Partage d'objets dans les systèmes distribués. Principes des ramasse-miettes

André Couvert, Aomar Maddi, René Pedrono

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 963

Programme 1

PARTAGE D'OBJETS DANS LES SYSTÈMES DISTRIBUÉS PRINCIPES DES RAMASSE-MIETTES

**André COUVERT, Aomar MADDI,
René PEDRONO**

Janvier 1989

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042-RENNES CÉDEX
FRANCE
Téléphone: 99 36 20 00
Télex: UNIRISA 950 473 F
Télécopie: 99 38 38 32

Partage d'objets dans les systèmes distribués Principes des ramasse-miettes *

Objects Sharing in Distributed Systems Principles of garbage collection

André Couvert, Aomar Maddi, René Pédrono
I.R.I.S.A
Campus universitaire de Beaulieu
avenue du Général Leclerc
35042 Rennes cedex

Publication Interne n° 419

Juin 1988 - 60 Pages

* Ce travail, supporté en partie par le GRECO C^3, a été réalisé dans l'équipe ADP de l'IRISA dirigée par Michel RAYNAL.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

PARTAGE D'OBJETS DANS LES SYSTEMES DISTRIBUES

PRINCIPE DES RAMASSE-MIETTES

André COUVERT, Aomar MADDI, René PEDRONO

Juin 1988

Publication Interne n° 419

Résumé

Ce rapport présente et compare les différentes approches proposées dans la littérature pour résoudre le problème de la gestion dynamique de la mémoire dans les systèmes distribués.

Après avoir précisé un modèle pour la gestion des objets d'un tas et les techniques de récupération d'objets inaccessibles dans un contexte centralisé, nous introduisons un mécanisme de partage d'objets entre processus et d'accès aux objets partageables dans un contexte distribué. Puis, nous analysons, de manière détaillée, les algorithmes de ramasse-miettes reposant sur les concepts de : compteurs de références, marquage global des objets et récupération différée des objets partageables. Enfin, les avantages et les inconvénients de chacune des méthodes sont mentionnés et les qualités espérées d'un "bon" algorithme de ramasse-miettes sont exhibées.

Mots clés

Gestion d'objets, ramasse-miettes, compteurs de références, marquage, système distribué, partage d'objets, références pondérées, arborescence de contrôle.

Abstract

This paper presents and compares various approaches proposed in the literature to solve the problem of dynamic memory management in distributed systems.

After the presentation of a heap object managing model and methods to retrieve inaccessible objects in a centralized context, we introduce a mechanism for object sharing between processes and access to shared objects in a distributed context. Then we propose a detailed analysis of garbage-collection algorithms depending on the following concepts : reference counters, global marking of objects and delayed recovery of shared objects. Finally, we mention the advantages and disadvantages of the various methods and pinpoint the qualities that a " good" garbage-collection algorithm should have.

Keywords

Object management, garbage-collectors, reference counters, marking, distributed systems, objects sharing, weighted references, control tree structure.

Avant propos

Dans ce rapport, nous présentons les différentes approches proposées dans la littérature pour résoudre le problème de la gestion dynamique de la mémoire dans les systèmes distribués.

Après avoir introduit, dans le paragraphe I, la terminologie employée, nous rappelons, dans le paragraphe II, les deux techniques utilisées dans le contexte centralisé. Dans le paragraphe III nous choisissons un mécanisme pour le partage d'objets dans un système distribué et dans les paragraphes IV et V nous présentons les méthodes reposant, respectivement, sur les concepts de compteurs de références et de marquage. Enfin, le paragraphe VI est consacré à l'étude des algorithmes provoquant une récupération différée des objets.

Sommaire

I Un modèle pour la gestion des objets

Un programme utilisateur, communément appelé *mutateur* [DIJ 78], accède à un espace mémoire, le *tas*, géré dynamiquement. Pour cela, il dispose de primitives d'acquisition *(allouer)*, de libération explicite *(libérer)* ou de modification d'accès *(affectation de pointeurs)* dans le tas. L'adresse d'un élément du tas *(pointeur)* possède une taille fixe et connue; par contre, l'objet repéré est de taille quelconque et peut, ou non, contenir des pointeurs. Un objet du tas est accessible par le mutateur si on peut l'atteindre, directement ou indirectement, depuis un objet de la pile associée à l'application :

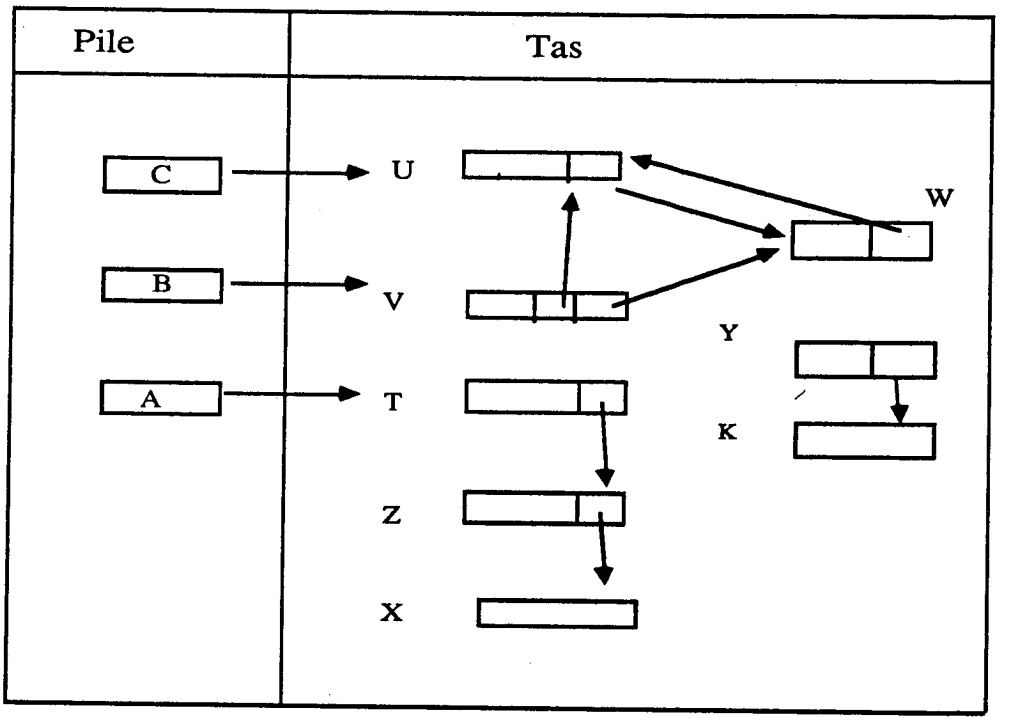


Figure I.1

T, U, V, W, X, Z sont des objets accessibles; Y et K sont inaccessibles; le reste du tas est libre.

Conformément à la démarche employée par tous les auteurs, nous faisons, par la suite, abstraction de la pile en la remplaçant par un objet unique et indestructible : la racine R_A des objets accessibles dans le tas. La figure I.1 devient alors :

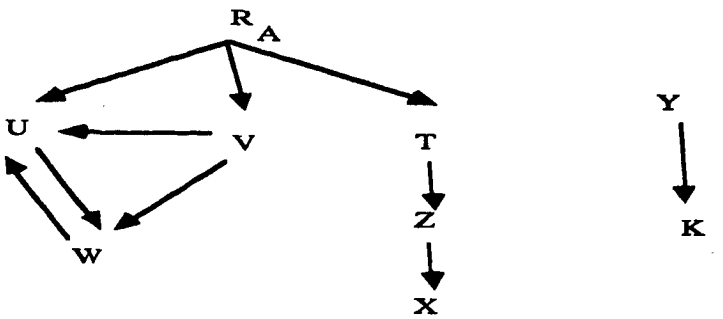


Figure I.2

Nous appelons *successeur* d'un objet *O* , au sens arc dans un graphe, tout objet du tas repéré par l'un des composants de *O;* la notion inverse est celle de *prédécesseur* d'un objet et la *fermeture transitive* de ces relations (i.e. l'existence d'un chemin) nous amène à parler de *descendants* ou *d'ascendants*.

A chaque instant le tas G est partitionné en trois graphes :

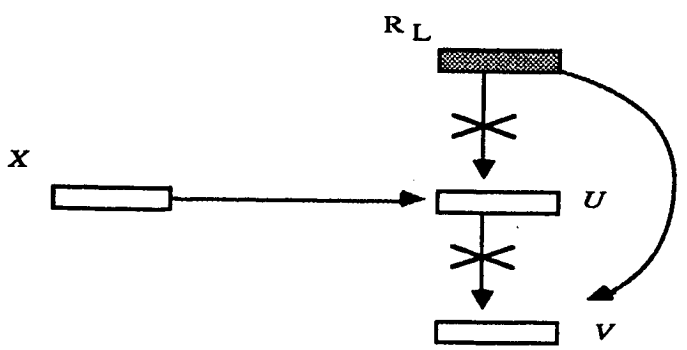
- G_A, graphe des objets accessibles depuis R_A
- G_L, graphe des objets libres
- G_I, graphe des objets inaccessibles et non libres.

Le graphe G_L, souvent appelé liste des libres, possède une racine indestructible R_L.

Le mutateur intervient sur les objets de $G_A \cup G_L$ à l'aide des opérations *allouer*, *libérer* et *affectation de pointeurs* (notée :-) exprimées en termes d'*ajouts* ou de *retraits* d'arcs dans G :

- *allouer (X)* , $X \in G_A, U \in G_L \Rightarrow$ ajout (*X*, *U*) ; ajout (R_L, *V*); retrait(R_L, *U*) ; retrait(*U*, *V*);

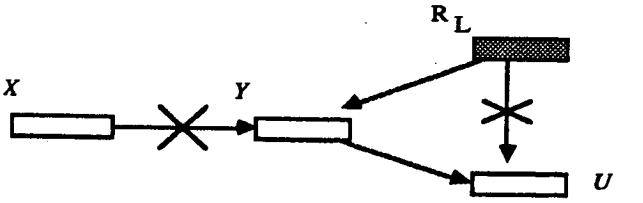
  puis $U \in G_A$



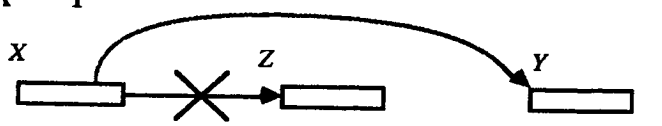
Les flèches ⟶ indiquent les ajouts d'arcs effectués

tandis que ⤫⟶ précisent les retraits d'arcs.

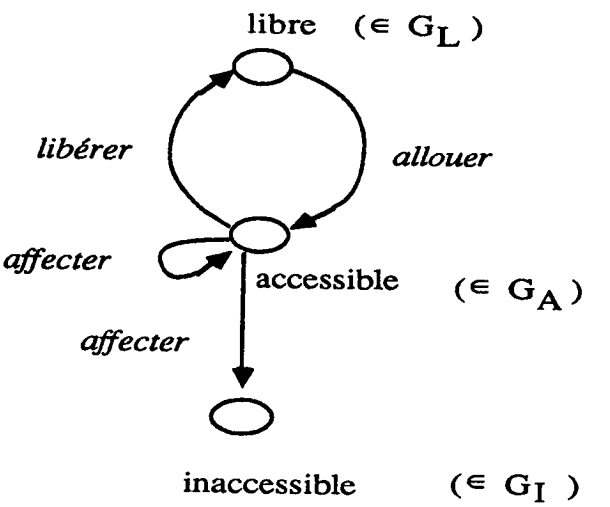
- *libérer (X)*, $X, Y \in G_A$, $\Rightarrow$ ajout (R_L, *Y*) ; ajout (*Y*, *U*) ; retrait(R_L, *U*) ; retrait(*X*, *Y*) ;
  puis $Y \in G_L$



- *X* :- *Y* , $X, Y, Z \in G_A$ $\Rightarrow$ ajout (*X*, *Y*) ; retrait (*X*, *Z*);
  puis $Z \in G_A \cup G_I$



Ainsi, l'état d'un objet peut évoluer de la façon suivante :



Le problème à résoudre est le passage des objets de G_I vers G_L et nous ignorons par la suite l'opération *libérer*.

Le rôle d'un algorithme de ramasse-miettes, appelé *collecteur* , est de faire l'union de G_L et de tout ou partie de G_I afin d'assurer que toute demande d'allocation d'espace est acceptée tant que l'espace non accessible depuis R_A est suffisant pour satisfaire cette demande.

Lors de chaque phase de collection, ou lorsqu'une demande d'allocation nécessite une taille de zone non disponible dans le tas, un tassement (ou compactage) du tas peut s'avérer indispensable; les techniques liées à ce type de traitement ne sont pas abordées dans cet article ([COH 81] fournit de nombreuses références sur ce sujet et [COH 83] compare plusieurs méthodes pour le tassement de la mémoire).

II Gestion d'un tas en contexte centralisé

Ce sujet a été abondamment traité et une remarquable synthèse est proposée dans [COH 81]; aussi, nous ne rappelons pas ici tous les articles consacrés à cette étude mais nous présentons les deux grandes familles d'algorithmes :

- les compteurs de références,
- le marquage et la récupération des objets non marqués

car les méthodes employées dans un environnement distribué s'en inspirent très largement.

II.1 Techniques par compteurs de références

Dans les propositions basées sur ce principe il n'y a pas une véritable distinction entre le collecteur et le mutateur; c'est, en fait, ce dernier qui prend totalement en charge la gestion des informations permettant de déterminer si un objet est accessible ou non. Le rôle du collecteur est alors limité à la mise en file des objets libérés et au retassement éventuel du tas.

La méthode consiste à associer, à chaque objet, un champ CR entier comptabilisant le nombre de ses prédécesseurs. Pour un objet accessible, les opérations du mutateur mettent CR à jour et l'objet devient libre (i.e. peut être ajouté en tête de la liste des libres G_L) lorsque son champ CR devient égal à 0 . Les opérations du mutateur considérées sont :

pour X, Y $\in G_A$:

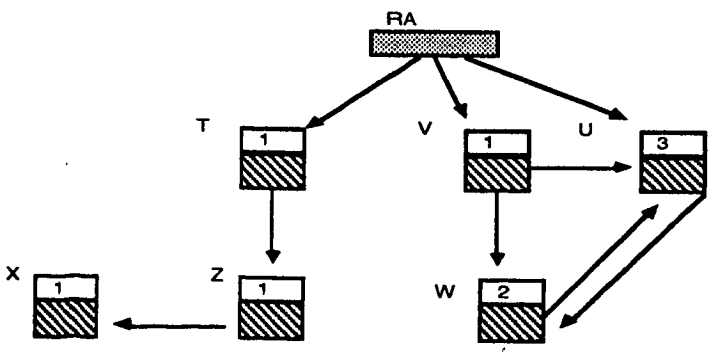
retrait(X,Y) $\Rightarrow$ CR(Y) := CR(Y) - 1

ajout(X,Y) $\Rightarrow$ CR(Y) := CR(Y) + 1

CR(X) = 0 $\Rightarrow$ mise de X dans G_L et éventuellement décrément de 1 des compteurs des successeurs immédiats de X (dans certaines mises en œuvre ce travail est fait dans allouer)

allouer(X) $\Rightarrow$ prend un élément V de G_L et effectue CR(V):=1

- Exemple :
  - * Soit le graphe suivant :



* retrait(T, Z)
ajout(T, V) ⇒

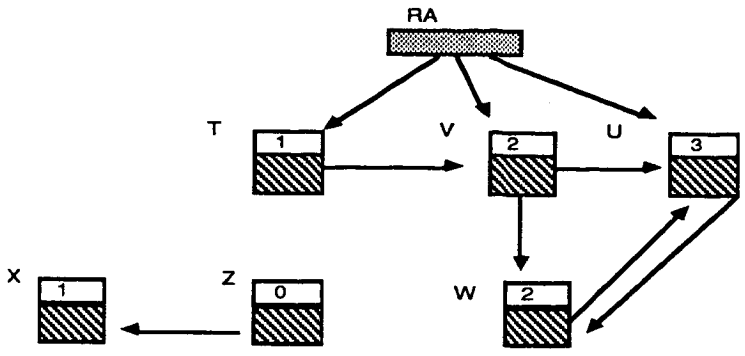


Figure II.1

Z (éventuellement X) est placé dans G_L.

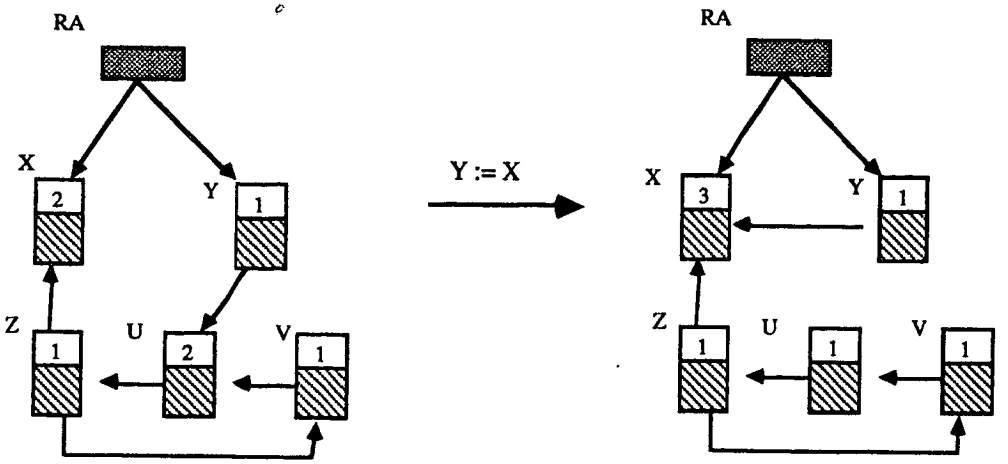
L'emploi de compteurs de références entraîne une bonne répartition, dans le temps, du travail de l'application et de la détection des objets libres; par ailleurs, la gestion des compteurs est simple et ne nécessite pas d'interruption du programme utilisateur. La récupération est immédiate, lorsque le compteur d'un objet Z passe à 0 l'objet est ajouté à G_L; pour la descendance de Z on peut :

- choisir de la traiter tout de suite; on décrémente les compteurs des successeurs de Z qui, à leur tour , peuvent éventuellement être ajoutés à G_L,
- différer son traitement; lorsque Z est à nouveau réattribué à l'application, on décrémente les compteurs des successeurs de Z qui peuvent, alors, devenir libres.

