Abstract

Different manner can be used to guarantee memory space collection in functional language and even in O.O languages. Garbage collection (the process of reclaiming storage that has been allocated by a client program, but is no longer accessible by following pointers from program variables) can be done by various algorithms. Distributed systems, which are characterized by a growing number of applications that communicate solely by exchanging messages, require new concepts which must be suitable for distributed environment and should take into account all the characteristics of theses environments. Our paper discuss two main ideas : first the implementation and the evaluation of the "Mark then Sweep" garbage collection technique in a centralized system, then we discuss basic ideas of the implementation of the garbage module in a distributed environment. We introduce new manner to generalize the garbage collection in a system constituted by more than one computation unity. We propose a protocol (based on the atomic commit protocol called : 2PC) which guarantees the distributed memory collection, and we give also, a solution to avoid the protocol blocking if some system failures occur.

Key-words : Garbage collector, functional language, distributed system.

Programmation fonctionnelle : Le glaneur des cellules centralisé et sa généralisation dans les environnements répartis

Tayeb LEMLOUMA [†] Abdelmadjid BOUDINA [*]

[†] *Opera Project, INRIA Rhône-Alpes, Zirst - 655 avenue de l'Europe, 38330 Montbonnot-St-Martin*
Phone: (+33) 4 76 61 52 81 Fax: (+33) 4 76 61 52 07

[†] *Institut d'informatique, U.S.T.H.B, B.P .32 El-Alia Bab Ezzouar, Alger 16111, ALGERIE.*
Tel / fax 213 2 24 76 07

Résumé

Différentes manières peuvent être utilisées pour assurer la récupération de l'espace mémoire dans les langages fonctionnels et mêmes dans les langages orientés objets. Depuis la naissance du premier langage basé sur la manipulation des listes, la fin des années cinquante et le début des années soixante, plusieurs algorithmes ont été proposés pour répondre à cet objectif d'une manière simple et efficace. L'apparition des systèmes répartis, qui est due essentiellement au développement technologique et aux progrès des techniques de transmission de données, nous pousse à voir comment l'opération de récupération, peut être mise en œuvre dans un environnement ou le seul moyen de communication est l'envoi et la réception de messages. Ce rapport discute deux aspects : l'implémentation et l'évaluation de l'opération de ramasse miettes, en se basant sur la technique "Mark then sweep", et cela dans un système centralisé. Le deuxième aspect discuté, est l'implémentation du glaneur dans un environnement réparti. Nous introduisons une manière pour généraliser la collection mémoire centralisé dans un système constitué de plus d'une unité de calcul. Assurer une certaine tâche dans un environnement réparti, revient à assurer un protocole qui permet à l'ensemble des unités participantes d'atteindre l'objectif voulu. Pour cela nous proposons un protocole, inspiré du protocole de la validation atomique 2PC, qui garantie l'opération de ramasse miettes distribuée. Nous donnons également une solution pour éviter le blocage du protocole qui peut survenir à cause des éventuelles pannes du système distribué.

Mots clés

Ramasse miettes, garbage collector, langages fonctionnels, système distribué,

1 Introduction

La mémoire est une ressource de stockage de données et de programmes. Elle est subdivisée en une suite finie de composants appelés emplacements [BEL 96]. Dans les langages fonctionnels, une zone mémoire de taille limitée est utilisée, dont les différents emplacements ou "cellules" libres forment ce qu'on appel par : "la liste libre" (free list).

Le module connu par le " garbage collector" (GC) ou le "glaneur de cellules", ou encore le "ramasse miettes", est un module responsable de la récupération de l'espace mémoire qui n'est pas utilisé. Le module de récupération, joue un rôle principal durant la vie d'un programme, les résultats expérimentaux faits sur un grand nombre de programmes LISP, ont montré que l'activité de récupération prend un pourcentage de dix à trente pour cent du temps d'exécution global [STE 75, WAD 76].

Les techniques de récupération peuvent être utilisées par tout système ou langage basé sur l'allocation dynamique, même s'il n'est pas de nature fonctionnelle, tel que Smaltalk [GOL 83] ou C/C++ [URL 3, URL 4]. La tâche du glaneur, dans les langages basés sur le concept d'objet par exemple, consiste à identifier les objets qui ne sont pas en cours d'utilisation et récupère l'espace qu'ils occupent. Un objet qui n'est pas en cours d'utilisation, c'est un objet qui n'est pas accessible par le programme dans son état actuel [URL 1].

Depuis la naissance du premier langage fonctionnel (LISP) basé sur la manipulation des listes, à la fin des années cinquante et le début des années soixante [MAC 62], plusieurs algorithmes ont été proposés pour répondre à cet objectif d'une manière simple et efficace, dans ce cadre, plusieurs travaux ont été publiés depuis 1973 [COH 81]. Dans les environnements répartis peut de travaux, à notre connaissance, ont été publiés. Dans [LES 90], un algorithme distribué de ramasse miettes est introduit pour les machines parallèles. Cet algorithme utilise les deux techniques de collection : celle basée sur "la recopie" [COH 81] et celle basée sur les "compteurs de références" [PEY 90], l'article est axé essentiellement pour les machines à plusieurs processeurs.

Dans ce rapport, nous discutons deux aspects : l'implémentation et l'évaluation de l'opération de ramasse miettes en se basant sur la technique "Mark-sweep", et cela dans un système centralisé. Le deuxième aspect discuté, est l'implémentation du glaneur dans un environnement réparti. Nous introduisons une manière pour généraliser la collection mémoire centralisé dans un système constitué de plus d'une unité de calcul, et nous montrons les primitives utilisées par le protocole de récupération, qui doivent être utilisées pour atteindre l'objectif de récupération distribuée.

Ce rapport est divisé en cinq sections. Dans la deuxième section on introduit la classe des langages fonctionnels, ses concepts de base, ses principaux avantages et la notion d'évaluation des programmes basés sur les manipulations de listes. Le concept de "Garbage Collector" ou récupérateur est en suite discuté dans la section 3. Dans cette section nous parlons des différentes techniques de récupération mémoire qui existe dans la littérature en se basant sur la technique Marquage-Balayage. Nous généralisons la récupération centralisée, dans un environnement réparti constitué de plus d'une unité de calcul, et nous proposons notre propre vision de la gestion de

mémoire et du protocole de communication utilisé dans de tel environnement. La quatrième section met en évidence notre mise en œuvre de l'application de "Garbage Collector" dans les deux environnements : centralisé et distribué. Nous commençons par la description de notre application DGC et les primitives qu'elle assure, en suite nous donnons les principales structures de données ainsi que le protocole, utilisées dans l'implémentation. A la fin de cette section nous discutons l'évaluation et les principaux résultats obtenus.

