# Adicionando um Mecanismo de Coleta de Lixo a um Middleware de Objetos Persistentes e Distribuídos

Rafael Ricardo Paiva Freitas
rfreitass@yahoo.com.br

Ricardo da Silva Pereira
rspereira78@gmail.com

Maria Alice Silveira de Brito
malice@ime.uerj.br

Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro
Rua São Francisco Xavier, 524 / sala 6020-B, CEP: 20.550 – 013, Rio de Janeiro, RJ, Brasil

**Resumo**

*Neste trabalho, implementamos e incorporamos um mecanismo de coleta de lixo (garbage collection) a um middleware, que administra objetos distribuídos e persistentes, denominado Shallow Persistent RMI (SPRMI) [1]. Esse middleware utiliza um armazém para os objetos persistentes, que lá permanecem após o término da execução da aplicação da qual eles tenham participado. Esse armazém necessita de um mecanismo, para detecção e remoção de objetos que se tornam inacessíveis, em outras palavras, tornam-se lixo. Entre os trabalhos relacionados, em [2], há uma discussão, que aponta os problemas enfrentados em coleta de lixo com distribuição envolvida. Nessa discussão, também são apresentadas as características, que incorporadas a um mecanismo de coleta de lixo, podem solucionar os problemas apontados. Um estudo dessas características para o nosso mecanismo, tanto conduziu as soluções em nosso trabalho, como também permitiu a sua classificação entre os trabalhos existentes nesse contexto. Mostramos uma visão geral do SPRMI e de como o mecanismo precisou ser tratado para que a coleta de lixo pudesse ser incorporada a esse middleware.*

**Abstract**

*In this work, we have implemented and added a mechanism for garbage collection to a middleware, which manages distributed and persistent objects, called Shallow Persistent RMI (SPRMI) [1]. This middleware uses a store for persistent objects, which remain there after the end of the execution in which they have participated. This store needs a mechanism for detection and removal of objects that become inaccessible, in other words, they become garbage. Among the related works, in [2], there is a discussion that points out the problems faced in distributed garbage collection. This discussion also presents characteristics, that if incorporated into a mechanism for garbage collection, can solve the problems raised. A study of these features regarding our mechanism, led to solutions in our work, but also allowed its classification among the existing work in this context. We show an overview of SPRMI and how the mechanism needed to be treated so that garbage collection could be incorporated into the middleware.*

**Keywords**

Java RMI, Persistence, Distributed Objects, Proxy.

**Palavras-chave**

Java RMI, Persistência, Objetos Distribuídos, Proxy.

## 1 Introdução

Muitas linguagens de programação provêm coleta de lixo a fim de desalocar automaticamente objetos inacessíveis. Essa automatização libera o programador da responsabilidade de desalocar memória, além de evitar erros por falta de memória, que são difíceis de detectar e reparar.

Com sua importância já bem fundamentada, no gerenciamento local de memória, é natural obter benefícios dos coletores de lixo também em ambientes distribuídos. As motivações para tal serviço são inúmeras. Sistemas distribuídos modernos provêm invocação de métodos a objetos locais e remotos, de modo uniforme e transparente. Tais sistemas também devem prover transparência, ao programador, dos aspectos de alocação, na gerência de objetos, o que pode ser alcançado através de um coletor de lixo distribuído. Além disso, a gerência de memória é uma tarefa que não deve ser da responsabilidade do programador.

Os objetos de dados que estamos tratando têm persistência. Esse recurso possibilita que objetos sobrevivam depois do término da execução da aplicação da qual eles tenham participado. Na persistência de um objeto, seu estado deve ser gravado em memória secundária, tal como disco, para depois ser recuperado em outra execução. Se durante uma execução, um considerável número de objetos pode ser persistido, é razoável que haja uma verificação daqueles que não

sejam mais alcançados por qualquer programa de aplicação e estejam gravados, para que sejam removidos do armazém.

Nossa motivação neste trabalho vem da possibilidade de surgir lixo no armazém de objetos controlados pelo middleware SPRMI, que implementa objetos distribuídos e persistentes. Esse armazém necessita de um mecanismo, para detecção e remoção dos objetos que se tornam inacessíveis.

Desde que objetos persistentes possam permanecer depois que seus clientes e servidores terminem, objetos que viraram lixo poderiam ficar no armazém da persistência para sempre, a menos que a coleta de lixo venha a ser efetuada. Quando a coleta de lixo é deixada ao encargo de programadores, o mesmo problema pode ocorrer, porque esse tipo de coleta requer a cooperação do programador em desenvolver um código bem comportado. Se alguns programas terminam sem o procedimento de remover suas referências, a memória permanece alocada, podendo causar inclusive falha no sistema.

Nosso objetivo é incorporar um mecanismo de coleta de lixo no armazém de objetos do SPRMI. Esse middleware é oferecido em forma de biblioteca, melhor dizendo, um *package* Java. Com esse mecanismo, pode ser evitada a acumulação de lixo no armazém de objetos.

Cabe notar, neste ponto, o fato de que os objetos controlados pelo SPRMI, quando instanciados, também contarão com os mecanismos de gerência de heap originais de Java, sendo, então, suas células recicladas da mesma forma que os objetos não controlados por SPRMI. Isso soa estranho, mas faz parte do controle de alocação da linguagem Java. O SPRMI controla a distribuição e persistência, sendo o enfoque neste trabalho, o mecanismo de Coleta de lixo destinado aos objetos persistentes que não são mais acessíveis, no armazém de objetos do SPRMI.

Todos os outros objetos, a seguir, estão fora do controle do SPRMI: a) de tipo *user-defined* mas não criados via SPRMI, b) os de tipos primitivos e c) *array*s, caso sejam acessados remotamente, serão tratados por Java RMI's *Distributed Garbage Collection* [3].

Recapitulando um mecanismo de coleta de lixo comumente é incorporado à gerência da memória *heap*, na qual encontram-se as células de memória destinadas às variáveis que podem ser alocadas dinamicamente, durante a execução de um programa. Neste trabalho, o mecanismo de coleta de lixo desenvolvido foi incorporado a um *middleware* que gerencia objetos persistentes e distribuídos, mais especificamente, ao seu módulo denominado *broker*. Esse módulo possui um armazém de objetos e cada objeto reside em apenas um armazém. Esses *broker*s comunicam-se entre si, pelo ambiente distribuído, resolvendo a entrega e retorno de cada mensagem, oriunda da invocação de um objeto emissor a um objeto alvo.

Quando um objeto persistente não é mais referenciado ele torna-se lixo, devendo ser removido do armazém.

Esse funcionamento será explicado em maiores detalhes, nas seções à frente, como podemos ver pela seguinte organização do trabalho: na seção 2, são apresentados os fundamentos básicos relacionados à coleta de lixo local e distribuída, as técnicas mais conhecidas, e uma discussão [2], que aponta os problemas enfrentados em coleta de lixo com distribuição envolvida, além da apresentação das características, que incorporadas a um mecanismo de coleta de lixo, podem solucionar os problemas apontados e que foram aproveitadas em nosso trabalho. Na seção 3, há uma breve introdução da estrutura do *middlewar*e SPRMI e das intervenções necessárias para a incorporação das propriedades apresentadas em [2] ao nosso mecanismo de coleta de lixo bem como uma breve descrição de sua implementação. Finalmente, na seção 4, encontram-se as conclusões.

