La gestion de la mémoire dans un système d'acteurs

K. Douzi, A. Elfaker

ENSIAS BP 713
Agdal Rabat
Maroc
Tel : 212 7 77 73 17 / 212 7 77 85 79
Fax : 212 7 77 72 30
E-mail : emi.ma\!ensias1\!douzi
: emi.ma\!ensias1\!elfaker

Résumé : La gestion manuelle de la mémoire est une source inépuisable d'erreurs faciles à commettre et difficiles à repérer. Le ramasse-miettes, ou **GC**[1], est une technique qui détecte et désaloue lui-même les objets inutiles dans la mémoire. Dans un système à acteurs beaucoup d'entités (acteurs) sont créées pour exécuter une tâche donnée et cesser d'exister ensuite. Le nombre important de ces acteurs temporaires pose un problème évident de gestion de mémoire qui influence la rapidité et l'efficacité du système tout entier. Nous proposons une technique basée sur une gestion de la mémoire centralisée au niveau d'un acteur spécial le **GCA**[2]. Cet acteur se charge de détecter et libérer l'espace mémoire occupé par les acteurs dont il détermine l'inutilité localement à un site. Il travaille en concurrence avec l'exécution des autres acteurs.

Mots clés :
Ramasse-miettes, Acteur, Concurrence, Méta-acteur.

Abstract : The garbage collector, or **GC**, is important in concurrent programming envirommments where large numbers of objects and process are being constantly created and discarded. This paper presents an algorithm to perform concurrent and local garbage collection of actors. The relevance of garbage collection in a system of actors is briefly discussed. The algorithm bases on central management at the level of a special actor **GCA**. This actor takes charge of collecting the garbages by communication with all actors in the site. The GCA and the actors run concurrently.

Keywords :
Garbage Collector, Actor, Concurrency, Meta-actor.

Introduction

Un gestionnaire de mémoire, ou **GC**, est un programme qui détecte les objets inutiles et les désaloue sans obliger le programme utilisateur à le faire explicitement. En effet la gestion manuelle de la mémoire est une source inépuisable d'erreurs faciles à commettre et difficiles à repérer : il suffit de désalouer par erreur un objet qui est encore utile, et le comportement du programme devient totalement erratique.

Parmi les objets concurrents actifs, il y a les acteurs. Un acteur est en fait un processus permanent ayant une partie mémoire réservée pour ses données. Un GC pour le modèle acteur ne réclame pas seulement l'espace occupé par un acteur inutile. Il réclame aussi des ressources utilisées par cet acteur. Nous proposons une technique de ramassage local pour

[1] Garbage Collector
[2] Garbage Collector of Actor.

le modèle Acteur sur un seul site (section 3). Cette technique utilise un acteur GCA, dédié pour accomplir la tâche du ramassage. Le GCA collecte les informations des acteurs du site pour construire un graphe local. Les informations qui doivent être fournies par un acteur concernent son activité et ses connaissances. En fait les acteurs n'ont aucune vision réflexive sur eux même et ne peuvent donc répondre à la demande du GCA. On introduit la notion de méta-acteur dans le modèle pour la gestion de la mémoire. Chaque acteur est identifié à son méta-acteur qui modélise les aspects structuraux, comportementaux de l'acteur et qui présente le traitement de coopération avec le GCA.

Dans la section 1, nous présentons le modèle Acteur. La définition d'un acteur miette et l'algorithme du marquage utilisé sont décrits dans la section 2. Dans la section 3 on définit les structures de données et le comportement de l'acteur GCA qui décrit la technique du ramassage local. Nous concluons en présentant quelques problèmes liés à la collection sur plusieurs sites.

1 Le modèle d'acteurs

Le modèle d'acteur défini par [HEWITT73] [HEWITT77] [AGHA86] est un modèle pour décrire des calculs concurrents dans lequel un problème est résolu par une communauté d'acteurs, entités indépendantes, qui communiquent entre elles et coopèrent pour la résolution d'un but commun. Chacun des acteurs de cette communauté travaille de façon autonome dans son domaine d'expertise en communiquant et échangeant des informations avec d'autres acteurs. Les acteurs modélisent des objets conceptuels ou physiques qui apparaissent dans l'application et les messages correspondent à des transferts d'informations, à des demandes d'informations ou à des réponses à ces demandes. Ainsi, par rapport à un problème posé, nous obtenons la structure d'une solution naturelle où la connaissance est répartie et le contrôle distribué entre les différents acteurs de la communauté.