2 Les langages fonctionnels

La principale caractéristique des langages fonctionnels, est la notion de "fonctions". En effet dans ce type de langage de programmation, la fonction est l'objet par excellence, en particulier les fonctions peuvent être des données ou des résultats d'autres fonctions ou programmes. Une autre caractéristique, est que ces langages reposent sur des théories mathématiques, comme le λ-calcul ou la logique combinatoire, qui constituent un formalisme pour ces langages et fournissent des outils puissants pour leur implantation.

Dans la programmation fonctionnelle, les fonctions peuvent être utilisées comme argument d'autre fonction. On peut aussi faire des combinaisons de fonctions, à l'aide de formes fonctionnelles pour produire de nouvelles fonctions [GLA 84].

2.1 Concepts de base et avantages de la programmation fonctionnelle

Un programme écrit dans le style fonctionnel, est considéré comme une fonction de calcule mathématique, qui décrit une relation entre les données en entrée et les résultats en sortie. Elle est basée sur une programmation applicative.

La programmation fonctionnelle a plusieurs avantages. Cela est dû aux différents concepts de base qui la caractérisent [SUP, KAD 92] :

1- *Existence de fondement théorique* : Le style fonctionnel repose sur des théories mathématiques comme le λ-calcul (LISP [MAC 62]), la logique combinatoire (SASL [TUR 76]) ou des théories équivalentes.

2- *Absence d'effet de bord* : Les langages fonctionnels ne manipulent pas des variables au sens impératif. Les seules variables utilisées représentent les arguments de fonctions et elles sont à assignation unique, i.e. elle ne change pas de valeurs.

3- *Transparence référentielle* : Cette caractéristique est liée à l'absence de l'effet de bord. Elle exprime le fait de pouvoir simplifier les expressions en remplaçant les sous-expressions communes par leur résultat.

4- *Indépendance dans l'ordre d'évaluation* : Il existe plusieurs manières pour évaluer une expression ou fonction, selon l'ordre d'évaluation.

5- *Principe d'évaluation* : L'interpréteur d'un langage fonctionnel est constitué de trois modules principaux : un module de lecture, un module d'évaluation et un module d'écriture. Ces trois modules sont appelés "Read-Eval-Print" [MAC 90].

2.2 L'évaluation dans les langages fonctionnelle

Les données des langages fonctionnels sont représentées sous forme d'expressions, qu'on peut les exprimer à l'aide de la théorie lambda-calcul développée par A.CHURCH [CHU 41]. L'évaluation d'une expression, consiste à la transformer en *forme normale* [PEY 90], i.e. une expression qui ne peut pas être réduite [SUP].
Considérons, par exemple la fonction *g* définie par :

$$g\,(x) = (\,x - 1\,) * (\,x + 2\,) * (\,x - 5\,)$$

Où "-", "+" et "*" sont les opérateurs définis sur ∠. Pour évaluer la fonction *g* à l'argument *n,* nous pouvons procéder de la manière suivante :
Evaluer (x-1) puis évaluer (x+2). Faire la multiplication des valeurs et multiplier le résultat par l'évaluation de (x-5). Ainsi, on peut donner le schéma d'évaluation suivant :

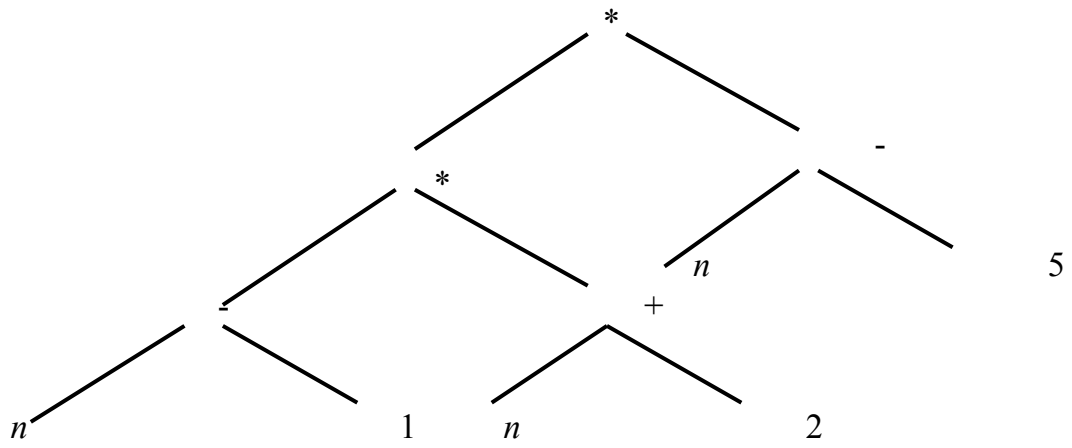

Figure 1. L'arbre d'évaluation de *g*(*n*)

Pour que l'intérpréteur des langages fonctionnels, puisse exécuter le module d'évaluation et l'appliquer aux expressions, ces dernières doivent être stockées quelque part. Qui dit stockage, dit gestion de l'espace qu'elles occupent. La gestion de l'espace occupé, inclus la représentation des données et leur manipulation suivant cette représentation.

Dans les langages fonctionnels, unes expression peut être représentée par un arbre, qu'on appel "*l'arbre abstrait*"; l'arbre est constitué d'une racine et un ensemble fini de nœuds. Lors de l'évaluation, l'arbre subit des transformations afin de calculer le résultat final [PEY 90]. De point de vue implémentation, les différents nœuds de l'arbre abstrait sont représentés par des "*cellules*". Une cellule n'est qu'un emplacement mémoire qui stocke une ou plusieurs données. La taille des cellules peut être constante

pour tous les nœuds, comme elle peut varier selon le type du nœud, et la même chose pour le nombre de champs dans les cellules. Généralement, une cellule doit contenir au moins deux champs : un champ de donnée et un autre pour l'adressage. Dans la plupart des cas, le champ de donnée contient une valeur atomique (un symbole d'opérateur, une valeur constante, … etc.). Le champ d'adressage, contient un pointeur vers la cellule suivante. Le pointeur n'est autre qu'une adresse qui permet de passer d'une cellule courante à une autre cellule. Notons que durant l'évaluation l'arbre de l'expression peut être transformé en un "graphe" [PEY 90].