## 2 Fundamentos e Trabalhos Relacionados

A coleta de lixo é um mecanismo que colabora com a gerência da memória *heap*, cujas células abrigam os objetos de dados, que são alocados dinamicamente. Um objeto é considerado lixo, se não é mais alcançável por algum programa ou, no caso de persistência ortogonal [4] por um objeto considerado raiz. Nessa condição, as células ocupadas por ele devem ser recuperadas e devolvidas à gerência da heap, que as tornará disponíveis, para que sejam reusadas por outra solicitação de memória. Esse mecanismo costuma ser incorporado ao código gerado por compiladores da maioria das linguagens de programação.

As duas fundamentais abordagens sobre coleta de lixo – varredura e marcação (*tracing*) [5] e a contagem de referências (*reference-counting*) [6] – foram apresentadas, coincidentemente, em 1960.

Na técnica de varredura e marcação, o conjunto de objetos de dados que podem ser acessados por um programa cliente é determinado percorrendo-se o caminho formado pelas referências e apontadores para objetos (o grafo de conectividade). Como ponto de partida, são usados todos os pontos de acesso do programa cliente para células de memória *heap*. Esse conjunto de pontos é denominado conjunto-raiz e pode incluir variáveis locais, parâmetros, membros de estrutura e endereços armazenados em registradores. A princípio, todos os objetos de dados alocados em células da memória *heap* que não pertencem ao grafo de conectividade não podem mais ser acessados pelo

programa cliente e, portanto considerados como lixo. Dessa forma, as células que os abrigam podem ser reusadas por outros objetos de dados a serem alocados posteriormente. Nessa condição, uma célula é registrada como disponível numa lista que o mecanismo de coleta de lixo utiliza para esse controle.

Na contagem de referências (*reference-counting*), cada objeto de dado, alocado na célula da memória *heap*, tem associado um contador, que indica o número de referências a esse objeto. Se a contagem de referências atinge zero, significa que o objeto de dado não é mais referenciado, tornando-se lixo e podendo então ser coletado.

Desde o surgimento do conceito de coleta de lixo, foram dedicados numerosos esforços aos dois paradigmas. Desses esforços, resultaram novas abordagens, consideradas refinamentos das duas originais, segundo os autores, em [7], cujo artigo traz uma classificação das técnicas de coleta de lixo, que apenas enumeramos pela sua identificação e referência nos dois parágrafos a seguir.

Para o método de varredura e marcação, as abordagens de refinamento mais importantes se intitulam como a seguir: *interative copying collection* [8], *generational collection* [9,10], *constant-space tracing* [11], *barrier optimization techniques* [12, 13, 14], *soft real-time collection* [15, 16, 17, 18, 19, 20, 21], *hard real-time collection* [22], e *multiprocessor concurrent collection* [23, 24, 25, 26, 27].

Para a abordagem de contagem de referências, as mais destacadas se intitulam, como a seguir: *incremental freeing* [28], *deferred reference counting* [29], *cycle collection* [30, 31, 32], *compile-time removal of counting operation* [33], e *multiprocessors concurrent collection* [34, 35, 36]**.**

O aspecto de distribuição no mecanismo de coleta de lixo o torna mais complexo do que o adotado em programação local, porque os coletores devem ser coordenados, para que as mudanças das referências entre nós (espaços de endereçamento distintos) sejam detectadas consistentemente. Os problemas de consistência complicam-se devido às falhas inerentes à distribuição, tais como perdas, duplicações e atrasos de mensagens, além de falhas nos espaços individuais. Assim, a reciclagem de células usadas de toda a espécie de estruturas de dados deve atender, no mínimo, às seguintes exigências: eficiência, escalabilidade, e tolerância à falha.

Com o intuito de atender a essas exigências, muitas abordagens têm surgido em coleta de lixo distribuída. Porém, a maioria dessas contribuições tem atendido parcialmente esses aspectos [37], resultando em um significativo número de propostas ainda incompletas, impedindo o atendimento total desses requisitos. A principal razão dessas soluções parciais é localizada na adaptação de algoritmos originalmente projetados para funcionar em um processador local para aqueles que devem funcionar em ambientes distribuídos. Essa adaptação não é direta. A técnica de varredura e marcação exige mecanismos de terminação caros, enquanto o contador de referências é prejudicado pelas falhas na entrega/resposta de mensagens.

A coleta de lixo é crítica para objetos distribuídos e persistentes, porque é difícil para o cliente manter a informação sobre as referências corretamente. Idealmente, em um sistema de coleta de lixo distribuído, os objetos continuam a existir enquanto eles são alcançáveis pelos clientes ou por um objeto raiz e deveriam ser reciclados quando não mais acessíveis. Na prática, isso é difícil em computação distribuída descentralizada em larga escala, devido às razões identificadas, em [38], que reproduzimos, a seguir: 1) objetos distribuídos e referências são dinamicamente criados, removidos, migrados, e compartilhados pela rede. Assim, fica difícil determinar: 1-a) quando um deles não é mais acessível, e, 1-b) qual o momento seguro que deve ser efetuada a reciclagem; 2) sistemas distribuídos são administrados, descentralizadamente, então a cooperação de clientes bem comportados não deveria ser esperada; 3) Sistemas distribuídos podem trabalhar com grandes volumes de dados, então é impossível obter uma visão global dos clientes, objetos e suas referências; 4) Servidores e clientes podem falhar durante o funcionamento da reciclagem; 5) Mensagens podem ser perdidas, e a rede pode ficar rompida por um período.

O algoritmo desenvolvido para coleta de lixo distribuída, apresentado em [2], possui um conjunto de propriedades, baseadas nas questões identificadas por seus autores, vistas à frente, que asseguram que dessa forma ele se torna consistente. Ao mesmo tempo, o seu uso é encorajado tanto em bibliotecas de tempo de execução de linguagens simbólicas de computação distribuída, como também em sistemas de arquivos distribuídos ou em sistemas de base de dados distribuídas para a coleta de objetos ou em arquivos não mais referenciados. A partir deste ponto, este algoritmo será chamado neste artigo de GC[2].

Os autores de GC[2] usaram alguns conceitos próprios para facilitar o seu entendimento. Chamaram de Nó um processador ou um processo em um processador capaz de gerenciar seu próprio espaço de memória. Um conjunto desses nós forma uma rede e sua comunicação é feita através da troca de mensagens. Esses Nós podem conter processos que são chamados de *Mutators*. Eles realizam computações independentes e alocam fatias de memória chamadas células tanto para a necessidade do próprio Nó como para servir a outros Nós da rede. Cada Nó também contém as chamadas raízes (roots) que são referências a células consideradas

úteis. Em particular, todas as referências de células conhecidas pelo *mutator* (especialmente em registradores ou em uma pilha de execução) são consideradas raízes. Células referenciadas por uma raiz direta ou indiretamente através de outras células são ditas alcançáveis ou vivas. As outras são ditas inalcançáveis ou mortas e constituem o lixo a ser reciclado pelo coletor. Uma referência a uma célula no mesmo nó é considerada local. Uma referência a uma célula em outro nó é considerada remota. A Coleta de Lixo dentro de um nó com base em raízes e referências locais é chamada de Coleta de Lixo local.

Uma referência remota a uma célula γ é representada por um item de saída (*exit item*) no mesmo nó, que referencia um item de entrada (*entry item*) em outro nó, que, por sua vez, referencia localmente a célula γ conforme mostra a figura 1. Para obter o valor de uma referência remota, três níveis de indireção e algum tempo de comunicação são necessários. É esperado obviamente que o número de referências remotas seja bem menor que o número de referências locais. Os itens de saída (e os de entrada respectivamente) são imutáveis com respeito às células as quais eles se referem e, desta forma, eles podem ser seguramente compartilhados: cada nó tem apenas um item de saída para todas as referências remotas para uma dada célula e apenas um item de entrada para toda célula local remotamente referenciada.