Dans ce modèle, la communication est le seul moyen d'échange entre acteurs. Elle est semblable à un service postal :

- **asynchrone** ce qui nécessite d'avoir une boite à lettres gérée par un arbitre,
- **point à point**, (il n'y a pas de diffusion),
- **unidirectionnelle** (seul l'émetteur connaît le destinataire),
- **sûre** (tout message envoyé sera reçu dans un délai fini non borné).

Un acteur est un objet actif concurrent [SALLÉ93]. Il est identifié, de manière unique, par une référence analogue à une adresse postale, c'est l'adresse de sa queue des messages ; il est créé avec un comportement initial. Un acteur est complètement défini par sa référence, son comportement courant et l'ensemble des messages reçus et non encore traités. Le comportement d'un acteur est composé de :

- **l'ensemble de ses accointances,** qui sont les acteurs dont il connaît les adresses (il est **une accointance** inverse de chacun de ces acteurs).

- **de son script** qui décrit ses réactions en fonction des messages reçus (contrôle).

Le cycle de vie d'un acteur est une boucle qui consiste à prendre le premier message de la boite à lettres et à le traiter. Au cours du traitement d'un message, à partir des informations contenues dans ce message et de ses accointances, un acteur peut :

- **créer** dynamiquement de nouveaux acteurs : **create**(initial_behavior),
- **envoyer** des messages aux acteurs qu'il connaît (y compris lui même) : **send**(@acteur, message),
- **remplacer** son comportement courant par celui qui va traiter le prochain message : **become**(behavior).

Les messages reçus sont traités successivement et il y a donc exclusion mutuelle automatique. Le message peut contenir l'adresse d'un client auquel il faudra renvoyer une réponse.

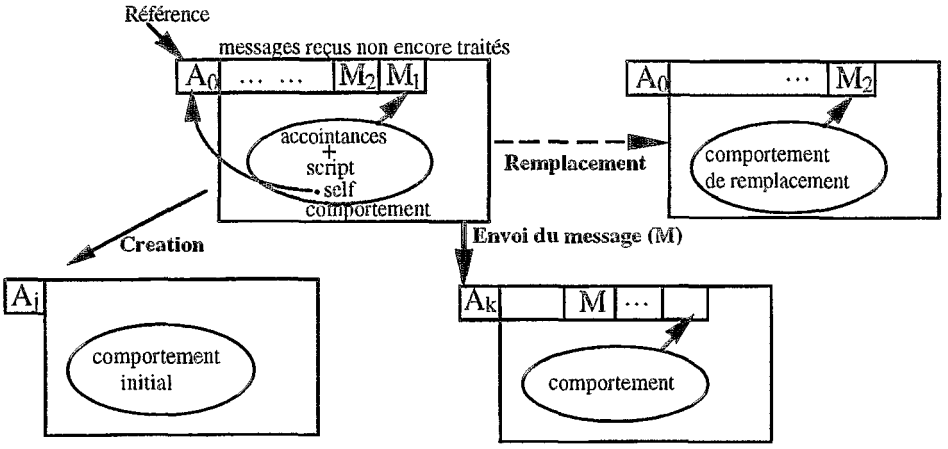

FIG. 1 - Traitement d'un message par l'acteur A_0

2 Ramasse-miettes et acteurs

Dans les systèmes manipulant des entités passives, la détection des miettes est équivalente à la détection des nœuds inaccessibles dans un graphe orienté. La définition des miettes dans un système gérant des acteurs, qui sont des objets actifs, est subtilement différente du fait du caractère exécutable des acteurs. Une première définition des acteurs inutiles a été énoncée dans [AGHA86], et a par la suite été précisée dans [KAFURA90]. Nous donnons tout d'abord une définition des miettes dans un système d'acteurs. Nous examinons ensuite une technique de ramassage pour le modèle d'acteurs.

