

# Garbage collector asynchrone, complet et distribué pour le projet NGrid

Projet logiciel personnalisé 2A - Séquence 7 - 2006

École Supérieure d'Électricité  
Supélec

Enseignant chercheur M. VIVINIS

Élèves

CARRÉ Grégory

HARCHAOUI Warith

JACQUIN Jonathan

## Résumé

Luis Veiga et Paulo Feirreira [2] expliquent que le problème du ramasse-miettes distribué peut être divisé en deux :

- le ramasse-miettes acyclique, problème résolu par la méthode du *Reference Listing* [1] en 1993
- le ramasse-miettes cyclique, problème que l'on pense résolu par leur méthode du *Graph Summarizer* [2] en 2004 qui repose sur la méthode du *Reference Listing*

Le but de ce document est d'implémenter l'algorithme du *Graph Summarizer* ainsi que quelques améliorations dans l'environnement distribué NGrid [3].

# Table des matières

<b>1</b>	<b>Présentation de l'article de Veiga et Ferreira</b>	<b>4</b>
1.1	Le Garbage Collector . . . . .	4
1.2	Les différents types de déchets . . . . .	4
1.3	Un environnement distribué . . . . .	4
1.4	L'algorithme . . . . .	5
<b>2</b>	<b>Intégration de l'algorithme dans le projet NGrid.</b>	<b>11</b>
2.1	Traduction des notions bas niveau en notions haut niveau . . . . .	11
2.2	La technique d' <i>Inhumation-Sérialisation</i> . . . . .	12
<b>3</b>	<b>Optimisations et améliorations de l'algorithme de Veiga et Feirrerera grâce à l'environnement NGrid</b>	<b>12</b>
3.1	La stratégie du tampon temporel . . . . .	12
3.2	L'initiative ou stratégie de regroupement . . . . .	13
3.3	Le stockage des nombreuses données de l' <i>initiative</i> . . . . .	13
3.4	La stratégie OneWay . . . . .	14

## Remerciements

Nous remercions tout particulièrement M. Bernard VIVINS, notre chargé de projet, pour son aide indispensable apportée, ses idées et ses suggestions et pour le temps qu'il nous a consacré.

Nous remercions également M. Joannès VERMOREL qui est le fondateur du projet NGrid sans qui ce projet n'aurait pas pu voir le jour.

## Introduction

Les récents progrès dans le domaine de la biologie nécessitent d'importantes ressources informatiques. Par exemple, si une personne est intéressée par la détermination du rôle précis des gènes dans les réactions chimiques, cette personne est alors confrontée à  $3.10^9$  nucléotides.

Dans le but de réaliser des calculs intensifs, il n'est pas toujours possible pour des raisons économiques de se procurer des machines uni-processeurs telles que les supercalculateurs. De plus, cette technologie est de plus en plus abandonnée. C'est pourquoi, l'industrie et la recherche s'orientent de plus en plus vers une solution distribuée qui met en réseau plusieurs machines de coût moyen.

Ainsi, on est confronté à deux problématiques en environnement distribué :

- l'espace mémoire
- les calculs

Le but de ce document est la construction d'un gestionnaire automatique de mémoire appelé ramasse-miettes ou *Garbage collector* dans un environnement distribué. On est tout d'abord confronté à la compréhension précise de deux publications [1] [2] traitant du ramasse-miettes distribué et de l'intégration des algorithmes dans le projet d'un environnement distribué NGrid [3].

# 1 Présentation de l'article de Veiga et Ferreira

## 1.1 Le Garbage Collector

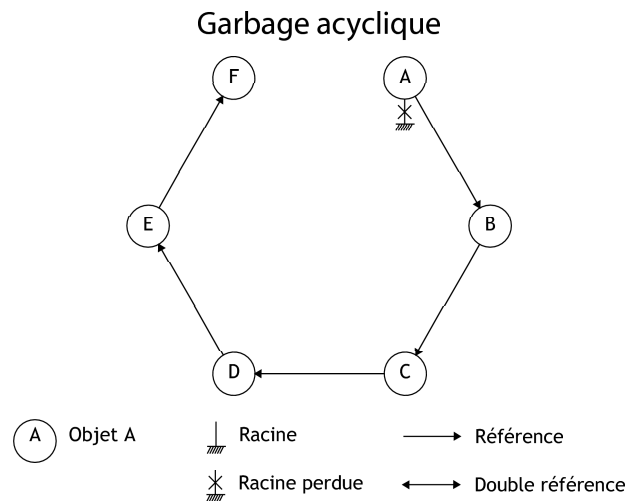
Dans les langages traditionnels de programmation sans ramasse-miettes tels que C ou C++, la mémoire est gérée par l'utilisateur et les mots-clés bien connus *malloc* et *free*. Depuis les années 90, les langages avec ramasse-miettes tels que Java, Python et C# ont un grand impact sur la façon dont les gens programment car les utilisateurs n'ont plus besoin de penser à libérer la mémoire grâce au ramasse-miettes. Dans des environnements locaux, le problème du ramasse-miettes a été résolu il y a presque 20 ans [4]. En environnement distribué, le problème se complique du fait des conflits entre les machines lors de l'accès aux données partagées. Ce phénomène est celui des *Race conditions*. En fait, le problème distribué a été partiellement résolu en 1993 par une méthode appelée le *Reference Listing* [1]. Nous pensons que le problème est résolu complètement depuis 2004 grâce à Veiga et Feirerra [2] avec un algorithme qu'on appellera *Graph Summarizer*.