Les inconvénients majeurs de l'approche par compteurs de références résident dans :

- l'obligation imposée à l'application de gérer les compteurs; ceci doit être rendu transparent à l'utilisateur (par un compilateur par exemple),
- la pénalité infligée au mutateur lors des opérations d'ajouts ou de retraits d'arcs dans G_A. [APP 87] pense que le faible coût de la mémoire permet d'envisager un vaste espace d'adressage : il n'est, alors, plus nécessaire d'imposer au mutateur du travail destiné à la récupération; lorsqu'une saturation du tas intervient, on active un algorithme de type marquage (cf II.2),
- l'encombrement inhérent aux compteurs; les champs compteurs de références sont souvent utilisés pour chaîner les objets libres, leur taille est donc celle d'un pointeur,

- l'inadéquation de la méthode à détecter les structures circulaires inaccessibles :



Le circuit inaccessible (Z, U, V) n'est pas récupérable.
Figure II.2

Pour borner la taille des compteurs [WEI 69], [KNU 73] et [DEU 76] proposent des techniques hybrides compteurs/marquage (cf II.2); tout compteur ayant atteint la valeur maximale reste constant et l'objet correspondant ne peut être récupéré que par un algorithme de marquage. En particulier [WIS 77], s'appuyant sur les travaux de [DEU 76], suggère l'emploi d'un seul bit pour chaque compteur : les objets libres ou référencés une seule fois ont un compteur de valeur 0, les autres objets ont un compteur de valeur (maximale) 1.

En ce qui concerne la récupération des structures circulaires, les solutions présentées dans la littérature sont axées sur des applications développées en LISP [FRI 79], [BOB 80], [SPE 82] mais ne sont pas généralisables sans l'emploi d'une méthode compteurs/marquage.

Signalons, enfin, le travail réalisé par [CHR 84] basé sur une technique hybride et destiné à permettre la manipulation de variables dynamiques en FORTRAN; malheureusement le ramasse-miettes proposé est d'un coût prohibitif (quatre balayages du tas).

II.2 Techniques par marquage

Les algorithmes basés sur cette technique réalisent les deux opérations suivantes :

- la construction d'une arborescence $\Gamma^*(R_A)$ recouvrant tous les objets accessibles (i.e. appartenant à G_A) afin d'en déduire le graphe des objets inaccessibles :

  $$G_I \Leftarrow G - G_L - \Gamma^*(R_A)$$

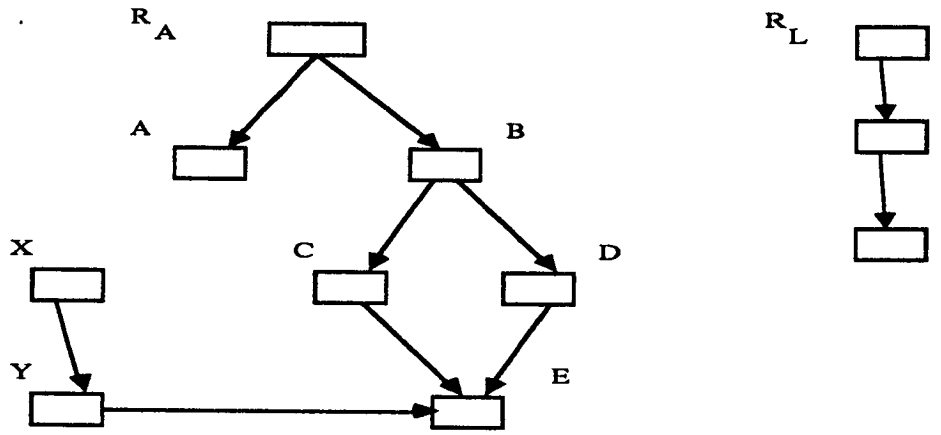
- la mise à jour du graphe G_L des objets libres : $G_L \Leftarrow G_L \cup G_I$

La construction de l'arborescence, ou phase de marquage, peut être réalisée par un parcours en largeur (utilisation d'une file) ou en profondeur (utilisation d'une pile).
La mise à jour de G_L est appelée :

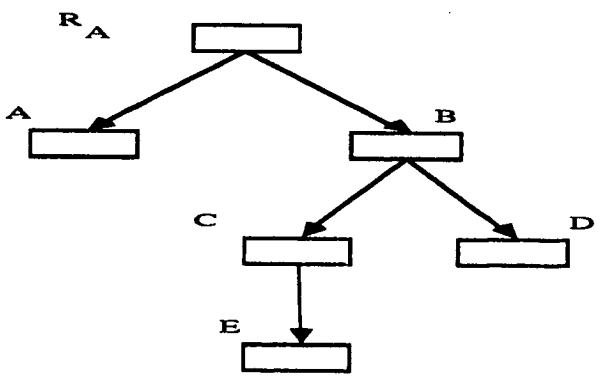
- phase de balayage lorsqu'elle est exécutée à la fin de la phase de marquage,
- phase de recopie lorsqu'elle est exécutée au cours de la phase de marquage.

La réalisation des phases marquage/balayage ou marquage/recopie est confiée à un processus appelé *collecteur*.

Ainsi pour le graphe G suivant :



la phase de marquage établit $\Gamma^*(R_A)$:



avec $G_I = \{ X, Y \}$
et la phase de balayage (ou de recopie) met à jour G_L :

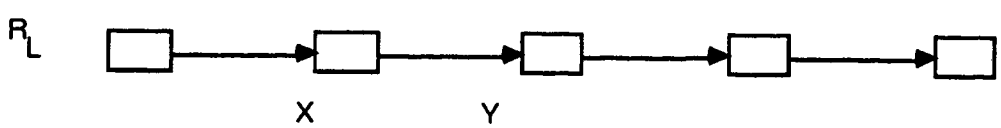


Figure II.3

L'existence des deux processus mutateur et collecteur nous amène à considérer les deux approches suivantes :

- le mutateur suspend son activité pendant l'exécution du collecteur,
- le mutateur et le collecteur s'exécutent de manière simultanée.

II.2.1 Inactivité du mutateur pendant l'exécution du collecteur

Le mutateur, lorsqu'il ne dispose plus d'un graphe des objets libres suffisant, active le collecteur et reste suspendu durant l'exécution de ce dernier. Ainsi, au cours de la phase de marquage, les graphes G_A, G_I et G_L sont statiques et il est possible de construire exactement l' arborescence $\Gamma^*(R_A)$ de G_A.

II.2.1.1 Phase de marquage

On associe à chaque objet x un champ *couleur* tel que, au cours du marquage, *x.couleur* est à *blanc* lorsque l'objet x n'a pas été visité et *x.couleur* est à *noir* lorsque x a été visité.

Une mise en œuvre, par le champ *couleur*, du calcul de $\Gamma^*(R_A) \cup \Gamma^*(R_L)$ permet de distinguer, en fin de marquage, les objets inaccessibles des objets accessibles ou libres.

Algorithme de marquage - calcul de $\Gamma^*(R_A) \cup \Gamma^*(R_L)$ - par un parcours en largeur

```
{f est la file des objets à marquer }
{ pour tout objet x, x.couleur = blanc }
mettre_en_file(f, R_A);  R_A.couleur := noir;
mettre_en_file(f, R_L);  R_L.couleur := noir;
tantque non file_vide(f)
faire
   oter_de_file(f, x);
   pourtout y successeur de x
   faire
        si y.couleur = blanc alors mettre_en_file(f, y );  y.couleur := noir fsi
   fait;
fait
{pour tout objet x : x.couleur = blanc ⇔ x inaccessible,
                     x.couleur = noir  ⇔ x accessible ou libre }
```

II.2.1.2 Phase de balayage

A l'issue de la phase de marquage, on parcourt linéairement tous les objets de la mémoire afin de mettre à jour G_L et de réinitialiser à blanc la couleur de chaque objet.

pourtout objet *x* de la mémoire
faire
    si *x.couleur* = *blanc* **alors** $G_L \Leftarrow G_L \cup \{x\}$
        sinon *x.couleur* := *blanc*
    fsi
fait
{ pour tout objet *x* , *x.couleur* = *blanc* }

II.2.1.3 Phase de recopie

Afin d'éviter la phase de balayage, l'algorithme proposé dans [FEN 69] partage la mémoire en deux demi-espaces communément appelés "fromspace" et "tospace", destinés à être commutés.

Pendant l'exécution du mutateur, le "fromspace" contient les objets créés avant la dernière commutation et le "tospace" les objets alloués depuis la dernière commutation.
Lors de son exécution, le collecteur calcule $\Gamma^*(R_A)$ en recopiant tous les objets accessibles dans le "tospace"; en fin de marquage, le "fromspace" devient ainsi disponible. Il suffit alors d'inverser les rôles des deux demi-espaces et le mutateur peut reprendre son exécution.

L'approche présentée ici permet de récupérer tous les objets inaccessibles sans demander de travail supplémentaire au mutateur.Les inconvénients résident dans :

- l'occupation mémoire nécessaire à l'algorithme de marquage, on trouvera dans [COH 81] de nombreuses références sur des algorithmes permettant de réduire, voire supprimer, la place mémoire utile au marquage,
- l'inactivité imposée au mutateur lors de l'exécution du collecteur.

II.2.2 Exécution simultanée du mutateur et du collecteur

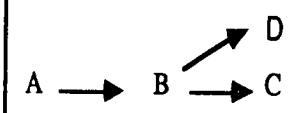
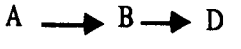
Les deux processus, mutateur et collecteur, s'exécutent de manière simultanée avec des contraintes d'exclusion et de synchronisation.

Au cours de la phase de marquage, les graphes G_A, G_L et G_I sont dynamiques du fait des opérations de retrait ou d'ajout d'arcs effectuées par le mutateur :

- sur un retrait d'arc, G_A peut diminuer et G_I augmenter,
- sur un ajout d'arc, G_A peut augmenter et G_L diminuer.

II.2.2.1 Suppression d'arcs lors du marquage

Considérons la situation suivante où $t_0 < t_1 < t_2 < t_3$:

	Mutateur	Collecteur	Graphe G_A
t_0			
t_1		marquage A B C	
t_2			
t_3		marquage D fin de marquage	

- à l'instant t_1, le collecteur a marqué les objets A, B, C (à noir),
- à l'instant t_2, le mutateur retire l'arc (B, C) du graphe G_A,
- à l'instant t_3, l'objet C, devenu inaccessible, est considéré accessible par le collecteur.

Ainsi, le collecteur récupère un graphe $G'_I \subset G_I$ et les objets inaccessibles de $G_I - G'_I$ seront pris en compte lors de la prochaine activation du ramasse-miettes.

II.2.2.2 Ajout d'arcs lors du marquage

Soit la situation suivante où $t_0 < t_1 < t_2 < t_3 < t_4$:

	Mutateur	Collecteur	Graphe G_A
t_0			A → B → C
t_1		marquage de A B en attente de marquage	
t_2	* A → C * B ✗ → C		
t_3		marquage de B	A → B C
t_4		fin de marquage	A → B C

- à l'instant t_1, le collecteur marque l'objet A et place B dans la file d'attente f,
- à l'instant t_2, le mutateur ajoute l'arc (A, C) puis retire l'arc (B, C),
- à l'instant t_3, le collecteur marque B,
- à l'instant t_4, l'objet accessible C est considéré inaccessible par le collecteur.

Pour éviter la récupération à "tort" d'objets accessibles, le mutateur doit participer au marquage lors de l'ajout d'arcs. Ainsi, lors de l'ajout de l'arc (A, C), si le mutateur réalise l'opération atomique (indivisible et notée < >) :

< ajout(A, C); **si** C.couleur = blanc **alors** mettre_en_file(f, C); C.couleur := noir **fsi** >

l'objet C ne sera pas récupéré par le collecteur.

Remarques

- Pour la phase de marquage [DIJ 78] introduit trois couleurs :
  blanc, noir et gris; les objets x de couleur gris correspondent ici aux éléments de la file f , c'est-à-dire aux objets considérés accessibles ou libres non encore étudiés.
- On obtient une parallélisation "grossière" en conservant les algorithmes de marquage et balayage donnés en II.2.1 et en supposant les opérations atomiques :
  * pour le marquage :
    • les primitives de file <mettre_en_file(f, X); X.couleur := noir >,
    < file_vide(f) >,
    <oter_de_file(f, X) >
    • <**pourtout** y successeur de x **faire** ... **fait** >
  * pour le balayage : $< G_L \Leftarrow G_L \cup \{ x \} >$
- [BAK 78] a adapté, au temps réel, l'algorithme de marquage/recopie de [FEN 69].

II.2.2.3 Autres travaux

- On trouvera dans [COH 81] plusieurs références concernant des preuves pour les algorithmes de marquage; plus récemment [BEN 84] puis [VAN 87] ont fourni des variantes de l'algorithme de [DIJ 78] permettant d'en "simplifier" la preuve.
- [LIE 83] a adapté l'algorithme de [BAK 78] pour un ramasse-miettes basé sur la durée de vie des objets, le collecteur privilégiant la récupération des objets les plus récents.
- [RAM 85] propose un algorithme parallèle de collection dans une mémoire virtuelle. Afin de diminuer les contraintes de synchronisation entre mutateur et collecteur, l'architecture considérée est basée sur trois processeurs (mutateur, collecteur et gestion de l'espace virtuel). [ABR 87] reprend la solution de[RAM 85] en évitant les contraintes d'exclusion (opérations atomiques) entre le mutateur et le collecteur. Pour cela, le tas G est sauvegardé dans G' avant le début de la phase de marquage, puis le mutateur continue à travailler sur G pendant que le collecteur effectue le marquage du graphe statique G'. La phase de balayage permet de récupérer les objets inaccessibles ($G'_I \subset G_I$).
- Pour une architecture multiprocesseurs avec mémoire commune [SHI 85] propose un algorithme de ramasse-miettes s'appuyant sur [DIJ 78] tandis que [HAL 84] et [ELL 88] généralisent l'algorithme de [BAK 78].
- Les premiers algorithmes de ramasse-miettes ont été conçus pour le langage LISP et [HIC 84] en a étudié les performances. Actuellement, des algorithmes sont proposés et réalisés pour le langage PROLOG [AHS 87], [BEK 86], [PIT 85] et une étude quantative est effectuée dans [RID 87].

III Partage d'objets dans un système distribué

On considère un système distribué composé de plusieurs sites ayant des identités distinctes; sur chacun des sites résident deux processus particuliers appelés *mutateur* et *collecteur*. Le premier est chargé de l'exécution d'un programme utilisateur alors que le second est chargé de la récupération des objets rendus inaccessibles par les opérations du programme utilisateur. Les sites communiquent entre eux par échanges de messages au travers d'un réseau de communications que nous supposons fiable, c'est-à-dire que, sauf indications contraires, les hypothèses suivantes sont vérifiées :

- il n'y a ni perte ni altération de messages,
- les communications sont asynchrones,
- le déséquencement de messages est possible,
- le délai de transmission d'un message est quelconque mais fini.

Chaque programme utilisateur dispose d'un espace d'adressage virtuel lui permettant d'accéder à des objets sur son site de résidence et à des objets sur d'autres sites. Pour ces derniers, les programmes usagers peuvent se communiquer (par messages) des références ou des valeurs d'objets.

Exemple avec 3 sites

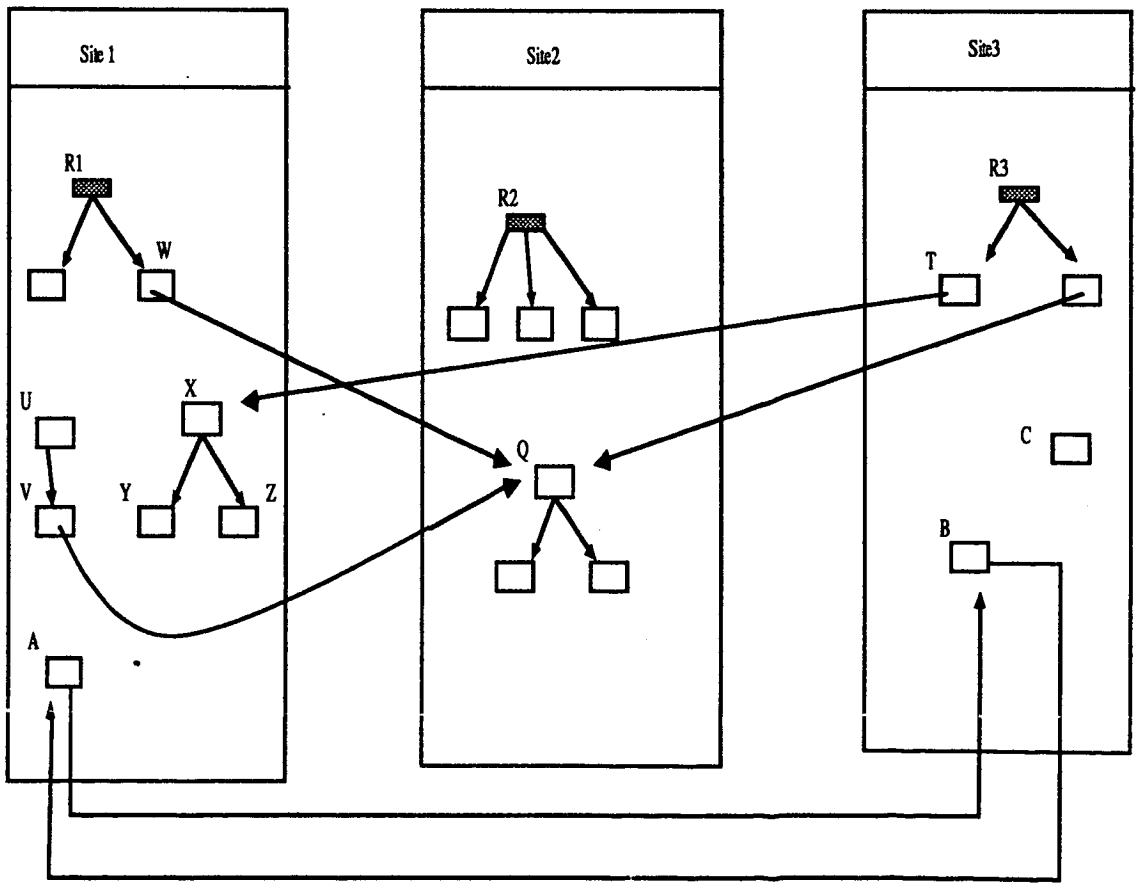


Figure III.1

On note G le graphe, distribué sur l'ensemble des sites, de tous les objets (accessibles, libres ou inaccessibles) du système. Un objet quelconque d'un site i est accessible lorsqu'il appartient à $\bigcup_{\text{site j}} \Gamma^*(R^j_A)$ et libre lorsqu'il appartient à $\Gamma^*(R^i_L)$.

