

Rapport de DEA
Spécification et implémentation d'un service
de ramasse-miettes pour une plateforme à
objets répartis

Philippe Bidinger

13 juin 2000

Résumé

Le but de ce projet est d'implémenter un service de *ramasse-miettes* (*garbage collector*) pour une plate-forme à objets répartis. Dans un premier temps, il s'agit d'étudier les différents algorithmes existants. Ensuite il faut déterminer un algorithme à implanter dans la plate-forme Jonathan développée par France Télécom.

1 Introduction

La récupération automatique de la mémoire (garbage collection) fait partie du support d'exécution d'un système ou est une librairie supplémentaire qui permet de déterminer automatiquement quelle partie de la mémoire un programme n'utilise plus, et permet de la recycler pour d'autres utilisations. À l'inverse la gestion manuelle de la mémoire est une contrainte importante pour le programmeur. Cela décroît sa productivité et est source d'erreur. L'utilisation de garbage collector lui permet de se situer à un plus haut niveau d'abstraction et simplifie les programmes.

Le but de ce projet est de spécifier et d'implémenter un service de garbage collector pour une plate-forme à objets répartis, Jonathan, développée par France Telecom R&D (centre de recherche et de développement de France Telecom).

Les chapitres suivants consistent en une brève présentation de l'entreprise, puis un état de l'art des techniques de GC centralisés et distribués, une présentation de Jonathan et enfin les détails de l'implémentation d'un GC dans Jonathan.

2 France Télécom R&D

France Télécom R&D, auparavant appelé le CNET (Centre National d'Étude des Télécommunications), est l'unité de recherche et développement de l'opérateur téléphonique France Télécom, géant présent dans 75 pays et au chiffre d'affaire de 27 milliards d'euros. France Telecom R&D compte 3800 salariés répartis dans neuf sites en France, dont un à Grenoble. Celui-ci abrite entre autres le département ASR (Architecture de Systèmes Répartis) de la direction DTL (Direction des Techniques Logicielles).

La DTL a pour mission de mener des études exploratoires sur les techniques novatrices en matière de traitement de l'information. Le rôle du département ASR est d'étudier et de définir des architectures pour systèmes répartis ouverts, d'organiser une veille technologique, et d'offrir une assistance technique aux autres directions en matière de systèmes distribués.

3 Garbage Collector centralisés

L'objectif de cette partie est de donner les principes et les motivations des techniques de GC. Elle fixe aussi le vocabulaire de base et les principaux algorithmes qui permettront d'aborder les techniques utilisées dans un contexte distribué. Une description très complète est donnée dans [7].

3.1 Principes

Les techniques de Garbage Collection ont pour but la récupération automatique de la mémoire. Une application peut accéder à la mémoire dans deux zones. Les *racines (roots)*, qui constituent la mémoire directement atteignable par les variables du programme (en général, les variables automatiques sont stockées dans la pile, alors que les variables globales le sont dans la zone statique) et le *tas (heap)*, qui est la zone à laquelle le programme accède indirectement via des pointeurs. Généralement, le programmeur doit libérer explicitement la mémoire dont il n'a plus besoin à l'aide de fonctions comme `free`. La présence d'un GC libère le programmeur de cette contrainte. Le rôle du GC est de détecter les objets (au sens large) qui ne sont plus utilisés par le programme et rendre la mémoire qu'ils occupent disponible pour une utilisation future.

On distingue deux types d'états pour un objet à un moment donné de l'exécution. Soit il est atteignable depuis les racines, soit on ne peut pas l'atteindre en suivant un chemin de pointeur (*garbage*). Par ailleurs, on dit qu'un GC est *correct* s'il ne réclame que de la mémoire que le programme ne pourra plus utiliser, et *complet* s'il réclame tous les objets inatteignables (au bout d'un temps fini).

3.2 Motivations

L'utilisation d'un GC facilite la programmation modulaire en réduisant les dépendances inutiles entre modules. Deux modules peuvent être indépendants fonctionnellement et toutefois utiliser une même structure de données. Ils devront alors se concerter pour la libération de la mémoire occupée par la structure, ce qui limite leur indépendance.

Plus couramment, la libération de la mémoire est connue pour être une activité difficile et une source d'erreur importante. La mémoire non récupérée s'accumulant peut fortement ralentir l'application (fuite de mémoire). À l'in-

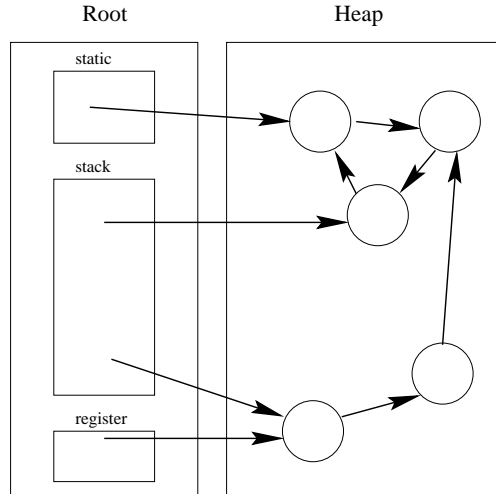


FIG. 1 – racines et tas d'un processus

verse un objet récupéré trop tôt peut engendrer une erreur à l'exécution. Ces erreurs sont particulièrement dangereuses dans la mesure où elles présentent un caractère aléatoire qui rend le debuggage particulièrement ardu.

Pour finir, contrairement à une idée reçue, l'utilisation d'un GC n'est pas forcément beaucoup plus coûteuse que la gestion explicite de la mémoire. On peut considérer qu'un ramasse-miette bien programmé entraîne (très approximativement) une baisse de performance de 10 pourcent par rapport à une désallocation explicite de la mémoire. Ce surcoût peut être considéré comme élevé, mais si l'on tient compte des avantages que procure un ramasse-miette, il est finalement très acceptable.

3.3 Principaux algorithmes

Le fonctionnement de base d'un GC consiste en deux phases:

- Distinguer les objets atteignables à partir des variables de l'application des autres (*détection*)
- Libérer la mémoire utilisée par les objets non atteignables (*réclamation*)

En pratique, ces deux phases peuvent être couplées de manières très diverses, ce qui conduit à une grande variété d'algorithmes différents. En particulier,

il est habituel de séparer la première phase en deux techniques différentes: le comptage de références et le traçage. Le comptage de référence consiste à associer à chaque objet un compteur du nombre de références sur lui. Les méthodes de traçage déterminent les objets atteignables en parcourant concrètement le graphe des références dans le tas en partant des racines.

Dans la suite, nous allons présenter les trois principaux algorithmes. Le premier utilise la méthode de comptage de référence, les deux suivants sont de type traçage, mais le dernier a la propriété de compacter le tas.

3.4 Comptage de références

À chaque objet du tas, le GC associe un compteur, qui correspond au nombre de pointeurs référençant cet objet. Dès qu'un pointeur est affecté, comme dans $P := Q$, le compteur de l'objet sur lequel pointe Q se voit incremented d'une unité. Réciproquement, si un pointeur est supprimé, le compteur est décrementé. Lorsqu'un compteur atteint la valeur zéro, on sait avec certitude que l'objet est inaccessible et que la mémoire qu'il occupe peut être réutilisée. Quand l'objet est réclamé, les pointeurs contenus dans l'objet sont examinés afin de décrementer également leur compteur de références.

Dans cet algorithme, la phase de détection correspond à la mise à jour des compteurs et la phase de réclamation intervient quand un compteur devient nul.

Le grand avantage du comptage de référence est sa nature incrémentale, c'est à dire que la charge supplémentaire qu'il engendre est répartie dans l'exécution du programme. Avec quelques ajustements, il convient à des systèmes temps réels.

Un petit désavantage est la place prise par les compteurs de références, en particulier, lorsqu'un compteur atteint sa valeur maximal, l'objet ne peut plus être réclamé (pour assurer la correction de l'algorithme).

