UFSM

Trabalho de Graduação

JRastro: Ferramenta para o rastreamento de programas Java paralelos e distribuídos usando JVMTI

Geovani Ricardo Wiedenhoft

Curso de Ciência da Computação

Santa Maria, RS, Brasil

2005

JRastro: Ferramenta para o rastreamento de programas Java paralelos e distribuídos usando JVMTI

por

Geovani Ricardo Wiedenhoft

Trabalho de Graduação apresentado ao Curso de Ciência da Computação – Bacharelado, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação.

Curso de Ciência da Computação

Trabalho de Graduação nº 204

Santa Maria, RS, Brasil

2005

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada, aprova o Trabalho de Graduação

JRastro: Ferramenta para o rastreamento de programas Java paralelos e distribuídos usando JVMTI

elaborado por

Geovani Ricardo Wiedenhoft

como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Comissão Examinadora:

Prof. Dr. Benhur de Oliveira Stein
(Orientador)

Prof[a]. Dr[a]. Márcia Pasin

Prof[a]. Dr[a]. Andrea Schwertner Charão

Santa Maria, 20 de Dezembro de 2005.

Aquele que pergunta, pode ser um tolo por cinco minutos. Aquele que deixa de perguntar, será um tolo para o resto da vida.

(Provérbio Chinês)

A sabedoria da vida não está em só se fazer aquilo que se gosta, mas, também, de se gostar de tudo aquilo que se faz!

(Leonardo Da Vinci)

Algo só é impossível até que alguém duvide e acabe provando o contrário.

(Albert Einstein)

À minha mãe, aos meus irmãos e à todos que colaboraram de alguma forma na realização deste trabalho.

Agradecimentos

Agradeço inicialmente à minha mãe Iria, por sempre estar comigo nos momentos em que precisei e preciso, também por ter me proporcionado todas as oportunidades que me levaram a esta conquista. Enfim meu muito obrigado à essa pessoa que eu prezo muito. Agradeço também aos meus irmãos Gilnei e Gilmar por terem me ajudado financeiramente e emocionalmente com palavras de apoio e incentivo.

Agradeço ao meu professor e orientador Benhur de Oliveira Stein, pela dedicação, disponibilidade e boa vontade em ajudar, tanto na realização deste trabalho, como em resolver qualquer outro problema existente.

Um agradecimento em especial à Ani Carla Marchesan, por estar comigo e entender os momentos difíceis que tive durante a realização deste trabalho.

À todos os colegas e integrantes do Laboratório de Sistemas da Computação, pelas conversas e idéias construtivas que tivemos durante o curso, e pelos trabalhos realizados pelo famoso grupo LSC. À secretária do curso Marinelma Aimi de Carvalho, pela paciência e disposição em ajudar a resolver os problemas dos formulários, documentos e papéis necessários ao meu currículo e em geral entregues atrasados.

Ao sistema de buscas Google, pelas diversas ajudas com informações e serviços que facilitaram a vida, na solução de problemas de aulas, trabalhos e até mesmo na solução de provas.

Por fim, agradeço à todos que colaboraram de alguma forma para a conclusão deste trabalho, independentemente de como o fizeram.

Sumário

Lista de Tabelas **viii**

Lista de Figuras **ix**

Resumo **x**

1 Introdução **1**

2 Depuração de programas **3**

2.1 Monitoramento de programas 4

2.2 Registro e recuperação das informações de monitoramento 5

2.2.1 libRastro 5

2.3 Visualização dos dados monitorados 7

3 Depuração de programas Java **9**

3.1 JVMTI 10

3.2 Ferramentas para a depuração de programas Java 14

4 Agente de rastreamento **16**

4.1 Arquitetura do agente 16

4.2 Características do agente de rastreamento 17

4.3 Eventos rastreados 20

4.3.1 Rastreamento de *threads* 20

4.3.2 Rastreamento de chamadas de métodos 22

4.3.2.1 Implementação usando a notificação de eventos . . . 24

4.3.2.2 Implementação usando BCI 25
4.3.2.3 Problema da ocorrência de exceções 27
4.3.3 Rastreamento de monitores 28
4.3.3.1 Métodos sincronizados 29
4.3.3.2 Operações "*wait*" e "*notify*" 31
4.3.4 Rastreamento do uso de memória 31
4.3.4.1 Eventos do coletor de lixo 32
4.3.4.2 Eventos de alocação e liberação de memória 32
4.3.5 Rastreamento de comunicações RMI 34
4.4 Tipos de visualização 36
4.4.1 Visualização por *threads* 36
4.4.2 Visualização por objetos 37
4.4.3 Visualização por objetos classificados pelas classes 37

5 Avaliação 39
5.1 Ambiente de execução 39
5.2 Testes realizados 40
5.3 Resultados obtidos 40

6 Conclusão 43

Referências Bibliográficas 45

A Manual de instalação e utilização da *JRastro* 48
A.1 Requisito 48
A.2 Instalação 48
A.3 Utilização 49
A.4 Eventos e funções da JVMTI 50

Lista de Tabelas

5.1 Resultados obtidos nos diferentes testes realizados. 41

A.1 Eventos disponibilizados pela JVMTI. 51

A.2 Classificação das funções da JVMTI. 51

Lista de Figuras

2.1 Fases para a visualização do comportamento de programas 3

2.2 Janela do Pajé com os objetos visualizáveis 8

3.1 Agente de Rastreamento controla quais eventos serão monitorados pela JVMTI. 11

3.2 BCI na chamada e retorno de métodos. 12

4.1 Passos para o rastreamento, conversão e visualização de um programa. 17

4.2 Arquivo exemplo de seleção dos eventos *Method e Monitor*. 19

4.3 Rastreamento de *threads*. 21

4.4 Rastreamento de chamadas de métodos 23

4.5 Visualização de chamadas/retornos de métodos e exceções no Pajé . . 28

4.6 Rastreamento de monitores 29

4.7 Visualização de monitores no Pajé 30

4.8 Rastreamento da alocação e liberação de memória 32

4.9 Visualização dos eventos da memória 34

4.10 Rastreamento de comunicações RMI. 35

4.11 Visualização por *thread* 37

4.12 Visualização por objeto 38

4.13 Objetos agrupados pelas suas classes 38

RESUMO

Trabalho de Graduação
Ciência da Computação
Universidade Federal de Santa Maria

JRastro: Ferramenta para o rastreamento de programas Java paralelos e distribuídos usando JVMTI

Autor: Geovani Ricardo Wiedenhoft

Orientador: Prof. Dr. Benhur de Oliveira Stein

Data e Local da Defesa: 20 de Dezembro de 2005, Santa Maria.

As ferramentas de visualização de programas auxiliam na análise do comportamento de programas para a identificação de problemas de desempenho ou de lógica nesses programas. Entretanto, essas ferramentas necessitam que os programas a serem visualizados registrem informações sobre os principais eventos ocorridos durante suas execuções, em arquivos chamados rastros de execução.

Este trabalho descreve o desenvolvimento de um agente de rastreamento que registra rastros de execução de programas escritos na linguagem Java. Esse agente tem como características ser transparente ao desenvolvedor, não precisar da modificação da máquina virtual Java (JVM) e também, ser configurável.

Na implementação do agente, foi utilizado o módulo JVMTI (*Java Virtual Machine Tool Interface*) disponibilizado para o monitoramento dos eventos gerados pela JVM e uma biblioteca de registro e recuperação de informações, chamada *libRastro*, desenvolvida no Laboratório de Sistemas da Computação (LSC) na Universidade Federal de Santa Maria. Foram também implementados conversores para três tipos de visualização na ferramenta Pajé, desenvolvida no LSC. Com isso, pode-se utilizar essa ferramenta de forma satisfatória para a depuração de programas paralelos e distribuídos, através do rastreamento de diversos eventos gerados no programa.

Capítulo 1

Introdução

A utilização da linguagem Java no desenvolvimento de aplicações paralelas e distribuídas de alto desempenho é crescente. Isso ocorre devido à linguagem ser orientada a objetos, ser portável e possuir suporte à programação paralela e distribuída. Esse suporte é disponibilizado por *threads*, mecanismos de sincronização, comunicação via rede através de *Sockets* e RMI (*Remote Method Invocation*) [Sun 99], entre outros.

Conseqüentemente, nesse contexto de programação paralela e distribuída surge também a necessidade de meios para a depuração do código Java desenvolvido. As ferramentas tradicionais de depuração de aplicações seqüenciais são em geral insuficientes. Entretanto, essa tarefa pode ser auxiliada quando se dispõe de ferramentas que permitem a visualização do comportamento do programa. Essas ferramentas necessitam que seja realizado o rastreamento dos principais eventos ocorridos durante a execução do programa, chamados rastros de execução, para posteriormente serem apresentados de forma gráfica. Os registros dos eventos podem ser realizados com chamadas a bibliotecas que gravem as informações dos eventos passados, ou com ferramentas que façam essas tarefas de forma transparente ao desenvolvedor.

Nesse sentido, este trabalho descreve uma ferramenta chamada JRastro que permite o rastreamento de programas paralelos e distribuídos implementados na linguagem de programação Java para posterior visualização. A característica dessa ferramenta é de ser transparente ao desenvolvedor da aplicação, pois não necessita alteração do código desenvolvido e de modificação da máquina virtual Java (JVM).

Além dessa característica, a ferramenta também é configurável através da possibilidade de seleção dos eventos, classes e métodos a serem monitorados. Para isso, essa ferramenta é baseada em uma interface de monitoramento de eventos gerados pela JVM, chamada *Java Virtual Machine Tool Interface*.

No próximo capítulo, descreve-se a depuração de programas através da visualização, com o monitoramento e registro das informações do comportamento do programa durante a execução. Além disso, apresenta-se como a depuração através da visualização pode ocorrer em programas desenvolvidos na linguagem de programação Java. No capítulo seguinte, discute-se com mais detalhes as características e arquitetura do agente de rastreamento de programas Java paralelos e distribuídos desenvolvido neste trabalho, com os eventos rastreados e os tipos visualizações possíveis. Após é realizado uma avaliação desse agente e concluindo com as contribuições desse trabalho e possíveis trabalhos futuros.

Capítulo 2

Depuração de programas

A depuração de programas é uma etapa importante no processo de desenvolvimento de aplicações. Essa etapa tem como objetivo detectar e corrigir possíveis erros lógicos no programa. No contexto de programas paralelos e distribuídos, além desse objetivo, também procura-se fazer uma análise do comportamento do programa para redução de gargalos, otimização dos algoritmos ou até mesmo melhor distribuição de carga. Isso possibilita a obtenção de melhor desempenho na execução de tarefas.

A maioria dos depuradores paralelos e distribuídos apenas estendem as funcionalidades dos depuradores seqüencias para vários processos. Dessa forma, esses depuradores possuem dificuldades na apresentação de dados mais globais da execução e da evolução do programa no tempo, como por exemplo as comunicações realizadas entre os processos e a concorrência entre eles. Para complementar essa deficiência existem as ferramentas de visualização. Entretanto, a visualização do comportamento de um programa somente ocorre com a realização de certas fases ilustradas na figura 2.1.