2.1 Définition des miettes dans un système d'acteurs

Un acteur est **actif** s'il a au moins un comportement actif. Dans le cas contraire, il est dit **bloqué**. Un acteur peut envoyer un message à un autre acteur, et de ce fait rendre ce dernier actif. Comme pour les systèmes manipulant des entités passives, un ensemble d'acteurs **racines** est distingué. Les objets racines sont indispensables au calcul, et sont toujours actifs. La définition des objets racines dépend du système considéré. Typiquement, ce sont les objets permettant au système d'interagir avec le monde extérieur : les objets d'entrée/sortie avec un clavier ou un écran, ou encore les objets utilisés pour le nommage externe d'objets sont des objets racine [PUAUT94].

Intuitivement, un acteur est une miette s'il ne pourra pas échanger d'informations avec le monde extérieur, c'est à dire avec un acteur racine. Autrement dit, un acteur est une miette s'il ne pourra jamais envoyer un message à un acteur racine, et s'il ne pourra jamais recevoir un message d'un acteur racine. Un acteur non miette sera dit utile.

Remarquons que du fait du caractère actif des acteurs et de la possibilité d'envoyer des références à travers les messages, un acteur qui ne peut pas à un instant donné communiquer avec un acteur O, pourra éventuellement le faire dans le futur, par exemple parce qu'un autre acteur lui a passé une référence sur O. Ce point est illustré dans l'exemple

qui suit. La figure 2 schématise un système d'acteurs actifs. Les acteurs racines sont représentés par des triangles, et des flèches désignent les références entre les acteurs. Nous utilisons les notations introduites dans [KAFURA90] pour différencier les objets bloqués des objets actifs : les premiers sont représentés par des carrés, alors que les derniers sont désignés par des cercles.

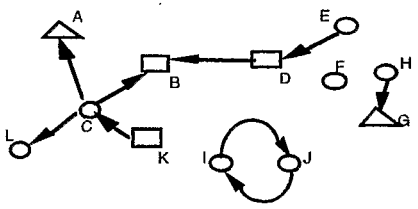

FIG. 2 - Un système d'acteurs

L'acteur H possède une référence sur l'acteur racine G et est actif : il peut donc communiquer avec G. Par conséquent, H n'est pas une miette. Il en est de même pour l'acteur C. Les acteurs I et J sont des miettes, car ils forment un cycle d'objets déconnecté du reste du graphe. C'est aussi le cas pour l'acteur F. L'acteur K est bloqué, et aucun autre acteur ne possède de référence sur lui. K est donc une miette. Les acteurs B, D, E et L ne sont pas des miettes, car tous les trois peuvent être activés, et à leur tour activer un acteur racine. Considérons par exemple le cas de l'acteur B : l'acteur E peut activer l'acteur D, qui peut alors activer l'acteur B ; si l'acteur C active ensuite l'acteur B en lui envoyant un message contenant la référence sur l'acteur racine A, B peut alors activer un acteur racine. Un raisonnement analogue peut être appliqué pour montrer que les acteurs D, E et L ne sont pas des miettes.

2.2 Un algorithme de ramassage

Un algorithme de ramasse-miettes prenant en compte le caractère actif des acteurs est décrit dans [KAFURA90]. Cet algorithme est une adaptation au cas des acteurs de l'algorithme de traçage classique. Trois couleurs sont utilisées pour marquer les acteurs :

- la couleur **blanche** désigne les acteurs qui ne peuvent pas communiquer avec un acteur racine,

- la couleur **grise** désigne les acteurs qui pourraient communiquer avec un acteur racine s'ils étaient activés,

- la couleur **noire** désigne les acteurs pouvant communiquer avec un acteur racine.

Initialement, tous les acteurs sont marqués en blanc, sauf les acteurs racines, qui sont marqués en noir. Cinq règles de marquage, énoncées ci-dessous, sont appliquées jusqu'à ce qu'aucun acteur ne soit plus colorié. Une règle de marquage ne s'applique que lorsque l'acteur à marquer possède une couleur strictement plus claire que la couleur avec laquelle on veut le marquer (blanc étant plus clair que gris, qui est plus clair que noir). La première règle marque en noir les acteurs pouvant être activés directement par un acteur utile. La deuxième règle marque en noir les acteurs pouvant activer directement un acteur utile. La troisième règle marque en noir les acteurs pouvant activer directement un acteur gris, c'est à dire un acteur utile s'il est activé. La quatrième règle marque en gris les acteurs bloqués qui pourraient, s'ils étaient activés, activer un acteur utile. Enfin la dernière règle marque en gris les acteurs bloqués, qui pourraient, s'ils étaient actives, activer un acteur gris.