Considérons la fonction suivante, $f(x) = (x - 1) * (x + 2)$. l'arbre d'évaluation de la fonction f appliquée à l'argument n est le suivant :

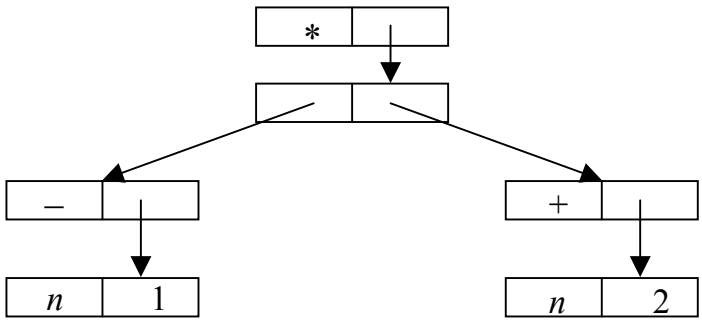

Figure 2. L'arbre de l'expression *g*(*n*)

Dans cette représentation, on remarque que chaque cellule à deux champs: <donnée, adresse> ou <adresse, adresse>. Si la taille d'un pointeur et d'une donnée atomique est la même, on peut conclure que la taille de la cellule est constante.

Comme le principe de l'optimisation mémoire est de ne sauvegarder que les données utilisées, les cellules qui ne seront pas utilisées lors de l'évaluation ne doivent être stockées, afin d'éviter toute occupation inutile de l'espace mémoire.

Pour gérer l'espace libre de la zone mémoire alloué initialement au programme (lors de l'interprétation), les langages fonctionnels se basent sur la notion de liste. Une cellule disponible ou libre, fait partie d'une liste appelée "*liste libre*" (free liste). La liste libre est une liste linéaire chaînée, qui enchaîne toutes les cellules disponibles au programme à interpréter. Comme toute liste, la liste libre est accessible à partir de l'adresse du premier élément, ou de la tête. Le chaînage existé entre les différentes cellules, permet de parcourir la totalité des éléments de la liste. Une cellule libre contient un champ d'adresse, l'adresse de la cellule suivante dans la liste libre, et un champ de donnée qui est inutilisé, ce dernier sera utilisé dans le cas où la cellule serait allouée pour une éventuelle évaluation .

Les opérations d'*allocation* et de *libération* de cellules, sont exécutées de la même manière que les primitives "const" et "tail" appliqués sur les listes [SUP, MAC 62].

L'allocation d'une cellule à comme effet l'extraction de cette cellule de liste libre. L'extraction se fait toujours de la tête de liste, par conséquent la liste libre est privée de

son élément de tête après l'extraction, c'est comme si on avait appliqué un " tail (free list) " ; sauf qu'ici, l'élément extrait est utilisé.

La libération d'une cellule lors d'une exécution, implique que la cellule libérée n'est plus utilisée par le programme. Dans ce cas la cellule doit être insérée dans la liste libre. L'opération d'insertion est réalisée d'une manière inverse à l'opération d'allocation, la cellule est insérée au début de la liste analogiquement à la primitive de construction de liste "cons".

Les requêtes de demandes d'espace mémoire sont satisfaites tant que la liste libre n'est pas vide, et cela en allouant à chaque fois de nouvelles cellules. Dans le cas ou la liste libre est vide, le système essaie de récupérer l'espace mémoire disponible, en exécutant le module appelé "Ramasse miettes" ou "Garbage Collector".

3 Le "garbage collector"

Les cellules mémoires sont manipulées par le programme utilisateur à l'aide d'un programme superviseur appelé "Allocateur de mémoire". Quand le nombre de cellules disponibles est réduit. Il se peut qu'après un temps *t*, il n'y aura pas de cellules disponibles.

Des études expérimentales ont montré que dans des telles situations, certaines cellules déjà allouées ne seront pas utilisées, et par conséquent elles peuvent être restituées à l'allocateur. Une cellule devient non utilisée, ou *récupérable*, si elle n'est pas atteignante par les champs pointeurs d'aucune autre cellule accessible. C'est la tâche du GC ("Garbage Collector", "Récupérateur" ou "Ramasse miette"), de récupérer cet espace mémoire non utilisé [COH 81].