Il y a deux désavantages plus importants: le problème des cycles et le problème de l'efficacité de la méthode.

3.4.1 Le problème des cycles

L'algorithme de comptage de références n'est pas complet. En effet, il ne peut pas réclamer les structures cycliques. Si les pointeurs dans un groupe créent un cycle, les compteurs de références des objets ne vaudront jamais zéro. En effet, un objet dans le cycle a un prédécesseur qui fait que son

compteur vaut au moins 1, et le prédécesseur ne pourra pas être collecté tant que l'objet qu'il réfère n'est pas collecté. Par conséquent, aucun objet appartenant à un cycle ne pourra être collecté.

On pourrait penser que des structures cycliques sont relativement rares, mais malheureusement ce n'est pas le cas. On peut citer par exemples des listes doublement chaînées ou des arbres où un noeud peut avoir accès à son père. Par conséquent, pour être complet, un algorithme de comptage de références doit être combiné à un autre GC qui peut récupérer ces cycles.

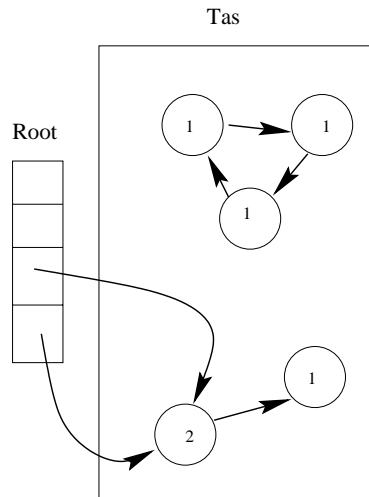


FIG. 2 – *problème des cycles*

3.4.2 Le problème de l'efficacité

Le problème du comptage de références est que le surcoût qu'il induit est proportionnel au travail effectué par le programme sous-jacent avec une constante de proportionnalité assez importante. En effet, chaque opération de copie ou de suppression de pointeur fait appel aux opérations de mise à jour des compteurs.

3.5 “Mark-and-sweep”

Comme son nom l'indique, cet algorithme procède en deux phases correspondant aux deux phases de détection et de réclamation citées précédem-

ment.

La première phase consiste à distinguer les objets inatteignables des autres (*mark*). Ceci est fait en parcourant le graphe des variables atteignables en partant des racines. Les objets sont marqués, soit en utilisant un bit à l'intérieur de l'objet, soit en les répertoriant dans une table.

La deuxième phase consiste à balayer (*sweep*) la mémoire et réclamer tous les objets qui ne sont pas marqués. Généralement, comme pour le comptage de références, ces objets sont mis dans une ou plusieurs listes d'objets qui sont accessibles aux routines d'allocation.

Il y a plusieurs problèmes majeurs associés à cette technique. Premièrement, il est difficile de gérer des objets de taille variable sans le problème de la fragmentation de la mémoire (ce problème existe aussi pour le comptage de références). Le second problème est que le coût est proportionnel à la taille du tas, comprenant à la fois les objets accessibles et les objets inaccessibles. Tous les objets accessibles doivent être marqués, et tous les objets qui ne sont plus accessibles doivent être collectés. Pour finir, contrairement au comptage de références, l'exécution du GC entraîne l'arrêt total de l'exécution du programme, ce qui peut être inacceptable pour certaines applications. Toutefois, comme tout algorithme basé sur une technique de traçage, il récupère également les objets appartenant à des cycles.

3.6 Copy collector

Contrairement au mark-and-sweep, cet algorithme compacte la mémoire dans le tas. Il regroupe la phase de détection et de réclamation en une seule phase. Il alloue les objets dans une zone de la mémoire complètement disponible et recopie les objets atteignables dans une autre zone.

L'algorithme de base divise le tas en deux parties appelées demi-espaces (A et B). L'application n'utilise que la partie A du tas. Au début de la phase de garbage collection, les deux espaces sont échangés. Le GC copie les racines dans la partie B . Il parcourt alors tous les objets accessibles et les recopie à leur tour dans la partie B . Chaque fois qu'un pointeur allant de B vers A est trouvé, le GC recopie l'objet de A dans B et modifie le pointeur pour qu'il pointe sur le nouvel objet. Le contenu de l'objet de A est modifié de façon à pointer sur la copie. Cela permet de ne pas recopier deux fois le même objet. Cet algorithme est très efficace puisqu'il défragmente le tas, ce qui optimise les performances de la mémoire virtuelle. Par ailleurs, la complexité de l'algorithme est seulement proportionnelle aux objets atteignables, et pas à la

taille du tas comme pour le mark-and-sweep.

Le gros désavantage de cet algorithme est que l'application ne peut utiliser que la moitié de la mémoire totale disponible. Par ailleurs, cet algorithme s'adapte facilement pour des systèmes temps-réel en permettant une exécution concurrente avec l'application.

3.7 Conclusion

L'algorithme de comptage de références n'est que très rarement employé dans des contextes où l'efficacité est importante. En pratique tous les algorithmes utilisent des techniques de traçage.

Le domaine des GC centralisés est un domaine de recherche encore très actif. Parmi les techniques plus avancées, on peut citer par exemple les techniques incrémentales, et les "conservative GC".

Les algorithmes présentés ici présentent le fâcheux inconvénient de bloquer l'activité du programme pendant la phase de GC, ce qui les rend inutilisables pour certaines applications comme par exemple les applications temps-réel où le temps de réponse est primordial. Toutefois, il existe de nombreux algorithmes (GC incrémentaux) qui tentent de résoudre ce problème.

La deuxième classe de GC avancés est celle des "conservative GC". Il s'agit d'algorithmes destinés à des langages *non coopératifs* tels que *C*, qui ne fournissent pas d'information sur le type d'un objet à l'exécution. Une telle information est nécessaire pour distinguer un pointeur d'une donnée classique.

4 Garbage Collectors répartis

Le but de cette partie est de présenter les principaux algorithmes de GC dans un contexte réparti et en particulier, les nouveaux problèmes qui se posent par rapport aux algorithmes centralisés. Pour une description plus détaillée, le lecteur pourra se référer à [8].

4.1 Modèle

Nous allons d'abord présenter le modèle de système répartis que nous envisageons pour décrire les différents algorithmes de GC. Ce modèle devrait être suffisamment général pour s'adapter à une grande variété de systèmes répartis.

Nous considérons un système réparti comme un ensemble d'espaces d'adressage (ou processus) qui peuvent communiquer entre eux à l'aide d'un protocole fiable. Chaque espace d'adressage peut créer des objets (un enregistrement de données et un ensemble d'opération qui peuvent être invoqués sur cet objet). L'espace d'adressage qui a créé un objet est son *propriétaire*. Un objet peut être référencé par d'autres espaces que son propriétaire. L'objet est dit *exporté* et les espaces le référençant sont appelés *espaces clients*, les références sont dites *distantes*. Les rôles de clients et de propriétaires se définissent par rapport à un objet donné : en particulier, un espace peut être client d'un objet et propriétaire d'un autre.

En pratique, la référence distante pointe sur un objet local au client (*stub* ou *proxy*) qui à son tour fait référence à un objet dans l'espace du propriétaire (*skeleton* ou *proxy serveur*). Finalement, le *skeleton* fait référence à l'objet via un pointeur local au serveur.

Les rôles des stubs et des skeletons est de rendre l'appel de méthode sur un objet distant transparent pour l'utilisateur, c'est à dire similaire (dans la mesure du possible) à un appel vers un objet local.

Comme conséquence de l'appel de méthodes à distance, les espaces peuvent échanger des références sur des objets. Sur la figure 3, un objet X envoie une référence sur Z à un objet Y . En envoyant le message contenant cette référence, l'espace A crée un *skeleton* local a . À la réception du message, l'espace B installe un *stub* b initialisé avec le pointeur distant trouvé dans le message.