R1 : Marquer en noir tous les acteurs référencés par des acteurs noirs.
R2 : Marquer en noir tous les acteurs actifs référençant des acteurs noirs.
R3 : Marquer en noir tous les acteurs actifs référençant des acteurs gris.
R4 : Marquer en gris tous les acteurs bloqués référençant des acteurs noirs.
R5 : Marquer en gris tous les acteurs bloqués référençant des acteurs gris.

A la fin de l'algorithme, tous les acteurs de couleur noire sont utiles. **Les acteurs de couleur blanche ou grise sont des miettes**, et peuvent être détruits. La terminaison du programme peut être facilement prouvée en remarquant que chaque acteur change de couleur au plus deux fois : une fois lorsqu'il est blanc et est marqué en gris, et une fois lorsqu'il est gris et est marqué en noir.

L'ordre d'exécution des règles de marquage des acteurs n'est pas fixé par les règles de marquage énoncées ci-dessus. Deux algorithmes réalisant ces règles de marquage sont détaillés dans [KAFURA90]. La complexité de ces deux algorithmes est de O(N) en occupation mémoire, et de $O(N^2)$ en temps d'exécution dans le pire des cas, où N désigne le nombre total d'acteurs. A titre de comparaison, la complexité d'un algorithme de marquage et balayage séquentiel est de O(N) en occupation mémoire et de O(N) en temps d'exécution [COHEN86]. Une amélioration de ces deux algorithmes en temps d'exécution, au détriment de l'occupation mémoire est présentée dans [DICKMAN91].

Cet algorithme, qu'on vient de décrire, présente la manière par laquelle on va détecter et réclamer les acteurs miettes. Le ramassage utilise le graphe des acteurs du site. Ce graphe résume les références entre acteurs et leurs états. Tout le problème réside en fait dans la collection de ces informations et la construction d'un graphe cohérent. Des questions telles que : Quand est-ce que l'acteur doit-il envoyer ses informations ? Les enverra-t-il à l'instant qu'il faut ? Connaît-il en fait ces informations pour les transmettre. Nous répondrons à ces questions dans la section 3.

3 Un GC concurrent pour les acteurs

Dans le but de décrire une technique de ramassage dans un environnement distribué, nous considérons pour chaque site un acteur spécial appelé GCA. Cet acteur est chargé de faire le ramassage local. Il coopère aussi avec les autres acteurs (GCAs) des autres sites pour un ramassage global. On ne s'intéressera dans ce papier qu'à la gestion de mémoire dans un seul site. Nous évoquerons dans la conclusion quelques idées sur la réalisation d'un GC sur plusieurs sites. L'acteur GCA a besoin de connaître l'état du système à un instant donné. Cet état représente le graphe des acteurs existants dans le site et plus précisément les accointances et l'état d'activité pour chaque acteur. Les informations (accointances et état d'activité) sont envoyées par les acteurs eux même au GCA. Cet envoi peut être réalisé à chaque fois que le GCA le demande à l'aide d'un message particulier **[Info?]**, ou d'une façon systématique : chaque création d'un nouvel acteur ou envoi d'un message sont déclarés au GCA. Lorsque le GCA possède une vue globale et cohérente des acteurs du site, il commence le marquage à l'aide de l'algorithme de Kafura (voir la section 2.2). A la fin du marquage, les acteurs gris et blancs sont des miettes qu'il faut réclamer. Le GCA envoie un message **[Détruire]** à chaque acteur miette. Ce message entraîne la mort de l'acteur (la fin du processus avec libération de l'espace mémoire qu'il occupait). L'envoi des informations à la demande du GCA souffre cependant d'un certain nombre de problèmes liés au fait que le modèle est asynchrone. Le message [Info?] peut ne jamais être traité ; c'est un facteur de blocage du système. En effet l'acteur qui exécute une instruction " let x = create(b) in ...", consistant à créer un nouvel acteur, et attend que l'espace soit alloué pour ce nouvel acteur peut ne jamais recevoir le message [Info?] (si la mémoire est pleine, voir figure 3).

