Eduardo Helton Akatsu

Coleta de Lixo Distribuída no Ambiente Virtuosi

Dissertação de Mestrado apresentado ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática.

Curitiba
2007

Eduardo Helton Akatsu

Coleta de Lixo Distribuída no Ambiente Virtuosi

Dissertação de Mestrado apresentado ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática.

Área de Concentração: Sistemas Distribuídos

Orientador: Prof. Dr. Alcides Calsavara

Curitiba
2007

Akatsu, Eduardo Helton
A313c
2007
Coleta de Lixo Distribuída no Ambiente Virtuosi. / Eduardo Helton Akatsu; orientador, Alcides Calsavara– 2007.
74f. : il.; 30 cm.

Dissertação (mestrado) – Pontifícia Universidade Católica do Paraná, Curitiba, 2007
Bibliografia: f. 72-74

1. Programação orientada a objetos (Computação) 2. Sistemas Operacionais Distribuídos (Computadores) 3. Lixo - Eliminação - Processamento de dados I.Calsavara, Alcides. II.Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Informática. III.Título

CDD 21. ed. – 005.11
055.4476

Sumário

Lista de Figuras

Resumo

O projeto de pesquisa *Virtuosi* da PUCPR é um ambiente de execução distribuída de sistemas de software orientado a objetos formado por um conjunto de máquinas virtuais cooperativas. Neste ambiente de execução distribuída busca-se a abstração dos detalhes de baixo nível da programação distribuída a fim de deixar transparente a manipulação de objetos remotos e locais e facilitar a mobilidade dos objetos entre as máquinas.

Assim como em outros vários ambientes orientados a objetos existentes, a incorporação do gerenciamento automático de memória na *Virtuosi* traria ganhos no nível de abstração ao livrar o desenvolvedor da necessidade de lidar explicitamente com gerenciamento de memória. Evitando assim erros de programação na liberação da memória não utilizada. Além disso, facilita a modularização, pois a gerência manual sendo um elemento global de um sistema poderia facilmente espalhar-se por todas as interfaces de seus componentes. Desta forma, qualquer alteração neste gerenciamento afetaria todo o sistema.

A contribuição deste trabalho é propor um modelo de coleta de lixo distribuída para a *Virtuosi*. Neste modelo implementou-se dois algoritmos de coleta de lixo distribuída baseados na contagem de referências: *SSP Chains* e de *Maheshwari e Liskov* dentro do ambiente *Virtuosi*. Apresentaremos neste trabalho os dados colhidos durante os testes de ambos algoritmos como: a quantidade de mensagens trocadas e a ocupação da banda. Também, serão listadas as alterações necessárias na implementação dos modelos de objetos e de chamadas remotas de métodos da Virtuosi.

Palavras-chave: Coleta de Lixo Distribuída; Virtuosi; Sistemas Distribuídos; Máquina Virtual.

Abstract

The PUCPR research project *Virtuosi* is an environment for distributed oriented-object software systems composed of a set of cooperative virtual machines. The objective this distributed execution environment is to abstract the distributed programming low level details, thus it is makes transparent handling of remote and local objects and facilitates object mobility.

Just as many existing object-oriented systems, the addition of automatic memory management in *Virtuosi* should increase the level of abstraction, it frees the developer from explicity memory management. Therefore it avoids programming errors caused by deallocation of the non-utilized memory. Besides it facilitates modular programming because the manual memory management as a global element in the system it could easily span throughout interface components. Then any change in this management would affect the whole system.

The contribution of this work is to propose a distributed garbage collection model in the *Virtuosi*. Two algorithms of distributed garbage collections based on reference count have been implemented: *SSP Chains* and Maheshwari and Liskov in *Virtuosi* environment. We will present in this work the data gathered in test for both algorithms, such as: the number of messages and their sizes. Moreover, the necessary changes in the object model and remote procedure call model of *Virtuosi* will be listed.

Keywords: Distributed Garbage Collection; Virtuosi; Distributed System; Virtual Machine.

Capítulo 1

Introdução

1.1 Motivação

O projeto de pesquisa *Virtuosi* da PUCPR é um ambiente de execução distribuída de sistemas de software orientado a objetos formado por um conjunto de máquinas virtuais cooperativas. Neste ambiente de execução distribuída é possível abstrair-se de detalhes de baixo nível da programação distribuída. Deixar transparente a manipulação de objetos remotos e locais, facilitar a mobilidade dos objetos entre as máquinas. Estas características são desejáveis por permitir que seus usuários explorem somente os problemas da computação distribuída como: a concorrência, balanceamento de carga, sincronização e etc., que já são complexos por si só. A *Virtuosi* diferencia-se de outras arquiteturas de execução orientadas objetos devido ao sua forte integração aos princípios da orientação a objetos, dos mecanismos de comunicação entre objetos e dos mecanismos de reflexão computacional. Os benefícios desta integração são: a simplificação na compilação de linguagens orientadas a objetos, otimização do código executável, simplificação na comunicação entre objetos, portabilidade do código gerado, possibilidade de uso de linguagens diferentes e a execução em plataformas heterogêneas.

Citando [Jon96] dos objetivos principais da engenharia de software perseguidos: a abstração e a modularidade. O gerenciamento automático da memória certamente eleva o nível de abstração ao livrar o desenvolvedor da necessidade de lidar explicitamente com a desalocação da memória quando esta não é mais utilizada. Não raro, acontecerem erros de programação que podem levar a vazamentos de memória (*memory leaks*) ou na liberação prematura de memória, criando referências inválidas (*dangling pointers*). Facilita também a modularização, pois a gerência manual de memória como um elemento global de um sistema poderia facilmente espalhar-se por todas as interfaces de seus componentes. Desta forma, qualquer alteração no gerenciamento afetaria todo o sistema.

1.2 Objetivo

Alinhado com o objetivo proposto em [Cal00] de se construir um ambiente completo de execução distribuída orientada a objetos. Este trabalho apresenta uma proposta inicial de um modelo para a coleta de lixo distribuída, além propor alterações necessárias em outros módulos da *Virtuosi*. Como forma avaliação deste modelo implementou-se nele conhecidos algoritmos de coleta de lixo distribuído.

1.3 Metodologia

Para atingir o objetivo principal de implementar a coleta de lixo na *Virtuosi*, primeiro analisou-se os diversos módulos que compõe a *Virtuosi* a fim de se definir o escopo do trabalho e levantar os seus possíveis impactos durante a implementação. O próximo passo dado foi realizar um estudo dos fundamentos da coleta de lixo e realizar um levantamento dos algoritmos de coleta de lixo distribuída, com o objetivo de se escolher os melhores candidatos a serem implementados.

Escolhido o primeiro algoritmo de coleta de lixo distribuída. Este foi implementado e testes foram realizados dentro da *Virtuosi*. Com os problemas encontrados e as soluções aplicadas durante a implementação, criou-se um modelo mais genérico de coleta de lixo distribuída e nele reimplementou-se o primeiro algoritmo e um novo algoritmo. Para finalizar, foi preciso criar um ambiente de simulações para se testar os algoritmos.

1.4 Organização

Este trabalho será apresentado da seguinte forma: o capítulo 2 trata do ambiente *Virtuosi* para melhor entendimento e delimitação do escopo deste trabalho, o capítulo 3 faz uma revisão dos conceitos da coleta de lixo e a compilação de algoritmos de alguns coleta de distribuída de lixo. O capítulo 4 trata da implementação dentro da *Virtuosi* de um modelo de gerenciamento da coleta lixo distribuída. A validação deste modelo através de simulações encontra-se no capítulo 5 e o último capítulo é destinado à conclusão deste trabalho e trabalhos futuros.

Capítulo 2

Ambiente VIRTUOSI

A Virtuosi é um ambiente de execução distribuída sobre plataformas heterogêneas, cuja arquitetura é baseada nos conceitos de orientação a objetos. Conceitos que são formalizados através de um diagrama de classes da UML chamado de metamodelo da Virtuosi.

2.1 Metamodelo da Virtuosi

Nesta seção apresentaremos um resumo da versão de metamodelo da Virtuosi proposta na dissertação [Kol04]. O metamodelo da Virtuosi possui classes e associações que representam os elementos encontrados em uma linguagem de programação orientada a objeto que dê suporte aos conceitos suportados pela Virtuosi. Por isso, as classes do metamodelo podem ser chamadas de meta-classes. Para auxiliar o entendimento dos conceitos explicados ao longo dessa seção, são utilizados trechos de código fonte de uma aplicação de software orientado a objeto escritos na linguagem Aram[1] [BOR04].

O metamodelo da Virtuosi pode ser entendido como um diagrama de classes que descreve conceitos de orientação a objeto e como tais conceitos se relacionam entre si. São exemplos de meta-classes modeladas na Virtuosi:

- literais;
- referência a literal;
- referência a bloco de dados;
- referência a índice;
- classe;
- atributos;
- invocáveis (métodos, construtores);
- etc..

[1] Aram é a primeira linguagem de programação compatível aos conceitos de orientação a objeto definidos pelo metamodelo da Virtuosi e também foi desenvolvida dentro do Projeto Virtuosi

2.1.1 Literais

Valor Literal

Um valor literal é uma seqüência de caracteres sem semântica definida. A Figura 2.1 mostra o uso de valores literais como parâmetro, numa atribuição e numa comparação.

```
class Principal {
   constructor iniciar( ) exports all {
   ...
   Integer massa = Integer.make( 70 );// 70 é um valor literal
                                      // utilizado como parâmetro.
   }
   ...
}
...
class Boolean {
   enum { true, false } value = false; // false é um valor literal
                                       // numa atribuição.
   ...
   method void flip( ) exports all
   {
       if ( value == true )  // true é um valor literal
           value = false;    // numa comparação.
       else
           value = true;
   }
   ...
}
```

Figura 2.1: Código fonte em Aram mostrando os possíveis usos de um valor literal

Referência a Literal

Uma meta-classe que representa uma **referência a literal** se relaciona através de associações com outros componentes do metamodelo da Virtuosi nas seguintes situações:

1. como parâmetro formal;
2. como parâmetro real;
3. como origem de atribuição em um comando de atribuição de variável enumerada;
4. como uma das possibilidades para o segundo elemento de uma comparação de valor de variável enumerada.

2.1.2 Bloco de Dados e Índice

Referência a Bloco de dados

Uma **referência a bloco de dados** consiste em uma referência para uma seqüência contígua de dados binários em memória. Esse tipo de referência é geralmente utilizado para a construção de classes pré-definidas (*Integer*, *Real*, *Boolean*, *Character* e *String*) utilizadas para compor outras classes.

```
class Inteiro{
    datablock valor;

    method void set(Inteiro i) exports{all}
    {
        // referência k para um bloco de dados
        datablock k = i.valor;
        valor = k.clone();
    }
    ...
}
```

Figura 2.2: Código fonte em Aram de uma classe que faz um bloco de dados

Referência a Índice

Para a manipulação de um bloco de dados existe uma referência especial chamada **referência a índice**. Um índice pode ser obtido a partir de uma referência a bloco de dados. O intervalo de valores de um índice fica entre zero e o tamanho do bloco de dados menos um.

2.1.3 Classes

A meta-classe que representa uma **classe** se relaciona através de associações com outros componentes do metamodelo da Virtuosi nas seguintes situações:

1. como possuidora de atributos referência a objeto;
2. como possuidora de atributos referência a bloco de dados;
3. como possuidora de atributos de variáveis enumeradas;
4. como tipo de uma referência a objeto;
5. como possuidora de invocáveis;[2]
6. como cliente da lista de exportação de um invocável;

[2]Do inglês ***invocable*** (o termo **invocável** ainda não está registrado nos dicionários da língua portuguesa).

Herança de Classes

Duas classes podem estabelecer um relacionamento de herança entre si. Segundo o metamodelo da Virtuosi, uma classe pode ter apenas uma classe ancestral direta[3], mas pode ter muitas classes herdeiras recursivamente. Entretanto, uma classe não pode ser direta ou indiretamente ancestral de si própria.

Existem dois tipos de classe, a **classe de aplicação** e a **classe raiz**. Toda classe da aplicação possui uma classe ancestral, sendo que esta pode ser uma outra classe de aplicação ou a classe raiz.

A classe raiz representa a classe ancestral – direta ou indireta – de todas as classes de aplicação, por este motivo não possui ancestral. Ela é única em toda a hierarquia de classes.

2.1.4 Atributos

Os atributos de uma classe segundo o metamodelo da Virtuosi podem ser de três tipos, a saber:

- referência a objeto;
- referência a bloco de dados;
- variável enumerada.

Referência a Objeto

A meta-classe que representa uma **referência a objeto** se relaciona com outros componentes do metamodelo da Virtuosi em várias situações. A seguir são relacionadas algumas delas:

1. como parâmetro;
2. como atributo;
3. como tipo de classe;
4. em comparações;
5. em atribuições;
6. como referência retornada pela invocação de um método ou ação;
7. como variável local;
8. como alvo de invocação de método;
9. como alvo de invocação de uma ação.

[3]Essa propriedade é normalmente denominada herança simples, em contra-partida à herança múltipla, quando uma classe pode ter muitas ancestrais diretas.

2.1.5 Invocáveis

Segundo o metamodelo da Virtuosi, uma operação de uma classe pode ser implementada de três maneiras, a saber:

- como um **método**;
- como uma **ação**;
- e como **construtor**.

Método – É a forma mais comum de implementar uma operação definida por um *TAD* (tipo abstrato de dado). Podendo ser de dois tipos: com retorno de valor e sem retorno de valor.

Ação – Uma ação pode ser vista como uma operação cujo retorno é utilizado para uma tomada de decisão. Em outras palavras, o retorno de uma ação permite ao comando de desvio decidir dentre duas seqüências de comandos deve ser interpretada.

Construtor – Uma classe deve implementar este método especial para criar novas instâncias.

2.1.6 Comandos

Na Virtuosi, toda a computação é realizada através da interpretação dos comandos que compõem um invocável. Num invocável podemos encontrar comandos de:

- declaração de variáveis;
- atribuição;
- invocação de outros invocáveis.

Os comandos podem ser **simples**, como uma atribuição. Ou **comandos compostos** que são seqüências de comandos simples ou compostos.

2.2 Árvores de Programas

Nesta seção será apresentada de forma resumida a utilização das árvores de programa [Kol04] no ambiente *Virtuosi*.

Como em algumas máquinas virtuais por exemplo Java, onde o código fonte não é executado diretamente, mas antes compilado para uma representação intermediária chamada *bytecode*. Na Virtuosi, utilizou-se a idéia de árvores sintáticas abstratas encontradas em [KF97] para criar uma representação intermediária própria no formato de **árvore de programa**. Uma árvore de programa pode ser vista com um grafo cujos nós são objetos que representam os elementos encontrados em uma linguagem de programação orientada a objeto, ou seja, uma árvore de programa é um grafo cujos nós são instâncias das metaclasses definidas pelo metamodelo da Virtuosi e as ligações entre estes nós são as relações

```
class Pessoa {
   composition Integer posicao;
   ...

   method void setPosicao(Integer p) exports all {
       posicao = p;
   }
   ...
}
```

Figura 2.3: Fragmento de código fonte da classe `Pessoa`

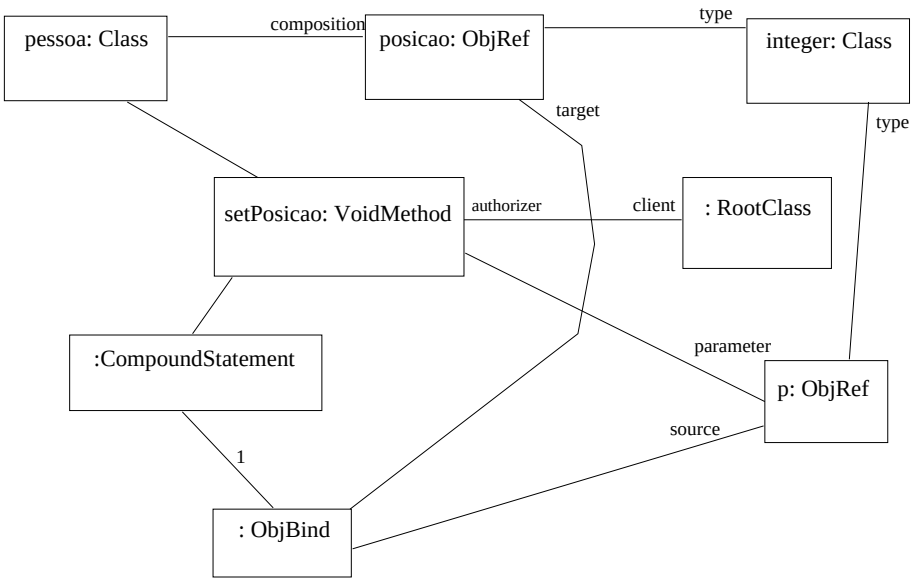

Figura 2.4: Árvore de programa parcial referente à classe Pessoa

entre tais meta-classes. Desta forma, quando o código de uma nova classe é carregada na Virtuosi é criada uma árvore de programa que a representaria.

A figura 2.4 mostra a árvore de programa correspondente ao código fonte da figura 2.3 através de um diagrama de colaboração UML.

Com base na Figura 2.4, deve-se notar que:

1. Existe um objeto chamado `pessoa` que é uma instância da meta-classe utilizada para representar uma classe de aplicação. Este objeto representa a classe de aplicação `Pessoa`;

2. Associado ao objeto chamado `pessoa` existe um objeto instância da meta-classe utilizada para representar uma referência a objeto, chamado `posicao`. Este objeto está ligado ao objeto chamado `pessoa` por uma associação e realiza o papel de atributo por composição;

3. O objeto chamado `pessoa` também possui uma associação com um objeto chamado

`setPosicao` que é instância da meta-classe que representa um método sem retorno. Esse objeto – que representa um método sem retorno da classe de aplicação `Pessoa` – por sua vez, possui uma associação com um objeto instância da meta-classe que representa uma referência a objeto chamado `p` que no caso é do tipo *Integer* e tem o papel de parâmetro do método *setPosicao*;

4. O objeto chamado `setPosicao` possui uma associação com um objeto chamado `raiz` que é instância (única em todo o sistema) da meta-classe que representa a classe raiz. Essa associação é responsável por definir a lista de exportação do método em questão, ou seja, quais as outras classes de aplicação que podem invocar o método em questão. Neste caso especifico, qualquer classe poderá invocar o construtor;

5. Observa-se que o objeto chamado `setPosicao` possui uma associação com um objeto não nomeado instância da meta-classe que representa um comando composto, e este, por sua vez, possui associação com um objeto instância da meta-classe que representa um comando simples de atribuição de referência a objeto. O objeto que representa um comando de atribuição possui duas associações com objetos instância da meta-classe que representa referência a objeto, um representando o alvo da atribuição e o outro representando a origem da atribuição, neste caso específico o par: `posicao` – `p`.

Deve-se notar que os dois objetos instância da meta-classe que representa uma referência a objeto – `posicao` e `p` – possuem uma associação com um objeto instância da meta-classe que representa uma classe de aplicação. Esta associação indica a classe de aplicação da qual a referência a objeto em questão é instância, neste caso específico a classe pré-definida *Integer*. Nota-se, portanto, que a árvore de programa em questão não está completa, uma vez que a classe pré-definida *Integer* não consta no diagrama. Isto ocorre porque cada classe de aplicação ou classe pré-definida define uma árvore de programa particular.

2.3 Arquitetura *Virtuosi*

A figura 2.5 mostra os principais componentes dentro da arquitetura da Virtuosi, e as as dependências entre eles.

O ambiente Virtuosi tem como um de seus princípios fundamentais o uso de máquinas virtuais distribuídas. O ambiente Virtuosi, portanto, define uma máquina virtual – a **Máquina Virtual Virtuosi (MVV)**.

A tarefa básica de uma instância da MVV é interpretar os comandos definidos por invocáveis pertencentes a uma árvore de programa. O restante dessa Seção tem como objetivo mostrar os principais elementos da Máquina Virtual Virtuosi e como estes interagem.

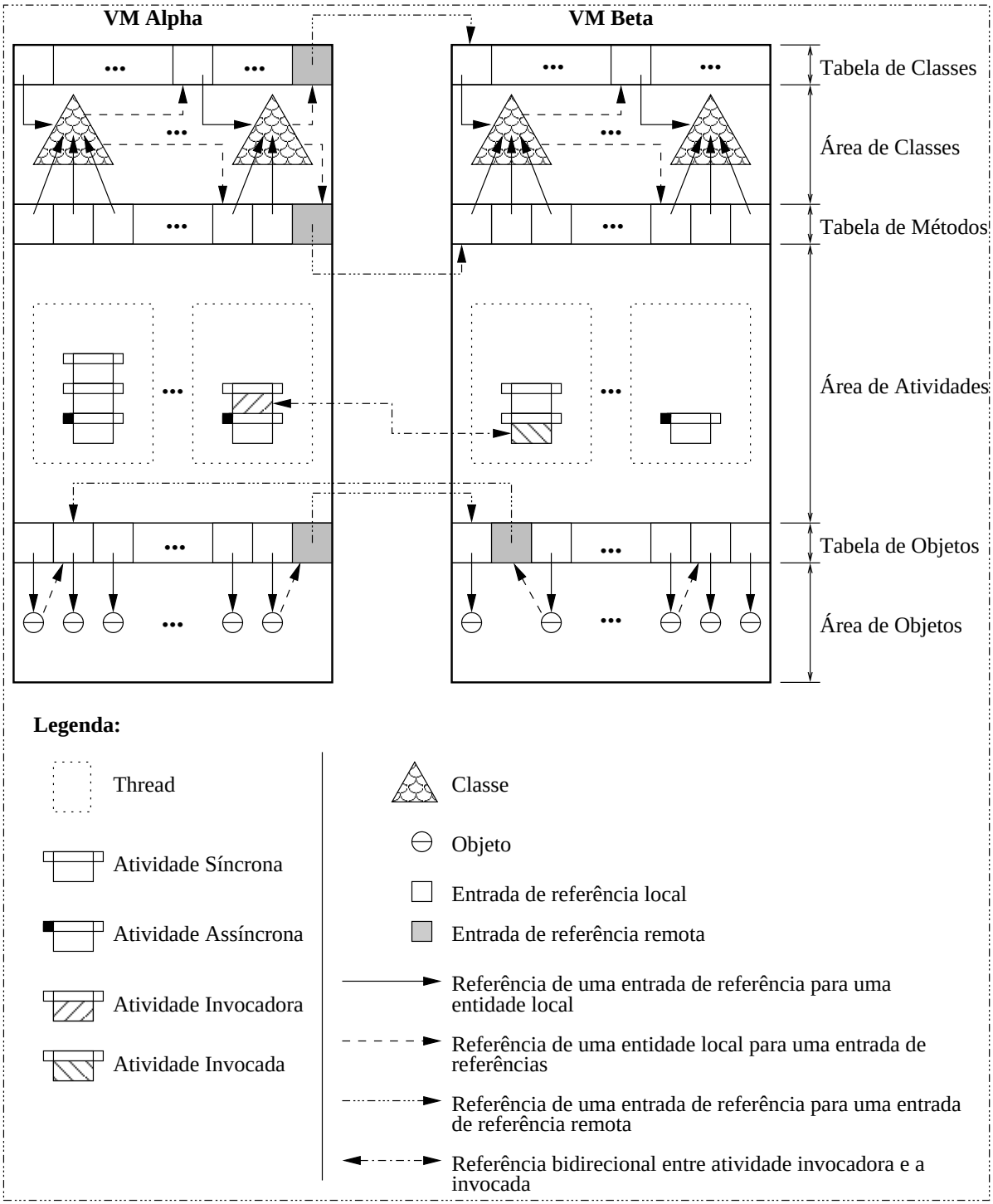

Figura 2.5: Visão Geral da Arquitetura da Máquina Virtual Virtuosi

2.3.1 Área de Classes

A área de classes é a região de memória na qual as árvores de programa de cada uma das classes de aplicação são armazenadas.

Dentro da MVV as referências entre árvores sempre são feitas de forma indireta, ou seja, os pontos de ligação entre duas árvores de programa – os objetos instâncias da meta-classe que representa uma referência a objeto e os objetos instâncias da meta-classe que representa uma invocação de invocável – nunca apontam diretamente para seus respectivos alvos – um objeto da meta-classe classe e um objeto da meta-classe invocável. Essas duas ligações na perspectiva da aplicação significam, respectivamente, o tipo de um objeto e o método de uma invocação.