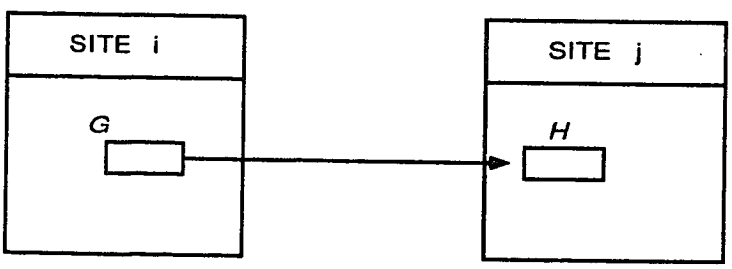
A un instant donné, un objet accessible est :

- *privé* s'il n'est référencé par aucun mutateur distant,
- ou *partageable* s'il est référencé directement par un sous-ensemble non vide de mutateurs distants et éventuellement par le mutateur local. Pour l'exemple de la figure III.1, les objets W, U, V, X, Y, Z et A du site 1 vérifient :
- W, Y et Z sont accessibles et privés,
- U, V et A sont inaccessibles,
- X est accessible et partageable.

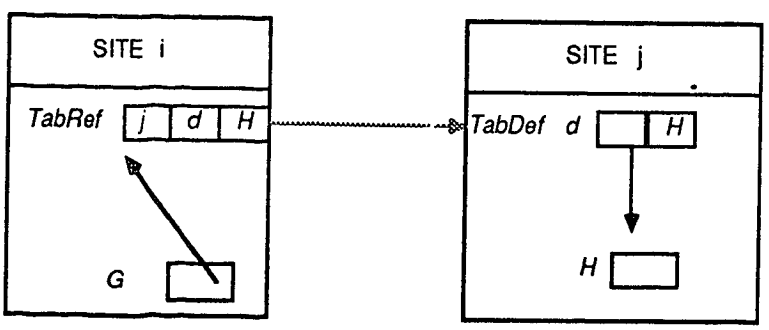
III.1 Les tables d'adressage

Les techniques proposées pour la mise en œuvre des objets partageables utilisent généralement :

- d'une part, deux tables pour chaque site i :
  - *TabDef*$_i$: contient l'ensemble des objets partageables situés sur le site i,
  - *TabRef*$_i$: contient l'ensemble des objets référencés par le mutateur du site i et disponibles sur d'autres sites,
- d'autre part, un mécanisme de "triple indirection" entre la référence à un objet et l'objet lui-même. Ainsi, la référence G--H entre les sites i et j :



est mise en œuvre par



Le couple *(j, d)* , où *j* est le numéro du site propriétaire de l'objet *H* et *d* l'entrée associée à *H* dans $TabDef_j$, est appelé *pointeur distant.*

Ce mécanisme de triple indirection donne la possibilité au site propriétaire de reloger un objet partageable (en cas de retassement de la mémoire par exemple) sans avertir les sites référençant cet objet.

La manière de désigner un objet (par son nom, l'entrée dans un catalogue, ...) étant étroitement liée à l'application, nous utiliserons par la suite une lettre pour "nommer" un objet (ici *H*).

Avec l'exemple de la figure III.1, on obtient la mise en œuvre suivante :

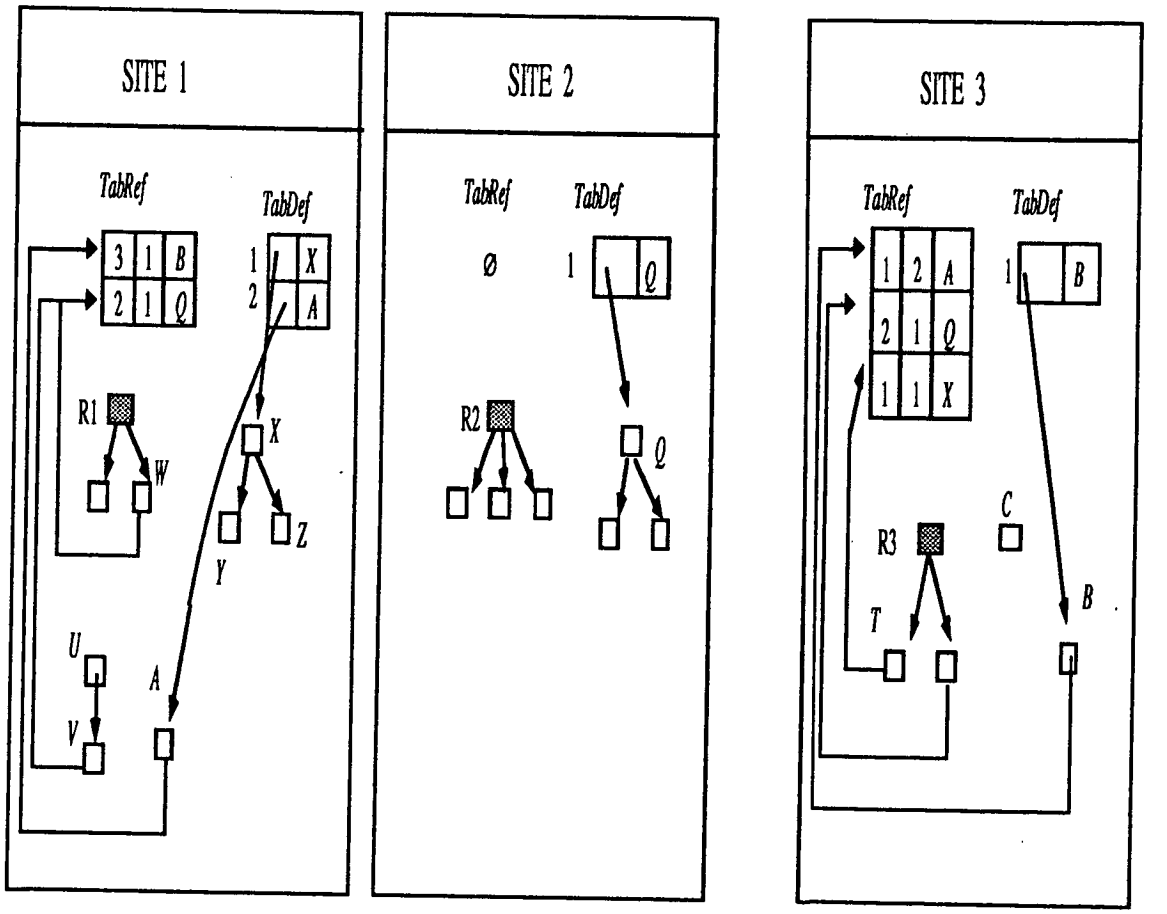


Figure III.2

III.2 Les opérations sur les objets

Il convient d'ajouter, dans un système distribué, aux opérations locales du mutateur (cf I), des primitives de communications.

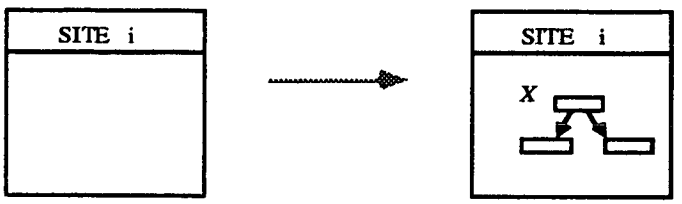
III.2.1 Création de références distantes

A partir des choix de [ALI 85], [HUD 82], [HUG 85]et [RUD 86], on définit trois catégories de primitives pour la création de références distantes.

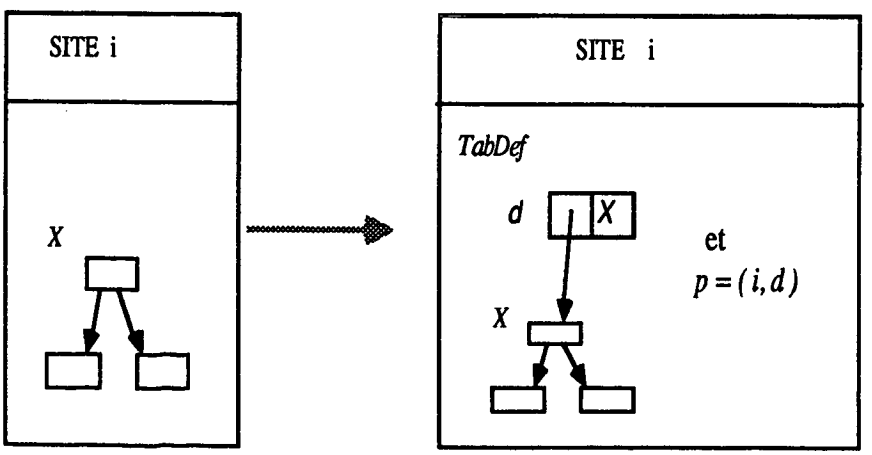
- **Création de références par diffusion**

Chaque mutateur i peut diffuser des références aux autres mutateurs, pour cela, il dispose des fonctions :

* *Implanter (X)* où *X* est un objet local au site i



* *CréerDef(X, P)* où *X* est un objet local au site i.
Permet de créer localement un pointeur distant *P* en vue de son transfert vers un autre site.



* *DiffuserRef(X, P)*
Permet au site i d'envoyer vers tout ou partie des autres sites la désignation de l'objet *X* et le pointeur distant associé *P*.

* **Lors de la réception de** *DiffuserRef(X, P)* **de** site k
faire
si $X \notin TabRef_i$ **alors** création d'une entrée *(P, X)* dans $TabRef_i$ **fsi**
fait

- **Création de références par demande au site propriétaire**

Chaque mutateur *i* peut demander des références aux autres mutateurs, il utilise alors les primitives :

* *LierRef(X, O, site j)* { *O* est un objet du site i, }
{ *X* est la désignation d'un objet du site j }

cas $X \in TabRef_i$ {entrée r} ⇒ *O* repère r
$X \notin TabRef_i$ ⇒ **envoyer** *DemandeRef(X, O)* **à** site j
fcas

* **Lors de la réception de** *DemandeRef(X, O)* **de** site k
faire
cas $X \notin site_i$ ⇒ **envoyer** *RefusRef(X, O)* **à** site k
$X \in site_i$ ⇒ **si** $X \notin TabDef_i$ **alors**
si *X* non implanté **alors** *Implanter (X)* **fsi;**
CréerDef(X, P)
sinon { entrée d}
P:= (i, d)
fsi;
envoyer *RetourRef(X, O, P)* **à** site k
fcas
fait

* **Lors de la réception de** *RetourRef(X, O, P)* **de** *site j*
faire
création de *(P, X)* dans $TabRef_i$ {entrée r }
O repère *r*
fait

- **Création de références par demande à un site non propriétaire**

Chaque mutateur *i* peut demander une référence à un site non propriétaire de l'objet mais disposant de sa référence. La primitive *DemandeRef* est alors à remplacer par *PasserRef(X, O)* et

* **Lors de la réception de** *PasserRef(X, O)* **de** site k
faire
cas $X \in TabRef_i$ {entrée r (P, X)} ⇒ **envoyer** *RetourRef(X, O, P)* **à** site k
$X \notin TabRef_i$ ⇒ suivant la stratégie du mutateur : message de refus ou diffusion de la demande vers un autre site
fcas
fait

Ainsi dans la figure III.2, le mutateur du site 3 a pu demander au mutateur du site 1 de lui donner la référence à *Q*.

III.2.2 Accès aux références distantes

Lorsque les références distantes sont établies, un mutateur a la possibilité d'accéder aux objets disponibles sur les autres sites. L'accès à une référence peut concerner :

- l'envoi de données ou de code du site i vers l'objet distant (i.e. l'écriture à distance),
- la recopie, sur le site i, du contenu de l'objet distant (i.e. la lecture à distance),
- l'exécution du code contenu dans l'objet distant (i.e. l'exécution à distance).

Chaque mutateur i dispose des primitives suivantes :

* **envoyer** *AccèsRef(d, ...)* **à** site j
{ l'entrée de $TabRef_i$ concernée par la référence contient *(j, d)* }

* **Lors de la réception de** *AccèsRef(d,...)* **de** site k
faire
{$TabDef_i[d]$ est définie }
....
fait

IV Ramasse-miettes de type compteurs de références

IV.1 Solution dérivée du centralisé

Comme nous l'avons vu en II.1, le travail du collecteur est, en grande partie, effectué par le mutateur; pour adapter la technique par compteurs de références, dans un contexte distribué autorisant le partage d'objets entre sites, il est nécessaire d'aménager les opérations du mutateur décrites en III.

Exemple : supposons existants les objets A et B respectivement sur le site 1 et sur le site 2, on souhaite établir une liaison de A vers B; l'opération *LierRef(B, A, 2)* est alors exécutée par le mutateur du site 1 qui envoie le message *DemandeRef(B, A)* au site 2 :

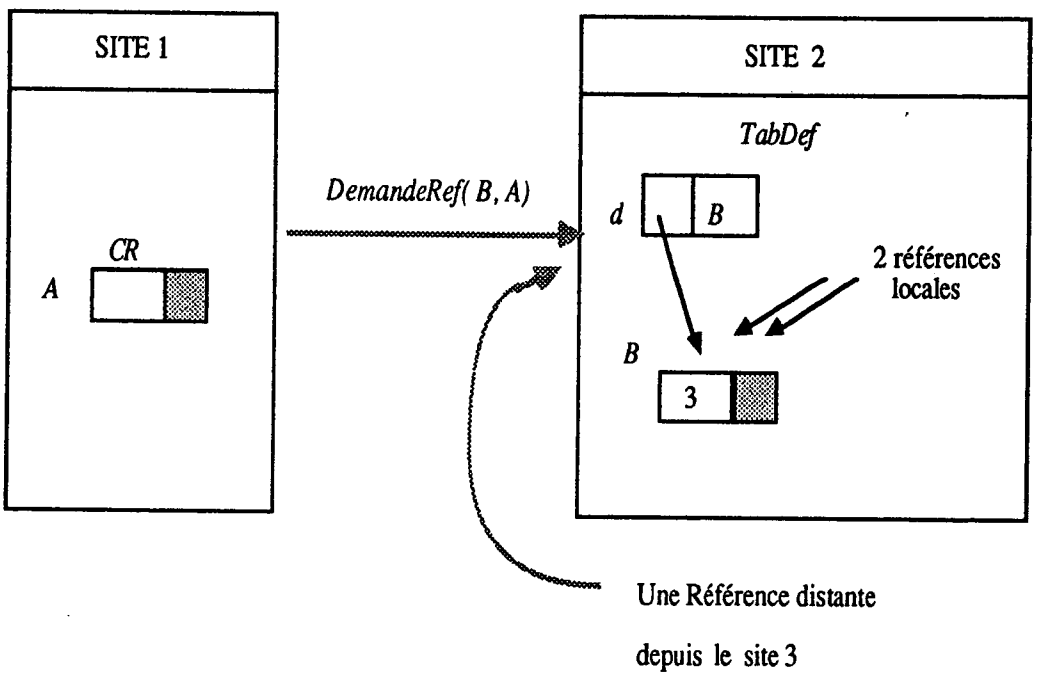


Figure IV.1

Lors de la réception de *DemandeRef(B, A)* **de** site 1,
le site 2 effectue :

. *CR(B) := CR(B) + 1*;

. **envoyer** *RetourRef(B, A, P)* **à** site 1, avec *P = (2, d)*

Après la réception de *RetourRef(B, A, P)* par le site 1, on a :

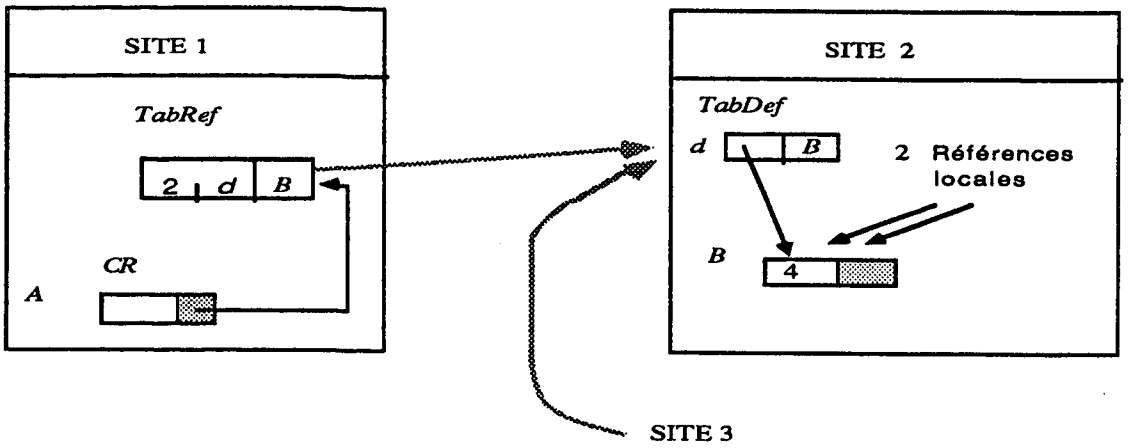


Figure IV.2

Afin de maintenir à jour *CR(B)*, l'opération *LierRef* doit transmettre le message *DemandeRef* même si le site demandeur possède déjà la référence souhaitée.