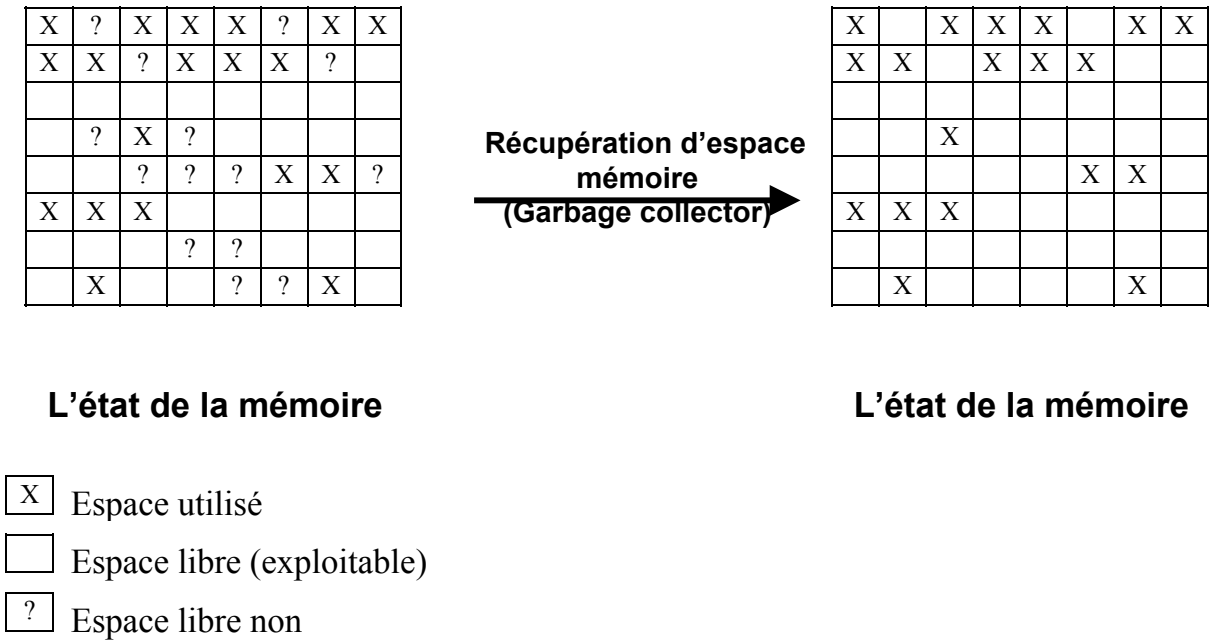

Figure 3. L'opération de récupération de la mémoire

Selon l'endroit de l'espace mémoire manipulé par le programme utilisateur, i.e. qu'il soit local ou réparti, la récupération peut avoir deux sens : le sens habituelle, où tout est centralisé (Allocateur, "Garbage Collector"… etc.) et le sens distribué où de nouvelles primitives doit être implémentées. Dan ce qui suit, nous discutons ces deux aspects.

3.1 Les techniques du "Garbage Collector"

Plusieurs méthodes de récupération existent dans la littérature. Parmi ces méthodes, figurent les techniques : marquage-balayage ("Mark-Sweep"), recopie ("Copying") et compteur de référence ("Reference Counting") [COH 81, PEY 90, URL 2].

Tout algorithme de récupération de mémoire peut être subdivisé en deux étapes :

1- Identification de l'espace à récupérer.
2- Incorporation de l'espace identifié dans la mémoire disponible pour l'utilisateur.

Ces deux étapes peuvent être réalisées par différentes manières : La technique de *recopie*, consiste à faire une recopie de l'espace accessible vers une partie de l'espace d'adressage appelée espace d'arrivé. La recopie est faite d'une manière contiguë, donc à la fin de cette opération, on aura deux blocs : un bloc alloué, et un block libre qui peut être exploité pour l'allocation. Dans le cas ou il n'y a plus de cellule qui peut être alloué, les rôles des deux espaces sont intervertis. L'inconvénient majeur de cette technique et qu'elle nécessite une taille importante d'espace mémoire.

La technique "compteur de référence" se base sur l'utilisation de compteurs pour effectuer la première étape de la récupération, i.e. l'identification de l'espace à récupérer. Un compteur de référence, est un champ de cellule qui contient le nombre de fois où cette cellule est référencée, i.e. le nombre de pointeurs qui pointent vers cette cellule. Une cellule qui n'est pas utilisée, est une cellule dont le compteur de référence est nul, dans ce cas la cellule peut être allouée.

Dans ce rapport nous nous sommes basés sur la technique la plus connue, qui est la technique marquage-balayage (ou "Merk-sweep") que ce soit dans le cas centralisé ou réparti. Cette technique consiste à exécuter deux phases : Dans la première phase, toutes les cellules accessibles de la mémoire sont marquées. Dans la deuxième phase, un balayage de la mémoire est fait afin de récupérer les cellules non marquées.

Dans cette technique, le "Garbage Collector" est composé de deux modules : Le module de marquage (marker), et le module de balayage (sweeper). Le module de marquage, a comme tâche d'identifier toutes les cellules accessibles. Les cellules peuvent être marquées en utilisant plusieurs méthodes, par exemple, à l'aide d'un bit (un champ dans la cellule) ou à l'aide d'une table qui contient les marques de toutes les cellules (cette table est appelée en général *bitmap*) [CHR].

Selon l'ordre et le lancement des deux phases de Marquage et Balayage par rapport à l'évaluation du programme utilisateur, plusieurs variantes du Marquage-Balayage,

peuvent être implémentées pour assurer la récupération. Dans [CHR], quatre techniques sont introduites :

1- *Marquage-Balayage traditionnel*.
2- *Marquage et Balayage par nécessité*.
3- *Marquage-Balayage parallèle*.
4- *Marquage durant le Balayage*.

La première technique, fait l'objet de notre travail. Dans cette technique le marquage et le balayage se font après une certaine durée d'évaluation par le programme utilisateur. La figure suivante illustre cela :

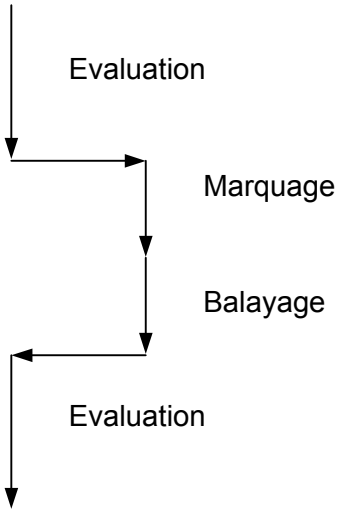

Figure 4. Le Marquage-Balayage Traditionnel

L'algorithme de cette technique peut être donné comme suit :

Procédure *Marquage*;
Début
  Parcourir la totalité des cellules réservées au programme utilisateur, et affecter au bit de marque la valeur zéro;
  Parcourir les cellules accessibles par le programme utilisateur, et affecter au bit de marque la valeur un;
Fin;
Procédure *Balayage*;
Début
  Récupérer les cellules marquées "zéro", en les réinsérant dans la liste libre (free list);
Fin;

3.2 Le "garbage collector" dans un environnement centralisé

Dans cet environnement, le " Garbage Collector " est souvent lancé automatiquement soit quand l'allocateur centralisé est exécuté dans une situation où il n'y a pas d'espace local disponible, ou avant cela d'une petite durée. Dans la plus part des cas, les langages évolués possèdent des primitives de manipulations de mémoire destinées à l'allocateur local. Le "Garbage Collector" peut être lancé quand l'une de ces primitives sera exécutée. Par exemple, dans LISP, la fonction "cons" appelle le "Garbage Collector". La fonction *cons* [SUP, MAC 62], permet la construction des listes, en rajoutant un élément à la tête de la liste. L'ajout d'un élément, implique l'allocation d'un espace mémoire. Par conséquent si l'allocateur local ne trouve pas d'espace disponible, le "Garbage Collector" est lancé.

3.3 Le "garbage collector" dans un environnement réparti

Nous introduisons dans cette partie, une manière de généraliser la collection mémoire centralisé, dans un système constitué de plus d'une unité de calcul. Cette généralisation est inspirée des sites WEB. En effet, sur INTERNET un site *S* peut être amené à utiliser des données (pages html, sons, images, vidéos … etc.) réparties sur d'autres sites distants. Bien que ces données soient éloignées, elles font partie de l'espace utilisé par le site *S;* en d'autres termes, elles font partie de l'espace *mémoire* du site, par conséquent une gestion de cette espace doit être assurée.

Comme nous avons déjà dit, le module de récupération mémoire ou le "Garbage Collector", constitue un module très important dans la gestion mémoire. Dans un environnement distribué, l'aspect de récupération est plus compliqué que dans un environnement centralisé. Quand un site *S*, demande de l'espace mémoire sachant qu'il n'y a plus d'emplacement ou cellule libre, le module de récupération mémoire distribuée doit être lancé afin de récupérer le maximum d'espace libre non utilisé.

Le "Garbage Collector" distribué (ou DGC : "Distributed Garbage Collector") est constitué de deux phases : la première, consiste à lancer une récupération locale. La deuxième phase, consiste à envoyer des requêtes d'allocation distantes, aux sites qui hébergent des données du site initiateur. Puisque la deuxième phase nécessite une communication entre plusieurs sites, l'ensemble des participants doit suivre un ensemble de règle ou *protocole* pour atteindre l'objectif commun. Dans la section 3.6, nous introduisons un protocole qui assure la communication entre l'initiateur du DGC, et le reste des participants.

3.4 La gestion de la mémoire dans l'environnement réparti

Pour un site donné, la mémoire est constituée de l'ensemble d'emplacements utilisés par ses applications locales. Cela inclut tout espace utilisé par le site, que ce soit locale

ou distant sur un autre site. Ainsi la mémoire d'un programme peut être formée par l'union de plusieurs espaces éparpillés sur le réseau .

Avec cette nouvelle vision, les demandes d'allocations et de libérations, peuvent mettre en jeu plusieurs sites. Un site peut alors, allouer de la mémoire en utilisant de l'espace sur un site distant. Le nouvel espace alloué, sera considéré exactement comme s'il faisait partie de la mémoire locale du site allocateur. Analogiquement pour la libération, un site peut être amener à libérer de l'espace mémoire sur un autre site du système distribué.

En suivant notre protocole, introduit dans la section 3.5, une allocation distribuée d'un espace de taille *t,* est matérialisé par la réception du message <Décision, *t*> et une libération par la réception du message <Libération, *t*>.

Si *T* est la taille de la mémoire globale pour un site *p*, *l_alloc* est la taille de l'espace mémoire utilisé par les applications locales du site *p*, *d_alloc* est la taille de l'espace locale utilisé par le réseau, et *dispo* est la taille de l'espace libre sur la mémoire locale de *p*. On a, dans le cas général, la formule suivante :

$$T = l\_alloc + d\_alloc + dispo$$

Si cette égalité n'est pas vérifiée, on aura $T < l\_alloc + d\_alloc + dispo$. Cette situation arrive quand on a l'espace libre qui n'est pas utilisé; en d'autres termes : quand on a de l'espace libre, qui n'est pas considéré libre. Dans cette situation le module "Garabage Collector" distribué doit être lancé.

3.5 Le protocole "DGCP"

Un protocole distribué, est un algorithme exécuté par un ensemble de processus (ou sites), appelés *participants*, afin de résoudre un problème donné. Le protocole est lancé par un site particulier appelé *initiateur*.

Le but de notre protocole, est d'assurer le module de la récupération distribué de mémoire; et cela en suivant un ensemble de règles. Le protocole du DGC est lancé par un site donné, après l'exécution de son "Garbage Collector" locale, le protocole permet au site initiateur d'utiliser de l'espace mémoire, qui peut être lui aussi récupéré sur des sites distants.

Le protocole que nous introduisons ici, est inspiré du protocole, appelé 2PC (two phases commit protocol) [BHG 87], utilisé dans la validation atomique des bases de données réparties (dans [LEM 99] une évaluation expérimentale de ce protocole est détaillée). Sauf qu'ici, le temps d'exécution est amélioré car la notion d'atomicité n'est pas importante.

Notre protocole, que nous appelons DGCP (Distributed Garbage Collector Protocol), consiste en deux phases : Une phase de *proposition*, et une phase de *décision*. Dans la phase de proposition, l'initiateur envoie une requête d'allocation mémoire et attend les réponses de différents sites participants. Les sites participants, sont les sites qui hébergent des données pour le site initiateur, autrement dit, une partie de leurs mémoires, fait partie de la mémoire globale du site initiateur. Dès que

l'initiateur reçoit des réponses qui peuvent satisfaire sa demande, il procède à l'exécution de la deuxième phase. La deuxième phase consiste à envoyer les décisions aux sites concernés. Une décision est un ordre définitif d'allocation de mémoire envoyé par l'initiateur vers le site correspondant. Notons que l'initiateur peut envoyer des décisions même si la totalité des réponses n'est pas encore reçue.

Le lancement du protocole DGC est illustré par l'algorithme suivant :

```
Tant que (vrai) faire
  Utiliser de l'espace local disponible;
  Si (l'espace local  n'est pas suffisant)
  Alors
     lancer le GC local;
     Si (l'espace récupéré  n'est pas suffisant)
     Alors lancer le DGC;
     Fsi;
  Fsi;
Fait;
```

Le protocole DGC simplifié :

```
Algorithme initiateur;

Début
   taille := taille demandée;
   envoyer ( Demande , taille ) à tous les participant;
   attendre (Réponse , valeur ) de tous les participants;
   lors de la réception de (Réponse , valeur ) du site p faire
            Si (valeur < taille )
           Alors
                    taille := taille - valeur ;
                    envoyer ( Décision , valeur ) à p ;
           Sinon
                    envoyer ( Décision , taille ) à p ;
                      taille := 0 ;
            Fsi;
   Fait;
Fin.
```

```
Algorithme participant;
```

Début
attendre (Demande, valeur) de l'initiateur;
Lors de la réception de (Demande , valeur) **du site** initiateur **faire**
lancer le GC local**;**
envoyer (Réponse , taille_disponible) **à** l'initiateur **;**
Fait;
Lors de la réception de (Décision , valeur) **du site** initiateur **faire**
allouer (valeur) **à** initiateur**;**
Fait;
Fin.

Sachant que *n* est le nombre de participants, i.e. le nombre de sites dans lesquels il existe un espace mémoire non nul utilisé par l'initiateur du protocole, le nombre de messages envoyé à chaque tour du protocole est égal à n. Par conséquent la complexité de notre protocole est dans le pire des cas égale à 3*n* messages.

Comme c'est le cas dans le 2PC, notre protocole est bloquant en cas de panne de site. Exemple : Si l'initiateur du protocole tombe en panne après l'envoi de sa requête de demande de mémoire. Un site *p* qui a envoyé sa réponse reste bloquant. Le site *p* ne peut pas utiliser la taille de mémoire déclarée disponible qu'après la réception de la réponse de l'initiateur. Puisque l'objectif de ce rapport n'est pas la tolérance de panne, nous nous satisfaisons de donner une proposition pour résoudre le problème de blocage. Notre proposition se résume en l'utilisation du paradigme du consensus dans le DGC en utilisant la solution de Chandra et Toueg qui se base sur l'utilisation de la classe de détection de défaillance ◊*S* [CHA 96]. L'application du paradigme du consensus a fait l'objet de beaucoup de travaux récents [BAD 98], l'implémentation de tel paradigme a été bien détaillée dans [LEM 99].

4 Mise en œuvre et résultats

Nous décrivons dans cette section, l'application "DGC" qui permet d'appliquer le principe de la récupération mémoire et les principaux résultats obtenus.

4.1 Description de l'application "DGC"

L'implémentation du "Garbage Collector" a été depuis longtemps un intérêt majeur de beaucoup de travaux de recherche. Notre application, que nous appelons DGC (Distributed Garbage Collector), est réalisée dans le but de voire l'intérêt du module "Garbage Collector" pour les langages de programmation qui utilisent le style fonctionnel et tout autre langage ou système basé sur le stockage dynamique. L'implémentation des notions de récupération a été faite dans le but de supporter les deux environnements : centralisé et distribué.

Dans notre l'implémentation, nous nous sommes basés sur la technique connue par "Mark-Sweep" ou marquage et balayage. Cette technique consiste à effectuer deux étapes, une de marquage et l'autre de balayage.

L'implémentation d'un algorithme centralisé dans un environnement distribué, nécessite de nouveaux outils pour assurer l'obtention du résultat voulu dans le nouveau système constitué de plus d'une unité de calcul. L'application DGC permet d'assurer la récupération mémoire centralisée, et même pour tout programme qui est en interaction avec le réseau. En effet, notre application peut être exécutée sur une seule machine, comme elle peut être exécutée sur un réseau local ou sur Internet, dans le dernier cas il suffit d'indiquer les adresses IP des unités participantes.

Quatre fonctions principales sont assurées par la DGC :

1- L'allocation mémoire (centralisée et distribuée).
2- La libération (centralisée et distribuée).
3- La simulation de la perte d'espace.
4- Le module de récupération ou le "Garbage Collector" dans les deux cas : centralisé et
distribué.

Notre Application permet de le "Garbage Collector", dans plusieurs états de l'environnement. En effet, l'utilisateur peut varier lui-même les paramètres de l'environnement (taille de la mémoire locale sur les différents sites, les requêtes d'allocations et de libérations, … etc.).

• A chaque simulation l'utilisateur peut introduire des différents paramètres d'environnement :

* La taille de la mémoire locale.
* Les adresses I.P. des différents sites du système réparti.
* L'adresse du port local.
* L'adresse du port distant (dans le cas où il existe un seul Client et un seul Serveur).
* La taille de l'espace demandé, lors du lancement du protocole de récupération distribuée.

Dans le cas contraire, les paramètres par défaut seront pris en compte.

• Après avoir configurer l'environnement de l'exécution du DGC, l'utilisateur peut manuellement simuler :

* Des allocations mémoire.
* Des libérations.
* Des pertes mémoire (les pertes sont matérialisées par des liens rompus aléatoirement dans les listes chaînées déjà allouées).

• Si l'utilisateur exécute une requête d'allocation, et il n'y a pas d'espace mémoire local suffisant, le DGC se lancera automatiquement. Le DGC essaiera de récupérer de

l'espace mémoire locale s'il existe. Dans le cas échéant, le DGC lance le protocole - que nous avons déjà introduit - pour la récupération distribuée de l'espace mémoire.

• L'utilisateur peut aussi lancer le DGC manuellement.
• L'utilisateur peut consulter à n'importe quel moment le scénario d'envoi et de réception de messages du protocole DGC en cliquant sur "Voir le journal du DGC".

L'application DGC, a été réalisée sous une plate forme WINDOWS sous Borland Delphi professionnel, version 4.0 (build 5.37). Sous cet environnement, les solutions algorithmiques peuvent être facilement implémentées. En plus de cela l'environnement est riche dans le domaine de communication réseau à travers les protocoles qu'il assure.

Dans la partie suivante, nous allons décrire les principales structures de donnés utilisées.

4.1.1 Structures de donnée

Dans notre mise en œuvre, nous avons utilisé quatre structures de donné principales :

1- Une structure "cellule mémoire".
2- Une table de symboles.
3- Une table de répartition de l'espace local utilisé par le réseau.
4- Une table de répartition de l'espace provenant du réseau.

La première structure de donnée, est la brique de base de toutes les autres structures
utilisées. Comme nous avons déjà dit, l'espace mémoire alloué à une application, est un ensemble de cellules mémoire qui sont utilisées pour représenter les expressions du programme.

Dans LISP, chaque cellule contient deux champs *left* (ou *car*) et *right* (ou *cdr*), ces champs pointent soit vers d'autres cellules ou vers des *atomes*, les atomes sont des cellules qui ne contiennent aucun pointeur. Une cellule LISP, contient aussi deux champs de type booléen, un de ces deux champs est utilisé pour faire la différence entre les cellules atomiques et non atomiques, et l'autre est utilisé pour le Marquage [COH 81]. Pour raison de simplification, une cellule mémoire dans notre implémentation, a la structure suivante :

```
ptr = ^cellule;
cellule = record
            info : char;
             marque :boolean;
            svtmem,svtaloc: ptr;
           end;
```

Initialement, le champ info de la cellule n'a pas de sens, ce n'est qu'après l'allocation de cette cellule que ce champ sera utilisé. Le champ marque est utilisé par la technique "Mark-Sweep" du récupérateur, il indique l'état de la cellule i.e. si la cellule est libre ou allouée. Le reste des champs, jouent le rôle de pointeurs de chaînage.