Um coletor de lixo de grupo (*group GC*) é um coletor de lixo não-local, isto é, envolve mais de um nó. Um coletor de lixo de grupo opera em um grupo de nós, reciclando qualquer célula do grupo que esteja comprovadamente inacessível a partir de qualquer raiz de qualquer nó. É importante notar que itens de entrada ou de saída não fazem parte do conjunto de raízes. A criação de uma referência remota envolve a criação sincronizada de itens de entrada e saída associados.

Itens de entrada têm um contador de referências que é igual ao número de itens de saída que os estão referenciando. Quando um item de saída é reciclado, uma mensagem de decremento é enviada para o contador do item de entrada que estava sendo referenciado. Se o contador do item de entrada é zerado, o mesmo também é reciclado. Este é o único mecanismo disponível para reciclar itens de entrada e é seguro, porque nós não-cooperativos (que estão fora de operação) não enviam mensagens de decremento e, desta forma, as células as quais elas se referem não podem ser recicladas (sem uma intervenção externa).

Pela natureza do nosso ambiente, que incorpora distribuição e persistência, consideramos estas questões e propriedades, identificadas em [2], apropriadas para a definição do problema, e as reproduzimos, a seguir:

1) Embora o sistema seja distribuído, o processador administra seu espaço de dados, ajudado por um coletor de lixo local.

2) Objetos referenciados, remotamente, possuem um contador de referências para que uma grande parte desses objetos seja desalocada, simplificadamente, quando se tornar inacessível.

3) Processadores são organizados em grupos.

4) Os processadores de um grupo cooperam com coletores de lixo parciais que são globais em relação ao grupo: o objetivo de um grupo é, entretanto, descobrir e reciclar células inalcançáveis e, particularmente, referências circulares, espalhadas pelos nós do grupo, por intermédio de um coletor de varredura e marcação concorrente.

5) Inúmeros grupos de coletores de lixo podem ter elementos em comum e estarem simultaneamente ativos.

6) Quando um processador ou link de comunicação falha em um sistema local ao qual um dos coletores de lixo pertence, os grupos dos quais ele fazia parte não são afetados, pois são capazes de se reorganizar e continuar seu trabalho normalmente.

7) Eventualmente todos os objetos não-referenciados distribuídos irão pertencer a um grupo que irá reciclar os mesmos quando eles finalizarem seu coletor de lixo associado.

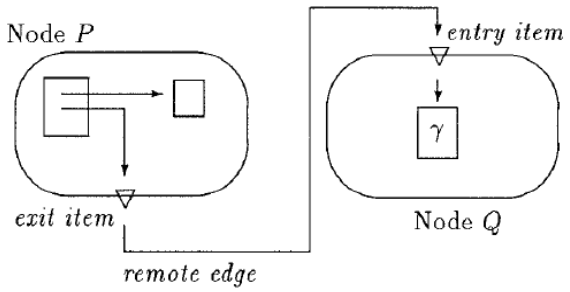


**Fig. 1. Referência Remota** - reproduzido de GC[2]

Além dessas propriedades, os autores em [2] apontam as seguintes necessidades em um coletor de lixo, nesse contexto:

a) O coletor de lixo dá suporte à falha de processadores, e, nesse caso, ainda permanece ativo para proceder a coleta nos processadores, que ainda encontram-se funcionando. Ele também deve dar suporte à adição de novos processadores, ou trocas na topologia de rede.

b) O coletor de lixo não necessita de um controle centralizado: por exemplo, uma rede pode se desmembrar em duas sub-redes desconectadas entre si e cada uma delas continuar a cuidar da coleta de lixo em seu próprio espaço. Múltiplos grupos de coletores de

lixo podem estar simultaneamente ativos, cada um deles preocupado com seu escopo de memória aonde podem atuar.

c) Pode ser usada qualquer variante de algoritmo de varredura e marcação para procedimentos de coleta de lixo local, contanto que o coletor transmita as marcas usadas pelo grupo de referências de entrada remota para referências de saída remota. Para o grupo de coletores não são necessários bits especiais durante as coletas locais.

A detecção e/ou reciclagem de células sem uso não requer a migração de objetos de processador para processador: o algoritmo respeita a localidade de objetos conforme decidido pelo *mutator*, que é um processo em cada nó da rede, realizando computações independentes e responsáveis pela alocação de células de memória, tanto para a necessidade do próprio nó como para servir a outros nós.

Na próxima seção, identificamos os fatores que simplificam nosso mecanismo, depois relacionamos a nossa solução com os componentes referidos no algoritmo GC[2], encerrando com uma breve descrição do funcionamento do nosso mecanismo e dos recursos adicionados aos elementos da arquitetura SPRMI, que foram necessários à sua implementação.

## 3 Modelo de Computação

Alguns dos termos de referência a elementos da arquitetura do SPRMI, como *broker* e armazém de objetos já apareceram, acima. Eles serão novamente referenciados no curso da descrição do nosso mecanismo de coleta de lixo, bem como outros elementos da arquitetura. Dessa forma, cabe a seguir uma breve descrição do papel de cada elemento nesta arquitetura.

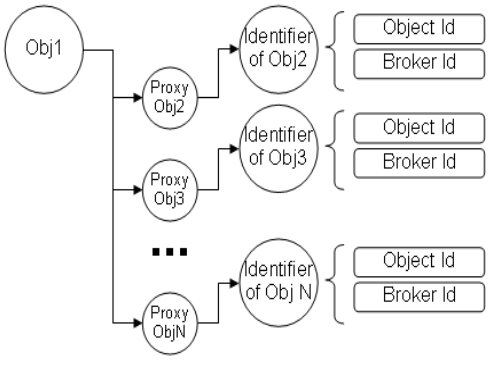


**Fig. 2. Atributos do objeto Obj1, neste exemplo, são *proxies* que possuem o atributo identificador com os dois campos numéricos. Cada atributo desses assume o papel da referência ao objeto. Referências assim são resolvidas pelo SPRMI, inicialmente, o *proxy* repassa a invocação ao *broker* (Broker Id), que a repassa à entrada no armazém (Object Id), que, finalmente, a repassa ao objeto efetivo.**

Os principais componentes da biblioteca SPRMI são os seguintes:

1) ***Proxy*** - O *proxy* foi implementado com o emprego da API *Dynamic Proxy* [39]. Ele é um representante local de um objeto e dentro da estrutura do SPRMI ele usa uma classe **`Identifier`** para guardar as informações sobre a localização, fazendo também o papel de identificador do objeto que ele representa. Esse identificador é formado por um par de informações numéricas: o identificador do *broker*; e a posição da entrada associada ao objeto na tabela do armazém de objetos. O fato desse identificador ser numérico propiciou a instanciação rasa de um objeto, que é uma das principais características da biblioteca SPRMI. Essa característica evita que, na instanciação de um objeto, os seus atributos, que possuírem referências desse tipo, propaguem a instanciação dos objetos referenciados, quebrando dessa forma o grafo de composição de objetos de Java, ver figura 2. Um objeto pode ser instanciado de forma rasa caso os seus atributos sejam de dois tipos: ou objetos referenciados por esses *proxies*, ou atributos de tipos primitivos. Mas não podemos nos esquecer que um objeto pode incluir também referências como as comuns de Java para outros objetos *user-defined*. Nessas últimas, a árvore de composição estará representada no objeto, e, conseqüentemente, salva/restaurada no armazém.