## 1.2 Les différents types de déchets

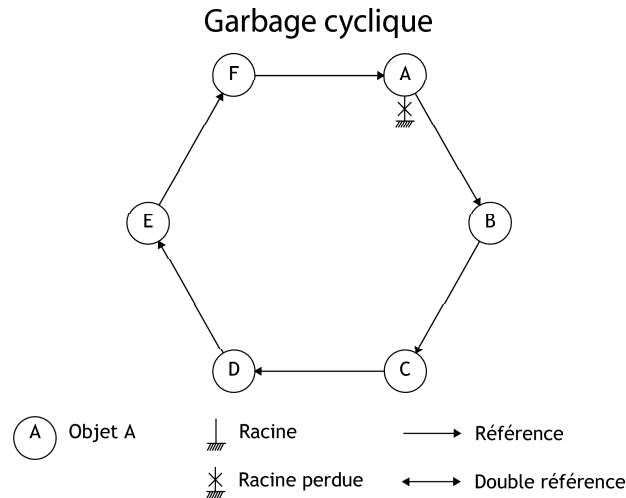
Un simple déchet *singleton* est créé à cause d'un objet local issu de l'appel d'une méthode. L'objet ne peut être atteint par aucune référence après l'exécution de la méthode mais bloque cependant l'espace mémoire. On peut dire que l'objet est un déchet singleton car il a perdu sa racine, c'est-à-dire son pointeur. En fait, il existe deux types de déchets.

**le déchet acyclique** l'objet temporaire est le premier élément d'une liste elle-même temporaire.

A la fin de la méthode, la liste entière devient un déchet car la seule racine était le premier élément. La liste est un déchet acyclique. Cette forme de déchet est rétroactivement détruite par la méthode du *Reference Listing*. On peut facilement comprendre qu'un déchet singleton est en fait un déchet acyclique.



**le déchet cyclique** l'objet initial est la racine d'un graphe cyclique qui peut être traité par l'algorithme du *Graph Summarizer* de Veiga et Feirerra.



Comme écrit précédemment, les *Race conditions* font du ramasse-miettes distribué un problème plus difficile que dans le cas local.

### 1.3 Un environnement distribué

Un ramasse-miettes efficace doit travailler en prenant compte de l'environnement qui l'entoure. Cela signifie que l'utilisateur ne doit pas avoir à se préoccuper de la gestion de la mémoire. Il doit être complètement transparent *a fortiori* dans un environnement distribué où de lourds calculs sont faits. Le projet NGrid est un système d'exploitation distribué où chaque machine possède un ramasse-miettes local hérité de l'environnement .Net pour les objets simples reliés dans la machine courante. De plus, dans NGrid, les objets distribués appelés *GObjects* (G pour Grid, grille) sont gérés par le ramasse-miettes distribué. Pour plus de simplicité, les deux sortes d'objets ne seront pas distinguées dans la mesure où ce document ne traite que des *GObjects*. Le véritable but du document est de comprendre l'algorithme du *Graph Summarizer* et de l'implémenter dans le projet NGrid. Jusqu'à présent, nous pensons que ce problème n'a jamais été traité aussi précisément dans la littérature sans synchronisation globale ni consensus global. Ainsi, nous ne sommes pas en mesure de fournir de comparaison avec d'autres algorithmes.

### 1.4 L'algorithme

Les grandes forces du *Graph Summarizer* sont :

**la complétude** : toutes les formes de déchets sont finalement collectées (déchet cyclique et acyclique).

**la tolérance aux fautes** : une machine peut disparaître (*crasher*) sans perturber le calcul du ramasse-miettes.

**l'absence d'une quelconque synchronisation globale** ou consensus, ce qui fait l'avantage crucial de l'algorithme puisqu'il économise des communications inutiles à travers le réseau.

L'idée principale du *Graph Summarizer* est simple.

*Transformer un déchet cyclique en déchet acyclique.*

En effet, la vieille stratégie du *Reference Listing* traite de façon rétroactive les déchets acycliques. Plus précisément, dès la perte de sa racine, l'objet est collecté. Ainsi, il serait intéressant

de retirer quelques éléments d'un déchet cyclique afin de le transformer en acyclique. Cependant, il reste encore le problème de la détection des déchets cycliques. Cela doit être fait sans erreur car supprimer des objets non garbage est inacceptable. La première idée que l'on peut avoir sur le problème est certes simple mais fautive. Elle ressemblerait à :

```

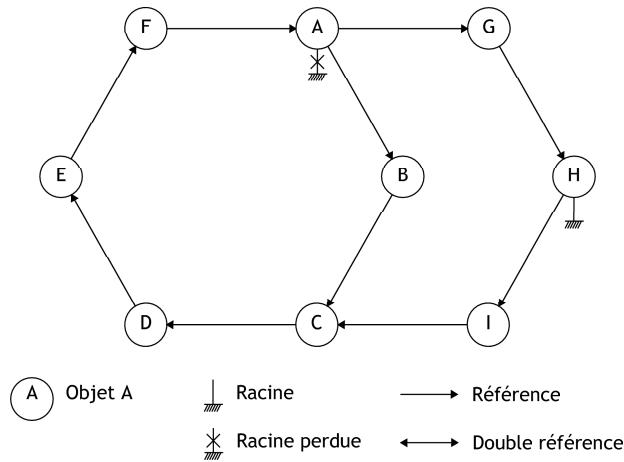
Data: Un candidat est soupçonné d'être garbage
foreach objet o lié au candidat do
  Lire le graphe local des références de o
  if o est enraciné (accessible localement depuis une racine) then
    | Le candidat n'est pas garbage
  else
    | if o est rencontré pour la deuxième fois then
      | Le candidat est garbage
    | else
      | Relancer l'algorithme avec o comme nouveau candidat
    | end
  end
end
Result: Est-ce que le candidat est garbage ?

```

**Algorithm 1:** Algorithme naïf

L'exemple qui suit montre pourquoi cet algorithme naïf n'est pas correct.

**Garbage cyclique en forme de 8**



Dans l'ordre alphabétique, en commençant par A et en finissant par F, A est rencontré deux fois sans rencontrer de racine. En fait, la racine H rend le graphe non garbage.

On peut donc facilement comprendre qu'un algorithme plus sophistiqué est requis pour détecter un déchet acyclique. Les auteurs [2] simplifient considérablement le problème en ajoutant une nouvelle hypothèse. La stratégie du *Reference Listing* [1] détruit tous les déchets acycliques. Ainsi

*Les déchets restants sont nécessairement cycliques.*

Ceci implique qu'un objet sans racine directe et sans référence (comme l'extrémité d'un graphe acyclique) ne peut exister.