Il faut, enfin, prévoir des opérations correspondant à la suppression de références distantes : la destruction de la liaison *A--B* provoque l'envoi, au site 2, d'un message indiquant qu'il faut décrémenter *CR(B)* de 1.

Cette technique est, de toute évidence, très coûteuse puisque les affectations locales (*LierRef* avec référence déjà dans *TabRef)* nécessitent un envoi de message au site propriétaire.

Son principal inconvénient réside dans son inadéquation à résoudre le problème suivant :

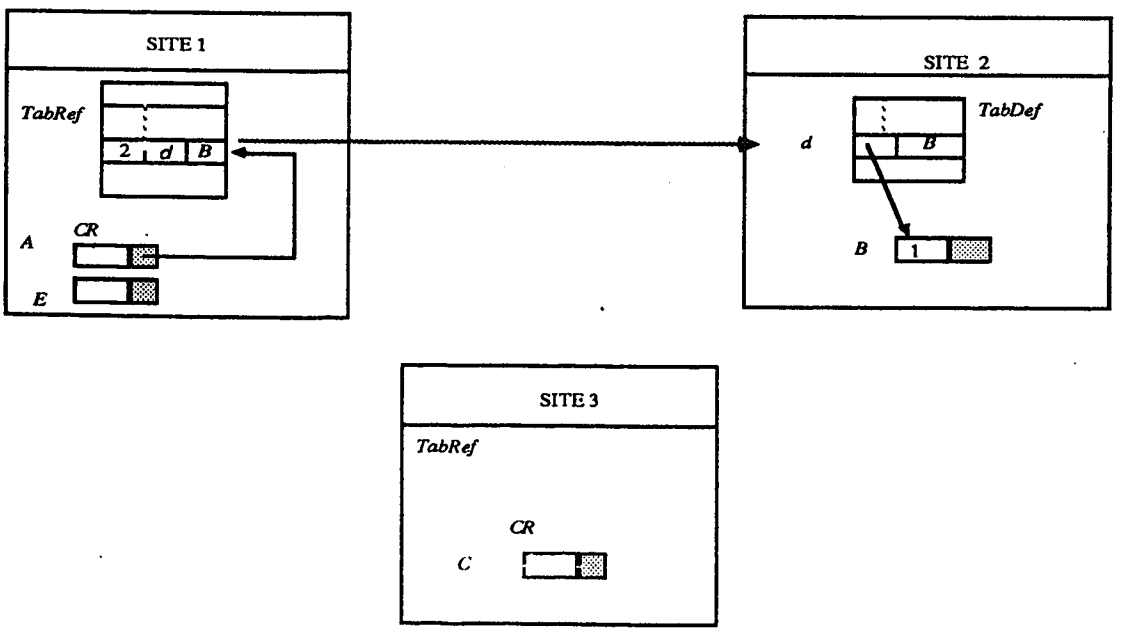


Figure IV.3

a) demande d'une référence déjà présente sur le site

Si le mutateur du site 1 exécute consécutivement :

- *LierRef(B, E, 2)* : envoi d'un message pour incrémenter le compteur de références de *B*,
- Suppression de la liaison *A--B* : envoi d'un message pour décrémenter le compteur de *B*

alors un déséquencement des deux messages provoque une libération prématurée de *B*.

b) passage de références entre sites

Si on a les opérations consécutives suivantes:

C3 : **envoyer** *PasserRef(B, C)* **à** site 1 par le site 3

C1 : **envoyer** *RetourRef(B, C, (2, d))* **à** site 3 par le site 1

puis les opérations "parallèles" :

P1 : suppression de la liaison *A --B* par le site 1
⇒ **envoyer** *%Décrémenter le compteur de d %* **à** site 2

P3 : **Lors de la réception de** *RetourRef(B,C, (2, d))* **de** site 1 par le site 3
faire
envoyer *%Incrémenter le compteur de d %* **à** site 2
fait

Tout se passe bien si le message émis par P3 parvient au site 2 avant le message émis par P1, sinon l'objet *B* est libéré prématurément par le site 2 propriétaire.

Pour assurer un ordre total, et donc l'exécution correcte d'une telle séquence d'opérations, on peut envisager diverses stratégies, par exemple celle que propose [LER 86] : elle impose des liaisons FIFO entre sites; le site 1, auquel la référence à *B* est demandée, avertit le site 2 propriétaire de l'établissement de la liaison *C--B* puis transmet le pointeur distant au site 3. La contrainte FIFO permet alors au site 1 de détruire, sans danger, la liaison *A--B*.

IV.2 Distribution du compteur

Nous proposons ici une approche mieux adaptée au partage d'objets distants. Pour un tel objet *X* , plutôt que de comptabiliser dans le compteur de références *CR(X)* le nombre total de références à *X* (locales ou distantes), nous ne prenons en considération que les références locales et nous ajoutons 1 si l'objet est partagé (i.e. est présent dans *TabDef)*.

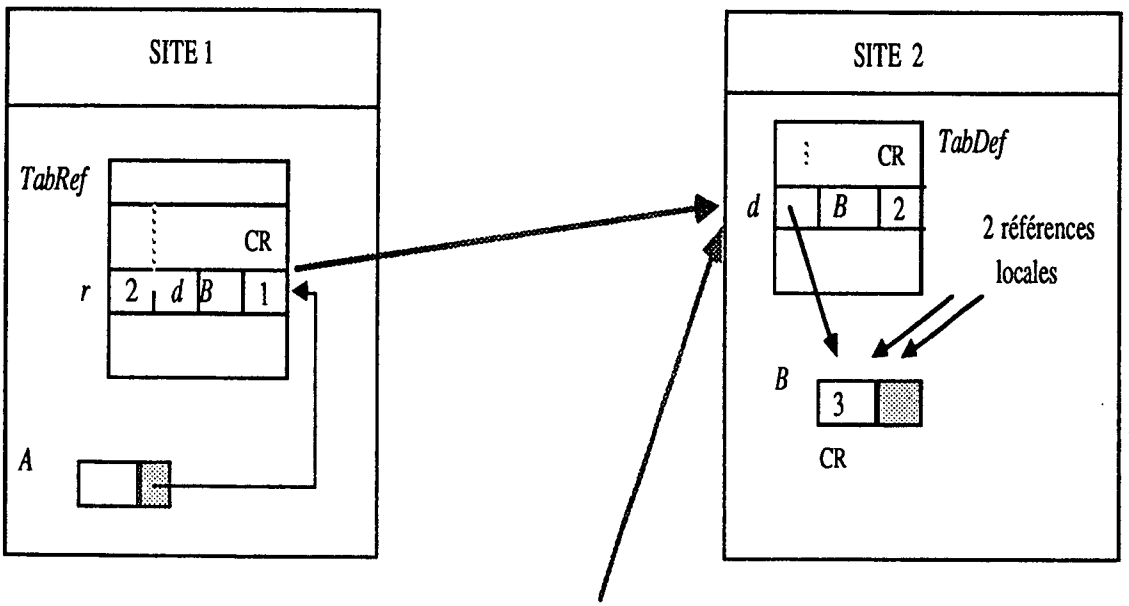
Chaque élément de *TabDef* et de *TabRef* possède un champ compteur de références :

- pour un élément $TabDef_i[d]$, le champ *CR* comptabilise le nombre de sites possédant la référence distante *(i, d)*,
- pour un élément $TabRef_i[r]$, le champ *CR* est le nombre d'objets locaux au site i qui référence l'objet adressé par *r* .

Le nombre de références effectives à un objet *X* du site i vaut :

- *CR(X)* si l'objet est privé
- $CR(X) + TabDef_i[d].CR - 1$ si l'objet est partagé.

Ainsi, pour l'exemple de la Figure IV.2, on obtient :



(2, *d*) accessible aussi par le site 3

$TabRef_1[r].CR$ = 1 car, sur le site 1, seul *A* utilise r dans une référence distante

$TabDef_2[d].CR$ = 2 car les sites 1 et 3 possèdent la référence distante *(2,d)*
CR(B) = 3 = 2 (références locales) + 1 (car *B* est dans *TabDef*)

Figure IV.4

Cette mise en œuvre permet de traiter ***localement*** (i.e. sans envoi de messages) l'établissement de liaisons distantes vers des objets résidant déjà dans *TabRef* : ainsi, sur le site 1, *LierRef(B, E, 2)* nécessite seulement d'augmenter $TabRef_1[r].CR$ de 1. De la même manière, les suppressions de références sont traitées localement tant que $TabRef_1[r].CR$ reste positif. Lorsque $TabRef_1[r].CR$ devient nul, il appartient au site 1 de

déterminer si la référence est devenue inutile ou non; dans l'affirmative, un message est transmis vers le site 2 (possesseur de l'objet d) afin qu'il décrémente $TabDef_2[d].CR$ de 1; si ce compteur passe à 0, *CR(B)* est décrémenté de 1, l'entrée *d* est disponible, B devient privé puis libre lorsque *CR(B)* atteint la valeur 0.

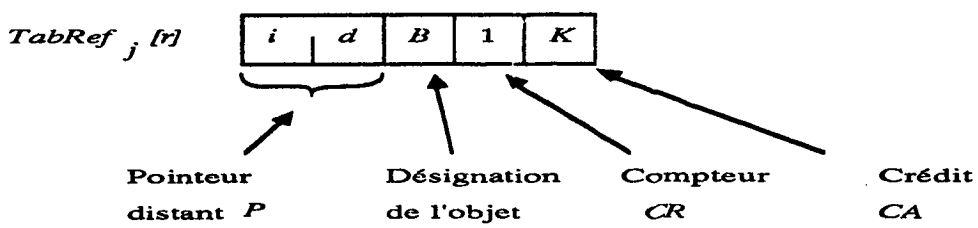
IV.3 Les références pondérées

La démarche précédente nécessite de prévenir le site 2, propriétaire de l'objet *B*, lors du traitement d'une opération *PasserRef* par le site 1 non propriétaire; ceci impose des liaisons FIFO entre sites (cf Figure IV.3, cas b).

Le mécanisme des références pondérées [BEC 86] permet de résoudre ce problème. La notion de référence pondérée, que nous appellerons crédit, représente, d'une certaine manière, le nombre maximum des références vers un objet (et non pas le nombre exact comme pour les compteurs de références). Lorsqu'un site reçoit un message *DemandeRef(B, A)* pour un objet *B* qui lui appartient et qui ne figure pas dans sa table *TabDef*:

- *B* est implanté dans *TabDef[d]* avec un crédit total *CT* (ce champ remplace le champ *CR* dans *TabDef*) que l'on supposera égal à une puissance de 2 (ce choix est justifié à la fin du paragraphe),
- on initialise un nouveau champ *TabDef[d].CA*, qualifié de crédit alloué, avec la valeur *TabDef[d].CT* **div** 2; celui-ci représente la disponibilité du propriétaire pour l'objet partageable, c'est-à-dire le nombre de références potentielles, sur cet objet, attribuables à d'autres sites,
- enfin, on transmet le pointeur distant, accompagné d'un crédit alloué de valeur *TabDef[d].CT* **div** 2.

Lors de la réception de *RetourRef(B, A, P, k)*, le site demandeur *j* crée, dans sa table *TabRef*, une entrée de la forme suivante :



où le champ $TabRef_j[r].CA$ indique la "capacité" du site j à transmettre, vers d'autres sites, le pointeur distant P; le champ $TabRef_j[r].CR$ permet de conserver un traitement banalisé au niveau local (cf IV.2).

Les opérations du mutateur interviennent, bien sûr, sur la gestion du champ *CA* :

(A) Lors de la transmission, par un site k (propriétaire ou non), d'un pointeur distant *P* vers un site *j* ($j \neq k$), le crédit alloué à *P* pour le site *k* est divisé par deux : la moitié reste disponible pour le site *k*, l'autre moitié est accordée au site demandeur *j*. La stratégie de partage inter-sites adoptée maintient pour tout objet *B* partagé : $\Sigma\, CA_B = CT_B$.

Exemple

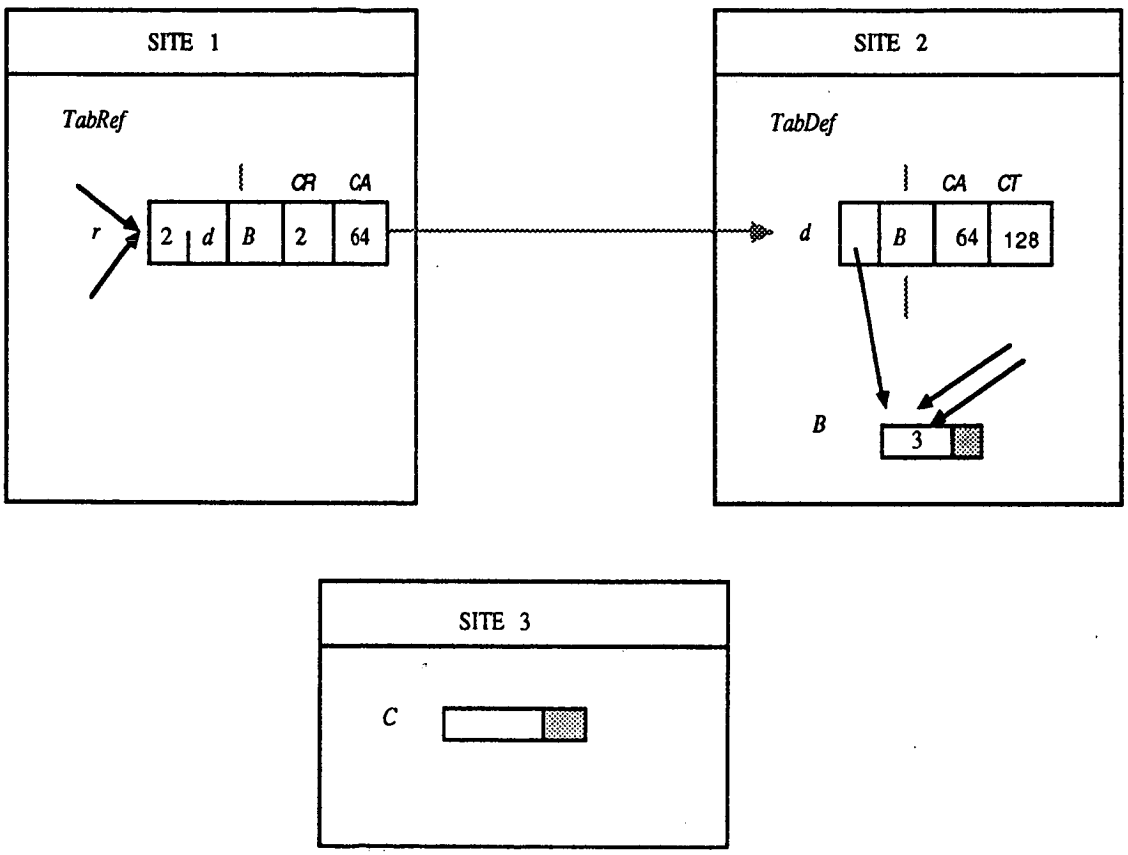


Figure IV.5

On a bien, pour l'objet B: $TabDef_2[d].CT = 128 = TabDef_2[d].CA + TabRef_1[r].CA$.

Avec les traitements suivants sur le site 3 :

envoyer *PasserRef(B, C)* **à** site 1

puis

Lors de la réception de *RetourRef(B, C, (2, d), 32)* **de** site 1

la figure IV.5 devient :

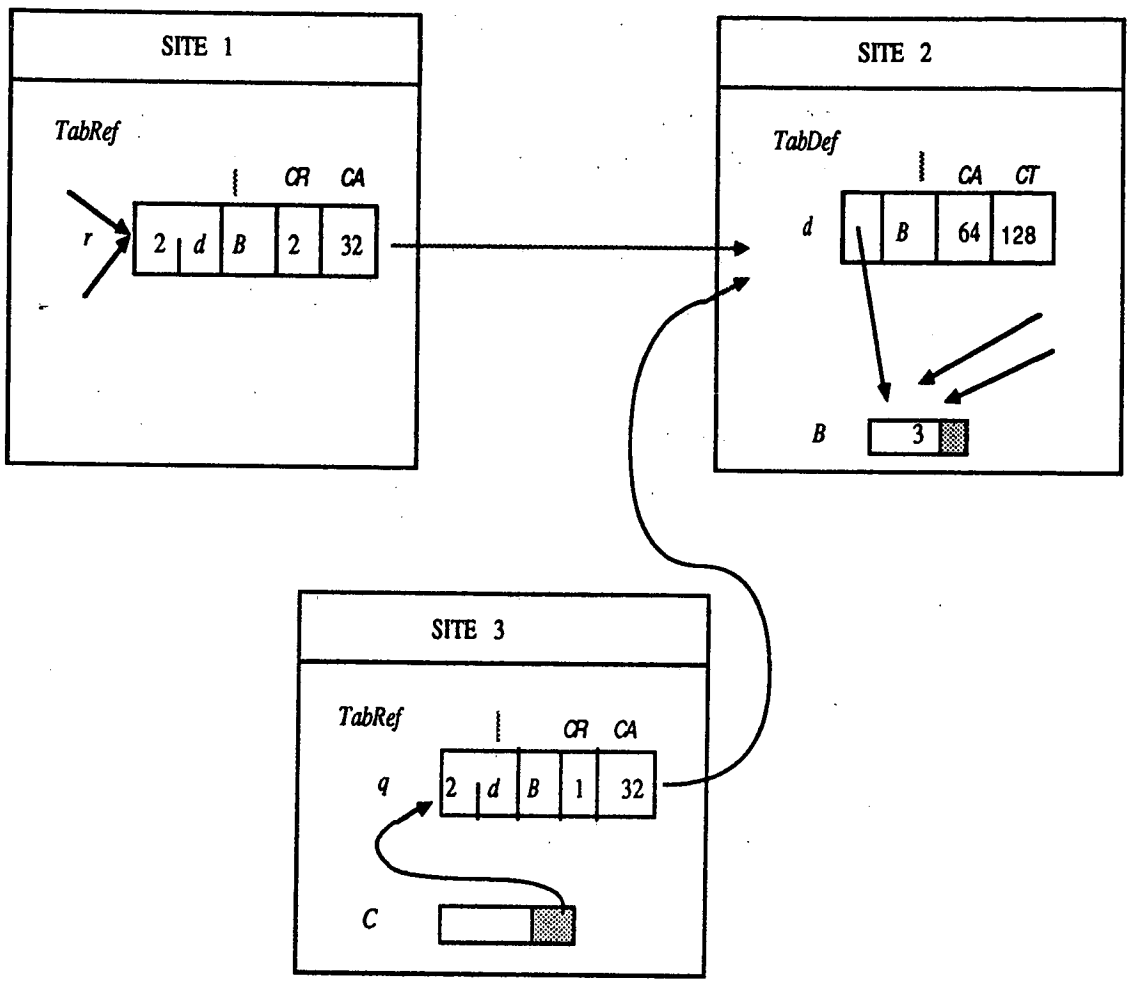


Figure IV.6

On constate que le site 2, propriétaire de l'objet, n'est pas concerné par cet échange de référence.