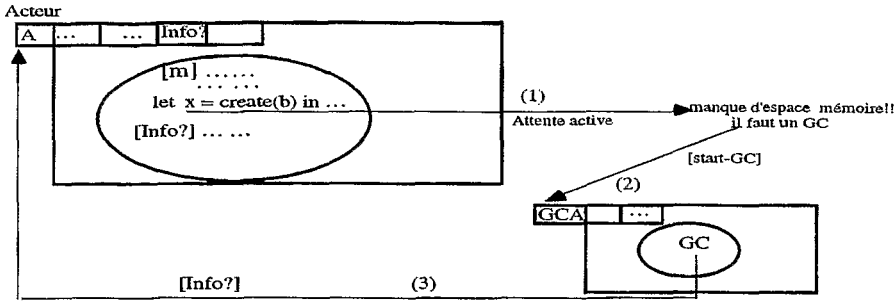

FIG. 3 - A est bloqué donc il ne peut pas traiter le message [Info?]

L'envoi systématique évite aux acteurs de coopérer avec le GCA au moment du GC. De plus le GCA est toujours au courant de tous les changements concernant les liaisons d'accointances dans le graphe des acteurs. Ceci assure la consistance des informations. Cependant cette méthode est lourde parce qu'elle impose un surcoût au mutateur[3] . En effet, les envois de messages pour déclarer des informations au GCA sont exécutés par les acteurs. Cette méthode est particulièrement inefficace si aucun GC n'est réalisé. Toutes les informations sont non exploitées. Cette méthode peut être améliorée en imposant aux acteurs de n'envoyer des information que dans un intervalle précis défini par le GCA.

Nous présentons dans la section 3.2 une autre technique permettant à chaque acteur d'avoir des connaissances sur lui même. Ceci est réalisé en introduisant au sein de l'acteur une activité annexe qui s'occupe exclusivement de la coopération avec le GCA. Cette approche peut être comparée à l'approche méta-objet dans [YONEZAWA88].

3.1 Détermination des accointances locales et l'état d'activité d'un acteur

Dans un site, il n'y a pas de messages en transit. L'envoi d'un message consiste en effet à déposer l'adresse du message dans la queue du destinataire. Par conséquent il n'y a pas des accointances en transit. Les accointances sont les acteurs dont les adresses se trouvent dans la file des messages y compris le message courant. Elles peuvent aussi être des acteurs créés pendant le traîtement courant ou des acteurs globaux qui sont liés au script courant. On tient compte aussi bien des adresses qui se trouvent dans la queue de messages que des adresses qui sont liées au script courant. Cela permet d'éviter la destruction des acteurs dont la seule référence est dans les queues de messages. Pour bien illustrer ceci la figure 4 présente un système composé de trois acteurs A, B et C. B connaît A et C. Supposons que la file de messages de A n'est pas vide et que l'acteur B envoie l'adresse de C à A et détruit juste après sa référence à C. Pendant le cycle du GC si on ne tient pas compte de l'adresse de C qui se trouve dans la queue de messages de A. L'acteur C peut être réclamé incorrectement.

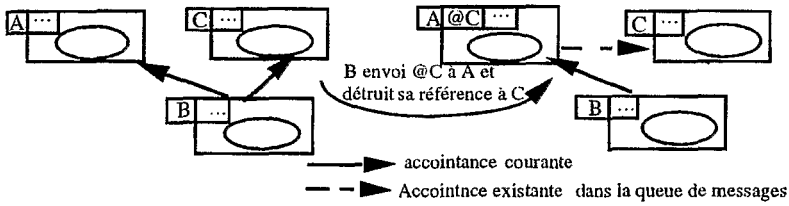

FIG.4 - Le GCA doit tenir compte des adresses d'acteurs existants dans le message

[3] Activités de l'application

La recherche dans la file de messages consiste à déterminer parmi les valeurs communicables celles qui sont succeptibles d'être des adresses d'acteurs du site. En ce qui concerne les accointances courantes, pour avoir celles qui sont créées au cours du traitement, on associe à chaque comportement une liste, **liste-accointances**. Après chaque création d'un acteur, l'adresse de ce dernier est insérée dans cette liste d'accointances. La liste est mise à vide après chaque changement de comportement. Les adresses des acteurs globaux sont des données explicites et sont connues dès la création de l'acteur. En ce qui concerne l'état d'activité d'un acteur, le fait que la file de messages est vide (respectivement pleine) est suffisant pour affirmer que l'acteur est bloqué (respectivement actif).