La nouvelle notion introduite dans l'article est celle du *Graph Summary*. Elle contient deux ensembles :

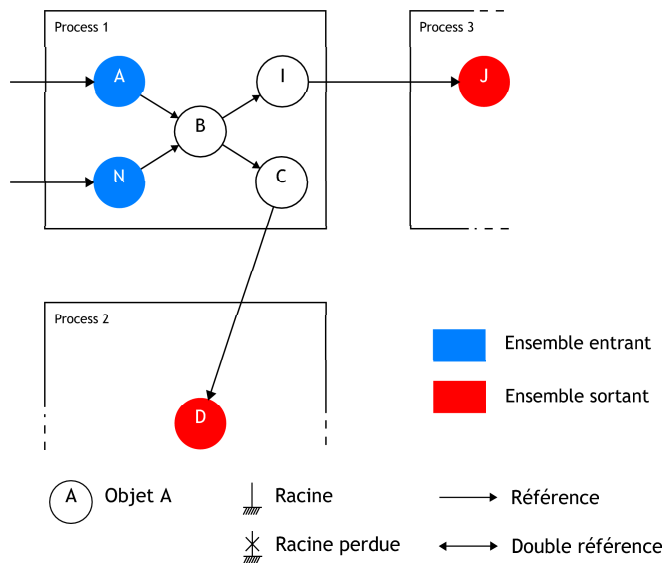
**l'ensemble entrant** composé par des identifiants d'objets du domaine avec une référence d'un autre domaine.

**l'ensemble sortant** composé par des identifiants d'objets extérieurs au domaine référencé depuis le domaine courant.

Un *Graph Summary* local résume une branche d'un graphe d'objets localement connexe dans un domaine donné.

Cet exemple <sup>1</sup> présente simplement cette notion :

*Graph summary*  
sur le Process 1 avec B comme candidat (ou A, N, I et C)



Par définition, un candidat point de départ est suspecté d'être dans un déchet cyclique. L'algorithme constitue le résumé du graphe connexe au candidat dans tous les domaines. Il n'existe que deux façons d'arrêter l'algorithme et une dernière qui ne permet pas à elle seule de conclure :

**le graphe a une racine** : la suspicion était fautive.

**les ensembles sortant et entrant sont égaux** : la suspicion était juste !

**on rencontre un objet déjà rencontré.**

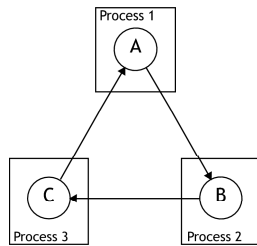
L'article de Veiga-Ferreira présente mais n'explique pas l'algorithme, il a donc fallu déduire un pseudo-code à partir de la présentation.

Lors de notre première maquette, nous avons voulu résoudre le problème le plus simple, celui d'un cycle simple contenant ou pas des racines. L'algorithme évident que nous utilisons s'appuyait sur le raisonnement suivant :

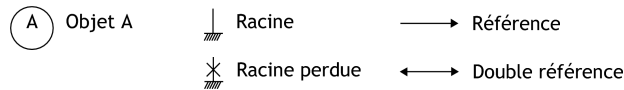
- 1) on choisit un candidat dans le process que l'on étudie. On suppose lors de ce choix qu'il y a une chance que cet objet fasse partie d'un graphe *garbage*.
- 2) on vérifie qu'au sein du process, le candidat ne soit pas relié à un objet racine, auquel cas, la détection peut continuer (sinon, on sait tout de suite que le graphe n'est pas *garbage*).

<sup>1</sup> Domaines, machines et process ne sont pas différenciés pour plus de clarté

- 3) on réalise le graph summary du process en cours (c'est-à-dire on construit les deux ensembles entrants et sortants) et on le fusionne à un graph summary dit *globale*. (on fusionne l'ensemble entrant avec les ensembles entrants déjà détectés et on fait de même pour les ensembles sortants).
- 4) on compare dans le graph summary globale ainsi obtenu l'ensemble sortant à l'ensemble entrant.
  - Si ces deux ensembles sont égaux, on a parcouru un cycle : la détection du graphe est finie et ce dernier est *garbage*.
  - Si ces deux ensembles ne sont pas égaux, le candidat n'est pas forcément garbage : la détection n'est pas finie.
- 5) on considère chaque élément de l'ensemble sortant comme étant le candidat. (appel récursif)  
Vérifions étape par étape, sur l'exemple suivant, le bon fonctionnement de ce raisonnement.



Graphe cyclique en triangle



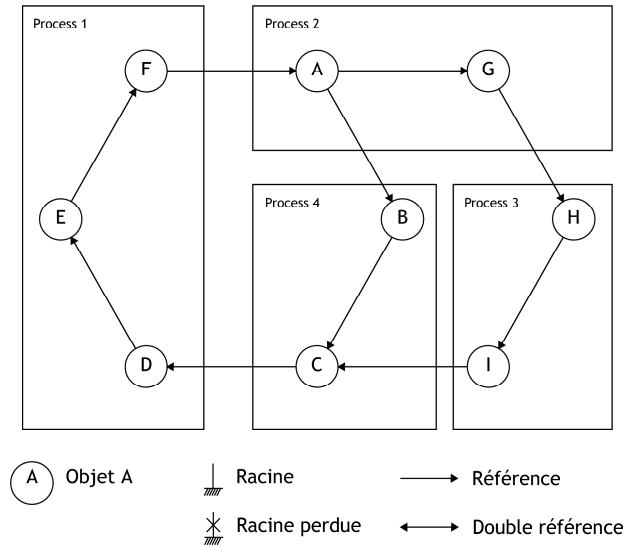
Ensemble sortant	Ensemble entrant.
A	B
AB	BC
ABC	BCA

Trace de l'algorithme

L'ensemble est *garbage* (pas de racine), notre raisonnement doit donc éliminer un tel graphe. Prenons A comme candidat. A appartient au process 1 donc on réalise en premier le *Graph Summary* du process 1. Puis on prend l'élément sortant B qui appartient au process 2 : on réalise le *Graph Summary* du process 2 et on le fusionne au *Graph Summary* globale (qui n'est pour l'instant que constitué du *Graph Summary* du process 1). Puis on fait de même pour C (process 3). On remarque alors que les ensembles entrant et sortant sont égaux : on a détecté un graphe *garbage*.