Si un espace envoie à un autre une référence sur un objet distant (donc un pointeur sur un *stub* dans son espace), deux possibilités se présentent : soit

le passage se fait exactement comme précédemment en considérant le stub comme un objet particulier (voir 4.4.2) qui est alors exporté en créant un skeleton qui pointe localement sur ce stub, soit au contraire en créant dans l'espace d'arrivée un stub pointant sur l'objet distant, qui correspond à une copie du premier stub. La première solution engendre la création de chaîne d'indirection qu'il faudra supprimer, par contre elle peut faciliter la mise en place d'un GC, dans la mesure où l'envoi de référence ne concerne que deux espaces d'adressage. Sauf en 4.4.2, nous supposons que c'est la dernière solution qui est choisie. Nous ne discuterons pas ici des avantages et inconvénients de chacune des méthodes.

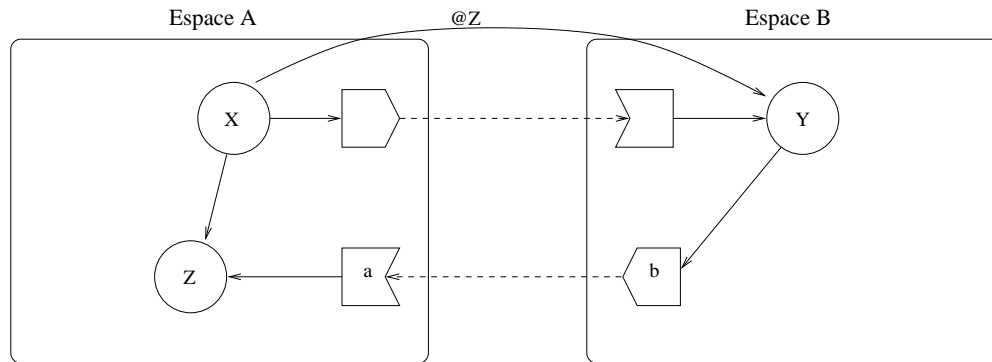


FIG. 3 – *Modèle de systèmes répartis*

4.2 Présentation du problème

La conception d'un GC réparti est un problème difficile, comme le prouvent les nombreuses solutions incomplètes proposées. Les raisons sont les problèmes de synchronisation entre les GC locaux et les problèmes de pannes qu'il faut prendre en compte.

On attend d'un GC réparti d'avoir les propriétés suivantes:

- Il devrait récupérer la mémoire occupée par tous les objets inatteignables et seulement ces objets.
- Il devrait tolérer les pannes de processus et de communications.

- Il devrait pouvoir utiliser les garbage collector locaux, et si possible, être indépendant de ces derniers.
- La présence d'un GC doit être transparente pour l'utilisateur
- La surcharge imputable au GC doit être faible, il ne doit pas y avoir trop d'envoi de messages supplémentaires ou de synchronisation entre les processus.

On retrouve les deux même groupes que pour les GC centralisés, à savoir les GC de type comptage de références, et ceux de type traçage. Comme nous l'avons déjà dit, les techniques de comptage de références sont généralement considérées comme inefficaces sur des architectures centralisées, toutefois elles sont très souvent utilisées pour des systèmes répartis. Leur principal inconvénient est de ne pas pouvoir récupérer les objets formant des cycles. À l'inverse, les algorithmes de traçage peuvent récupérer les structures cycliques mais restent inefficaces, essentiellement parce qu'ils nécessitent des contraintes fortes de synchronisation entre les différents espaces. Dans les paragraphes suivants, nous allons détailler ces différents algorithmes.

4.3 Comptage de références

L'extension naïve du comptage de références associe un compteur de référence à chaque skeleton. Le compteur est mis à jour à chaque création ou duplication de référence. Lors de la duplication d'une référence, un espace client informe le propriétaire qu'une nouvelle référence a été créée sur un objet particulier. Dans le but de garder la valeur du compteur exacte, les clients envoient à chaque duplication ou suppression de référence, un message de contrôle au propriétaire. Toutefois, les messages de contrôle doivent être délivrés dans leur ordre causal pour empêcher des problèmes d'accès concurrents. Un tel problème peut survenir quand un espace A envoie une référence sur un objet situé dans C à un espace B et la supprime juste après. A envoie un message de décrémentation à C . B envoie un message d'incrément à C . Si le message de décrémentation arrive le premier, alors le compteur correspondant pourra valoir zéro, et l'objet distant être collecté prématurément. Sur la figure 4, si aucune précaution particulière n'est prise, le message (c) peut arriver avant (b) et l'objet Z risque d'être collecté alors qu'il est encore référencé par l'espace B .

Une façon de supprimer ces accès concurrents est d'attendre des accusés de

réceptions aux messages d'incrémentation avant de supprimer un stub. Par exemple sur la figure 4, l'espace A devrait attendre la confirmation par B que C a bien eu le message d'incrémentation (il y a donc deux messages supplémentaires, un de C vers B et un de B vers A).

Malgré l'utilisation d'accusés de réception, l'extension naïve du comptage de références ne tolère pas les pannes de communication parce que les messages d'incrémentation et de décrémentation ne sont pas idempotents (c'est à dire que la valeur du compteur ne sera pas la même après l'envoi d'un ou de deux messages identiques). Les algorithmes que nous allons présenter dans la suite tentent de remédier aux problèmes de pannes et d'accès concurrents.

Cet algorithme est inefficace, toutefois il a le mérite de mettre en avant les difficultés que présentent les GC répartis.

Ses inconvénients sont les suivants:

- il ne récupère pas les cycles
- il nécessite l'envoi de beaucoup de messages
- il ne tolère pas les pannes

Rappelons ici ce que l'on entend par panne:

- terminaison anormale de processus
- panne de réseau

4.3.1 Comptage de références pondéré

L'algorithme suivant proposé par [2] est une variante très astucieuse de la méthode précédente. Le comptage de références pondéré est une amélioration de l'algorithme précédent dans la mesure où il nécessite l'envoi de moins de messages (1 message par transmission de référence) et par la même occasion, il supprime le problème de concurrence d'accès. En contrepartie, il nécessite le maintien d'un invariant.

Chaque skeleton possède deux poids, un poids total et un poids partiel. De plus, chaque stub possède un poids partiel. Le poids total reste inchangé, quelques soient les opérations effectuées, excepté les suppressions de stub. Lorsqu'un skeleton est créé, le poids total et le poids partiel sont initialisés