L'opération s'effectue d'une manière similaire (en divisant $TabDef_2[d].CA$ par 2) si on s'adresse au propriétaire de l'objet.

Il faut, par ailleurs, prévoir une stratégie particulière lorsque le crédit du site récepteur de la demande est insuffisant (i.e. si le champ *CA* vaut 1), ce cas est repris ultérieurement.

(B) Lors de la demande de la suppression, par un site j, d'une référence $TabRef_j[r] = ((i, d), B, CA)$, un message est envoyé, vers le site propriétaire i, pour qu'il décrémente $TabDef_i[d].CT$ de la valeur CA; lorsque $TabDef_i[d].CT = TabDef_i[d].CA$, l'objet B n'est plus utilisé à distance, $CR(B)$ est décrémenté de 1 et B devient libre si $CR(B)$ vaut 0.

Cas particulier d'une demande sur crédit de valeur 1

- Si on s'adresse au site propriétaire i, il suffit d'augmenter $TabDef_i[d].CT$ d'une valeur 2^q et d'attribuer ce crédit au site demandeur.

- Si on s'adresse à un site j non propriétaire de l'objet, on peut :
  - demander au site propriétaire d'augmenter son crédit total d'une valeur 2^q [BEC 86],
  - créer un objet d'indirection [BEV 87] [WAT 87] : ceci consiste à établir un objet particulier dans $TabDef_j$ avec un nouveau crédit total, et à donner une partie de ce crédit au site demandeur (on a alors une quadruple et non plus une triple indirection) :

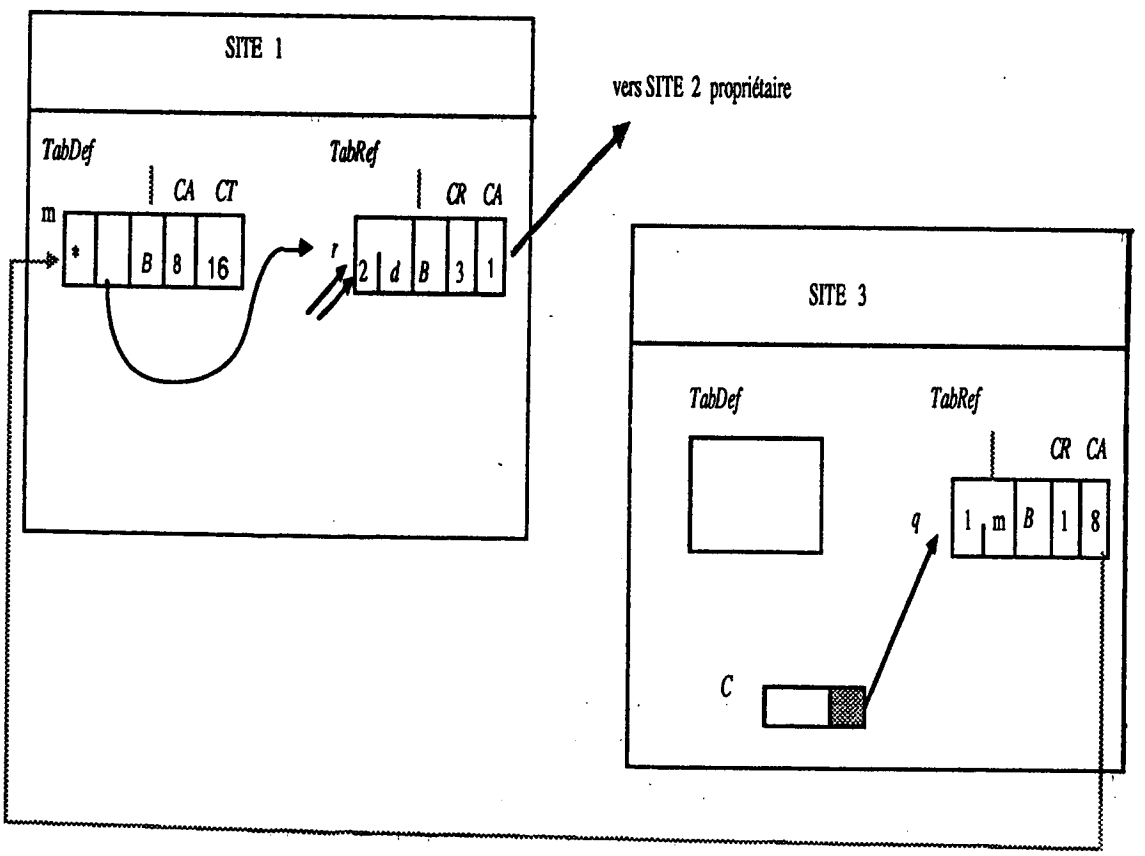


Figure IV.7

L'approche par références pondérées permet de résoudre le problème de l'ordre total (cf IV.1) concernant la création et la suppression de liaisons distantes et, ce, sans imposer des communications FIFO ou des messages d'acquittements.

Exemple

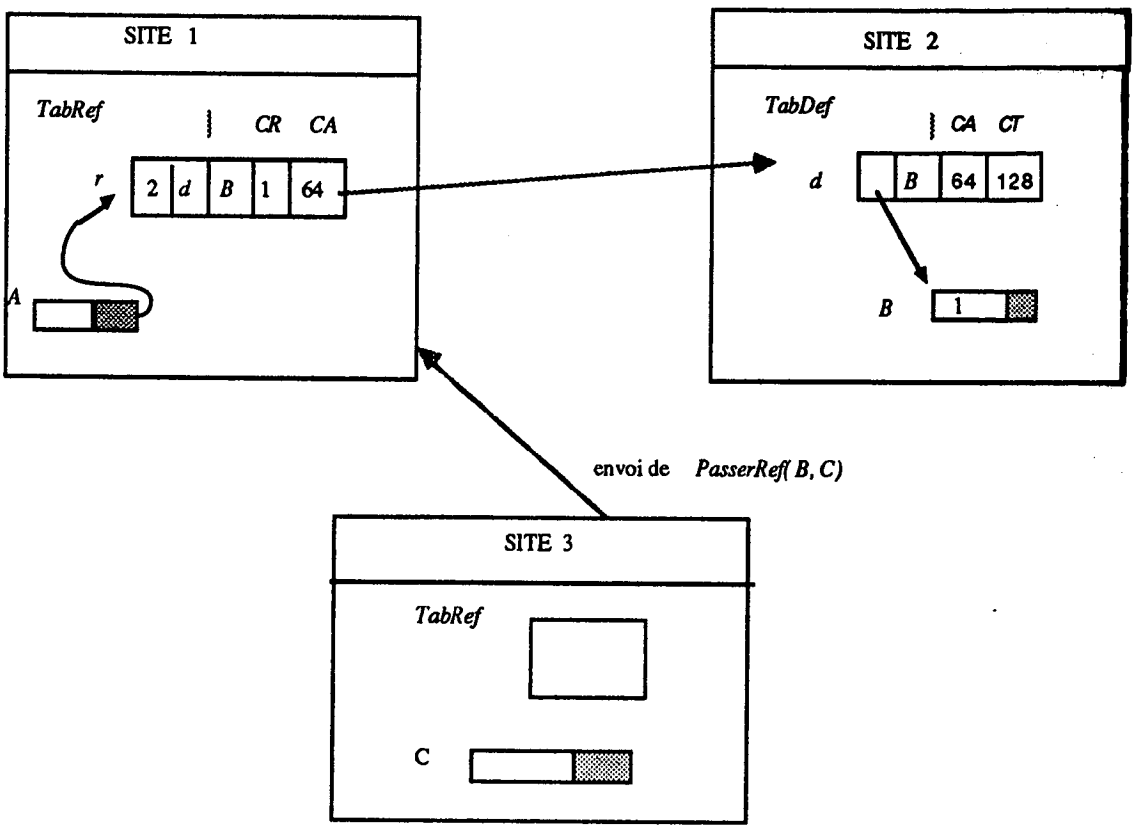


Figure IV.8

Ainsi pour la situation décrite dans la figure IV.8 :
à la réception du message *PasserRef*, le site 1 exécute successivement

- $k := (\ TabRef_1[r].CA\ )$ **div** 2; { $k = 32$}
  $TabRef_1[r].CA := k$;
  envoyer *RetourRef(B, C, (2, d), k)* à site 3;

- Suppression de la liaison *A--B*
  envoyer *Décrémenter(d, $TabRef_1[r].CA$)* à site 2
  (i.e. décrémenter $TabDef_2[d].CT$ de la valeur 32)

Le deuxième message ne rend pas l'objet *B* libre car, lors de sa réception par le site **2 :**
$TabDef_2[d].CA = 64 < TabDef_2[d].CT = 96$

Le choix initial d'un crédit total égal à une puissance de 2 et la transmision de la moitié du crédit alloué ont été proposés simultanément par [BEV 87] et [WAT 87]; de cette manière tout champ *CA* est de la forme 2^m et on ne représente que m (ainsi, si tout crédit total initial est ≤ 128 alors 3 bits suffisent pour représenter les crédits alloués ($0 \leq m \leq 7$)).

IV.4 Conclusion

L'approche par références pondérées est , a priori, séduisante par :

- sa facilité de mise en œuvre,
- son adéquation à toute topologie de réseau (on alloue un crédit élevé aux nœuds "fortement" connectés),
- l'absence de synchronisation,

il n'en reste pas moins qu'elle s'avère incapable de détecter les circuits inaccessibles. Seule une technique mixte références pondérées/ marquage permet de résoudre ce problème. Certains auteurs, [BRO 85], [ECK 87], proposent, pour des extensions du concept classique de compteurs de références, des méthodes détectant les circuits inaccessibles; malheureusement leurs solutions ne sont réalistes que dans un cadre d'applications trop particulier et nécessitent une technique hybride compteurs/marquage dans le cas général; les articles cités, bien que situés par leurs auteurs dans le cadre "d'applications distribuées", ne précisent pas le comportement des algorithmes dans un environnement distribué (protocoles de partage, messages ...). Une autre méthode hybride plus orientée "matériel" est également proposée dans [WIS 85].

Finalement, on peut remarquer que l'approche par références pondérées constitue un bon mécanisme de partage d'objets entre sites, ceci grâce à son adéquation à une topologie quelconque de réseau. Elle peut, ainsi, cohabiter avec des collecteurs locaux de type marquage (cf II.2); le champ *CR* des tables *TabRef* est alors remplacé par un champ "couleur". Les collecteurs locaux travaillent indépendamment les uns des autres et signalent aux sites propriétaires les objets partagés qui ne sont plus accessibles localement. Les structures cycliques inter-sites ne sont pas récupérables; [BEC 86] propose, à cette fin, qu'un processeur particulier P_0 possède une vue du graphe des objets partagés, cette vue est mise à jour à l'issue de chaque phase de collection locale à un site; les arcs du graphe sont de la forme ((i, d), (j,r)) (i.e. du type def-ref) et indiquent les liaisons entre sites. En calculant, périodiquement, la fermeture transitive du graphe qu'il gère, P_0 détecte les structures cycliques d'objets partagés (ce mécanisme est repris dans le paragraphe VI).

V Ramasse-miettes de type marquage global

Le principe de ces algorithmes est de construire une arborescence globale recouvrant tous les objets accessibles. Un processus particulier, appelé *synchroniseur*, joue le rôle de racine virtuelle pour cette arborescence. Ainsi pour l'exemple donné à la figure III.1, une arborescence globale des objets accessibles peut être :

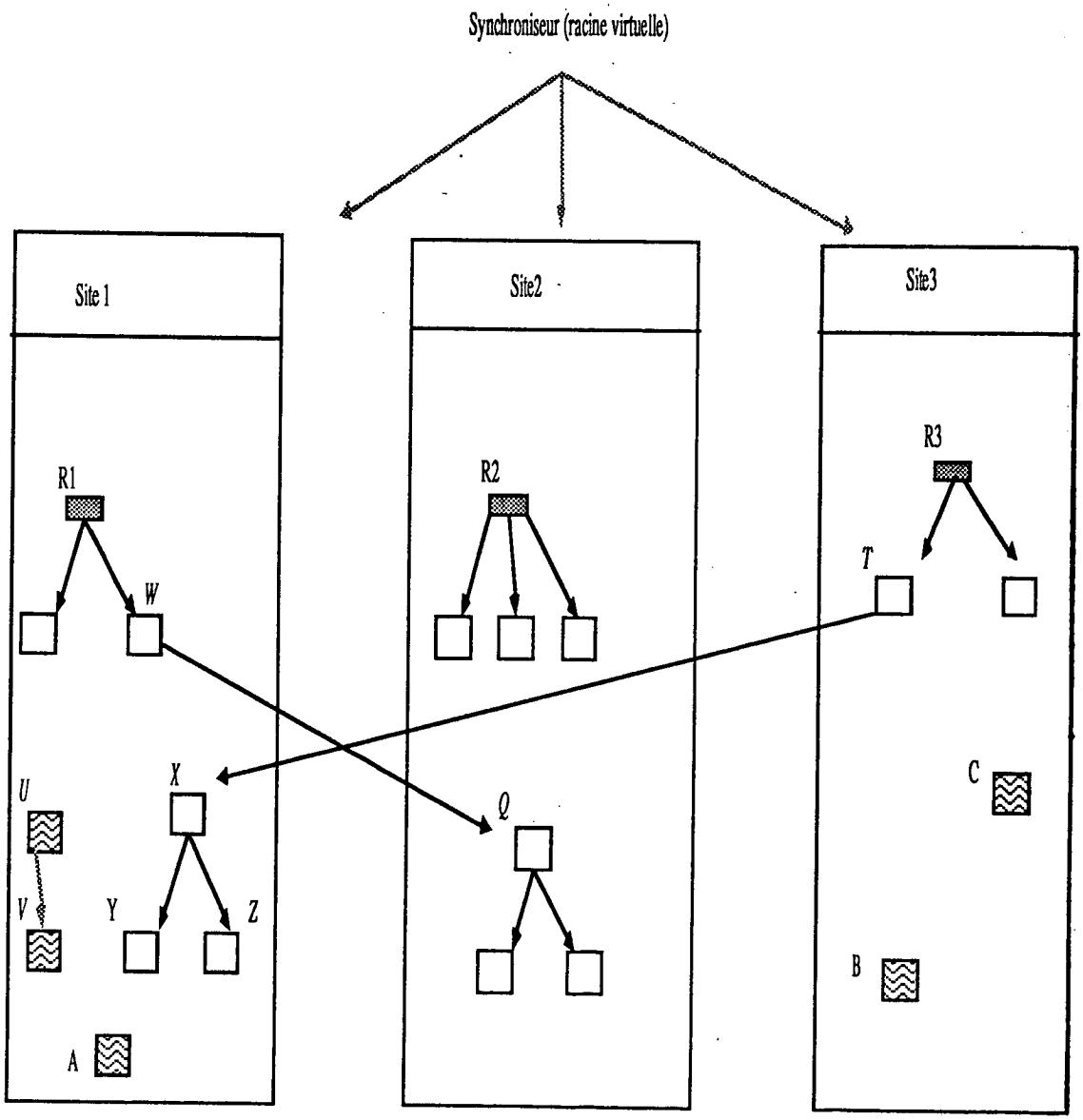


Figure V.1

et les objets A, U, V (respectivement B, C) sont récupérables par le site 1 (respectivement site 3).

Lors d'une demande de ramasse-miettes, la tâche du synchroniseur, pour des collecteurs supposés de type marquage/balayage est de :

1 - lancer les phases de marquage de tous les collecteurs C_i (pour tout site i); en effet, il apparait clairement sur l'exemple que le lancement des seuls collecteurs C_1 et C_2, provoque la récupération (à tort), par le site 1, de l'objet *X* référencé par le site 3,

2 - détecter la terminaison de toutes les phases de marquage des collecteurs C_i (l'arborescence globale est alors construite),

3 - lancer les phases de balayage de tous les collecteurs C_i,

4 - détecter la terminaison des phases de balayage.