Cependant, nous nous sommes rendu compte que sur le graphe suivant, notre algorithme ne suffisait pas :

## Garbage cyclique en forme de 8



Dès qu'un noeud a plus d'une sortie (c'est-à-dire, dès qu'un huit apparaît dans notre cycle), notre algorithme est inefficace.

On a donc eu l'idée de *scinder* les différentes branches de notre graphe. C'est-à-dire que dès que l'on rencontre une bifurcation, nous créons un *Graph Summary* pour chacune des branches. Dès qu'une branche a été parcouru (c'est-à-dire lorsque nous rencontrons un noeud déjà vu), la détection se met en attente (Stand by) pour cette partie. Le *Graph Summary* "intermédiaire" (c'est-à-dire concernant seulement la branche) est rapatrié au niveau de l'embranchement.

Dès que tous les *Graph Summary* intermédiaires sont rapatriés, on fait la fusion de tous ces graph pour obtenir le *Graph Summary* global.

On teste finalement sur le graph summmary global l'égalité des ensembles entrant et sortant. Sur l'exemple précédent, on applique notre algorithme :

Ensemble sortant	Ensemble sortant.
D	A
DA	ABH

Trace de l'algorithme dans les process 1 et 2

Au niveau du process 2, nous avons une bifurcation donc l'apparition de deux *Graph Summary* intermédiaires.

Ensemble sortant	Ensemble sortant.
DAH	ABHC
DAHC	ABHCD

*Stand by*

Trace de l'algorithme dans une branche

Ensemble sortant	Ensemble sortant.
DAB	ABHD

*Stand by*

Trace de l'algorithme dans l'autre branche

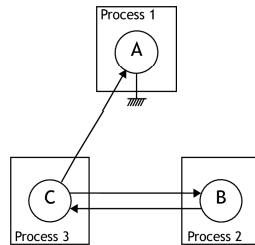
Dès que l'on retrouve le noeud E, les *Graph Summaries* intermédiaires se mettent en *Stand by* et sont rapatriés au niveau de l'embranchement (au process 2). On réalise ensuite la fusion.

Ensemble sortant	Ensemble sortant.
DABHC	ABHCD

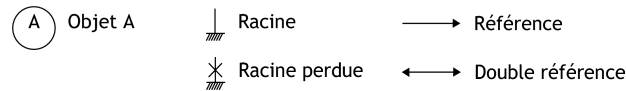
Trace finale de l'algorithme

On voit bien que les ensembles entrant et sortant sont égaux donc que l'on a bien détecté le graphe comme garbage.

Malheureusement, sur l'exemple suivant apparaît un autre problème :



Graphe hybride en triangle avec racine



En effet, lorsqu'il y a bifurcation, on peut se demander si on doit faire les tests d'égalité intermédiaires ou non. Exécutons notre nouvel algorithme sur cet exemple en prenant B comme candidat :

Ensemble sortant	Ensemble sortant.
C	B A

On a une bifurcation au process 3 :

Ensemble sortant	Ensemble sortant.
CB	BC

et

Ensemble sortant	Ensemble sortant.
CA	A

Racine !

On fusionne les branches et on obtient :

Ensemble sortant	Ensemble sortant.
CB	BCA

On considère donc tout le graphe comme non garbage et les noeuds B et C (qui sont garbage) ne sont pas éliminés. Notre solution n'est donc pas complète.

Ce problème vient du fait que nous mélangeons des graphes cycliques et acycliques. On pourrait espérer que la détection cyclique traite ce cas mais il faudrait alors ne pas considérer le cycle comme tel, ce qui est impossible. Il fallait donc changer l'algorithme. La solution apparaît

sur l'exemple suivant : il suffit d'effectuer le test d'égalité entre les ensembles entrant et sortant au niveau du *Graph Summary* intermédiaire. En effet, on a alors bien l'égalité pour le *Graph Summary* intermédiaire de la branche CB et donc une destruction de cette bifurcation.

Le pseudo code est alors :

```

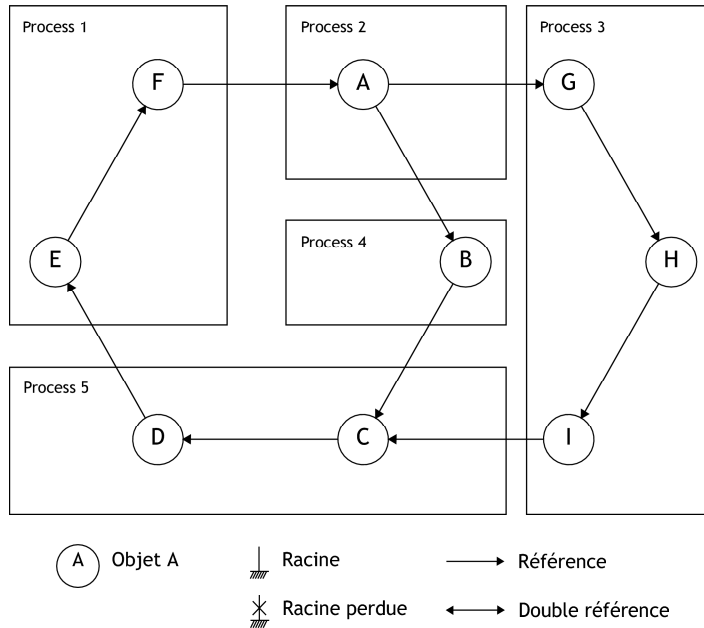
Data: un candidat est soupçonné d'être garbage et le Graph Summary est initialement
vide
if le graphe local est enraciné localement then
| le graphe n'est pas garbage
else
| if l'ensemble sortant du Graph Summary local est vide then
| intégration du Graph Summary local dans le Graph Summary global
| if l'ensemble entrant et sortant du Graph Summary global sont égaux then
| | le graphe est garbage
| else
| | le graphe n'est pas garbage
| end
| else
| foreach élément s de l'ensemble sortant du Graph Summary local do
| | if s appartient à l'ensemble entrant du Graph Summary global then
| | | stand by
| | else
| | | fusion du Graph Summary intermédiaire partant de s avec le Graph
| | | Summary global
| | | if les ensembles entrant et sortant du Graph Summary intermédiaire sont
| | | égaux then
| | | | la branche est garbage
| | | else
| | | | on lance l'algorithme avec s pour candidat et on appelle le résultat Bs
| | | end
| | end
| end
| fusion de tous les Bs en stand by
| if les ensembles entrant et sortant sont égaux then
| | le graphe est garbage
| else
| | le graphe n'est pas garbage
| end
| end
end
end
Result: Est-ce que le candidat est garbage?