No caso dos objetos instâncias da meta-classe que representa uma referência a objeto, a ligação com o seu respectivo tipo é feita de forma indireta através de uma tabela chamada **tabela de classes**. Esse tipo de estratégia foi utilizado por [HCWZ03] e é utilizada devido à natureza distribuída do ambiente Virtuosi, na qual uma referência a objeto pode ser de um tipo (uma classe de aplicação) cuja árvore pode estar localizada na própria instância da MVV ou localizada remotamente em outra instância da MVV. A figura 2.3.1 exemplifica este referenciamento indireto realizado pela tabela de classes, na qual existem entradas locais para classes localizadas na própria MVV e também entradas remotas para classes referenciadas numa MVV remota.

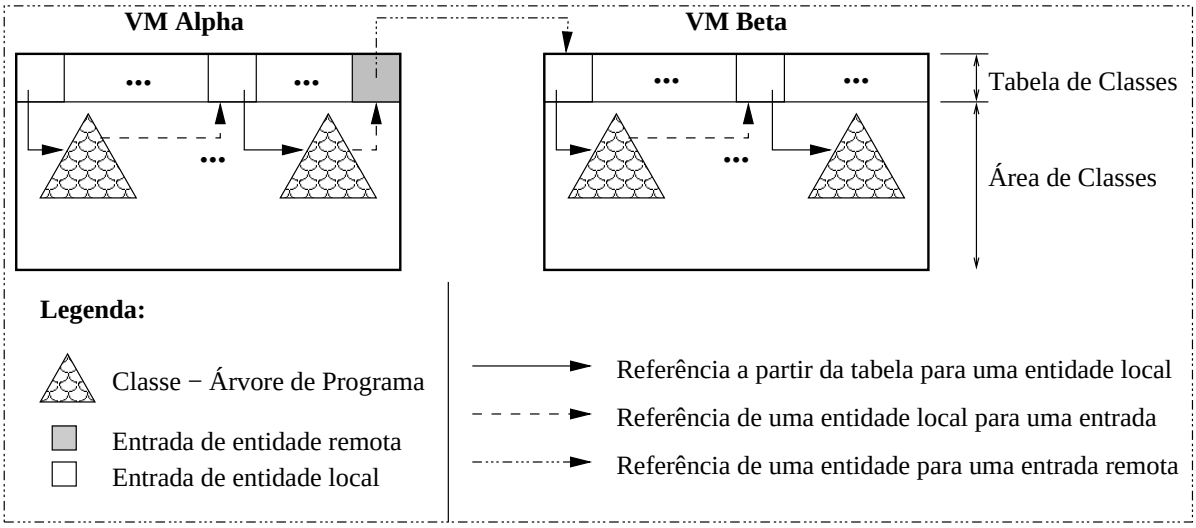

Figura 2.6: Tabelas de classes com referências locais e remotas

Tabela de Classes

Cada uma das classes de aplicação carregadas na MVV tem uma entrada correspondente na tabela de classes. Cada entrada possui o nome da classe de aplicação para a qual aponta e um apontador para a área de memória na qual a árvore está localizada. Uma árvore pode referenciar uma árvore localizada em outra máquina virtual. Portanto, existem dois tipos de entrada na tabela de árvores, a saber:

- Entrada de Referência de Classe Local (ERCL);
- Entrada de Referência de Classe Remota (ERCR) – neste caso a entrada também armazena o nome da classe de aplicação, mas ao invés de possuir um apontador para uma área de memória, a entrada possui a identificação da MVV remota e a posição da entrada da tabela de árvores remota.

No caso dos objetos instâncias da meta-classe que representa uma invocação de invocável a ligação também é feita de forma indireta através de uma tabela chamada **tabela de invocáveis**.

Tabela de Invocáveis

Cada um dos invocáveis de cada uma das classes de aplicação carregadas na MVV tem uma entrada correspondente na tabela de invocáveis. Cada entrada possui o nome do invocável apontado, o nome da classe de aplicação que possui o invocável e um apontador para a área de memória na qual o invocável está localizado. Como pode ser visto na figura 2.7, é possível invocar um invocável de um objeto local ou de um objeto remoto se o apontador referenciar uma entrada para um invocável remoto.

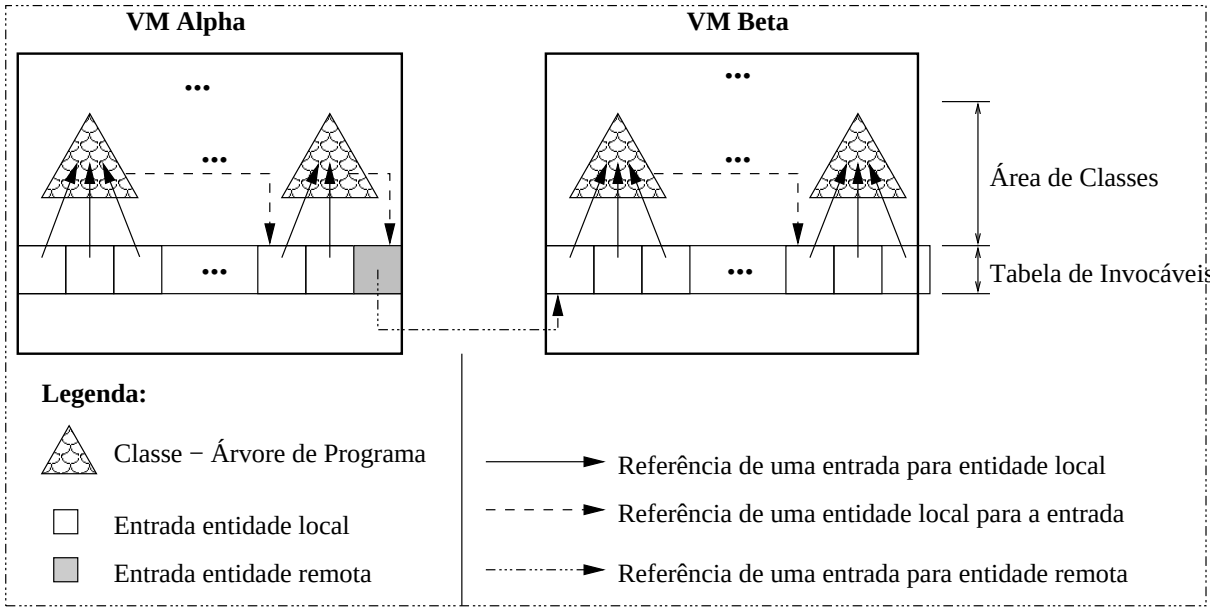

Figura 2.7: Tabela de invocáveis com referências locais e remotas

Portanto, existem dois tipos de entrada na tabela de invocáveis, a saber:

- Entrada de Referência de Invocável Local (ERIL);
- Entrada de Referência de Invocável Remoto (ERIR) – neste caso, a entrada armazena a identificação da MVV remota e a posição da entrada da tabela de invocáveis remota.

2.3.2 Área de Objetos

A área de objetos é a região de memória na qual objetos instâncias de classes de aplicação são armazenados.

Dentro da MVV as referências entre objetos sempre são indiretas, ou seja, um objeto nunca referencia diretamente um outro objeto. A ligação entre os objetos é feita sempre através da **tabela de objetos**. Essa estratégia é utilizada devido à natureza distribuída do ambiente Virtuosi, no qual um objeto pode estar localizado na própria instância da MVV ou remotamente em outra instância da MVV.

Tabela de Objetos

Para cada um dos objetos que a MVV instancia, há uma entrada correspondente na tabela de objetos. Cada entrada possui o nome do objeto o qual aponta, o nome do objeto contentor (no caso do objeto estar contido em outro objeto) e um apontador para a área de memória na qual o objeto está localizado.

Um objeto pode referenciar um objeto localizado em outra máquina virtual. Portanto, existem dois tipos de entrada na tabela de objeto como pode ser visto na figura 2.8, a saber:

- Entrada de Referência Objeto Local (EROL);
- Entrada de Referência Objeto Remoto (EROR) – neste caso a entrada também armazena o nome do objeto e nome do objeto contentor (se for o caso), mas ao invés de possuir um apontador para uma área de memória, a entrada possui a identificação da MVV remota e a posição da entrada da tabela de objetos remota.

A Entrada de Referência de Objeto Local, tem outra atribuição além de que somente referenciar o objeto. Existe um campo de um bit denomidado *freeze* em sua estrutura. Quando o campo *freeze* está igual a zero indica que o objeto está disponível para alteração, senão o objeto fica indisponível e as alterações devem guardar até que o objeto esteja disponível novamente [dCCF04].

2.3.3 Área de Atividades

Conforme descrito em [Cal00], a Virtuosi tem como forma mais básica de comunicação entre objetos a invocação de uma atividade de um objeto por outro objeto. Esse mecanismo possibilita a concepção de uma aplicação baseada no encadeamento de atividades de um conjunto de objetos.

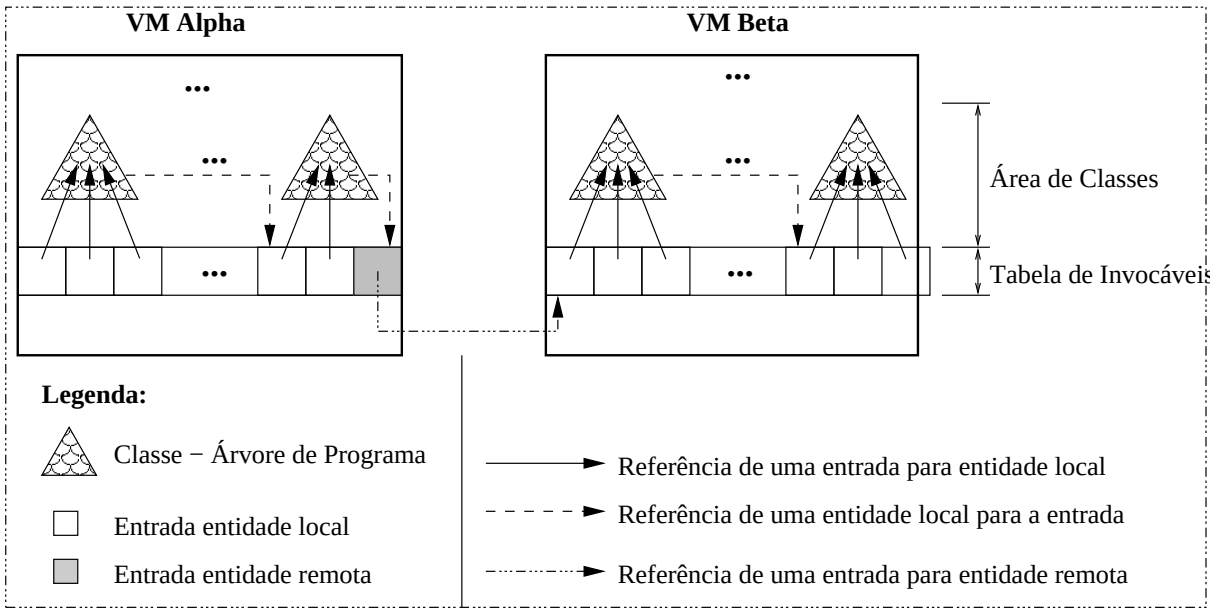

Figura 2.8: Tabela de Objetos com entradas locais e remotas de objetos

E, a área de atividades é a região de memória da MVV na qual as pilhas de atividade são armazenadas.

Atividade de Objeto

Uma **atividade** de um objeto corresponde à interpretação de um de seus invocáveis (construtor, método ou ação), ou seja, cada invocação de invocável de um certo objeto dá início a uma nova atividade deste objeto. Toda a computação realizada pela MVV é obtida através da interpretação dos comandos definidos por um invocável. Uma atividade termina quando a interpretação do invocável termina, seja normal ou anormalmente (quando ocorre uma exceção. Duas invocações de dois métodos distintos dão início a duas atividades independentes. Da mesma forma, duas invocações do mesmo invocável também dão início a duas atividades independentes. Assim, para cada instante no tempo, cada objeto na MVV pode ter zero ou mais atividades, dependendo de como são utilizados pelas aplicações. Um mesmo objeto pode, inclusive, ter duas atividades simultâneas pertencentes a aplicações distintas. Nesse modelo de execução, uma aplicação consiste em um encadeamento de atividades, envolvendo um conjunto de objetos relacionados. Cada aplicação determina quais objetos são relacionados, que invocável invoca qual invocável, em que ordem e sob quais condições ocorrem as invocações e ainda, para cada par de **atividade invocadora** e **ativividade invocada** o modo de invocação com relação ao sincronismo.

Em relação ao sincronismo, uma atividade invocada pode ser classificada como **síncrona** ou **assíncrona**.

Atividade Síncrona A atividade invocadora fica bloqueada até que a atividade termine. Necessariamente, então, a atividade invocadora tem que ser notificada do término da

atividade invocada, seja ele normal (com um possível retorno) ou anormal (com geração de exceção).

Atividade Assíncrona A atividade invocadora não fica bloqueada devido à invocação e nem recebe qualquer notificação de término. Assim, a atividade invocadora pode encerrar-se independentemente do que aconteça com a atividade invocada. Nesse caso, se a atividade tiver um retorno, este será ignorado. Igualmente, se gerar alguma exceção, esta não será notificada na atividade invocadora.

Deve-se observar que o modo de sincronismo para um certo invocável não precisa ser fixo, isto é, pode ser escolhido em tempo de execução pela atividade invocadora. Assim, é possível que um mesmo invocável seja interpretado como atividade síncrona em uma situação e como atividade assíncrona em outra.

Estrutura de uma Atividade

Uma atividade é sempre referente a um dos invocáveis de um objeto de uma classe de aplicação. Portanto, a estrutura de uma atividade na MVV é composta por:

1. um apontador para o objeto dono da atividade localizado na área de objetos;
2. um apontador para o invocável que está sendo interpretado;
3. um conjunto de parâmetros;
4. um conjunto de variáveis locais.

A figura 2.9 mostra a representação gráfica de uma atividade de um objeto.

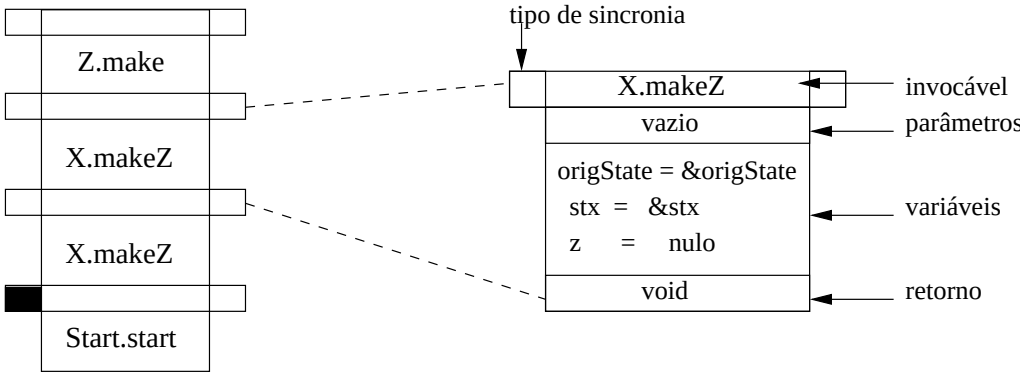

Figura 2.9: Exemplo de uma pilha de execução e da estrutura de uma atividade

Pilha de Atividades

Quando uma atividade interpreta um comando de invocação de um construtor, método ou ação é criada uma atividade para a interpretação do invocável em questão. A invocação de uma atividade síncrona causa o seu empilhamento sobre uma outra atividade

também síncrona. O desempilhamento da atividade ocorre ao final de sua interpretação. Este processo recursivo de invocação de atividades faz com que todas as atividades síncronas sejam empilhadas na mesma pilha de atividades onde a atividade invocadora existe.

Quando uma atividade a ser invocada é assíncrona, então é criada uma nova pilha de atividades, por isso na área de atividades podem existir várias pilhas de atividades sendo executadas simultaneamente.

Uma nova atividade síncrona ou assíncrona pode ser criada em uma outra instância da MVV. No caso de uma atividade assíncrona uma nova pilha de atividades é criada remotamente e a nova atividade é colocada como o primeiro elemento da pilha. No caso de uma atividade síncrona a nova atividade também é colocada como base da pilha de atividades na MVV remota, porém somente ao término da atividade invocada remotamente, a atividade invocadora local é desbloqueada e continua a sua interpretação normalmente.

2.3.4 Ciclo de Vida de Uma Aplicação

Uma instância da MVV pode interpretar mais de uma aplicação Virtuosi ao mesmo tempo.

Para dar início à interpretação de uma aplicação, deve-se informar à máquina virtual o nome de uma classe de aplicação – classe inicial – e o nome de um construtor desta classe – construtor inicial. A partir dessas duas informações o ciclo de vida de uma aplicação na MVV segue uma seqüência de eventos bem definida:

1. A máquina virtual utiliza o subsistema de carga de árvore para carregar as árvores de programa que compõem a aplicação para a **área de classes**;
2. A máquina virtual cria um novo objeto instância da classe inicial – objeto inicial – e adiciona-o na **área de objetos**;
3. A máquina virtual cria uma nova **atividade** – atividade inicial – associada ao construtor inicial e ao objeto inicial;
4. A máquina virtual empilha a atividade inicial na **pilha de atividades**, o que faz com que a atividade inicial seja interpretada. Novas atividades são empilhadas – recursivamente – na pilha de atividades em resposta à interpretação de comandos de invocação de método e construtor ou em resposta a um comando de desvio condicional cujo teste seja uma ação;
5. Após o término da interpretação da atividade inicial a máquina virtual a retira do topo da pilha de atividades e caso não existam outras pilhas de atividades com atividades empilhadas, a aplicação é encerrada.

2.4 Mobilidade na *Virtuosi*

Nesta seção apresentaremos um resumo do modelo de mobilidade de objetos proposto na dissertação [dCCF04].

2.4.1 Primitivas

A mobilidade dos objetos na *Virtuosi* é suportada por cinco primitivas básicas da linguagem que são associadas através de invocáveis, a saber:

- `fix()`: fixa o objeto na MVV local.
- `unfix()`: torna o objeto móvel.
- `refix(MVVNome)`: move e fixa o objeto na MVV cujo identificador corresponde ao objeto MVVNome.
- `locate():MVVNome`: retorna o objeto MVVNome que identifica a MVV onde o objeto se encontra.
- `move(MVVNome)`: move o objeto na MVV cujo identificação é representada pelo objeto MVVNome.

O único pré-requisito para que um objeto possa ser migrado, é que ele não esteja fixado por meio primitiva. Caso a primitiva *move* seja invocada para um objeto fixado, será disparado uma *exceção*.

2.4.2 Protocolo de Mobilidade

A movimentação do objeto ocorre sempre quando a primitiva `move`. Este cenário é descrito pelas figuras 2.10, 2.11 e 2.12 para demonstrar como ocorre o gerenciamento das referências na Virtuosi.

A figura 2.10 representa o estado inicial deste cenário, no qual os objetos **A** e **B** estão na MVV **Vênus**. Sendo que **A** referencia **B**, por isso **A** tem um ponteiro **p** que aponta para a Entrada Local **j** do objeto **B**.

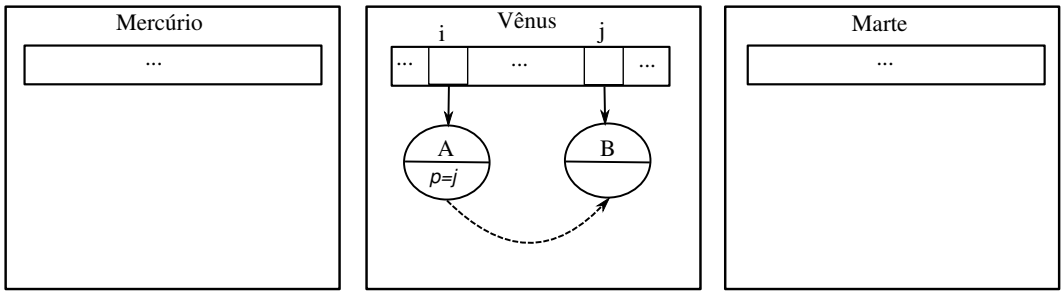

Figura 2.10: Início com objetos em Vênus

Na figura 2.11 ocorre a migração do objeto **B** para a MVV **Marte**. Por esta razão, na tabela de objetos de **Marte** é criada uma Entrada Local **k** para **B**. E, para que a referência de **A** para **B** se mantenha, a Entrada Local **j** em **Vênus** é zerada e é preenchida a sua Entrada Remota (representada na figura pela cor cinza) que aponta para **k**. Deste modo, não foi preciso alterar o ponteiro **p** para apontar diretamente para **k** em **Marte**. Assim, qualquer invocação de método de **B** realizada pelo objeto **A** passará a ser remota ao se descobrir que a entrada **j** agora é remota.

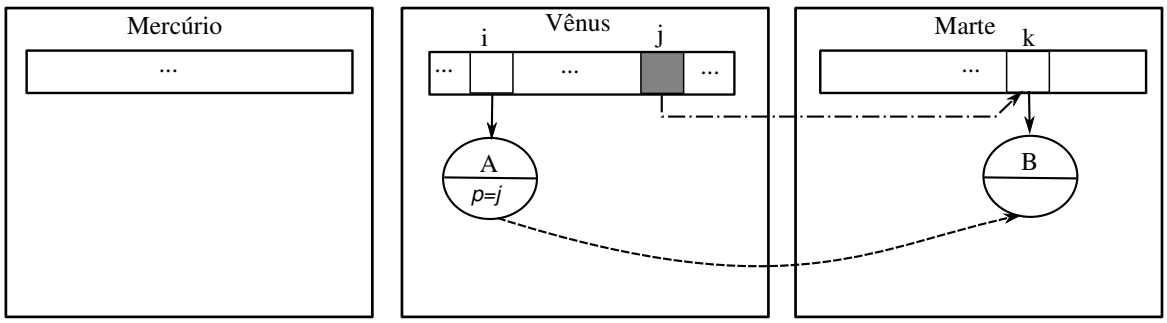

Figura 2.11: Migração do objeto B para Marte

Na figura 2.12 o objeto **A** migra para a MVV **Mercúrio**. E como ocorreu na migração do objeto **B**, é criada a entrada local **m** na tabela de objetos de **Mercúrio** para o objeto **A**, e é preenchida a Entrada Remota de **i** em **Vênus** referenciando **m**. Além disso, criou-se a entrada **n** na tabela de objetos de **Vênus** que representa a referência de **A** para **B**. Assim, o ponteiro **p** aponta **n**, cuja Entrada Remota aponta para **j** em **Vênus** que tem a Entrada Remota apontando **k** em **Marte**.

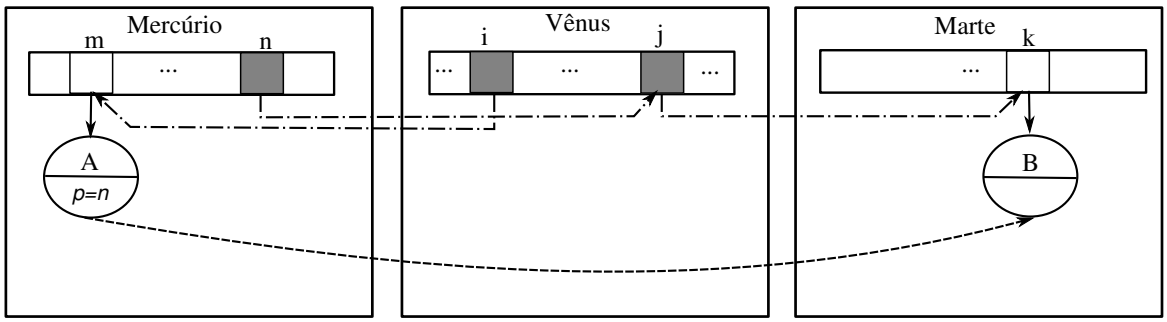

Figura 2.12: Migração do objeto A para Mercúrio

Além da alteração das entradas na tabela de objetos a fim de manter corretas referências entre os objetos, também as classes precisam ser migradas. A migração de uma classe ocorre quando a classe de um objeto a ser migrado não existe na MVV destino. Mas, diferente do acontece com os objetos, há uma replicação da classe para a máquina destino. E, se existirem referências para classes que não existam na MVV destino são criadas referências remotas na tabela de classes, e entradas remotas na tabela de invocáveis também para os invocáveis não existentes localmente.