3.2 Ramasse-miettes à l'aide du méta-acteur

Chaque acteur A du système peut être vu sous deux aspects structurel et comportemental. Le premier aspect représente sa queue **q**, son comportement **b**, sa mémoire locale **state** et son état d'activité **mode**. Alors que l'aspect comportemental représente ses différentes réactions vis à vis des messages (réception, acceptation, traitement ...). Ces deux aspects peuvent être modélisés à l'aide d'une seule entité : le **méta-acteur** $\hat{A}$. Ce dernier est un acteur contenant les informations du méta-niveau à propos de A. La structure de A est représenté comme une donnée dans la mémoire locale de $\hat{A}$. L'aspect comportemental est décrit par le script du méta-acteur $\hat{A}$. Dans notre cas particulier du ramasse-miettes, on ajoute à chaque méta-acteur la tâche de coopération avec le collecteur local GCA.

Au méta-acteur $\hat{A}$ deux activités sont associées. Une est dédiée au traitement des messages enfilés dans la queue de messages du méta-acteur. L'autre représente la capacité du calcul de l'acteur dénotation A : Elle consiste à exécuter les actions correspondantes aux messages destinés à A (figure 5).

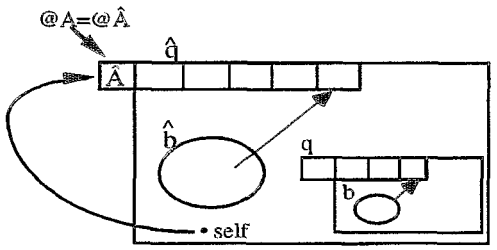

FIG. 5 - Définition d'un méta-acteur

La création d'un acteur A avec un comportement b entraîne la création d'un méta-acteur $\hat{A}$ de comportement $\hat{b}$. Ce qui revient à traduire chaque ordre de création create(b) par create($\hat{b}$). Le comportement $\hat{b}$ consiste à insérer les messages destinés à A dans la file q. Il consiste aussi à coopérer avec le GCA. A la réception du message [Info?], le méta-acteur cherche ses accointances qu'il transmet ensuite au GCA accompagnées de son adresse et de son état d'activité.

```
(meta-actor Â
(behavior b̂ {q, b, state, mode)
  (=>>[m] (q = q++m)
   =>>[Info?] (mode = (q={})? bloqué : actif)
      (acc = (find-acc q b state))
      (send ([self, mode, acc], GCA)))
```

FIG. 6 - Le comportement d'un méta-acteur

La figure 6 représente le comportement d'un méta-acteur associé à A. L'aspect structurel de l'acteur A est représenté comme des valeurs des variables locales (b, state, q et mode) du méta-acteur Â. L'état courant de l'acteur représenté par la valeur mode indique si l'acteur est une *racine*, s'il est *bloqué* c'est à dire que sa file q est vide ou s'il est *actif*.

Chaque méta-acteur représente sa dénotation. Pour abréger nos propos on utilisera dans toute la suite indifféremment le mot acteur ou méta-acteur.

Avant de spécifier le protocole de communication entre le GCA et les méta-acteurs du site, nous décrivons d'abord l'acteur GCA.

3.3 Les structures de données pour le GCA

L'acteur GCA connaît tous les acteurs existants dans le site. Dès la création d'un acteur x (c'est à dire à l'exécution d'une instruction "x = create(b)" par un acteur existant), son adresse est envoyée au GCA qui crée un nœud et l'insère dans une liste d'allocation, **act-list**. Chaque nœud de cette liste contient les quatre champs (voir figure 7) :

- un champs, **Adr,** contenant l'adresse de l'acteur nouvellement créé

- un champs de deux bits, **Marq**, pour coder les trois couleurs blanc, gris et noir qui sont utilisés pour distinguer (respectivement) si un acteur est non exploré, peut être utile ou utile respectivement.