```

**Algorithm 2:** Pseudo-code efficace

Un autre problème s'est posé à nous. Il est présenté sur l'exemple suivant :

## Garbage cyclique en forme de 8



On voit sur cet exemple le problème d'avoir un noeud comme objet entrant pour deux branches distinctes (ici par exemple l'objet C). En effet, on voit alors que la branche principale (EFABCD) est considérée comme garbage, malgré toute racine sur la branche secondaire. Le premier réflexe a été de vouloir changer l'algorithme. Nous avons trouvé la solution mathématique suivante à la place de la précédente boucle *for* :

```

foreach élément s de l'ensemble sortant du Graph Summary local do
  if s appartient à l'ensemble entrant du Graph Summary global then
    stand by
  else
    fusion du Graph Summary intermédiaire partant de s avec le Graph Summary
    global on lance l'algorithme avec s
  end
end
if on a une racine then
  le graphe n'est pas garbage
else
  il est garbage
end
  
```

**Algorithm 3:** Pseudo-code efficace corrigé

Cette solution suppose que même en tombant sur un objet contenant l'information *root*, on puisse continuer la détection jusqu'à la position de *stand by*. Cela n'est malheureusement pas possible. En effet, un objet *root* ne nous permet d'accéder aux objets qu'il pointe. Cette solution n'est donc pas réalisable.

Une autre solution était de forcer les ensembles entrant et sortant du *Graph Summary* intermédiaire du cas précédent à ne pas être égaux. Pour cela, il suffit d'accéder aux process qui rendent l'objet C entrant. C'est-à-dire, dans notre exemple, que l'on n'aura pas que deux objets

entrants pour le process 4 mais trois ! Il y aura B bien sûr, mais également le C provenant du process 3 et le C provenant du process 2, ce qui fait non plus deux objets entrants mais bien trois. En réalité, notre avant-dernier algorithme convient car un objet entrant pour deux process différents aura deux ambassadeurs différents. Il sera donc considéré naturellement comme un objet *double*.

Ainsi sur l'exemple précédent, on obtient les tableaux suivants avec E comme candidat :

Ensemble sortant	Ensemble sortant.
$E_5$	$A_1$
$E_5 A_1$	$A_1 B_2 G_2$

puis par le process 4

Ensemble sortant	Ensemble sortant.
$E_5 A_1 B_2$	$A_1 B_2 C_4$
$E_5 A_1 B_2 C_4 C_3$	$A_1 B_2 C_4 E_5$

et par le process 3

Ensemble sortant	Ensemble sortant.
$E_5 A_1 G_2$	$A_1 G_2 C_3$

et enfin

Ensemble sortant	Ensemble sortant.
$E_5 A_1 B_2 C_4 C_3 G_2$	$A_1 B_2 C_4 E_5 G_2 C_3$

On a bien les 2 ensembles égaux à la fin !

Il est bien évident que les tests n'ont pas seulement été effectués sur les exemples présentés précédemment mais aussi sur de nombreux autres tels que les graphes en spirale, les graphes locaux et distribués, les graphes qui ont une référence à une racine avec des éléments déchets,... Les tests étaient purement algorithmiques. Aucun domaine n'avait pu être implémenté jusque là, nous avons besoin de savoir si l'algèbre du *Graph Summary* fonctionnait véritablement ou non. Nous avons pensé que c'était le meilleur moyen de comprendre précisément l'algorithme et les raisons de son fonctionnement. En fait, ces tests nous donnent aussi une idée des chiffres impliqués.

Le nom *Graph Summary* est très bien choisi. Par exemple, un graphe dans un process peut avoir des milliers d'objets dans 10 ordinateurs mais seulement 20 objets entrant ou sortant. Ainsi, le facteur de compression peut être très élevé ce qui préserve les communications à distance entre les domaines.

Malheureusement, dans un environnement non distribué, le  $\lambda$ -calcul peut aider à démontrer des algorithmes mais un environnement concurrent ne possède pas un tel outil de démonstration. Bien sûr, cette technique n'est pas adaptée à notre environnement à cause des *Race conditions*. Malgré tout, nous avons choisi de faire confiance aux tests et d'utiliser l'algorithme.

## 2 Intégration de l'algorithme dans le projet NGrid.

Afin d'utiliser le *Graph Summarizer*, nous avons eu à faire deux modifications :

- Traduire des notions bas niveau en des notions haut niveau
- Définir une technique pour lire un graphe de références.

L'algorithme ne fonctionne pas avec des objets mais seulement avec des identifiants car nous ne pouvons pas nous permettre de détruire des objets tenus par une référence...