La deuxième structure utilisée, est la table de symboles (TS). Dans cette table on sauvegarde des pointeurs vers les listes utilisées par le programme utilisateur. Lors de chaque libération ou allocation de mémoire, cette table est mise à jour par l'allocateur de mémoire. La structure table des symboles est déclarée comme suit :

```
TS : array [0..N-1] of ptr;
```

Avec N est le nombre d'entrées de la table des symboles. Si on suppose que chaque entrée de la TS pointe vers au moins deux cellules chaînées; N sera au maximum égale à la moitié de la taille mémoire en nombre de cellules.

La table de répartition de l'espace local utilisé par le réseau, indique l'ensemble des cellules locales utilisées par le réseau en précisant pour chaque sous ensemble, ou liste, l'adresse IP du site qu'il l'alloue.

```
type
elt = record
        IP:string;
        svt : ptr;
      end;

TS_Sites: array [0..S] of elt;
```

La table de répartition de l'espace provenant du réseau, indique la taille de l'espace utilisé localement mais qui provient du réseau. Chaque entrée dans cette table est un couple <taille, adresse IP> :

```
type
espace = record
    IP : string;
    taille : integer;
   end;
Sites : array [0..S] of espace;
```

Les deux dernières structures, sont utilisées dans l'implémentation du "Garbage Collector" en environnement distribué. Ces deux structures, sont mises à jours par l'envoi et la réception des requêtes d'allocation ou de libération mémoire distribuées.

4.1.2 Le protocole UDP

Pour assurer la communication entre les différentes unités du système distribué, notre application utilise le protocole de communication appelé UDP (User Datagram Protocol). Ce protocole permet l'envoi des paquets datagrammes par un réseau Intranet ou par Internet en mettant en disposition un grand nombre de primitives et méthodes.

Pour pouvoir envoyer des paquets datagrammes en utilisant UDP, le programme client doit connaître l'hôte et le port distants auxquels envoyer des données et cela pour spécifier les propriétés "RemoteHost" et "RemotePort". Pour envoyer réellement les données, le programme client peut utiliser la méthode "SendBuffer" pour envoyer des tampons (tableaux de caractères) à l'hôte distant ou la méthode "SendStream" pour envoyer des flux de données.

Pour pouvoir recevoir des données UDP, la propriété "LocalPort" doit être initialisée.
Quand des données UDP sont prêtes à être lues, l'événement "OnDataAvailable" est appelé. Dans cet événement, le client peut utiliser la méthode "ReadBuffer" pour lire les données dans un tampon, la méthode "ReadStream" pour les lire dans un flux.

4.2 Résultats et discussion

Une exécution d'un programme basé sur le traitement des listes, peut invoquer plusieurs exécutions du module de récupération mémoire. Le nombre de ces exécutions, dépend de l'activité du programme utilisateur de point de vue allocation et libération mémoire.

Pour évaluer le module de récupération distribué que nous avons implémenté, nous allons calculer le temps d'une exécution du "Garbage Collector" distribué:

Supposons que n est le nombre moyen de cellules marquées dans une récupération, et m est le nombre total de cellules mémoire réservées au départ au programme utilisateur. Nous avons alors $m - n$ cellules récupérées. Le temps d'une récupération locale est donné par la formule suivante :

$$\text{Temps de récupération locale} = \alpha\, n + \beta\, (m - n) \ldots..(I)$$

Avec α, est le temps moyen, nécessaire pour marquer une cellule utilisée; et β est le temps moyen nécessaire lors de l'exécution effective de l'opération de récupération d'une cellule non utilisée. Dans [KNU 73], on trouve plus de détail concernant les paramètres α et β, et leur estimation.

Pour calculer le temps de récupération par mot (unité de mémoire), il suffit de diviser les deux membres de l'égalité (I) par $(m - n)$. Si $\rho = n / m$ est le pourcentage

d'occupation de mémoire, i.e. le pourcentage de cellules utilisées par rapport au nombre total de cellules mémoires, on aura :

$$\text{Temps de récupération locale par mot} = \frac{\alpha \rho}{1 - \rho} + \beta \quad \text{.....(II)}$$

Nous proposons la figure suivante, qui illustre la relation entre le temps de récupération par mot, et le pourcentage d'occupation de mémoire, le α et β étant fixes.

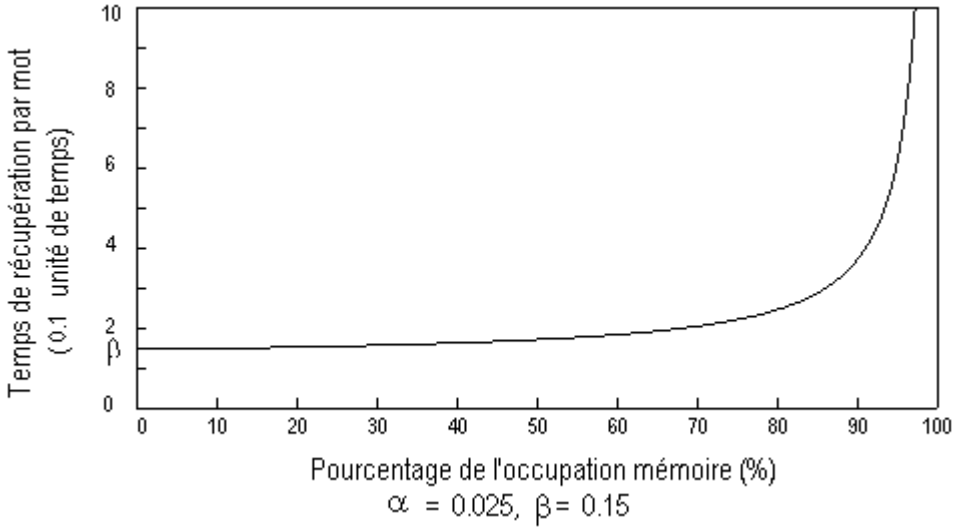

Figure 5. Le temps de récupération des cellules

Notre graphe, montre clairement l'inefficacité de la récupération quand la mémoire devient surchargée.