Exemple pour 3 sites

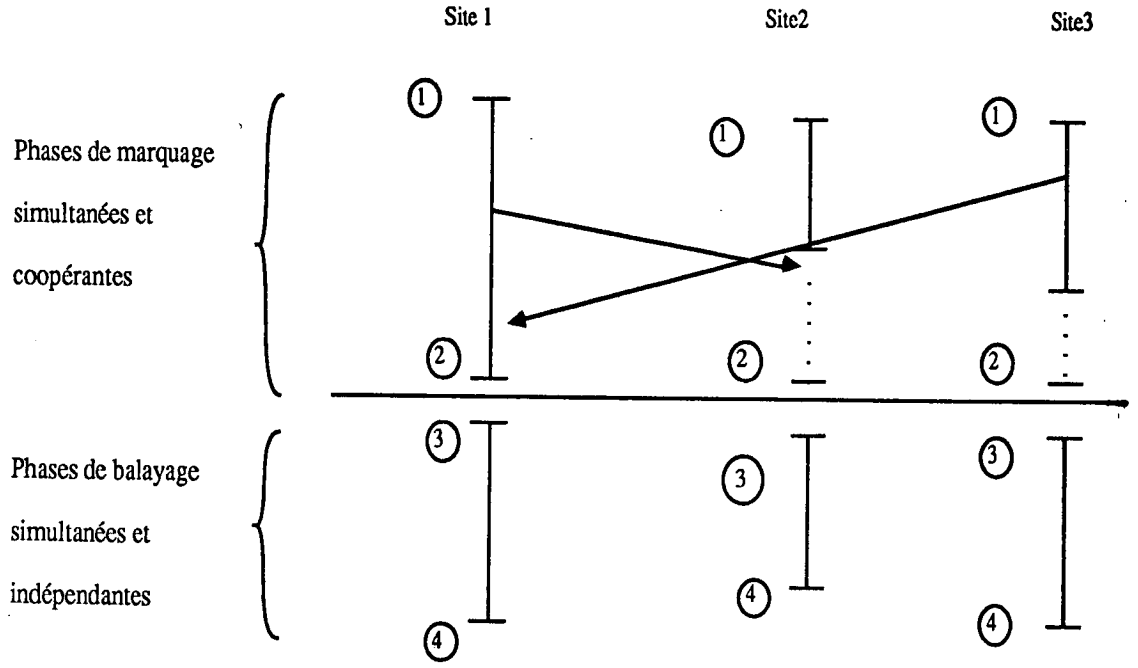


Figure V.2

La phase de marquage de chaque collecteur traite les objets locaux visités (cf II.2) et, pour les objets distants, envoie un message de demande de marquage au site propriétaire.

Afin d'appréhender, de manière simple, le travail de marquage des collecteurs, nous supposons dans un premier temps :

- l'arrêt du mutateur lors de l'activation du collecteur placé sur le même site (ainsi chaque graphe local est statique lors du marquage et le graphe distribué G est statique lorsque tous les collecteurs sont en cours de marquage),
- l'absence de messages en transit entre mutateurs lors du lancement des collecteurs.

Ensuite, nous prendrons en considération les références en transit et la coopération entre les mutateurs et les collecteurs.

V.1 Algorithme de construction de l'arborescence globale

Nous proposons, avec les limitations précédentes, une mise en œuvre du synchroniseur sur un site "particulier" et l'algorithme de collection pour chacun des sites.

V.1.1 Etude du synchroniseur

Le synchroniseur s'exécute sur le site 0, connait le nombre n de collecteurs présents dans le système et procède par échanges de messages avec les différents sites. Il dispose, en outre, de la variable *nbrep* donnant, à tout instant, le nombre de réponses attendues. Les assertions {point i } se réfèrent à la figure V.2.

Lors du lancement d'un ramasse-miettes
faire
nbrep := n ;
∀ i (i = 1 .. n) : **envoyer** *DebMarquage* **à** site i **{ point 1 }**
fait

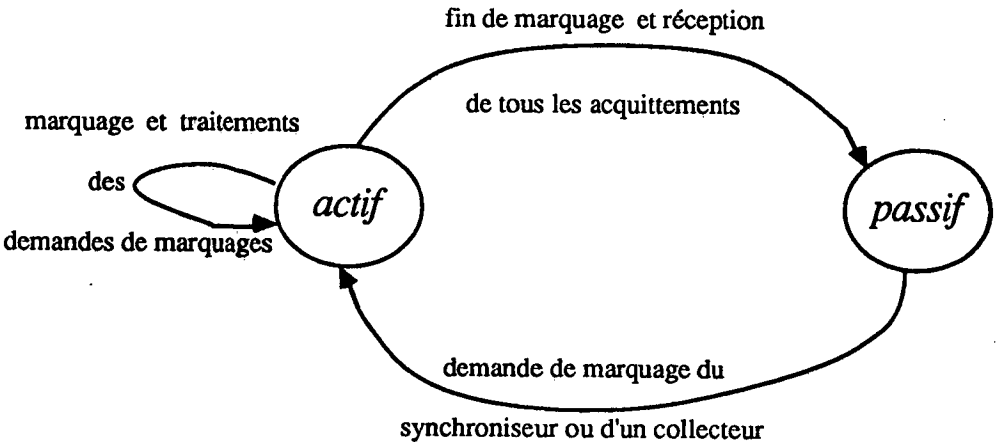
Lors de la réception de *AcqMarquage* **de** site i
faire
nbrep := *nbrep* -1;
si *nbrep* = 0 **alors** **{ point 2 }**
nbrep := n;
∀ i (i = 1 .. n) : **envoyer** *DebBalayage* **à** site i **{point 3}**
fsi
fait

Lors de la réception de *AcqBalayage* **de** site i
faire
nbrep := *nbrep* -1;
si *nbrep* = 0 **alors** { **point 4** } **fsi**
fait

V.1.2 Etude des collecteurs

La phase de marquage de chaque collecteur est dans l'un des deux états :

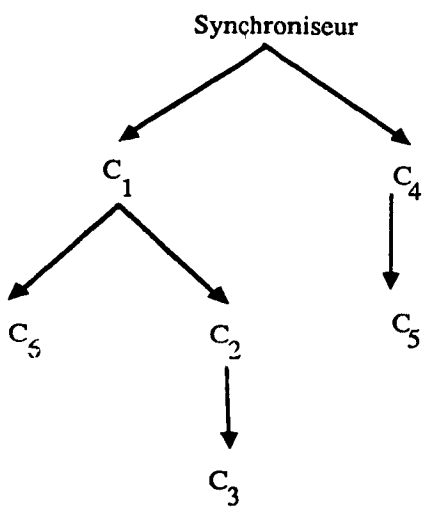
- *actif* lorsque le collecteur est en cours de marquage ou en attente d'acquittements des messages (de demande de marquage) émis vers d'autres collecteurs,
- *passif* lorsque le collecteur a terminé son marquage et a reçu les acquittements de tous les messages émis vers d'autres collecteurs.



Pour assurer la terminaison de toutes les phases de marquage, on construit une arborescence de contrôle distribuée contenant à chaque instant les collecteurs actifs. Dans cette arborescence :

- le synchroniseur est représenté par la racine,
- chaque collecteur actif correspond à un nœud dont le père est le processus l'ayant activé.

Exemple d'arborescence de contrôle



Le synchroniseur a activé C_1 et C_4

C_1 a activé C_2 et C_6

C_4 a activé C_5

C_2 a activé C_3

Lorsqu'il devient passif, un collecteur, représenté par une feuille de l'arborescence, envoie un message d'acquittement à son père puis disparait de l'arborescence. La technique utilisée assure que chaque objet est marqué au plus une fois; il y a un nombre fini d'objets, toutes les phases de marquage se terminent et l'arborescence est alors réduite à la racine.

Algorithme pour le collecteur C_i

Tous les objets du site i, y compris les entrées de $TabDef_i$ et $TabRef_i$ possèdent un champ couleur (cf II.2).

Chaque collecteur C_i dispose des variables suivantes :

- $reçu_i$: indique si C_i a déjà reçu la première demande de marquage, soit du synchroniseur (*DebMarquage*), soit d'un collecteur (*Marquage*) (initialement $reçu_i$ est à *faux*),
- $nbacq_i$: désigne le nombre de messages non acquittés de demande de marquage émis vers d'autres collecteurs (initialement $nbacq_i = 0$),
- $père_i$: permet de repérer le processus ayant activé C_i dans l'arborescence de contrôle,
- f_i : est la file des objets locaux à marquer.

{ initialement les objets ont la couleur blanc }
Lors de la réception de *DebMarquage* **de** site 0 { du synchroniseur }
faire
si non $reçu_i$ **alors** {$état_i$ = passif }
$reçu_i$:= **vrai**;
mettre_en_file (f_i, R^i_A); R^i_A.couleur := noir;
mettre_en_file (f_i, R^i_L); R^i_L.couleur := noir;
$père_i$:= site 0;
{ $état_i$ = actif }
marquage; { la procédure *marquage* peut modifier $nbacq_i$ }
si $nbacq_i = 0$ **alors envoyer** *AcqMarquage* **à** $père_i$ {$état_i$ = passif } **fsi**
sinon envoyer *AcqMarquage* **à** site 0
fsi
fait

Lors de la réception de *Marquage(entrée)* **de** site j
faire
si $TabDef_i[entrée].couleur$ = noir **alors envoyer** *AcqMarquage* **à** site j
sinon
mettre_en_file(f_i, $TabDef_i[entrée]$); $TabDef_i[entrée]$.couleur := noir;
si $nbacq_i = 0$ **alors** { $état_i$ = passif }
si non $reçu_i$ **alors**
$reçu_i$:= **vrai;**
mettre_en_file(f_i, R^i_A); R^i_A.couleur := noir;
mettre_en_file(f_i, R^i_L); R^i_L.couleur := noir
fsi;
$père_i$:= site j;
{ $état_i$ = actif }
marquage; { cet appel peut modifier $nbacq_i$ }
si $nbacq_i$ =0 **alors**
envoyer *AcqMarquage* **à** $père_i$ {$état_i$ = passif}
fsi
sinon { $état_i$ = actif }
marquage;
envoyer *AcqMarquage* **à** site j
fsi
fsi
fait

lors de la réception de *AcqMarquage* **de** site j
faire
{$état_i$ = actif }
$nbacq_i := nbacq_i - 1;$
si $nbacq_i = 0$ **alors envoyer** *AcqMarquage* **à** $père_i$ {$état_i$ = passif} **fsi**
fait

Lors de la réception de *DebBalayage* **de** site 0 {du synchroniseur}
faire
{∀ i (i = 1 .. n), $état_i$ = passif }
balayage; {cf II.2.1 }
envoyer *AcqBalayage* **à** site 0;
fait

où la procédure marquage est définie par :

```
Procédure marquage;
debut
    tantque non file_vide (f_i)
    faire
        oter_de_file(f_i, x);
        pourtout y successeur de x
        faire
            si y.couleur = blanc alors
                cas   y ∉ TabRef_i  ⇒ mettre_en_file(f_i,y) {objet local }
                      y ∈ TabRef_i  ⇒ {pointeur distant }
                                      envoyer Marquage(y.entrée) à y.site;
                                      nbacq_i := nbacq_i +1
                fcas;
                y.couleur := noir
            fsi
        fait
    fait
fin
```

La synchronisation des phases de marquage et de balayage, effectuée dans l'algorithme par le synchroniseur, peut être réalisée par un collecteur quelconque.

Ce principe de construction d'une arborescence de contrôle (appelé aussi arbre de contrôle) a été introduit pour la détection de la terminaison par [DIJ 80]; il a été abondamment repris et utilisé dans de nombreux algorithmes distribués. Dans [CHA 83] et [HEL 87], un tel arbre est utilisé pour permettre à un processus de savoir s'il est ou non définitivement bloqué.

[HUD 82] a été le premier à utiliser un arbre de contrôle pour déterminer les objets accessibles dans un système distribué. [ALI 85] a rajouté à [HUD 82] un mécanisme de crédit permettant à un instant donné d'avoir au maximun k messages de marquage en transit et, par conséquent, de prévoir l'espace nécessaire pour les traiter.

Enfin, [TEL 87], en supposant un processus par objet, construit une arborescence sur les objets et non pas sur les collecteurs comme nous l'avons proposé; ainsi, il montre que l'algorithme de marquage de [HUD 82] peut être obtenu par transformation de l'algorithme de détection de la terminaison de [DIJ 80].

V.2 Références en transit lors de l'activation des collecteurs

On conserve l'hypothèse où chaque mutateur est suspendu lors de l'activation du collecteur placé sur le même site.

Le partage d'objets entre sites et les délais de communications nous amènent à considérer les traitements associés aux références en transit lors de l'activation des collecteurs.

Considérons la situation suivante où M désigne le mutateur et C le collecteur :

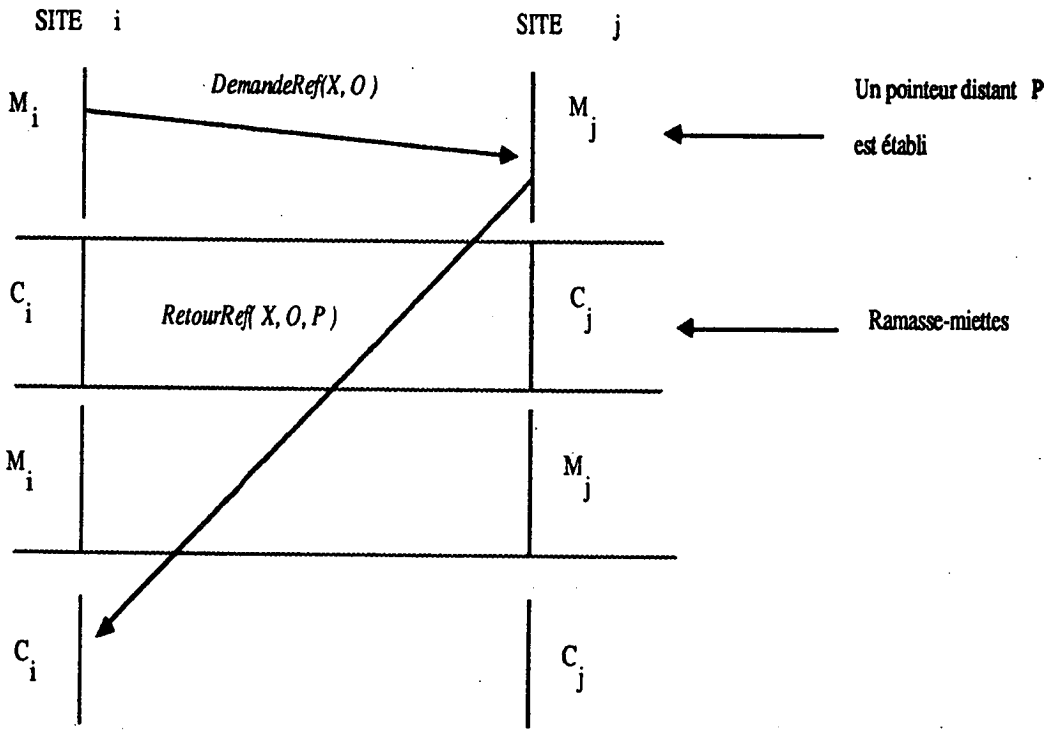


Figure V.3

Lors de l'exécution du ramasse-miettes, l'objet désigné par X, et attribué par M_j, n'est pas connu du site i; il n'y a donc pas de message de demande de marquage de C_i à C_j pour cet objet, et C_j peut récupérer "à tort" l'emplacement occupé par l'objet X et sa descendance.

Le problème se pose également lorsque la réception du message *RetourRef* est consécutive à une demande *PasserRef* à un site non propriétaire.

La propriété à maintenir est de considérer accessible tout objet *P* figurant dans un message *RetourRef*. Pour éviter la récupération de tels objets (attribués et non référencés), plusieurs solutions sont envisagées selon que le mutateur participe ou non au marquage des objets.

V.2.1 Les mutateurs ne participent pas au marquage des objets

Pour vérifier la propriété précédente, nous présentons deux solutions nécessitant des hypothèses supplémentaires pour le réseau de communications :

- la première suppose la connaissance du délai maximum (Δ) de transmission d'un message entre tout couple de sites, (le temps de traitement du message est considéré comme non significatif ou compris dans Δ),
- la seconde, proposée dans [ALI 85], considère des canaux de liaison FIFO (i.e. le déséquencement de messages n'est plus possible).

V.2.1.1 Le délai maximum Δ est connu

Lorsqu'un message de demande de référence (*DemandeRef* ou *PasserRef*) émis par M_i à l'instant t est traité par M_j (au plus tard à l'instant $t + \Delta$) , le site i reçoit l'acquittement *(RetourRef)* au plus tard à l'instant $t + 2\Delta$.