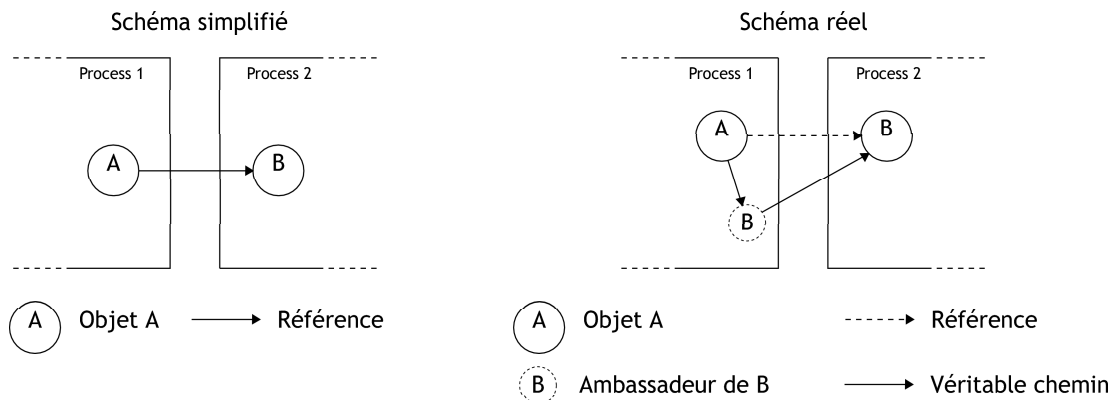
## 2.1 Traduction des notions bas niveau en notions haut niveau

L'article [2] possède une approche bas niveau *cls* pour présenter l'algorithme. En fait, NGrid est un environnement haut niveau. C'est pour cela que des notions bas niveau telles que les scions (skeletons pour Java RMI) et les stubs ne sont pas pertinentes dans un environnement haut niveau. Nous avons donc décidé de les remplacer par des notions d'objets entrant et sortant (du point de vue du domaine courant).

En résumé, parmi l'ensemble des objets, il y a d'abord les objets banals qui n'ont pas de contact avec les process extérieurs, puis les objets sortants du process et enfin les objets entrant dans le process.

- Un objet sortant n'est pas dans le domaine courant. Seul un ambassadeur (ou *proxy*) se trouve sur le processus courant. Il nous dit simplement où est le véritable objet c'est pour cela qu'un objet sortant est appelé substitut dans la littérature du *Reference Listing* [1].

### La notion d'ambassadeur



Ainsi, comme sur l'illustration, l'arc de A vers B n'est pas direct puisqu'il y a l'ambassadeur B. Ce niveau de précision sera dorénavant ignoré sur les schémas.

- Un objet entrant est un objet présent dans le domaine courant qui est référencé par un autre objet dans un autre domaine. Dans la littérature du *Reference Listing*, c'est un objet domicilié qui possède un *dirty set* avec au moins deux domaines.

Par définition, le *dirty set* d'un objet est l'ensemble des identifiants des processus où est représenté l'objet par un ambassadeur. Sur l'exemple, l'objet B du process 2 a un *dirty set* qui contient l'identifiant du process 1 tandis que le *dirty set* de l'objet A dans le process 1 est vide.

## 2.2 La technique d'*Inhumation-Sérialisation*

Veiga et Ferreira n'expliquent pas comment accéder au graphe de références d'un domaine. C'est pour cela que nous avons implémenté la méthode de l'*inhumation*. Elle est basée sur la fabrication en série simplement utilisée par le cadre .Net pour une communication à distance. L'idée consiste à lire le résultat de la sérialisation car elle contient nécessairement toutes les données des références dont nous avons besoin, c'est-à-dire les arcs entre les noeuds objets.

Cependant, la sérialisation est coûteuse et de plus elle ne peut pas être réalisée sur des objets actifs, racines. La solution est d'ignorer les références des objets actifs car elles ne concernent pas *a fortiori* les déchets. De plus, nous devons éviter de déranger les processus des domaines en bloquant tout pour la sérialisation. Nous suggérons de sérialiser les objets en les bloquant un

par un tandis que l'on retarde les appels aux objets en sérialisation tant que cette dernière n'est pas terminée.

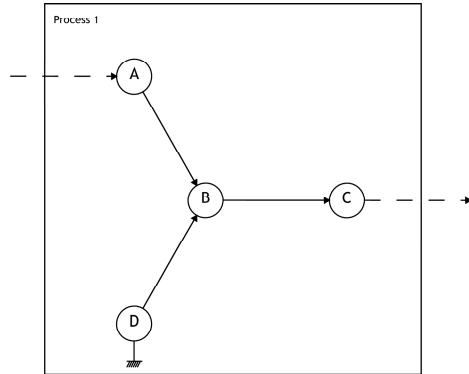
La définition d'un objet avec racine est jusqu'ici un peu floue. Dans un domaine, un objet est enraciné s'il est lui-même une racine ou s'il est directement référencé par une racine ou s'il est indirectement localement enraciné au process. Notre idée est de simplifier la notion d'objet avec racine en considérant tous les objets référencés directement ou non par des racines comme de vraies racines.

L'idée principale est de transformer et de simplifier le graphe des références. Du point de vue du ramasse-miettes, le graphe simplifié est équivalent au premier. Voici un exemple qui explique la technique d'inhumation :

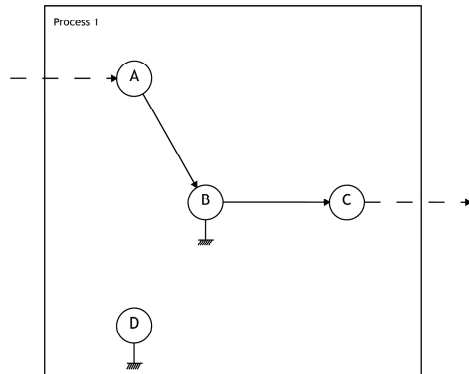
## La technique de l'inhumation

Nous pensons que ces 3 graphes sont équivalents en terme de détection de garbage

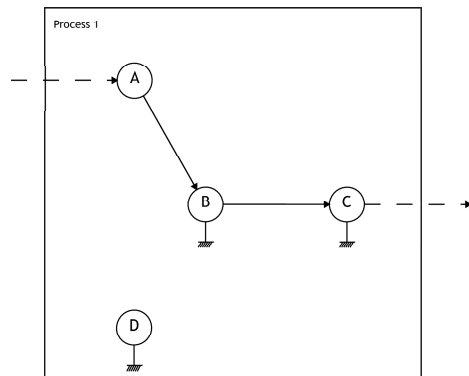
Le graphe du process tel qu'il est



Ce qu'on peut en voir



Ce qui est finalement interprété



### 3 Optimisations et améliorations de l’algorithme de Veiga et Feirrerà grâce à l’environnement NGrid

Voici quelques optimisations que nous avons décidé d’implémenter :

- Le tampon temporel inspiré des estampilles des systèmes de gestion de base de données afin de résoudre le problème des déplacements et modifications des objets pendant la détection.
- Le regroupement des détections en initiative afin d’économiser les calculs.
- Le stockage des données de l’initiative dans chaque domaine afin d’économiser les communications réseau.
- La stratégie *OneWay* afin de réduire le nombre de domaines impliqués en même temps dans le processus de détection cyclique.