La formule (II), donne le temps d'exécution d'une récupération locale, soit *TRL* cette valeur. Calculons maintenant, le temps de récupération distribuée. Comme nous avons déjà vu, la récupération distribuée nécessite l'exécution d'un protocole que nous avons appelé DGCP. Si t est le temps moyen nécessaire pour le transfert d'un message dans le réseau; et ξ est le temps de traitement moyen d'un message; la formule suivante donne le temps d'exécution de notre protocole :

$$\text{Temps d'exécution du DGCP} = 3(t + \xi)$$

Puisque le DGCP exige que chaque site participant doive lancer le "Garbage Collector" local à son niveau, le temps global de récupération est donné par la formule suivante :

$$\text{Temps de récupération} = \sum_{i=1}^{s} (\alpha_i n_i + \beta_i (m_i - n_i)) + 3s(t + \xi)$$

Avec *s* est le nombre de participants dans le DGCP. Le temps de récupération distribuée par mot pour chaque site est donné par :

$$\text{Temps de récupération distribuée par mot} = \frac{\alpha \rho}{1 - \rho} + \beta \frac{3s(t + \xi)}{m - n}$$

Notons que le temps de récupération distribuée est plus élevé que dans le cas centralisé, mais il ne faut pas oublier qu'avec un système centralisé (une seule unité de calcul), l'espace d'adressage, ou l'espace mémoire utilisé par un programme utilisateur, est réduit à la capacité de la mémoire locale. Or avec un système distribué, un site peut utiliser la capacité énorme mise à sa disposition par l'ensemble des sites qui lui sont connectés.

5 Conclusion

Depuis la naissance des langages basés sur la manipulation des listes, plusieurs algorithmes ont été étudiés afin d'assurer la récupération de l'espace inutilisé d'une manière puissante et efficace.

Dans ce rapport, nous avons étudié la récupération mémoire dans deux environnements différents, l'environnement centralisé classique et l'environnement réparti, et cela en utilisant la technique connue par "Mark-Sweep" ou Marquage-Balayage. Nous avons proposé un protocole de communication, inspiré du protocole 2PC, qui s'adapte bien à la récupération distribuée; nous avons aussi proposé une solution qui évite le blocage du protocole en cas de défaillances du système. L'application "DGC" réalisée, permet de voir clairement l'application pratique des principes du récupérateur, pour les programmes basés sur la gestion dynamique de mémoire.

Notre rapport peut être vu comme une contribution aux travaux de recherches, qui sont fait dans le domaine de récupération mémoire dans les systèmes distribués. Comme perspectives nous proposons d'implémenter d'autres techniques de récupération sur un système distribué et de les évaluer, en changeant les paramètres du réseau de communication.

Références

[BAD 98] BADACHE Nadjib.
Ordre causal et tolérance aux défaillances en environnement mobile.
Thèse de doctorat d'état en informatique ,USTHB,1998.

[BEL 96] Dr. BELKHIR Abdelkader
Systeme d'exploitation, mécanisme de base.
Office des publication universitaires. Edition 2.08.4241, 11-1996.

[BHG 87] BERNSTEIN Philip .A. ,HADZILACOS Vassos. ,GOODMAN Nathan.
Concurrency control and recovery in database systems.
Addison–Wesley , Reading , Massachusetts.
1987 , 370 pages.

[CHA 96] CHANDRA T.D., HADZILACOS V.,TOUEG S.
The weakest Failure Detector for solving Consensus .
Journal of the ACM, vol. 43(4), 1996, pp. 685-722.
(Une première version de cet article est parue dans *Proc. 11 th* ACM *Symposium on Principles of Distributed Computing* ,August 1992, ACM Press, p. 147-158).

[CHU 41] A. Church
The calculing of Lambda Conversion.
Prinston University Press, 1941.

[CHR] Christian Queinnec, Barbara Beaudoing, Jean-Pierre Queille
Mark during sweep rather than mark then sweep.
LNCS. Vol 365, pp 224-237.

[COH 81] Cohen Jacques
Garbage collection of linked Data structures.
ACM Commputing serveys. Vol 13, 3, pp 341-367, 1981.

[GLA 84] H. Glasser, C. Hankin, D. Till
Principles of functional programming
Printice Hall inter, 1984.

[GOL 83] Goldberg Adele, Robson David
Smaltalk 80, the langage and its implementation.
Addision Wesley, 1983.

[KAD 92] Lies KADDOURI

Intégration de la programmation fonctionnelle et de la programmation orientée objet. Thèse de Magistère en informatique.
Institut d'informatique, USTHB octobre 1992.

[KNU 73] Knuth, D. E.
The art of programming.
Fundamental algorithms, vol. 1, Addison Wesley (ACGMNPRS), Reading Mass., 1973.

[LEM 99] Tayeb LEMLOUMA, Sami REZGUI
Mise en œuvre d'un protocole de validation atomique basé sur le consensus.
Thèse d'Ingénieur d'état en informatique, promoteur: Dr. Nadjib BADACHE.
Institut d'informatique, USTHB. Juin 1999.

[LES 89] Lester David R
An efficient distributed garbage collection algorithm.
ACM Commputing serveys. Vol 13, 3, pp 207-223, 1989.

[MAC 62] MacCarthy John
The lisp programmers manual.
MIT press, Cambridge, May 1962.

[MAC 90] MacLennan B. J.
Functional programming, practice and theory.
Addision-Wesley, 1990.

[PEY 90] Peyton Jones S.L
Mise en œuvre des langages fonctionnels de programmation.
Departement of computer science, University of College London.
Traduction par Michel M., Maison Paris 1990.

[STE 75] Steele G.L
Multiprocessing compactifying garbage collection
ACM Commun. 18, 9, pp 495-508 (CGP), September 1975.

[SUP] Support du cours du module "Programmation avancée", assuré par Dr. Belkhir Abdelkader,
Institut d'informatique, USTHB. 1999/2000.

[TUR 76] D. A. Turner
SASL Language manual.
Technical report, St Andrews University, 1976.

[URL 1] http://sunsite.ust.hk/javatutorial/refobjs/garbage/

[URL 2] http://www.parc.xerox.com

[URL 3] http://www.dartmouth.edu/~emk/dylan/gc.html

[URL 4] http://www.iecc.com/gclist/GC-faq.htm

[WAD 76] Wadler, P.L
Analysis of an algorithm for real time garbage collection.
ACM Commun. 19, 9, pp 491-500 (AP), September 1976.