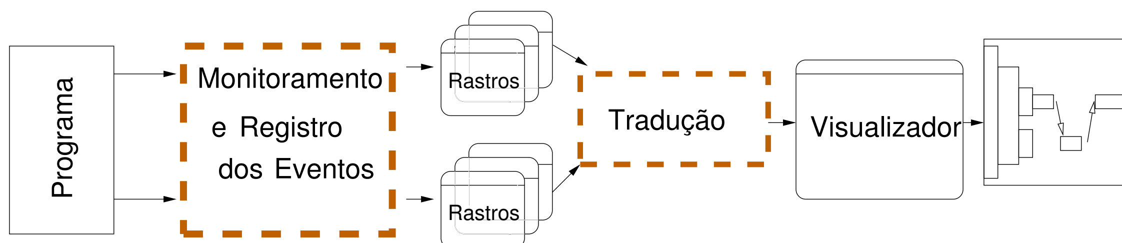

Figura 2.1: Fases para a visualização do comportamento de programas

Para visualizar o comportamento de programas, primeiramente deve-se monitorar e registrar os principais eventos ocorridos durante a execução. Os eventos são ações que ocorrem durante a execução do programa, como por exemplo a chamada e retorno de métodos. Após a realização do registro desses eventos é necessário traduzi-los ao formato do visualizador a ser utilizado, com objetivo final de visualização.

Nas seções seguintes, essas fases e as ferramentas utilizadas são descritas com maiores detalhes.

2.1 Monitoramento de programas

O monitoramento se faz necessário quando se deseja obter informações da execução de um programa. As duas técnicas de monitoramento mais comumentes utilizadas são o monitoramento com a modificação do código e o monitoramento transparente.

Na primeira técnica altera-se o código fonte em pontos específicos, inserindo chamadas a funções de registro. Um exemplo de uso dessa técnica é a adição no programa de chamadas a funções antes do envio e da recepção de mensagens em programas que usam essa forma de comunicação, para a posterior visualização das comunicações realizadas pelos módulos do programa. Essa técnica exige a alteração do código fonte sempre que for necessário o monitoramento, podendo ser diferente a cada tipo de programa.

Na segunda técnica, o monitoramento é realizado de forma transparente ao programador. Essa técnica não necessita alteração do código, o que facilita o monitoramento de diferentes programas. Esse monitoramento é feito através de bibliotecas que interceptam os eventos no programa.

2.2 Registro e recuperação das informações de monitoramento

Para que o comportamento de um programa seja visualizado, é necessário registrar os eventos gerados durante a sua execução. Esse registro dos eventos é realizado em arquivos chamados rastros de execução.

Nessa fase do registro deve-se ter o cuidado de não interferir excessivamente na execução do programa, para que, os dados obtidos correspondam a uma execução normal. Com esse objetivo deve-se registrar o mínimo de informações possíveis. A partir dessa compactação, é necessário fazer a recuperação das informações nos arquivos de rastros e traduzi-las do formato compactado para o formato da ferramenta de visualização utilizada.

A seguir descreve-se uma biblioteca de registro e recuperação dos dados utilizada neste trabalho, chamada *libRastro*.

2.2.1 libRastro

A *libRastro* [SIL 2004] é uma biblioteca de registro e recuperação de informações dos eventos monitorados. Essa biblioteca foi desenvolvida em linguagem C no Laboratório de Sistemas da Computação (LSC).

Os tipos de dados manuseados pela biblioteca foram padronizados para obterem o exato número de bits que ela utiliza, independente da arquitetura. Devido a isso, ela possui capacidade de rastrear dados de 8, 16, 32 e 64 bits bem como float e double. Além disso, a *libRastro* garante suporte a múltiplas *threads*[1], com a utilização da biblioteca *pthreads* como padrão, mas também fornece ao programador a possibilidade de utilização de outra biblioteca de *threads*.

Essa biblioteca permite registrar rastros com baixa intrusividade na execução do programa. Para isso, é necessária a utilização de funções que armazenam as informações em um formato próprio. Ela, além de possuir algumas funções básicas, também disponibiliza a criação de novas funções para o registro das informações

[1] *Threads* são múltiplos fluxos de execução em um mesmo processo. [TAN 97]

necessárias antes da compilação. As funções mais utilizadas são descritas abaixo:

- Funções básicas da biblioteca
  - rst_init(...) -> Inicializa a biblioteca
  - rst_finalize() -> Finaliza a biblioteca
  - rst_event(tipo) -> Registra a ocorrência de um evento de um tipo
- Funções geradas pela biblioteca
  - rst_event_* -> Possibilita a criação de outras funções
  - rst_event_il(tipo, int, long) -> Registra o tipo do evento, um inteiro e um long
  - rst_event_dfcws(...) -> Registra o tipo do evento, double, float, char, short e uma string

Em cada registro realizado pela *libRastro*, além das informações do evento, também é registrado o tempo em que o evento ocorreu. Esse registro do tempo tem como objetivo permitir a visualização posterior dos eventos armazenados em um diagrama tempo-espaço.

Nos programas paralelos e distribuídos que são executados em máquinas distintas, existe o problema da necessidade de sincronização dos relógios[2]. Para resolver esse problema, a biblioteca possui funções de sincronização. Essas funções deverão ser chamadas antes e depois da execução do programa rastreado para se obter os tempos de ajuste dos rastros.

Como a *libRastro* registra os rastros em um formato próprio, torna-se necessária a leitura e tradução dos rastros para o formato do visualizador. Para isso, a biblioteca possui funções de leitura de múltiplos arquivos de rastros.

Devido a todas essas características, a *libRastro* tornou-se uma biblioteca adequada às necessidades deste trabalho.

[2]Sincronização dos relógios é uma necessidade que ocorre quando um programa é executado em diferentes máquinas e, conseqüentemente, existem diferentes relógios. Dessa forma, como os tempos obtidos não são comparáveis diretamente, é necessário fazer um ajuste em relação a um tempo pré-determinado.

2.3 Visualização dos dados monitorados

As ferramentas de visualização são muito úteis no processo de depuração de programas, pois auxiliam o programador a compreender como foi a execução de um programa, a encontrar possíveis problemas de lógica e pontos para melhoramento de desempenho. A representação gráfica mais comum dessas ferramentas é a forma de diagrama tempo-espaço, que mostra o comportamento de um programa em um determinado tempo a partir de arquivos de rastros gerados anteriormente.

A ferramenta de visualização de programas paralelos e distribuídos utilizada neste trabalho é o visualizador Pajé [STE 99, STE 2000]. Essa ferramenta é baseada na visualização *post-mortem*[3]. Essa visualização ocorre com a geração dos rastros durante a execução do programa e conversão posterior ao formato próprio do Pajé. Pajé tem como característica ser adaptável a diferentes tipos de dados, pois a forma como os dados devem ser representados é descrita no próprio arquivo que contém os dados.

Os objetos que podem ser visualizados na ferramenta Pajé são *eventos, estados, variáveis, links e containers.* Esses objetos são visualizados na figura 2.2 e descritos abaixo.

- Os *eventos* representam uma ação que ocorreu em um determinado tempo. Esses objetos são visualizados na forma de círculos.
- Os *estados* representam um estado em um intervalo de tempo. Esses são visualizados através de retângulos na horizontal, sendo que a esquerda do retângulo representa o início do estado e a direita mostra o fim desse estado.
- Os objetos *variáveis* representam os valores associados a uma variável. A visualização desses objetos é feita através de linhas relativas aos valores da variável.

[3] Visualização post-mortem é a visualização que ocorre após o termino da execução do programa, a partir de rastros gravados durante a execução.

- Os *links* representam uma ligação entre dois objetos, que pode representar uma comunicação entre eles. Esses objetos são visualizados através de setas.

- Os *containers* representam objetos que contem os quatro primeiros objetos citados. Na hierarquia é visualizado como um nó na árvore.

Essas características facilitam na conversão para visualização de diferentes formas, auxiliando na depuração mais aprimorada e com mais opções. Dessa forma, o Pajé permite que com os mesmos rastros gerados possam ser convertidos para visualizações diferentes.

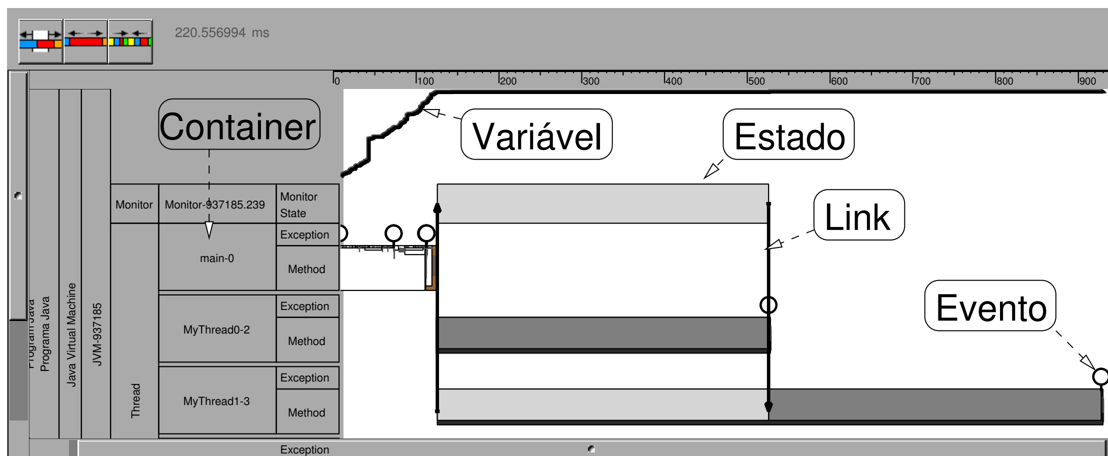

Figura 2.2: Janela do Pajé com os objetos visualizáveis

Capítulo 3

Depuração de programas Java

Java é uma linguagem de programação orientada a objetos desenvolvida pela Sun Microsystems que tem como umas das principais características a portabilidade. Essa característica permite que aplicações em Java possam ser executadas em qualquer máquina, independente de arquitetura ou sistema operacional. A portabilidade é disponibilizada porque, durante a compilação, ocorre a geração de uma linguagem intermediária chamada de *bytecodes*, ao invés de uma linguagem específica da arquitetura (linguagem de máquina). A partir disso, a execução é realizada através da interpretação desses *bytecodes* para a linguagem de máquina específica, com a utilização de uma JVM (Máquina Virtual Java) específica a cada tipo de arquitetura e sistema operacional.