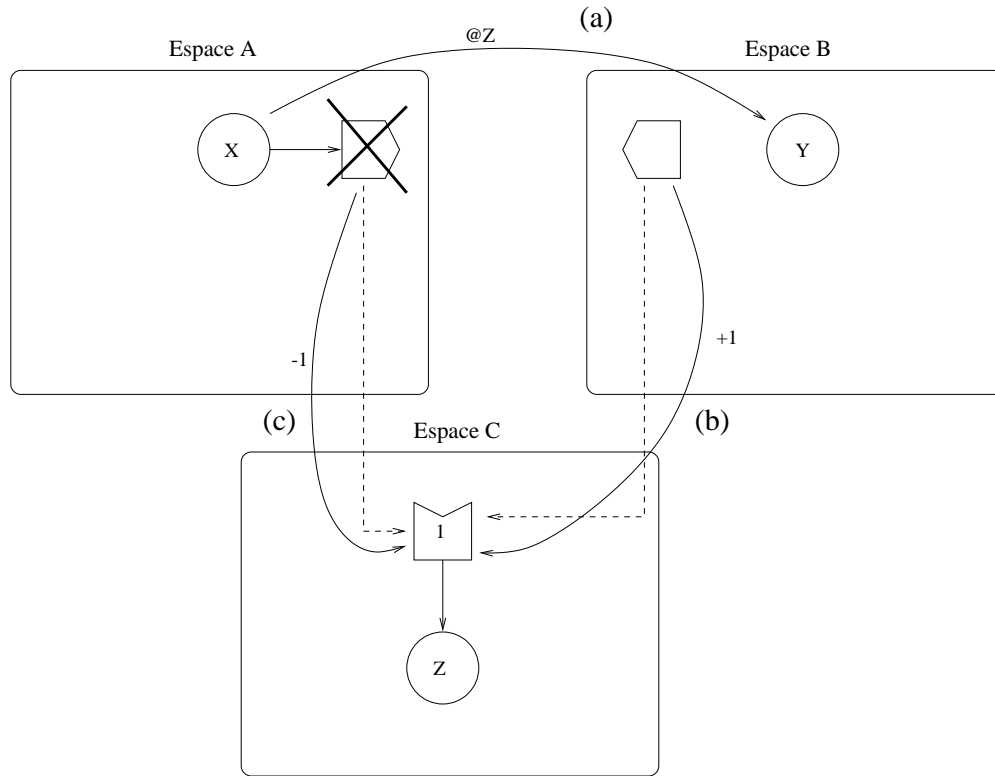


FIG. 4 – *Exemple d'accès concurrent*

à la même valeur. Chaque fois qu'une référence sur un même objet est créée, le poids partiel est divisé en deux et est envoyé avec la référence. De même lorsqu'une référence est dupliquée, le poids partiel contenu dans le stub est divisé en deux, et lorsque le stub est créé du côté client, son poids partiel est initialisé avec cette valeur. Lorsqu'un stub est supprimé, son poids partiel est soustrait au poids total du skeleton correspondant.

Comme conséquence de cette gestion des poids, l'invariant suivant est vérifié : le poids total d'un skeleton est égal à la somme des poids partiels de tous les stubs associés. On peut dire qu'un objet n'est plus référencé par des espaces clients lorsque son poids total est nul.

Cet algorithme est très efficace dans la mesure où il nécessite seulement l'envoi d'un message supplémentaire lorsqu'une référence est supprimée. Du même coup, les problèmes d'accès concurrents disparaissent complètement.

Le principal inconvénient de cette technique est que si le poids initial vaut 2^k une référence ne pourra être copiée que k fois.

De plus cette technique ne tolère pas les pannes. D'une part les pannes de réseau, essentiellement parce que les différentes opérations ne sont pas idempotentes, par exemple si un message de suppression de stub échoue mais que la mise à jour des poids a été quand même effectuée, si l'on réitère la suppression, l'objet distant pourra être réclamé trop tôt. D'autre part si un espace client termine anormalement, le propriétaire de l'objet référencé par ce client ne pourra pas connaître le poids contenu par le client et l'objet ne pourra jamais être réclamé.

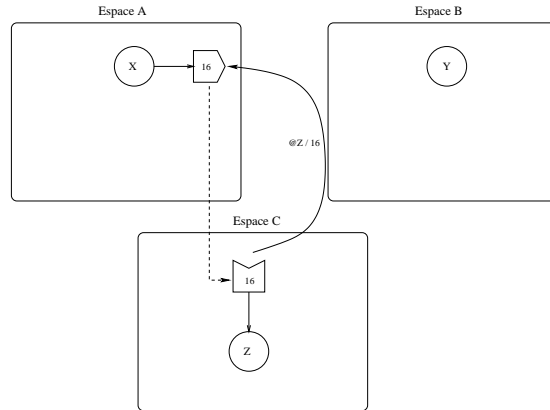
4.4 Liste de références (Reference listing)

Il s'agit d'une variante des algorithmes de comptage de références. Plutôt que d'utiliser un compteur, un objet maintient une liste des espaces qui lui font référence. Le nombre d'espaces correspond à la valeur du compteur dans le comptage de références classique. Les messages d'incrémentation et de décrémentation sont remplacés par des messages d'insertion et de suppression. L'avantage majeur de garder une liste des clients plutôt que le nombre de référence est de rendre l'envoi de message idempotent et par conséquent de pouvoir rendre l'algorithme tolérant aux pannes de réseau. Par exemple, un message de suppression peut être envoyé plusieurs fois sans conséquences sur les invariants. De la même façon, si un message de suppression a déjà été reçu, le suivant est alors simplement ignoré. En fait la propriété d'idempotence autorise le renvoi d'un message lorsque le premier envoi est susceptible d'avoir échoué.

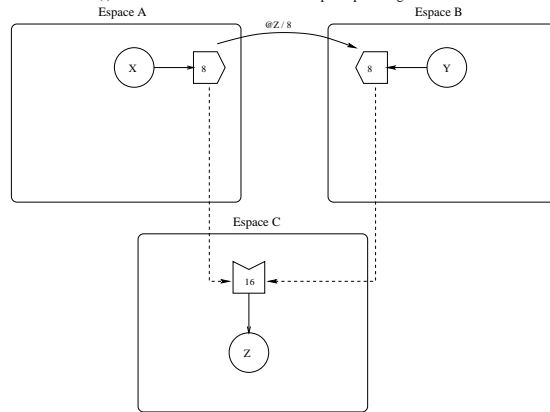
Malheureusement, comme pour le comptage de références classique, cet algorithme ne récupère pas les structures cycliques. De plus, les messages n'étant pas instantanés, on retrouve le problème d'accès concurrent vu précédemment.

4.4.1 Network Objects

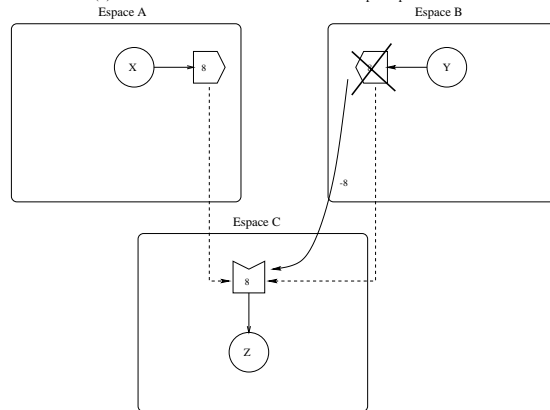
L'algorithme que nous présentons maintenant utilise la technique de liste de références. Il résout les problèmes de pannes de réseaux et de processus, ainsi que les problèmes d'accès concurrents. On pourra en trouver une description complète ainsi qu'une preuve dans [3]. En outre, une variante de cet



(i) A obtient une référence sur Z avec un poids partiel égal a 16



(ii) Retransmission de la référence vers B avec un poids partiel de 8



(iii) Suppression de la référence sur Z dans l'espace B

FIG. 5 – comptage de références pondéré

algorithme est utilisée dans Java RMI [11].

Accès concurrents Le problème d'accès concurrents (décrit plus haut) est résolu en empêchant le stub dans l'espace client d'être supprimé dans l'espace qui envoie cette référence tant que l'opération d'insertion n'a pas été confirmée par un message d'acquiescement (acknowledgment). Ceci est réalisé astucieusement en initialisant un pointeur local sur le stub. Cela n'a pas d'incidence au niveau du client, si ce n'est que le stub ne sera pas supprimé trop tôt.

Ce procédé est décrit sur la figure 6. Dans cet exemple, l'espace *A* envoie une référence sur *Z* à l'espace *B*. Juste avant, le GC réparti crée un pointeur sur le stub, ce qui empêchera ce dernier d'être collecté avant la confirmation que les différentes opérations se sont bien passées. À la réception de la référence, l'espace *B* envoie alors un message d'insertion à l'espace propriétaire de l'objet et attend l'accusé de réception de *C*. L'espace *B* renvoie alors un accusé de réception à *A* qui peut supprimer le pointeur sur le stub.

On peut noter que lorsque la référence est passée comme un argument d'appel de méthode par l'espace client, l'accusé de réception est inutile. Le retour de la méthode peut jouer ce rôle. Par contre, si l'envoi du stub correspond à un retour de procédure, l'envoi d'un message supplémentaire est nécessaire.

Pannes de réseaux Même en utilisant un protocole de transport fiable, des pannes de réseaux peuvent survenir et empêcher de mettre à jour la liste d'espaces clients. On suppose que tout appel de méthode se termine avec une indication de succès ou d'échec. Toutefois, lorsqu'une erreur est reportée, on ne sait pas si l'opération a été effectivement réalisée.