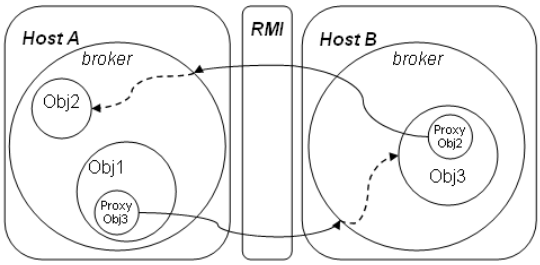


**Fig. 3. Caminho percorrido, somente na ida de uma invocação, a partir de Obj1, invocando um método de Obj3, que por sua vez invoca um método do Obj2. Esse exemplo ilustra a condição de cada objeto sempre ser instanciado no local escolhido, originalmente, no momento da sua criação.**

Quando uma referência a um *proxy* é usada em uma invocação, o primeiro passo, no caminho a ser percorrido, antes de chegar ao método do objeto alvo, é o método **`invoke`** do *proxy*. Neste método, a invocação ao objeto alvo é serializada (*marshalled*) em forma de uma mensagem, para que possa ser repassada no passo seguinte ao *broker* aonde encontra-se o objeto alvo. Retomando a descrição do caminho, ainda no *proxy*, após o seu método **`invoke`** ter serializado a invocação, é efetuada uma comunicação do *proxy* para com o *broker*, que ocorre através da conexão RMI. Esta

comunicação é efetuada por uma invocação ao método do *broker* **`runMethod`** e o parâmetro é uma mensagem serializada com todas as informações: método, referência ao objeto alvo e a lista de argumentos. Essa invocação é especificada na sintaxe comum de envio de mensagem em Java, porém o pacote RMI encontra-se agindo de forma transparente. Chegando ao *broker*, seu método **`runMethod`** efetua a desserialização (*unmarshalling*) do parâmetro, para descobrir a posição da entrada no armazém e obter o método e os parâmetros originais da invocação. Neste ponto, o método **`runMethod`** da entrada é invocado com os parâmetros: método e os parâmetros originais da invocação. No método **`runMethod`** da entrada, então, é possível fazer a invocação efetiva ao objeto alvo, obedecendo a seguinte sintaxe:

"**`method.invoke(class.cast(object), args)`**"

Cabe notar que neste ponto todas as informações necessárias para efetuar a invocação efetiva, acima, estão disponíveis, como a saber: **`class`** e **`object`** são atributos da entrada e que **`method`** e **`args`** são parâmetros do método corrente (**`runMethod`** da entrada).

O caminho de volta será efetuado pela forma comum de retorno de método e todas as invocações no caminho aguardam o seu retorno. Somente, no retorno do *broker* para o *proxy*, é que o RMI novamente entrará em cena para transmitir o resultado para o *proxy*. A Figura 3 ilustra, abreviadamente, um caminho de ida, empregando dois *host*s.

2) ***Broker*** - O *broker* é um componente que implementa o padrão *message broker*. Ele recebe chamadas serializadas e as invoca nos objetos alvo que residem em seu armazém, devolvendo o resultado. As chamadas podem ter origem em objetos locais ou remotos. Os métodos serializados, recebidos pelo *broker* através de uma rede, são reconstruídos pelos mecanismos de reflexão e finalmente invocados no objeto alvo. O seu principal atributo é o armazém de objetos. O *broker*, em cooperação, com o armazém de objetos, é quem tem o controle da persistência e da transmissão da invocação de métodos aos objetos alvos, oriunda de uma aplicação ou de métodos de outros objetos instanciados no ambiente distribuído, lembrando-se que invocações propagam-se pela composição dos objetos.

3) **Armazém de objetos** – O armazém de objetos é a estrutura que abriga os elementos (entrada) que referenciam os objetos persistidos pelo *broker*, bem como os próprios objetos. Ele é um atributo do *broker*, chamado **`apObjs`** e implementado por uma **`HashTable`** e os seus elementos são da classe **`TableEntry`**. Cada instância de uma **`TableEntry`** está associada a um objeto persistente e distribuído. Quando um objeto assim é criado, é criada também sua entrada na tabela com as informações desse objeto.

O SPRMI também possui alguns componentes auxiliares. O *centralhost* é um *broker* responsável por manter uma tabela com todos os *broker*s que fazem parte da configuração distribuída existente. A importância do *centralhost* está associada à habilidade de referenciar outros *broker*s. O *System Identifier Store* é responsável por criar as identificações numéricas dos objetos e *broker*s de forma única a fim de que não haja colisão de identificadores.

Com essa breve apresentação dos elementos principais do middleware SPRMI, podemos agora descrever o processo de adaptação do algoritmo GC[2] à sua arquitetura, de uma forma mais confortável, no momento em que for preciso referenciar algum(ns) de seus termos.

### 3.1 Adaptando GC[2] ao SPRMI

Pelo nosso mecanismo, os *broker*s fazem parte de um grupo apenas, em vez de grupos de b*roker*s, como é considerado em GC[2]. O coletor de lixo local a um *broker* é acionado pelo *broker* que encontra-se no papel de *centralhost*. Cada *broker* corresponde ao nó do algoritmo GC[2], e é capaz de administrar os objetos do seu armazém e fazer a varredura desses objetos, bem como descobrir todas as referências que eles possuem, tanto locais quanto remotas. Além disso, cada broker será capaz de fazer a marcação de todos os objetos do seu armazém que foram referenciados em todo o ambiente distribuído. Isso é possível, por que cada broker organiza uma coleção, separada para cada broker, que for referenciado, com os objetos referenciados que residem nesse broker referenciado. No caso extremo, cada broker organiza um número de coleções igual ao número de brokers. Cabe notar, que nesse processo, ele organiza uma coleção para si também, porque precisa marcar também os seus próprios objetos. A funcionalidade do *mutator*, em GC[2], de alocar memória para servir a referências locais ou remotas também é provida pelo *broker*, como observamos acima. A identificação das referências a objetos de dados pelo *mutator* é resolvida pelos objetos armazenados na pilha de alocação dinâmica e nos registradores em GC[2], enquanto que o nosso algoritmo tem condições de descobrir, investigando os atributos de cada objeto, iniciando pelos objetos *top-level*. O nosso mecanismo estará varrendo as referências aos objetos que são persistentes e distribuídos, quer dizer, apenas aqueles que se encontram em alguma entrada da tabela que representa o objeto no armazém. Nessa investigação, como cada referência contém a localização do *broker* aonde reside o objeto, caso esse *broker* seja remoto, essa condição é usada na

cooperação para informar que esse objeto remoto é referenciado pelos objetos em seu local. Como já anunciamos, acima, todas as referências encontradas nos objetos de um *broker* são guardadas numa estrutura do *broker*, reunidas separadamente para cada *broker* remoto, lembrando os *exit itens* de GC[2]. Os contadores de referência de cada entrada associada a um objeto, no armazém de objetos, lembra os *entry itens* de GC[2].

Os coletores de lixo locais aos *broker*s funcionarão em *thread*s exclusivas a cada um, independentemente, das *thread*s das aplicações, não dando chances ao problema de *stop world*, que surge na maioria dos coletores de lixo, inclusive locais.

Um aspecto importante para a simplificação do mecanismo de coleta de lixo em nosso sistema é o fato dos objetos serem persistentes e alocados apenas no armazém controlado pelo SPRMI, aonde residem. Nenhum objeto é passado por parâmetro para outro *host*. Esta impossibilidade de migração de um objeto pelo ambiente distribuído elimina uma das dificuldades enumeradas, em [38], economizando o esforço de efetuar a varredura nessa cópia em local distinto de onde ele foi criado e reside.