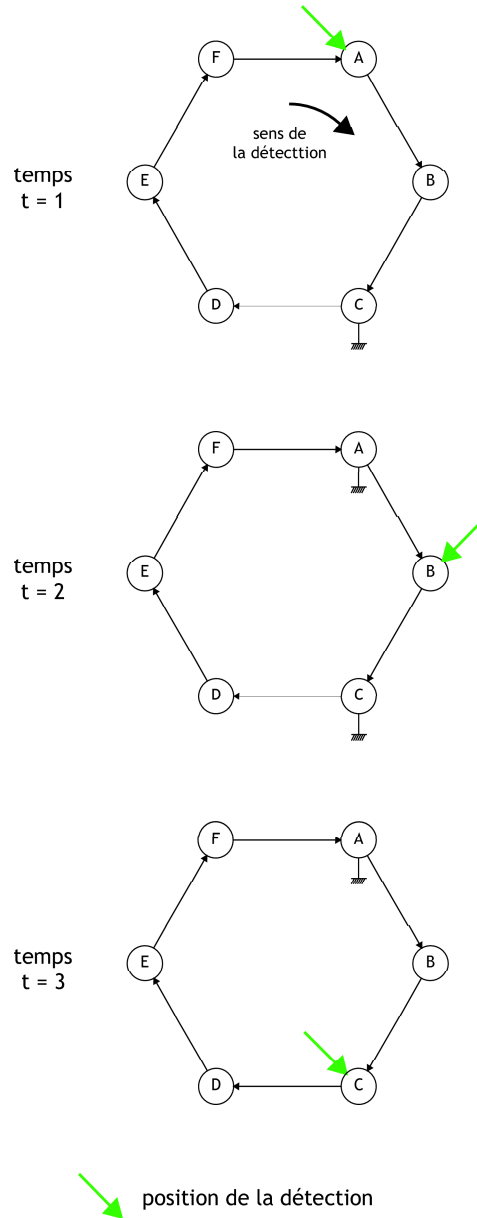
#### 3.1 La stratégie du tampon temporel

Nous avons montré la difficulté de se procurer le graphe des références du domaine courant. De plus, des objets peuvent être modifiés après le processus de sérialisation et avant la fin de la détection. Si c’est le cas, ces objets ont été changés via leurs références, ils sont donc directement ou indirectement liés à une racine et bien sûr non garbage.

Afin de se procurer cette information cruciale, nous utilisons une stratégie de tampon temporel : tout accès à un objet incrémente son compteur. Pendant la sérialisation du graphe de références du domaine, une photographie des tampons temporels est faite. À la fin de la détection, si le détecteur affirme que le graphe est un déchet nous devons alors vérifier si les tampons temporels ont changés. Si c’est le cas, alors le graphe n’est pas un déchet ! Si ce n’est pas le cas, alors le détecteur a raison.

Voici un exemple qui montre à quel point la stratégie du tampon temporel est cruciale :

Exemple qui montre à quel point  
les tampons temporels sont cruciaux



Dans cet exemple, sans information de la part du tampon temporel, l'algorithme détruirait un graphe non-déchet car la détection n'a rencontré aucune racine. La stratégie du tampon temporel repose sur un simple fait : si on accède à un objet alors celui-ci n'est pas un déchet puisque l'accès se fait forcément via une racine.

### 3.2 L'initiative ou stratégie de regroupement

L'article explique l'algorithme du *Graph Summarizer* pour un candidat mais on peut tenter de rentabiliser le plus possible le lourd processus de sérialisation. Notre idée est de lancer une détection de déchet avec de nombreux candidats du même domaine. Cette idée se justifie par le fait que les traitements des *Graph Summaries* ont un coût négligeable par rapport à la sérialisation donc autant utiliser le plus possible celle-ci avec plusieurs détections.

Les différents états de détection sont enregistrés dans un plus gros processus appelé *initiative*.

Si une détection se termine sans pouvoir conclure, elle doit être fusionnée avec une autre détection de la même *initiative* qui a un élément connexe.

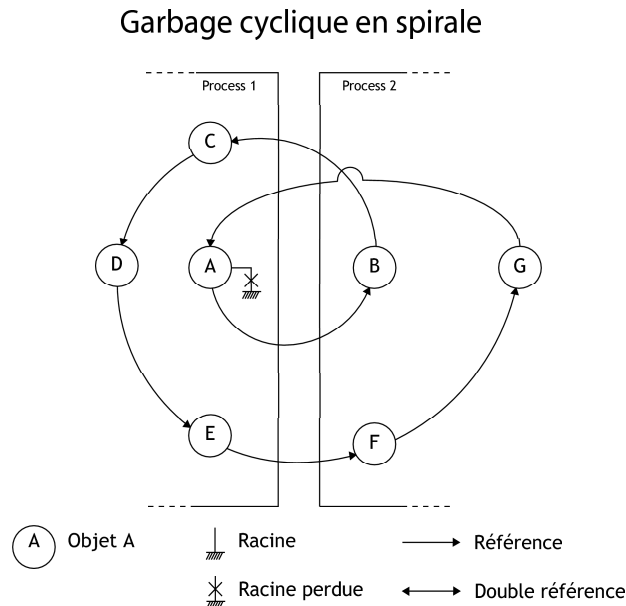
En fait, l'ensemble analysé par une détection peut être énorme mais la réunion des ensembles entrant et sortant est suffisante! Par définition, vérifier si deux *Graph Summaries* peuvent fusionner ne coûte pas beaucoup de calculs car les ensembles sont petits.