Lorsqu'un appel d'insertion échoue, aucun stub n'est créé dans l'espace client car l'espace propriétaire de l'objet peut ne pas avoir reçu le message d'insertion, par contre un appel de suppression est effectué. Cet appel n'est pas forcément nécessaire si l'insertion n'a pas eu lieu, mais dans ce cas, la suppression n'aura aucun effet.

Lorsqu'un message de suppression échoue, l'appel est simplement réitéré jusqu'à ce qu'il réussisse ou que la terminaison du serveur soit détectée.

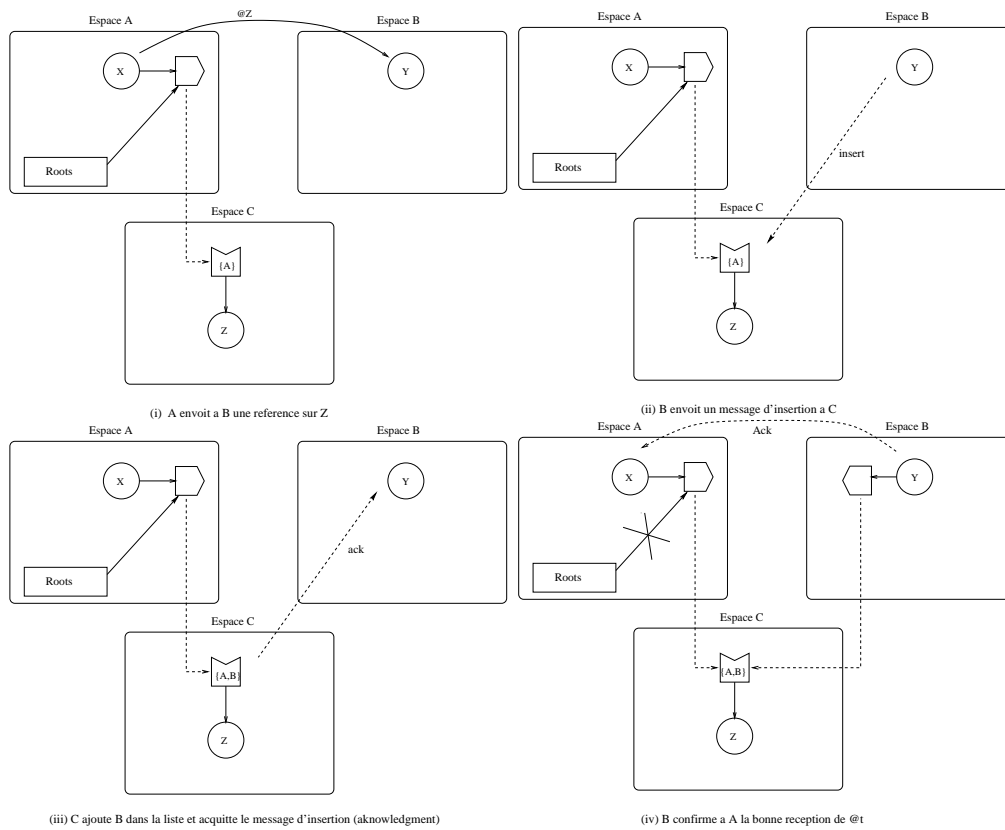


FIG. 6 – Duplication de référence dans Network Object

Terminaison des processus Les processus qui terminent anormalement ne sont pas censés prévenir les espaces propriétaires d'objets exportés pour lesquels ils possèdent des stubs. Par conséquent, un tel objet exporté ne pourra jamais être récupéré puisque dans sa liste de clients, il y aura toujours les espaces qui se sont arrêtés anormalement. Par conséquent le GC réparti doit détecter les processus qui sont arrêtés afin de gérer convenablement les listes de clients. La méthode utilisée est la suivante : le serveur teste régulièrement l'activité des clients (message de type *ping*). Si le client ne confirme pas avant un certain temps qu'il est actif, il est supprimé de toutes les listes de références. Dans Java RMI, le procédé est légèrement différent. Lorsqu'un client effectue un message d'insertion, il se voit attribué un *bail* (*lease*) qu'il doit renouveler avant expiration par un nouvel appel d'insertion. Dans les deux cas, il existe un risque de confondre une terminaison de processus avec une panne de réseau.

4.4.2 Chaînes de PPS

Les chaînes de paires souche-scion ([10]) constituent une technique pour référencer les objets dans un système réparti, très similaire au modèle décrit plus haut. Ce système propose également un service de ramasse-miette de type liste de références ayant plus ou moins les mêmes caractéristiques (aussi bien fonctionnellement qu'au niveau des performances) que celui de Network Objects. La différence importante concerne le passage en paramètre d'un stub à un objet distant, qui entraîne la création d'un nouveau skeleton (qu'ils appellent *scion*) suivi d'un *stub* dans l'espace d'arrivée. D'habitude, le stub est simplement copié d'un espace à l'autre. L'avantage de créer une chaîne d'indirection est de simplifier un peu l'algorithme du GC réparti, l'inconvénient est que cette chaîne occupe de la mémoire et augmente le temps nécessaire à l'appel de méthode distante. Toutefois, les chaînes sont court-circuitées comme effet de bord des appels de méthodes distantes. L'algorithme de GC a été récemment amélioré afin de récupérer également les cycles ([5]).

4.5 Traçage (Tracing)

Les techniques de comptage de références peuvent être efficaces et résister aux différentes pannes que l'on rencontre dans des systèmes distribués. Toutefois elles ne peuvent pas récupérer des groupes d'objets dont le graphe

de dépendance forme un cycle réparti sur plusieurs espaces d'adressage. À l'inverse les techniques de traçage sont intrinséquement cycliques mais ne résistent presque jamais aux pannes et sont souvent inefficaces. Pour de plus amples informations, le lecteur est invité à consulter [1], [6] qui fournissent les informations de base sur ce domaine.

4.6 Conclusion

La conception d'un algorithme de garbage collector réparti est une tâche difficile. La raison est que les objectifs à atteindre sont contradictoires, par exemple efficacité et tolérance aux pannes sont difficiles à atteindre simultanément. Le nombre de proposition d'algorithmes incomplets illustre cette difficulté.

L'adaptation des algorithmes centralisés n'est pas directe et conduit à des algorithmes qui ne répondent qu'à une partie du problème. Ainsi les algorithmes basés sur le comptage de références ne peuvent pas récupérer les structures cycliques. À l'inverse les méthodes de traçage peuvent récupérer les structures cycliques réparties mais ne tolèrent pas les pannes et sont souvent inefficaces.

Certains algorithmes récents sont à la fois complets et tolérants aux pannes (au moins aux pannes de réseau), on peut citer notamment [5] qui constitue une extension de [10], et [9] qui propose une amélioration de [3].

5 Jonathan

5.1 Présentation

Jonathan est une plate-forme à objets répartis (*ORB*) ouverte, dans le sens où elle permet de personnaliser les liaisons entre les différents objets en interaction. Ainsi, le développeur d'application peut agir sur le fonctionnement interne de l'ORB et le spécialiser pour implanter des nouvelles fonctionnalités (protocole, qualité de services...).

La principale caractéristique de l'architecture de l'ORB Jonathan est sa capacité d'adaptation à diverses politiques de liaison entre les objets au delà du modèle de liaison pour les modèles client-serveur des architectures standards telles que CORBA ou RMI. Cette caractéristique est réalisée en concevant un noyau d'ORB minimal dont le rôle est de fournir un environnement générique que des *usines à liaisons* (binding factories) peuvent utiliser pour créer et gérer des liaisons spécifiques. Cette approche permet de créer par exemple un ORB conforme aux spécifications RMI (jérémie) comme une personnalité particulière de Jonathan.