O nosso mecanismo de coleta de lixo não assume a responsabilidade sobre os objetos criados em execuções paralelas ao processo de varredura e marcação. Somente os objetos persistentes e distribuídos e que não se encontram instanciados é que poderão ser reciclados. Isso significa que os objetos candidatos à reciclagem não participam de qualquer execução, ou seja, não estão trocando mensagens, subtraindo do mecanismo mais essa responsabilidade de tratamento quando surgem mensagens perdidas. Essa condição nos objetos também elimina a dúvida sobre o momento seguro para a reciclagem deles. Mais um outro benefício ainda pode ser percebido, na situação em que algum *host* ou nó (*broker*) falhar, bastando apenas descartar a execução da coleta de lixo, porque as únicas alterações efetuadas no estado do ambiente são as marcas nas entradas do armazém e as coleções de referências. Isso é possível porque essas estruturas são iniciadas na fase inicial do nosso mecanismo de coleta de lixo. Na hora da reciclagem, os elementos deverão ser removidos, mas caso algo fique perdido, uma nova coleta descobrirá esse lixo novamente.

Após a fase de marcação, a reciclagem de um objeto armazenado será efetuada de forma isolada, porque a sua representação (estado) será removida com o objeto, não importando se os atributos são *proxies* ou tipos primitivos ou referências a objetos *user-defined* de Java. A reciclagem de um objeto vai significar a sua exclusão do armazém, quer dizer, o seu espaço na memória secundária será desocupado e serão removidos o próprio objeto e a sua entrada na tabela, que representa o armazém.

Com esses elementos identificados e relacionados aos termos introduzidos em GC[2], acima, descrevemos a seguir o funcionamento do nosso mecanismo. Resolvemos apresentá-lo antes de enumerar os recursos que foram adicionados ao *broker*, empregados na integração de nosso mecanismo ao SPRMI. Esta decisão foi tomada com a finalidade de facilitar a leitura da seção que enumera os recursos adicionados.

## 3.2 Funcionamento do Nosso Mecanismo

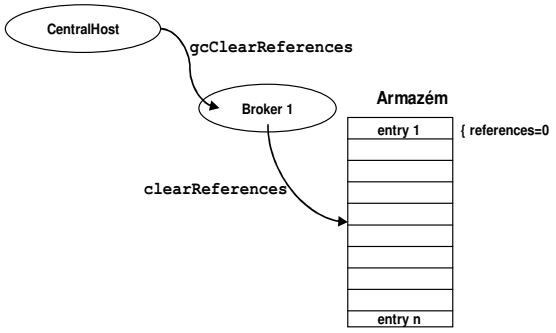


**Fig. 4. Desmarcando os objetos**

Como vimos na seção anterior, o nosso mecanismo foi baseado no algoritmo GC[2], que enfrenta as dificuldades trazidas pela distribuição, através de um conjunto de propriedades, que tornam uma coleta de lixo consistente

.

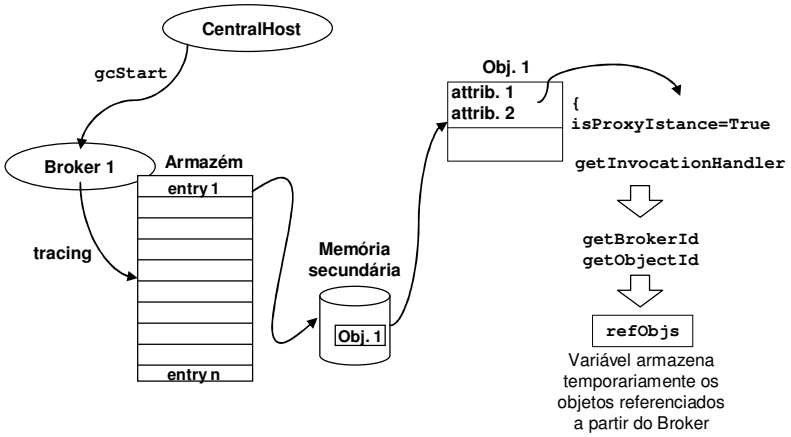


**Fig. 5. Varrendo objetos do armazém.**

A seguir, descrevemos o funcionamento do nosso mecanismo, que foi baseado em GC[2], guiado pelos passos descritos acima e acompanhado dos recursos Vimos também que a arquitetura SPRMI permitiu que a nossa solução ficasse ainda mais simples do que a apresentada pelo algoritmo GC[2]. Tanto o nosso mecanismo quanto o de GC[2] reproduzem a seqüência padrão em coleta de lixo do tipo varredura e marcação, que inicialmente remove todas as marcas por ventura deixadas pela coleta anterior, para então prosseguir nos seguintes passos: 1) o passo considerado o mais difícil, na literatura, porque pode propagar-se em várias recursões pelas estruturas dos objetos, para que sejam encontradas todas as referências existentes no ambiente de execução. Essas referências devem ser reunidas numa lista, que deve ser usada pela marcação, que é o passo seguinte; 2) a partir de cada referência da lista, deve ser marcada a célula (no nosso caso o atributo de marca da entrada do objeto no armazém de cada *broker*). Essa marca indicará que este objeto não é lixo; e 3) fazer nova varredura, agora, pelas células (no nosso caso, pelas entradas dos objetos na tabela do armazém de cada *broker*), aonde encontram-se os objetos, reciclando todos que encontram-se desmarcados.

Como já anunciamos, o nosso algoritmo fará o exame da cada atributo de cada objeto, em cada *broker*, com o interesse de identificar se a referência é uma instância de *proxy*. É exatamente essa condição que indica que o objeto referenciado neste atributo pertence ao armazém de objetos de algum *broker*. Como este objeto está sendo referenciado, então é providenciada a sua marcação, para que ele não seja reciclado. Ao final do processo, quando todos os três passos forem cumpridos, com exceção dos objetos do tipo *top-level*, todos aqueles que não estiverem marcados deverão ser reciclados. O *top-level* não é referenciado, porque é o objeto principal de uma aplicação, cuja execução é solicitada, na maioria das vezes, por usuários.

Durante essa descrição, muitas vezes nos referimos aos elementos da arquitetura SPRMI. Para efeitos de simplificação, foi dada preferência aos termos *proxy*, *broker*, entrada, tabela, armazém, que já foram apresentados, em seções anteriores, com os seus respectivos identificadores empregados na implementação. Os termos tabela e armazém muitas vezes serão confundidos, porque o armazém é implementado pela tabela.

Cabe notar, neste ponto, que foi exaustivamente explorada uma disciplina de programação no SPRMI, com a finalidade de respeitar o encapsulamento dos objetos. Sempre que o *broker* precisou efetuar uma ação cujo alvo não era referenciado por um atributo seu, foi criado um método de mesmo nome ou parecido, na classe da declaração do atributo que deveria ter possibilidade de acesso direto ou indireto ao objeto alvo. Assim, numa cadeia de composições, aparecem repetidos os métodos até que seja atingido o objeto que efetivamente vai executar um método ou acessar ou alterar algum atributo. Essa disciplina pode ser constatada, por exemplo, nas invocações de limpeza de marcas; marcação; e no **`runMethod`**.