### 3.3 Le stockage des nombreuses données de l'initiative

Malheureusement, afin de sauvegarder les communications à distance entre les domaines, nous devons stocker les nombreuses données de tous les domaines visités telles que les tampons temporels.

De plus, comme les *Graph Summaries* ne sont coûteux ni en temps ni en espace mémoire en comparaison avec la sérialisation, nous calculons tous les *Graph Summaries* même s'ils ne sont pas nécessaires. En fait, comme une initiative peut rencontrer un domaine deux fois, nous ne savons pas si un *Graph Summary* sera utile ou non. Mais afin d'éviter des graphes incohérents dus à des photos de domaines à des instants différents, nous sélectionnons seulement un des vieux *Graph Summaries* locaux, demandés et déjà calculés.

Sur cet exemple, en commençant par le candidat A, on comprend qu'en quittant et revenant sur le même process (ici 1), il est dommage de resérialiser.



Cependant, toutes ces données sont liées à une initiative, et quand cela est terminé, une dernière méthode réseau peut détruire toutes les données de l'initiative dans chaque domaine visité.

Enfin, nous devons éviter les boucles infinies dues à des photographies incohérentes du fait des *Race conditions* où un substitut sortant pointe vers un substitut sortant du même objet dans un autre domaine. La solution est d'arrêter un processus de détection s'il demande deux fois le même *Graph Summary* local ce qui est facile quand les *Graph Summaries* locaux sont stockés dans des domaines.

### 3.4 La stratégie OneWay

Enfin, voici la dernière amélioration de notre travail basée sur le *Graph Summarizer* : transformer l'algorithme en un autre non récursif sans *boucle for*. Lors du parcours du graphe de références d'un objet candidat, l'algorithme peut rencontrer des croisements entre des domaines.

La première idée naturelle est d'analyser les domaines dans une *boucle for*. Mais cela coûterait trop de communications à distance. En effet, lors d'une bifurcation, un domaine devra attendre un autre jusqu'à ce que le précédent domaine visité retourne une réponse. Cela crée ainsi un arbre de domaines d'attente. Attendre que chaque méthode se termine entraîne un *manque à gagner* en termes de performance, surtout en sachant que les appels eux-mêmes sont indépendants et n'ont donc pas besoin de s'attendre.

Notre idée est de fabriquer la *boucle for* grâce à l'environnement .Net et plus précisément la balise *OneWay*. Cette fonctionnalité autorise des appels asynchrones sans attendre le résultat. Le seul problème est de garantir l'indépendance de ces appels.

En fait, nous pouvons montrer que les branches d'un graphe connexe peuvent être analysées dans tous les sens. Dans un croisement, deux processus de détection peuvent vouloir analyser des domaines différents. En particulier quand les graphes ne sont pas facilement connexes, c'est-à-dire quand le contact se fait en dehors du process courant.

On réalise un vote entre les *Graph Summaries* de la même initiative pour déterminer quel va être le prochain domaine visité. Une initiative est terminée lorsqu'il n'y a plus de domaine à visiter.

Quand le vote n'est pas en accord avec un des processus de détection, il ne se passe rien pour cette détection mais heureusement, les *Graph Summaries* sont petits, ce qui sauvegarde les communications à distance.

Utiliser ce dispositif puissant du *OneWay* signifie qu'il y a un maximum de deux domaines qui travaillent en même temps dans un processus de détection d'une initiative.

L'intégration de ces améliorations a été implémentée avec succès dans le projet NGrid. Les codes sont disponibles à : <http://sourceforge.net/projects/ngrid>.

## Conclusion

L'algorithme du *Graph Summarizer* est une solution qui suit une approche hybride : un collecteur acyclique distribué basé sur l'algorithme du *Reference Listing* [1] et un détecteur cyclique de déchets qui complète le premier fournissant ainsi une solution complète au problème de la collection de déchets distribués.

Nous avons géré l'implémentation des algorithmes avec quelques améliorations dues à l'environnement haut-niveau : NGrid. Désormais, les heuristiques publiées quant à la prédiction de la durée de vie des objets devraient résoudre le problème du soupçon du bon candidat avant le lancement du grand algorithme présenté ou peut-être le bon domaine ...

Les principales contributions de notre travail sur le problème de la collection de déchets sont :

- L'implémentation de l'algorithme du *Graph Summarizer* pour détecter les déchets cycliques.
- Une version *OneWay* de l'algorithme du *Graph Summarizer*.

Aujourd'hui, notre implémentation d'un ramasse-miettes cyclique distribué est lancée par l'utilisateur avec tous les objets du domaine courant comme candidats. Un module prédicteur de durée de vie pourrait être intégré pour lancer automatiquement notre algorithme.

Ce projet nous a permis de maîtriser plusieurs nouveaux outils tels que C#, CVS et  $\LaTeX$ , de s'immerger dans la compréhension d'un article scientifique récent [2] (2004) et enfin de vivre l'expérience originale qui consiste à participer à un vaste projet informatique libre NGrid [3].

## Références

- [1] Greg NELSON Susan OWICKI Andrew BIRELL, David EVERS and Edward WOBBER. Distributed garbage collection for network objects. Technical Report 116, digital - Systems Research Center, Palo Alto, California, United States of America, December 1993.
- [2] Paulo FEIRRERA Luis VEIGA. Asynchronous, complete distributed garbage collection. Technical Report RT/11/2004, INESC-ID/IST, Lisboa, Portugal, June 2004.
- [3] Joannès VERMOREL. Ngrid is an open source (lgpl) grid computing framework written in c#. <http://ngrid.sourceforge.net/>.
- [4] Paul R. WILSON. *Uniprocessor garbage collection techniques*. Springer-Verlag.s, Saint-Malo, France, 1992.