L'ORB minimal Jonathan introduit la notion d'*objet de liaison*. Deux objets *A* et *B* peuvent interagir de deux façons différentes : soit *A* invoque directement une méthode de *B* (les objets sont situés alors dans le même espace d'adressage), soit il invoque la même opération sur un objet de liaison dont le rôle est de transmettre l'invocation à *B* et de retourner le résultat si nécessaire (les objets sont dans deux espaces d'adressage différents) (figure 7). Les objets de liaisons ne correspondent pas à des objets au sens de Java, mais plutôt à un ensemble de tels objets répartis entre plusieurs espaces d'adressage. Ils représentent la liaison complète entre deux objets utilisateur, incluant les ressources de communication, le protocole, les talons (stub et skeleton). Les liaisons sont construites par des entités particulières : les usines à liaison dont le rôle est de créer et de gérer les données de liaison).

5.2 La personnalité Jérémie

La personnalité Jérémie est construite à partir d'une usine à liaison spécifique, en définissant certains éléments de l'objet de liaison pour s'adapter aux spécifications RMI. On y reconnaît les différentes couches de l'architecture RMI, avec le protocole de session propriétaire JIOP de Jérémie.

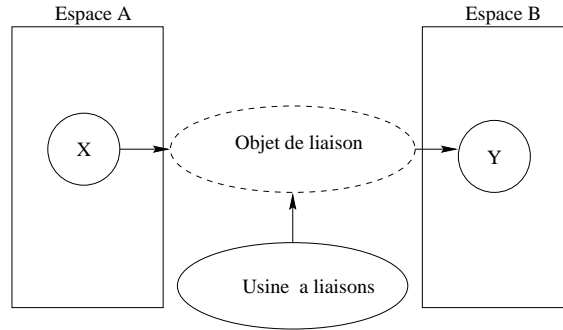


FIG. 7 – Une liaison point à point

5.2.1 Exemple d'utilisation

Une application répartie construite à l'aide de Jérémie (ou encore Java RMI) est composée d'objets exportés (accessibles à distance) et d'objets clients (qui accèdent aux objets exportés). La manière la plus commune d'exporter un objet est de le faire dériver de la classe `org.objectweb.jeremie.lib.contexts.moa.UnicastRemoteObject`. L'objet doit également implémenter une interface définie par l'utilisateur qui contient les méthodes de l'objet accessible à distance. Cette interface doit dériver de `java.rmi.Remote`. Elle est commune au client et au serveur et permet d'assurer le typage correct des objets exportés chez les différents clients. Finalement, un objet client appelle les méthodes de manière transparente sur un *stub* qui se substitue à l'objet distant et qui se charge de transmettre ces appels. La classe du stub est générée automatiquement à partir du fichier source.

L'exemple suivant définit un objet client qui invoque une méthode sur un objet serveur. Le méthode ne fait que renvoyer une chaîne de caractères.

Interface commune au client et au serveur définissant les méthodes invocables à distance.

```

/* interface definissant les methodes invocables a distance par
 * le client
 */
public interface Hello extends java.rmi.Remote {
    public String sayHello() throws java.rmi.RemoteException ;
  
```

```
}
```

Côté client:

```
import java.rmi.RMISecurityManager;
import org.objectweb.jeremie.libs.services.registry.Naming;

public class Client {
    public static void main(String args[]) {
        try {

            String registryHost = "";
            if (args.length != 0) {
                registryHost = args[0];
            }

            /*
             * Cet appel permet de recuperer une reference
             * sur l'objet distant
             */
            Hello obj = (Hello) Naming.lookup("jrm://"+
                + registryHost + "/helloobj");
            System.out.println();
            System.out.println(obj.sayHello());

        } catch (Exception e) {
            System.err.println("Hello Client exception");
            e.printStackTrace();
        }
    }
}
```

Côté Serveur:

```

import java.rmi.RemoteException;
import jeremie.libs.contexts.moa.UnicastRemoteObject;
import jeremie.libs.services.registry.Naming;

class HelloImpl extends UnicastRemoteObject
    implements Hello {

    HelloImpl() throws RemoteException {
    }

    public String sayHello() {
        return "Hello World!";
    }
}

/*
 * La classe Server ne contient qu'une methode main pour
 * demarrer l'objet serveur
 */
public class Server {
    public static void main (String[] args) {
        try {
            String registryHost = "";
            if (args.length != 0) {
                registryHost = args[0];
            }

            /*
             * cet appel permet d'enregistrer l'objet serveur
             * dans un service de nommage
             */
            Naming.rebind("jrmi://"
                + registryHost + "/helloobj", new HelloImpl());

            System.out.println("Hello Server ready");
        }
    }
}

```

```
    } catch (Exception e) {  
        System.err.println("Hello Server exception");  
        e.printStackTrace();  
    }  
}  
}
```

6 Implantation

Cette partie détaille l'implémentation d'un GC pour la personnalité Jérémie de Jonathan. L'algorithme utilisé est inspiré de l'article Network Objects décrit dans la partie 4.4.1.

Cette implémentation présente de nombreuses difficultés, autant dans le caractère plutôt complexe de l'algorithme que dans l'interaction avec Jonathan. En fait, la compréhension des sources de Jonathan a représenté un travail important. Par ailleurs, lors de l'implémentation de l'algorithme, une contrainte était de modifier le moins possible les sources déjà existants. J'ai choisi cet algorithme pour les raisons suivantes:

- c'est un algorithme efficace qui remplit la plupart des fonctions qu'on attend de lui, excepté la récupération des structures cycliques.
- la plupart des algorithmes qui récupèrent les cycles sont beaucoup plus compliqués et peu sont effectivement implémentés. Cette difficulté est accrue par la contrainte de baser cette implantation sur la plate-forme Jonathan.
- le fait d'avoir déjà été implanté dans des systèmes industriels comme Java RMI, lui confère une certaine légitimité, en particulier sur la faisabilité d'une implémentation dans les limites de mon stage.

La question s'est posée d'implémenter la méthode de [10], en raison notamment de l'amélioration récente ([5]) qui propose un algorithme performant. Finalement, l'algorithme de 4.4.1 semblait plus en adéquation avec le modèle de Jérémie.

Les nouvelles classes se situent dans le package `org.objectweb.jeremie.-libs.dgc` (sauf indication contraire, toutes les classes décrites dans ce chapitre se trouvent dans ce package). Pour simplifier le travail, j'ai séparé la réalisation de l'algorithme en 3 phases successives. La première partie consiste simplement à mettre en place les structures de données qui permettent d'associer à chaque objet exporté la liste des machines virtuelles clientes. Il n'y a pas de récupération automatique de mémoire à ce niveau là. La deuxième partie consiste à faire le lien entre la partie précédente et le GC local de java. Le rôle de la dernière partie est de prendre en compte les pannes potentielles du réseau et les terminaisons anormales de processus. Les deux chapitres suivants décrivent les deux premières étapes. Le troisième chapitre concerne les modifications apportées à Jérémie.

6.1 Mise en place des listes de références

6.1.1 Partie serveur

Dans un premier temps, on veut pouvoir associer à chaque objet exporté une liste des machines virtuelles clientes. Pour cela, on utilise une table dans chaque JVM qui répertorie les objets exportés avec comme information supplémentaire une liste de VMID (un identificateur de JVM, cette classe est décrite dans l'annexe B).

Pour gérer cette association, on utilise principalement deux classes, la classe `ObjectTable` et la classe `Target`.

Target La classe `Target` contient un identificateur d'objet `ObjId`, une référence sur l'objet exporté et un tableau de VMID. (`ObjId` est la classe des identificateurs d'objet. Elle permet d'identifier des objets exportés de manière unique dans différentes JVM. On trouvera une description dans l'annexe B). Les méthodes `referenced(VMID vmid)` et `unreferenced(VMID vmid)` de la classe permettent d'ajouter ou d'enlever un identificateur de JVM de la liste.