- un champs, **Acc,** contenant un pointeur sur la tête de la liste qui contient les accointances de l'acteur d'adresse Adr.

- un champs, **etat,** contenant la valeur de l'état d'activité de l'acteur.

- un bit, **Rec,** pour signaler le fait que le GCA a reçu des informations d'un acteur. Ce bit est initialisé à 0 et positionné à 1 lorsque le GCA reçoit les informations d'un acteur.

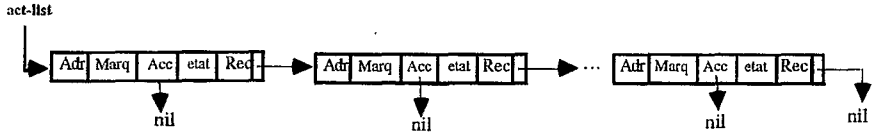

FIG. 7 - La liste d'allocation act-list du GCA

La liste act-list est mise à jour après chaque création d'acteur par l'ajout, à la fin de la liste, d'un nœud contenant son adresse comme valeur du champs Adr, la couleur blanche pour le champs Marq et la valeur nil pour le pointeur Acc. Après la réclamation des miettes le GCA supprime les éléments correspondants aux acteurs miettes de la liste d'acteurs (act-list) et réinitialise en outre tous les champs des éléments restants de act-list pour se préparer à un nouveau cycle de GC.

3.4 Protocole de communication entre le GCA est les méta-acteurs du site

L'acteur GCA déclenche un GC en diffusant à tous les méta-acteurs existants un message [Info?]. Quand un méta-acteur Â reçoit ce message, il cherche l'ensemble des accointances, acc, dans la queue des messages (q), le comportement courant et dans la mémoire locale (state) de sa dénotation. Il envoie ensuite une liste **Info(@Â,mode,acc)**, contenant son adresse, son état d'activité et l'ensemble de ses accointances, au GCA. Quand ce dernier

reçoit cette liste, il positionne le bit Rec, correspondant à Â dans act-list, à 1, insère l'ensemble des accointances dans la liste pointée par Acc et affecte la valeur du mode au champs etat.

Si un méta-acteur est créé pendant le GC, alors l'acteur GCA doit recevoir les informations concernant ce méta-acteur. Il se peut en effet qu'il possède une référence sur un autre méta-acteur et le GCA doit en tenir compte. L'acteur GCA envoie au méta-acteur, nouvellement créé, le message [Info?] dès la réception de son adresse. Au moment du GC s'il n'y a pas de l'espace mémoire alors les instructions, qui consistent en la création des méta-acteurs seront bloquées jusqu'à la fin du GC. Ce blocage n'entraîne pas une interruption du calcul mais seulement un retard de l'exécution de ces instructions. Lorsque tous les bits Rec dans la liste act-list sont mis à 1, cela veut dire que le GCA a reçu des informations de tous les méta-acteurs existants dans le site. Avant de commencer le marquage le GCA doit avoir un graphe cohérent. En effet ce graphe doit refléter à un instant dans le passé l'état global du site. Un algorithme de calcul de l'état global cohérent d'un système caractérisé par l'absence des messages en transit dans un contexte asynchrone peut être trouvé dans [HELARY90].

Comportement du GCA

```
(actor GCA
(behavior Comp_GCA {act-list, End_Rec}

=>> [start_GC]                          {envoyé par le système si la mémoire est pleine}
   - End_Rec := false,        {End_Rec détecte la terminaison de la collecte des informations}
   - send_all([Info ?]),                {le GCA diffuse ce message à tous les méta-acteurs }
   - become(Comp_GCA(act-list, End_Rec)).

=>> [@Â]                                {le méta-acteur Â est nouvellement créé}
   - créer un nœud avec Adr := @Â, Rec := 0, Marq := blanc et Acc := nil,
   - insérer le nœud à la fin de la liste act-list,
   - si End_Rec = false alors send(Â, [Info ?]),
   - become(Comp_GCA(act-list, End_Rec)).

=>> [Info (@Â, mode, acc)]          {réception de l'information depuis le méta-acteur Â}
   - mettre_à_jour(Rec),                {cette fonction met Rec correspondant à @Â, à 1}
   - insérer (acc, Acc),          {insère l'ensemble acc dans la liste Acc correspondant à @Â}
   - etat := mode,                      {affecte mode à etat correspondant à @Â }
   - End_Rec = tous les bits Rec de act-list sont mis à 1,
   - si End_Rec = true alors
                - appliquer l'algorithme de marquage de Kafura,
                - Pour tout méta-acteur miette Â faire send(Â, [Détruire]),
                - supprimer_de_act-list(les méta-acteurs_miettes),
                - pour tout méta-acteur initialiser Rec, Marq et Acc.
                fsi.

   - become(Comp_GCA(act-list, End_Rec))))
```

Conclusion

Nous avons proposé dans ce travail une technique de ramasse-miettes pour les acteurs. Un acteur GCA est dédié pour accomplir la tâche du GC. Le comportement de cet acteur et le protocole de communication avec les autres acteurs du site ont été décrits. Notre technique utilise la notion de méta-acteur pour permettre à chaque acteur d'avoir des

connaissances sur lui-même. Le GC proposé est concurrent avec les autres activités de l'application. Etant basé sur une technique de marquage opérant par traçage, il détecte aussi les miettes cycliques. Notre gestionnaire se limite à un seul site.

Dans un système distribué, les acteurs sont répartis sur plusieurs sites. Il est possible de concevoir un GC distribué en associant un collecteur local à chaque site. Un GC global se charge de collecter des acteurs qui sont référencés, ou ayant une référence dans un site distant [WASHABAUGH91]. Bien que les collecteurs locaux s'exécutent de manière indépendante, ils doivent se synchroniser pour détecter les acteurs globaux inutiles.

Nous souhaitons relâcher cette synchronisation conformément au modèle d'acteurs. Nous estimons aussi qu'il est nécessaire que l'exécution du ramasse-miettes ne bloque pas les acteurs pendant leur exécution.

Références

[AGHA86] G. Agha. *Actors: A model of concurrent computation in distributed systems.* MIT Press 1986.

[COHEN86] J. Cohen. *Garbage Collection of Linked Data Structures.* ACM Computing Srveys, 13(3):341-367, septembre 1986.

[DICKMAN91] P Dickman. *Distributed Object Management in a Non- Small Graph of Autonomous Networks With Few Failures.* MIT Press Septembre 1991.

[HELARY90] Helary J.M., Plouzeau N., Raynal M. *Computing particular snapshots in distributed systems.* Proc. 9th IEEE Int. Phoenix Conf. on Comp. and Comm., (1990), pp. 116-123.

[HEWITT73] C. Hewitt, P. Bishop, R. Steiger, *A universal modular formalism for artificial intelligence*, proceedings of the IJCA'73 1973.

[HEWITT77] C. Hewitt, *control structures as patterns of passing messages.* journal of Artificial Intelligence 8, 1977.

[KAFURA90] D.Kafura, D.M. Washabaugh J. Nelson. *Garbage collection of Actors.* Dans Proc. of the 1990 ECOOP/OOPSSLA conference, pp. 126-133, 1990.

[PUAUT94] I. Puaut. *A Distributed Garbage Collector for Active Objects.* Proc. of the 9th ACM-SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications Potland, Oregon, octobre. 1994.

[SALLÉ93] P. Sallé, J-P.Arcangeli, A. Marcoux, C. Maurel. *La programmation concurrente par acteurs : le système Plasmall.* Technical report, Institut de Recherche en Informatique de Toulouse, université Paul Sabatier. 1993.

[WASHABAUGH91] D. M. Washabaugh et D. Kafura. *A Distributed Garbage Collection of Active Objects.* Dans Proc. of 11th International Conference on Distributed Computing Systems, 11 (7):369-376, mai 1991.

[YONEZAWA88] A.Yonezawa. and T.Watanabe. *Reflection in an Object Oriented Concurrent Language.* Proceding OOPSLA 1988.