2.4.3 Fragmentação de Referências

Uma referência remota entre duas entidades que precise percorrer mais de duas *referências intermediárias*[4] é considerada uma referência fragmentada. Na figura 2.13 temos um exemplo de uma fragmentação no qual o objeto **A** na MVV Mercúrio referencia **B** na MVV Marte, pois a referência **p** precisa percorrer as entradas **j**, **k** e **m** até alcançar o objeto **B**. A fragmentação de referências se dá pela movimentação dos objetos. Quanto mais os objetos se movimentarem pela rede maior tende a ser fragmentação de suas referências. A fragmentação de referências como conseqüência pode aumentar o número de acesso à rede e também o uso de memória pois existirão mais entradas para se referenciar um objeto.

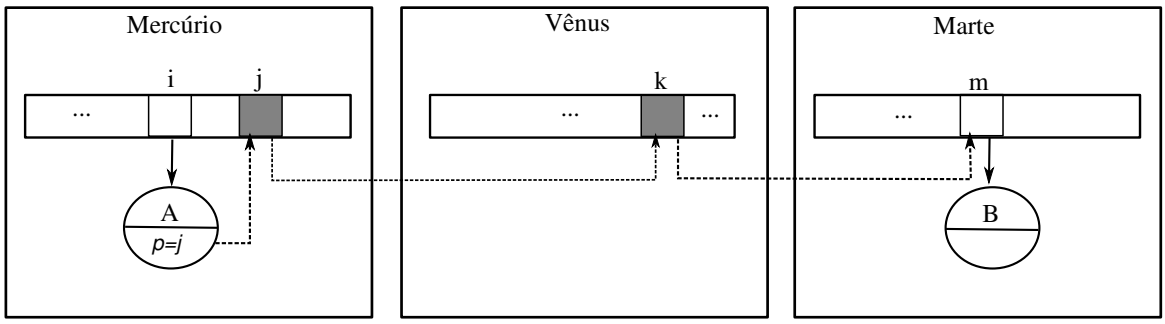

Figura 2.13: Exemplo de fragmentação de referências

[4]Estruturas que tem como objetivo redirecionar referências, não sendo o objeto final do acesso. Na *Virtuosi* todas as entradas das tabelas de referências são consideradas intermediárias

2.4.4 Mecanismo de Mobilidade de Atividades

A Virtuosi permite o que objetos com atividades sejam migrados. A implementação deste mecanismo tem um custo caro para o sistema, já que é preciso relacionar as atividades com o objeto que os gerou. A abordagem escolhida para a *Virtuosi* é semelhante à adotada pelo *Emerald* [Hut87]. Resumindo, o mecanismo congela a atividade a ser migrada e todas as outras dependentes do objeto gerador da atividade. A atividade que invocou localmente a atividade a ser migrada, é transformada numa atividade invocadora remota. E, se existir uma atividade invocada localmente será transformada numa atividade invocada remotamente.

Na figura 2.14 temos o cenário inicial antes da migração das atividades do objeto **x** para Vênus. Os retângulos **E** e **W** em cinza representam atividades remotas sejam elas invocadoras ou invocadas.

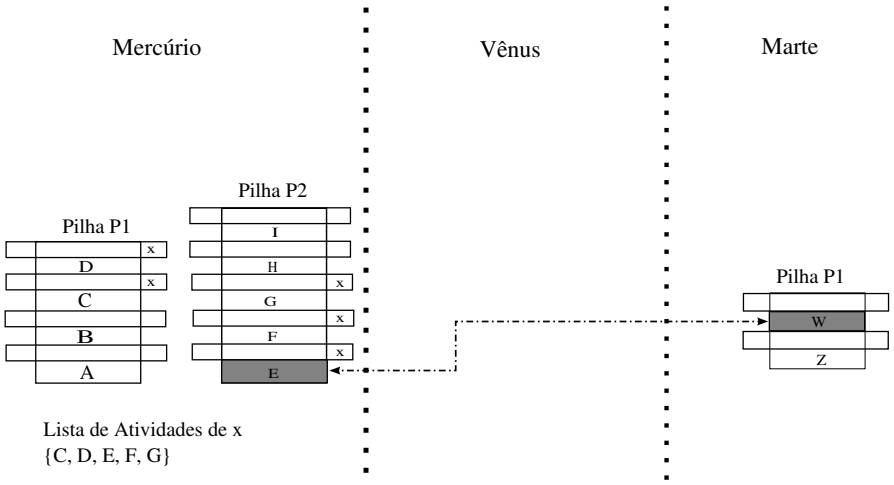

Figura 2.14: Cenário inicial de migração das atividades do objeto X

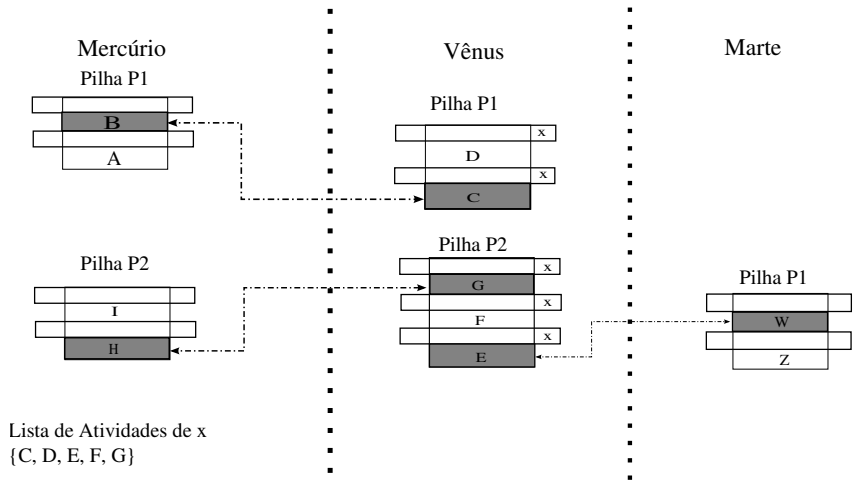

Figura 2.15: Cenário após a migração das atividades do objeto X

É mostrada na figura 2.15 como as atividades são organizadas uma vez que o objeto **x** é migrado para a MVV Vênus. São criadas duas pilhas em Vênus, pois as atividades **A** e **B** se encontravam numa pilha diferente de **E**, **F** e **G** em Mercúrio. E, as atividades **C** e **G** são transformadas em invocadoras remotas; e em Mercúrio as atividades **B** e **H** são transformadas em atividades invocadas remotamente.

2.5 Invocação de Métodos Remotos na *Virtuosi*

Nesta seção apresentaremos o resumo do modelo de invocação remota proposto na dissertação [Nod05].

2.5.1 Protocolo

O mecanismo de comunicação entre as máquinas virtuais e da máquina virtual com o serviço de nomes está baseados em *sockets* sob o protocolo TCP. O fato da máquina virtual *Virtuosi* utilizar *sockets* permite a execução de várias máquinas virtuais em mesmo computador utilizando portas diferentes.

A solitação de execução de procedimentos é realizado através do envio de mensagens. As mensagens podem ser trocadas entre máquinas virtuais e entre máquinas virtuais e o serviço de nomes.

2.5.2 Invocação de Métodos Remotos

Durante o procedimento de inicialização de uma MVV é atribuída um nome. Em seguida, é obtido um número de porta livre para escutar a requisições de invocação de métodos remotos. Para que uma máquina virtual se comunique com outra máquina virtual é necessário que ocorram registros das máquinas virtuais no servidor de nomes. Então, se

cria uma *thread* para a execução dos métodos.

O processo de chamada de um método remoto ocorre de forma síncrona, ou seja, a MVV invocadora fica aguardando o término da execução do método remoto para continuar a execução. E, a passagem de parâmetros pode ser por referência (nome da máquina virtual e o índice na tabela de objetos) ou por valor se o objeto for do tipo literal. Para descobrir se o alvo de um método é um objeto remoto ou local, verifica-se se a entrada na Tabela de Objetos é do tipo EROR (Entrada de Referência Objeto Remoto), a mesma verificação para métodos remotos, neste caso a entrada deve ser do tipo EIR (Entrada para Invocável Remoto) na Tabela de Invocáveis.

E através da invocação de um método remoto com retorno ou passando um objeto como parâmetro para um método remoto, são outras maneiras de se obter referências para objetos remotos, além da migração de objetos.

Invocação de Método Remoto com Retorno

Em uma invocação de método remoto com retorno, o valor retornado do método remoto contém apenas informações de localização do objeto[5] que é retornado. Assim que a máquina virtual recebe estas informações, ela cria uma entrada de objeto remoto (EROR) na tabela de objetos.

Na figura 2.16 temos um cenário no qual é invocado o método `obterConta()` do objeto remoto `cliente` que retornará uma referência para um outro objeto remoto, `conta1`.

```
//Invoca método remoto com retorno
Conta conta1 = cliente.obterConta();
```

Figura 2.16: Trecho de código com a invocação de um método remoto com retorno

[5]Estas informações são: índice da tabela de objetos e o nome da máquina virtual.

A figura 2.17 mostra o cenário antes da invocação do método `obterConta()`. Vemos que existe somente uma entrada remota **x** na tabela de objetos de Mercúrio para o objeto cliente em Vênus.

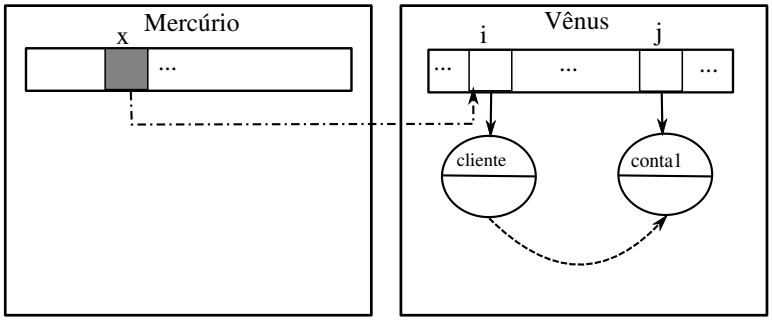

Figura 2.17: Cenário antes da invocação do método remoto com retorno

Já na figura 2.18 podemos notar que após a invocação do método `obterConta()`, foi criada uma entrada uma entrada remota **y** na tabela de objetos de Mercúrio para o objeto **conta1** retornado por Vênus.

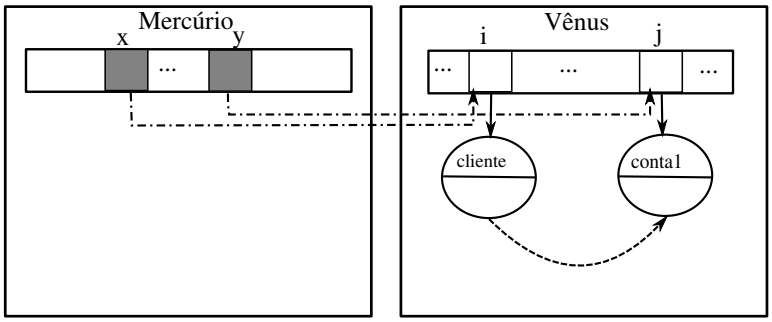

Figura 2.18: Cenário após a invocação do método remoto com retorno

Invocação de Método Remoto com Parâmetro Passado por Referência

Em uma invocação de método remoto que possua parâmetro, o que é passado como parâmetro na verdade são informações de localização (índice na Tabela de Objetos e nome da máquina virtual), a menos que os parâmetros sejam do tipo *literal*, pois neste caso é passado como valor. Desta forma a máquina que receba como a referência de um objeto criará em sua tabela de objetos um entrada de objeto remoto (EROR).

```
// Substitui nova conta, invocação método remoto com parâmetro
Conta contaNova = Conta.make( 5678 );
cliente.atualizarConta(contaNova);
```

Figura 2.19: Trecho de código de invocação de um método remoto com passagem de uma referência de um objeto como parâmetro

Na figura 2.20 vemos que não existe nenhuma entrada remota na tabela de objetos de Vênus para o objeto **contaNova**, antes da invocação do método remoto `atualizarConta` do objeto `conta`.

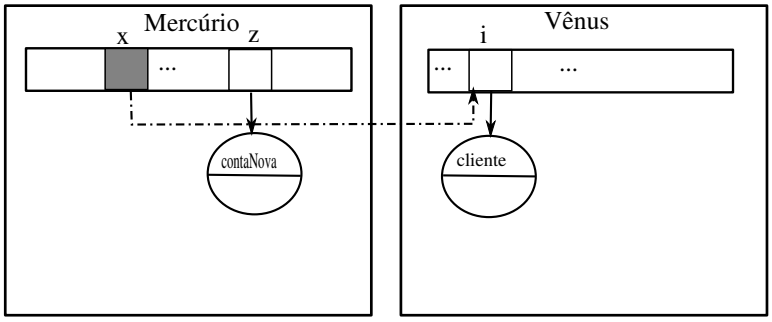

Figura 2.20: Cenário antes da invocação do método remoto com parâmetro

A figura 2.21 representa o momento após o término da execução do método `atualizarConta`. Pode ser constatada a criação da entrada remota **k** na tabela de objetos de Vênus para o objeto remoto **contaNova** em Mercúrio.

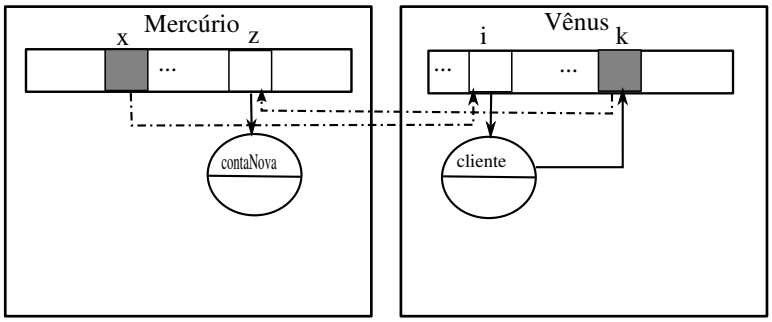

Figura 2.21: Cenário depois da invocação do método remoto com parâmetro

Capítulo 3

Fundamentos da Coleta de Lixo

Este capítulo está dividido da seguinte na forma: nas primeiras seções 3.1, 3.2 e 3.3 encontram-se a revisão dos conceitos da teoria da coleta de lixo; e nas próximas seções são apresentados algoritmos de coleta de lixo distribuída.

3.1 Visão Geral da Coleta de Lixo

3.1.1 Coleta de Lixo

A coleta de lixo segundo [Jon96], é o gerenciamento automático da memória dinamicamente alocada.

3.1.2 Mutator

Mutator de acordo com a terminologia de Dijkstra, seria a aplicação do usuário, cujo papel é alterar ou mutar o grafo de conexões entre as estruturas de dados ativas no *heap*.

3.1.3 Abstração Duas-Fases

A coleta de lixo deve recuperar automaticamente o espaço ocupado pelos dados que o *mutator* nunca poderá mais acessar. Tais dados são considerados como *lixo*. O funcionamento básico de um coletor de lixo, consiste basicamente de duas partes [Wil92]:

1. Distingüir os objetos vivos do lixo de alguma maneira, e;
2. Recuperar o espaço de armazenamento ocupado pelo lixo, para que a aplicação do usuário possa reutilizá-lo.

O estado "vivo" é definido a partir de um *conjunto raiz* e a *alcançabilidade* a partir destas raízes. Quando ocorre a coleta de lixo, todas as variáveis globais dos procedimentos ativos são consideradas *vivas*, também qualquer variável local de qualquer procedimento ativo. O *conjunto raiz* conseqüemente é constituído das variáveis globais, variáveis locais na pilha de ativação e registradores utilizados pelos procedimentos ativos. Os objetos no *heap* que são diretamente alcançáveis a partir por qualquer objeto raiz devem ser preservados. Assim, além destes objetos, são considerados todos objetos que são alcançáveis ao longo da cadeia de referências a partir do *conjunto raiz*.

Qualquer objeto que é não alcançável a partir do *conjunto raiz* é considerado lixo, isto é, inútil, porque não existe uma seqüência legal de ações que programa poderia tomar para acessar tal objeto. Objetos lixo tampouco podem afetar a execução de um programa, desta forma o seu espaço pode ser recuperado com segurança.

3.2 Técnicas Básicas de Coleta de Lixo Local

A partir da abstração de duas-fases de um coletor de lixo muitas variações são possíveis. A primeira parte, de distingüir objetos vivos do lixo pode ocorrer de várias maneiras: por contagem de referências, marcação e cópia.

3.2.1 Contagem de Referências

Num sistema de contagem de referências [Col60], todo objeto tem um contador de referências, ponteiros que apontam para ele. A cada referência que é criada para o objeto, por exemplo, quando um ponteiro é copiado de lugar para outro através de uma atribuição, o contador é incrementado. Quando uma referência existente de um objeto é eliminada, o contador é decrementado. A memória ocupada pelo objeto pode ser recuperada quando o contador do objeto é cai a zero, pois isto indica que não existem ponteiros para o objeto pelos quais o *mutator* possa acessá-lo.

A principal vantagem da contagem de referências é que o *overhead* do gerenciamento de memória ocorre de forma incremental. Pois, o gerenciamento dos objeto vivos e do lixo acontece conjuntamente com a execução do programa do usuário. Em contraste com as técnicas não-incrementais como as de marcação, nas quais o processamento da aplicação é suspenso enquanto o coletor do lixo está rodando.

A incapacidade de detectar referências cíclicas é a principal desvantagem da contagem de referência. Se os ponteiros de um grupo de objetos cria um ciclo, o contador de referências não caem à zero, mesmo que não exista um caminho destes objetos até o conjunto raiz [McB63].

3.2.2 *Mark-Sweep*

Os coletores *mark-sweep* [McC60] são assim denominados por causa dos nomes de suas fases. Segue o detalhamento destas duas fases:

1. A primeira fase de distingüir os objetos vivos do lixo é dita *mark*. Nela percorre-se o grafo de ponteiros de relacionamentos partindo do conjunto raiz. Isto pode ser realizado em profundidade ou em largura. Durante este processo os objetos alcançáveis são *marcados* de alguma maneira, seja alterando bits dentro dos objetos ou guardando-os numa coleção.

2. A fase de recuperação do espaço ocupado pelo lixo é dita *sweep*. Uma vez que os objetos vivos foram diferenciados do lixo, a memória é *limpa*, isto é, ela é exaustivamente examinada para encontrar todos os objetos não-marcados para recuperar o seu espaço.

Este algoritmo tem duas vantagens [Jon96] sobre a contagem de referências: as referências cíclicas são resolvidas pelo algoritmo sem nenhum tratamento especial e não ocorre o *overhead* quando da manipulação dos ponteiros. A desvantagem é que se trata de um algoritmo *stop/start*: toda a computação é suspensa durante o processo de coleta de lixo, e estas pausas podem ser significativas.

3.2.3 Cópia

A coleta de lixo por *cópia* não "coleta" realmente o lixo, pois todos os objetos vivos são movidos para uma área do *heap*, enquanto o resto do *heap*, então fica somente com lixo. Para realizar este processo de cópia [FY69] o *heap* é divido em dois semi-espaços que alternam os papéis de espaço de *origem* e espaço de *destino*. A "coleta de lixo" nestes sistemas é implícita, e alguns pesquisadores evitam aplicar este termo a este processo preferindo o termo *scavenging*.

Segundo [Jon96], a principal vantagem da coleta por cópia é o custo de alocação extremamente baixo, pois a fragmentação do *heap* é eliminada. A compactação dos objetos ativos é realizada quando estes são copiados para o espaço *destino*. Já o custo da alocação em sistemas que não realizam a compactação é muito maior, particularmente se existirem objetos de tamanho variável. Porque é difícil encontrar um local onde alocá-los de modo mais barato. A desvantagem do algoritmo é o custo do uso de dois semi-espaços, o espaço de endereçamento exigido seria o dobro do utilizado por outros tipos de coletores.

3.3 Algoritmos de *Tracing* Incremental e Generacionais Locais

Os algoritmos de marcação e cópia também podem ser classificados como algoritmos *tracing* por percorrerem o grafo de referências entre os objetos do *heap* para descobrir quais objetos estão "vivos". Para aplicações de tempo real o aparecimento da coleta incremental foi necessária. Pois, nas implementações mais simples dos algoritmos de marcação e cópia por um tempo considerável a execução do programa do usuário é suspensa, enquanto o coletor de lixo estiver trabalhando. Para diminuir este tempo num algoritmo incremental, a coleta de lixo é feita em pequenos passos intercalada com a execução da aplicação do usuário. Para conseguir isto, é preciso a cooperação entre o *mutator* e coletor de lixo.

Já nos algoritmos de generacionais, o espaço de procura por objetos vivos é diminuído, permitindo que menos tempo seja exigido na coleta de lixo. O *heap* é dividido em gerações, assim, os objetos recém criados são alocados numa geração chamada *nursery*. Estes algoritmos se valem da hipótese de que objetos mais "novos" tendem a ser descartados mais rapidamente. Por isso, primeiro realizam-se coletas na geração mais novas e se necessário partem para a coleta nas gerações mais velhas. E os objetos que sobrevivem à coleta na geração mais nova são promovidos para as gerações mais velhas.

3.4 Coleta de Lixo Distribuída

Os coletores de lixo distribuídos são muitas vezes mais complexos do que os coletores de lixo locais [PS94]. Pois estes coletores devem coordenados para manter a consistência das mudanças de referências os espaços de endereçamento. Além de outros importantes fatores que devem ser levados em conta: como a perda de mensagem e a disponibilidade da rede.

3.5 Algoritmos de *Marcação* Distribuída

Segundo [PS94], uma abordagem padrão para algoritmos de marcação como o *mark-sweep* é combinar coletores de lixo locais e independentes com um coletor global. Os dois tipos de coletores comunicam-se entre si através dos objetos remotos, isto é, objetos que refereciam ou são referenciados por objetos que encontrem em outros espaços.

O principal problema é sincronizar a fase global de detecção de lixo com as fases locais de desalocação de memória. Durante a fase de detecção de lixo, os coletores locais enviam e recebem mensagens indicando que determinados objetos devem ser marcados. O coletor local retoma sua execução sempre que recebe uma mensagem indicando a marcação de um objeto local, e pára sua execução quando envia uma mensagem solicitando a marcação de um objeto remoto. Sendo assim, os espaços estão cooperando alternadamente para a detecção de lixo global. A fase de detecção de lixo está completa quando todos os objetos públicos alcançáveis foram marcados e não existem mais mensagens de marcação ou confirmação em trânsito. Em seguida, os coletores locais iniciam individualmente a fase de desalocação de memória a fim de se desfazer dos objetos públicos e locais que não serão mais utilizados.

Nas subseções a seguir são apresentados alguns algoritmos *mark-sweep* descritos em [Jon96].

3.5.1 Algoritmo de Hudak e Keller

O algoritmo Hudak e Keller foi um dos primeiros coletores de *mark-sweep* distribuídos [Hud82]. Desenhado para linguagens funcionais, é baseado no esquema em tempo de execução de Dijkstra. A recuperação da memória é executada em paralelo com a execução do programa, e não existe um controle central além do rendezvous entre as fases do coletor. É também capaz de encontrar e depois apagar os processos ativos que não mais necessários para o aplicativo.

O algoritmo utiliza uma árvore de marcação que pode ser alterada cooperativamente entre o coletor e *mutator*. O *mutator* distribuído pode adicionar galhos à árvore. São usadas três cores para marcação para o controle do coletor: branco, cinza e preto.

Inicialmente todos os nós são brancos. Na fase de marcação são iniciadas tarefas de marcação a partir dos objetos raízes. Quando um nó visitado, este é pintado de cinza. Se na fase marcação se descobre que o nó é uma folha, ele é pintado de preto, e depois imediatamente é feito um retorno para seu pai. Então, qualquer nó que for criado a partir desta folha preta já é criado como preto. Uma vez que a fase de marcação é finalizada,

no processo de limpeza procura-se todos os objetos brancos e estes são postos na lista de memória livre.

3.5.2 Algoritmo de Ali

Este algoritmo permite que cada processador execute o *mark-sweep* no seu próprio heap independentemente [MA84]. Ao final da coleta local, cada processador informa seus pares quais ponteiros remotos que ele ainda possui, assim os outros processadores os tratam como se fossem raízes que devem ser marcados durantes suas coletas locais. Este algoritmo reduz o custo de sincronização pois os coletores trabalham independentemente. Mas, não é um algoritmo para aplicações de tempo real pois o momento da coleta de lixo pode variar de máquina para máquina. E também não consegue detectar ciclos globais.

3.5.3 Algoritmo de Liskov e Ladin

No algoritmo Liskov e Ladin ao invés do poder de decisão do que é lixo ou não estar distribuída [Lis86], nele este serviço está logicamente centralizado, mas fisicamente replicado para conseguir tolerância a faltas e disponibilidade. Os relógios estão sincronizados e o atraso das mensagens é controlado, permitindo que este serviço seja consistente. Todas as atualizações das referências são reportadas pelos coletores locais para o serviço centralizado. Baseado nas informações coletadas, o servidor constrói um grafo global das referências, e informam os coletores locais os objetos que são acessíveis pelas raízes.

3.5.4 Coleta de lixo no Sistema Emerald

Emerald é um sistema de distribuído de objetos [Hut87]. A coleta de lixo do Emerald é hierárquica e foi implementada numa rede local de estações de trabalho. O coletor global roda em cada um dos nós do sistema continuamente, adaptando-se à situação atual do sistema. Também existem coletores locais que proporcionam maior eficiência pois os objetos tendem a ter vida curta e serem locais.

Qualquer nó pode iniciar uma coleta global. Assim, cada coletor faz a marcação independentemente. Os objetos não locais são marcados como cinza e colocadas e numa lista de objetos cinza. E, este objeto não é retirado da lista enquanto o dono da referência remota ao objeto não informar se a cor do objeto é cinza ou preta. Uma vez que não exista mais nenhum objeto nesta lista a fase de marcação é terminada. Durante esta fase de marcação global o programa (*mutator*) deve ser suspenso.

3.6 Contagem de Referências Distribuída

De acordo com [PS94], a implementação mais simples de contagem de referências distribuídas é uma simples extensão da contagem de referências locais. Num sistema fracamente acoplado, a criação de uma nova referência para um objeto requer que uma mensagem seja enviada para esse objeto a fim de que seu contador de referências seja incrementado. Da mesma forma, caso uma referência remota seja removida, uma men-

sagem deve ser enviada informando ao objeto que seu contador deve ser decrementado. Deve-se ter um cuidado especial para prevenir que um objeto seja coletado enquanto ainda existe uma referência remota para ele. Isso pode acontecer quando as mensagens são recebidas numa ordem diferente da esperada. Seu protocolo de comunicação requer que as mensagens entre pares de objetos sejam entregues na ordem em que foram enviadas. As mensagens devem ser confirmadas e um objeto só pode ser coletado se um número igual de mensagens de duplicação, remoção e confirmação foi recebido por esse objeto. Esse protocolo fornece um esquema de contagem de referências distribuídas ao custo de três mensagens por referência entre espaços.

De acordo com [Jon96], a contagem de referências tem uma série de vantagens sobre a coleta por *mark and sweep* distribuída, o que torna sua aplicação mais atrativa em sistemas distribuídos. A coleta é executada em pequenos passos durante a execução do programa, não é preciso uma estrutura de dados globais, e não ocorre a perda de desempenho com aumento do número de objetos.

3.6.1 Contagem Indireta de Referências

A contagem indireta de referências mantém uma árvore de difusão que representa a história das cópias dos ponteiros [Piq90]. Ela utiliza dois campos extras para cada ponteiro: uma referência para o ponteiro pai na árvore de difusão e um contador para seus filhos. O ponteiro para o pai serve somente para a coleção distribuída, e pode referenciar um objeto ou um outro ponteiro remoto. O conjunto dos ponteiros remotos que referenciam um objeto forma um grafo distribuído que pode ser percorrido utilizando ponteiros indiretos. A criação de novas células ou a cópia de novos ponteiros é executada localmente sem a necessidade de qualquer comunicação. A exclusão de um ponteiro pode gerar mais de uma mensagem por referência.

3.6.2 Garbage collecting the World

Lang [Lan92] descreve um coletor distribuído que pode recuperar lixo cíclico. O coletor é híbrido, utilizando a contagem de referências para objetos globais e um coletor por *tracing* para objetos locais. Os nós da rede são organizados em grupos que cooperam na coleta do lixo. Cada grupo dá um identificador único para cada ciclo de coleta e vários grupos sobrepostos de coleta podem estar ativos. Se um nó falha na cooperação, o grupo ao qual ele pertence é reorganizado para excluir o nó e para coleta continuar.

A coleta distribuída começa com uma negociação em grupo. Todos objetos nós pertencentes ao grupo são identificados e marcados como *hard* ou *soft*. Um objeto é *hard* se ele é referenciado por alguém fora do grupo, ou se ele é acessível por uma raiz. Outros objetos que são acessíveis apenas por outros nós do grupo são marcados como *soft*. O contador de referências inicia a marcação dos objetos de entrada de um grupo, esta marcação então é propagada para os objetos de saída pelos coletores locais. As marcas dos objetos de saída são propagadas para os objetos de entrada que referenciam um grupo de coleta. Este processo é repetido até que as marcas dos objetos de entrada e saída de um grupo não mais se alterem. Todos os objetos acessíveis ou por uma raiz ou por um nó

fora do grupo são marcados como hard. Objetos de entrada marcados como *soft* devem ser partes isoladas de ciclos de um grupo e podem ser recuperadas.

3.6.3 Protocolo SSP (Stub-Scion Pair) Chains

SSP Chains descrita em [Sha93] é uma técnica desenvolvida para referenciar objetos num sistema distribuído. Assim, para um *software* cliente, qualquer referência para qualquer objeto aparenta ser um ponteiro local; quando o alvo é um objeto remoto, o *SSP Chains* adiciona uma quantidade indeterminada de níveis de indireção. Copiar uma referência por um sistema distribuído estende em um (1) a cadeia *SSP*; também, migrar o objeto-alvo estende a cadeia em um (1). O sistema de invocação é otimizado para criar um atalho caso existam vários níveis de indireção, ao retornar a localização atual do objeto na chamada remota de um método. O sistema de coleta distribuído de lixo suportado pelo *SSP Chains* é um variante da contagem de referências, contudo, só realiza a coleta de lixo acíclico.

Características

As principais características do *SSP Chains* são:

- Não existe controle, estruturas de dados ou sincronização globais. As informações somente são trocadas entre pares de nós. Ainda são tratadas: a perda, atraso, duplicação ou a falta de ordem mensagens trocadas entre os nós.
- Uma referência remota é representada por uma cadeia de tamanho arbitrário de pares de *stub-scion*. Cada *stub* de saída possue as localizações de dois *scions* de chegada. Uma localizador *strong* indica o próximo *scion*; já a localizador *weak* indica algum *scion* da mesma cadeia da *strong*, porém mais perto do objeto alvo. O uso do localizador *weak* é de melhorar desempenho ao evitar que a comunicação passe por toda cadeia de referências.
- O algoritmo de coleta de lixo distribuído utilizado é uma variante conservadora da contagem de referências. Já os coletores locais podem ser baseados em qualquer algoritmo de *tracing* padrão.

A figura 3.1 ilustra o modelo de objetos do SSP Chains. Nela vemos uma cadeia de referências do objeto **x** em Mercúrio para **y** em Vênus, representada pelo par stub $\mathbf{y}_A$ e scion **b**. Também, notamos que os localizadores *weak* e *strong* do stub $\mathbf{y}_A$ apontam para o mesmo scion **b** pois o tamanho da cadeia é de somente um par *stub-scion*.

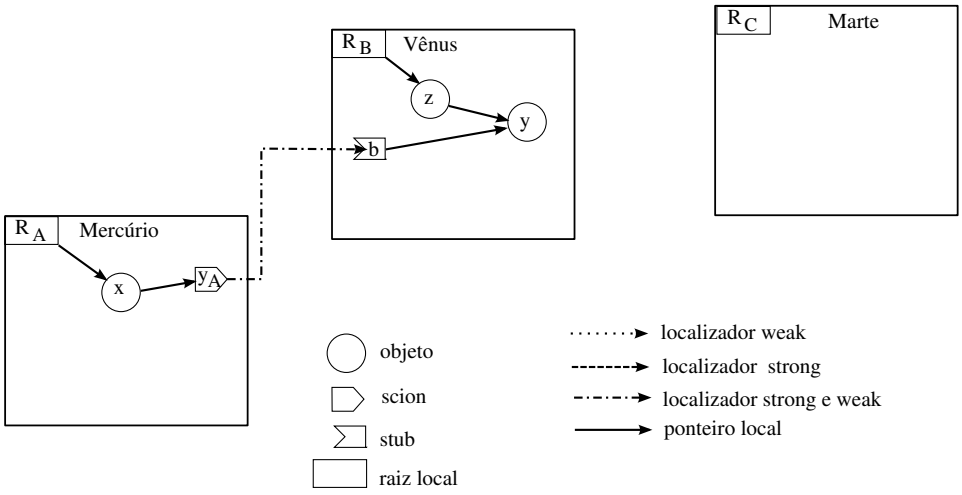

Figura 3.1: Modelo de Objetos e Referenciamento de Objetos - Snapshot 1

A figura 3.2 representa o estado das referências entre os objetos no SSP Chains depois da migração do objeto **y** de Vênus para Marte. Nota-se que a cadeia de referências do objeto **x** para **y** foi acrescida de mais um par *stub-scion* ($\mathbf{y}_B$-**c**). Já os localizadores e *strong* e *weak* apontam para *scions* diferentes. O localizador *weak* é atualizado toda vez que algum método remoto de um objeto (neste caso **y**) é invocado, pois seu endereço "real" é retornado na invocação dos métodos remotos.

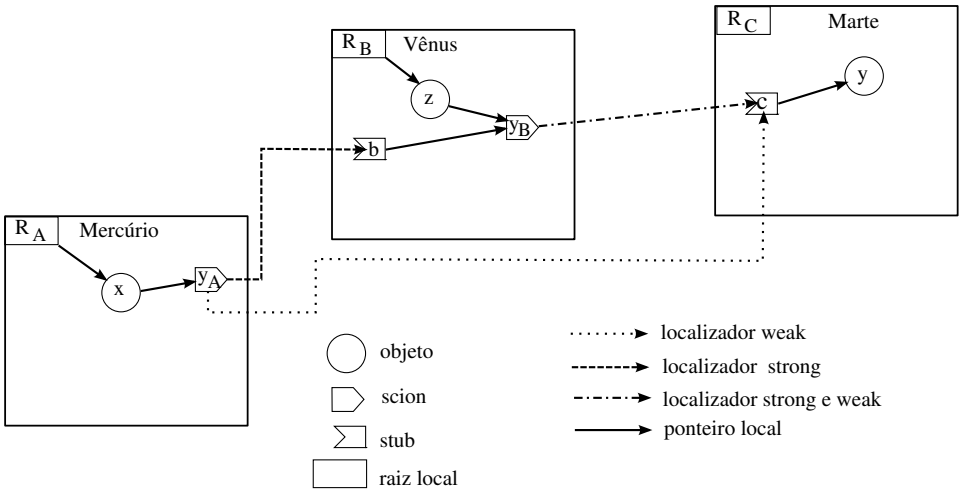

Figura 3.2: Modelo de Objetos e Referenciamento de Objetos - Snapshot 2

Espaços

No *SSP Chains* o sistema distribuído está particionado em espaços disjuntos identificados unicamente que interagem entre si através de troca de mensagens, usando canais

de comunicação não confiáveis.

Cada espaço tem um relógio local, sendo que o tempo é incrementado a cada leitura do relógio. Os relógios não precisam estar sincronizados uns com os outros. Estes tempos são utilizados para estampar as mensagens de envio de referência, os *stubs* e os *scions* mantidos respectivamente nas tabelas de saída e de entrada. Ambas tabelas são conservadoras, isto é, se dois diferentes espaços referenciam o mesmo objeto, cada espaço terá um item associado a uma entrada nestas tabelas, esta característica é ilustrada pela figura 3.3 pois notamos que existem dois diferentes *scions c* e *c'*, em Marte para o mesmo objeto **y**, para os espaços Vênus e Terra. Permitindo, desta forma que as mensagens recebidas sejam descartadas, ou seja, mensagens que tiverem um valor menor que o maior tempo associado ao espaço que enviou a mensagem.

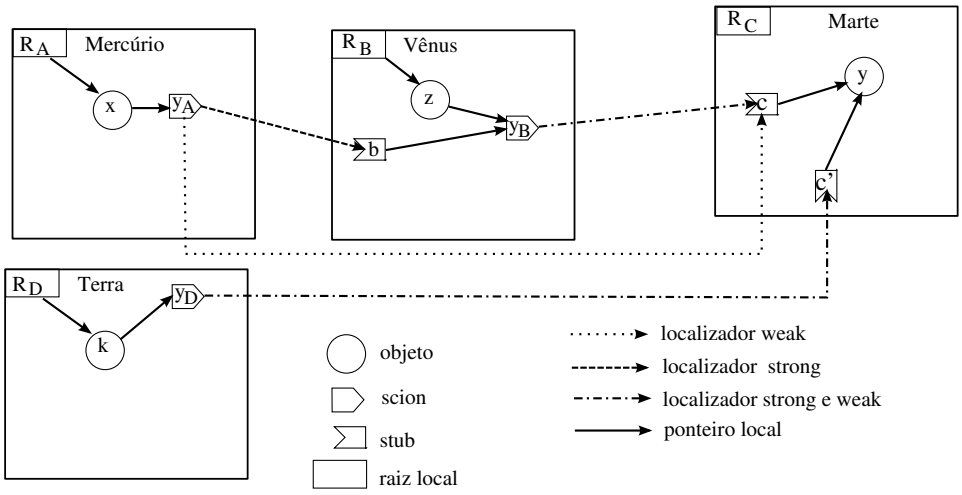

Figura 3.3: Referenciamento de Objetos Remotos

Gerenciamento das Referências

Como uma cadeia de referências pode crescer indefinidamente, ocorrem problemas de desempenho na comunicação, ao precisar de inúmeros *hops* para a execução de uma mensagem. Para resolver este problema, existe um localizador no *stub*: o *weak scion*. O *weak scion* é somente utilizado na comunicação, sempre que uma referência é processada se verifica, se a referência corresponde a um objeto real ou a um *stub*. Caso a referência seja para um *stub* então o *stub* do *weak scion* é propagado para o espaço que o referenciará. O efeito disto é que o *weak scion* apontará para diretamente para o objeto, a menos que ele tenha migrado para um outro espaço.

Existe também o processo de *shortcutting* para redirecionar uma referência indireta para apontar diretamente ao objeto alvo. Para evitar o uso de uma mensagem específica para este fim, este processo somente é disparado quando são realizadas as chamadas remotas de mensagens. Por exemplo, quando é enviada uma mensagem para um objeto o seu *scion* verifica se o espaço para qual a mensagem foi originalmente enviada está diferente do espaço atual do objeto. Ocorrendo esta situação, é criado num novo *scion* cujo endereço é retornado na resposta da mensagem. Desta forma quem enviou a mensagem

criará um novo *stub* para este *scion*. Ao não utilizar mensagens específicas evita-se o custo extra na comunicação, e se um objeto nunca for invocado este processo de *shortcutting* não ocorrerá.

A figura 3.4 mostra o momento em que foi realizada uma chamada de uma mensagem remota de Mercúrio para o objeto **y** em Marte, e na resposta desta mensagem vem o endereço real de **y**, neste caso o endereço é Marte.

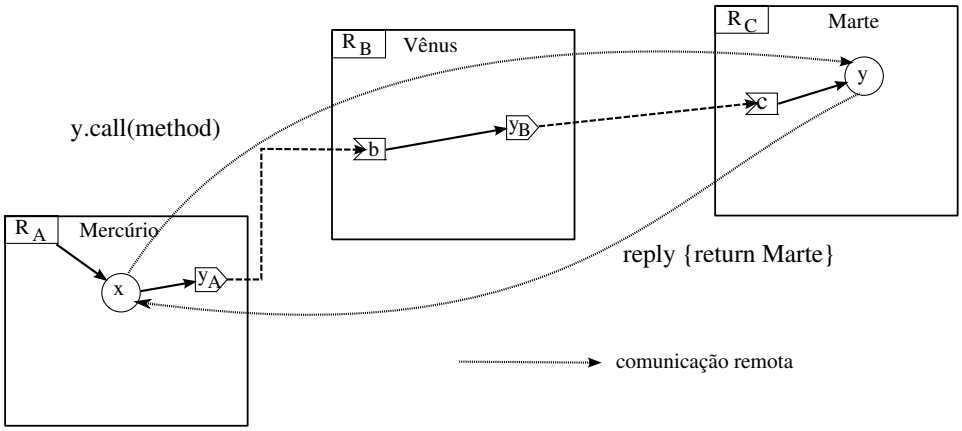

Figura 3.4: Shortcutting Snapshot 1

A figura 3.5 mostra o estado das referências para o objeto **y** depois de realizado o processo de *shortcutting*. Nela vemos que existe um novo par *stub-scion* ($\mathbf{y}_C$**-c'**) que faz a referência entre **x** e **y**. Já a antiga cadeia de referências não é mais utilizada por Mercúrio.

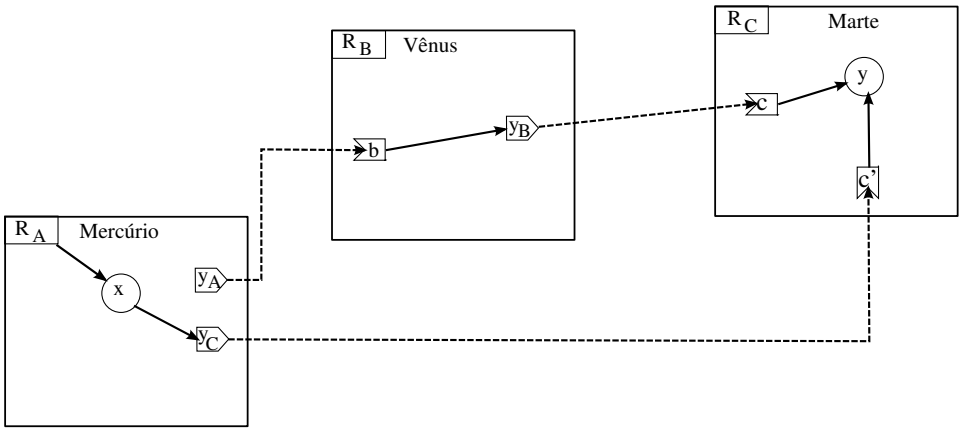

Figura 3.5: Shortcutting Snapshot 2

Coleta de Lixo Distribuído

O processo *shortcutting* deixa lixo, isto é, referências que não são mais utilizadas. A detecção destas referências é feita pelo protocolo *cleanup*. Este protocolo especifica que

sejam trocadas de tempos em tempos, mensagens *live* entre os espaços. Uma mensagem *live* do espaço A para B contém a lista de todos os *scions* no espaço B que os *stubs* em B referenciam em A. Somando este conjunto de *scions* que estão, ou foram referenciados por A, ao outros conjuntos enviados por outros espaços, B pode computar quais *scions* não estão sendo mais referenciados e que podem ser removidos. No entanto, originalmente, o coletor de lixo é incapaz de detectar ciclos de lixo entre espaços. Estudos para detecção de ciclos de lixo no *SSP Chains* foram feitos por Kordale e Ahamad.

No instante representado pela figura 3.5 a mensagem *live* enviada por Mercúrio para Vênus teria o *scion* **b** pois o *stub* $\mathbf{y}_B$ ainda aponta ele. Já na figura 3.6 temos o momento após em que o coletor de lixo local rodou em Mercúrio e notamos que o stub $\mathbf{y}_B$ foi coletado, pois ele não era alcançável por nenhum objeto raiz. A mensagem *live* de Mercúrio para Vênus irá com uma lista vazia e o espaço Vênus ao receber esta mensagem poderá retirar o *scion* **b** da sua lista local de objetos raízes.

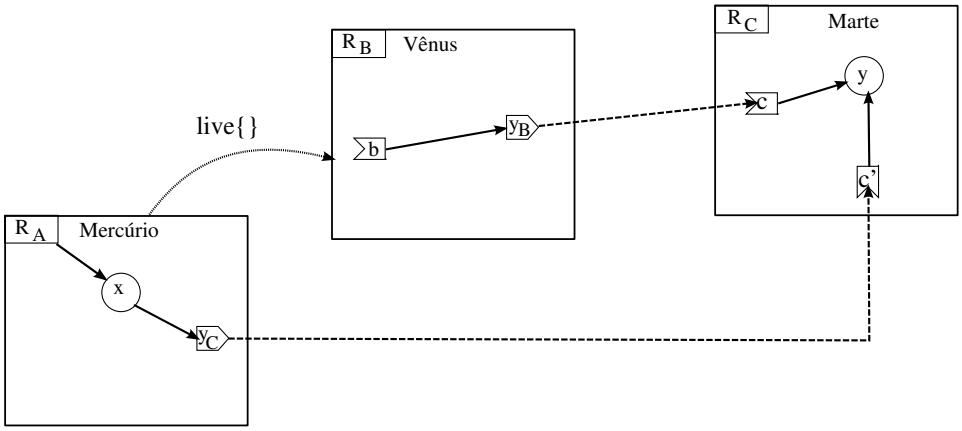

Figura 3.6: Mensagens *Live*

Simplificações

Entre as possíveis simplificações no *SSP Chains*, ao assumir o uso de protocolo confiável como *TCP* ou *ISO-TP* as mensagens não seriam perdidas, duplicadas e estariam sempre em ordem *FIFO*. Neste ambiente o mecanismo de *threshold* de mensagens poderia ser eliminado. A troca periódica de mensagens *live* também não seria necessária. Ao invés, sempre que um *stub* for coletado pelo coletor local, uma única mensagem *delete* seria enviada ao seu correspondente *scion*.

3.6.4 Algoritmo de Maheshwari e Liskov

Este coletor distribuído apresentado em [ML95] é baseado na contagem de referências capaz também de coletar o lixo cíclico distribuído. A heurística utilizada para determinar se os objetos estão num ciclo inter-nó de lixo é baseada na propagação das distância dos objetos em relação aos objetos raiz locais. E, a coleta do lixo propriamente

dita é realizada por um nó para qual os objetos são migrados, cujo coletor local é baseado na técnica de *tracing*. O algoritmo foi projetado para ser utilizado no banco de dados de objetos Thor, mas pode ser aplicado em outros sistemas que também armazenem objetos em múltiplos nós. Nestes sistemas, objetos podem referenciar outros objetos que estejam num outro nó.

Modelo de Referenciamento de Objetos e Coleta de Lixo

Dentro deste ambiente, define-se um objeto alcançável por outro quando existem referências para ele. É dito que um objeto está vivo se o mesmo é alcançável por pelo menos um objeto raiz **persistente**. Objetos que não são alcançáveis por objetos raízes são considerados lixo.

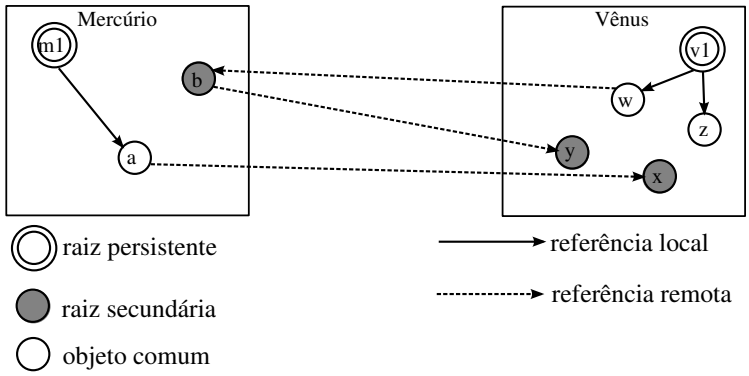

Figura 3.7: Modelo de Referenciamento

Na contagem de referências distribuída, cada nó faz a coleta local baseada no *tracing* das referências a partir do conjunto de objetos raízes, no qual incluem-se os objetos locais persistidos bem como os objetos locais referenciados remotamente por outros nós. Estas raízes *secundárias* são mantidas numa lista de referências, por exemplo, no caso representado pela figura 3.7, esta lista do nó Mercúrio seria como em 3.8:

```
Inlist[Vênus] = {b};
```

Figura 3.8: Inlist do nó Mercúrio para Vênus

E Vênus teria uma *inlist* para Mercúrio como esta:

```
Inlist[Mercúrio] = {x, y};
```

Figura 3.9: Inlist do nó Vênus para Mercúrio

Desta forma haverá tantas inlists quantos forem os nós que referenciem objetos locais. E, cada vez que um objeto local é referenciado remotamente ele é inserido na *inlist* apropriada.