Além disso, um dos grandes motivos da popularidade atual do Java é a disponibilidade gratuita na Internet de um ambiente de desenvolvimento chamado JDK (*Java Development Kit*). Esse ambiente possui a JVM, ferramentas para a compilação, depuração de programa Java e documentação. JDK também disponibiliza classes e pacotes básicos e complexos para o desenvolvimento de aplicações, como por exemplo acessos a bancos de dados, interfaces gráficas, entre outras.

O ambiente de desenvolvimento JDK oferece também um módulo de interface com a JVM. Esse módulo permite fazer o monitoramento e a recuperação de eventos gerados pela JVM. O monitoramento realizado pelo módulo não necessita de instrumentação do código ou mesmo da JVM, pois é baseada na notificação de eventos. Além disso, esse módulo é configurável, pois permite a seleção dos eventos a serem

monitorados, para assim diminuir a interferência na execução do programa.

O módulo de interface com a JVM teve uma primeira implementação chamada JVMPI (*Java Virtual Machine Profiler Interface*) [Sun 2004] que foi experimental. A JVMPI foi utilizada em uma implementação anterior deste trabalho desenvolvido no LSC (Laboratório de Sistemas da Computação) [SIL 2003, SIL 2002].

Atualmente a JVMPI está em fase de obsolescência, e deixará de acompanhar as próximas versões do Java, sendo substituída por uma segunda versão chamada JVMTI [Kel 2004].

A próxima seção deste trabalho descreve a JVMTI e as diferenças entre as duas versões (seção 3.1). Conclui-se o capítulo com as ferramentas para a depuração de programas Java existentes (seção 3.2).

3.1 JVMTI

O monitoramento se faz necessário quando se deseja visualizar o comportamento de um determinado programa. Para isso, o ambiente de desenvolvimento *JDK* do Java possui, a partir da versão 1.5 (disponível desde setembro de 2004), um módulo de monitoramento e depuração chamado JVMTI (*Java Virtual Machine Tool Interface*) [Sun 2004a]. A JVMTI permite:

- habilitar os eventos de interesse e registrar as funções responsáveis por tratá-los. Como exemplos desses eventos, pode-se citar a criação e destruição de fluxos de execução, chamadas e retornos de métodos, alocação e liberação de memória, entre outros ocorridos durante a execução do programa sendo rastreado.

- chamar as funções de tratamento quando os eventos acontecem na JVM (*callbacks*).

- disponibilizar funções de acessos a informações complementares da JVM, que podem ser usadas pelas funções de tratamento dos eventos.

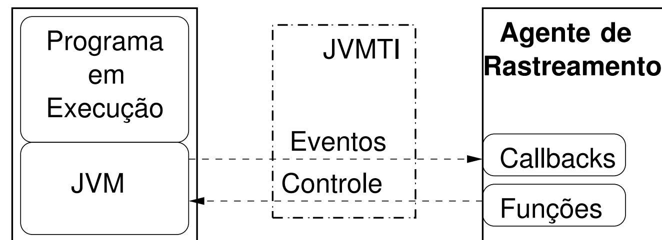

Figura 3.1: Agente de Rastreamento controla quais eventos serão monitorados pela JVMTI.

Conforme a figura 3.1, a JVMTI intercepta os eventos gerados pela JVM durante a execução do programa, e notifica um cliente JVMTI, chamado agente de rastreamento[1], que o evento ocorreu, através de funções (*callbacks*) pré-estabelecidas, e assim realizar o tratamento do evento. Além disso, JVMTI oferece funções para o agente obter informações da JVM e para fazer o controle do monitoramento, como por exemplo, selecionar os eventos a serem monitorados e habilitar as funções que irão tratar esses eventos.

A JVMTI difere da antiga interface JVMPI por possuir um conjunto maior de funções e notificações de eventos. Essas funções podem obter informações sobre variáveis, campos, métodos e classes. Adiciona-se a isso, as funções da JVMTI fazem uso dos objetos JNI (Java Native Interface) [LIA 99], referência aos objetos, enquanto que, na JVMPI é utilizado identificadores dos objetos (IDs).

A JVMTI também possui uma forma diferente de notificar e selecionar os eventos que serão monitorados, o que a torna mais eficiente que a predecessora. Na JVMPI, a JVM chama a mesma função do agente sempre que um evento deve ser reportado, independentemente do tipo de evento gerado. Essa função deve então selecionar a ação a ser tomada dependendo do valor de um argumento que representa o tipo de evento reportado. No caso da JVMTI, é possível registrar uma função exclusiva a cada tipo de evento. Isso possibilita, por exemplo, que a criação de *threads* tenha uma função de tratamento diferente da chamada de métodos. Essas funções, então, não necessitam de código extra para selecionar o tratamento do

[1] Agente de rastreamento é um cliente JVMTI que recebe as notificações de ocorrência de eventos e faz o registro das informações desses eventos. Além disso, esse agente também faz o controle da JVMTI e pode recuperar informações da JVM.

evento.

Uma importante característica da JVMTI é que ela possui suporte a BCI (*Bytecode Instrumentation*) [COH 2004]. BCI é a técnica destinada a modificação direta do código objeto intermediário (*bytecodes*). Com essa técnica é possível modificar instruções no código objeto (*.class*) em pontos específicos, alterando o comportamento de métodos do programa do usuário. Essa técnica pode ser utilizada com objetivo de alterar o início e o final de um método, adicionando código de tratamento, que realiza ações de registro da chamada e do retorno desse método, como ilustra a figura 3.2.

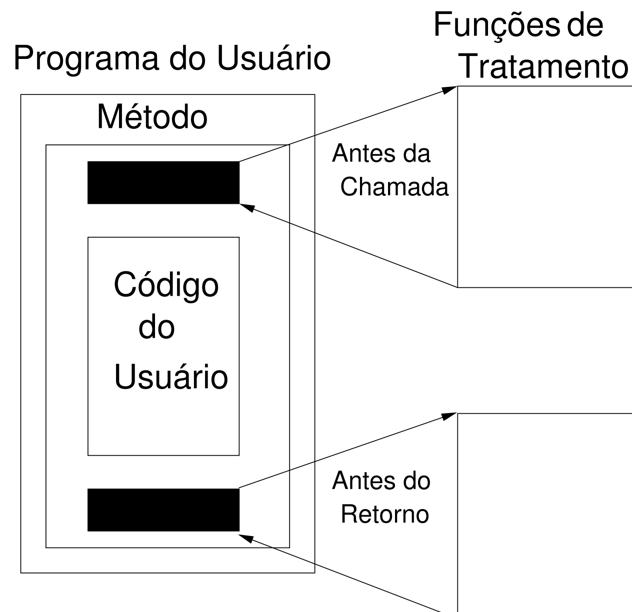

Figura 3.2: BCI na chamada e retorno de métodos.

A instrumentação dos *bytecodes* pode ser realizada em três momentos diferentes:

- Instrumentação estática: a instrumentação dos arquivos "*.class*" é realizada antes do momento da execução. Essa opção não é muito útil na instrumentação com a JVMTI.

- Instrumentação antes do momento da carga: a instrumentação é realizada antes da carga das classes para a JVM. Para isso, a JVMTI notifica o agente

através do evento `JVMTI_EVENT_CLASS_FILE_LOAD_HOOK` que uma determinada classe será carregada, possibilitando a modificação dessa classe.

- Instrumentação dinâmica: a instrumentação pode ser realizada em qualquer momento após a classe ter sido carregada, mesmo ela tendo sido executada. Essa opção é possível através da função JVMTI *RedefineClasses*. Com isso, classes podem ser modificadas múltiplas vezes e também podem ser retornadas aos seus estados originais.

Dessa forma, através dos eventos `JVMTI_EVENT_CLASS_FILE_LOAD_HOOK` e das funções *RedefineClasses*, a JVMTI permite a realização de BCI, porém, não é realizada a instrumentação propriamente dita. Para isso, uma biblioteca dinâmica demonstrativa é oferecida, chamada *java_ crw_ demo* (*Java Class Read-Write Demo*).

A biblioteca *java_ crw_ demo* oferece funções básicas para manipulação de *bytecodes* que realizam inserções de chamadas a métodos estáticos de classes indicadas. Esses métodos, por sua vez, chamam funções em códigos nativos (JNI) de tratamento. As funções de manipulação de *bytecodes* permitem a alteração do código objeto em quatro momentos diferentes.

- No momento da chamada dos métodos de uma determinada classe.
- No momento do retorno dos métodos da classe
- Na criação de objetos Java, que ocorre através da modificação do método *init* com descrição `()V` da classe `java/lang/Object`, pois todos objetos que são criados passam por esse método.
- Imediatamente antes de qualquer criação de *array*. Com isso, pode-se capturar o momento da criação de um objeto *array*.

Com a utilização de BCI, a espera pela ocorrência desses eventos não é mais necessária, pois a função de registro será inserida diretamente no código do método da classe monitorada. Além disso, a seleção de quais métodos devem ser monitorados

pode ser feita inserindo-se códigos somente nesses métodos, ao invés de realizar a verificação da necessidade de monitoramento a cada interceptação de evento.

Dessa forma, é mais vantajosa a utilização de *callbacks* ou o uso de BCI do que colocar chamadas a bibliotecas diretamente no código do usuário. Uma dessas vantagens é a portabilidade oferecida pelo Java, pois a JVMTI independe de arquitetura ou sistema operacional. Outra vantagem do uso de uma biblioteca dinâmica que utiliza JVMTI é que a depuração independe do programa Java que se deseja rastrear. Isso porque, não é mais necessária a modificação manual do código Java do usuário, aumentando a transparência da ferramenta de rastreamento.

3.2 Ferramentas para a depuração de programas Java

A depuração de programas Java pode ser realizada com a visualização do comportamento da execução desses programas. Na literatura, existem várias ferramentas que fazem o monitoramento e registro para posterior visualização. Dentre as ferramentas livres existentes pode-se citar o *HProf* [Kel 2004a], *NetBeans* [Sun 2004b], *Eclipse* [The 2004], *JProf*, entre outras. Como alternativas comerciais cita-se o *JBuilder* [BOR 2004], *Borland OptimizeIt Profiler* [BOR 2004a], *VisualCafé*, entre outras.

Entretanto, a maioria dessas ferramentas possui certas dificuldades indesejadas. Um exemplo disso é que grande parte das ferramentas não atendem às novas necessidades dos programas paralelos e/ou distribuídos, como o monitoramento das comunicações, tornando-se inadequadas a esses programas. Outra dificuldade dessas ferramentas é a de serem específicas a uma determinada aplicação. Ou então, de gerarem rastros específicos a uma determinada ferramenta de visualização, sendo que a conversão dos rastros para outra ferramenta um grande problema. Também existem as ferramentas que necessitam da interferência do programador para a modificação do código do programa ou da máquina virtual.

Entre as ferramentas que fazem o rastreamento de programas Java, pode-se

descrever a ferramenta *HProf*. A *HProf* monitora e coleta informações do comportamento durante a execução, através do uso da JVMTI. Ela se destaca pelo conteúdo e quantidade de informações resultantes e pela sua grande distribuição, pois está incluída no ambiente de desenvolvimento JDK. Essa ferramenta tem uma grande aplicabilidade na depuração de programas seqüenciais. Entretanto não atinge tal eficiência em programas paralelos e/ou distribuídos, pois não prevê o monitoramento das comunicações. Além disso, o *HProf* tem como saída um arquivo texto de difícil compreensão, pois seus dados não são organizados convenientemente para a análise, tanto pelo caráter textual quanto pela sua distribuição no texto.

Outra importante ferramenta a ser ressaltada é a versão anterior deste trabalho [SIL 2003]. Essa ferramenta tem como características a transparência da geração dos rastros ao desenvolvedor da aplicação e a não instrumentação da máquina virtual Java. Entretanto, ela utilizou a JVMPI para o monitoramento de eventos gerados pela JVM. Dessa forma, essa versão tornou-se obsoleta, pois a JVMPI foi experimental e deixará de acompanhar as próximas versões do Java, sendo substituída pela JVMTI, utilizada na versão desenvolvida neste trabalho. Além disso, este trabalho diferencia-se da verão anterior por possuir uma maior configurabilidade, pois permite ao usuário selecionar quais eventos, métodos e classes que serão rastreados, por utilizar a instrumentação de *bytecodes* para o rastreamento de diversos eventos, e entre outras características.

Dessa forma, uma ferramenta de rastreamento de programas Java paralelos e distribuídos foi desenvolvida neste trabalho para resolver as deficiências das ferramentas existentes.

Capítulo 4

Agente de rastreamento

A depuração de programas através da visualização necessita da geração de rastros durante a execução do programa para posteriormente serem convertidos ao formato do visualizador. Para isso, este trabalho desenvolveu um agente de rastreamento, conceituado no capítulo anterior, como sendo um cliente JVMTI que faz o monitoramento e registro dos eventos gerados pela JVM.

A próxima seção descreve a arquitetura do agente de rastreamento desenvolvido com suas características. Após, são apresentados diversos tipos de eventos que o agente monitora. Conclui-se o capítulo com a apresentação dos conversores de arquivos de rastros implementados.

4.1 Arquitetura do agente

O agente desenvolvido neste trabalho realizada o monitoramento e registro das informações dos principais eventos ocorridos na JVM durante a execução do programa cuja execução se deseja visualizar.

A figura 4.1 ilustra o rastreamento passo a passo utilizando o agente desenvolvido e as respectivas ferramentas utilizadas até o comportamento do programa a ser visualizado.

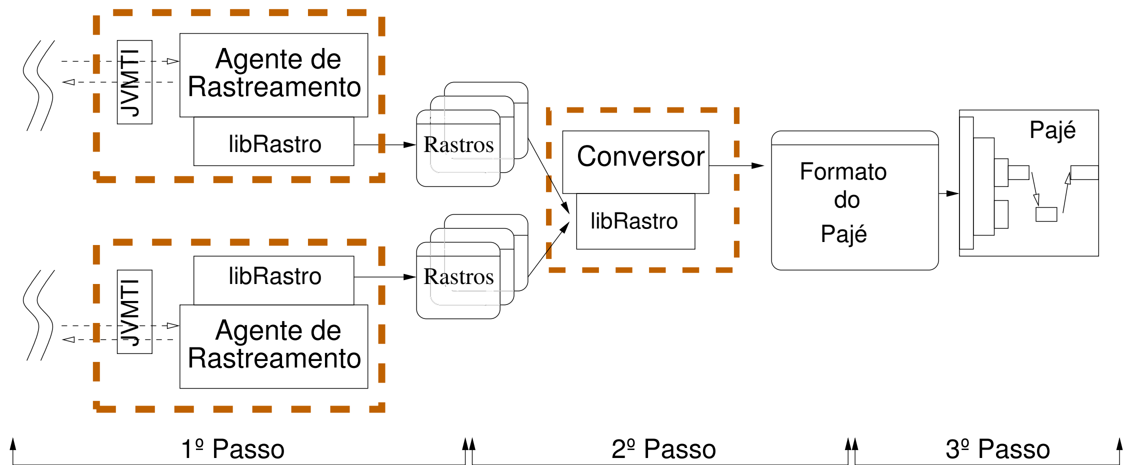

Figura 4.1: Passos para o rastreamento, conversão e visualização de um programa.

O primeiro passo é o monitoramento da execução do programa pelo agente de rastreamento desenvolvido. O agente utiliza e controla o módulo JVMTI. Esse módulo intercepta os eventos da JVM e os envia ao agente através de funções pré-estabelecidas. Os eventos são habilitados para o monitoramento pelo agente com chamadas de controle a funções do módulo. Entretanto, o monitoramento apenas notifica as ocorrências dos eventos, sendo necessário registrar as informações desses eventos em arquivos de rastros. Para o registro foi escolhida a biblioteca *libRastro*, devido às características citadas no capítulo 2.

O segundo passo é necessário porque os rastros gerados no primeiro momento estão em um formato próprio da biblioteca *libRastro*. Dessa forma, após a geração dos rastros de execução é necessário convertê-los, através de conversores implementados com funções de leitura da própria *libRastro*, ao formato do visualizador Pajé, ferramenta escolhida neste trabalho.

O terceiro passo é a visualização propriamente dita, na ferramenta Pajé. Nesse passo, os rastros se encontram no formato dessa ferramenta.

4.2 Características do agente de rastreamento

O rastreamento realizado pelo agente tem como características ser transparente ao desenvolvedor e não necessitar da instrumentação da JVM. Além disso, este

agente é configurável.

Como conseqüência do uso do módulo de interface com a JVM, o agente de rastreamento é transparente e não necessita de instrumentação da JVM. Através desse módulo, o agente foi projetado para ser uma biblioteca dinâmica que é passada como parâmetro para a JVM na execução de programas Java com objetivo de rastreamento desses programas.

A configurabilidade do agente permite ao usuário selecionar quais eventos, métodos e classes que serão rastreados. A seleção pode ser realizada em tempo de conversão dos rastros e no tempo da própria visualização. Entretanto, a possibilidade da escolha em tempo de geração dos rastros é uma boa alternativa para a diminuição da intrusão do agente de rastreamento na execução do programa. Assim, através dessa seleção o agente permite ao usuário informar o quanto de informações precisa para a depuração do programa com o objetivo de controlar a interferência na execução.

A configuração é realizada através de dois arquivos que são passados como parâmetro para o agente no momento do lançamento da execução da aplicação Java:

- O primeiro arquivo contém a *seleção dos eventos a serem monitorados.* Através desse arquivo, o agente habilita a JVMTI fazer a notificação da ocorrência dos eventos selecionados. Esse arquivo pode conter:
  - O comando *NoTraces* indica que as informações monitoradas não serão registradas em arquivos.
  - O comando *Method* seleciona o monitoramento de chamadas e retornos de métodos.
  - O comando *Monitor* seleciona o monitoramento do bloqueio e liberação de monitores.
  - O comando *MemoryAllocation* indica o monitoramento da alocação e liberação de memória na execução do programa, e também, do monitoramento da execução do coletor de lixo.

  - O caracter "#" comenta uma linha no arquivo de configuração. Dessa forma, o evento comentado não será selecionado.

A figura 4.2 ilustra um exemplo de arquivo com a seleção dos eventos de chamada e retorno de métodos, e do bloqueio e liberação de monitores.

```
#NoTraces
Method
Monitor
#MemoryAllocation
```

Figura 4.2: Arquivo exemplo de seleção dos eventos *Method e Monitor*.

- O segundo arquivo contém *as classes e os métodos a serem monitorados*. Esse arquivo possui um formato próprio, sendo que, o caractere "C" indica que a próxima palavra é o nome da classe a ser monitorada e as linhas na seqüencia com o caractere "M" representam os métodos dessa classe. Já o caractere "*" pode representar todas as classes ou todos os métodos, dependendo do caso utilizado. A partir disso, esse arquivo pode conter quatro possibilidades de seleções diferentes, que são:

  - A seleção de todos os métodos de todas as classes para o monitoramento. Isso através da opção:

    C * M *

  - A seleção de todos os métodos de uma classe específica. Por exemplo, com a seleção de todos os métodos da classe *MinhaClasse*, através da linha:

    C MinhaClasse M *

  - A seleção de alguns métodos específicos de todas as classes. Por exemplo, com a seleção dos métodos *entrar e correr* de todas as classes, através da opção:

C * M entrar

M correr

– A seleção de alguns métodos específicos de uma classe específica. Por exemplo, com a seleção dos métodos *entrar, sair e correr* da classe *MinhaClasse*, através da opção:

C MinhaClasse M entrar

M sair

M correr

4.3 Eventos rastreados

Os eventos que podem ser rastreados através da JVMTI são muito numerosos. Desse modo, o agente desenvolvido permite ao usuário selecionar quais eventos são necessários para visualização. As seções seguintes descrevem os eventos registráveis por JRastro e as técnicas utilizadas.

4.3.1 Rastreamento de *threads*

A utilização do recurso de *threads* no desenvolvimento de programas de alto desempenho é freqüente, permitindo explorar o nicho de máquinas multiprocessadas e/ou a concorrência interna de partes do programa. A visualização das *threads* auxilia na depuração do código, pelos seguintes motivos: permite a observação da ordem de execução delas, qual o tempo em que cada uma demorou na execução de um determinado método, a competição para a entrada em regiões críticas (protegidas por monitores) e também, o acesso a objetos compartilhados.

A visualização esperada no Pajé é apresentada na figura 4.3. Nessa representação, é ilustrado o momento de criação (1) e destruição (2) das *threads* do programa. Além disso, é observado o momento de lançamento de uma nova *thread* com o uso de uma seta indicando esse disparo (3), o início da sincronização realizada através da chamada do método *join* da classe `java/lang/Thread` pela *thread* criadora (4) e a sincronização propriamente dita com a seta de retorno da *thread* criada, finalizando

o método *join* (5).

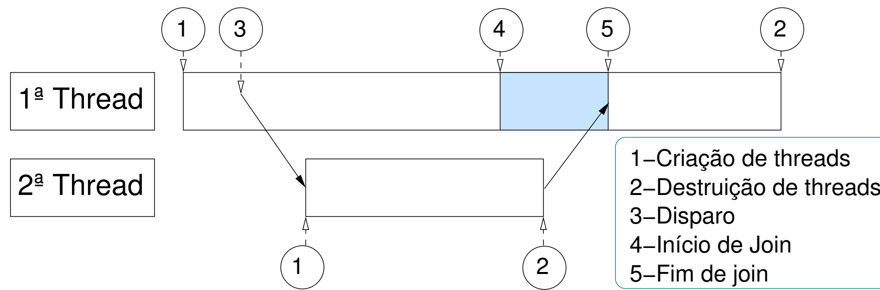

Figura 4.3: Rastreamento de *threads*.