O *centralhost* foi o *broker* escolhido para dar o ponto de partida na execução do mecanismo da coleta de lixo. Essa decisão foi conduzida pelo fato de que, entre outras responsabilidades, ele detém o conhecimento de todos os outros *broker*s, em uma estrutura denominada **`neighborhood`**, construída com o emprego da classe parametrizada **`Hashtable`**, que guarda uma referência a cada um dos *broker*s que compõe o ambiente distribuído.

Para esse ponto de partida, foi especificado um método de instância na classe **`Broker`** chamado **`gcClearReferences`**. A sua finalidade é efetuar uma varredura pelas entradas da tabela do armazém, limpando as marcas deixadas por uma coleta anterior.

No início do mecanismo, então, o *centralhost* faz uma invocação com esse método a todos os *brokers*. Em cada entrada da tabela do armazém de cada *broker*, é registrada a contagem de referências. Esse contador faz o papel de uma marca, se zero é considerado desmarcado e se maior que zero é considerado marcado. Quem efetivamente limpa as marcas, quer dizer, zera o contador de referências do objeto, no atributo **`references`** da entrada, é o método **`clearReferences`**" especificado na classe **`TableEntry`**. Esse método é invocado, pelo método **`gcClearReferences`** do *broker*. Ao final desta varredura temos todos os objetos da rede de *broker*s desmarcados, quer dizer, com o contador de referências zerado. A figura 4 ilustra a cooperação entre o *centralhost*, *broker*, armazém e entrada.

Zerada a referência de cada entrada, o próximo passo é descobrir todos os objetos que são referenciados, em cada *broker*. Cada referência encontrada deve ser incluída na estrutura **`refObjs`** pertencente ao *broker*. Para isso, o *centralhost* invoca o método **`gcStart`** de todos os *broker*s da rede, cuja funcionalidade é efetuar uma varredura no armazém do *broker*, instanciando todos os seus objetos.

Para possibilitar o acesso aos atributos de cada um desses objetos instanciados, contamos com os recursos de reflexão do pacote **`java.lang.reflect`**. Cada um desses atributos é analisado, verificando se é uma instância de **`Proxy`** através do método "**`isProxyClass`**" da classe **`Proxy`**. Somente nessa condição, quer dizer, quando o objeto referenciado pelo atributo for uma instância de **`Proxy`**, deve ser obtido o **`InvocationHandler`** associado, através do método "**`getInvocationHandler`**" da classe **`Proxy`**. O **`InvocationHandler`** associado ao **`Proxy`** é do tipo **`ShallowInvocationHandler`** que foi estendido pelo pacote SPRMI. A ele foi acrescentado um atributo do tipo **`Identifier`** criado no pacote SPRMI. Refinando mais o conteúdo do atributo **`id`** de **`InvocationHandler`** encontramos as informações necessárias para o acesso ao objeto referenciado, o **`objectId`** e o **`brokerId`**

Após essa varredura, na fase de marcação, é necessário marcar todos esses objetos referenciados, tendo cada um desses a chance de pertencer a qualquer *broker* da rede. Essas referências encontradas foram guardadas em cada *broker* local, na sua estrutura **`refObjs`**. Após o término do método **`gcStart`**, o *centralhost* providencia a emissão de invocações do método **`gcMark`** a todos os *brokers*. Cabe lembrar que **`refObjs`** guarda as referências, separando-as conforme o *broker* a que pertencem os objetos dessas referências. Assim, o método **`gcMark`**, em execução, considera cada *broker* desses como o objeto alvo das invocações a serem emitidas com um segundo método **`gcMark`** que tem como parâmetro a lista de referências dos objetos pertencentes a esse objeto alvo. A funcionalidade desse segundo **`gcMark`** é marcar as entradas correspondentes às referências da lista, recebida como parâmetro.

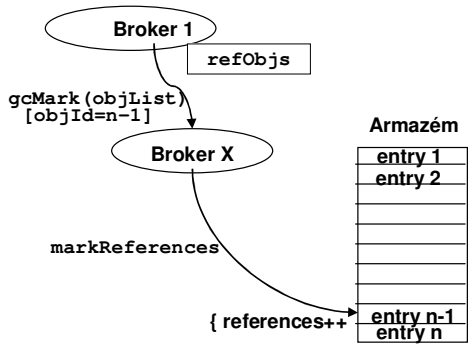


**Fig. 6. Marcando os objetos referenciados, quer dizer, incrementando o contador de referências**.

No segundo método **`gcMark`**, para cada referência, é obtida a entrada correspondente, que é o alvo da invocação do método **`markReferences`**. Esse método, como o próprio nome diz, efetua a marcação do objeto, incrementando o atributo **`references`** de sua entrada.

O mecanismo toma cuidado com um fato que pode surgir, em meio ao processo de coleta de lixo. Como nas execuções de aplicações, no ambiente distribuído, novos objetos podem ser criados e persistidos nos *broker*s. Para que esse objetos não sejam reciclados, o atributo **`references`** da entrada associada a cada novo objeto deve receber o valor inicial igual a 1. Dessa forma, o mecanismo não vai tentar reciclar esse novo objeto.

Encerrada a fase de marcação, será iniciada a de reciclagem, que é o último passo a ser tomado, pelo mecanismo.

O *centralhost* emite a invocação do método **`gcSweep`**, a cada *broker* do ambiente distribuído. Esse método, no *broker*, efetua uma varredura no seu armazém, analisando cada entrada, exceto aquelas associadas a objetos do tipo *top-level*. Nesta análise, se o contador de referências, o atributo **`references`**, estiver zerado, o objeto associado a essa entrada será considerado lixo. Então é invocado o método **`deleteObjFile`** dessa entrada, que providencia a remoção do arquivo aonde persiste o objeto que foi considerado lixo. Em seguida, o método **`gcSweep`**

exclui a própria entrada do armazém, associada a esse objeto considerado lixo. Ao final da execução do método **gcSweep**, em cada *broker*, os objetos considerados lixo terão sido reciclados. A figura 7 ilustra este procedimento.

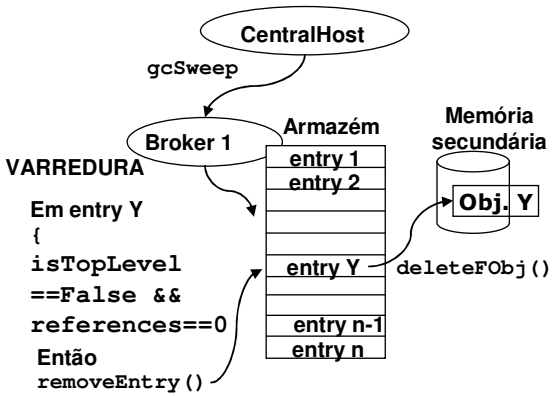


**Fig. 7. Removendo um objeto inacessível (lixo), sua entrada, e o arquivo na memória secundária aonde ele persistia.**

Quando esse método já tiver sido encerrado, em todos os *brokers*, terá também sido encerrada a fase de reciclagem e, conseqüentemente, a própria coleta de lixo.

### 3.3 Recursos adicionados ao SPRMI

Como o mecanismo de coleta de lixo foi um passo adiante em termos de controle sobre os objetos já administrados pelo *broker*, podemos caracterizar a implementação do nosso mecanismo como um conjunto adicional de objetos, métodos e ações, descritos a seguir, que enriqueceu a versão original deste *middleware* SPRMI, pela integração desse mecanismo de coleta de lixo.

**Método `gcLoad` do *broker*** Este método, pertencente ao *broker*, é o ponto de partida do mecanismo de coleta de lixo e é invocado somente se o *broker* estiver no papel de *centralhost*.