Pour recevoir les messages *RetourRef* on introduit une attente de 2Δ avant le début du marquage :

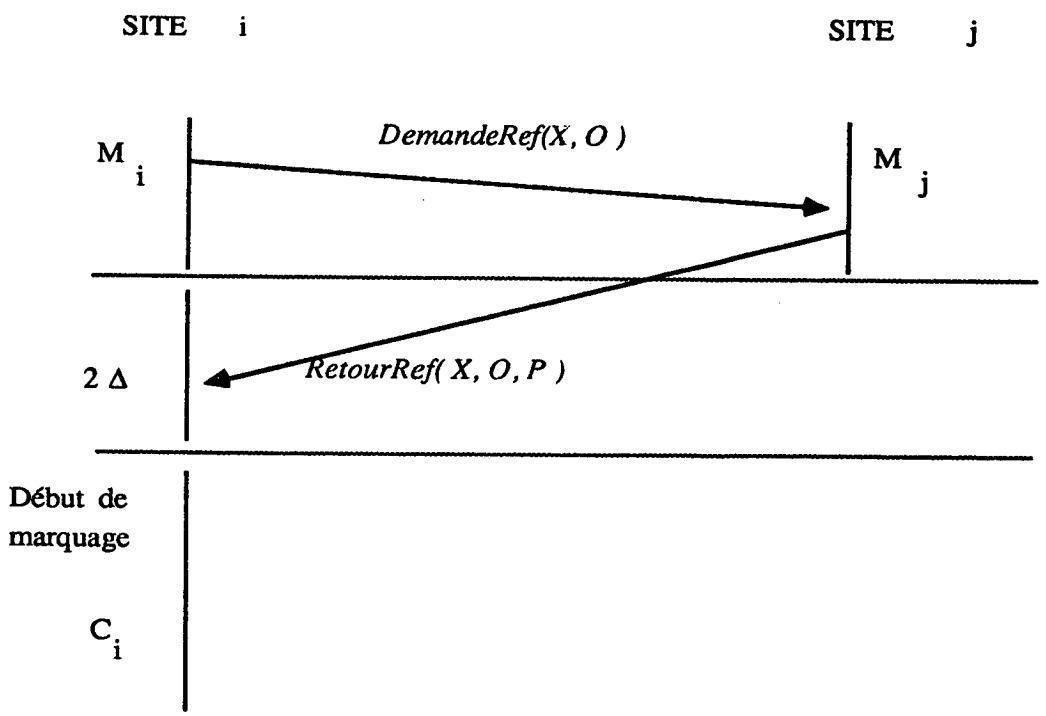


Figure V.4

Lors de la réception de la première demande de marquage, provenant du synchroniseur (*DebMarquage*) ou d'un collecteur (*Marquage*) (cf. algorithme présenté en V.1.2), le collecteur C_i attend un temps Δ pendant lequel il prend en compte les messages *RetourRef* :

si non $reçu_i$ **alors**
  jqa attente de 2 Δ
  faire
    Lors de la réception de *RetourRef(X, O, P)* **de** site j
    faire
      envoyer *Marquage(P.entrée)* **à** *P*.site;
      {*P*.site = site j pour l'acquittement de *DemandeRef* }
      {*P*.site ≠ site j pour l'acquittement de *PasserRef* }
      $nbacq_i := nbacq_i + 1;$
      {le message *RetourRef* sera traité par le mutateur lors de sa réactivation }
    fait
  fait;
  $recu_i$:= **vrai;** puis cf algorithme donné en V.1.2
fsi

On peut diminuer le temps d'exécution de la phase de marquage d'un collecteur en prenant en compte les messages *RetourRef* avant la fin du marquage. Pour cela, chaque phase de marquage doit durer au moins 2Δ (dans le cas contraire une attente est nécessaire), accepter les messages *RetourRef(X, O, P)* et demander le marquage de l'objet *P*.

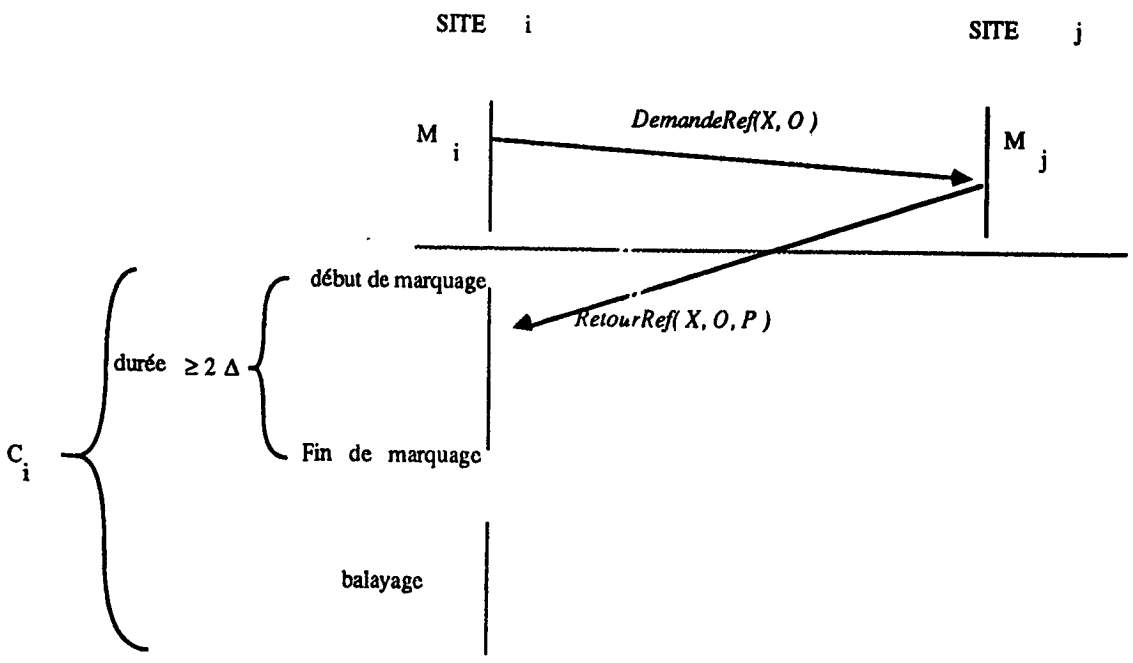


Figure V.5

V.2.1.2 Les canaux de liaison sont FIFO

L'algorithme proposé dans [ALI 85] repose sur cette idée d'assurer la réception des messages *RetourRef* avant la fin des phases de marquage. La connaissance du délai n'est plus imposée; par contre, les canaux de liaison sont supposés FIFO et chaque collecteur doit connaitre le nombre n de sites.

Lorsqu'un site i décide de lancer l'algorithme de ramasse-miettes, son collecteur joue le rôle de synchroniseur et avertit les autres collecteurs par :

envoyer *LancerMarquage* **à** site j , $\forall$ j $\neq$i;
puis, il effectue sa phase de marquage
et attend la réception de (n-1) messages *DebMarquage* provenant des autres sites.

Pour chaque collecteur C_j (j$\neq$ i) :
Lors de la réception de *LancerMarquage* **de** site i
faire
envoyer *DebMarquage* **à** site k , $\forall$ k$\neq$j ;
phase de marquage avec attente de (n-2) messages *DebMarquage* provenant des autres sites k ($\forall$ k $\neq$i, j)
...
fait

Tous les canaux FIFO étant visités (n(n-1) canaux logiques), les messages envoyés par chaque mutateur, avant le message *LancerMarquage* (pour le site i) ou *DebMarquage* (pour les autres sites), parviennent à destination avant la fin des phases de marquage.

V.2.2 Les mutateurs participent au marquage des objets

Tout objet *P* figurant dans un message *RetourRef* peut être considéré accessible, sans faire d'hypothèses supplémentaires pour le réseau de communications, lorsque les mutateurs participent au marquage des objets.

Le mutateur M_j, lors de l'attribution d'une référence X au mutateur M_i sur réception du message *DemandeRef(X, O)* , "gèle" l'objet désigné par X [RUD 86], c'est-à-dire le considère marqué jusqu'à sa première utilisation par M_i.

Ainsi le mutateur M_j exécute :

Lors de la réception de *DemandeRef(X, O)* **de** site i
faire
construire le pointeur distant P:= (j, d);
geler $TabDef_j[d]$ pour le site i;
envoyer *RetourRef(X, O, P)* **à** site i
fait

Lors de la réception de *AccèsRef(d,...)* **de** site i
faire
traitement de l'accès à la référence;
si $TabDef_j[d]$ est gelé pour le site i **alors** dégeler $TabDef_j[d]$ pour le site i
fsi
fait

Un objet gelé, pour au moins un des sites le référençant, doit être considéré comme accessible par l'algorithme de marquage du collecteur.

Le mutateur M_j, lors de la transmission d'une référence au mutateur M_i sur réception de *PasserRef(X, O)*, "gèle" la réference de l'objet désigné par *X* :

Lors de la réception de *PasserRef(X, O)* **de** site i
faire
récupérer le pointeur distant *P* {entrée *r* de $TabRef_j$ };
geler $TabRef_j[r]$ pour le site i;
envoyer *RetourRef(X, O, P)* **à** site i;
fait

L'objet $TabRef_j[r]$ contient le pointeur distant *P=(k, d);* aussi, lors de la prochaine activation du collecteur C_j, un message de demande de marquage de l'objet *P* sera envoyé au site k.

Le mutateur M_i, après son premier accès à l'objet *P*, doit demander à M_j de "dégeler" $TabRef_j[r]$ pour le site i.

V.3 Exécution simultanée sur chaque site du mutateur et du collecteur

Sur chaque site i, le mutateur M_i et le collecteur C_i s'exécutent simultanément [AUG 87], [HUD 82], [HUD 83], [RUD 86].

Chaque graphe local est dynamique et M_i doit participer au marquage des objets lors de l'ajout d'arcs (cf II.2) :

<ajout(A, C) et **si** C.couleur =blanc **alors** mettre_en_file(f_i,C); C.couleur := noir **fsi** >

Enfin, les références en transit peuvent être traitées par le mécanisme de "gel" abordé au paragraphe précédent.

VI Récupération différée des objets partageables

Les algorithmes de ramasse-miettes présentés dans les paragraphes IV et V ne distinguent pas, lors de la récupération, les objets privés des objets partageables.

Le principe retenu ici consiste à privilégier la récupération des objets privés : chaque site réalise ce travail indépendamment des autres sites en employant une des techniques proposées dans le paragraphe II et la récupération des objets partageables est effectuée de manière "paresseuse" [LIS 86]. Un site particulier, appelé détecteur, gère une vue globale du graphe des objets partageables.

VI.1 Principe de la récupération différée

Chaque site active, quand il le souhaite, un algorithme de ramasse-miettes; compte-tenu du type des informations qui doivent être déterminées pour les besoins du détecteur, un algorithme de type marquage est mieux adapté qu'une approche par compteurs de références.

Le collecteur local C_i calcule une arborescence locale des objets accessibles depuis R^i_A, R^i_L et $TabDef_i$; ceci permet au site i :

- de récupérer immédiatement les objets privés inaccessibles,
- de déterminer l'ensemble des objets distants manipulés par l'arborescence locale; cet ensemble est envoyé dans un message *Info* au détecteur, alors que les autres objets distants (non référencés) sont retirés de $TabRef_i$.

A l'aide des informations transmises par les collecteurs, le détecteur détermine les objets partageables non référencés par d'autres sites que leur propriétaire; il adresse à chaque site concerné un message *NonRéférencés*(E_i).

Sur réception du message *NonRéférencés*(E_i) , le site i retire de sa table $TabDef_i$ tous les objets nommés dans l'ensemble E_i : les emplacements correspondants peuvent être récupérés (s'ils ne sont pas accessibles localement) au cours du cycle suivant de ramasse-miettes.

Exemple : pour simplifier le schéma, nous notons les pointeurs distants par une lettre, plutôt que d'employer un doublet (n° site i, entrée dans $TabDef_i$)

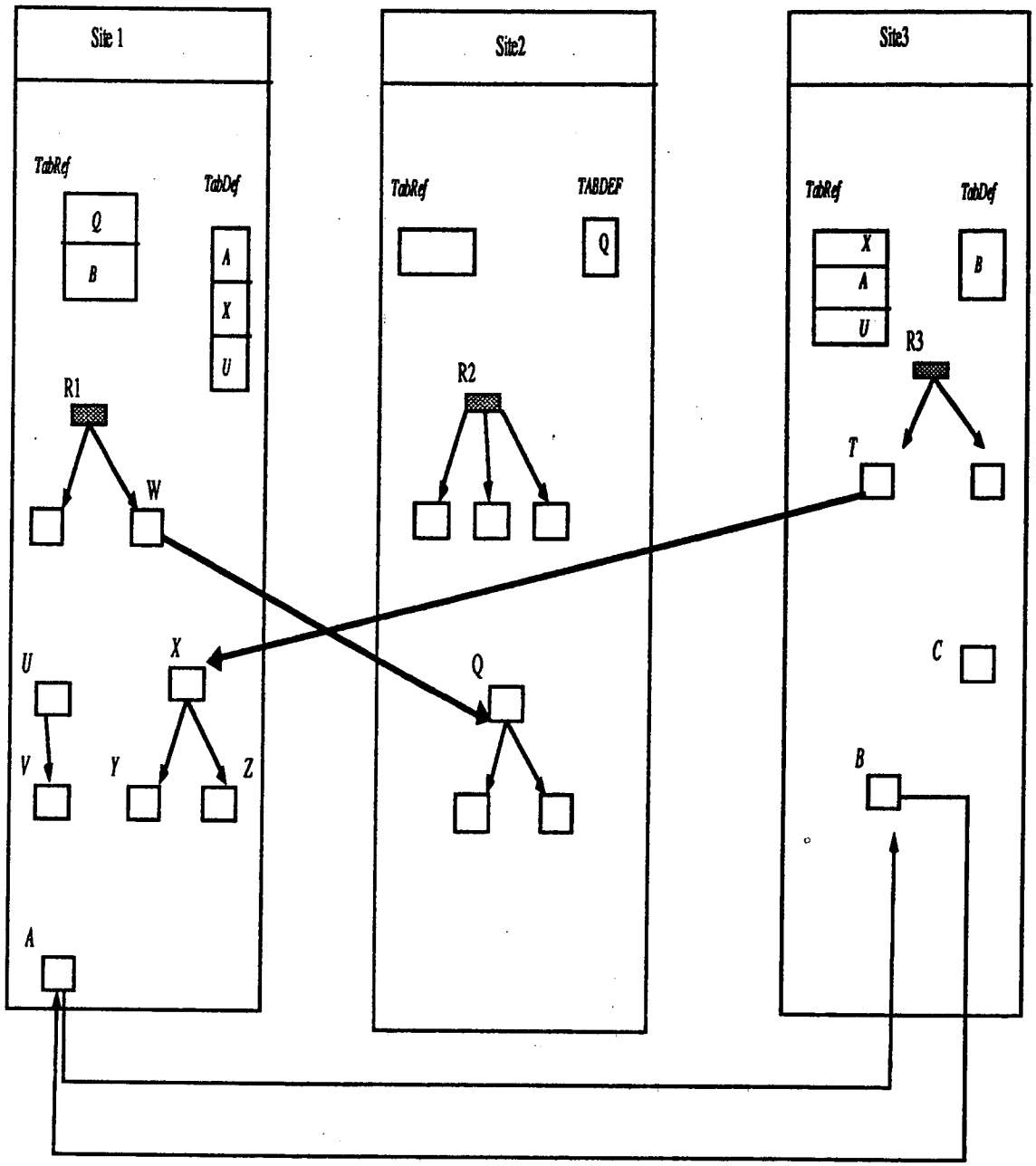


Figure VI.1

Lors de son activation, le collecteur du site 1 ne récupère aucun objet privé et envoie le message *Info({Q, B})* au détecteur.

Au niveau du site 3 le collecteur récupère l'objet privé C, retire U de $TabRef_3$ et envoie *Info({X, A })* au détecteur.

La vue du détecteur évolue de la façon suivante :

état avant réception des messages *Info* :

Détecteur
à chaque site est associé l'ensemble des objets qu'il référence 1 : { Q, B } 2 : ∅ 3 : {X, A, U}

état après réception des messages *Info* :

Détecteur
à chaque site est associé l'ensemble des objets qu'il référence 1 : { Q, B } 2 : ∅ 3 : {X, A}

et le détecteur envoie le message *NonRéférencés ({U})* à site 1.

Lors de la réception de *NonRéférencés({U})* **par** le site 1
faire
retirer U de $TabDef_1$ {U devient privé }
fait

Lors de sa prochaine activation, le collecteur C_1 récupère les objets U et V (devenus inaccessibles).

VI.2 Gestion d'un graphe global des objets partageables

L'exemple précédent met en évidence l'insuffisance des informations connues par le détecteur : celles-ci ne permettent pas de récupérer les circuits inaccessibles (A--B--A pour la figure VI.1). Il faut, en fait, que le détecteur possède aussi les arcs du graphe des objets partageables alors qu'on ne lui fournit, pour l'instant, que les sommets de ce graphe. Les arcs sont de deux natures :

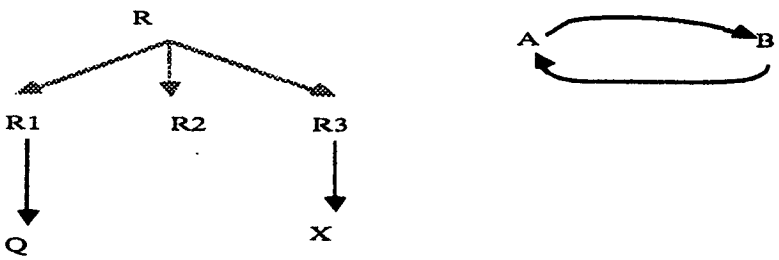
- ceux qui lient une racine R^i_A à un pointeur distant $D_{i,j}$; seule l'extrémité $D_{i,j}$ est utile et est placée par le collecteur C_i dans l'ensemble Acc_i (R_1--Q),
- les arcs (P_i, $D_{i,j}$) tels que P_i est un objet partageable, appartenant au site i, non accessible depuis R^i_A et il existe, sur le site i, un chemin de P_i à $D_{i,j}$; ces arcs sont placés, par le collecteur C_i, dans l'ensemble $Chemins_i$ (A--B, B--A).

Après chaque cycle de collection C_i, le site i envoie Acc_i et $Chemins_i$ au détecteur . Si l'on prend l'exemple précédent , à l'issue de la réception des *Info* émis par les sites 1 et 3, le détecteur possède la vue globale suivante :

Détecteur

	Acc	Chemins
1	Q	(A,B)
2	ø	ø
3	X	(B, A)

Soit :



où R représente l' "union" des racines des sites.

Figure VI.2

Le calcul de la fermeture transitive de ce graphe permet au détecteur de déterminer l'inaccessibilité des objets A et B et d'en avertir les sites 1 et 3.

Le détecteur a donc pour rôle de déterminer une arborescence globale ne concernant que les objets partageables, alors que les techniques proposées en V calculent une arborescence globale sur la totalité des objets.

VI.3 Généralisation : références en transit et distribution du détecteur

Dans le pragraphe VI.2, nous avons ignoré les point suivants :

- le traitement des références en transit,
- la mise en œuvre du détecteur.

Notre démarche a consisté à présenter, d'abord, les principes retenus pour une récupération différée; il nous faut, maintenant, préciser que ceci résulte de l'étude des travaux de [BEC 86], [HUG 85] et [LIS 86] mais ces auteurs abordent les problèmes précédents d'une manière différente.