Com objetivo de visualizar essa representação no Pajé, é necessário monitorar e registrar informações nos momentos da criação e destruição das *threads*. Essas informações registradas devem conter o identificador da *thread* e da JVM a que ela pertence, bem como o seu respectivo nome e outros dados. Além disso, a visualização do lançamento necessita que seja monitorado o momento do disparo da nova *thread* (a execução do método *start* da classe `java/lang/Thread`) para registrar os identificadores das *threads* criadora e criada, com o objetivo de visualizar a origem da seta. Para o destino dessa seta, basta registrar no momento da criação da nova *thread* os identificadores da *thread* criadora e da *thread* criada. A sincronização é realizada com o registro dos identificadores das *threads* criada e criadora (origem da seta) no momento da finalização da *thread* criada e também, do registro desses identificadores na execução do método *join* (destino da seta).

Para a criação e destruição de *threads*, a JVMTI oferece a possibilidade de notificação de dois eventos principais, `JVMTI_EVENT_THREAD_START` e `JVMTI_EVENT_THREAD_END`. O evento `JVMTI_EVENT_THREAD_START` inicializa a biblioteca de registro e armazena as informações necessárias à visualização da *thread* criada. O evento `JVMTI_EVENT_THREAD_END` finaliza a biblioteca de registro, e força a gravação do *buffer* no arquivo de rastro. Porém, esse último evento pode não ocorrer, porque a *thread* é um objeto, então, está sujeita a ser varrida da memória pelo coletor de lixo quando não estiver sendo mais usada. Dessa forma, é possível que não seja gerado o evento de finalização, conseqüentemente, não sendo gravadas as

informações do *buffer* no arquivo de rastro. A solução encontrada é monitorar o evento `Agent_OnUnload`[1] e invocar a função `rst_flush_all` da *libRastro*, que força a gravação de todos os *buffers* existentes em seus respectivos arquivos de rastros.

Para as visualizações do lançamento e da sincronização, a JVMTI não possui eventos específicos. A possibilidade é o monitoramento direto dos métodos *start* e *join* da classe `java/lang/Thread`. Entretanto, o problema é que no método *start* falta a identificação de qual a *thread* será criada e no método *join* falta a identificação de qual *thread* que a criou. Dessa forma, não foi possível encontrar uma solução para esse problema, não se obtendo a visualização das setas de sincronização.

Outro problema é que a JVMTI não dispõe de identificadores para as *threads*, mas a referência JNI a elas. Sendo assim, foi necessário gerar um identificador próprio e único a essas *threads*.

A biblioteca de registro *libRastro* usa a abordagem de um *buffer* a cada *thread*, em vez de apenas um *buffer* para todo o processo. A partir disso, um dos problemas no registro dos eventos interceptados foi o gerenciamento manual do *buffer*, devido à necessidade de utilização de outra biblioteca de *threads* e não a *pthreads* suportada nativamente pela *libRastro*. A solução encontrada foi utilizar funções da JVMTI que estabelecem regiões de memória específicas a cada *thread*. Dessa forma, na criação de *thread* é necessário alocar, inicializar e indicar à JVMTI o *buffer* privado a essa *thread*, através da função *SetThreadLocalStorage*. Para o registro de outros eventos no *buffer* da *thread*, a recuperação é feita através da função *GetThreadLocalStorage*.

4.3.2 Rastreamento de chamadas de métodos

Os eventos importantes a serem rastreados são as chamadas e retornos de métodos. Com a visualização desses eventos é possível verificar o que a *thread* está executando em um período e quanto tempo durou essa execução. Além disso, é possível detectar concorrências entre os objetos, quais objetos estão sendo mais utilizados e também, identificar erros ou mesmo pontos para melhora de desempenho.

[1] O evento `Agent_OnUnload` é gerado no momento que antecede a liberação da biblioteca desenvolvida da memória.

A figura 4.4 ilustra como pretende-se visualizar a chamada e o retorno de métodos de uma *thread* no Pajé. Nessa visualização são apresentados quais métodos uma determinada *thread* invoca, sendo representados pelo início de cada retângulo (1) e o retorno dos métodos através do fim desses mesmos retângulos (2). Além disso, é apresentado o método chamador (3) e o método chamado (4).

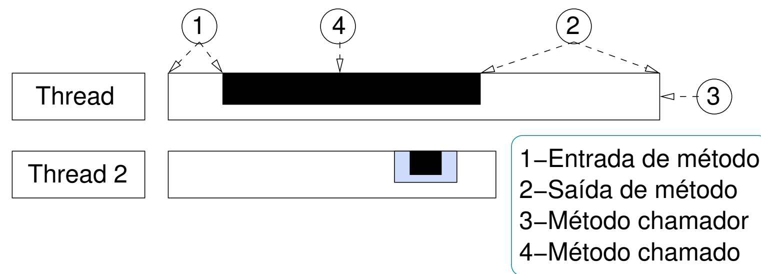

Figura 4.4: Rastreamento de chamadas de métodos

Para essa visualização no Pajé, é necessário monitorar os momentos das chamadas e dos retornos de métodos. Através desse monitoramento, registra-se as informações do método, como por exemplo, o registro do seu nome, para distinguir na visualização de qual método está sendo executado, e o identificador do objeto ao qual o método pertence. Esse registro é feito no arquivo de rastro correspondente à *thread* que está executando o método. Uma alternativa para melhorar o desempenho do agente é registrar nesses eventos o identificador do método (um inteiro) ao invés de registrar o nome em todos os eventos. Para isso, detecta-se o momento que o método está sendo carregado e registra-se o identificador e o nome desse método, e também o identificador da classe à qual esse método pertence. Desse modo, após os rastros terem sido gerados, o conversor terá as informações necessárias para a conversão dos eventos de chamada e retorno de métodos.

A JVMTI possui duas diferentes possibilidades de rastreamento de chamadas e retornos de métodos, que são através da notificação de eventos e através da instrumentação de *bytecodes*. Este trabalho, primeiramente, foi desenvolvido através da notificação de eventos (seção 4.3.2.1), mas foram encontrados problemas nessa implementação. A partir disso, foi escolhido a técnica da instrumentação de *bytecodes* (seção 4.3.2.2). Nas seções seguintes descrevem-se as duas versões desenvolvidas

neste trabalho e após apresentam-se o problema da ocorrência de exceções e a solução obtida (seção 4.3.2.3).

4.3.2.1 Implementação usando a notificação de eventos

Primeiramente foram utilizados dois eventos principais da JVMTI, que notificam a chamada e retorno de métodos, o `JVMTI_EVENT_METHOD_ENTRY` e o `JVMTI_EVENT_METHOD_EXIT`.

- O primeiro evento, `JVMTI_EVENT_METHOD_ENTRY`, é interceptado pelo agente na invocação de um método. Esse evento possui o identificador da *thread* e do método para o registro. Sendo assim, para não necessitar obter e registrar o nome desse método e o identificador da classe sempre que esse evento for gerado, foi utilizado o evento `JVMTI_EVENT_CLASS_PREPARE` para inicializar os métodos. Esse evento é gerado quando a preparação de uma classe está completa. Nesse ponto, os campos da classe, métodos e interfaces implementados estão disponíveis e nenhum código da classe foi executado. Dessa forma, nesse evento, através da classe, são obtidos os métodos disponíveis e registrados os seus respectivos nomes e identificadores e também o identificador da classe da qual os método pertencem.

- O segundo evento, `JVMTI_EVENT_METHOD_EXIT`, notifica ao agente o momento em que um método é retornado. Esse evento não necessita do registro de informações adicionais (apenas que o evento ocorreu), pois na conversão as informações do método retornado são obtidas através da análise do último método que foi chamado.

O problema com essa implementação foi que esses eventos não dispunham do identificador do objeto ao qual o método pertencia. Além disso, foi constatado o baixo desempenho do agente (281,67μs/evento). Isso deve-se ao fato do agente necessitar monitorar todos os eventos de chamada e retorno de métodos, e verificar se o método iria ser registrado a cada evento interceptado. Essa verificação é necessária,

porque o agente permite ao usuário selecionar (através do arquivo de configuração) quais métodos e classes deverão ser registrados.

4.3.2.2 Implementação usando BCI

A segunda versão foi implementada com a utilização da técnica de instrumentação dos *bytecodes*, BCI. Com essa técnica, não é necessário o monitoramento pela JVMTI dos eventos citados para a outra versão, pois com a instrumentação adiciona-se, em pontos específicos, chamadas a funções que irão tratar a ocorrência desses eventos (1,64μs/evento).

Como apresentado no capítulo anterior, a JVMTI notifica a ocorrência do evento `JVMTI_EVENT_CLASS_FILE_LOAD_HOOK` no momento que antecede a carga da classe para a memória, sendo que esse evento é gerado para todas as classes (inclusive para as classes antes da JVM ter sido inicializada). Através desse evento é permitido a instrumentação dos *bytecodes* de todos os métodos das classes ou a realização da seleção dos métodos das classes indicados pelo usuário. Entretanto, não é possível realizar a instrumentação para as classes a serem carregadas nos momentos que antecedem o fim da inicialização da JVM, pois essas classes são básicas da JVM (classes essenciais para a execução de um programa Java) e não estão na memória, apesar do evento ter sido gerado. A solução é armazenar informações dessas classes no evento `JVMTI_EVENT_CLASS_FILE_LOAD_HOOK` enquanto a JVM estiver inicializando até a notificação do evento `JVMTI_EVENT_VM_INIT`. Esse evento indica que a JVM completou a inicialização e, nesse momento, é chamada a função *RedefineClasses* para as classes armazenadas anteriormente. Essa função permite a redefinição dessas classes, forçando a chamada do evento `JVMTI_EVENT_CLASS_FILE_LOAD_HOOK` para cada classe, novamente, possibilitando a instrumentação nesse momento, devido a essas classes já terem sido carregadas.

Como a JVMTI não realiza a instrumentação diretamente, ela disponibiliza em sua distribuição um exemplo de biblioteca dinâmica para a instrumentação de métodos e classes, chamada *java_ crw_ demo*, que foi utilizada neste trabalho. A instrumentação foi realizada em dois pontos diferentes, no momento da chamada

dos métodos de uma determinada classe e no momento do retorno desses métodos. Porém, surgiu um problema: essa biblioteca inseria chamadas a métodos com parâmetros desnecessários e incompletos para o agente. A solução a esse problema foi modificar o código da biblioteca de alteração de *bytecodes* com intuito de atender às necessidades do agente.

Uma das modificações na biblioteca *java_ crw_ demo* foi em relação aos parâmetros passados à função inserida na chamada de métodos. Para isso, foi necessário conhecer a pilha de execução de um método para a recuperação do objeto e retirada das informações desnecessárias. A partir disso, foram alterados os parâmetros inseridos, com objetivo de conter apenas o identificador do método e o objeto proprietário desse método. Entretanto, os métodos estáticos não possuem objetos. Dessa forma, foi necessário identificar os métodos estáticos e instrumentá-los de forma diferente, passando como parâmetro, à função inserida, apenas o identificador do método. Além disso, os identificadores dos métodos, passados como parâmetros, são números gerados pelo agente, sendo necessário a partir dessa versão, utilizar sempre esses identificadores e não os identificadores dos métodos passados pela JVMTI.

Outra modificação na biblioteca foi a retirada de todos os parâmetros da função inserida no momento do retorno de métodos. Essa alteração foi realizada, pois não são necessárias informações adicionais dos métodos no retorno, apenas que o evento ocorreu.

Com essas alterações na biblioteca *java_ crw_ demo*, é inserida uma chamada a uma função estática com os parâmetros necessários à visualização relativa ao ponto de inserção. Por sua vez, essa função quando chamada invoca uma função de tratamento em código nativo (JNI), passando os parâmetros recebidos e a *thread*, que é obtida na própria função inserida. Dessa forma, essa função de tratamento recupera o *buffer* da respectiva *thread* e registra os identificadores do método e do objeto, se for o caso e se existir. Nesse ponto, surgiu outro problema com o identificador do objeto: o objeto passado é a referência ao objeto e não o identificador desse objeto. Para resolver isso, foi necessária a utilização da função *GetTag* para recuperar a *tag* (valor único) desse objeto, usado como seu identificador no agente.

Essa *tag* é atribuída ao objeto com a função *SetTag* no momento da criação desse objeto.

4.3.2.3 Problema da ocorrência de exceções

Um grave problema surge na ocorrência de *exceções*. Quando uma exceção ocorre é possível que métodos tenham suas execuções interrompidas, não executando o código de registro inserido no final. Sendo assim, para a visualização, o método interrompido não terá finalizado. Além disso, o próximo evento de retorno de método observará as informações do método errado (método interrompido, mas não finalizado) e não as suas informações, pois para a visualização o último método que foi chamado foi o método interrompido, finalizando-o erroneamente.

Com objetivo de resolver esse problema, é monitorado a ocorrência do evento `JVMTI_EVENT_EXCEPTION`, disponibilizado pela JVMTI, que identifica a ocorrência de exceções. Quando esse evento ocorre, as informações da exceção são registradas, e é necessário habilitar através da função da JVMTI *NotifyFramePop*, a ocorrência do evento `JVMTI_EVENT_FRAME_POP` para os *frames* atual e anterior. O evento `JVMTI_EVENT_FRAME_POP` é interceptado pelo agente quando um *frame* sai da pilha de execução, porque foi interrompido ou finalizado normalmente. Com esse evento, é possível identificar se o método foi interrompido ou foi finalizado normalmente após a execução. Caso o método tenha sido interrompido, é registrada a ocorrência da interrupção do método com intuito de forçar a sua finalização. Além disso, como não se sabe quantos métodos foram interrompidos com essa exceção, é necessário habilitar o monitoramento desse evento novamente para o *frame* anterior, até que ocorra a notificação de um *frame* que tenha terminado normalmente.

A figura 4.5 é a visualização em Pajé do comportamento de um programa, que foi obtido através da *JRastro* com a seleção dos eventos de chamada e retorno de métodos, e a seleção de métodos. Nessa visualização, pode ser observada a chamada de métodos através do início dos retângulos, e o retorno com o fim desses. Além disso, a figura mostra a ocorrência de uma exceção no programa, através do evento do Pajé mais escuro, e a finalização forçada do método que obteve a exceção, evento

mais claro. Esses eventos são apresentados no Pajé com o nome de "*ExceptionE*".

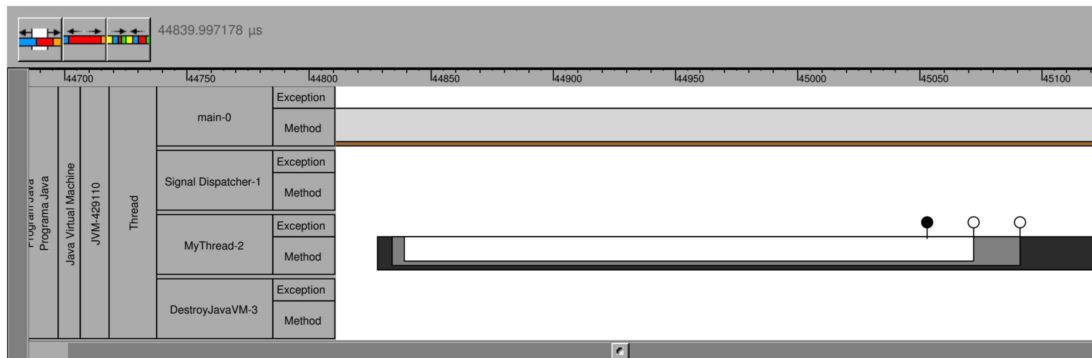

Figura 4.5: Visualização de chamadas/retornos de métodos e exceções no Pajé

4.3.3 Rastreamento de monitores

Os monitores garantem que apenas uma *thread* estará ativa em uma região em qualquer instante de tempo, obtendo a exclusão mútua. Eles são utilizados para conter regiões que possuem memória compartilhada entre várias *threads*, sem corromper as informações através do acesso concorrente. A visualização dos monitores é bastante útil, pois possibilita observar o impacto no desempenho de possuir regiões bloqueantes e auxilia na detecção de impasses (*deadlocks*).

A figura 4.6 representa a visualização desejada no Pajé. Nessa visualização, pode ser observado quando uma *thread* fica bloqueada por tentar entrar em uma região com exclusão mútua. Isso é observado através das mudanças de estados dessa *thread* para *Thread blocked* (2) e do estado do monitor para *Monitor locked* (5), e também, da seta com origem na *thread* que ficou bloqueada e destino no monitor. Além disso, é observado a liberação da *thread* (3) e do monitor (6) que são desempilhados, o que representa que não estão mais bloqueados, além da seta que tem origem no monitor e destino na *thread* liberada.

Para essa visualização, é preciso monitorar os momentos em que uma *thread* fica bloqueada e quando é liberada. Nesses momentos é registrado no arquivo específico da *thread* o identificador do objeto proprietário do método sincronizado. Esse identificador é necessário para o reconhecimento do monitor em que a *thread* está

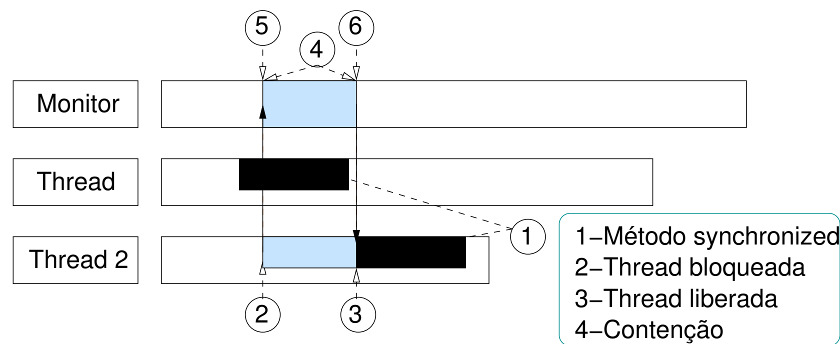

Figura 4.6: Rastreamento de monitores

bloqueada.

A JVMTI possibilita rastrear dois tipos de mecanismos para sincronização de *threads* oferecidos pelo Java, o mecanismo através de métodos sincronizados ("*synchronized*") e o mecanismo através de operações "*wait*" e "*notify*". As seções seguintes descrevem as implementações realizadas para o rastreamento desses mecanismos.

4.3.3.1 Métodos sincronizados

Para a implementação do rastreamento de métodos sincronizados são utilizados dois eventos disponíveis na JVMTI relacionados ao bloqueio e liberação de *threads* por regiões sincronizadas. No momento em que uma *thread* tenta e não consegue entrar em uma região sincronizada, devido a outra *thread* estar ativa na região, ela fica bloqueada e o evento `JVMTI_EVENT_MONITOR_CONTENDED_ENTER` é gerado. Através desse evento é possível detectar gargalos, indicados por muitas *threads* estarem bloqueadas por um mesmo monitor. A outra situação é quando uma *thread* após ter estado bloqueada por um monitor é liberada e o evento `JVMTI_EVENT_MONITOR_CONTENDED_ENTERED` gerado. Em ambos eventos citados o identificador do objeto proprietário do método sincronizado é registrado nos arquivos específicos das respectivas *threads*, pois esses eventos fornecem as mesmas informações. Esse identificador do objeto registrado é obtido através da função *GetTag* com a referência ao objeto recebido pelo evento. A partir dessas informações é possível converter os dados registrados para o Pajé, obtendo a visualização esperada.

A figura 4.7 é a visualização em Pajé do comportamento de um programa,

que foi obtido através da *JRastro* com a seleção dos eventos de monitores. Nesse programa, são criados três *threads* que chamam um mesmo objeto com um método sincronizado cinco vezes. Isso pode ser observado na figura através das três *threads*: *MyThread-1, MyThread-2 e MyThread-3* , e com as chamadas ao método sincronizado, representado na figura com um retângulo mais escuro. O bloqueio de uma *thread* é representado através do início de um retângulo mais claro, um evento no Pajé (mais claro) e uma seta indicando para qual monitor ela ficou bloqueada. Para a liberação de uma *thread* a representação é realizada com o fim desse retângulo, um evento no Pajé (mais escuro) e através de uma seta de retorno do monitor para a *thread* desbloqueada. Esses eventos são apresentados no Pajé com o nome de "*MonitorE*". Dessa forma, essa visualização ilustra a concorrência para execução do método compartilhado entre duas das três *threads* (*MyThread-1, MyThread-2*) e a outra *thread* (*MyThread-3*) ficando bloqueada até as duas terminarem a execução do método, para nesse momento chamar em seqüencia esse método. Essa visualização permitiria ao desenvolvedor da aplicação encontrar meios de melhorar o desempenho. Por exemplo, a *thread* que fica bloqueada poderia executar outras tarefas antes de chamar o método sincronizado, ou mesmo, criar alternativas para esse método sincronizado.

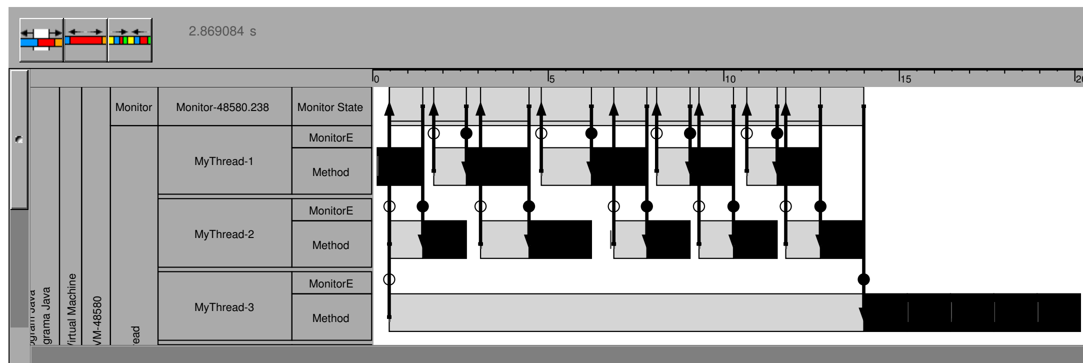

Figura 4.7: Visualização de monitores no Pajé

4.3.3.2 Operações "*wait*" e "*notify*"

Outra forma de obter exclusão mútua em Java é através das operações *wait* e *notify*. Através dessas operações, é possível bloquear e desbloquear *threads*.

O rastreamento dessa outra forma de obter exclusão mútua é feita através dos eventos `JVMTI_EVENT_MONITOR_WAIT` e o `JVMTI_EVENT_MONITOR_WAITED`. O evento `JVMTI_EVENT_MONITOR_WAIT` indica que a *thread* passada como argumento está esperando (bloqueada). O evento `JVMTI_EVENT_MONITOR_WAITED` representa que a *thread* que estava esperando foi desbloqueada. As técnicas do registro e da visualização desses eventos são as mesmas dos métodos sincronizados descritas na seção anterior.

4.3.4 Rastreamento do uso de memória

O gerenciamento de memória da linguagem Java oferece um recurso chamado coletor de lixo (*garbage collector*), que faz a desalocação automática da memória que não é mais referenciada pelo programa. Esse recurso é acionado periodicamente ou quando a memória é muito usada para fazer a desalocação dos objetos que não possuem mais referências (objetos que não serão mais usados no programa).

O rastreamento da alocação e da liberação de memória possibilita que o usuário verifique o estado de utilização de memória no programa. A partir disso, é possível detectar regiões com excessos de utilização de memória. Esses excessos podem afetar no desempenho da aplicação, devido, à sobrecarga ocasionada ao sistema de memória virtual e à ação freqüente do coletor de lixo.

A figura 4.8 apresenta a visualização desejada no Pajé, para os eventos de memória e para os estados da JVM. Nessa visualização, os eventos de alocação e liberação de memória são observados em um gráfico (4), sendo a alocação um acréscimo nesse gráfico e a liberação de memória uma diminuição. Uma observação importante sobre essa liberação de memória é que ela só ocorre no momento da execução do coletor de lixo. Além dessa representação, é apresentado também o estado da JVM em uma régua de estados. Essa representação inicia no estado

"*Executando*" (1). Quando o coletor de lixo começa a executar é alterado o estado inicial para "*Coletor de Lixo*" (2) e no término é desempilhado esse estado (3).

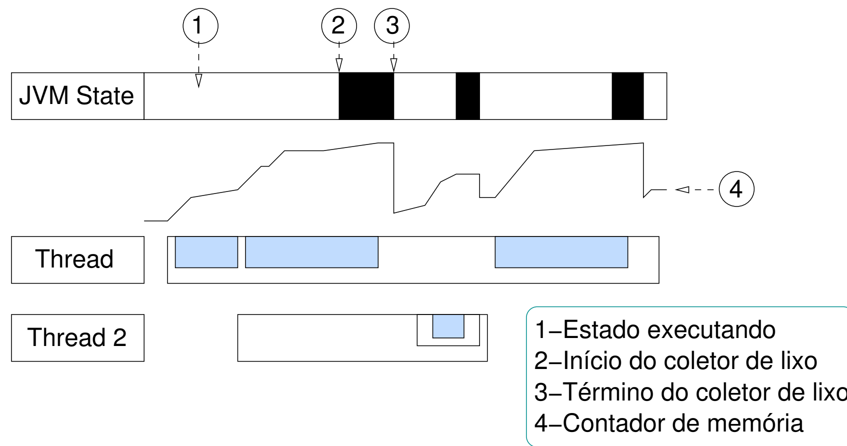

Figura 4.8: Rastreamento da alocação e liberação de memória

As seções seguintes descrevem com mais detalhes o rastreamento dos eventos da memória.

4.3.4.1 Eventos do coletor de lixo

Com objetivo da visualização dos eventos do coletor de lixo, é necessário determinar o momento do começo e do término da execução do coletor no programa. A partir disso, a ocorrência desses eventos é registrada.

O monitoramento do começo de um ciclo de execução do coletor de lixo é realizado através do evento `JVMTI_EVENT_GARBAGE_COLLECTION_START`, disponibilizado pela JVMTI. Para o monitoramento do momento de término desse ciclo, a JVMTI possui a notificação do evento `JVMTI_EVENT_GARBAGE_COLLECTION_FINISH`. Uma importante observação sobre esses eventos é que são gerados com a JVM parada. Além disso, na ocorrência deles é registrado apenas que ocorreram, pois não possuem informações adicionais.

4.3.4.2 Eventos de alocação e liberação de memória

A visualização do estado de memória em um determinado tempo na execução do programa pode ser realizada através do rastreamento dos eventos de alocação e

liberação de objetos. No evento de alocação de objetos, são registrados o identificador do objeto e qual o espaço utilizado por ele, com objetivo de ser adicionado à variável de memória total ocupada do Pajé no momento da conversão. No evento de liberação, é preciso apenas registrar o identificador do objeto, pois o espaço que ele ocupa pode ser recuperado no momento da conversão e nesse momento o valor recuperado diminuído da variável.

Para o monitoramento da alocação de memória, a JVMTI não dispõe de eventos diretos. Entretanto, a biblioteca *java_ crw_ demo* suportada pela JVMTI, oferece funções para a instrumentação em pontos específicos da criação de objetos e de *arrays*, descritos no capítulo 3. Através de BCI nesses pontos, é possível interceptar o momento da alocação de memória, recebendo como parâmetro a referência ao objeto alocado. Com a interceptação do evento, o agente necessita obter o espaço de memória alocado para o objeto recebido, através da função *GetObjectSize* da JVMTI. Além disso, o agente precisa atribuir um identificador ao objeto, através da função *SetTag*, pois esse objeto foi criado nesse momento. Após, o agente realiza o registro das informações da alocação do objeto, armazenando o identificador do objeto, o seu espaço utilizado, e também a classe desse objeto, obtida através de funções JNI.

O monitoramento da liberação de memória é realizado através do evento `JVMTI_EVENT_OBJECT_FREE` disponível na JVMTI. Esse evento é gerado sempre após a execução do coletor de lixo, indicando que um objeto específico foi liberado pelo coletor. Para a ocorrência desse evento com um determinado objeto, a JVMTI exige atribuição de um valor a *tag* desse objeto, para não haver notificações de liberação de objetos desconhecidos. O registro desse evento é feito com o identificador do objeto apenas.

A figura 4.9 ilustra a alocação e liberação de memória, através do gráfico de linha. Além disso, podem ser observados os estados da JVM na régua *JVM State*, com a ocorrência do coletor de lixo em diferentes momentos da execução do programa. Como é visualizado nessa figura, a liberação só ocorre com a execução do coletor de lixo.

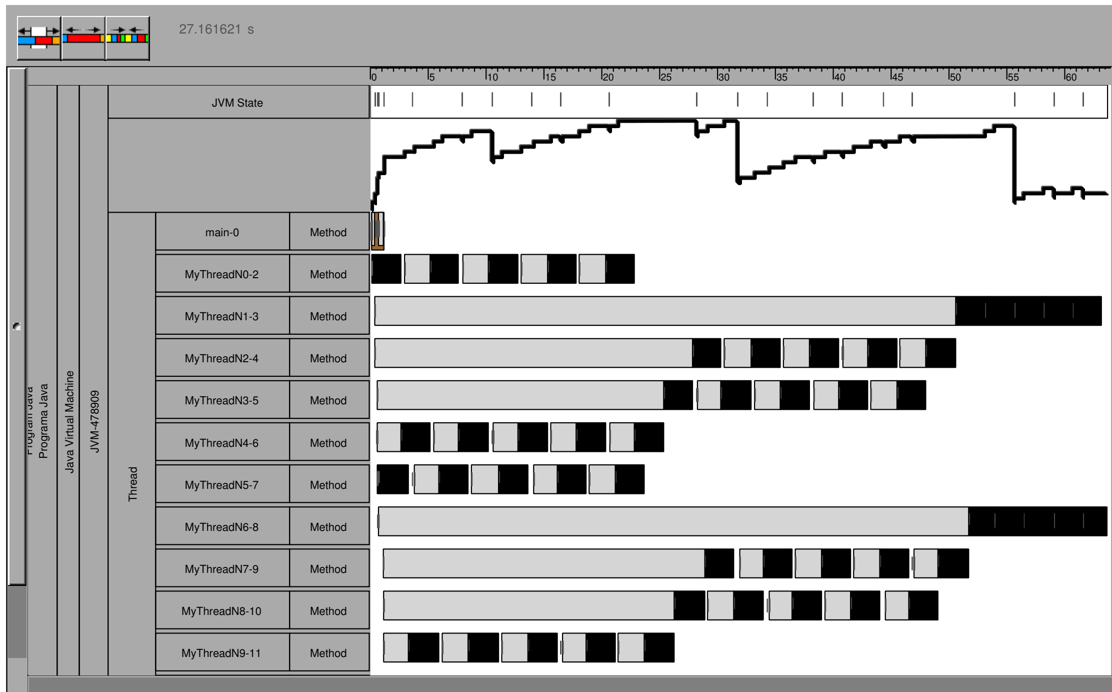

Figura 4.9: Visualização dos eventos da memória

4.3.5 Rastreamento de comunicações RMI

Java possui como um dos recursos para o desenvolvimento de aplicações distribuídas a invocação de métodos remotos (*RMI - Remote Method Invocation*). RMI é a adaptação para a linguagem orientada a objetos da chamada de procedimentos remotos (*RPC - Remote Procedure Call*). RPC é a execução de um procedimento em um outro processo, sendo esse processo local ou em uma outra máquina qualquer na rede. O modelo utilizado na execução é o cliente/servidor. O processo chamado servidor disponibiliza serviços que podem ser acessados remotamente, e o processo cliente é o que faz a chamada remota (requisita esses serviços). A abordagem orientada a objetos desse modelo é que o servidor implementa métodos que podem ser acessados remotamente por outros objetos. Dessa forma, é importante na depuração dessas aplicações, a visualização dessas comunicações, pois permite ao desenvolvedor identificar pontos de congestionamentos no servidor, por exemplo.

A figura 4.10 ilustra um esquema de como a visualização das chamadas remotas poderia ser realizada em Pajé. A representação seria semelhante a chamadas de

métodos locais, e teria duas setas entre JVMs diferentes. Uma seta com origem no método da JVM local (cliente) e com destino no método da JVM remota (servidor), indicando a chamada de método remoto (1), e uma outra seta contrária, indicando o retorno do método remoto (2).