Um objeto sai da *inlist* quando não existir mais nenhum nó o referenciando. Para se determinar quando um objeto local deixou de ser referenciado o nó responsável pelo objeto recebe *outlists* de outros nós. Uma *outlist* é lista de objetos que um nó referencia remotamente. *As outlists* são montadas e enviadas ao final do processo de coleta de lixo local. Utilizando novamente o caso da figura 3.7, a *outlist* de Mercúrio seria esta:

```
Outlist[Vênus] = {x, y};
```

Figura 3.10: Outlist do nó Mercúrio para Vênus

A outlist de Vênus para Mercúrio seria esta:

```
Outlist[Mercúrio] = {b};
```

Figura 3.11: Outlist do nó Vênus para Mercúrio

Heurística da Distância

Um ciclo de lixo multi-nó acontece quando pelo menos dois objetos lixo estão em diferentes nós e são alcançáveis entre si. Na figura 3.12 os objetos **a**, **y**, **w** estão num ciclo multi-nó.

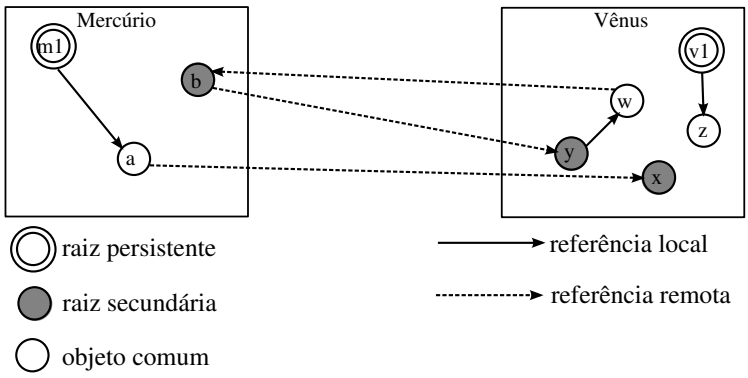

Figura 3.12: Ciclo multi-nó

A detecção de um ciclo de lixo é baseada nas *distâncias* dos objetos. A distância de um objeto é a menor quantidade de referências inter-nó entre todos caminhos a partir de um objeto raiz persistente qualquer. A distância é associada a cada referência no conjunto de raízes. A distância de um objeto raiz persistente é implicitamente igual a zero e cada referência na *inlist* tem um atributo para a distância. Quando uma nova entrada é adicionada na *inlist* sua distância é inicializada com um (1).

Então, o coletor local propagará as distâncias das raízes para as *outlists*, atualizando a distância da entrada da *outlist* adicionando um (1) à menor distância de qualquer raiz alcançável. O coletor atualiza estas distâncias pelo *tracing* completo de referências das raízes. Isto é, faz-se o rastreamento de todas as referências de uma raiz antes de partir para outra. As raízes são percorridas em ordem crescente de distância. Mas, antes disto, o nó utiliza a *outlist* recebida por um outro nó para substituir sua *inlist* para aquele nó. A propagação é feita através dos coletores locais, e o envio das mensagens de atualização de *outlists* resulta no aumento sem limites das distâncias dos lixos cíclicos. Isto pode ser observado quando os nós que possuem um ciclo de lixo realizam a coleta local e trocam mensagens de *outlists* periodicamente, pois as distâncias dos objetos lixo aumentarão a razão média de um (1) a cada coleta.

Como vemos na figura 3.13 os objetos **a** e **z** tem distâncias iguais a zero pois são alcançáveis por raízes locais. A distância do objeto **x** é *1* pois a distância propagada foi de a *distância(a) + 1*. Supondo, que o objeto **b** tivesse neste momento a distância *1*, ele propaga a sua *distância + 1* para **y** que por sua vez a repassa para **w**, por isso *2* é o valor das distâncias de **y** e **w**.

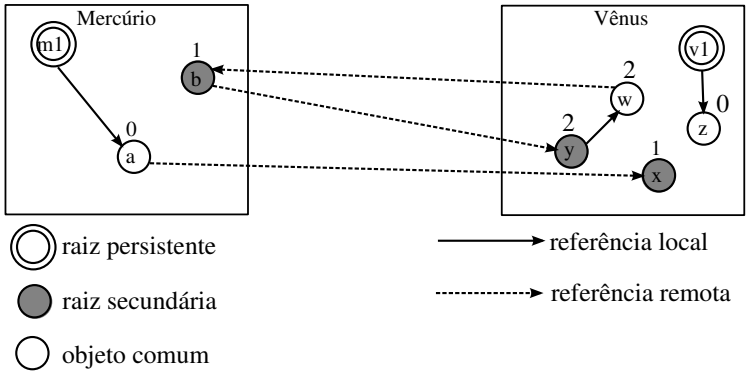

Figura 3.13: Distâncias de objetos - Snapshot 1

Se o nó Vênus rodasse o coletor de lixo local teríamos a seguinte situação, as distâncias dos objetos ainda não se modificariam, mas a *outlist* enviada para Mercúrio teria o objeto **b** associada a distância *3*, vemos isto em 3.14:

```
Outlist[Mercúrio] = {b.distância = 3};
```

Figura 3.14: Outlist do nó Vênus com distâncias para Mercúrio - Snapshot 2

Então, Mercúrio ao rodar o seu coletor local utilizaria a *outlist* recebida de Vênus e atualizaria a distância de **b** para *3*, pois é a menor distância (e única) encontrada. Depois, atualizaria a *outlist* de Vênus com a distância de **x** igual a *1* e **y** com *4*. Como é mostrado na figura 3.15:

```
Outlist[Vênus] = {x.distância = 1, y.distância = 4};
```

Figura 3.15: Outlist do nó Mercúrio com distâncias para Vênus - Snapshot 3

Como as distâncias dos objetos vivos ou que não estejam em num ciclo de lixo não crescerão indefinidamente, é possível determinar uma distância limite (*threshold*), assim todos os objetos com uma distância maior ou igual a ela, podem pertencer ao ciclo de lixo. Por isso somente estes objetos são migrados para um nó para se determinar se realmente estão num ciclo. A escolha do *threshold* depende da distância esperada para os objetos vivos. Como a distância estimada dos objetos vivos pode divergir temporariamente das distâncias reais, este limite escolhido deve ser o menor múltiplo da maior distância esperada. Felizmente, a penalidade pela má escolha de um limite não é grave: se for muito

baixo, alguns objetos vivos podem ser migrados, mas sem comprometer a segurança, pois somente os objetos-lixo serão removidos; caso o limite seja alto, todos os ciclos de lixo demorarão para serem detectados.

A propagação da distância precisa somente da cooperação dos nós onde os objetos lixo estão. Se algum nó cair ou houver particionamento da rede ou ocorrer lentidão na coleta de lixo local, somente os nós onde ocorre a coleta dos objetos-lixo serão prejudicados.

O *overhead* deste esquema é pequeno. Os campos de distâncias somente são associados às entradas da *inlist* e da *outlist*, e não a todos os objetos. Nenhuma mensagem extra é necessária além daquelas utilizadas para a coleta de lixo não-cíclica distribuída.

Migração

Consolidar um ciclo distribuído através da migração apresenta alguns problemas práticos. Primeiro, migrar um objeto referenciado remotamente que referencie objetos locais provavelmente criará mais objetos remotamente referenciados do que havia anteriormente. Esse esquema faz isto através de uma migração em lote: uma vez que a distância esteja acima do limite, todos os objetos que foram rastreados a partir de um mesmo objeto raiz são migrados conjuntamente.

Segundo, escolher o local para onde os objetos migrarão. Para evitar várias migrações são propagadas as estimativas do nó máximo (aquele com o maior identificador). As entradas das *inlists* e *outlists* com as distâncias maiores que o limite tem um campo para o nó destino da migração. Quando o coletor cria uma entrada na *outlist*, seu campo destino é preenchido com o maior identificador entre os identificadores do nó local e dos campos destinos (se existir) das entradas da *inlist*

3.7 Conclusão

Neste primeiro estudo escolhemos os seguintes candidatos à implementação: o *SSP Chains* e o *Maheshwari e Liskov*. A escolha inicial de ambos se deveu por serem algoritmos de contagem distribuída, pois de acordo com [Jon96] estes tipos algoritmos são os mais indicados para os sistemas distribuídos, porque a coleta é executada em pequenos passos durante a execução do programa, não é preciso uma estrutura de dados globais e o desempenho não se degrada com aumento do número de objetos.

A escolha do algoritmo de coleta de lixo do protocolo *SSP Chains* se baseou na semelhança no uso de indireções na migração de objetos (seção 3.6.3) como na *Virtuosi*. Quanto a escolha do segundo algoritmo se deveu por enviar mensagens com os objetos que estão sendo referenciados, ao contrário da simplificação proposta (seção 3.6.3) pelo *SSP Chains* que envia uma mensagem com os objetos que deixaram de ser referenciados. O que poderia variar no tamanho das mensagens quando a população de objetos varia, ou seja, quando ocorre aumento, diminuição ou estabilização da população de objetos vivos.

Capítulo 4

Implementação da Coleta de Lixo Distribuída na *Virtuosi*

Uma das principais características da arquitetura Virtuosi é a utilização da tabelas de referências, a utilização dessas tabelas proporciona a facilidade de controlar, manter e distribuir os objetos entre as máquinas virtuais. As tabelas de referências constituem um importante fator para que a manipulação dos objetos seja transparente. Por isso a escolha dos algoritmos *SSP Chains* e *Maheshwari e Liskov* se deu por serem simples do ponto de vista da implementação e serem sistemas utilizadores de listas de referências de objetos que são facilmente adaptáveis a este esquema de referenciamento de objetos da Virtuosi.

Além dessas características, apesar de não estar dentro do escopo deste trabalho otimizar a execução dos métodos remotos o algoritmo *SSP Chains* foi escolhido por apresentar soluções para otimizar a comunicação entre objetos remotos utilizando-se dos localizadores *weak* e do mecanismo de *shortcutting*. Pois a Virtuosi apresenta o mesmo tipo de problema ao produzir uma cadeia de indireções até um objeto alvo como o do protocolo *SSP Chains*.

4.1 Escopo

O escopo deste trabalho é a coleta de lixo na área de objetos da Virtuosi (seção 2.3.2). Os objetos são as instâncias das classes de uma aplicação. Portanto, não será contemplada a coleta de lixo de classes ou atividades que não estejam mais sendo referenciadas.

Apesar de ser um importante aspecto, a tolerância a faltas na coleta de lixo distribuída não será tratada neste trabalho. Mesmo sendo intríseco ao trabalho implementar o coletor de lixo local, este também não será discutido neste trabalho.

4.2 Premissas

Foram adotadas as seguintes premissas a fim de facilitar a implementação do coletor de lixo local e distribuído:

- todos objetos tem tamanhos iguais;
- implementou-se um coletor de lixo local *mark-sweep*, do tipo *stop and collect*. Ou

seja, quando o coletor de lixo é acionado o *mutator* é interrompido, só retornando ao final da coleta de lixo;

- além de reciclar os objetos alocados também se recicla as entradas das tabelas de objetos para sua posterior reutilização;
- o tamanho do heap e o tamanho da tabela de objetos podem ser configurados;
- o algoritmo de coleta de lixo distribuído a ser utilizado deve ser definido na inicialização da máquina virtual. Não pode ser trocado durante a execução de uma aplicação;
- os algoritmos de coleta de lixo distribuída devem ser os mesmos em todas as VMs para que possam rodar corretamente.

4.3 Objetivo

O principal objetivo do modelo a ser apresentado foi de minimizar alterações no modo de referenciamento dos objetos da *Virtuosi*. E como um modelo ele deveria ser suficientemente adaptável de maneira a suportar a implementação de algoritmos diferentes de coleta de lixo distribuída na mesma máquina virtual. Assim, permitir a escolha de algoritmos de coleta de lixo distribuído na inicialização de uma *MVV*, no caso deste trabalho seriam os algoritmos: *SSP Chains* ou *Maheshwari e Liskov*.

4.4 Ambiente de Desenvolvimento

Foi utilizada a linguagem Java para a implementação dos coletores de lixo. Justifica-se o seu uso, pois este ambiente de desenvolvimento já foi utilizado em outras funcionalidades da Virtuosi, como o sistema de RPC e de mobilidade de objetos.

4.5 Estratégia de Implementação

Como estratégia de implementação foram definidas algumas interfaces Java como especificação na implementação do gerenciamento de memória na Virtuosi além daquelas feitas através de modelos UML. Optou-se por essa abordagem por facilitar a substituição das implementações e flexibilizar a interação dos elementos envolvidos. Pois na interface define-se apenas as assinaturas de métodos e não se especifica a implementação. Além de ser uma maneira mais próxima da implementação, que ao mesmo tempo se define o comportamento dos componentes e proprociona sua utilização direta na implementação.

Neste esquema foi definido o modelo de objetos da área de objetos (seção 2.3.2) da Virtuosi, pois se trata de uma parte já bem estudada e definida em estudos anteriores. E como modo de se manter fiel a estudos já realizados. Já na parte da gerência de memória não optou-se por este modelo de definições através de interfaces por ainda se tratar de um primeiro estudo na Virtuosi sobre tal área. Mesmo assim se buscou uma

implementação mais modularizada a fim de proporcionar em futuros estudos a extração de uma especificação dos principais elementos envolvidos na coleta de lixo distribuída na Virutosi.

4.6 Visão Geral das Alterações na *Virtuosi*

Como a principal característica da Virtuosi é a mobilidade de objetos conseguida a através do referenciamento entre os objetos é feita através da tabela de objetos, evitou-se a troca ou alterações que descaracterizassem este modelo.

Para ambas implementações dos algoritmos citados, uma alteração no metamodelo original da arquitetura da *Virtuosi* foi realizada, substituir os tipos de entradas na Tabela de Objetos: Entrada de Referência Objeto Local e Entrada de Referência Objeto Remoto (seção 2.3.2).

Aqui, nesta proposta só haveria somente um tipo de entrada: Entrada de Referência para Objeto. Esta entrada apontaria para um *Objeto Remoto* ou *Objeto Local*, para realizar o diferenciamento entre um objeto local ou remoto, o que antes era feito pelos tipos de entradas na tabela de objeto. Deste maneira seriam acrescentados ao metamodelo dois tipos de objetos

- Objeto Local - representando os objetos locais.
- Objeto Remoto - representando objetos remotos, ou seja, referências remotas para objetos locais ou referências locais para objetos remotos.

Assim uma invocação remota de um método se dá quando a entrada da tabela aponta para um Objeto Remoto, ao invés de se verificar o tipo de entrada na tabela de objetos (seção 2.5.2). Outra mudança ocorre na migração de objetos, o preenchimento das entradas remotas com os endereços de entradas de uma tabela de objetos remota, é substituído pela criação de objetos remotos. Pois, um objeto remoto guarda o endereço remoto do objeto que ele representa. Esta abordagem de objetos remotos é semelhante ao utilizada no protocolo *SSP Chains* e aos outros sistemas distribuídos como CORBA [cor] e RMI Java [jav]. A motivação desta alteração foi simplificar a desalocação dos objetos locais e remotos, por conseqüência a liberação das entradas na tabela de objetos.

Assim, utilizando a nomenclatura do protocolo *SSP Chains* chamaremos os objetos remotos que referenciam remotamente objetos em outras MVVs de *stubs* e os objetos remotos que representam objetos que estão referenciados objetos locais à MVV de *scions*.

Para os dois algoritmos implementados houve a inclusão de uma *inlist* para guardar os objetos remotamente referenciados, os *scions*. Estes objetos são considerados como raízes para a coleta de lixo, assim, os objetos alcançáveis por meio deles não são considerados lixo. E somente, para o algoritmo de *Maheshwari e Liskov* a inclusão de uma *outlist*, para manter os objetos que referenciam remotamente outros objetos em outras MVVs.

4.7 Implementação da Área de Objetos

Dentro da área de objetos na Virtuosi foram implementados os seguintes elementos:

- Objetos: os objetos são as instâncias das classes de aplicação na Virtuosi.
- Referências: os relacionamentos entre os objetos numa MVV seja por meio de atributos, parâmetros ou retorno de um método sempre é feito através de referências. Nunca o objeto é manipulado diretamente.
- Tabela de Objetos: relembrando, é por onde são realizados todos os referenciamentos entre os objetos.
- Entrada da Tabela: elemento por onde os objetos são referenciados. Seria o endereço do objeto.

4.7.1 Interfaces da Área de Objetos

A seguir, a definição das interfaces da área de objetos. Os métodos descritos estão na linguagem Java, contudo nem todos os métodos implementados serão listados aqui neste trabalho, somente os relevantes para o entendimento da implementação dos coletores de lixo.

VObject

É a interface que representa um objeto da área de objetos da Virtuosi, ou seja, uma instância de classe de aplicação.

Lista de métodos para as referências entre os objetos, seja por atributos, parâmetros ou retornos de métodos:

- `void addReference(String id)`: adiciona uma entrada para uma referência. O parâmetro é um identificador único para a referência;
- `void bindReference(String id, VReference ref)`: atribui uma referência, a VReference recebida como parâmetro a entrada definida pelo parâmetro `id`;
- `void removeReference(String id)`: remove uma referência dado pelo identificador recebido como parâmetro;
- `Collection<VReference> getReferences()`: retorna todas as referências pertencentes ao objeto.

Lista de métodos referentes à coleta de lixo:

- `void setDistance(int dist)` - atribui uma distância ao objeto, utilizado pelo algoritmo Maheshwari e Liskov;
- `void setMark(boolean mark)` - método utilizado para fazer a marcação de um objeto durante a fase de marcação do coletor de lixo local.

- `void setRecycle(boolean recycle)` - marca o objeto como reciclável, realizado durante a fase de *sweep* do coletor de lixo local. Indica que o objeto pode ser reutilizado.

VEntryTable

Representa a entrada da tabela de objetos. A interface é semelhante da **VObject** pois na implementação dos métodos desta interface é repassar, seguindo o padrão de projetos *Delegation*, as chamadas para a instância de **VObject** associada à entrada da tabela de objetos.

Lista de métodos para as referências entre os objetos, seja por atributos, parâmetros ou retornos de métodos:

- `void addReference(String id)`: adiciona uma entrada para uma referência. O parâmetro é um identificador único para a referência;
- `void bindReference(String id, VReference ref)`: atribui uma referência, a VReference recebida como parâmetro a entrada definida pelo parâmetro `id`;
- `void removeReference(String id)`: remove uma referência dado pelo identificador recebido como parâmetro;
- `Collection<VReference> getReferences()`: retorna todas as referências pertencentes ao objeto.

Lista de métodos referentes à coleta de lixo:

- `void setDistance(int dist)` - atribui uma distância ao objeto, utilizado pelo algoritmo Maheshwari e Liskov;
- `void setMark(boolean mark)` - método utilizado para fazer a marcação de um objeto durante a fase de marcação do coletor de lixo local.
- `void setRecycle(boolean recycle)` - marca o objeto como reciclável, realizado durante a fase de *sweep* do coletor de lixo local. Indica que o objeto pode ser reutilizado.

VReference

Interface que representa a referência de um objeto. Associada a esta classe estará uma **VEntryTable**. A diferença é que poderão existir várias instâncias de **VReference** para uma mesma **VEntryTable**. Já que uma referência pode ser copiada implicitamente. Isto acontece no retorno de um método, ou quando são passadas como parâmetros de métodos elas são copiadas. Da mesma forma que a **VEntryTable** a referência tem interface semelhante à da **VObject**, pois ela meramente repassa também as chamadas para **VEntryTable** associada.

Lista de métodos para as referências entre os objetos, seja por atributos, parâmetros ou retornos de métodos:

- `void addReference(String id)`: adiciona uma entrada para uma referência. O parâmetro é um identificador único para a referência;
- `void bindReference(String id, VReference ref)`: atribui uma referência, a VReference recebida como parâmetro a entrada definida pelo parâmetro `id`;
- `void removeReference(String id)`: remove uma referência dado pelo identificador recebido como parâmetro;
- `Collection<VReference> getReferences()`: retorna todas as referências pertencentes ao objeto.

Lista de métodos referentes à coleta de lixo:

- `void setDistance(int dist)` - atribui uma distância ao objeto, utilizado pelo algoritmo Maheshwari e Liskov;
- `void setMark(boolean mark)` - método utilizado para fazer a marcação de um objeto durante a fase de marcação do coletor de lixo local.
- `void setRecycle(boolean recycle)` - marca o objeto como reciclável, realizado durante a fase de *sweep* do coletor de lixo local. Indica que o objeto pode ser reutilizado.

VObjectTable

Interface para a tabela de objetos.

- `VReference addObject(VObject object)`: adiciona o objeto recebido como parâmetro à tabela de objetos e retorna uma referência para o objeto inserido.
- `void removeObject(VObject object)`: remove um objeto passado como parâmetro da tabela de objetos.

4.7.2 Relacionamento entre as Interfaces na Área de Objetos

Uma referência de um objeto é representada por uma instância da interface VReference. Similar a um ponteiro que aponta para um endereço de memória, a VReference aponta para uma entrada da tabela de objetos. E a VEntryTable representa uma entrada desta tabela aponta para uma instância de VObject.

Reforçando, o que foi descrito nas seções anteriores a interface VObject representa uma instância de uma classe de aplicação e os relacionamentos entre outras instâncias é feita pela interface VReference, exemplificando, dada a classe **Casa** que possua um atributo de nome **proprietario** da classe **Pessoa**, como na figura 4.1: .

```
class Casa{
    Pessoa proprietario;
}
```

Figura 4.1: Trecho de código em Aram com a classe Casa

Dentro desta especificação apresentada, para materializar uma instância da classe **Pessoa** teríamos os seguintes passos:

1. Instanciar um VObject para a classe Casa no heap da máquina virtual; .

```
VMObject obj = _heapLocalObjects.allocate();
if(obj == null){
   throw new VMMemoryManagerException("The heap is exhausted");
}
```

Figura 4.2: Trecho de código em Java para instanciar uma VObject

2. Adiciona-se a instância da **VObject** criada à tabela de objetos, na **VTableObject** através do método `addObject`, para atribuir um índice ao objeto criado. Ao fazer isto é criada também uma **VReference**;

```
VReference ref = _table.addObject(obj);
```

Figura 4.3: Trecho de código em Java adicionando o objeto criado à tabela de objetos

3. Com a instância de **VReference** retornada pelo método `addObject`, cria-se uma entrada para o atributo **proprietario** por meio do método `addReference`, que recebe como parâmetro um identificador.

```
ref.addReference("proprietario");
```

Figura 4.4: Trecho de código em Java adicionando uma entrada para o atributo proprietario

Para relacionar as instâncias entre de objetos teríamos a chamada do método `bindReference`. Utilizando o exemplo da Casa descrito anteriormente, para relacionar uma Pessoa à instância de Casa criada o seguinte código seria executado, 4.5.

```
ref.bindReference("proprietario", refPessoa);
```

Figura 4.5: Trecho de código em Java para atribuir uma referência de um objeto Pessoa o atributo proprietario

Inicialmente, uma referência é criada ao se instanciar um objeto e adicioná-lo à tabela de objetos. Outra maneira de fazê-lo é através da cópia de uma referência já existente. Esta situação ocorre quando as referências são repassadas através de parâmetros ou do retorno de métodos.

4.8 Gerenciamento das Referências

Assim, como no protocolo *SSP Chains* a cadeia de referências na *Virtuosi* pode crescer indefinidamente (2.4.3), devido a várias migrações que um objeto pode realizar. Também, como no *SSP Chains* sempre existirá um *stub* apontando para um único *scion*. Nunca o *stub* referenciará diretamente um objeto local, quem faz esta referência é sempre um *scion*. Isto, é semelhante ao que acontecia anteriormente com as entradas remotas (seção 2.4.2), quando as entradas remotas nunca apontavam diretamente para a entrada na tabela remota do objeto referenciado.

4.8.1 Referenciamento Remoto entre Objetos

Originalmente na Virtuosi a diferenciação do endereço remoto de um local de objeto se dava na verificação de qual tipo de entrada da tabela de objetos estava preenchida. As entradas podiam ser locais ou remotas. No entanto, em razão de se facilitar a identificação das entradas das tabelas e do heap dos objetos que já são mais referenciados, substituiu-se estes tipos de entradas por tipos de objetos, respectivamente objetos locais e objetos remotos. Solução similar ao que é adotada em outros sistemas distribuídos como o CORBA [cor] e RMI Java [jav].

Assim, outra alteração ocorre nas chamadas remotas na Virtuosi que acontecem se o alvo for um objeto remoto, não mais pelo tipo de entrada, procedimento descrito na seção 2.5.2. Pois um objeto remoto tem informações sobre qual máquina virtual pertence e o seu endereço na tabela de objetos remota.

Além de existir tipos de objetos (local e remoto), alterou-se o modo de se referenciar um objeto remoto. Anteriormente, uma entrada remota na tabela objetos apontava diretamente para uma outra na tabela de objetos remota (ver figura 4.6), e esta que é apontada remotamente não tem nenhuma informação dos seus dependentes remotos. Desta maneira, um coletor local coletaria o objeto referente a esta entrada, porque não haveria nenhum caminho de referências que a partir do conjunto raiz local alcançasse tal objeto. Para que não ocorresse esta coleta incorreta, um objeto remoto aponta para um outro que o representa na máquina virtual alvo, referenciamento similar ao que é implementado no *SSP Chains*, onde temos objetos que representam as referências remotas. Estes objetos são considerados como raízes, assim, os objetos do quais eles dependem não são coletados pelo coletor de lixo local (implementação mostrada pela figura 4.7).

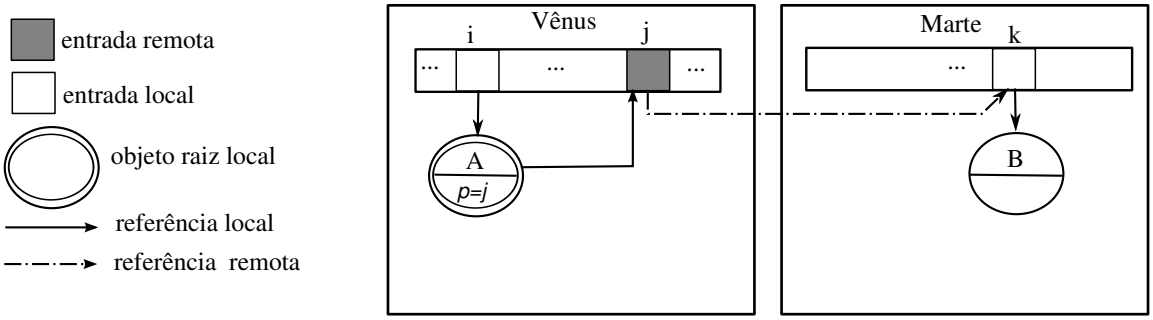

Figura 4.6: Referenciamento remoto utilizando entradas remotas.

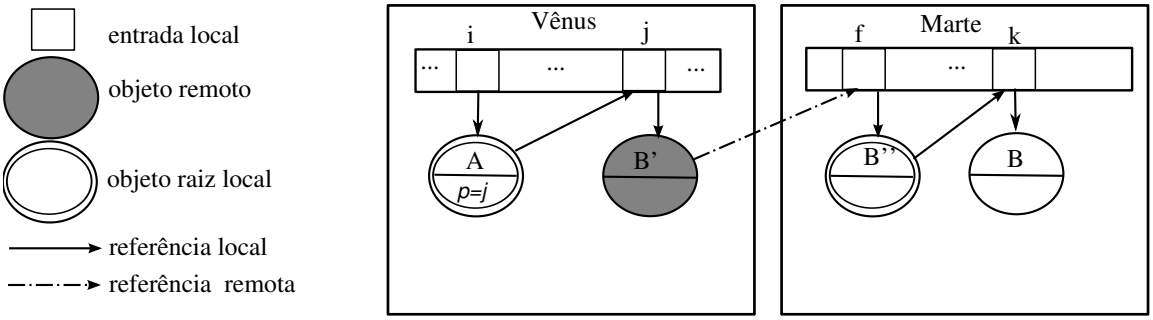

Figura 4.7: Referenciamento remoto utilizando objetos remotos.

Neste trabalho, será adotada nomenclatura igual a do protocolo *SSP Chains* para estes objetos remotos. Assim, os *stubs* representam as referências remotas que uma MVV possui em outra máquina remota. E os *scions* representam aquelas referências remotas que chegam na MVV. Um *stub* apontará sempre para um *scion*. Um *stub* conterá o nome da MVV e o índice da entrada do *scion* correspondente. E o *scion* aponta para a uma entrada de um objeto local. Estes objetos são transparentes para a aplicação, eles servem somente para fazer a comunicação remota entre os objetos que a aplicação realmente manipula. Com esta alteração, as referências remotas entre os objetos da aplicação ficariam da forma apresentada na figura 4.7. Neste caso o *stub* seria o objeto **B'** e *scion* **B"**.

E, semelhante, ao que ocorre no SSP Chains optou-se também por não usar um mesmo scion para máquinas virtuais diferentes, assim se existir um objeto X sendo referenciado por n MVVs teremos n scions. Outra característica do scion, quando criado ele é

colocado no conjunto raiz, saindo deste, somente quando a MVV responsável informar que não está mais "interessada" no objeto que o scion aponta. Conforme é ilustrado na figura 4.8, na qual o objeto **B** é referenciado por dois *scions* um para cada máquina virtual.

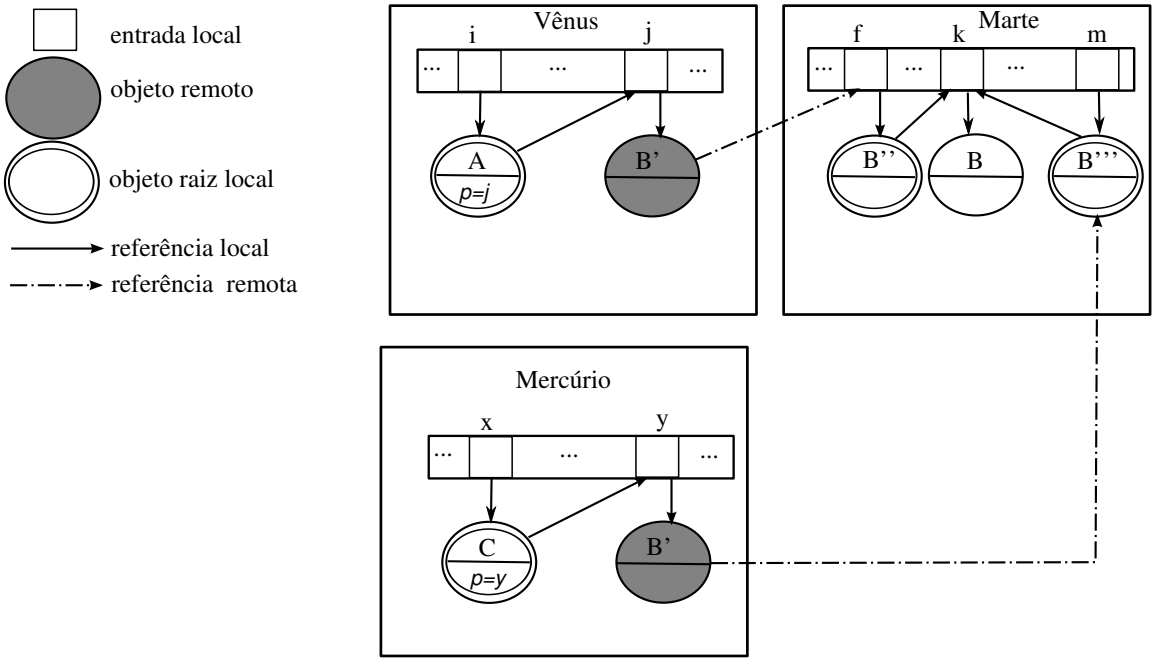

Figura 4.8: Vários referenciamento remotos para um objeto.

4.8.2 Inlists

Um dos tipos de objetos que pertencerão ao conjunto de objetos raízes na Virtuosi serão os scions. Pois se não for assim, o coletor de lixo local os coletaria pois não haveria nenhuma referência local para tais objetos. A forma de mantê-los é através das *inlists*, semelhantes ao do algoritmo *Maheshwari e Liskov*. A *inlist* implementada na Virtuosi pela classe VMInlist. Os principais métodos para manutenção da *inlist* são:

- `void add(String vmName, Integer address)`: adiciona um scion a *inlist*;
- `void remove(String vmName, Integer address)`: remove um scion a *inlist*;
- `void getObjects()`: retorna todos os objetos adicionados à inlist pelo método `add`.

Os parâmetros dos métodos `add` e `remove` são: `vmName`, que seria o nome da MVV que está referenciando um objeto; o segundo parâmetro `address` representaria o índice na tabela de objetos.

O método `add` cria um *scion*, ou seja um objeto local referenciando o objeto indicada pelo índice recebido como parâmetro. E o método `remove` retira um *scion* da *inlist*.

O método `getObjects()` retorna todos os scions da inlist que são utilizados pelo coletor de lixo como objetos raízes.

A atualização da inlist é feita com a comunicação entre as MVVs que atualizam os objetos que são referenciados remotamente. As MVVs geram mensagens de atualização da *inlist* logo ao final do processo de coleta de lixo. Esta mensagem é dita *outlist*, o conteúdo desta mensagem é uma lista de endereços dos scions que ainda estão sendo referenciados pela máquina virtual em questão. Todo este processo de atualização é semelhante ao do algoritmo de *Maheshwari e Liskov*. Assim a *inlist* é atualizada pelos seguintes métodos:

- `void bufferNewInlist(VMMessage message)`: guarda as novas referências remotas para *scions* contidas no parâmetro message.
- `void updateInlist()`: este método faz atualização dos scions que foram recebidos pelo método `bufferNewInlist`. A antiga lista de *scions* de todas MVVs são substituídas pelas novas recepcionadas.

Através dos dois métodos da `VMInlist` anteriormente descritos a mensagem *outlist* é recebida por um máquina virtual e depois atualiza-se a *inlist* na MVV recebedora. No entanto, esta nova *inlist* substitui a antiga no início da próxima execução da coleta de lixo local, a fim de evitar alterações concorrentes. Pois, poderia haver casos em que mensagens de *outlist* fossem processadas durante a coleta de lixo, portanto todo o grafo percorrido anteriormente e depois da atualização da *inlist* estaria incorreto, uma vez que o conjunto raiz se modificaria durante este processo.

Atualização da Inlist

Para exemplificar como é feita a atualização de uma inlist, suponhamos o seguinte cenário dado pela figura 4.9, em que o temos o objeto **A** em Vênus referenciando os objetos **B** e **C** em Marte.

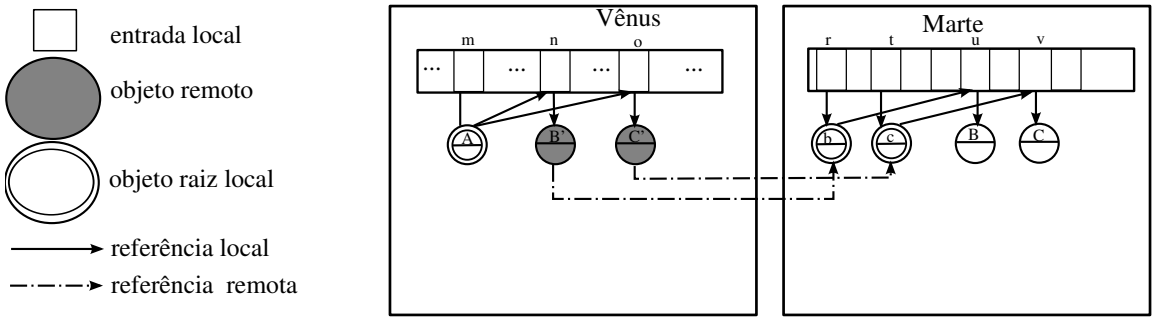

Figura 4.9: Referenciamento dos objetos remotos. Snapshot 1

A *inlist* de Vênus em Marte neste primeiro momento seria esta em 4.10:

```
VMInlist[Vênus] = {b, c};
```

Figura 4.10: Pseudo-código da atualização da inlist de Vênus em Marte. Snapshot 1

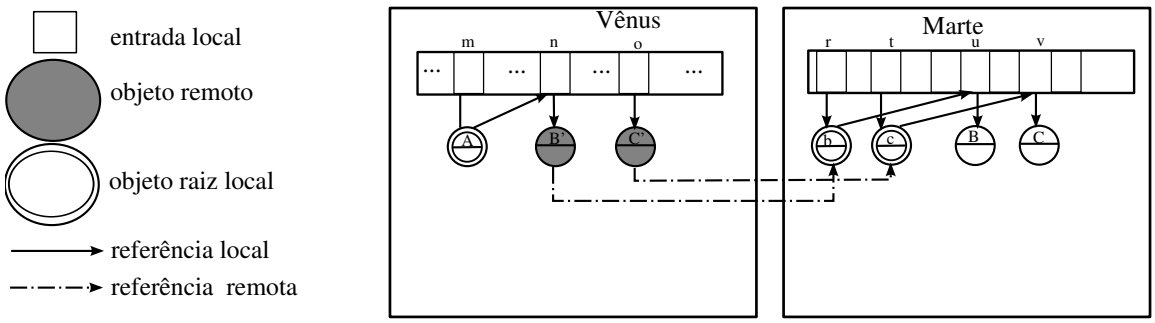

Figura 4.11: Referenciamento dos objetos remotos. Snapshot 2

Agora num próximo momento dado pela figura 4.11 quando o objeto **A** raiz removeu a referência para **C**. A outlist enviada para Marte seria igual que temos em 4.12.

```
Vênus.sendOutlist(Marte, {b});
```

Figura 4.12: Pseudo-código do envio da outlist de Vênus para Marte. Snapshot 1

Depois de processar a *outlist* pela mensagem `bufferNewInlist`, que iria armazenar esta nova *inlist*. A máquina Vênus ao rodar um próxima coleta de lixo local atualizaria sua *inlist* e retiraria o *scion* **c** do conjunto raiz e conseqüemente liberaria os espaços ocupados por **c** e pelo objeto **C**.

Poderia optar-se ao invés de existirem vários *scions* para um mesmo objeto numa máquina local, por um único scion com um contador de referências. Mas, esta implementação foi mantida por ser original ao algoritmo de *SSP Chains*, e nele o seu uso se justifica como forma evitar condições de corrida, em ambientes em que a ordem das mensagens e garantia de entrega de mensagens não são garantidas pelo protocolo de comunicação.

Se justifica o uso da informação da máquina nas *inlists* que referencia um *scion* apesar de ser não utilizada neste trabalho, também como parte tantos dos algoritmos de *SSP Chains* e *Maheshwari e Liskov* foi mantido. Pois no caso algoritmo de *SSP Chains* serve como forma de evitar as condições de corridas e no algoritmo de *Maheshwari e Liskov* é uma informação importante para resolução da coleta de lixo cíclico. Pois, a mensagem de migração do ciclo de lixo inter-nó se dá seguindo de modo inverso as referências entre os objetos de um suposto ciclo. Outra possibilidade pensada na Virtuosi seria a uso desta informação em casos em que se deseja descobrir de um modo direito o dono da referência está ainda no rodando. Por exemplo, caso não esteja por uma parada inesperada, eliminar-se-ia o seus scions.

4.9 Visão Geral do Gerenciamento de Memória na Virtuosi

Nesta seção apresentamos uma visão geral da implementação das principais classes na coleta de lixo distribuído. Dando uma descrição sucinta, para facilitar o entendimento das próximas seções referentes à implementação dos algoritmos propriamente dita.

4.9.1 Classes Principais

A seguir são apresentadas as principais classes implementadas no Gerenciamento de Memória para a *Virtuosi*. A figura 4.13 apresenta os relacionamentos entre estas classes num diagrama de classes utilizando a notação UML (*Unified Modeling Language*).

- VMMemoryManager – classe responsável pelo gerenciamento de memória da MVV.
- VMObject – classe representando uma instância local de uma classe da aplicação.
- VMRemoteObject – classe representando uma instância remota de uma classe da aplicação.
- VMObjectTable – implementa a Tabela de Objetos (seção 2.3.2) da MVV .
- VMEntryTable – representa uma entrada na Tabela de Objetos.
- VMHeap – esta classe representa o local onde os objetos `VMObject` são alocados na MVV.
- VMHeapRemote – esta classe representa o local onde os objetos `VMRemoteObject` são alocados na MVV.
- VMInlist – esta classe é responsável por guardar as instâncias de `VMRemoteObject` referenciadas pelas MVVs remotas.
- VMOutlist – classe responsável por manter as instâncias de `VMRemoteObject` que são referenciadas pela MVV local em outras MVVs.

4.9.2 Mensagens

Para a troca de mensagens entre os coletores de lixo distribuído, foram definidas as seguintes classes:

- VMMessage – classe que representa mensagem utilizada entre os coletores.
- VMMessageParameter – classe que representa um parâmetro de uma mensagem utilizada entre os coletores.

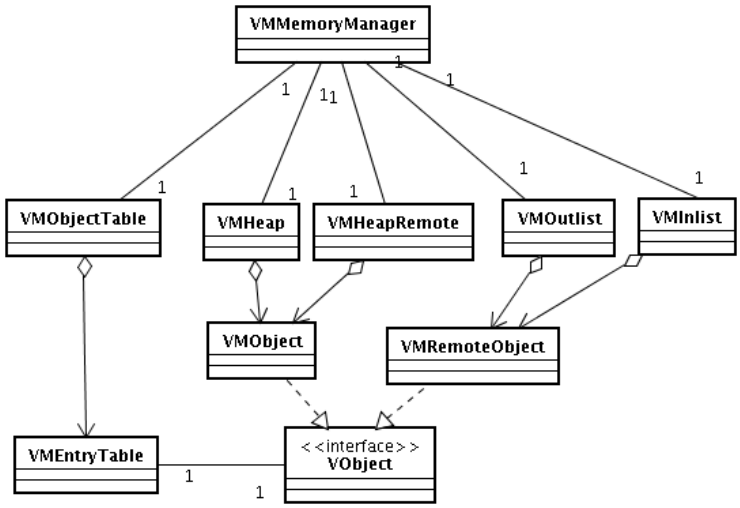

Figura 4.13: Diagrama de Classes

Estrutura das Mensagens

A classe VMMessage é composta dos seguintes atributos:

- Nome – Nome da Mensagem
- Destino – Nome registrado no servidor de nomes (seção 2.5.2) da Máquina Virtual *Virtuosi* destino.
- Origem – Nome registrado no servidor de nomes da Máquina Virtual originadora da mensagem.
- Parâmetros - Parâmetros da VMMessage.

Mensagens Implementadas

Estas mensagens foram implementadas para a coleta distribuída de lixo para o ambiente *Virtuosi*:

- Delete – Mensagem utilizada no algoritmo do protocolo *SSP Chains*.
- Outlist – Mensagem utilizada no algoritmo de *Maheshwari e Liskov*.
- Migration – Mensagem utilizada para migrar um objeto.
- Answer – Mensagem utilizada para a resposta das mensagens acima.

Mensagem Delete Nesta mensagem os campos da VMMessage são preenchidos da seguinte forma:

- Nome – String *delete*.
- Destino – String com o nome da MVV destino.

- Origem – String com o nome da MVV de origem da mensagem.
- Parameters – Tantos objetos VMParameters quanto forem necessários para os endereços que não estão mais sendo referenciados na MVV de origem.

O objeto da classe VMParameter tem o atributo `parameters` preenchido com o endereço do objeto que não está mais sendo referenciado. Neste caso o índice da tabela de objetos da MVV destino, cujo o *scion* era referenciado na MVV de origem.

Mensagem Outlist Nesta mensagem os campos da VMMessage são preenchidos da seguinte forma:

- Nome – String *outlist*.
- Destino – String com o nome da MVV destino.
- Origem – String com o nome da MVV de origem da mensagem.
- Parameters – Tantos objetos VMParameters quanto forem necessários para os endereços que estão sendo referenciados na MVV de origem.

O objeto da classe VMParameter tem o atributo `parameters` preenchido com o endereço do objeto que está sendo referenciado. Neste caso o índice da tabela de objetos da MVV destino, cujo o *scion* era referenciado na MVV de origem. E, também com a distância deste *scion* referenciado.

Mensagem Migration Nesta mensagem os campos da VMMessage são preenchidos da seguinte forma:

- Nome – String *migration*.
- Destino – String com o nome da MVV destino.
- Origem – String com o nome da MVV de origem da mensagem.
- Parameters – Tantos objetos VMParameters quanto forem necessários para os endereços que estão sendo referenciados na MVV de origem pelo objeto que está sendo migrado.

Mensagem Answer Nesta mensagem os campos da VMMessage são preenchidos da seguinte forma:

- Nome – String *answer*.
- Destino – String com o nome da MVV destino.
- Origem – String com o nome da MVV de origem da mensagem.
- Parameters – É preenchido com um objeto VMParameter com valor `OK` ou `NOT OK`. E no caso de sucesso na execução da mensagem *Migration* são adicionados mais objetos VMParameter com valores dos endereços dos objetos migrados para a MVV.

4.9.3 Objetos Raízes

O algoritmo escolhido para fazer a coleta local foi o *mark-sweep*, este algoritmo, para distingüir os objetos vivos do lixo, faz uma marcação dos objetos partindo do conjunto de raízes. Pois, através de um objeto raiz se determina quando um objeto está vivo ou não. Assim, qualquer objeto que seja alcançável por um objeto raiz é considerado como um objeto vivo, ou seja, no caso da *Virtuosi* não estará ligado a nenhuma atividade (seção 2.3.3).

Deste modo, na *Virtuosi* pode se considerar como objetos raízes objetos que sejam donos de uma atividade. Também, o objeto inicial da Classe Raiz (seção 2.1.3) que inicia uma aplicação também pode ser tido como um objeto raiz. Além destes citados, os objetos remotos que são referenciados por outras MVVs, também serão considerados como objetos raízes, ou seja, aqueles que estarão na *VMInlist*.

Foi acrescentado o atributo *permanentObjects* para facilitar a simulação na classe *VMMemoryManager* para guardar os objetos que permanecem vivos enquanto uma MVV estiver rodando.

4.10 Implementação do *SSP Chains* na *Virtuosi*

Como o meio de comunicação na *Virtuosi* atualmente utilizado é o *socket* sob o protocolo TCP (seção 2.5.1), a implementação do *SSP Chains* realizada não foi a original, mas foi modificada pela implementação das sugestões de simplificações em [Sha93] e resumidas neste trabalho na seção 3.6.3.

O funcionamento do coletor local (*mark-sweep*) utilizando o algoritmo *SSP Chains* resumidamente acontece da seguinte forma:

1. A classe *VMMemoryManager* determina se é preciso rodar a coleta de lixo local.

2. A *VMMemoryManager* inicia o processo de marcação de objetos (*mark*) que são alcançáveis pelos objetos raízes, através da navegação de suas referências (seção 2.1.4).

3. Terminado o processo de marcação inicia o processo de limpeza (*sweep*), ou seja a *VMMemoryManager* percorre a *VMHeap* e faz a desalocação do objetos locais não marcados e da sua correspondente entrada na tabela de objetos locais. Faz o mesmo processo na *VMHeapRemote* que além de fazer a desalocação dos objetos remotos não marcados, cria uma lista de endereços (que seria o nome da MVV remota e o endereço remoto do objeto).

4. No fim da coleta de lixo é criada uma instância de *VMMessage Delete* para cada MVV remota com os endereços dos objetos que não estão mais sendo referenciados.

5. A MVV destino ao receber a *VMMessage Delete* remove o objeto remoto correspondente ao endereços contidos na mensagem, da sua *VMInlist*.

Exemplificando, dado o cenário pela figura 4.14 no qual temos os objetos **B**, **C** e **D** referenciados remotamente pela máquina Vênus em Marte. Em Marte os *scions* **b**, **c** e **d** são os objetos raízes pois são referenciados pelos respectivos *stubs*.

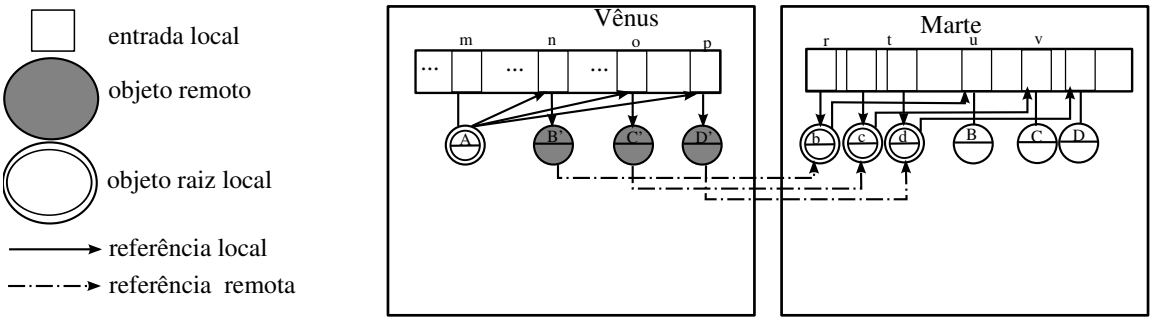

Figura 4.14: Implementação do SSP Chains. Snapshot 1

Num segundo momento dado pela figura 4.15, a referência de **A** para **B** é removida.

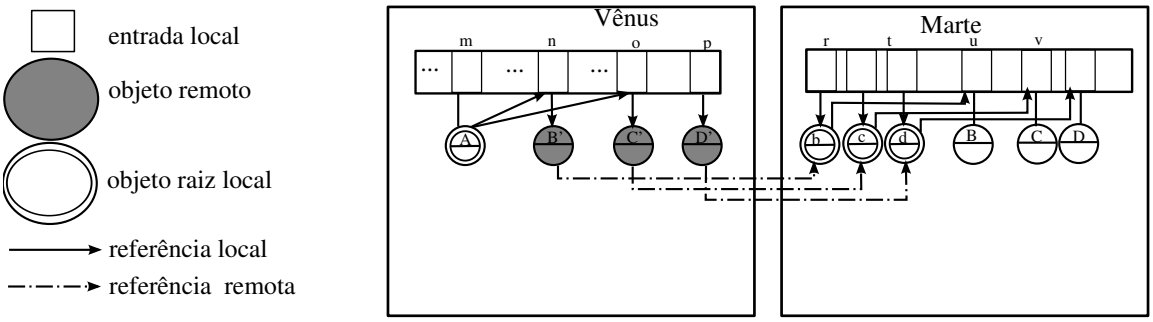

Figura 4.15: Implementacao do SSP Chains. Snapshot 2

Ao rodar a coleta de lixo em Vênus. Teríamos a seguinte situação os objetos **B** e **C** estariam marcados pois são alcançáveis pela raiz **A**. Na fase de sweep seria desalocada a entrada **p** e o objeto **D**. Ao final desse processo Vênus enviaria a seguinte mensagem *Delete*, como no código 4.16:

```
Vênus.sendDeleteMessage(Marte, {d});
```

Figura 4.16: Pseudo-código da mensagem Delete de Vênus para Marte. Snapshot 2

E a máquina Marte ao receber a mensagem de *Delete* atualizaria sua *inlist*, antes do início do processo de coleta de lixo local, removendo o scion **b** da inlist Vênus.

4.11 Implementação do algoritmo de Maheshwari e Liskov

A implementação do algoritmo de Maheshwari e Liskov não foi completa. Não foi implementada na totalidade a parte do algoritmo que trata da coleta de lixo cíclico.

Somente foi implementada a parte da heurística da distância, ou seja, na propagação da distância (seção 3.6.4), faltando o mecanismo de migração (seção 3.6.4) de objetos para a resolução do lixo cíclico distribuído. Mesmo assim, é possível descobrir quando existem objetos "suspeitos" de estarem num ciclo de lixo distribuído pela distância dos objetos remotos.

O funcionamento do coletor local (*mark-sweep*) utilizando o algoritmo de *Maheshwari e Liskov* resumidamente acontece da seguinte forma:

1. A classe *VMMemoryManager* determina se é preciso rodar a coleta de lixo local.

2. *VMMemoryManager* inicia o processo de marcação de objetos (*mark*) que são alcançáveis pelos objetos raízes. Aqui os objetos raízes são ordenados em ordem crescente das suas distâncias como explicado na seção 3.6.4, para que durante a marcação dos objetos tenham associada a menor distância possível em relação ao objeto raiz.

3. Terminado o processo de marcação inicia o processo de limpeza (*sweep*), a *VMMemoryManager* percorre a *VMHeap* e faz a desalocação dos objetos locais não marcados e da sua correspondente entrada na tabela de objetos locais. Diferente do que acontece no algoritmo *SSP Chains* ao se percorrer *VMHeapRemote* é criada uma lista do objetos remotos marcados, isto é, os objetos vivos.

4. Ao final do processo de limpeza é enviada uma mensagem *Outlist* para cada MVV referenciada, nela vão os endereços que continuam sendo referenciados e além da distância do objeto mais um (1). Caso, por exemplo a MVV não referencie nenhum objeto numa MVV que anteriormente tinha alguma referência a mensagem *Outlist* irá sem nenhum parâmetro, indicando que não está referenciando mais nenhum objeto.

5. A MVV destino ao receber a mensagem *Outlist*, faz atualização das entradas *VMInlist* correspondente à MVV de origem pelos novos endereços recebidos.

Exemplificando, dado o mesmo cenário já apresentado no algoritmo *SSP Chains* pela figura 4.17 no qual temos os objetos **B**, **C** e **D** referenciados remotamente pela máquina Vênus em Marte. Em Marte os *scions* **b**, **c** e **d** são os objetos raízes pois são referenciados pelos respectivos *stubs*.

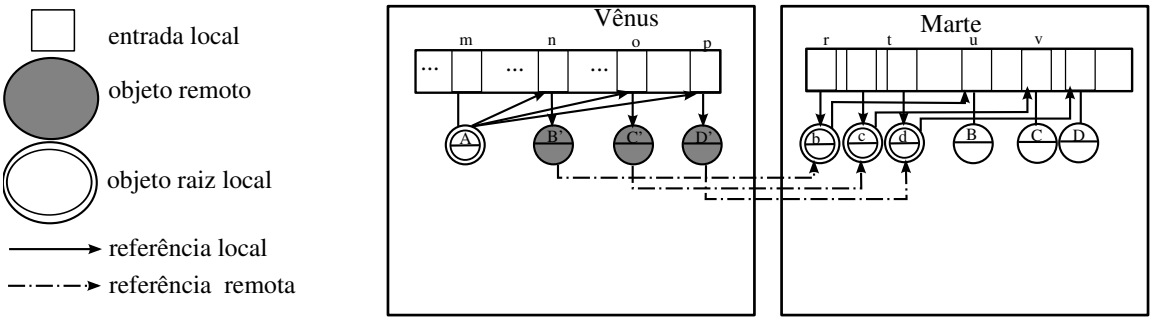

Figura 4.17: Implementação do algoritmo de Maheshwari e Liskov. Snapshot 1

E da mesma forma o objeto D tem sua referência com desfeita. O coletor de lixo local que é o mesmo, executará o mesmo processo de marcação dos objetos **B** e **C**. Reciclará a entrada p e espaço ocupado por D. Contudo a mensagem enviada será a de outlist neste caso irá com os objetos **B** e **C**, como mostrado em 4.18, além dessa informação das referências ainda utilizadas, associadas as elas estão as distâncias de cada uma relação à raiz local. Neste caso, a distância da raiz local das duas referências seria zero, e ao transmitir esta distância é acrescentada em um (1).

```
Vênus.sendOutlistMessage(Marte, {b, c});
```

Figura 4.18: Pseudo-código da mensagem Outlist de Vênus para Marte.

4.12 Conclusão

Durante a implementação dos algoritmos escolhidos foi possível realmente comprovar as poucas alterações no ambiente da *Virtuosi*. Houve pequenas modificações na tabela de objetos, quanto aos tipos de entradas, substituídas por tipos de objetos. Assim, atingiu-se o objetivo de isolar os impactos na incorporação da coleta de lixo distribuída. Também foi atingido o objetivo de se criar um modelo para o gerenciamento de coleta de lixo distribuída que permitisse o uso de dois algoritmos de coleta de lixo.

Capítulo 5

Simulações

5.1 Ambiente de Simulação

O ambiente Java foi utilizado na simulação. Seguindo, as especificações da Arquitetura da *Virtuosi* [Cal00] e o modelo de migração de objetos [dCCF04], implementou-se somente a parte referente ao gerenciamento da Área de Objetos (seção 2.3.2) e a migração de objetos sem a migração de classes. Os testes foram realizados numa mesma máquina com as seguintes especificações: processador Pentium 4 2.8GHz com 512 MB de memória, sistema operacional Linux Fedora 6. Limitou-se a simulação a duas MVVs e a população máxima de objetos locais a cerca de 800 objetos.

A justificativa em se utilizar um ambiente de simulação ao invés de um sistema real é de se ter um maior controle sobre o ambiente em observação. Simulou-se o que seriam os comportamentos típicos de uma aplicação em relação ao uso de memória, nos quais temos fases de crescimento da população de objetos alocados, de estabilização desta população e de diminuição da quantidade objetos. O processo de simulação ocorre em paralelo em todas as máquinas, conforme a configuração dos parâmetros está em 5.1, conseguiu-se a variação necessária da população de objetos. Seria possível trabalhar com outras configurações para se obter a mesma variação da população, mas utilizou-se esta configuração devido a limitação do ambiente de testes.

fase	criar	raiz	referenciar	migrar	remover
crescimento	100	10	10	2	0
estável	100	10	10	2	10
diminuição	0	0	0	2	1

Tabela 5.1: Tabela com parâmetros da simulação.

O objetivo da simulação foi medir a utilização da banda consumida pelos algoritmos de coleta de lixo implementados. A medição baseou-se na quantidade de parâmetros utilizados nas mensagens do coletor de lixo. Os parâmetros são os endereços na tabela objetos. No caso algoritmo *SSP Chains* são os endereços dos objetos removidos e no *Maheshwari e Liskov* endereços dos objetos vivos. Foi medida somente a quantidade de endereços pois as outras estruturas da mensagem são praticamente iguais, exceto, a distância associada aos objetos na mensagem de *outlist* no algoritmo *Maheshwari e Liskov*.

Não foi utilizado uma medida como bytes por exemplo, pois desta forma a medida está "normalizada", assim continuaria válida mesmo se existisse uma mudança na codificação dos endereços transmitidos. A expectativa para os testes é que o algoritmo de SSP Chains obtenha uma melhor desempenho nas fases em haja um maior número de objetos sendo criados do que removidos. Seriam nas fases de crescimento e estabilização da população de objetos, já que a mensagem *Delete* tem como parâmetros os objetos que deixaram de ser referenciados e a mensagem *Outlist* de *Maheshwari e Liskov* seria maior pois os parâmetros são os objetos ainda referenciados.

5.2 Crescimento da População de Objetos Vivos

Para simular o crescimento da população de objetos vivos, a cada iteração da simulação foram criados 100 novos objetos locais, 10 objetos locais foram escolhidos aleatoriamente para serem raízes, foram criadas 10 referências para cada um dos objetos criados e 2 objetos eram migrados. O gráfico representado pela figura 5.1, mostra o comportamento típico durante as várias simulações da população de objetos para as duas máquinas virtuais e os dois algoritmos de coleta de lixo escolhidos: *SSP Chains* e *Maheshwari e Liskov*, não houve nenhuma influência tanto da MVV ou do algoritmo, pois são somente influenciados pelos parâmetros de configuração da simulação. Os dados no gráfico foram colhidos a cada cinco iterações.

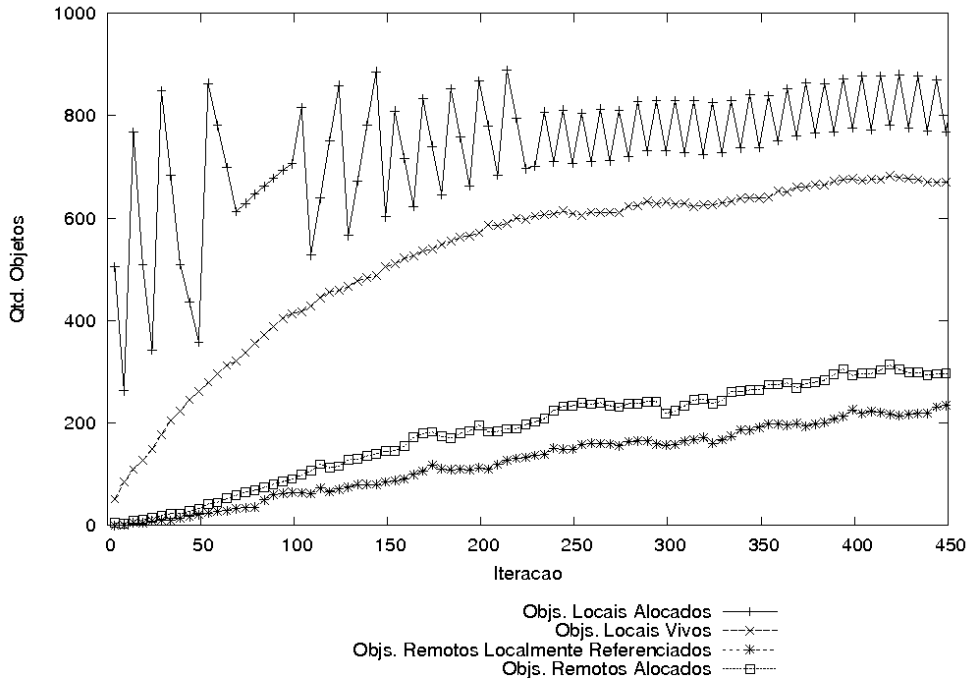

Figura 5.1: Crescimento da Quantidade de Objetos Vivos

A linha *Objs. Locais Alocados* representa o número total de objetos locais alocados, estejam eles vivos ou não. Os picos de objetos situam-se em torno de 800 objetos, pois o limite para disparar o coletor de lixo local era 200 células livres no *VMHeap*. Nota-se também a presença de "vales" e "picos" nesta linha, isto se dá pela execução do coletor

de lixo durante a simulação. Os "vales" não chegam a tocar a linha *Objetos Vivos* como poderia se supor, pois a coleta de dados se dava logo após a alocação dos objetos. Nota-se isto pelo intervalo de cerca de 100 objetos entre duas linhas.

A linha *Objs. Locais Vivos* representa o número total de objetos locais alocados que são alcançáveis por objetos raízes (seção 4.9.3). Nota-se como esperado o seu crescimento.

A linha *Objs. Remotos Referenciados Localmente* representa o número total de objetos remotos alocados que eram referenciados por algum objeto local. Como esperado, percebe-se o crescimento desta população.

A linha *Objs. Remotos Alocados* representa o número total de objetos remotos alocados estejam referenciados localmente ou remotamente. Também vemos o crescimento desta população.

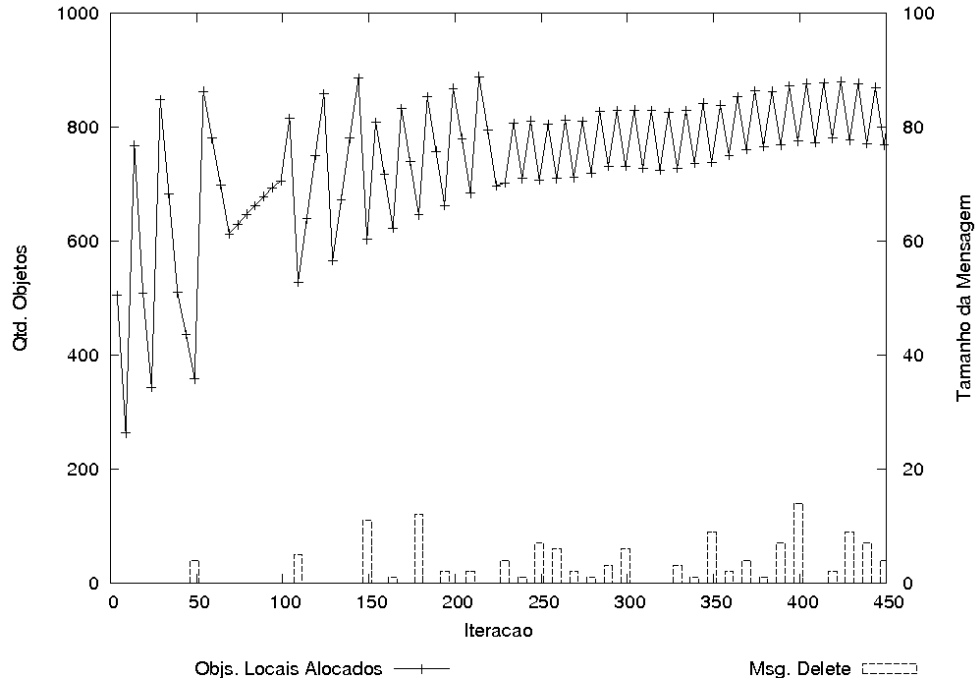

Figura 5.2: Tamanho da Mensagens Trocadas Durante o Aumento da População de Objetos - SSP Chains

No gráfico representado pela figura 5.2, a abscissa representa as iterações da simulações, o tamanho da mensagem *Delete* enviada pelo algoritmo *SSP Chains* (seção 4.9.2), neste caso é dado pelo número de parâmetros (VMParameter). A freqüência das mensagens varia de acordo com a freqüência dos vales da linha *Objs. Locais Alocados*, ou seja, os vales ocorrem devido a execução da coleta de lixo e por conseqüência são enviadas mensagens *Delete*. O tamanho das mensagens varia pouco porque a distância entre as curvas das *Objs. Remotos Localmente Referenciados e Objs. Remotos Alocados* no gráfico 5.1 se mantiverem quase constante. Esta diferença é igual ao total de objetos remotos que deixaram de ser referenciados.

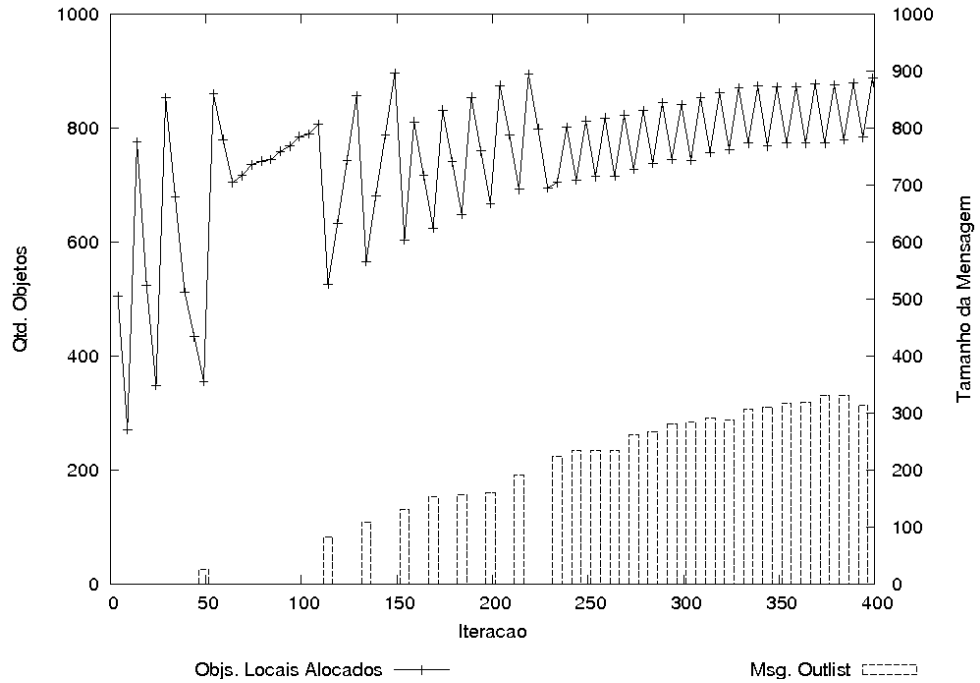

Figura 5.3: Tamanho da Mensagens Trocadas Durante o Aumento da População de Objetos da População de Objetos - Maheshwari e Liskov

No gráfico representado pela figura 5.3 a abscissa representa as iterações da simulações, o tamanho da mensagem *Outlist* (seção 4.9.2) enviada pelo algoritmo *Maheshwari e Liskov* é dado pelo seu número de parâmetros (*VMParameter*). Aqui, também pode ser notado que a freqüência das mensagens *Outlist* varia de acordo com a freqüência do vales da *Objs. Locais Alocados*, quando ocorre a coleta de lixo e o envio das mensagens. Já o tamanho das mensagens aumenta com a população de *Objs. Remotos Referenciados Localmente* mostrada na curva como pode ser visto no gráfico 5.1

5.3 Estabilização da População de Objetos Vivos

Para simular a estabilização da população de objetos vivos, a cada iteração da simulação foram criados 100 novos objetos locais, 10 objetos locais foram escolhidos aleatoriamente para serem raízes, e 10 objetos do conjunto de objetos raízes eram retirados. Também foram criadas 10 referências para cada um dos objetos criados e 2 objetos eram migrados. O gráfico representado pela figura 5.4 mostra o comportamento típico da população de objetos para as duas máquinas virtuais e os dois algoritmos de coleta de lixo: *SSP Chains* e *Maheshwari e Liskov*, pois não houve influência da MVV ou do algoritmo, somente da configuração dos parâmetros da simulação.

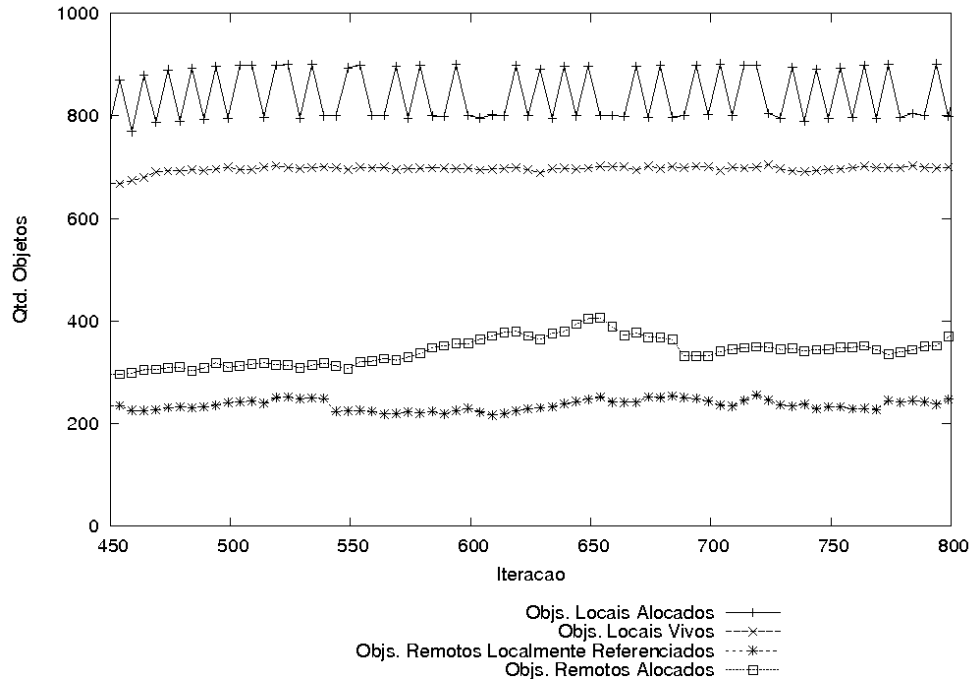

Figura 5.4: Estabilização da Quantidade de Objetos Vivos

A linha *Objs. Locais Alocados* como o esperado se manteve estável. E os picos também situam-se por volta de 800 objetos por causa da configuração coletor local. Também, como já foi comentado na seção 5.2 os vales da linha amostrada fica por 100 objetos acima da linha *Objs. Locais Vivos*.

A linha *Objs. Locais Vivos* como o esperado esta linha se manteve estável.

A linha *Objs. Remotos Localmente Referenciados* também apresentou poucas alterações.

A linha *Objs. Remotos Alocados* também apresentou poucas alterações.