**Método `gcClearReferences` do *broker* –** Este método funciona, em cada *broker*, na fase inicial da coleta de lixo, limpando todas as marcas de alguma coleta anterior. Ele percorre o armazém de objetos do *broker* e solicita que o atributo "**`references`**" das entradas da tabela seja zerado.

**Método `gcStart` do *broker* –** Este método é responsável por executar uma varredura procurando nos objetos do armazém por todas as referências, colecionando-as na estrutura **`refObjs`** do broker.

**Método `gcMark` do *broker* –** Este método tem a função de marcar os objetos que encontram-se referenciados em **`reObjs`**, em todos os *brokers*, para que os mesmos não sejam reciclados.

**Método `gcSweep` do *broker* –** O método `gcSweep` efetua a última fase do mecanismo de coleta de lixo, percorrendo o armazém de objetos e reciclando aqueles que não foram marcados na fase de marcação, quer dizer, o atributo **`references`** da entrada do objeto encontra-se zerado.

**Método `put` -** A este método foi acrescentado um parâmetro chamado "**`isTopLevel`**" do tipo **`boolean`** que informa se um objeto que está sendo inserido no armazém é *top-level* (raiz) ou não

**Interface `BrokerInterface`** A assinatura dos métodos **`gcClearReferences`**, **`gcStart`**, **`gcMark`** e **`gcSweep`** foram acrescentadas à interface **`BrokerInterface`** para que fosse possível a invocação remota dos mesmos.

**Classe `TableEntry` e recursos adicionados** Esta é a classe de acesso a cada objeto no armazém. Uma instância desta classe representa uma entrada na tabela que implementa o armazém. Ela mantém a referência ao objeto, o nome do arquivo no qual ele foi persistido, a classe e a identificação deste objeto.

**Atributos Adicionados à `TableEntry`:**

**Atributo `references`** Este atributo indica a quantidade de objetos que referenciam o objeto associado a esta entrada.

**Atributo `topLevel` -** Este atributo pode possuir dois valores, **`true/false`**, indicando se o objeto associado a esta entrada é ou não *top-level*.

**Métodos Adicionados à `TableEntry`:**

**Método `getTopLevel` -** Este método retorna a condição *top-level* ou não do objeto.

**Método `setTopLevel` -** Este método preenche a condição *top-level* ou não do objeto.

**Método `markReferences` -** Este método incrementa o atributo **`references`** da entrada, que significa mais uma referência ao seu objeto.

**Método `getReferences` -** Este método retorna o valor do atributo **`references`**, que é o número de referências ao objeto associado.

**Método `clearReferences` -** Este método zera o atributo **`references`**, significando que não há mais referências ao objeto.

**Método `getObjFName` -** Este método retorna o nome dado ao arquivo no qual o objeto associado à entrada foi persistido.

**Método `deleteObjFile` -** Este método remove o arquivo no qual o objeto associado à entrada foi persistido.

**Classe `ShallowInvocationHandler`** Esta classe implementa a interface fornecida no pacote de reflexão de Java, **`InvocationHandler`**. Um objeto

desta classe é fornecido como parâmetro na criação de uma instância de *proxy* que é um representante de outro objeto, quer dizer, sua referência. Quando um método é invocado sobre um atributo que possui um *proxy*, essa invocação é repassada ao método **`invoke`** de **`ShallowInvocationHandler`**, que por sua vez, após procedimentos de *marshalling*, a encaminha para os passos seguintes (RMI, *broker*, entrada) até que chegue no objeto alvo.

**Método `getBrokerId` -** Método criado para devolver a identificação do *broker*, que fica preenchida no estado do *proxy*.

**Método `getObjectId` -** Método criado para devolver a identificação do objeto, que fica preenchida no estado do *proxy*.

**Método `newTopLevelInstance` -** Este método foi criado para possibilitar a criação de objetos *top-level* no armazém de objetos de um *broker*. A criação de objetos através deste método passa o parâmetro real com o valor **`True`** para o parâmetro formal **`isTopLevel`**.

**Método `ShallowInvocationHandler`** – Foi acrecentado a este método o parâmetro **`isTopLevel`**, para que, no momento da criação de um objeto, o *broker* pudesse ser avisado que trata-se de um objeto *top-level*.

Os recursos que foram adicionados durante a implementação da integração do mecanismo de coleta de lixo ao SPRMI foram descritos acima. A oportunidade de evitar a herança das classes, principalmente, a do *broker*, veio do compromisso com a modularização, adotado no projeto do SPRMI. No desenvolvimento do SPRMI, o encapsulamento foi esgotado ao extremo, prevendo as funcionalidades que ainda precisavam ser implementadas. Esse compromisso pode ser percebido nas várias vezes que o *centralhost* providenciou algo que devesse ser espalhado pelos *brokers* do ambiente distribuído. Essas providências nos *brokers* muitas vezes não eram diretas. Um nível ou dois de indireção a mais fazia-se necessário. Para evitar a quebra de encapsulamento, com acesso a um atributo externo, adotamos o uso de métodos com o mesmo nome ou parecidos, que tem como funcionalidade apenas repassar a invocação de atributo em atributo, pela composição, até chegar ao objeto que efetivamente efetue a funcionalidade desejada.

## 4 Conclusões

O nosso objetivo inicial foi o de incorporar um mecanismo de coleta de lixo ao armazém de objetos do SPRMI, para que todos os objetos inalcançáveis por referências fossem reciclados.

A computação distribuída dá chances aos objetos e suas referências de serem dinamicamente criados, removidos, migrados, e compartilhados pela rede. Com tal comportamento fica difícil determinar, segundo os autores, em [38], condições importantes para uma coleta de lixo, como a seguir: 1) quando um objeto não é mais referenciado por todo o ambiente distribuído e qual o momento seguro para ser efetuada a reciclagem; 2) se sistemas distribuídos são administrados, de forma descentralizada, então a cooperação de clientes bem comportados não deveria ser esperada; 3) os dados podem atingir um volume grande, dificultando uma visão global dos clientes, objetos e suas referências; 4) servidores e clientes podem falhar durante o funcionamento da reciclagem; e 5) mensagens podem ser perdidas, e a rede pode ficar rompida por um período.

Vimos, porém, que tanto a arquitetura SPRMI quanto o nosso objetivo de reciclar apenas objetos não instanciados do armazém ofereciam várias oportunidades de evitar alguns dos problemas que costumam surgir na coleta de lixo distribuída.

Entre essas chances, a mais importante para essa simplificação é o fato dos objetos serem persistentes e alocados apenas no armazém controlado pelo SPRMI, aonde residem. Nenhum objeto é passado por parâmetro para outro *host*. Esta impossibilidade de migração de um objeto pelo ambiente distribuído elimina uma das dificuldades enumeradas, em [38], economizando o esforço de efetuar a varredura nessa cópia em local distinto de onde ele foi criado e reside. Uma outra decisão tomada foi a de tirar a responsabilidade do nosso mecanismo sobre os objetos criados em execuções paralelas ao processo da coleta de lixo. Assim, somente os objetos persistentes e distribuídos e que não se encontram instanciados é que podem ser reciclados. Isso garante também que os objetos investigados não encontram-se trocando mensagens, subtraindo do nosso mecanismo de coleta de lixo mais um peso, que é o tratamento de mensagens perdidas ou das conseqüências trazidas quando a rede fica rompida. Essa condição nos objetos também elimina a dúvida sobre o momento seguro para a reciclagem deles. Além disso, se um host falhar, como esses objetos não encontram-se participando de qualquer execução, o processo pode ser simplesmente descartado, porque as únicas intervenções no estado dos *brokers* são as marcas nas entradas para cada objeto e a estrutura que guarda as referências. Como no início de um processo de coleta esses controles são iniciados, não há necessidade de recuperá-los.