VI.3.1 Algorithme de Beckerle et Ekanadham

L'approche retenue dans [BEC 86] prolonge l'étude de [ALI 85] et emploie le mécanisme de références pondérées pour le partage d'objets distants (cf IV); le problème des références en transit est ainsi résolu.

Par contre, la détection des circuits inaccessibles entre objets distants nécessite l'introduction d'un détecteur spécialisé. Celui-ci fonctionne suivant le principe décrit au paragraphe VI.2.

Enfin, aucune stratégie n'est proposée pour décentraliser le détecteur; ceci pose évidemment le problème crucial lié à une panne du site détecteur.

VI.3.2 Algorithme de Hughes

L'algorithme proposé [HUG 85] suppose un délai de transmission nul, le problème des références en transit est donc, ainsi, éludé.

La méthode envisagée impose l'existence d'une horloge logique globale et à tout objet (local ou non) est associée une estampille indiquant l'heure du dernier accès à cet objet. Chaque site i active, quand il le désire, son collecteur local C_i; celui-ci travaille indépendamment des autres collecteurs. Le collecteur C_i propage les estampilles à tous les objets (locaux ou distants) accessibles par le site i.

A la fin de sa phase de collection, chaque site i conserve l'heure courante dans une variable locale $REDO_i$ et cherche à connaître la valeur minimale MINREDO de l'estampille de tous les objets accessibles du système ($MINREDO = \underset{\text{site i}}{\text{Min}}(REDO_i)$).

Ainsi, les objets dont l'estampille est inférieure à MINREDO peuvent être récupérés sur les différents sites.

Le travail du détecteur, consistant à calculer la valeur minimale (MINREDO), est distribué sur l'ensemble des sites et est mis en œuvre sur un anneau logique.

VI.3.3 Algorithme de Liskov et Ladin

La proposition effectuée dans [LIS 86] est dérivée des travaux réalisés pour le langage ARGUS [LIS 84] où la préoccupation principale est le comportement en cas de pannes. Le mode de dialogue retenu amène à parler de serveur plutôt que de détecteur; chaque site :

- informe le serveur à l'issue de chaque cycle de collection,
- sollicite le serveur par un message particulier (*Query*) afin de connaître les objets publics (partageables) lui appartenant et qui ne sont référencés par aucun site.

Les références en transit sont gérées par le serveur et le site émetteur : il n'y a pas de mécanisme d'acquittement entre le site émetteur et le site récepteur; ceci a pour conséquences:

- un gel au niveau du site émetteur, jusqu'à ce que le serveur ait pris en considération la référence en transit,
- l'introduction des deux contraintes suivantes :
  * la connaissance d'un majorant du délai Δ de transmission entre tout couple de sites,
  * l'utilisation d'horloges locales vérifiant la propriété : $|H_i - H_j| \leq \varepsilon$ connu $\forall i,j\ (i \neq j)$.

Une gestion d'objets distants par références pondérées (cf IV) ou par une technique de gel (cf V) permettrait de lever ces deux contraintes introduites dans ARGUS pour d'autres applications [LIS 84].

L'approche de [LIS 86] a, pour des considérations de résistance aux pannes, proposé une "distribution" du serveur. Celui-ci est "repliqué" en plusieurs agences, possédant une vue globale du graphe des objets partageables; chaque collecteur informe une agence et chaque site sollicite une agence du serveur quant à l'inaccessibilité de ses objets; les agences échangent leurs connaissances afin d'avoir une vue du graphe aussi récente que possible.

VII Conclusion

Nous avons présenté trois catégories d'algorithmes de ramasse-miettes distribués; nous résumons, brièvement ici, les avantages et les inconvénients de chacune d'elles.

Algorithmes de type compteurs de références

Les inconvénients majeurs résident dans :

- leur inadaptation à détecter les circuits inaccessibles, ce qui nécessite généralement l'usage d'une technique mixte compteurs/marquage,
- la place mémoire occupée par les compteurs et la valeur maximale imposée à ceux-ci,
- la nécessité de faire réaliser, par les mutateurs, le travail dévolu aux collecteurs (envois de messages pour incrémenter/ décrementer les compteurs ...).

On peut noter que des collecteurs locaux de type marquage permettent d'éviter ces inconvénients au niveau de chaque site; cependant, ceux-ci demeurent pour les objets partageables. Par contre :

- la récupération des objets inaccessibles n'implique aucune synchronisation,
- le traitement des références en transit est résolu de façon simple par un mécanisme de références pondérées.

Algorithmes de type marquage global

Les principales critiques à leur sujet sont dues :

- au lancement simultané de tous les collecteurs, mêmes ceux des sites non concernés par une éventuelle récupération,
- à l'existence de synchronisations à la fin des phases de marquage et de balayage,
- à la place mémoire (file ou pile) nécessaire à la phase de marquage des collecteurs.

L'avantage principal de ces algorithmes est la récupération de tous les objets inaccessibles, au plus tard lors de la prochaine activation du ramasse-miettes. Dans le cas d'une exécution simultanée du mutateur et du collecteur, ce dernier fonctionne par cycles et la récupération est plus rapide; toutefois, le mutateur doit alors participer au marquage lors des ajouts d'arcs.

Algorithmes du type récupération différée

Ils permettent, essentiellement, de disposer de collecteurs indépendants et d'éviter tout problème de synchronisation. La récupération "paresseuse" peut cependant s'avérer dangereuse (saturation au niveau d'un site par des objets partageables qui ne sont plus accédés) si certains sites n'activent jamais leur collecteur; il faut prévoir un mode "panique" entraînant l'activation de certains collecteurs pour éviter un tel problème.

Qualités souhaitées pour un algorithme de ramasse-miettes distribué

Nous énumérons ci-dessous les propriétés attendues d'un "bon" algorithme de ramasse-miettes distribué :

1 - la synchronisation doit être réduite; les collecteurs doivent être le plus indépendants possible (i.e. distinction entre objets locaux et partageables),

2 - les références en transit (émanant du site propriétaire ou d'un autre site) doivent être traitées de manière simple,

3 - tout objet inaccessible doit être récupéré avant que l'espace qu'il occupe ne devienne nécessaire; en particulier, une récupération différée des objets partageables ne doit en aucun cas bloquer un site sur lequel résident des objets inaccessibles,

4 - lorsqu'un site désire obtenir des informations sur l'accessibilité de ses objets partageables, seuls les sites concernés par l'utilisation de ces objets doivent être consultés,

5 - les travaux de collection d'objets partageables ne doivent pas être confiés à un site spécial,

et de plus :

- les contraintes du réseau (communications FIFO ou instantanées) ou de connaissance globale (heure globale, topologie particulière, connaissance du nombre de sites ...) sont à éviter,
- la solution proposée doit résoudre tout ou partie des points précédents sans engendrer une dépense mémoire prohibitive ni un envoi excessif de messages; il faut, par ailleurs, qu'elle permette un comportement correct lorsque des pannes se produisent (déséquencement, duplication, perte de messages, panne de lignes ou de site...).

Aucun des algorithmes que nous avons étudié ne respecte la totalité des "qualités" précédentes. Nous ne souhaitons pas proposer ici un nouvel algorithme qui tendrait à "faire mieux" que les autres; cependant, nous pensons qu'il est possible de retenir, parmi tous ces algorithmes, les aspects positifs tendant à satisfaire une grande partie des points énoncés.

Les points 1 et 3 rejettent respectivement les techniques de marquage global et les approches par compteurs de références. Il ne subsiste donc que les méthodes dites de "type récupération différée"; malheureusement, les trois techniques de ce type analysées dans ce document présentent toutes des inconvénients (détecteur/serveur spécialisé ou liaisons instantanées).

Le point 2 (références en transit) peut être résolu, à faible coût, par un mécanisme de "gel" ou mieux par un partage basé sur les références pondérées. Pour satisfaire le point 4, on peut remarquer que, lorsqu'un site désire acquérir des informations quant à l'accessibilité de ses objets partageables, il est préférable de procéder par une construction ascendante et non descendante : il s'avère nécessaire de gérer, au niveau de chaque site, les informations utiles pour une telle construction; ceci engendre un nouveau besoin de mémoire, mais celui-ci est limité seulement aux objets partageables.

Lors du blocage d'un site, par manque de place, la récupération de ses objets partageables susceptibles d'être libérés peut être garantie par une procédure d'urgence basée sur un parcours ascendant. Une telle méthode permet de laisser chaque site décider du besoin d'activer ou non la procédure d'urgence (i.e. point 5, pas de spécialisation des sites).

BIBLIOGRAPHIE

[ABR 87] S.G. ABRAHAM , J.H. PATEL
Parallel Garbage Collection on a virtual memory system
proceed. of the 1987 Intern. conf. on Parallel processing, août 1987, Springer Verlag, pp.243-246

[AHS 87] K. APPLEBY, S. HARIDI, D. SAHLIN
Garbage Collection for PROLOG based on WAM
IBM Research Report RC 12941 (#57635) mars 1987, 27p.

[ALI 85] K.M. ALI et S. HARIDI
Global Garbage Collection for Distributed Heap Storage Systems
IBM Research Report RC 11082 (# 49769) avril 1985, 46p.

[APP 87] A.W. APPEL
Garbage Collection can be faster than stack allocation
Inf. proces. Letters 25 (1987), pp.275-279.

[AUG 87] L. AUGUSTEIJN
Garbage Collection in a Distributed Environment
LNCS n°259, Parallell Architectures and Languages Europe Volume II Eindhoven, Springer Verlag, juin 1987, pp. 75-93

[BAK 78] H.G. BAKER
List Processing in real-time on Serial Computer
Comm.ACM vol 21, 4, april 1978, pp.280-294

[BEC 86] M.J. BECKERLE, K. EKANADHAM
Distribtued Garbage Collection wirth no Global synchronization
IBM Research Report RC 11667 (#52377) janvier 1986, 10p.

[BEK 86] Y. BEKKERS, B. CANET, O. RIDOUX, L. UNGARO
Mali : a memory with a real-time garbage collector for implementing logic programming languages
Proc. of the 3rd Symp. on Logic Program., IEEE, sept. 1986, Salt-Lake City USA, pp.258-264

[BEN 84] M. BEN-ARI
Algorithms for on-the-fly garbage collection
ACM trans. on prog. lang. and syst., vol 6, n°3, juillet 1984, pp.333-344.

[BEV 87] D.I. BEVAN
Distributed Garbage collection using reference counting
LNCS n°259, Parallell Architectures and Languages Europe Volume II
Eindhoven, Springer Verlag, juin 1987, pp. 176-187

[BOB 80] D.G. BOBROW
Managing Reentrant Structure using Reference Counts
ACM Trans on prog. lang. and syst., Vol. 2 n°3, juillet 1980, pp. 269-273

[BRO 85] D.R. BROWNBRIDGE
Cyclic Reference Counting For Combinator Machines
LNCS n°201, Functional Programming Languages and Computer Architecture, Nancy, France, septembre 1985,Springer Verlag, pp.273-288

[CHA 83] K.M. CHANDY, J. MISRA, L. HAAS
Distributed Deadlock Detection
ACM TOCS, Vol. 1,2, may 1983, pp. 144-156

[CHR 84] T.W. CHRISTOPHER
List processing in real time on a serial computer
Comm.ACM, vol 21, n°4, avril 1978, pp. 280-294

[COH 81] J. COHEN
Garbage Collection og Linked Data Structures
Computing Surveys, Vol 13, n°3, sept. 1981, pp. 341-367

[COH 83] J. COHEN, A. NICOLAU
Comparison of Compacting Algorithms for Garbage Collection
ACM trans. on prog. lang. and syst., vol 5, n°4, oct. 1983, pp.532-553

[DEU 76] L.P. DEUTSCH, D.J. BOBROW
An efficient incremental automatic garbage collector
Comm. ACM Vol. 19, 9, sept. 1976, pp. 522-526

[DIJ 78] E.W. DIJKSTRA, L. LAMPORT, A.J. MARTIN, C.S. SCHOLTEN, E.F.M. STEFFENS
On-the-fly garbage collection : an exercice in cooperation
Comm.ACM Vol 21 n°11, nov. 1978, pp. 966-975

[DIJ 80] E.W. DIJKSTRA, C.S. SCHOLTEN
Terminaison Detection for Diffusing Computations
Inf. Proc. Letters, Vol. 11,1, august. 1980, pp. 1-4

[ECK 87] J. D. ECKART, R.J. LEBLANC
Distributed Garbage Collection
Proceed. of the SIGPLAN'87 Symp. on Interpreters and Interpretive Techniques, St. Paul, Minnesota, juin 1987,SIGPLAN Notices Vol 22 n° 7, juin 1987, pp.264-273

[ELL 88] J.R. ELLIS, K. LI, A.W. APPEL
Real-Time Concurrent Collection on Stock Multiprocessors
Digital Systems Research Center, février 1988, 23p.

[FEN 69] R. FENICHEL, J. YOCHELSON
A Lisp Garbage Collector for Virtual-memory Computer Systems
Comm. ACM Vol. 12, 11, nov. 1969, pp. 611-612

[FRI 79] D. P. FRIEDMAN, D.S. WISE
Reference Counting can manage the circular environments of mutual recursion
Inf. Proces. letters, Vol 8, n° 1, January 1979, pp.41-45.

[HAL 84] R. H. HALSTEAD
Implementation of Multilisp : LISP on a Multiprocessor
Conf. Record of the 1984 ACM symp. on LISP and Functional Programming, Austin, texas, août 1984, pp.9-17

[HEL 87] J.M. HELARY, A. MADDI, M. RAYNAL
Controlling Information Transfers in Distributed Algorithms : Application to Deadlock Detection
Inter. Conf. on Parallel Processing and Applications, L'Aquila, Italy, sept. 1987, pp.87-96.

[HIC 84] T. HICKEY, J. COHEN
Performance analysis of on-the-fly garbage collection
Comm. ACM vol 27, n°11, 1984, pp.1143-1154.

[HUD 82] P. HUDAK , R. M. KELLER
Garbage Collection and Task Deletion in Distributed Applicative Processing Systems
Conf. Record 1982 ACM Symp. on LISP and Functional Programming, ACM, 1982, pp.168-178

[HUD 83] P. HUDAK
Distributed Task and Memory management
Proceed. of the second Annual ACM symp. on Principles of Distributed Computing, Montreal, Canada, août 1983, pp.277-289.

[HUG 85] J. HUGHES
A distributed Garbage Collection Algorithm
LNCS N°201, Functional Programming languages and Computer Architecture, Nancy, France, Septembre 1985, Springer Verlag, pp. 256-272

[KNU 73] D.E. KNUTH
The art of computer programmin Vol I : Fundamental Algorithms
Addison-Wesley, Reading, Mass. 1973

[LER 86] C.W. LERMEN, D. MAURER
A protocol for distributed reference counting
Proceed. of the 1986 ACM Conf. on LISP and Functional programming, Cambridge, Massachussets, août 1986, pp.343-350

[LIE 83] H. LIEBERMAN, C. HEWITT
A Real-Time Garbage Collecteur Based on the Lifetimes of Objects
Comm.ACM, Vol 26, n° 6, juin 1983, pp. 419-429.

[LIS 84] B. LISKOV
The Argus Language and System
LNCS n°190, Springer Verlag, 1984, pp.343-430.

[LIS 86] B. LISKOV, R. LADIN
Highly-Available Distributed Services and Fault-Tolerant Distributed Garbage Collection
Proceed. of the Fifth annual ACM Symp. on princ. of distributed computing, Calgary, Alberta, Canada, août 1986, pp.29-39

[PIT 85] E. PITTTOMVILS, M. BRUYNOOGHE, Y.D. WILLEMS
Towards a real time garbage collector for Prolog
Proc. Symp. on Logic Programm., 1985, pp. 185-198

[RAM 85] A. RAM, J.H. PATEL
Parallel garbage collection without synchronization overhead
Proceed. int. symp. on computer architecture, 1985, pp. 84-90

[RID 87] O. RIDOUX
Deterministic and Stochastic Modeling of Parallel Garbage Collection - Towards Real - Time Criteria
The 14th Annual Inter. Symp. on Computer Architecture,juin 1987, Pittsburgh, pp. 128-136

[RUD 86] M. RUDALICS
Distributed Copying Garbage Collection
Proceed. of the ACM Conf. on LISP and Functional Programming, Cambridge, Massachusetts, août 1986, pp.364-372

[SHI 85] H. SHIN, M. MALEK
Parallel garbage collection with associative tag
Proceed. int. symp. on computer architecture, 1985, pp. 369-375

[SPE 82] D. SPECTOR
Minimal Overhead Garbage Collection of Complex List Structure
ACM SIGPLAN Notices, Vol 17, 3, 1982, pp. 80-82

[TEL 87] G. TEL, R.B. TAN, J. VAN LEEUWEN
The derivation of the on-the-fly garbage algorithms from distributed termination detection protocols
LNCS 247, 4 th annual symp. on theoretical aspects of computer science, Passau, Allemagne fédérale , fevrier 1987, pp.445-455.

[VAN 87] J. L. A. VAN DE SNEPSCHEUT
"Algorithms for on-the-fly garbage collection" revisited
Inf. proces. Letters 24 (1987) , pp. 211-216

[WAT 87] P. WATSON, I. WATSON
An efficient garbage collection scheme for parallel computer
LNCS n°259, Parallell Architectures and Languages Europe Volume II
Eindhoven, Springer Verlag, juin 1987, pp.432-443

[WEI 69] J. WEIZENBAUM
Recovery of Reentrant List Structures in SLIP
Comm. ACM, Vol 12, 7, july 1969, pp.370-372

[WIS 77] D.S. WISE, D. P. FRIEDMAN
The One-bit Reference Count
Bit 17 (1977), pp. 351-359

[WIS 85] D. S. WISE
Design for a multiprocessing Heap with on-board reference counting
LNCS N°201 , Functional Programming languages and Computer Architecture,Nancy, France, Septembre 1985,Springer Verlag, pp. 289-304.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique