texttt{Transpose\_av}(op_i,\, op_j) = op'_j$. Cette fonction est l'inverse de la transposé en avant, notée $\texttt{Transpose\_av}^{-1}$ et définie formellement comme suit :

$$\texttt{Transpose\_av}^{-1}(op_i, \texttt{Transpose\_av}(op_i, op_j)) = op_j$$

En appliquant, l'inverse de la transposée en avant sur les opérations op'_1 et op_2, ($\texttt{Transpose\_av}^{-1}(op'_1, op_2)$) on obtient l'opération op'_2 (Fig. 8.1–b/c). L'opération op'_2 réalise donc la même intention que op_2 et est définie sur le même état que op'_1. On a vu que transposer une opération en avant par rapport à un couple d'opération $op.\texttt{annuler}(op)$ ne doit pas modifier l'opération. Or ici, on cherche précisément à transposer en avant par rapport à $op'_1.\texttt{annuler}(op'_1)$. Il en résulte que l'opération qu'on cherche est tout simplement op'_2.

Pour résumer, la transposée en avant d'une opération op_2 par rapport à une opération $op_1 = \texttt{annuler}(op'_1)$ c'est l'inverse de la transposée en avant de op_2 par rapport à op'_1 :

$$\texttt{Transpose\_av}(\texttt{annuler}(op'_1),\ op_2) = \texttt{Transpose\_av}^{-1}(op'_1, op_2).$$

La seule exception concerne le cas où $op_2 = \texttt{annuler}(op'_2)$ et op'_2 est la *même* opération que op'_1. Alors, la transposition en avant aura pour résultat une opération identité.

Transposition en avant d'une opération `annuler(...)`

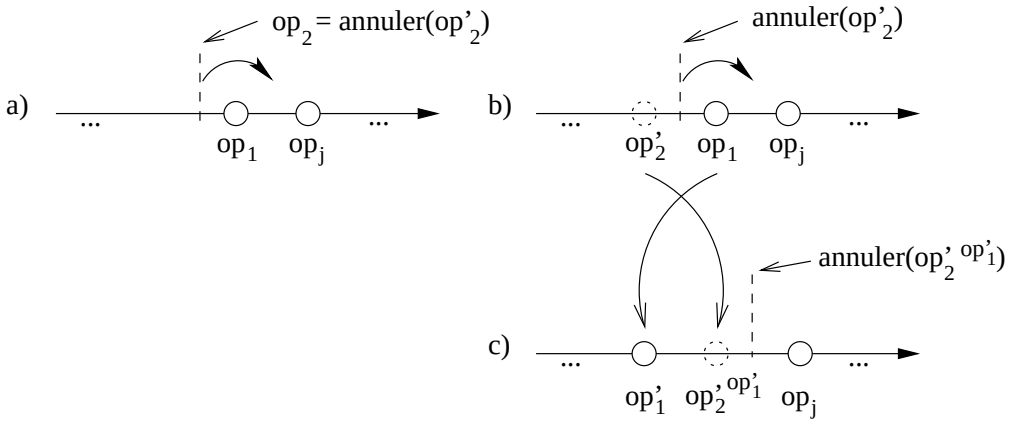


FIG. 8.2 – Transposition en avant d'une opération d'annulation

Intéressons-nous à l'exemple figure 8.2–a. On veut transposer en avant une opération $op_2 = \texttt{annuler}(op'_2)$ par rapport à une opération op_1. Cela suppose donc que $\texttt{annuler}(op'_2)$ est définie sur le même état que op_1. Par conséquent, on peut considérer comme illustré figure 8.2–b, que virtuellement, op'_2 est l'opération qui a été exécutée avant op_1. Cela est cohérent puisque $\texttt{annuler}(op'_2)$ est l'opération qui annule op'_2.

Comme le montre la figure 8.2–c, on peut obtenir l'opération qui va annuler op'_2 après que op_1 ait été exécutée en transposant en arrière le couple d'opérations (op'_2, op_1). On obtient alors le couple d'opérations (op'_1, $op_2^{'op'_1}$). La transposée $op_2^{'op'_1}$ est l'opération qui réaliserait l'intention de op_2 si op'_1 avait été exécutée avant elle. Il en résulte que la transposée en avant de $\texttt{annuler}(op'_2)$ par rapport à op_1 est l'opération $\texttt{annuler}(op_2^{'op'_1})$

Pour résumer, la transposée en avant d'une opération $\texttt{annuler}(op'_2)$ par rapport à une opération op_1 c'est l'opération d'annulation de la transposée en avant de op'_2 par rapport à l'inverse de la transposée en avant de op_1 par rapport à op'_2 :

$$\texttt{Transpose\_av}(op_1, \texttt{annuler}(op'_2)) = \texttt{annuler}(\texttt{Transpose\_av}(\texttt{Transpose\_av}^{-1}(op'_2, op_1), op'_2))$$

Si op_1 est une opération d'annulation, on utilise la définition précédente.

La fonction de transposition en avant est donc étendue en ajoutant les cas supplémentaires pour tenir compte de l'annulation.

```
Transpose_av(op1, op2) =
    cas (op1 = annuler(op'1))
        cas (op2 = annuler(op'2) et op'1 =id op'2)
            retour identité(op2) ;
        sinon
            retour Transpose_av^-1(op'1, op2) ;
        fincas
    cas (op2 = annuler(op'2))
        retour annuler(Transpose_av(Transpose_av^-1(op'2, op1),op'2)) ;
    ...
    fincas
```

Inverse de la transposition en avant d'une opération par rapport à une opération `annuler(...)`

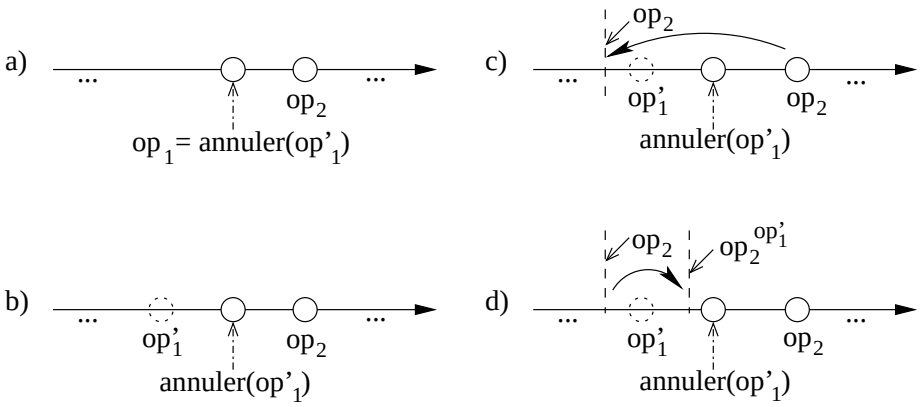


FIG. 8.3 – Inverse de la transposition en avant d'une opération par rapport à une opération d'annulation

Supposons que l'on veuille obtenir l'inverse de la transposée en avant d'une opération op_2 par rapport à une opération $op_1 = \texttt{annuler}(op)$ (Fig. 8.3). On peut dans un premier temps supposer qu' op'_1 est virtuellement l'opération qui a été exécutée juste avant $\texttt{annuler}(op'_1)$ (Fig. 8.3-b). Comme la transposée en avant d'une opération op par rapport à un couple opération/annulation doit être l'opération op elle-même, on admettra facilement que l'inverse de la transposée en avant de op_2 par rapport à $op'_1.\texttt{annuler}(op'_1)$ soit l'opération op_2 elle-même (Fig. 8.3-c). Dans ces conditions, on peut obtenir l'inverse de la

transposée en avant de op_2 par rapport à $\texttt{annuler}(op'_1)$ en transposant en avant op_2 par rapport à op'_1. On peut donc donner la définition formelle suivante :

$$\texttt{Transpose\_av}^{-1}(\texttt{annuler}(op),\, op_j) = op_j^{op}$$

Inverse de la transposition en avant d'une opération `annuler(...)`

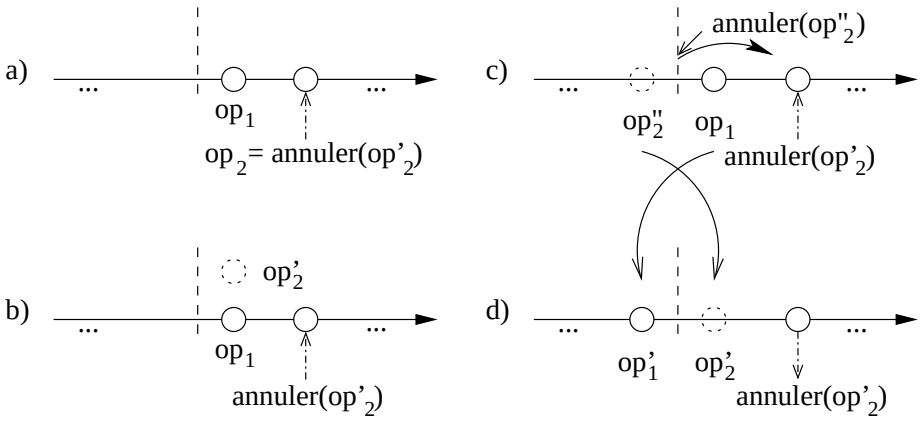


FIG. 8.4 – Inverse de la transposition en avant d'une opération d'annulation

On suppose maintenant qu'on veuille calculer l'inverse de la transposition en avant de $op_2 = \texttt{annuler}(op'_2)$ par rapport à une opération op_1 (Fig. 8.4). Comme op_2 est définie sur l'état résultant de l'exécution de op_1, on peut admettre que op'_2 est virtuellement définie sur le même état que op_1 (Fig. 8.4-b). On cherche a obtenir l'opération dont l'exécution sur l'état où est exécuté op_1 réaliserait la même intention que op_2. Il s'agit de l'opération $\texttt{annuler}(op''_2)$ dont la transposée en avant par rapport à op_1 a pour résultat $\texttt{annuler}(op'_2)$ (Fig. 8.4-c). Si $\texttt{annuler}(op''_2)$ est définie sur le même état que op_1 alors on peut admettre que op''_2 est virtuellement l'opération dont l'exécution a fourni l'état sur lequel op_1 est définie. La transposée en avant de $\texttt{annuler}(op''_2)$ par rapport à op_1 s'obtient en calculant la transposée en arrière du couple $(op''_2,\, op_1)$ qui fournit le couple $(op'_1,\, op'_2)$. Dans ce dernier, op'_2 est le résultat de la transposée en avant de op''_2 par rapport à op'_1.

On peut alors raisonnablement penser que pour obtenir op''_2, il suffit de calculer l'inverse de la transposée en avant de op'_2 par rapport à op'_1. Malheureusement, en ayant connaissance uniquement de op_1, op_2 et par conséquent de op'_2, cela n'est pas possible. En effet, calculer l'inverse de la transposée en avant de op'_2 par rapport à op'_1 nécessite de calculer au préalable op'_1. Or par définition de la transposée en arrière, op'_1 est le résultat de l'inverse de la transposée en avant de op_1 par rapport à op''_2. Son calcul nécessite donc la connaissance de op''_2. Or op''_2 est précisément l'opération qu'on recherche.

Il n'existe donc pas d'autre solution que d'utiliser l'histoire qui précède op_1 et de la réordonner de sorte que l'opération op''_2 se retrouve en dernière position dans celle-ci. Une discussion plus détaillée sur ce problème peut-être trouvée à l'annexe C.

8.3 Annulation basée sur SOCT2

8.3.1 Architecture générale

La méthode d'annulation dont nous allons préciser le principe est adaptée aux algorithmes de type SOCT2. Ces algorithmes (cf. 2.4) reposent sur trois fonctionnalitées mises en œuvre dans des procédures appelées `Diffusion`, `Livraison_Causale` et `Intégration`. On rappelle leur principe dans la suite

La procédure `Diffusion` est exécutée consécutivement à la génération d'une opération sur le site. Elle prend en charge le formatage du message transmis aux sites participants en associant à l'opération l'identifiant et le vecteur d'état correspondant au site sur lequel elle a été générée. Elle assure la diffusion de ce message à tous les sites, y compris le site qui a généré l'opération. Le message diffusé pour une opération est un triplet $\langle op, S_{op}, V_{op}\rangle$ où op désigne l'opération, S_{op} le site sur lequel a été générée op, et SV_{op} le vecteur d'état associé à op.

La procédure `Reception` est chargée de rétablir l'ordre causal qui n'est pas assuré par le réseau de communication. Elle utilise le vecteur d'état associé aux opérations ainsi que celui du site où elle s'exécute pour délivrer à la procédure `Intégration` les opérations reçues suivant l'ordre causal (cf. 6.3.1). Cette procédure permet d'assurer que si op_1 précède causalement op_2 ($op_1 \rightarrow_c op_2$) alors op_1 sera exécutée avant op_2 [SCF97]. On notera que l'ordre de livraison des opérations concurrentes peut varier d'un site à l'autre.

La procédure `Integration` est appelée lorsqu'une opération, locale ou distante, doit être exécutée. Avant son exécution, et afin de satisfaire aux exigences du respect des intentions des utilisateurs et de la convergence des copies, une opération distante doit être transposée en avant pour pouvoir être exécutée sur l'état courant de l'objet. L'intégration d'une opération, qui peut être une opération d'annulation, se déroule suivant le même principe que dans l'algorithme SOCT2 [SCF98, Sul98]. Seules des fonctions de transposition adaptées en particulier à l'annulation sont nécessaires pour garantir une intégration correcte. Un exemple de telles fonctions sont données en annexe D.

8.3.2 Principe de l'algorithme d'annulation

Le principe de l'algorithme prenant en compte l'annulation est le suivant :
- Les opérations *classiques* sont traitées de manière identique à ce qui est fait dans SOCT2 tant en ce qui concerne la génération, la diffusion, l'exécution locale que l'intégration et l'exécution sur un site distant.
- Le traitement d'une opération d'annulation se passe de la manière suivante :
  - Détermination de l'opération à annuler, op. Le déroulement exact de cette phase dépendra de la politique d'annulation retenue. Elle pourra être exécutée de manière automatique ou nécessiter l'intervention de l'utilisateur en cas d'annulation libre.
  - Génération de l'opération `annuler`(op).

– Transposition en avant de l'opération `annuler`(op) par rapport à toutes les opérations qui suivent op dans l'histoire du site. Le but est d'obtenir l'opération d'annulation notée `annuler`(op') qui, exécutée sur l'état courant de la copie, réalise l'annulation de op. On associe à cette opération le vecteur d'état courant comme cela est fait pour une opération *classique*.
– Diffusion de l'opération `annuler`(op') sous cette forme à tous les sites pour y être exécutée.

8.3.3 Problématique liée à SOCT3, SOCT4 et SOCT5

Ni l'algorithme SOCT3, ni les algorithmes SOCT4 et SOCT5 présentés dans la première partie ne sont adaptés au traitement de l'annulation. En effet, l'absence de vérification de la condition C2 fait que des choix arbitraires peuvent être dépendants de l'exécution antérieure de certaines opérations. De ce fait, il devient impossible d'exprimer l'annulation sans violer l'intention des usagers. L'exemple suivant permet d'illustrer ce problème.

Exemple 8.1 – Impossibilité d'annuler si la condition C2 n'est pas vérifiée.

On dispose d'un objet de type chaîne de caractères dans l'état initial `"abc"` partagé par réplication entre trois sites. Sur chacun des sites et au même moment, une opération est générée en concurrence avec les autres. Le premier utilisateur dont l'intention est d'« effacer le caractère 'b' » génère l'opération op_1 = `effacer`(2) ; le deuxième utilisateur génère l'opération op_2 = `insérer`(2, 'y') traduisant son intention d'« insérer 'y' entre les caractères 'a' et 'b' » et enfin le troisième dont l'intention est d'« insérer 'x' entre 'b' et 'c' » génère l'opération op_3 = `insérer`(3, 'x'). On suppose dans la suite que l'indice i de l'opération op_i exprime l'ordre global continu utilisé dans les algorithmes SOCT3, SOCT4 et SOCT5. Il en résulte que sur chacun des sites l'exécution sera équivalente à la séquence : $op_1.op_2^{op_1}.op_3^{op_1:op_2}$.

En se reportant aux transpositions en avant décrites dans les exemples 1.3 et 1.4 présentés dans la première partie et qui satisfont uniquement à la condition C1, on en déduit que :

$$\begin{aligned} op_1 &= \texttt{effacer}(2), \\ op_2^{op_1} &= \texttt{insérer}(2, \text{'y'}), \\ op_3^{op_1} &= \texttt{insérer}(2, \text{'x'}) \text{ et} \\ op_3^{op_1:op_2} = (op_3^{op_1})^{(op_2^{op_1})} &= \texttt{insérer}(2, \text{'x'}). \end{aligned}$$

Les exécutions sont donc équivalentes à l'exécution en série des opérations `effacer`(2), `insérer`(2, 'y') et `insérer`(2, 'x'). L'état de l'objet après cette exécution est donc `"axyc"`. Les intentions des trois utilisateurs à l'origine de la génération des trois opérations sont respectées. En effet, l'utilisateur qui a généré op_1 a manifesté l'intention d'« effacer le caractère 'b' », ce que reflète

bien l'état de l'objet `"axyc"`. L'utilisateur qui a généré op_2 a manifesté l'intention d'« insérer 'y' entre les caractères 'a' et 'b' ». Considérant que 'b' a été effacé, cette intention ne peut se traduire que par « insérer 'y' entre 'a' et 'c' » et cette intention est bien respectée dans `"axyc"`. De la même façon, l'intention originale traduite par op_3, « insérer 'x' entre 'b' et 'c' » ne peut, après effacement de 'b', qu'être traduite par « insérer 'x' entre 'a' et 'c' » qui est une intention respectée au vue de l'état de l'objet, `"axyc"`. Maintenant si on considère l'annulation de op_1, on remarque qu'elle ne conduit pas à un état correct puisque ni l'état `"abxyc"`, ni l'état `"axbyc"`, ni l'état `"axybc"` ne représentent le résultat d'une exécution qui respecte les intentions que traduisent les opérations op_2 et op_3. Pourtant ce sont les seuls états qui pourraient représenter le résultat d'une exécution respectant l'intention « annuler op_1 ». On aboutit donc a l'impossibilité de traduire l'annulation d'une opération, qui respecte les intentions des usagers.

8.4 Application aux situations dites problématiques

8.4.1 Introduction

Cette section, illustre le fonctionnement de l'algorithme d'annulation sur les exemples dits problématiques du chapitre 5.4 et permet ainsi de valider l'approche utilisée.

8.4.2 Situation d'ordre perdu

On rappelle qu'une situation d'ordre perdu (Fig. 5.9) est une situation dans laquelle l'annulation de deux opérations consécutives donne un résultat incorrect parce que l'ordre original des opérations n'est pas respecté.

Considérons la situation suivante. On manipule un objet ayant la valeur initiale "ab". L'utilisateur U_1, sur le site 1, génère et exécute l'opération $op_1 =$ `effacer`$(1, \text{'a'})$. Cette opération est transmise au site 2, où se trouve le deuxième utilisateur, pour y être exécutée. Ultérieurement, l'utilisateur U_2, sur le site 2, génère et exécute $op_2 =$ `effacer`$(1, \text{'b'})$. Cette opération est alors transmise au site 1 où elle est exécutée. L'état des copies de l'objet sur les deux sites est donc "" (c.-à-d. la chaîne vide). Les deux utilisateurs, décident maintenant d'annuler leur dernière opération, et ce de manière concurrente.

En réponse, à la requête d'annulation formulée par l'utilisateur 1, le système génère l'opération `annuler`(effacer$(1, \text{'a'})$) définie sur l'état résultant de l'exécution de op_1. Cette opération est ensuite transposée en avant par rapport aux opérations qui ont été exécutées postérieurement à l'exécution de op_1 sur le site 1. L'opération `annuler`(effacer$(1, \text{'a'})$) est donc transposée en avant par rapport à l'opération `effacer`$(1, \text{'b'})$. En appliquant la définition de la transposée en avant donnée section D.2.1, on voit que cette transposition consiste à :

1. calculer la transposée inverse de `effacer`$(1, \text{'b'})$ par rapport à `effacer`$(1, \text{'a'})$; ce qui donne `effacer`$(2, \text{'b'})$.

2. calculer la transposée en avant de `effacer`(1, ‘a’) par rapport au résultat précédent ; ce qui donne `effacer`(1, ‘a’).

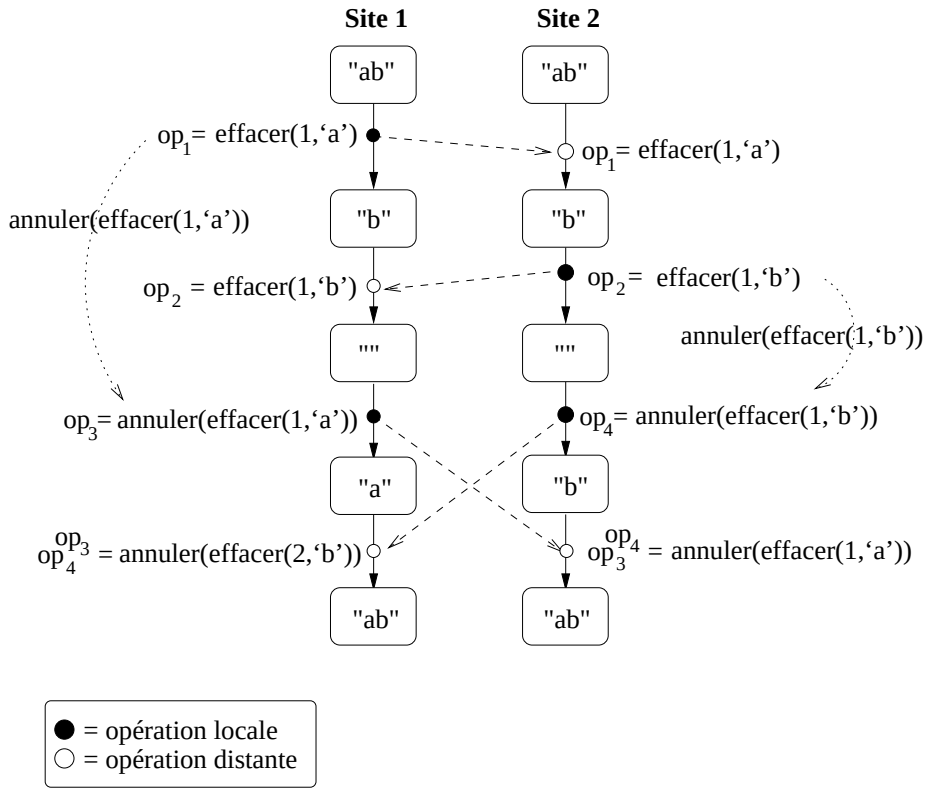


FIG. 8.5 – Ordre perdu

Le résultat de la transposition en avant de `annuler`(`effacer`(1, ‘a’)) par rapport à `effacer`(1, ‘b’) est donc `annuler`(`effacer`(1, ‘a’)). On désignera cette opération comme étant l'opération op_3. Le vectuer d'état courant lui est attribuée. Elle est exécutée localement avant d'être transmise aux sites distants pour y être exécutée. L'état de la copie de l'objet sur le site 1 est donc maintenant “a”.

Sur le site 2, le processus est similaire mais plus simple du fait que l'opération devant être annulée, op_2, n'est suivie d'aucune autre opération. En réponse, à la requête d'annulation formulée par l'utilisateur U_2, le système génère l'opération `annuler`(effacer(1, ‘b’)) définie sur l'état résultant de l'exécution de op_2. Cette opération n'a pas à être modifiée puisqu'il n'existe pas d'opération qui ait été exécutée postérieurement à l'exécution de op_2 sur le site 2. L'opération `annuler`(`effacer`(1, ‘b’)) est donc conservée telle quelle. On désignera cette opération comme étant l'opération op_4. Le vecteur d'état courant lui est attribuée après quoi, elle est exécutée localement et transmise aux sites distants pour y être exécutée. Après exécution de cette opération, la copie de l'objet sur le site 2 est donc dans l'état “b”

Sur le site 1, lorsque l'opération distante op_4 est reçue, elle est transposée en avant

par rapport aux opérations concurrentes. Il n'en existe qu'une ici qui est op_3 = `annuler`(`effacer`(1, 'a')). La transposée en avant de op_4 = `annuler`(`effacer`(1, 'b')) par rapport à op_3 est obtenue en calculant l'inverse de la transposée en avant de `annuler`(`effacer`(1, 'b')) par rapport à `effacer`(1, 'a') (cf. D.2.1). Le calcul de l'inverse de cette transposée en avant consiste à (cf. D.3.1) utiliser l'histoire du site pour recalculer l'opération op'_4 telle que la transposée en avant de op'_4 par rapport à `effacer`(1, 'a') = op_1 soit égale à op_4. On obtient cette opération en réordonnant l'histoire du site de telle sorte que op_2 soit exécutée avant op_1, c.-à-d. en transposant en arrière le couple (op_1, op_2). Puis en calculant l'opération qui générée sur l'état résultant de l'exécution de op_2 annule cette dernière. C'est l'opération op'_4 = `annuler`(`effacer`(2, 'b')).

Le résultat de la transposée en avant de `annuler`(`effacer`(1, 'b')) par rapport à `annuler`(`effacer`(1, 'a')) sera donc `annuler`(`effacer`(2, 'b')). L'exécution de cette opération amène donc la copie de l'objet sur le site 1 à l'état "ab" qui est bien l'état attendu.

Sur le site 2, à la livraison de l'opération distante op_3, celle-ci est transposée en avant par rapport aux opérations concurrentes. Il n'en existe qu'une ici qui est op_4 = `annuler`(`effacer`(1, 'b')). La transposée en avant de op_3 = `annuler`(`effacer`(1, 'a')) par rapport à op_4 est obtenue en calculant l'inverse de la transposée en avant de `annuler`(`effacer`(1, 'a')) par rapport à `effacer`(1, 'b'). Le calcul de l'inverse de cette transposée en avant consiste à utiliser l'histoire du site pour recalculer l'opération op'_3 telle que la transposée en avant de op'_3 par rapport à `effacer`(1, 'b') = op_2 soit égale à op_3. On obtient cette opération en réordonnant l'histoire du site de telle sorte que op_1 soit exécutée avant op_2, c.-à-d. qu'ici il n'est pas nécessaire de modifier l'histoire puisqu'elle se trouve déjà dans cette configuration. On peut donc calculer directement l'opération qui générée sur l'état résultant de l'exécution de op_1 annule cette dernière. C'est l'opération op'_3 = `annuler`(`effacer`(1, 'a')).

Le résultat de la transposée en avant de `annuler`(`effacer`(1, 'a')) par rapport à `annuler`(`effacer`(1, 'b')) sera donc `annuler`(`effacer`(1, 'a')). L'exécution de cette opération amène donc la copie de l'objet sur le site 2 à l'état "ab" qui est bien l'état attendu.

L'état de l'objet après les deux annulations est le même sur les deux sites : "ab". La cohérence des copies et le respect des intentions des utilisateurs sont assurés.

8.4.3 Annulation d'opérations concurrentes réalisant la même intention

On rappelle que la situation à laquelle nous nous intéressons ici est une situation dans laquelle l'annulation d'une opération ramène de façon prématurée dans un état antérieur. Le retour à cet état antérieur ne devrait, en effet, survenir que lorsque *toutes* les opérations réalisant la même intention ont été annulées.

Un exemple de telle situation est le suivant. L'état initial de l'objet est "a". Deux utilisateurs génèrent et exécutent en concurrence la *même* opération `effacer`(1, 'a') ; l'utilisateur U_1 génère l'opération op_1 et l'utilisateur U_2 génère l'opération op_2. Sur le site 1, à la livraison de l'opération distante op_2 celle-ci est transposée en avant par rapport à op_1 pour

donner `identité`(`effacer`(1, ‘a’))[20]. De même sur le site 2, l'opération op_1 est transposée par rapport à l'opération locale concurrente op_2 pour donner `identité`(`effacer`(1, ‘a’)).

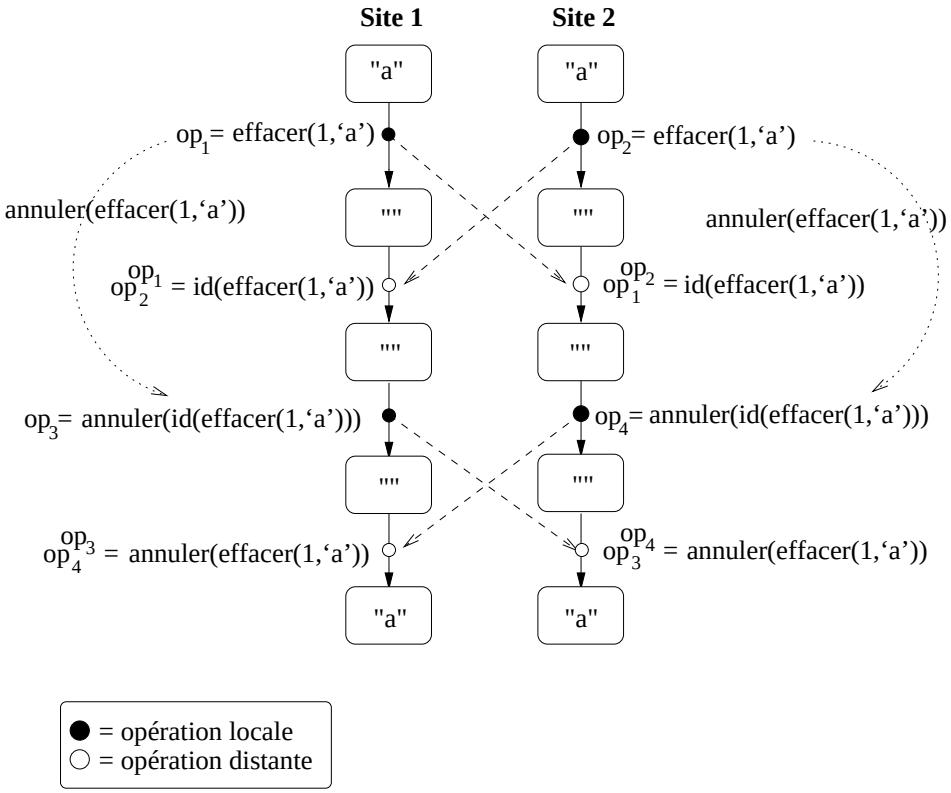


FIG. 8.6 – *Solution au problème de l'annulation d'opérations concurrentes réalisant la même intention*

Supposons que l'utilisateur U_1 décide d'annuler l'opération op_1. Le système à partir de op_1 génère alors `annuler`(`effacer`(1, ‘a’)) qui est ensuite transposée en avant par rapport à l'opération $op_2^{op_1}$, qui a été exécutée postérieurement à op_1 sur le site 1. Ceci conduit à (cf. D.3.1) :

1. calculer l'inverse de la transposée en avant de `identité`(`effacer`(1 ,‘a’)) par rapport à `effacer`(1, ‘a’) ; ce qui donne `effacer`(1, ‘a’).
2. transposer en avant `effacer`(1, ‘a’) par rapport à ce résultat ; ce qui donne `identité`(`effacer`(1, ‘a’)).

Le résultat est l'opération op_3 = `annuler`(`identité`(`effacer`(1, ‘a’))) dont l'exécution laisse la copie de l'objet dans l'état “”. On remarque que cette fois la copie de l'objet ne retourne pas de manière prématurée à l'état “a”.

Supposons maintenant que l'utilisateur U_2 sur le site 2 ait lui aussi décidé d'annuler son opération. On passe sur l'exécution locale de cette opération pour s'intéresser à

[20] notée `id`(`effacer`(1, ‘a’)) sur la figure.

son exécution sur le site 1. Lorsque le site 1 reçoit l'opération op_4 = `annuler`(`identité`-(`effacer`$(1,$'a'$)$)), il doit la transposer par rapport à l'opération concurrente op_3 = `annuler`(`identité`(`effacer`$(1,$'a'$)$)). Or on a vu que `Transpose_av`(`annuler`$(op'_i), op_j)$ est obtenue à partir de `Transpose_av`$^{-1}(op'_i, op_j)$ (cf. page 124). Il faut donc calculer l'inverse de la transposée en avant de op_4 par rapport à l'opération qu'annule op_3, à savoir `identité`(`effacer`$(1$,'a')) que nous savons être $op_1^{op_2}$. Le problème est que op_4 est une opération d'annulation, or on a vu que ce calcul n'est pas possible si on ne dispose que des opérations op_4 et op_3 (cf. page 127).

La seule solution consiste à réordonner l'histoire du site 1 de telle sorte que op_3 soit précédée de l'opération op'_3 qu'elle annule, et que op'_3 soit elle-même précédée de l'opération op'_4 qu'annule op_4. Ainsi la transposée en avant de op_4 par rapport à op_3 sera simplement `annuler`(op'_4). L'opération op_4 annule op_2 et l'opération op_3 annule op_1. Il suffit donc, pour obtenir la configuration que nous venons d'évoquer, de transposer en arrière le couple $(op_1, op_2^{op_1})$. Le résultat obtenu sera le couple (op'_4, op'_3) = (`effacer`$(1,$'a'), `id`(`effacer`$(1,$'a'$)$)). La transposée en avant de op_4 par rapport à op_3 est donc `annuler`(`effacer`$(1,$'a'$)$).

Le résultat obtenu est satisfaisant. La convergence des copies est assurée ainsi que le respect des intentions des utilisateurs. Sur le site 2, le même raisonnement peut être appliqué car la situation est similaire à celle du site 1.

8.4.4 Situation de fausse concurrence

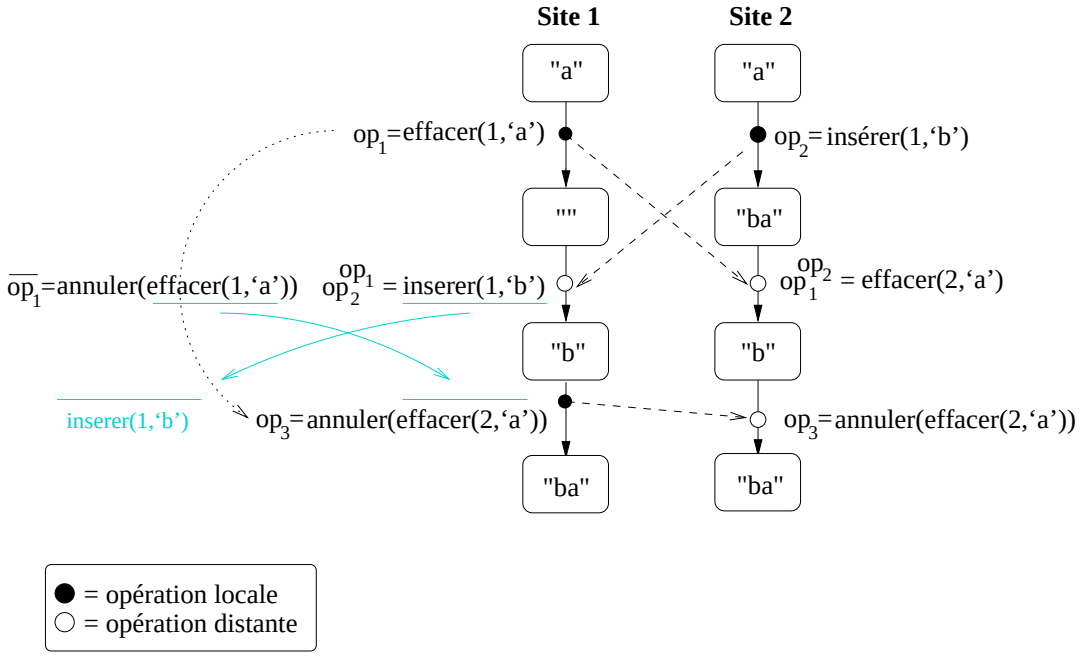


FIG. 8.7 – Situation de fausse concurrence

On rappelle que la situation de fausse concurrence (Fig. 5.11) est une situation dans

laquelle l'annulation est considérée, à tort, comme une opération concurrente *classique*. Cela va entraîner un résultat incorrect lors de l'emploi des transpositions. Prendre en compte l'annulation comme une opération spécifique et adapter les fonctions de transposition en conséquence permet de résoudre ce problème.

Considérons un objet ayant pour valeur initiale "a" partagé entre deux utilisateurs. Sur le site 1, l'utilisateur U_1 génère et exécute l'opération $op_1 = \texttt{effacer}(1, \text{'a'})$. Concurremment, l'utilisateur U_2, sur le site 2, génère et exécute $op_2 = \texttt{insérer}(1, \text{'b'})$. Les deux opérations sont transmises aux sites distants respectifs pour y être exécutées. Après livraison, les opérations sont donc transposées en avant par rapport aux opérations locales avant d'être exécutées. L'état des copies devient "b".

Sur le site 1, l'utilisateur U_1 décide alors d'annuler son opération $op_1 = \texttt{effacer}(1, \text{'a'})$. Le système génère donc l'opération $\overline{op_1} = \texttt{annuler}(\texttt{insérer}(1, \text{'a'}))$ qui est ensuite transposée par rapport à $op_2^{op_1}$. En utilisant les fonctions de transposition définies à l'annexe D, on sait que la transposée en avant d'une opération $\texttt{annuler}(op_i)$ par rapport à une opération op_j, c'est l'opération d'annulation de la transposée en avant de op_i par rapport à l'inverse de la transposée en avant de op_j par rapport à op_i, c.-à-d. $\texttt{Transpose\_av}(\texttt{Transpose\_av}^{-1}(op_i, op_j), op_i)$. Comme par ailleurs $\texttt{Transpose\_av}^{-1}(op_1, op_2^{op_1}) = op_2 = \texttt{insérer}(1, \text{'b'})$ il vient $\texttt{Transpose\_av}(op_2, op_1) = \texttt{effacer}(2, \text{'a'})$.

L'opération qui réalise l'annulation de op_1 après l'exécution de $op_1.op_2^{op_1}$ est donc $op_3 = \texttt{annuler}(\texttt{effacer}(2, \text{'a'}))$ dont l'exécution fournit un résultat correct. Les intentions des utilisateurs sont respectées et la convergence des copies est assurée.

8.4.5 Situation de référence ambiguë

On rappelle qu'une situation de référence ambiguë est une situation dans laquelle l'annulation d'une opération peut amener à l'obtention de deux états qui sont tous les deux valides. En fait, elle correspond à l'expression de deux intentions conflictuelles pouvant être résolues de deux manières différentes et toutes deux acceptables.

Considérons l'exemple suivant. Un objet de valeur initiale "a" est partagé entre deux utilisateurs. Sur le site 1, l'utilisateur U_1 génère et exécute l'opération $op_1 =$ *effacer(1, 'a')*. Cette opération est ensuite transmise au site 2 distant. Après que op_1 ait été reçue et exécutée sur le site 2, l'utilisateur U_2, sur ce même site, génère et exécute $op_2 =$ *inserer(1, 'b')*. Cette opération est ensuite transmise au site 1 pour y être exécutée.

L'état de la copie sur les deux sites est maintenant "b". Supposons que l'utilisateur sur le site 1, décide d'annuler sa dernière opération. On applique l'algorithme proposé comme pour les cas précédents. Puisque les conditions C1 et C2 sont vérifiées, cette situation n'est pas problématique dans la mesure où on est assuré que les conflits d'intentions seront résolus de la même manière sur tous les sites. En conséquence, la convergence des copies sera assurée et les intentions des utilisateurs respectées, dans la mesure où le même choix arbitraire a été fait dans la transposition.

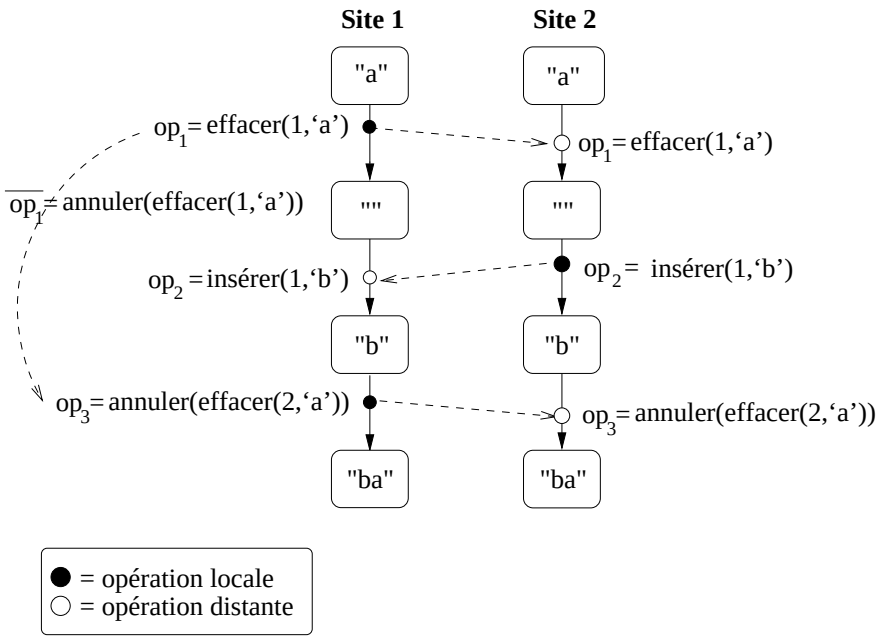


FIG. 8.8 – Situation de référence ambiguë

Conclusion

Dans ce chapitre, nous présentons tout d'abord un bref résumé des deux parties du document. Nous rappelons ensuite les principaux problèmes évoqués et les solutions que nous leur avons apportées. Enfin, nous évoquons les perspectives de recherches envisageables.

Algorithmes pour la convergence des copie

Dans la première partie, nous avons présenté la problématique générale des applications collaboratives. Nous avons vu que cette problématique comportait trois aspects, à savoir (1) le respect de la causalité, (2) le respect des intentions des utilisateurs et (3) la convergence des copies. Nous avons vu que le premier aspect était traité de manière satisfaisante par la mise en place d'une procédure de livraison causale ou séquentielle permettant de respecter l'ordre causal lors de l'exécution des opérations. Nous avons vu que le respect des intentions des utilisateurs était obtenu en utilisant la transposition en avant qui permet d'obtenir, à partir d'une opération et d'une autre qui lui est concurrente, une opération qui réalise la même intention que la première en tenant compte des modifications effectuées par la seconde. Nous avons précisé que l'application correcte de la transposition en avant demandait pour certains algorithmes l'utilisation de la transposition en arrière qui permet d'obtenir, pour un couple d'opérations concurrentes sérialisées, un couple d'opérations sérialisées dans l'ordre inverse. Enfin, nous avons vu que la convergence des copies nécessitait, pour la plupart des algorithmes, soit de devoir défaire puis refaire des opérations, soit d'avoir des transpositions (en avant et en arrière) qui vérifient les conditions C1 et C2. Nous avons montre en quoi la vérification de la condition C2, lorsqu'elle est possible, était difficile.

Une présentation des approches existantes nous a permis de mettre en évidence les contraintes que posent chacune d'elles. A cette occasion, nous avons montré que l'algorithme SOCT3, basé sur un ordre global continu et n'imposant pas la condition C2, présentait une faiblesse du fait de l'utilisation de la transposition en arrière. Nous avons finalement proposé deux nouveaux algorithmes, basés sur un ordre global continu, qui ne font pas obligation aux transpositions en avant de vérifier la condition C2, qui n'utilisent pas de transposition en arrière et qui ne requièrent pas de défaire puis refaire des opérations.

Le premier algorithme que nous avons défini, SOCT4, est une variante qui utilise la diffusion différée des opérations. La diffusion différée permet de construire un algorithme ayant un fonctionnement très simple. En effet, il utilise uniquement la transposition en avant pour satisfaire au respect des intentions des utilisateurs. De plus, les opérations

distantes sont délivrées suivant l'ordre global continu et exécutées dans ce même ordre. Cela conjugué à la diffusion différée permet, lorsqu'une opération est reçue, de ne pas avoir à séparer dans l'histoire les opérations qui lui sont concurrentes de celles qui la précèdent causalement, limitant par là-même, le nombre de transpositions en avant nécessaires entre la génération d'une opération et son exécution sur un site distant. Une autre conséquence est que les vecteurs d'états ne sont plus nécessaires. Enfin, les opérations reçues sont placées dans l'histoire sous la forme qu'elles ont lorsqu'elles sont reçues. Le principal inconvénient de cet algorithme est qu'il exploite mal la concurrence lors de la diffusion des opérations.

Pour parer à cet inconvénient, le deuxième algorithme que nous proposons, SOCT5, au prix d'un mécanisme plus complexe permet de mettre en œuvre la diffusion immédiate des opérations et profite donc de la concurrence dans la diffusion. Cette complexité se traduit par exemple par la nécessité d'employer à nouveau les vecteurs d'états. Ceux-ci sont utilisés non seulement pour assurer la livraison causale mais également pour garantir l'application correcte des transpositions en avant en permettant de distinguer, pour une opération donnée, les opérations qui lui sont concurrentes de celles qui la précèdent causalement. Toutefois, celles-ci ne sont pas entrelacées comme c'est le cas dans SOCT3, ce qui limite l'utilisation des transpositions en avant dont le nombre reste comparable à celui de SOCT4. Comme pour SOCT4, seules les transpositions en avant sont utilisées pour garantir le respect des intentions des utilisateurs.

Traitement de l'annulation

Dans la deuxième partie, nous nous sommes intéressés au problème de l'annulation dans les applications collaboratives. Nous y décrivons la problématique générale de l'annulation et les problèmes spécifiques qu'elle pose dans le cadre d'une application collaborative répartie. Ces problèmes sont illustrés sur des exemples litigieux déjà identifiés auxquels nous apportons un éclairage nouveau. Nous avons introduit deux nouvelles conditions, C3 et C4, qui doivent être vérifiées par les transpositions en avant et en arrière pour garantir des résultats corrects lorsqu'une opération d'annulation est traduite par l'inverse de l'opération qu'elle annule. La condition C3 précise la neutralité des couples opération/annulation lorsque l'on transpose une opération par rapport à une séquence contenant de tels couples . La condition C4 stipule quant à elle que la transposée d'une annulation par rapport à une séquence doit être égale à l'annulation de la transposée par rapport à cette même séquence.

À la lumière de ces nouvelles conditions, nous avons analysé les exemples litigieux. Nous avons ensuite présenté deux propositions récentes et avons précisé en quoi elles n'étaient pas satisfaisantes. Pour l'une d'elle, cela concerne les conditions d'applications de l'annulation qui sont restrictives puisque basées sur une annulation suivant l'ordre chronologique inverse. Pour l'autre, qui entendait apporter une solution générale au problème, nous avons montré que la convergence des copies n'était pas assurée. Notre proposition consiste à considérer l'annulation comme une opération à part entière qui nécessite de définir des transpositions spécifiques.

Perspectives

Les thèmes de recherches abordés dans ce mémoire ont été introduits pour répondre aux exigences du travail collaboratif. Il apparaît que de nouvelles perspectives d'applications des techniques mises en œuvre mériteraient d'être explorées. Nous les détaillons brièvement.

Synchroniseur "intelligent"

L'informatique aujourd'hui se conçoit de plus en plus comme une informatique nomade ou mobile. L'apparition des PDA ou la démocratisation des portables favorise encore ce phénomène. Ainsi il n'est pas rare qu'un utilisateur dispose d'un poste fixe, à son bureau, et d'un portable ou d'un PDA, c.-à-d. un mobile, utilisé lors de ses déplacements. L'utilisateur emporte avec lui certaines copies d'objets qui seront mises à jour au gré de son activité, durant ces déplacements. Cela peut par exemple concerner un annuaire des clients à visiter pour un commercial. Lorsque l'utilisateur modifie des copies sur son mobile, il doit de retour à son bureau mettre à jour les copies utilisées sur son poste fixe et qui sont devenues obsolètes. Le rôle d'un *Synchroniseur* est justement de permettre cette fusion de façon automatisée. Cependant, si les copies sur le poste fixe ont également été modifiées entre temps alors les Synchroniseurs existants ne peuvent bien souvent pas réaliser la synchronisation [BP98] et l'utilisateur doit recourir à une synchronisation manuelle. Les algorithmes utilisés dans le domaine du collaboratif pourraient, appliqués au problème de la synchronisation, permettre d'aller plus loin dans la réalisation de Synchroniseurs automatiques. Cela nécessiterait la définition des transpositions en avant spécifiques à la sémantique des opérations utilisées sur ces copies.

Compression d'histoire

La deuxième application possible concerne plus spécifiquement l'utilisation de la transposition en arrière. Beaucoup d'applications nécessitent la maintenance d'un journal retraçant l'histoire des opérations exécutées. C'est le cas notamment des procédures de reprise dans les systèmes transactionnels où le journal permet à partir d'une sauvegarde de retrouver l'état du système avant l'arrêt accidentel. Un problème qui peut survenir dans de tels systèmes est l'accroissement rédhibitoire de la taille du journal. Une solution consiste à compacter ce journal. Pour l'illustrer simplement, on peut prendre l'exemple des opérations définies sur une chaîne de caractères ; deux insertions de caractères à des positions consécutives peuvent être remplacées par une unique insertion des deux caractères. Cette réduction n'est possible que si les opérations sont regroupées en séquences cohérentes qui peuvent être remplacées par une opération unique. C'est précisément ce que permet la transposition en arrière.

Annexes

A

Problème lié à la définition de `Transpose_ar` dans SOCT3

Sommaire

A.1 Introduction

Dans cette annexe, nous allons montrer qu'il existe des cas où la non vérification de la condition C2 entraîne l'impossibilité de définir la transposée en arrière, `Transpose_ar`.

Nous rappellerons dans un premier temps les fonctions de transposition, en avant et en arrière, associées aux opérations {`insérer,effacer`} qui manipulent une chaîne de caractères. Ces transpositions sont issues de [Sul98] et satisfont aux conditions C1 et C2.

A.2 Transpositions en avant

```
Transpose_av(insérer(p1, c1, av1, ap1), insérer(p2, c2, av2, ap2)) =
   cas
      (p1 < p2) retour insérer(p2 + 1, c2, av2, ap2) ;
      (p1 > p2) retour insérer(p2, c2, av2, ap2) ;
      (p1 = p2)
         cas
            (ap1 ∩ av2 ≠ ∅) retour insérer (p2 + 1, c2, av2, ap2) ;
            (av1 ∩ ap2 ≠ ∅) retour insérer(p2, c2, av2, ap2) ;
            (code(c1) < code(c2)) retour insérer(p2, c2, av2, ap2) ;
            (code(c2) < code(c1)) retour insérer(p2 + 1, c2, av2, ap2) ;
            (code(c1) = code(c2)) retour identité(insérer(p2, c2, av2, ap2))
         fincas
   fincas


Transpose_av(effacer(p1), insérer(p2, c2, av2, ap2)) =
   si (p1 < p2) retour insérer(p2 − 1, c2, av2+{effacer(p1)}, ap2) ;
   sinon retour insérer(p2, c2, av2, ap2+{effacer(p1)}) ;
   finsi


Transpose_av(insérer(p1, c1, av1, ap1), effacer(p2)) =
   si (p1 ≤ p2) retour effacer(p2 + 1) ;
   sinon retour effacer(p2) ;
   finsi


Transpose_av(effacer(p1), effacer(p2)) =
   cas
      (p1 < p2) retour effacer(p2 − 1) ;
      (p1 > p2) retour effacer(p2) ;
      (p1 = p2) retour id(effacer(p2) ;
   fincas
```

A.3 Transpositions en arrière

```
Transpose_ar(insérer(p1, c1, av1, ap1), insérer(p2, c2, av2, ap2)) =
   cas
      (p1 < p2) retour (insérer(p2 − 1, c2, av2, ap2), insérer(p1, c1, av1, ap1)) ;
      (p1 ≥ p2) retour (insérer(p2, c2, av2, ap2), insérer(p1 + 1, c1, av1, ap1)) ;
   fincas


Transpose_ar(insérer(p1, c1, av1, ap1), effacer(p2)) =
   cas (p1 < p2) retour (effacer(p2 − 1), insérer(p1, c1, av1, ap1+{effacer(p2 − 1)})) ;
       (p1 > p2) retour (effacer(p2), insérer(p1 − 1, c1, av1+{effacer(p2)}, ap1)) ;
            (p1 = p2) est un cas impossible
   fincas


Transpose_ar(effacer(p1), insérer(p2,c2,av2,ap2)) =
   cas
      (p1 < p2) retour (insérer(p2 + 1, c2 ,av2-{effacer(p1)}, ap2), effacer(p1)) ;
      (p1 > p2) retour (insérer(p2, c2, av2, ap2-{effacer(p1)}),effacer(p1 + 1)) ;
▷     (p1 = p2) si (effacer(p1) ∈ av2)
                alors retour (insérer(p2 + 1, c2, av2-{effacer(p1)}, ap2), effacer(p1))
                sinon retour (insérer(p2, c2, av2, ap2-{effacer(p1)}), effacer(p1 + 1)) ;
   fincas


Transpose_ar(effacer(p1), effacer(p2)) =
   cas
      (p1 ≤ p2) retour (effacer(p2 + 1), effacer(p1)) ;
      (p1 > p2) retour (effacer(p2), effacer(p1 − 1) ;
   fincas
```

A.4 Conséquence de la non vérification de la condition C2

Considérons la fonction de transposition en arrière de `insérer`(p_2,c_2,av_2,ap_2) par rapport à `effacer`(p_1), l'avant-dernière dans la section précédente. Si on définit cette transposition sans tenir compte de la vérification de la condition C2, le cas marqué ▷ doit être réécrit car il exploite les paramètres supplémentaires qui ont justement été introduit pour la satisfaire.. Il faut donc utiliser une des deux alternatives en effectuant un choix arbitraires. Dans la suite, nous allons montrer que, quel que soit le choix effectué, il entrainera une divergence des copies de l'objet.

Comme expliqué à la section 1.3, la transposition en arrière permet de réordonner deux opérations qui étaient concurrentes et qui ont été sérialisées en transposant en avant l'une par rapport à l'autre. Si les opérations originales sont op_1 et op_2 et qu'elles sont sérialisées telle que op_1 précède op_2 alors op_2 a été transposée en avant par rapport à op_1

pour donner la séquence $op_1.op_2^{op_1}$. Transposer en arrière $op_2^{op_1}$ pr rapport à op_1 permettra d'obtenir la séquence $op_2.op_1^{op_2}$. Dans le cas qui nous préoccupe ici, op_1 = `effacer`(p_1) et $op_2^{op_1}$ = `insérer`(p_2,c_2).

A.4.1 Alternative 1

On considère que l'opération d'insertion, op_2 originale opère juste *avant* le caractère qu'efface l'opération op_1. L'insertion n'aura donc pas été modifiée lors de sa transposition en avant, la transposée en arrière doit donc la restituer telle qu'elle est. Par contre, exécutée avant l'opération d'effacement, l'insertion provoque le décalage de cette dernière d'une position vers la droite. Tout cela se traduit par la ligne pointée par le marqueur $\lhd$ dans l'algorithme suivant :

Transpose_ar(effacer(p_1), insérer(p_2,c)) =
cas p_1 ? p_2 **de**
  $p_1 < p_2$: **retour** (insérer($p_2 + 1$, c), effacer(p_1)) ;
  $p_1 > p_2$: **retour** (insérer(p_2, c),effacer($p_1 + 1$)) ;
  $p_1 = p_2$: **retour** (insérer(p_2, c), effacer($p_1 + 1$)) ; $\lhd$

A.4.2 Alternative 2

On considère que l'opération d'insertion, op_2 originale opère juste *après* le caractère qu'efface l'opération, op_1. L'insertion aura donc subi un décalage d'une position vers la droite lors de sa transposition en avant, la transposée en arrière doit donc la restituer en la décalant d'une position vers la droite. Par contre, exécutée avant l'opération d'effacement, l'insertion ne modifie pas cette dernière. Une fois de plus tout cela se traduit par la ligne pointée par le marqueur $\lhd$ dans l'algorithme suivant :

Transpose_ar(effacer(p_1), insérer(p_2,c)) =
cas p_1 ? p_2 **de**
  $p_1 < p_2$: **retour** (insérer($p_2 + 1$, c), effacer(p_1)) ;
  $p_1 > p_2$: **retour** (insérer(p_2, c),effacer($p_1 + 1$)) ;
  $p_1 = p_2$: **retour** (insérer($p_2 + 1$, c), effacer(p_1)) ; $\lhd$

A.5 Exemples prouvant la divergence des copies

Dans cette section, nous présentons pour chacune des alternatives un exemple d'exécution qui prouve l'existence d'un cas de divergence des copies lorsque l'algorithme SOCT3 est utilisé avec des transpositions ne vérifiant pas la condition C2.

Ces deux exemples font intervenir trois opérations, op_1, op_2 et op_3, concurrentes entre elles et générées respectivement par les sites 1, 2 et 3. Elles sont telles que op_1 *précède*$_S$ op_2 et op_2 *précède*$_S$ op_3. L'ordre d'exécution est donc op_1, op_2 puis op_3, sur le site 1 ; op_2,

op_1, op_3, sur le site 2 ; op_3, op_1, op_2, sur le site 3.

Les figures illustrent pour chaque site l'exécution qui a eu lieu. Ces exécutions mentionnent l'évolution des états de la copie au fur et à mesure de l'exécution des opérations reçues. Pour les sites 2 et 3, l'emploi de la transposition en arrière impose la représentation de l'exécution avant et après la transposition en arrière.

Sur la figure A.1 qui utilise l'algorithme de transposition en arrière décrit dans l'alternative 1, on note qu'il y a divergence des copies de l'objet. Cela est dû au résultat erroné fournit par la transposée en arrière sur le site 2. En effet, alors que l'opération originale $op_1 =$ `insérée`$(3, \text{'y'})$ a opérée sur une position placée *après* la position où opère l'opération $op_2 =$ `effacer`(2), elle est restituée, à tort, par la transposition en arrière comme `insérée`$(2, \text{'y'})$, opération qui aurait opérée sur une position placée *avant* la position de l'opération op_2.

La figure A.2 présente ce qu'on pourrait qualifier de cas dual. L'algorithme qui y est utilisé est celui décrit dans l'alternative 2. On note, ici aussi, qu'il y a divergence des copies de l'objet. Cette fois encore, cela est dû au résultat erroné fournit par la transposée en arrière sur le site 2. La transposition en arrière restitue à tort `insérée`$(3, \text{'y'})$, opération qui aurait opéré sur une position placée *après* celle où opère l'opération $op_2 =$ `effacer`(2) alors que l'opération originale $op_1 =$ `insérée`$(2, \text{'y'})$ est une opération qui opère sur une position placée *avant* la position où opère op_2.

A.6 Conclusion

Nous avons montré qu'il existe des cas où la définition des fonctions de transposition en arrière qui écartent tout risque de divergence des copies ne peut se faire sans que la condition C2 soit vérifiée. La nécessité de vérifier que pour ces cas les fonctions de transposition en arrière dans SOCT3 satisfont bien à la condition C2 est contraignante. Obtenir des algorithmes ne nécessitant pas l'emploi de la transposée en arrière présente donc un double avantage : (1) se passer de la définition des fonctions de transposition en arrière et (2) ne pas avoir à vérifier leur validité par rapport à la condition C2.

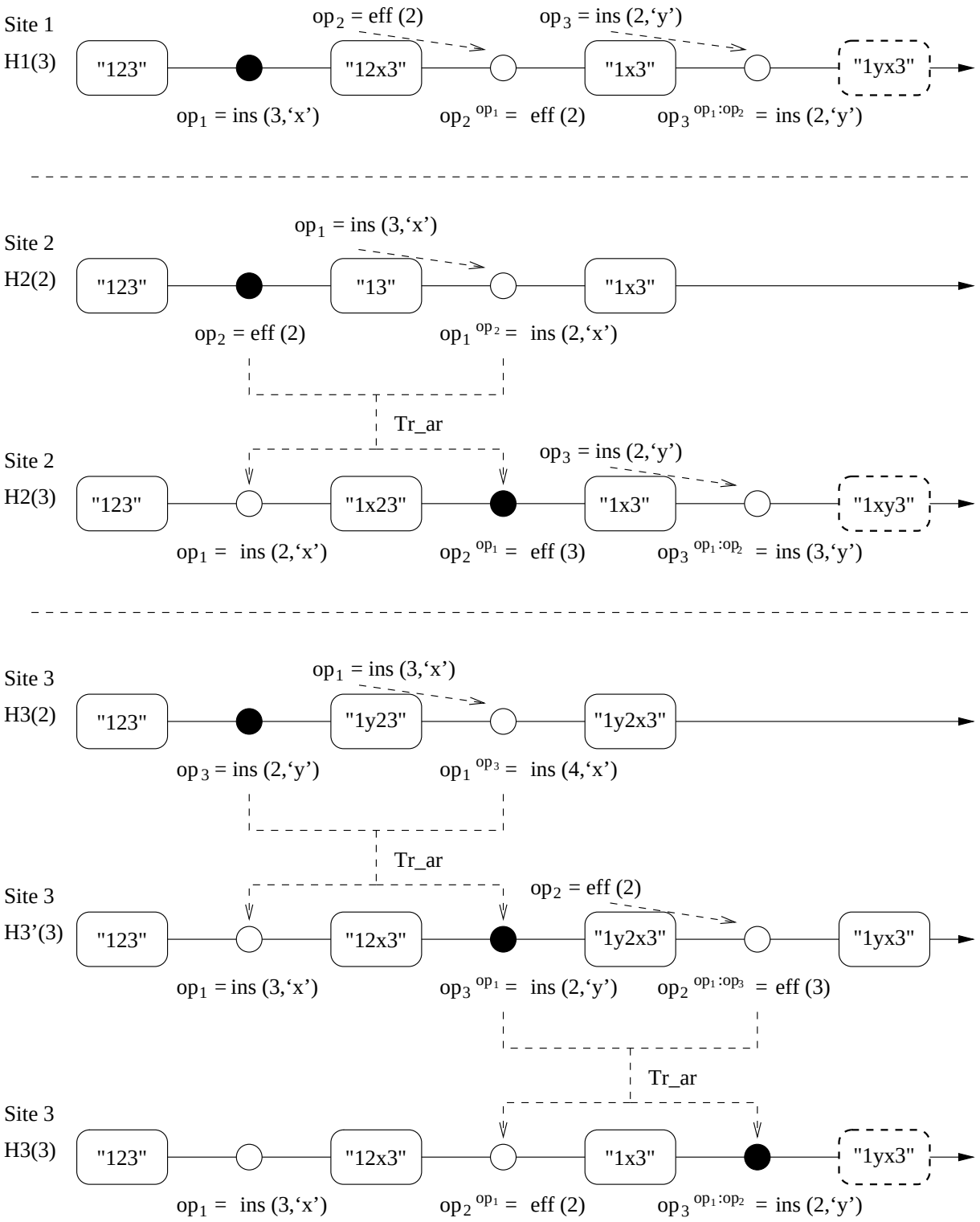


FIG. A.1 – Exemple d'exécution avec l'alternative 1

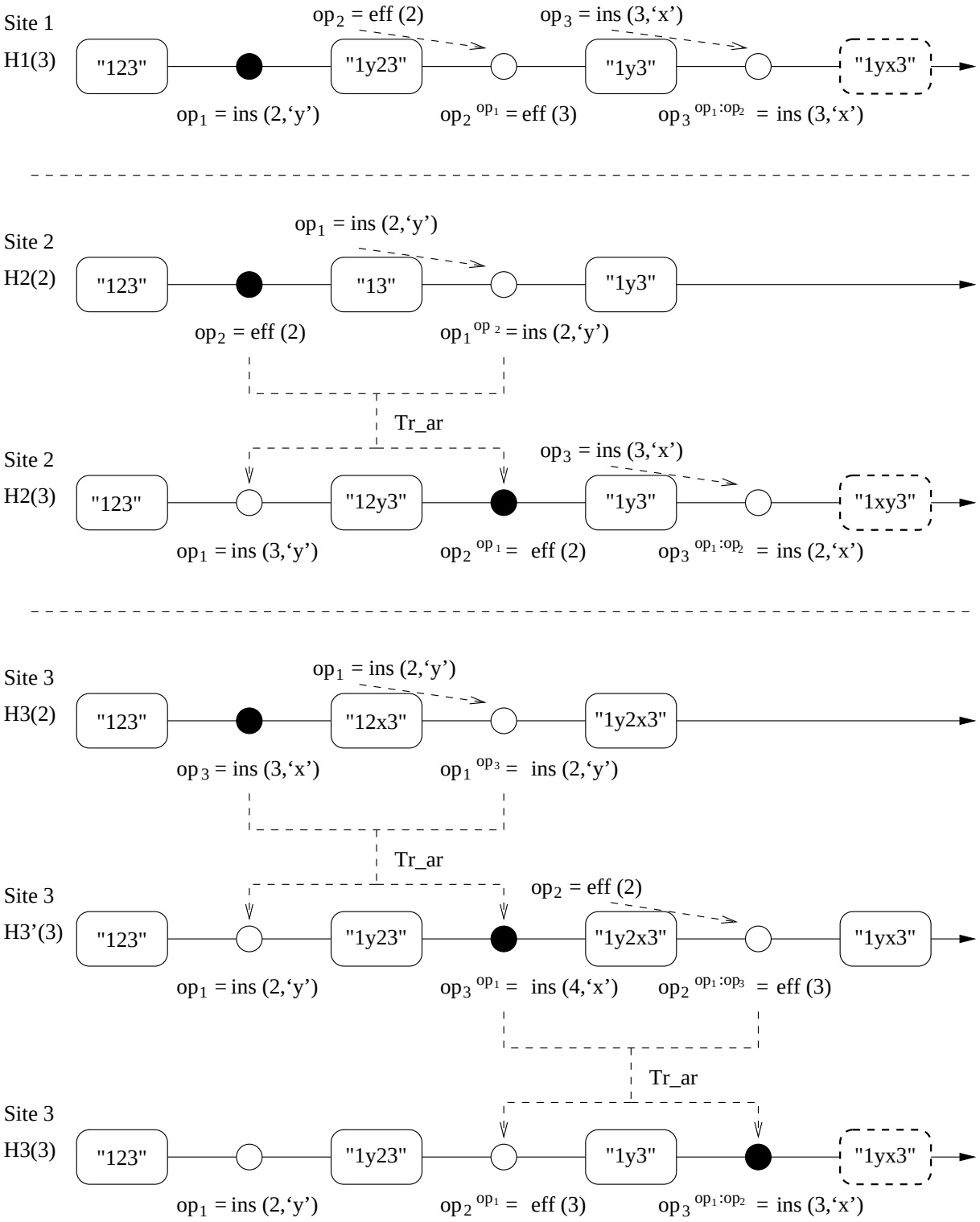


FIG. A.2 – Exemple d’exécution avec l’alternative 2

B

Preuves des algorithmes

Sommaire

B.1 Correction de SOCT4

La correction de SOCT4 consiste à prouver (1) le respect de la causalité, (2) la convergence des copies et (3) le respect de l'intention de l'usager.

B.1.1 Respect de la causalité

Une opération locale étant exécutée immédiatement après sa génération, les opérations qui la précèdent causalement sont celles qui ont été exécutées sur son site avant sa génération. En ce qui concerne des opérations distantes, la livraison séquentielle les délivre dans l'ordre $précède_S$ et la procédure Intégration les exécute suivant ce même ordre. Comme l'ordre $précède_S$ est compatible avec l'ordre $précède_C$, la causalité est respectée.

B.1.2 Convergence des copies

Pour montrer que SOCT4 assure la convergence des copies, il faut et il suffit de montrer que sur un site S, l'histoire réelle sur ce site est équivalente par transposition à l'histoire mémorisée par SOCT4. On rappelle que dans SOCT4 les opérations délivrées sur un site sont mémorisées dans l'histoire de ce site en suivant l'ordre des estampilles et que ces opérations sont identiques sur tous les sites, puisqu'elles sont intégrées sans modification.

On rappelle également que l'histoire $H_S(n)$ des n opérations exécutées et mémorisées sur S par SOCT4 est constituée de la concaténation (notée ".") de l'histoire des i (avec $i \geq 0$) opérations délivrées, notée $HD_S(n)$, et de l'histoire des m (avec $m \geq 0$) opérations locales en attente, notée $HL_S(n)$. On a : $H_S(n) = HD_S(n).HL_S(n)$. L'histoire réelle $HR_S(n)$ représente la séquence des opérations exécutées sur S, dans l'ordre chronologique de leur exécution.

Il faut montrer que l'histoire $H_S(n)$ mémorisée par SOCT4 est équivalente par transposition à l'histoire réelle $HR_S(n)$, c'est à dire que : $\forall n, H_S(n) \equiv_T HR_S(n)$. Cette preuve se fait par récurrence sur n.

On pose $i + m = n$.

Cas de base : $n = m$, c.-à-d. aucune opération n'a été délivrée ($i = 0$). $HD_S(n) = \emptyset$,

$$HL_S(n) = op_{L_1}.op_{L_2}.\ldots.op_{L_m}$$

et

$$HR_S(n) = op_{L_1}.op_{L_2}.\ldots.op_{L_m}.$$

On a donc bien $H_S(n) \equiv_T HR_S(n)$.

Hypothèse de récurrence :
Supposons que $H_S(n) \equiv_T HR_S(n)$ et montrons que $H_S(n+1) \equiv_T HR_S(n+1)$. L'histoire $H_S(n+1)$ mémorisée par SOCT4 est obtenue à partir de $H_S(n)$:

(1) Soit lors de l'exécution d'une nouvelle opération locale $op_{L_{m+1}}$,

(2) Soit lors de l'intégration d'une opération distante op_{i+1}.

Dans le cas (1), l'opération locale après exécution est mémorisée telle quelle à la fin de $HL_S(n)$ pour donner $HL_S(n+1)$.

Donc

$$\begin{aligned} H_S(n+1) &= HD_S(n+1).HL_S(n+1) \\ &= HD_S(n).HL_S(n).op_{L_{m+1}} \\ &\equiv_T HR_S(n).op_{L_{m+1}} \\ &= HR_S(n+1). \end{aligned}$$

Dans le cas (2), l'opération distante op_{i+1}, avant d'être exécutée, doit être transposée en avant par rapport aux opérations locales en attente, ce qui donne l'opération : $op_{i+1}^{op_{L_1}.op_{L_2}....op_{L_m}}$. L'histoire réelle devient alors $HR_S(n+1) = HR_S(n).op_{i+1}^{op_{L1}.op_{L_2}....op_{L_m}}$. L'opération op_{i+1} est mémorisée à la suite de l'histoire des opérations délivrées $HD_S(n)$ pour donner $HD_S(n+1)$. On a alors $HD_S(n+1) = HD_S(n).op_{i+1}$.

On a donc : $H_S(n+1) = HD_S(n+1).HL_S(n+1) = HD_S(n).op_{i+1}.HL_S(n+1)$.

En outre, chaque opération locale op_{L_k} doit être transposée en avant par rapport à la transposée de op_{i+1}, notée $op_{i+1}^{seq'_k}$, où $seq'_k = op_{L_1}.op_{L_2}.\ldots.op_{L_{k-1}}$ est la sous-séquence des opérations locales qui précèdent op_{L_k}, ce qui donne :

$$HL_S(n+1) = op_{L_1}^{op_{i+1}}.op_{L_2}^{op_{L_1}^{op_{i+1}}}.\ldots.op_{L_m}^{op_{L_1}.op_{L_2}....op_{L_{m-1}}^{op_{i+1}}}$$

En remplaçant dans $H_S(n+1)$, $HL_S(n+1)$ par les opérations qui la composent, on obtient :

$$H_S(n+1) = HD_S(n).op_{i+1}.op_{L_1}^{op_{i+1}}.op_{L_2}^{op_{L_1}^{op_{i+1}}....op_{L_m}^{op_{L_1}.op_{L_2}....op_{L_{m-1}}^{op_{i+1}}}}$$

En remarquant que `Transpose_ar`$(op_{i+1}, op_{L_1}^{op_{i+1}}) = (op_{L_1}, op_{i+1}^{op_{L_1}})$, que `Transpose_ar`$(op_{i+1}^{op_{L_1}}, op_{L_2}^{op_{i+1}^{op_{L_1}}}) = (op_{L_2}, op_{i+1}^{op_{L_1}.op_{L_2}})$ et que plus généralement `Transpose_ar`$(op_{i+1}^{op_{L_k}}, op_{L_{k+1}}^{op_{i+1}^{seq'_k}}) = (op_{L_{k+1}}, op_{i+1}^{seq'_{k+1}})$, si on transpose en arrière op_{i+1} avec la séquence des opérations qui la suivent dans l'histoire $H_S(n+1)$, on obtient :

$$H_S(n+1) = HD_S(n).op_{L_1}.op_{L_2}.\ldots.op_{L_m}.op_{i+1}^{op_{L_1}.op_{L_2}....op_{L_m}}$$

D'où

$$H_S(n+1) = HD_S(n).HL_S(n).op_{i+1}^{op_{L_1}.op_{L_2}....op_{L_m}}.$$

Soit $H_S(n+1) \equiv_T HR_S(n+1)$.

B.1.3 Respect de l'intention

Nous venons de prouver que l'histoire mémorisée par SOCT4 était équivalente par transposition à l'histoire réelle du site. En outre, on prouve dans [Sul98] que des histoires équivalentes par transposition réalisent les mêmes intentions. Comme sur chaque site, l'intention de l'usager est respectée pour ses opérations locales et que les opérations diffusées sont mémorisées dans le même ordre et sans modification, les intentions des usagers sont respectées sur tous les sites.

B.2 Correction de SOCT5

L'algorithme SOCT5 est très largement basé sur SOCT4, la seule différence étant liée à la répartition des traitements. La correction de SOCT5 peut donc être prouvée simplement de la même manière qu'a été prouvée la correction de SOCT4 que nous venons d'évoquer.

C

Inverse de la transposition en avant d'une opération d'annulation

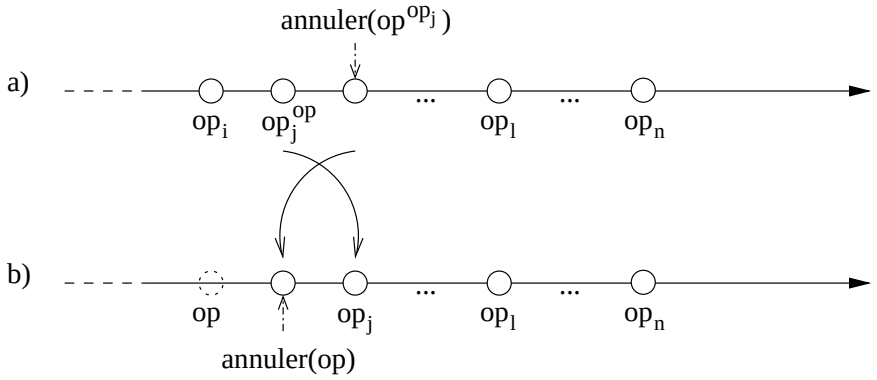


FIG. C.1 – Inverse de la transposition en avant d'une opération d'annulation

Supposons qu'on veuille transposer en arrière le couple d'opérations $(op_j^{op}, \texttt{annuler}(op^{op_j}))$ (Fig. C.1). Il faut dans un premier temps calculer l'inverse de la transposée en avant de $\texttt{annuler}(op^{op_j})$ par rapport à op_j^{op}. Or on a montré (cf. section 8.2.3) que la transposée en avant de $\texttt{annuler}(op)$ par rapport à op_j^{op} c'est l'annulation de la transposée en avant de op par rapport à l'inverse de la transposée en avant de op_j^{op} par rapport à op, c.-à-d. $\texttt{annuler}(op^{op_j})$. Il suffit donc de prendre l'inverse de la transposée obtenue. L'inverse de la transposée en avant de $\texttt{annuler}(op^{op_j})$ par rapport à op_j^{op} est donc $\texttt{annuler}(op)$.
La définition formelle est la suivante :

$$\texttt{Transpose\_av}^{-1}(op_j^{op}, \texttt{annuler}(op^{op_j})) = \texttt{annuler}(op)$$

Il suffit ensuite de transposer en avant op_j^{op} par rapport à $\texttt{annuler}(op)$. On l'a vu plus haut, cela revient à calculer l'inverse de la transposée en avant de op_j^{op} par rapport à op, ce qui donne op_j. La transposée en arrière du couple $(op_j^{op}, \texttt{annuler}(op^{op_j}))$ est donc le couple $(\texttt{annuler}(op), op_j)$.

$$\texttt{Transpose\_ar}(op_j^{op}, \texttt{annuler}(op^{op_j})) = (\texttt{annuler}(op), op_j)$$

Un problème subsiste néanmoins. En effet, si dans le cas du calcul de la transposée en arrière du couple $(\texttt{annuler}(op), op_j)$, il est possible à partir de $\texttt{annuler}(op)$ et de op_j de calculer op_j^{op} dans un premier temps puis op^{op_j} pour avoir $\texttt{annuler}(op^{op_j})$, il n'en est pas de même dans le cas du calcul de la transposée en arrière du couple $(op_j^{op}, \texttt{annuler}(op^{op_j}))$.

Malheureusement, dans ce cas de figure cela est impossible. En effet, il nous faut calculer $\texttt{annuler}(op)$. Or on connaît op^{op_j} à partir de $\texttt{annuler}(op^{op_j})$ et aussi op_j^{op}. Pour connaître op, on a donc besoin de connaître op_j et pour connaître op_j, on a besoin de connaître op. On est dans une situation où la connaissance des opérations ne suffit pas pour exécuter les fonctions de transpositions.

La solution consiste à utiliser l'histoire du site pour pouvoir effectuer la transposition en arrière d'une opération d'annulation. On sait, en effet qu'en retrouvant l'opération op, on pourra obtenir l'opération op_j. Or $\texttt{annuler}(op)$ est le résultat de l'inverse de la transposée en avant de $\texttt{annuler}(op^{op_j})$ par rapport à op_j^{op}. c.-à-d. l'opération qui, transposée en avant par rapport à op_j^{op}, a donné $\texttt{annuler}(op^{op_j})$. Il nous faut donc retrouver cette opération. La seule solution pour y parvenir consiste à retrouver, en réordonnant l'histoire,

l'opération `annuler`(op') qui, définie sur le même état que op_j^{op}, réalise l'annulation de op. On pourra alors calculer la transposée en avant de op_j^{op} par rapport à `annuler`(op) pour obtenir op_j.

L'exemple suivant permet d'illustrer ce problème. On dispose d'un objet O dont l'état initial est O_0. Sur l'état initial O_0 a été exécutée la séquence d'opérations suivante représentée par l'histoire :

$$H = op.seq.op_i,$$

Puis l'annulation de op a été demandée. Le processus d'annulation de op a été appliqué. L'opération `annuler`(op) définie sur l'état $O_o.op$ a été transposée en avant par rapport à la séquence seq ce qui a donné comme résultat :

$$\texttt{annuler}(op^{seq^{\frac{1}{op}}})$$

Où $seq^{\frac{1}{op}}$ représente l'inverse de la transposée en avant de seq par rapport à op.

Cette opération est à son tour transposée en avant par rapport à op_i ce qui donne :

$$\texttt{annuler}(op^{seq^{\frac{1}{op}}.op_i^{1/op^{seq^{1/op}}}})$$

Après l'annulation de op l'histoire est donc devenue :

$$H = op.seq.op_i.\texttt{annuler}(op^{seq^{\frac{1}{op}}.op_i^{1/op^{seq^{1/op}}}})$$

Pour transposer en arrière l'opération $annuler(op^{seq^{\frac{1}{op}}.op_i^{1/op^{seq^{1/op}}}})$ par rapport l'opération op_i il faut obtenir l'inverse de la transposée en avant de op_i par rapport à l'opération qui sur le même état (c.-à-d. $O_0.op.seq.op_i$) réalise la même intention que op. Cette opération est $op^{seq^{\frac{1}{op}}}$. Pour l'obtenir, il faudrait effectuer l'inverse de la transposée en avant de $op^{seq^{\frac{1}{op}}.op_i^{1/op^{seq^{1/op}}}}$ par rapport à $op_i^{1/op^{seq^{1/op}}}$. Cette dernière est l'inverse de la transposée en avant de op_i par rapport à l'opération qui sur l'état $O_0.op.seq.op_i$ réalise la même intention que op, c.-à-d. précisément l'opération qu'on recherche. On le voit il n'y a pas de solution au problème autre que de recalculer $op^{seq^{1/op}}$ à partir de l'histoire des exécutions, $H = op.seq.op_i$.

Il faut donc refaire les transpositions qui ont été faites pendant l'annulation de op. Cela permettra d'obtenir $op^{seq^{1/op}}$ qui servira à calculer $op_i^{1/op^{seq^{1/op}}}$. Une fois la transposition faite, l'histoire sera alors :

$$H = op.seq.\texttt{annuler}(op^{seq^{1/op}}).op_i^{1/op^{seq^{1/op}}}$$

D

Fonctions de transpositions génériques pour l'annulation

Sommaire

D.1 Introduction

Cette section définie les transpositions nécessaires à l'utilisation d'un algorithme de type SOCT2 pour gérer les accès concurrents à un objet du type chaine de caractères. L'ensemble d'opérations considéré permettant la manipulation de cet objet est le suivant : {`insérer`, `effacer`, `annuler`}. Ces opérations sont définis de la manière suivante :

`insérer(p, c)` : insère le caractère *c* à la position *p* dans la chaine de caractères,

`effacer(p, c)` : efface le caractère *c* se trouvant à la position *p* dans la chaîne.

`annuler(op)` : annule l'opération *op*.

Les algorithmes du type SOCT2 utilisent la fonction de transposition en avant (noté `Transpose_av`) ainsi que la fonction de transpositions en arrière (noté `Transpose_ar`). Il va donc falloir définir ces dernières pour chacun des couples d'opérations possibles.

La définition de `Transpose_av` est très largement issue de la définition donnée dans [Sul98]. Pour les transpositions en arrière, cependant, il m'a semblé plus judicieux de les définir en utilisant la forme générale donnée page 63 du même ouvrage. Celle-ci fait appel à la fonction $\texttt{Transpose\_av}^{-1}$ qui devra par conséquent être définie. L'auteur, avait lui donné des définitions de `Transpose_ar` dans lesquelles l'appel à $\texttt{Transpose\_av}^{-1}$ avait été remplacé directement par le résultat. Néanmoins, il n'a donné ces définitions directes que pour les couples d'opérations obtenus à partir de `insérer` et `effacer`. Or il existe un troisième type d'opérations, qui n'est pas utilisé par l'utilisateur mais par le système, les opérations `identité` pour lesquelles la définition de la transposée en arrière fait appel à $\texttt{Transpose\_av}^{-1}$.

La forme générique de la définition de `Transpose_ar`, est la suivante :

$$\texttt{Transpose\_ar}(op_1, op_2) = (op'_2, op'_1), \text{ avec :}$$
$$op'_2 = \texttt{Transpose\_av}^{-1}(op_1, op_2) \text{ et}$$
$$op'_1 = \texttt{Transpose\_av}(op'_2, op_1).$$

Voyons à présent les définitions de `Transpose_av` et $\texttt{Transpose\_av}^{-1}$ pour les couples d'opérations issues de {`insérer, effacer, identité, annuler`}.

D.2 Transpositions en avant

D.2.1 Transpose_av générique

```
Transpose_av(op1, op2) =
    cas (op1 =annuler(op'1))
          cas (op2 = annuler(op'2) et op'1 == op'2) retour identité(op2) ;
          sinon retour Transpose_av^-1(op'1, op2) ;
          fincas
        (op2 = annuler(op'2))
              retour annuler(Transpose_av(Transpose_av^-1(op'2, op1), op'2)) ;
```

```
        (op2 = identité(op'2)) et (op1 = identité(op'1))
              retour identité(Transpose_av(op'1, op'2)) ;
        (op1 = identité(op'1))
              retour op2 ;
        sinon retour Transpose_av_base(op1, op2) ;
    fincas
```

Où == dénote que le vecteur d'état de l'opération op'_1 est égal au vecteur d'état de l'opération op'_2 et qu'elles ont toutes deux été générées sur le même site (c.-à-d. $\mathtt{SV}_{op'_1} = \mathtt{SV}_{op'_2}$ et $\mathtt{S}_{op'_1} = \mathtt{S}_{op'_2}$).

D.2.2 insérer/insérer

```
Transpose_av_base(insérer(p1, c1, av1, ap1), insérer(p2, c2, av2, ap2)) =
    cas (p1 < p2) retour insérer(p2 + 1, c2, av2, ap2) ;
        (p1 > p2) retour insérer(p2, c2, av2, ap2) ;
        (p1 = p2) cas (ap1 ∩ av2 ≠ ∅) : retour insérer (p2 + 1, c2, av2, ap2)
                      (av1 ∩ ap2 ≠ ∅) retour insérer(p2, c2, av2, ap2) ;
                      (code(c1) < code(c2)) retour insérer(p2, c2, av2, ap2) ;
                      (code(c2) < code(c1)) retour insérer(p2 + 1, c2, av2, ap2) ;
                      (code(c1) = code(c2)) retour identité(insérer(p2, c2, av2, ap2))
                  fincas
    fincas
```

D.2.3 effacer/insérer

```
Transpose_av_base(effacer(p1, c1), insérer(p2, c2, av2, ap2)) =
    cas (p1 < p2) retour insérer(p2 − 1, c2, av2∪{effacer(p1, c1)}, ap2) ;
        (p1 ≥ p2) retour insérer(p2, c2, av2, ap2∪{effacer(p1, c1)}) ;
    fincas
```

D.2.4 insérer/effacer

```
Transpose_av_base(insérer(p1, c1, av1, ap1), effacer(p2, c2)) =
    cas (p1 ≤ p2) retour effacer(p2 + 1, c2) ;
        (p1 > p2) retour effacer(p2, c2) ;
    fincas
```

D.2.5 effacer/effacer

```
Transpose_av_base(effacer(p1, c1), effacer(p2, c2)) =
    cas (p1 < p2) retour effacer(p2 − 1, c2) ;
        (p1 > p2) retour effacer(p2, c2) ;
        (p1 = p2) retour identité(effacer(p2, c2)) ;
    fincas
```

D.3 Inverse de la Transposition en avant

D.3.1 Transpose_av^{-1} générique

```
Transpose_av^-1(op1, op2) =
    cas (op1 = annuler(op'1))
        cas (op2 = id(op'2) et SV_op1 = SV_op'2) retour op'2 ;
              sinon retour Transpose_av(op1,op2) ;
    cas (op1 = id(op'1))
              (op1 ≠ op2) retour Transpose_av^-1(op'1, op2) ;
            fincas
        (op2 = annuler(op'2))
              Utiliser l'histoire du site pour re-calculer
              l'opération op''2 telle que Transpose_av(
              annuler(op''2), op1) = annuler(op'2), en transposant
              en arrière l'opération dont annuler(op'2)
              réalise l'annulation.
        (op2 = identité(op'2))
              retour identité(Transpose_av(Transpose_av^-1(op'2, op1), op'2)) ;
        (op1 = identité(op'1))
              retour op2 ;
        sinon retour Transpose_av_base(op1, op2) ;
    fincas
```

D.3.2 insérer/insérer

```
Transpose_av^-1_base(insérer(p1, c1, av1, ap1), insérer(p2, c2, av2, ap2)) =
    cas (p1 < p2) retour insérer(p2 − 1, c2, av2, ap2) ;
        (p1 ≥ p2) retour insérer(p2, c2, av2, ap2) ;
    fincas
```

D.3.3 effacer/insérer

```
Transpose_av^-1_base(effacer(p1, c1), insérer(p2, c2, av2, ap2)) =
    cas (p1 < p2) retour insérer(p2 + 1, c2, av2\{effacer(p1, c1)}, ap2) ;
        (p1 > p2) retour insérer(p2, c2, av2, ap2\{effacer(p1, c1)}) ;
        (p1 = p2) cas (effacer(p1, c1)∈ av2)
                        retour insérer(p2 + 1, c2, av2\{effacer(p1, c1)}, ap2) ;
                  (effacer(p1, c1)∉ av2)
                        retour insérer(p2 + 1, c2, av2, ap2\{effacer(p1, c1)}) ;
    fincas
```

D.3.4 insérer/effacer

Transpose_av^{-1}_base(insérer(p_1, c_1, av_1, ap_1), effacer(p_2, c_2)) =
    cas ($p_1 < p_2$) retour effacer($p_2 - 1, c_2$) ;
        ($p_1 > p_2$) retour effacer(p_2, c_2) ;
    fincas

D.3.5 effacer/effacer

Transpose_av^{-1}_base(effacer(p_1, c_1), effacer(p_2, c_2)) =
    cas ($p_1 \leq p_2$) retour effacer($p_2 + 1, c_2$) ;
        ($p_1 > p_2$) retour effacer(p_2, c_2) ;
    fincas

Bibliographie

[ABB96] Acharya A., Bakre A., et Badrinath B. R. « IP Multicast Extensions for Mobile Internetworking ». Dans *Proc. of the 15-th IEEE Conference on Computer Communications, Infocom'96*, volume 1, pages 67–74, San Fransisco, California, USA, 24–28 mars 1996.

[AD92] Abowd G. D. et Dix A. J. « Giving Undo Attention ». *Interacting with Computers*, 4(3) : 317–342, 1992, Elsevier Science Inc.

[All89] Allison C. « Concurrency Control for Real Time Groupware ». Dans Paul A. et Sobolewski M., éditeurs, *Concurrent Engineering : Research and Applications. A global Perspective (CE94)*, pages 163–170, Pittsburgh, Pennsylvania, USA, août 1989. CTC.

[BAI94] Badrinath B. R., Acharya A., et Imilelinski T. « Designing Distributed Algorithms for Mobile Computing Networks ». *Computers and Communications*, 19(4) : 309–320, avril 1994.

[BBIM93] Badrinath B. R., Bakre A., Imielinski T., et Marantz R. « Handling Mobile Clients : A Case for Indirect Intercation ». Dans *Proceedings Fourth Workshop on Workstation Operating System*, pages 91–97, Napa, California, USA, 14–15 octobre 1993. IEEE Computer Society Press.

[BCF+97] Besancenot J., Cart M., Ferrié J., Guerraoui R., Pucheral P., et Traverson B. *Les systèmes transactionnels – concepts, normes et produits.* collection informatique. Hermes, novembre 1997.

[Ber94] Berlage T. « A Selective Undo Mechanism for Graphical User Interfaces Based on Command Objects ». *ACM Transactions on Computer-Human Interaction*, 1(3) : 269–294, septembre 1994, ACM Press.

[BI92] Badrinath B. R. et Imielinski T. « Replication and Mobility ». Dans *Second IEEE Workshop on Management of Replicated Data*, pages 9–12, Monterey, California, USA, novembre 1992.

[BKZ79] Banino J., Kaiser C., et Zimmermann H. « Synchronization for Distributed Systems using a Single Broadcast Channel ». Dans *Proceedings of the* 1^{st} *International Conference on Distributed Computing Systems*, pages 330–338, Huntsville, Alabama, USA, octobre 1979. IEEE Computer Society.

[BM00] Bouazza A. et Molli P. « Unifying coupled and uncoupled collaborative work in virtual teams ». Dans *ACM CSCW'2000 workshop on collaborative editing systems*, Philadelphia, Pennsylvania, USA, décembre 2000.

[BMGM94] Barbara-Milla D. et Garcia-Molina H. « Replicated Data Management in Mobile Environments : Anything New Under the Sun ? ». Dans Girault C., éditeur, *Applications in Parallel and Distributed Computing, Proceedings of the IFIP WG10.3 Working Conference on Applications in Parallel and Distributed Computing*, volume A-44 de *IFIP Transactions*, pages 237–246, Caracas, Venezuela, avril 1994. North Holland.

[BP98] Balasubramaniam S. et Pierce B. C. « What is a file synchronizer ? ». Dans *Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '98)*, octobre 1998.

[BR92] Badrinath B. R. et Ramamritham K. « Semantics-Based Concurrency Control : Beyond Commutativity ». *ACM Transactions on Database Systems*, 17(1) : 163–199, mars 1992.

[CD95] Choudhary R. et Dewan P. « A General Multi-User Undo/Redo Model ». Dans *Proceedings of the Fourth European Conference on Computer-Supported Cooperative Work*, CSCW Mechanisms II, pages 231–246, 1995.

[CGM85] Cordon R. et Garcia-Molina H. « The Performance of a Concurrency Control Mechanism That Exploits Semantic Knowledge ». Dans *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 350–358, Denver, Colorado, mai 1985. IEEE Computer Society.

[CL88] Carey M. J. et Livny M. « Distributed Concurrency Control Performance : A Study of Algorithms, Distribution, and Replication ». Dans Bancilhon F. et DeWitt D. J., éditeurs, *Proc. 14th Intl. Conf. on Very Large Data Bases*, pages 13–25, Los Angeles, California, 29 août – 1 septembre 1988. Morgan Kaufman.

[Cor95] Cormack G. V. « A Calculus for Concurrent Update (Abstract) ». Dans *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, page 269, Ottawa, Ontario, Canada, 2–23 août 1995. ACM Press.

[CS99] Chen D. et Sun C. « A Distributed Algorithm for Graphic Objects Replication in Real-time Group Editors ». Dans Hayne S. C., éditeur, *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work (GROUP'99)*, ACM Siggroup, pages 121–130, Phoenix, Arizona, 14–17 novembre 1999. ACM Press.

[EG89] Ellis C. A. et Gibbs S. J. « Concurrency control in group systems ». *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 18(2) : 399–407, juin 1989.

[EGR91] Ellis C. A., Gibbs S. J., et Rein G. L. Groupware : Some issues and experiences. Dans *Communications of the ACM*, volume 34, 1, pages 38–58. ACM Press, New York, NY, janvier 1991.

[EMP+97] Edwards W. K., Mynatt E. D., Petersen K., Spreitzer M., Terry D. B., et Theimer M. « Designing and Implementing Asynchronous Collaborative Applications with Bayou ». Dans *Proceedings of the 10th annual*

ACM symposium on User interface software and technology (UIST'97), pages 119–128. ACM Press, 14–17 octobre 1997.

[GFP95] Guerni M., Ferrié J., et Pons J.-F. « Concurrency and Recovery for Typed Objects using a New Commutativity Relation ». Dans Tok Wang Ling L. V., Alberto O.Mendelzon, éditeur, *Proceedings of the 4^{th} International Conference on Deductive and Object-Oriented Databases (DOOD '95)*, numéro 1013 dans Lectures Notes In Computer Science, pages 411–428. Springer, 1995.

[Her85] Herlihy M. « Atomicity vs. Availability : Concurrency Control for Replicated Data ». Rapport Technique CMU-CS-85-108, Carnegie-Mellon University, Department of Computer Science, Pittsburgh, PA, février 1985.

[KBL93] Karsenty A. et Beaudouin-Lafon M. « An Algorithm for Distributed Groupware Applications ». Dans Werner R., éditeur, *Proceedings of the 13^{th} International Conference on Distributed Computing Systems*, pages 195–202, Pittsburgh, PA, mai 1993. IEEE Computer Society Press.

[KS88] Kumar A. et Stonebraker M. « Semantics Based Transaction Management for Replicated Data ». Dans Boral H. et Larson P.-A., éditeurs, *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 117–126, Chicago, Illinois, 1–3 juin 1988. ACM Press.

[KTW96] Klingemann J., Tesch T., et Wäsch J. « Semantics-based transaction management for cooperative applications ». Dans *International Workshop on Advanced Transactions Models and Architectures (ATMA)*, pages 234–252, GOA, India, 31 août – 2 septembre 1996.

[KTW97] Klingemann J., Tesch T., et Wäsch J. « Enabling Cooperation among Disconnected Mobile Users ». Dans *Proceedings of the 2^{nd} IFCIS International Conference on Cooperative Information Systems (CoopIS '97)*, pages 36–45, juin 1997.

[Lam78] Lamport L. « Time, Clocks, and the Ordering of Events in a Distributed System ». *Communications of the ACM*, 21(7) : 558–565, juillet 1978.

[LL78] Le Lann G. « Algorithms for distributed data sharing systems which use tickets ». Dans *Proceedings of the 3^{rd} Workshop on Distributed Data Management and Computer Networks*, Berkeley, août 1978.

[LM99] Li D. et Muntz R. R. « Runtime Dynamics in Collaborative Systems ». Dans Hayne S. C., éditeur, *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work (GROUP'99)*, ACM Siggroup, pages 336–345, Phoenix, Arizona, 14–17 novembre 1999. ACM Press.

[LSZM00] Li D., Sun C., Zhou L., et Muntz R. R. « Operation Propagation in Real-Time Group Editors ». *IEEE MultiMedia*, 7(4) : 55–61, octobre 2000.

[LZM00] Li D., Zhou L., et Muntz R. R. « A new paradigm of user intention preservation in realtime collaborative editing systems ». Dans *Proceedings of The 7^{th} International Conference on Parallel and Distributed Systems (ICPADS'2000)*, Iwate, Japan, juillet 2000.

[Mat89] MATTERN F. « Virtual time and global states of distributed systems ». Dans COSNARD M. et OTHERS , éditeurs, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Amsterdam, 1989. Elsevier Science Publishers.

[Mau00] MAUVE M. « Consistency in Continuous Distributed Media ». Dans *Proceedings of the ACM 2000 Conference on Computer Supported Cooperative Work (CSCW-00)*, pages 181–190. ACM Press, 2000.

[MCFP96] MORPAIN C., CART M., FERRIÉ J., et PONS J.-F. « Maintaining database consistency in presence of value dependencies in multidatabase systems ». Dans *Proceedings of theACM SIGMOD International Conference on Management of Data*, volume 25, 2 de *ACM SIGMOD Record*, pages 459–468. ACM Press, 1996.

[MDL96] MANCINI R., DIX A., et LEVIALDI S. « Reflections on Undo ». Rapport Technique RR9611, University of Huddersfield, 1996.

[MSMO02] MOLLI P., SKAF-MOLLI H., et OSTER G. « Divergence Awarenes for Virtual Team through the Web ». Dans *Proceedings of the* 6^th *International Conference on Integrated Design and Process Technology (IDPT–2002)*, 2002. *(to appear).*

[NCDL95] NICHOLS D. A., CURTIS P., DIXON M., et LAMPING J. « High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System ». Dans *Proceedings of the ACM Symposium on User Interface Software and Technology*, Distributed User Interfaces, pages 111–120, 1995.

[PC98] PALMER C. R. et CORMACK G. V. « Operation Transforms for a Distributed Shared Spreadsheet ». Dans *Proceedings of ACM CSCW'98 Conference on Computer-Supported Cooperative Work*, Concurrency and Consistency, pages 69–78. ACM Press, 1998.

[PK92] PRAKASH A. et KNISTER M. J. « Undoing actions in collaborative work ». Dans *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'92)*, Consistency in collaborative systems, pages 273–280, Toronto, Ontario, 1992. ACM Press.

[PK94] PRAKASH A. et KNISTER M. J. « A Framework for Undoing Actions in Collaborative Systems ». *ACM Transactions on Computer-Human Interaction*, 1(4) : 295–330, 1994.

[PSL99] PRAKASH A., SHIM H. S., et LEE J. H. « Data Management Issues and Trade-Offs in CSCW Systems ». *Knowledge and Data Engineering*, 11(1) : 213–227, 1999.

[RG99] RESSEL M. et GUNZENHÄUSER R. « Reducing the Problems of Group Undo ». Dans HAYNE S. C., éditeur, *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work (GROUP'99)*, ACM Siggroup, pages 131–139, Phoenix, Arizona, 14–17 novembre 1999. ACM Press.

[RK79] REED D. P. et KANODIA R. K. « Synchronization with Eventcounts and Sequencers ». *Communications of the ACM*, 22(2) : 115–123, février 1979.

[RKT+95] Rusinkiewicz M., Klas W., Tesch T., Waesch J., et Muth P. « Towards a Cooperative Transaction Model : The Cooperative Activity Model ». Dans *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB '95)*, pages 194–205. Morgan Kaufmann Publishers, Inc., 1995.

[RNRG96] Ressel M., Nitsche-Ruhland D., et Gunzenhäuser R. « An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors ». Dans *Proceedings of the ACM 1996 Conference on Computer Supported Work*, pages 288–297, New York, 16–20 novembre 1996. ACM Press.

[SBM+97] Strom R., Banavar G., Miller K., Prakash A., et Ward M. « Concurrency Control and View Notification Algorithms for Collaborative Replicated Objects ». Dans *17th International Conference on Distributed Computing Systems (ICDCS'97)*, pages 194–203, Baltimore, MD, mai 1997. IEEE.

[SCF97] Suleiman M., Cart M., et Ferrié J. « Serialization Of Concurrent Operations In A Distributed Collaborative Environment ». Dans Hayne S. C. et Prinz W., éditeurs, *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work : The Integration Challenge (GROUP-97)*, pages 435–445. ACM Press, 16–19 novembre 1997.

[SCF98] Suleiman M., Cart M., et Ferrié J. « Concurrent operations in a Distributed and Mobile Collaborative Environment ». Dans *Proceedings of the* 14^{th} *IEEE International conference on Data Engineering*, pages 36–45, février 1998.

[SE98] Sun C. et Ellis C. S. « Operational Transformation in Real-Time Group Editors : Issues, Algorithms, and Achievements ». Dans *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW-98)*, pages 59–68, New York, 14–18 novembre 1998. ACM Press.

[SJZ+98] Sun C., Jia X., Zhang Y., Yang Y., et Chen D. « Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems ». *ACM Transactions on Computer-Human Interaction*, 5(1) : 63–108, 1998.

[SJZY97] Sun C., Jia X., Zhang Y., et Yang Y. « A Generic Operation Transformation Scheme For Consistency Maintenance In Real-Time Cooperative Editing Systems ». Dans Hayne S. C. et Prinz W., éditeurs, *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work : The Integration Challenge (GROUP-97)*, pages 425–434. ACM Press, 16–19 novembre 1997.

[SS02] Saito Y. et Shapiro M. « Replication : Optimistic Approaches ». Rapport Technique HPL-2002-33, HP Labs, 2002.

[Sul98] Suleiman M. « *Sérialisation des opérations concurrentes dans les systèmes collaboratifs répartis* ». PhD thesis, Université de Montpellier II, juillet 1998.

[Sun00] Sun C. « Undo Any Operation at Any Time in Group Editors ». Dans *Proceedings of the ACM 2000 Conference on Computer Supported Cooperative Work (CSCW-00)*, pages 191–200. ACM Press, 2000.

[TTP+95] Terry D. B., Theimer M. M., Petersen K., Demers A. J., Spreitzer M. J., et Hauser C. H. « Managing update conflicts in Bayou, a weakly connected replicated storage system ». Dans Jones M. B., éditeur, *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 172–182. ACM Press, 3–6 décembre 1995.

[VCFS00] Vidot N., Cart M., Ferrié J., et Suleiman M. « Copies Convergence in a Distributed Real-Time Collaborative Environment ». Dans *Proceedings of the ACM 2000 Conference on Computer Supported Cooperative Work (CSCW-00)*, pages 171–180. ACM Press, 2000.

[Wei88] Weihl W. E. « Commutativity-Based Concurrency Control for Abstract Data Types ». *IEEE Transactions on Computers*, 37(12) : 1488–1505, décembre 1988.

Table des figures

Liste des exemples

Convergence des Copies dans les Environnements Collaboratifs Répartis

Résumé.

Dans les environnements collaboratifs répartis temps réel, les objets répliqués, partagés par les utilisateurs sont soumis à des contraintes de concurrence. Pour les satisfaire, différents algorithmes de contrôle, exploitant les propriétés sémantiques des opérations et utilisant les Transformées opérationnelles, ont été proposés de façon à ordonner les opérations concurrentes et par-là garantir la convergence des copies d'un objet tout en respectant l'intention des usagers. Leur inconvénient est d'imposer ou bien que les opérations utilisées par les utilisateurs satisfassent une condition, difficile à vérifier et qu'il n'est pas toujours possible d'assurer, ou bien de défaire puis de refaire des opérations dans certaines situations.

Le premier objectif de la thèse est de présenter deux nouveaux algorithmes n'ayant pas ces défauts. Ils reposent sur la mise en œuvre d'un ordre global continu qui permet non seulement de s'affranchir de cette condition mais aussi de simplifier le processus d'intégration d'une opération. Dans l'un des algorithmes, SOCT4 dont nous donnons la preuve formelle, l'intégration est encore simplifiée en différant la diffusion des opérations alors que dans l'autre, SOCT5, le parallélisme dans la diffusion est privilégié. L'extension de ces algorithmes pour tenir compte de la présence de postes mobiles est abordée.

Le deuxième objectif est d'adapter les algorithmes de type SOCT2 pour permettre à un utilisateur d'annuler une opération dans la mesure où les rares propositions non restrictives faites pour résoudre ce problème compromettent dans certains cas la convergence des copies. Pour cela, plutôt que de manipuler directement l'opération inverse, on considère l'annulation comme une opération spécifique dont la transposition en avant doit satisfaire deux conditions générales que l'étude met en évidence. Le bien fondé de cette démarche est validée par l'étude de cas critiques.

Mots-clés: système collaboratif réparti, cohérence de copies, respect de l'intention, transformée opérationnelle, transposition, annulation, mobilité.

Copies convergence in Distributed Collaborative Environments

Abstract.

In real time collaborative systems, replicated objects, shared by users, are subject to concurrency constraints. In order to satisfy these, various algorithms, have been proposed that exploit the semantic properties of operations and use Operational transformations to serialize concurrent operations and achieve copy convergence of replicated objects, while preserving users intention. Their drawback is that they either require a condition on user's operations which is hard to verify when possible to ensure, or they need undoing then redoing operations in some situations.

The first objective of the thesis is to present two new algorithms that overcome these drawbacks. They are based upon the implementation of a continuous global order which enables that condition to be released, and simplifies the operation integration process. In one of these algorithms, SOCT4 for which we give a formal proof, this process becomes even more simplified thanks to deferred broadcast whereas in the other one, SOCT5, priority is given to concurrency during the broadcast. An extension to these algorithm to support mobile sites is introduced.

The second objective is to adapt SOCT2 type algorithms to make it possible for a user to undo an operation insofar as the rare nonrestrictive proposals made to solve this problem compromise in certain cases the convergence of the copies. For that, rather than to handle the opposite operation directly, we consider undo as a specific operation whose forward transposition must satisfy two general conditions that the study highlights. The reasonableness of this process is validated by the study of critical cases.

Keywords: distributed collaborative system, copies consistency, intention preservation, operational transformation, transposition, undo, mobility.