ObjectTable La classe `ObjectTable` est une classe dont toutes les méthodes et variables sont statiques. Elle permet de gérer une table qui associe des `ObjID` à des `Target`. Les méthodes de cette classe sont `removeTarget(Target target)` et `putTarget(Target target)` qui permettent de supprimer ou d'insérer l'objet `target` dans la table, `referenced(Object oid, VMID vmid)` et `unreferenced(Object oid, VMID vmid)` qui permettent d'ajouter ou de supprimer `vmid` dans l'objet de classe `Target` d'identificateur `oid` par appel aux méthodes de `Target` de même nom. Les objets exportés sont insérés dans la table de `ObjectTable` au moment de leur exportation dans la classe `UnicastRemoteObject`.

Les objets `Target` sont mis à jour par la classe `DGCImpl`. Cette classe permet de faire la jonction entre la partie client et serveur de l'algorithme. En effet la classe `DGCImpl` permet de définir un objet distant classique. Les méthodes distantes qu'il implémente sont définies dans l'interface `DGC`. Les méthodes `dirty` et `clean` correspondent respectivement aux messages d'insertion et de suppression d'espace client dans la liste de références d'un objet.

```
public interface DGC extends Remote {
```

```

public void dirty(VMID vmid, Object oid)
    throws RemoteException ;

public void clean(VMID vmid, Object oid)
    throws RemoteException ;
}

```

On instancie un objet de cette classe dans chaque JVM possédant un objet exporté. Pour cela on rajoute une variable statique de classe `DGC` dans `UnicastRemoteObject`. Cette variable est initialisée au premier appel de la méthode d'exportation.

L'implantation de ces methodes se fait directement compte tenu des spécifications des classes définies précédemment.

```

public void dirty(VMID vmid, Object oid)
    throws RemoteException {
    ObjectTable.referenced(oid, vmid) ;
}

public void clean(VMID vmid, Object oid)
    throws RemoteException {
    ObjectTable.unreferenced(oid, vmid) ;
}

```

Il subsiste une dernière difficulté: la partie client doit pouvoir localiser les objets de classe `DGCImpl` sur lesquels elle doit instancier les méthodes `dirty` et `clean`. À partir de chaque référence entrante dans une JVM, on doit pouvoir localiser (c'est à dire créer un stub) l'objet `DGCImpl` de la JVM référencée.

La méthode choisie dans un premier temps est d'ajouter dans la classe `RefImpl` une variable de classe `DGC` qu'on sérialise avec la référence.

6.1.2 Partie client

La partie client se résume essentiellement en la classe `DGCClient`.

```

public class DGCClient {

    // l'identificateur unique de JVM

```

```

public static VMID vmid = new VMID() ;

// point d'accès unique a cette classe
public static void registerRefs(RefImpl ref) ;

}

```

Le point d'accès à cette classe est la méthode `registerRefs`. Elle est appelée par chaque référence entrante dans la JVM. Plus précisément, l'appel de cette méthode se fait dans la méthode `readExternal` de la classe `refImpl` au moment du *demarshalling* de la référence entrante. C'est la classe `DGCClient` qui appelle les méthodes `dirty` et `clean`.

6.2 Interaction avec le GC local

Il y a deux types d'interaction entre le GC local et le GC réparti selon qu'on se place du côté du client ou du côté du serveur. Du côté du client, il s'agit d'informer le GC réparti qu'un objet distant n'est plus référencé. Du côté du serveur, il s'agit de supprimer les objets qui ne sont plus référencés par des JVM clientes (la liste de référence est vide) et qui ne sont pas non plus accessibles localement. Ces deux problèmes nécessitent une coopération avec le GC de la machine virtuelle.

Le langage Java permet une telle coopération par l'intermédiaire de références faibles que l'on peut trouver dans le package `java.lang.ref`. Ce package est décrit dans l'annexe A.

6.2.1 Côté client

Pour prévenir le serveur qu'un client n'a pas de référence sur un objet distant, il faut garder une trace chez le client de toutes les références qu'il possède sur un objet chez le serveur (caractérisé par son `ObjID`). Ces références correspondent à des stubs ou plus précisément des objets de classe `RefImpl`. Rappelons qu'un objet de cette classe permet d'acheminer un appel distant vers l'objet distant et que chaque stub contient un tel objet. Il contient en particulier toutes les informations pour le localiser ainsi que son identificateur (`ObjID`).

En pratique, chaque fois qu'un objet `RefImpl` entre dans la JVM client, il

faut le répertoire de manière à maintenir pour chaque objet serveur la liste des `RefImpl` associés. En effet, une JVM cliente peut contenir plusieurs objets `RefImpl` correspondant à un même objet distant. Ensuite, chaque fois qu'une `RefImpl` est collectée par le GC local, on la supprime (pour ne pas altérer le fonctionnement du GC local, on utilise des références faibles dans la liste) de la liste. Une fois que la liste associée à un `ObjID` est vide, on peut effectuer l'appel `clean` correspondant.

6.2.2 Côté serveur

Il s'agit ici de libérer la mémoire occupée par les objets exportés lorsque ces derniers ne sont plus accessibles localement, et que leur liste de référence est vide. En Java, la libération de la mémoire se fait automatiquement lorsque l'objet n'est plus référencé. On veut donc faire en sorte que:

- lorsque la liste de référence n'est pas vide, l'objet ne puisse pas être collecté.
- lorsque la liste de référence est vide, l'objet doit être collecté (ou encore non référencé localement) si et seulement si l'application (qui ne comprend pas le GC) ne possède plus de référence locale sur l'objet.

Le problème est que le runtime de Jérémie référence les objets exportés dans plusieurs tables, de même pour la `ObjectTable` du DGC.

La solution est la suivante: on remplace toutes les références sur l'objet exporté dans les classes de Jérémie par des références faibles, ce qui ne demande presque aucune modification. Toutefois, cela ne suffit pas, puisqu'un objet serait collecté dès qu'il n'est plus accessible localement. On modifie alors la façon dont l'objet est référencé dans la classe `ObjectTable` en utilisant la classe `LessWeakRef`.

```
public class LessWeakRef extends WeakReference {
    private Object trueRef = null ; ;

    public LessWeakRef(Object obj) {
        super(obj) ;
        setStrong() ;
    }
}
```

```

public void setStrong() {
    trueRef = this.get() ;
}

public void setWeak() {
    trueRef = null ;
}
}

```

Cette classe, permet à l'aide des méthodes `setWeak` et `setStrong` de rendre la référence faible ou normale. Dans ce dernier cas, l'objet ne peut donc pas être collecté. En conclusion, toutes les références dans les classes de Jérémie sont faibles, et celles de `ObjectTable` sont soit normales, soit faibles selon que la liste de references est vide ou pleine. Si elle est faible, l'objet sera collecté lorsqu'il n'est plus accessible localement.

6.2.3 Modifications dans Jérémie

Il y a finalement assez peu de modifications dans les classes de Jérémie. En plus de ce qui a déjà été décrit, les classes `RequestSession` et `StdStubFactory` ont été légèrement modifiés de telle sorte que les références sur les objets exportés soient faibles.

6.3 Conclusion

Cette implantation est très proche de l'algorithme décrit en 4.4.1. Il s'agit d'un premier prototype qui fournit les bases pour un algorithme plus efficace. En particulier, il reste à implémenter les aspects de tolérance aux pannes, mais cela ne devrait pas poser de problèmes majeurs, il s'agit essentiellement de réitérer les appels aux méthodes `dirty` et `clean`. Un dernier point à traiter est celui des accès concurrents qui nécessitent une petite modification du protocole utilisé dans les invocations distantes afin de permettre l'envoi d'un message d'accusé de réception.

L'étape suivante est d'envisager une évolution de cet algorithme pour la récupération des cycles. Un algorithme étendant celui utilisé ici et permettant la récupération des cycles est donné dans [9].

7 Bilan

7.1 Aspects liés à l'organisation

J'ai organisé mon travail de la façon suivante : les premiers mois à temps partiel ont été consacré à l'étude bibliographique. C'est à dire, la lecture d'articles concernant les techniques de garbage collector. La période à temps plein (à partir de mi-mars) s'est essentiellement divisée en trois parties. Premièrement il a fallu comprendre le fonctionnement de Jonathan, essentiellement par l'étude des sources (malheureusement, peu de documentation était disponible à ce moment). Ensuite il a fallu choisir un algorithme de GC à implanter. La dernière phase était celle de l'implémentation.

7.2 Bilan sur le projet

Ce stage a été intéressant dans la mesure où il comportait un bon compromis entre la théorie et la pratique. En particulier, l'implémentation d'un GC réparti constitue une bonne expérience dans le domaine de la programmation répartie puisqu'il présente toutes les caractéristiques des algorithmes répartis (problèmes de synchronisation, de panne etc...).

Les difficultés rencontrées ont été également d'ordre pratique et théorique. D'une part la compréhension des algorithmes, mais surtout la compréhension de Jonathan et l'implémentation d'un service pour Jérémie.

L'algorithme implémenté pour l'instant n'est qu'un prototype, il aurait été intéressant d'implanter un algorithme plus performant, notamment permettant la récupération des cycles.

7.3 Remerciements

Je tiens à remercier sincèrement le département ASR de m'avoir accueilli pour ce stage, et en particulier Kathleen Milsted et Pascal Déchamboux pour leur aide et leur disponibilité.

A Références faibles

Le package `java.lang.ref` du jdk 1.2 contient des classes pour définir des références sur des objets avec une interaction limitée avec le garbage collector. Par exemple, un programme peut avoir une référence sur un objet sans pour autant empêcher ce dernier d'être récupéré par le garbage collector de la JVM. De plus, à leur création, ces références peuvent s'enregistrer dans un objet de classe `ReferenceQueue`. Ils sont alors insérés dans une file (accessible via cet objet) lorsque le GC détecte un changement d'atteignabilité de l'objet faiblement référencé (par exemple, s'il n'est plus accessible par des références classiques).

Ces classes nous seront utiles pour l'implémentation d'un garbage collector réparti, en fournissant une interface entre le GC local et le GC réparti. Nous décrivons ici les classes intéressantes de ce package pour la réalisation d'un GC réparti.

A.1 Reference

La classe `Reference` est une classe abstraite dont dérivent les différents types de références faibles (en particulier `WeakReference`, `PhantomReference`). Un objet de classe `Reference` contient une référence sur un objet quelconque. La méthode principale de la classe `Reference` est `get` pour obtenir la référence avec laquelle l'objet a été créé.

A.2 ReferenceQueue

Cette classe gère une file d'attente dans laquelle sont insérés les objets après leur changement de classe d'atteignabilité. Les méthodes principales de cette classe sont `poll` et `remove` qui permettent de vérifier si la file d'attente contient un objet de classe `Reference`. La différence est que `remove` est bloquant alors que `poll` retourne `null` si aucune référence faible n'est disponible.

A.3 WeakReference et PhantomReference

Les deux sous-classes qui nous sont utiles sont `WeakReference` (références faibles) et `PhantomReference` (références fantômes). Les références faibles n'empêchent pas l'objet pointé d'être collecté. Elles

sont utilisées par exemple pour faire des dictionnaires dans lesquels les objets peuvent être récupérés quand ils ne sont plus accessibles autrement que par ce dictionnaire.

B VMID et ObjID

Dans les tables utilisées dans l’algorithme de “reference listing”, on doit associer à un objet exporté (c’est-à-dire accessible à distance) la liste des machines virtuelles qui possèdent des références sur cet objet. On a besoin d’une classe d’identificateur qui permette de caractériser de manière unique une JVM. De la même façon, on a besoin également d’identificateurs d’objets. Les classes `VMID` et `ObjID` remplissent ces fonctions.

B.1 VMID

```
package java.rmi.dgc ;

public final class VMID implements java.io.Serializable {

    public VMID();

    public static boolean isUnique();

    public int hashCode() ;

    public boolean equals(Object obj) ;

    public String toString() ;

}
```

Cette classe assure que deux objets de classe `VMID` instanciés dans deux JVM différentes, seront toujours différents. Toutefois, une condition pour garantir cette propriété est que l’hôte puisse connaître son adresse IP.

B.2 ObjID

La classe `ObjID` est une classe interne à la classe `RefImpl` de jérémie. Chaque référence contient un objet `ObjID` initialisé à sa création. Ainsi, il est possible d’identifier de manière unique un objet exporté dans un ensemble

de JVM. En particulier, on peut savoir si deux stubs font référence au même objet.

Références

- [1] Rivka Ladin Barbara Liskov. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Fifth ACM Symposium on the Principles of Distributed Computing*, pages 29–39, 1986.
- [2] David I. Bevan. Distributed garbage collection using reference counting. pages 117–187. Bakker et al., 1987.
- [3] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In Barbara Liskov, editor, *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 217–230, New York, NY, USA, December 1993. ACM Press.
- [4] Bruno Dumant, François Horn, Frédéric Dang Tran, and Jean-Bernard Stefani. Jonathan: an open distributed processing environment in java. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 175–190, London, September 1998. Springer-Verlag.
- [5] Fabrice Le Fessant, Ian Piumarta, and Marc Shapiro. An implementation for complete asynchronous distributed garbage collection. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Montreal, June 1998. ACM Press.
- [6] John Hugues. A distributed garbage collection algorithm. In ACM Conference on Functional Programming Languages and Computer Architecture, editors, *Tenth International Workshop on Distributed Algorithms WDAG'96*, number 201 in Lecture Notes in Computer Science, pages 256–272, Nancy, France, October 1985. Springer-Verlag.
- [7] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*. Wiley, New York, NY, USA, 1996. Reprinted in 1999 with improved index, and corrected errata.
- [8] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Second Closed BROADCAST Workshop*, Bruxelles (Belgique), November 1994. Broadcast Basic Research Action.

- [9] Helena C. C. D. Rodrigues and Richard E. Jones. A cyclic distributed garbage collector for Network Objects. In Ozalp Babaoglu and Keith Marzullo, editors, *Tenth International Workshop on Distributed Algorithms WDAG'96*, number 1151 in Lecture Notes in Computer Science, pages 123–140, Bologna, Italy, October 1996. Springer.
- [10] Marc Shapiro, Peter Dickman, and David Plainfossé. Robust, distributed references and acyclic garbage collection. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*, pages 135–146, 10–12 August 1992.
- [11] Sun. Java remote method invocation specification. Technical report, Sun Microsystems, 1998.

Table des matières

1	Introduction	1
2	France Télécom R&D	1
3	Garbage Collector centralisés	2
3.1	Principes	2
3.2	Motivations	2
3.3	Principaux algorithmes	3
3.4	Comptage de références	4
3.4.1	Le problème des cycles	4
3.4.2	Le problème de l'efficacité	5
3.5	“Mark-and-sweep”	5
3.6	Copy collector	6
3.7	Conclusion	7
4	Garbage Collectors répartis	8
4.1	Modèle	8
4.2	Présentation du problème	9
4.3	Comptage de références	10
4.3.1	Comptage de références pondéré	11
4.4	Liste de références (Reference listing)	13
4.4.1	Network Objects	13
4.4.2	Chaînes de PPS	17
4.5	Traçage (Tracing)	17
4.6	Conclusion	18
5	Jonathan	19
5.1	Présentation	19
5.2	La personnalité Jérémie	19
5.2.1	Exemple d'utilisation	20
6	Implantation	24
6.1	Mise en place des listes de références	25
6.1.1	Partie serveur	25
6.1.2	Partie client	26
6.2	Interaction avec le GC local	27
6.2.1	Côté client	27

6.2.2	Côté serveur	28
6.2.3	Modifications dans Jérémie	29
6.3	Conclusion	29
7	Bilan	30
7.1	Aspects liés à l'organisation	30
7.2	Bilan sur le projet	30
7.3	Remerciements	30
A	Références faibles	31
A.1	Reference	31
A.2	ReferenceQueue	31
A.3	WeakReference et PhantomReference	31
B	VMID et ObjID	33
B.1	VMID	33
B.2	ObjID	33