Seria interessante aplicarmos os mesmos testes que já foram realizados com programas, empregando o *middleware* SPRMI e, separadamente, só o RMI. As medidas de desempenho revelaram diferenças de 30 a

50% a favor do SPRMI. Um dado a mais que traz vantagens ao desempenho do nosso mecanismo é que os coletores de lixo locais aos *broker*s funcionam em *thread*s exclusivas a cada um, independentemente, das *thread*s das aplicações, não dando chances ao problema de *stop world*, que surge na maioria dos coletores de lixo, inclusive locais. Além disso, suas intervenções em objetos que estão em execução são suaves, ele faz apenas uma breve investigação no estado de cada objeto, para descobrir as referências.

## 5 Referências


[1] P.R. Motta, G. Moreira, and M.A. Brito, "SPRMI: Pure Java Based Shallow Persistent and Objects", *Cadernos IME – Série Informática*, vol. 22, IME/UERJ, Rio de Janeiro, 2007.

[2] B. Lang, C. Queinnec, and J. Piquer, "Garbage Collecting the World", In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '92)*. ACM Press, New Mexico, 1992.

[3] Sun Microsystems, "JavaTM Remote Method Invocation (Java RMI)", <*http://java.sun.com/j2se/1.5.0/docs/guide/rmi/index.html*>.

[4] M.P. Atkinson, P.J., K.J. Chisholm, W.P. Cockshott, and R. Morrison, "An Approach to Persistent Programming", *Computer Journal*, 1983.

[5] J. Mccarthy, "Recursive functions of symbolic expressions and their computation by machine", *Communications of the ACM*, vol. 3, no. 4, 1960.

[6] G.E. Collins, "A method for overlapping and erasure of lists", *Communications of the ACM*, vol. 3, no. 6, 1960.

[7] D.F. Bacon, P. Cheng, and V.T. Rajan, "A Unified Theory of Garbage Collection", In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*. Vancouver, Canadá, 2004.

[8] C.J. Cheney, "A nonrecursive list compacting algorithm", *Communications of the ACM*, vol. 13, no. 2, 1970.

[9] D.M. Ungar, "Generation scavenging: A non-disruptive high performance storage reclamation algorithm", In *Proceedings of the first ACM SIGSOFTSIGPLAN software engineering symposium on Practical software development environments*, Pennsylvania, 1984.

[10] A.W. Appel, "Simple generational garbage collection and fast allocation". *Software: Practice and Experience*, vol. 19, no. 6, 1989.

[11] H. Schorr, and W.M. Waite, "An efficient machine-independent procedure for garbage collection in various list structures", *Communications of the ACM*, vol. 10, no. 8, 1967.

[12] R.A. Brooks, "Trading data space for reduced time and code space in real-time garbage collection on stock hardware", In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*. Texas, 1984.

[13] K. Zee, and M. Rinard, "Write barrier removal by static analysis", In *Proceedings of the 17th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002)*, Seattle, Washington, 2002.

[14] B. Zorn, "Barrier methods for garbage collection", *Tech. Rep. CU-CS-494-90*, University of Colorado, Boulder, 1990.

[15] A.W. Appel, J.R. Ellis, and K. Li, "Real-time concurrent collection on stock multiprocessors", In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*. Georgia, 1988.

[16] H.G. Baker, "List processing in real-time on a serial computer", *Communications of the ACM*, vol. 21, no. 4, 1978.

[17] H.G. Baker, "The Treadmill, real-time garbage collection without motion sickness", *ACM Sigplan Notices*, vol. 27, no. 3, 1992.

[18] A.M. Cheadle, A.J. Field, S. Marlow, S.L. Peyton Jones, and R.L. While, "Non-stop Haskell", In *Proc. of the Fifth International Conference on Functional Programming*, Quebec, 2000.

[19] M.S. Johnstone, "Non-Compacting Memory Allocation and Real-Time Garbage Collection", *PhD thesis, University of Texas*, Austin, 1997.

[20] M. Larose, and M. Feeley, "A compacting incremental collector and its performance in a production quality compiler", In *Proc. of the First International Symposium on Memory Management, ISMM '98*, Vancouver, British Columbia, Canada, October, 1998.

[21] T. Yuasa, "Real-time garbage collection on general-purpose machines", *Journal of Systems and Software*, vol. 11, no. 3, 1990.

[22] S. Nettles, and J. O'toole, "Real-time garbage collection", In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, Albuquerque, New México, 1993.

[23] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens, "On-the-fly garbage collection: An exercise in cooperation", *Language Hierarchies and Interfaces*, *International Summer School,* Lecture Notes in Computer Science, vol. 46, 1976.

[24] D. Doligez, and X. Leroy, "A concurrent generational garbage collector for a multi-threaded implementation of ML", In *Conf. Record of the Twentieth ACM Symposium on Principles of Programming Languages*, 1993.

[25] H.T. Kung, and S.W. Song, "An efficient parallel garbage collection system and its correctness proof", In *IEEE Symposium on Foundations of Computer Science*, 1977.

[26] L. Lamport, "Garbage collection with multiple

processes: an exercise in parallelism", In *Proc. of the 1976 International Conference on Parallel Processing*, 1976.

[27] G.L. Steele, "Multiprocessing compactifying garbage collection", *Communications of the ACM*, vol. 18, 1975.

[28] J. WEIZENBAUM, "Symmetric list processor", *Communications of the ACM*, vol. 6, 1963.

[29] L.P. Deutsch, and D.G. Bobrow, "An efficient incremental automatic garbage collector", *Communications of the ACM*, vol. 19, 1976.

[30] T.W. Christopher, "Reference count garbage collection", *Software: Practice and Experience*, vol. 14, no. 6, 1984.

[31] A.D. Martínez, R. Wachenchauzer, and R.D. Lins, "Cyclic reference counting with local mark-scan", *Information Processing Letters*, vol. 34, no.1, 1990.

[32] D.F. Bacon, P. Cheng, V.T. Rajan, "A real-time garbage collector with low overhead and consistent utilization", In *Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Louisiana, 2003.

[33] J.M. Barth, "Shifting garbage collection overhead to compile time", *Communications of the ACM*, vol. 20, 1977.

[34] D.F. Bacon, C.R. Attanasio, H.B. Lee, V.T. Rajan, and S. Smith, "Java without the coffee breaks: A nonintrusive multiprocessor garbage collector", In *Proceedings of the SIGPLAN'01 Conference on Programming Language Design and Implementation*, Utah, 2001.

[35] J. Detreville, "Experience with concurrent garbage collectors for Modula-2+", *Tech. Rep. 64*, DEC Systems Research Center, Palo Alto, November, 1990.

[36] Y. Levanoni, and E. Petrank, "An on-the-fly reference counting garbage collector for java", In *Proceedings of the 16th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001)*, Florida, 2001.

[37] D. Plainfossé, and M. Shapiro, "A Survey of Distributed Garbage Collection Techniques", In *Proceedings of International Workshop on Memory Management*, Lecture Notes in Computer Science, Springer-Verlag, vol. 986, Kinross, Scotland, September, 1995.

[38] C. Neuman, and S. Ryu, "Garbage collection for distributed persistent objects", In *Proceedings of Workshop on Compositional Software Architectures*. Monterey, California, January, 1998.

[39] Sun Microsystems, "Dynamic Proxy Classes", *<http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html>*.