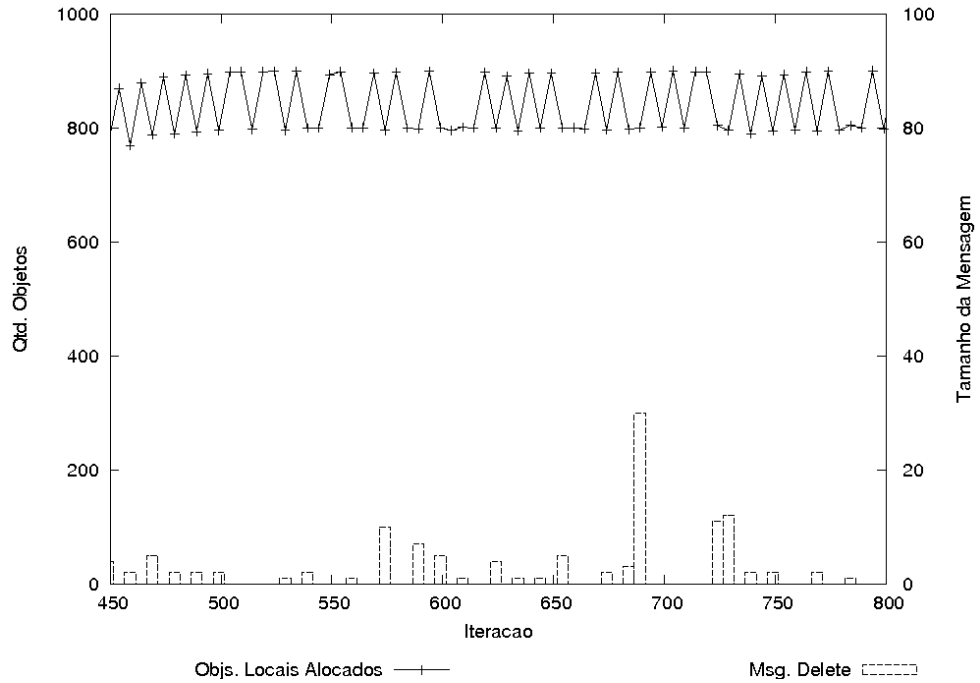

Figura 5.5: Tamanho da Mensagens Trocadas Durante a Estabilização da População de Objetos - SSP Chains

No gráfico representado pela figura 5.5, a abscissa apresenta as iterações da simulações, o tamanho da mensagem *Delete* enviada pelo algoritmo *SSP Chains* (seção 4.9.2), neste caso é dado pelo número de parâmetros (VMParameter). Também a freqüência da mensagem *Delete* como esperado varia de acordo com a freqüência da coleta de lixo representada, pela freqüência dos vales *Objs. Locais Alocados*. Os tamanhos das mensagens variam de acordo com a distância das linhas *Obj. Remotos Localmente Referenciados* e *Objs. Remotos alocados*. E, o tamanho máximo registrado ficou pouco abaixo dos 40 parâmetros.

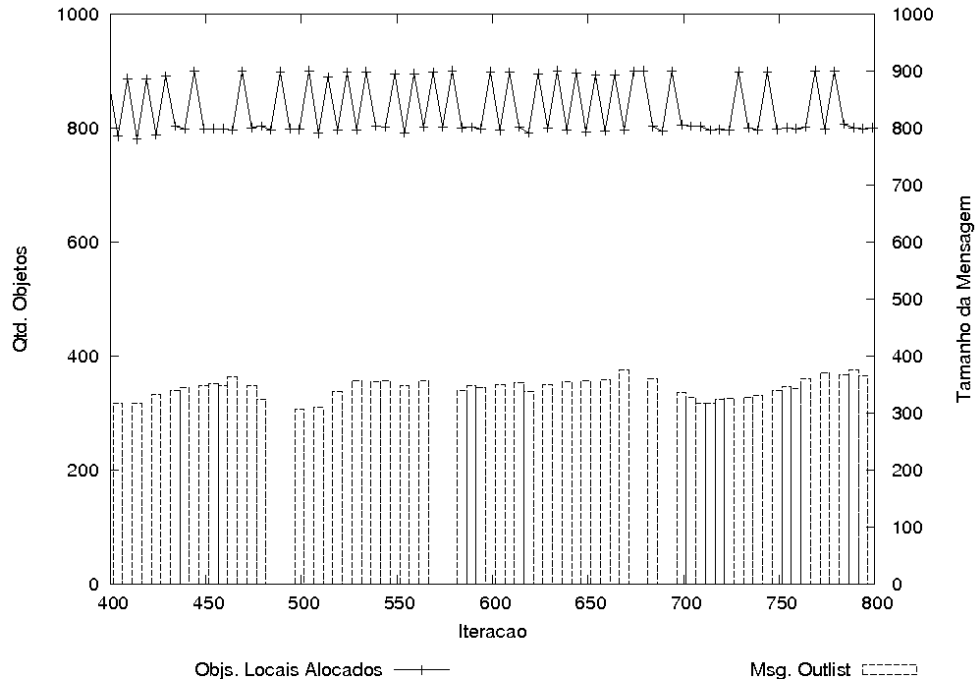

Figura 5.6: Tamanho da Mensagens Trocadas Durante a Estabilização da População de Objetos - Maheshwari e Liskov

No gráfico representado pela figura 5.6 apresenta a simulação com algoritmo de *Maheshwari e Liskov*. Aqui, também pode ser notado que a freqüência das mensagens *Outlist* varia de acordo com a freqüência do vales da linha de *Objs. Locais Alocados*, pois quando ocorre a coleta de lixo e o envio das mensagens. O tamanho das mensagens se mantém constante. Mas, em comparação ao tamanho das mensagens *Delete* enviadas pelo algoritmo *SSP Chains* no qual pelo gráfico 5.5 não chega a 40 aqui o tamanho fica por volta de 350, isto é devido a população de objetos remotos localmente referenciados estarem nesta faixa.

5.4 Diminuição da População de Objetos Vivos

Para simular a diminuição da população de objetos vivos, a cada iteração da simulação nenhum novo objeto local era criado, 1 objeto era retirado do conjunto raiz e as suas referências para outros objetos foram removidas e 2 objetos locais eram migrados. O gráfico representado pela figura 5.7 mostra o comportamento típico da população de objetos para as duas máquinas virtuais e para os dois algoritmos de coleta de lixo: *SSP Chains* e *Maheshwari e Liskov*, pois não houve nenhuma influência da MVV ou do algoritmo. Somente a influência dos parâmetros da simulação.

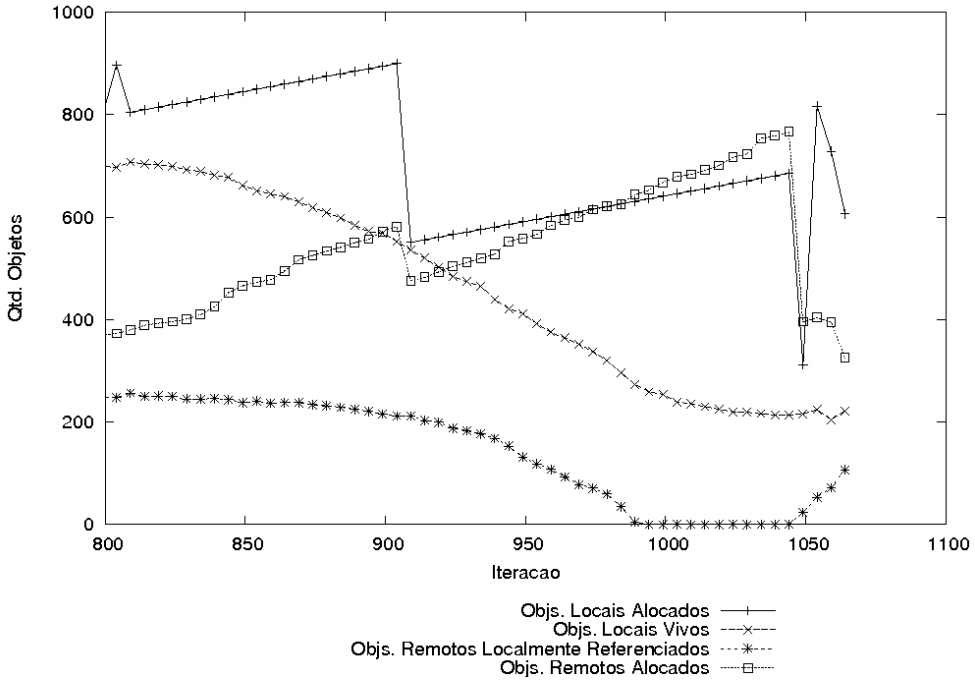

Figura 5.7: Diminuição da Quantidade de Objetos Vivos

A linha *Objs. Locais Alocados* como o esperado entrou em queda. Assim como a freqüência dos picos e vales diminuiu pois não havia mais a alocação de novos objetos durante a simulação.

A linha *Objs. Locais Vivos* como o esperado esta linha entrou em queda.

A linha *Objs. Remotos Localmente Referenciados* também entrou em queda e chegou a zero.

A linha *Objs. Remotos Alocados* apresentou um crescimento, pois o número de objetos migrados na configuração da simulação foi mantido em 2. E, como a coleta de lixo não era disparada com o passar do tempo a quantidade de objetos remotos aumentava.

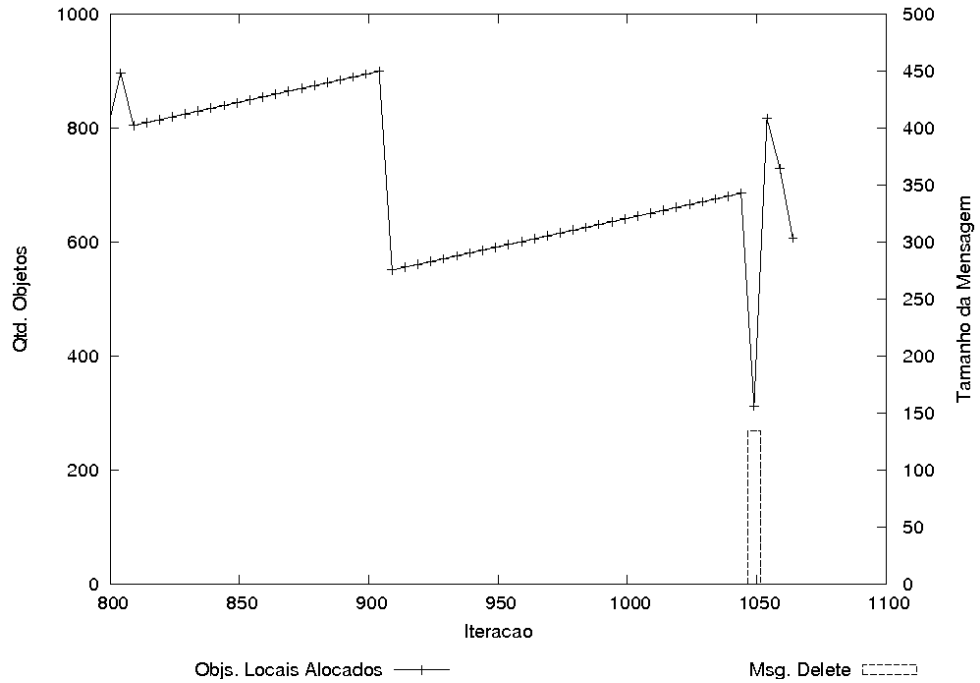

Figura 5.8: Tamanho da Mensagens Trocadas Durante a Diminuição da População de Objetos Locais - SSP Chains

No gráfico representado pela figura 5.8, nota-se a única freqüência do envio da mensagem *Delete*, a razão disto é devido a própria diminuição da população de objetos. Como a população estava diminuindo, o coletor local não foi disparado. Somente ocorreu uma chamada quando houve uma retomada do crescimento da população e a ocorrência de uma coleta de lixo. Importante, notar que o tamanho da mensagem *Delete* atingiu seu maior tamanho entre as outras simulações (de crescimento e de estabilização), chegando próximo a 150. Pois, durante esta simulação um maior número de objetos remotos deixou de ser referenciado, o que pode ser percebido na linha de *Objs. Remotos Localmente Referenciados* no gráfico 5.7.

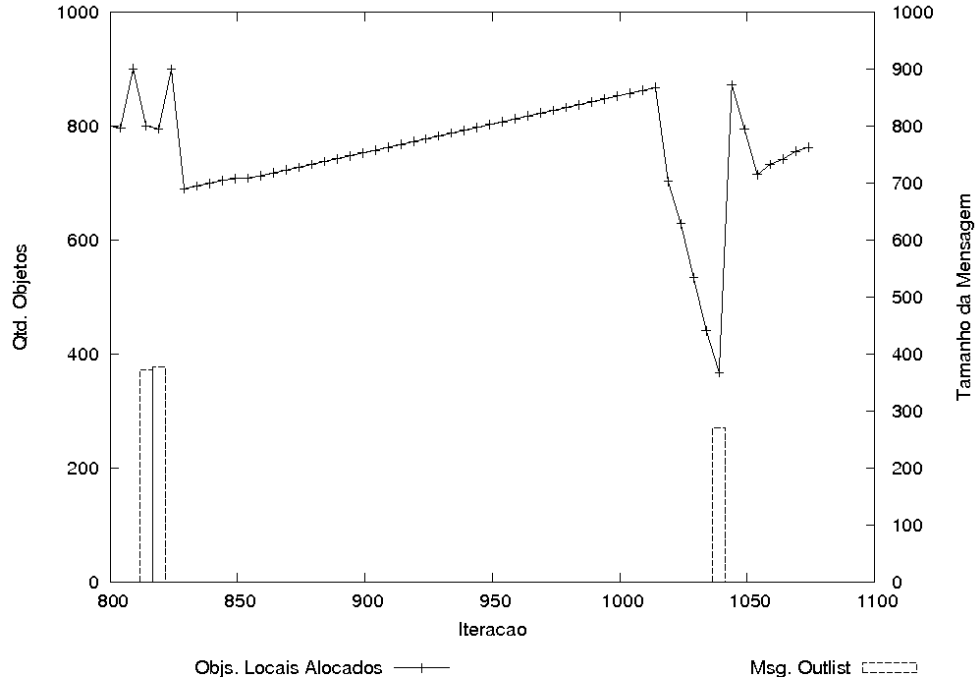

Figura 5.9: Tamanho da Mensagens Trocadas Durante a Diminuição da População de Objetos Locais - Maheshwari e Liskov

No gráfico representado pela figura 5.9, também se nota a pouca freqüência das mensagens *Outlist* ocorridas devido a não execução do coletor de lixo local durante a simulação. E o tamanho das mensagens se manteve maior que no algoritmo que no algoritmo *SSP Chains*.

5.5 Conclusão

Em todas as simulações realizadas notou-se que o algoritmo *SSP Chains* utilizou mensagens de tamanho menor em relação ao algoritmo de *Maheshwari e Liskov*. Isto ocorreu porque o *SSP Chains* na mensagem de *Delete* tem como parâmetros os objetos não mais referenciados, diferente do algoritmo de *Maheshwari e Liskov* que na mensagem *Outlist* tem como parâmetros os objetos remotos que ainda estão vivos. Como nas simulações de crescimento e estabilização de população de objetos existiam mais objetos vivos do que mortos, o algoritmo *SSP Chains* obteve um desempenho melhor no consumo de banda. E, mesmo quando se reduziu a população de objetos, situação em que haveria mais objetos mortos, não houve ganho para o algoritmo de *Maheshwari e Liskov* pois o coletor de lixo local não foi ativado devido ao nível de ocupação da memória isto não era necessário, conseqüentemente a mensagem de *Outlist* não foi enviada. Isto só viria a ocorrer caso houvesse um novo aumento, suficiente para provocar uma coleta de lixo, mas mesmo neste caso a mensagem de *Outlist* estaria preenchida com novas referências criadas. Já a freqüência das mensagens para ambos algoritmos mostrou-se similar uma vez que isto dependia da ocorrência de uma coleta local de lixo que foi o mesmo para os dois algoritmos.

Estes comportamentos esperados serviram para validar a implementação dos coletores de lixo no ambiente *Virtuosi* e mesmo em outros que utilizem o mesmo tipo de referenciamento de objetos, através de uma tabela de objetos. As implementações se mostraram bem adaptáveis à Virtuosi, pois não exigiram nenhuma alteração no conceito do referenciamento utilizado. Alteração, apenas na implementação das tabelas ao se substituir as entradas locais e remotas pelos respectivamente pelos objetos locais e remotos.

Mas, vale ressaltar que o algoritmo completo de *Maheshwari e Liskov* tem como característica a detecção de lixo cíclico distribuído. Algo, que pode ser importante em sistemas que rodem por um longo tempo ou possuam esquemas de persistência de objetos [RS00]. Outro ponto a se ressaltar, é a necessidade de outros experimentos em que se meçam outras propriedades importantes como:

- A latência da rede, neste caso simular as trocas de mensagens em VMs localizadas em máquinas diferentes.
- A escabilidade dos algoritmos implementados quando se utilizam várias máquinas ou se aumenta a quantidade de objetos manipulados.
- A influência da freqüência de coleta dos algoritmos locais. Algoritmos que rodem influenciados por outros fatores além do limite de ocupação do heap.

Capítulo 6

Conclusão e Trabalhos Futuros

Um desafio enfrentado neste trabalho foi não alterar os modelos de objetos, de classes e de atividades já existentes na *Virtuosi* ao se introduzir o gerenciamento de memória distribuída. Outro desafio, foi adaptar conhecidos coletores de lixo distribuídos dentro da *Virtuosi* segundo seu esquema de objetos, de referências por meio de uma tabela de objetos e de chamadas de remotas de procedimentos.

Partindo destes desafios, concluiu-se a viabilidade de implementação de um mecanismo de coleta de lixo distribuída dentro da *Virtuosi* sem a alteração de seu modelo de referenciamento de objetos através de uma tabela de objetos. Também, a possibilidade de se implementar o mesmo esquema coleta de lixo distribuído apresentado neste trabalho em sistemas distribuídos que utilizem o mesmo modelo de referenciamento de objetos.

A principal contribuição deste trabalho foi realizar um estudo inicial de conhecidos de coletores distribuídos no ambiente *Virtuosi*. Assim, servir de base de comparação para a implementação de outros algoritmos de coleta de lixo distribuída. Pois, desenvolveu-se um método instrumentado de verificação e de análise de desempenho de algoritmos de coleta de lixo distribuída. E, com base nas simulações apresentadas neste trabalho é possível estabelecer alguns parâmetros para a utilização de um ou outro algoritmo implementado.

Segue, uma lista de itens como trabalhos futuros:

- Realizar simulações com maior variação nos seus parâmetros;
- Implementar detecção de ciclos multi-nós;
- Implementar *shortcutting* (seção 3.6.3);
- Implementar algoritmos de marcação distribuída;
- Implementar algoritmos mais recentes como [Fes01], [KNBH01] e [VF05];
- Implementar em sistemas reais;
- Desenvolver coleta de lixo para a Área de Classes da Virtuosi.

Como um item listado, destaca-se a implementação de algoritmos marcação distribuída e outros mais recentes. Pois além de acrescentar novos dados para comparação entre si, pode comprovar ou não, quão genérico e adaptável é o esquema de coleta de lixo para a Virtuosi apresentado aqui para outros diferentes algoritmos.

Referências Bibliográficas

[Bac04] Bacon, David F. Cheng, Perry Rajan, V. T. A unified theory of garbage collection. *OOPSLA 04*, 2004.

[BOR04] Aron BORGES. Uma linguagem de programação orientada a objetos mínima. 2004. 50 f. Trabalho de Conclusão de Curso (Graduação em Bacharelado Em Ciência da Computação Pontifícia Universidade Católica do Paraná. Orientador: Alcides Calsavara.

[Cal00] Calsavara, Alcides. Virtuosi: Máquinas virtuais para objetos distribuídos. *Trabalho apresentado em concurso para professor titular da PUC-PR*, 2000.

[Col60] George E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, 1960.

[cor] http://www.corba.org/.

[dCCF04] Juarez da Costa Cesar Filho. Mecanismo de mobilidade de objetos para a virtuosi. Master's thesis, Pontifícia Universidade Católica do Paraná– PUCPR, July 2004.

[Fes01] Fabrice Le Fessant. Detecting distributed cycles of garbage in large-scale systems. In *Conference on Principles of Distributed Computing(PODC) 2001*, volume 21, 2001.

[FY69] Robert R. Fenichel and Jerome C. Yochelson. A lisp garbage-collector for virtual-memory computer systems. *Commun. ACM*, 12(11):611–612, 1969.

[HCWZ03] Y. Charlie HU, A. Cox, D. Wallach, and W. Zwaenepoel. Run-tim support for distributed sharing in safe languages. *ACM Transactions on Computer Systems*, 21(1), 2003.

[Hud82] Keller, R.M. Hudak, Paul. Garbage collection and task deletion in distributive applicative processing systems. In *Conference Report of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 168–178. ACM Press, 1982.

[Hut87] Norman Hutchinson. *Emerald: An Object-Based Language for Distributed Programming.* PhD thesis, University of Washington, January 1987.

[jav] http://java.sun.com/.

[Jon96] Jones, Richard Lins, Rafael. *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996.

[KF97] Thomas Kistler and Michael Franz. A tree-based alternative to java byte-codes. In *Proceedings of the International Workshop on Security and Efficiency Aspects of Java '97*, January 1997. Also published as Technical Report No. 96-58, Department of Information and Computer Science, University of California, Irvine, December 1996.

[KNBH01] Erik Klintskog, Anna Neiderud, Per Brand, and Seif Haridi. Fractional weighted reference counting. In *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 486–490, London, UK, 2001. Springer-Verlag.

[Kol04] Carlos José Johanh Kolb. Um sistema de execução para software orientado baseado em Árvores de programa. Master's thesis, Pontifícia Universidade Católica do Paraná– PUCPR, July 2004.

[Kor93] Kordale, K.Ahamad, Mustaque. A scalable cyclic garbage detection for distributed system. *OOPLSA*, 1993.

[Lan92] Land, BernardQuenniac, Christian Piquer, José. Garbage collecting the world. *POPL*, 1992.

[Lis86] Landin, Rivka Liskov, Barbara. Highly available distributed services and fault-tolerant distributed garbage collection. In *Proceeding of the Fifth Annual ACM Symposium on Principles on Distributed Computing*, pages 29–39. ACM Press, 1986.

[MA84] Khayri A. Mohamed-Ali. *Object Oriented Storage Management and Garbage Collection in Distributed Processing Systems*. PhD thesis, Royal Institute of Technology, December 1984.

[McB63] J. Harold McBeth. Letters to the editor: on the reference counter method. *Commun. ACM*, 6(9):575, 1963.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960.

[Met80] Metropolis, N Howllet, J Rota, Gian-Carlo. *A History of Computing in Twentieth Century*. Academic Press, 1980.

[ML95] Umesh Maheshwari and Barbara Liskov. Collecting cyclic distributed garbage by controlled migration. In *Proceedings of PODC'95 Principles of Distributed Computing*, 1995.

[Nod05] Agnaldo Kiyoshi Noda. Mecanismo de invocação remota de métodos em máquinas virtuais. Master's thesis, Pontifícia Universidade Católica do Paraná– PUCPR, July 2005.

[Piq90] M. José Piquer. Sharing date structures in distributed lisp. In *Proceedings of High Performance and Parallel Computing in Lisp Workshop*, 1990.

[PS94] David Plainfosse and Marc Shapiro. A survey of distributed garbage collection techniques. Technical report, BROADCAST, 1994.

[RS00] Nicolas Richer and Marc Shapiro. The memory behavior of the WWW, or: The WWW considered as a persistent store. In Graham Kirby, editor, *Int. W. on Persistent Obj. Sys.*, volume 2135 of *Lecture Notes in Computer Science*, pages 169–184, Lillehammer (Norway), September 2000. Springer-Verlag. `http://www-sor.inria.fr/publi/TMBotWoTWCaaPS_pos2000.html`.

[Sha93] M. et al. Shapiro. Ssp chains: Robust, distributed references supporting acyclic garbage collection. RFC 2, University of Newcastle, 1993.

[Ung84] Ungar, David M. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 1984.

[VF05] Luís Veiga and Paulo Ferreira. Asynchronous complete distributed garbage collection. In *19 IEEE International Parallel and Distributed Processing Symposium*, volume 1, 2005.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.