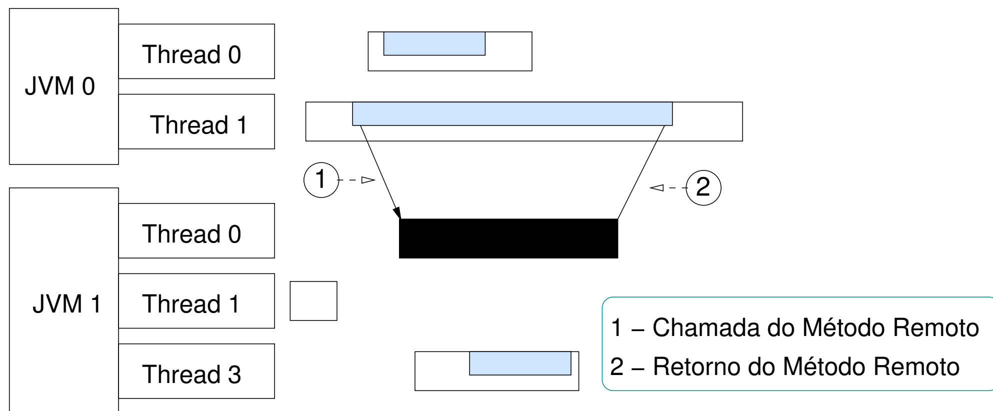

Figura 4.10: Rastreamento de comunicações RMI.

Para realizar essa representação no Pajé é necessário rastrear os eventos em que são realizadas as chamadas e retornos de métodos remotos. Nesses momentos são necessárias as identificações de quais *threads* e JVMs que chamam ou mesmo executam os métodos remotos. A partir disso, fazer a interação entre as JVMs com as setas.

Para isso, a JVMTI não possui eventos de notificação. A alternativa para o rastreamento desses eventos é detectar os métodos que sempre são executados em uma chamada remota no cliente e no servidor. Além dos métodos, também é necessário detectar quais as *threads* executam as chamadas remotas. A partir disso, é preciso verificar a existência de informações para distinção de qual a JVM que será chamada para executar um método remoto (seta de chamada de métodos remotos) e para qual JVM o método remoto retornará (seta de retorno de métodos remotos).

Entretanto, ainda não foi verificado quais são esses métodos e *threads* que sempre executam e não se sabe a possibilidade de encontrar as informações necessárias para distinguir as JVMs no processo de comunicação. Dessa forma, o estado atual do agente é de permitir a visualização de diferentes JVMs, mas não a interação entre

elas.

4.4 Tipos de visualização

O agente de rastreamento utiliza a biblioteca *libRastro* para o registro das informações dos eventos monitorados. Essa biblioteca gera rastros de execução em um formato próprio para aumentar o desempenho no momento da gravação. Sendo assim, após o rastreamento da aplicação é necessária a utilização de programas que usam funções de leitura da *libRastro*, para a conversão dos rastros gerados para o formato da ferramenta de visualização Pajé.

A ferramenta Pajé permite que a forma de apresentação dos dados seja informada juntamente com esses dados. Essa ferramenta é facilmente adaptável a diversas visualizações. Dessa forma, é possível converter os mesmos rastros gerados para diferentes tipos de visualizações no Pajé. A partir disso, o usuário tem a liberdade de escolher o tipo de visualização que melhor esclarece a execução do programa para ajudar na depuração da aplicação.

Neste trabalho, foram desenvolvidos programas para a conversão dos rastros gerados pelo agente de rastreamento, com o uso da biblioteca libRastro para a leitura e controle desses rastros para três diferentes tipos de visualizações: por *threads* (seção 4.4.1), por objetos (seção 4.4.2) e por objetos hierarquizados por suas respectivas classes (seção 4.4.3). Esses tipos de visualizações são descritos nas próximas seções.

4.4.1 Visualização por *threads*

Na visualização por *threads* são apresentados os eventos e estados (chamada e retorno de métodos, entre outros) associados a *thread* que os executou. Isso auxilia na depuração de programas, pois permite ao usuário observar a ordem de execução das *threads*, os métodos que demoraram mais tempo, as *threads* que ficaram bloqueadas em uma região compartilhada e as que obtiveram o monitor, entre outras possibilidades. A figura 4.11 foi obtida através da conversão dos rastros gerados

durante a execução de um programa para a visualização por *threads*. Nessa figura, é observado a criação de duas novas *threads* e a chamada e retorno de métodos selecionados para a visualização.

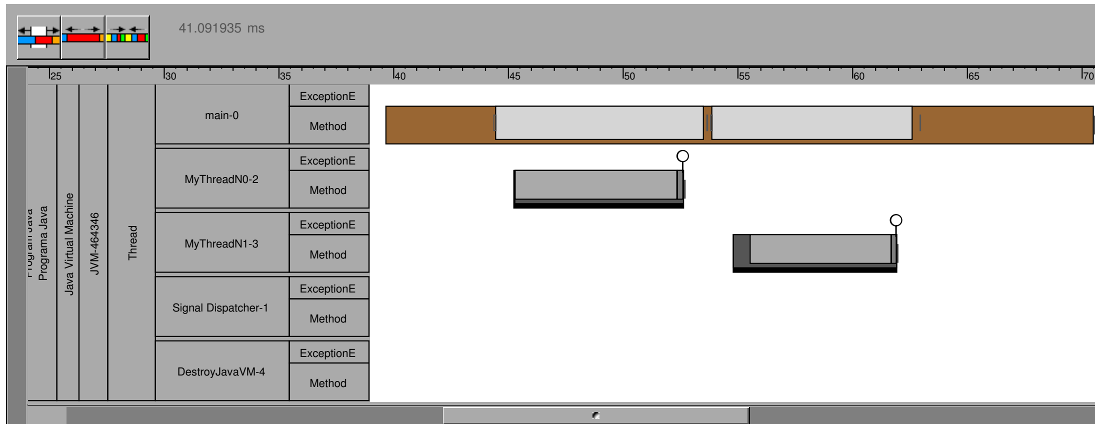

Figura 4.11: Visualização por *thread*

4.4.2 Visualização por objetos

O segundo tipo de visualização, por objeto, permite observar quais métodos um objeto está executando em um determinado tempo. A partir disso, essa visualização possibilita verificar o nível de execução concorrente no interior do objeto. O problema dessa visualização é a grande quantidade de objetos [2] com suas respectivas informações e sem agrupamentos específicos, dificultando na verificação do comportamento do programa. A figura 4.12 apresenta a visualização por objetos. Essa visualização foi obtida pela conversão dos mesmos rastros utilizados na conversão para a visualização por *threads*.

4.4.3 Visualização por objetos classificados pelas classes

O terceiro tipo hierarquiza a visualização através do classificação dos objetos pelas suas respectivas classes, melhorando a compreensão dos dados visualizados. Com isso, o problema mencionado no segundo tipo de visualização é parcialmente resolvido, pois a verificação dos objetos é facilitada com o agrupamento, apesar da

[2] Na execução de um programa Java é realizada diversas criações e destruições de objetos.

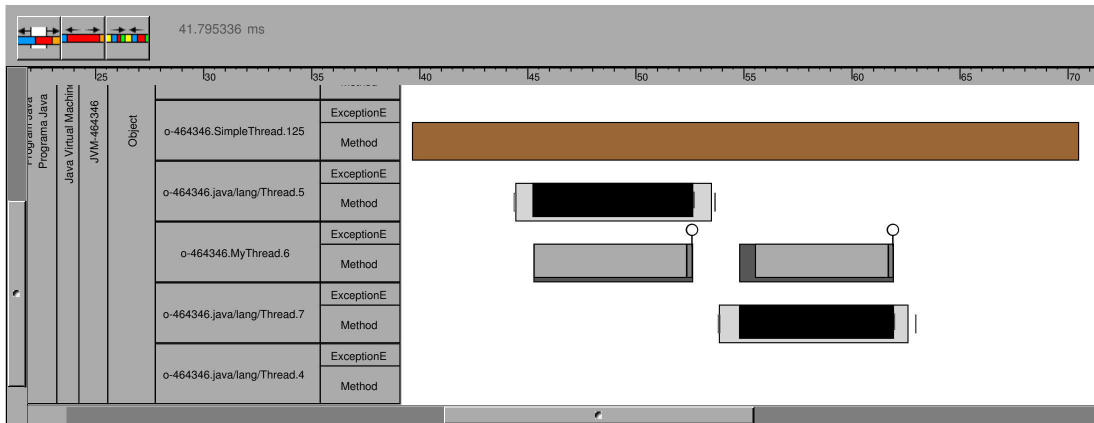

Figura 4.12: Visualização por objeto

grande quantidade de informações existentes. A figura 4.13 apresenta os objetos agrupados pelas suas classes.

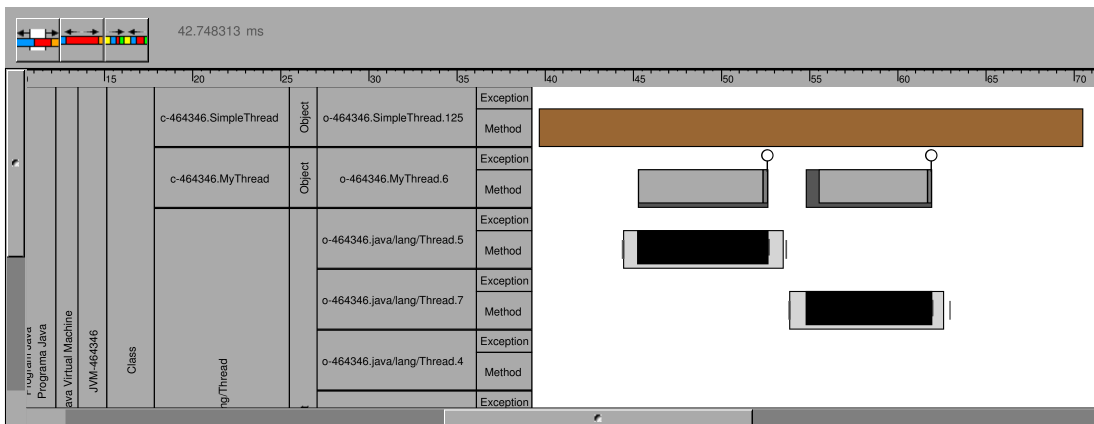

Figura 4.13: Objetos agrupados pelas suas classes

Capítulo 5

Avaliação

Este trabalho permite ao usuário visualizar o comportamento de programas Java paralelos e distribuídos. Para isso, foi desenvolvido um agente que realiza o rastreamento dos principais eventos para posterior visualização. Esse agente de rastreamento possui como características ser transparente ao desenvolvedor, não precisar de modificação da JVM e também, ser configurável.

Neste capítulo descreve-se as avaliações realizadas nesse agente com a seleção de diferentes eventos e quantidades de dados para o rastreamento. Na próxima seção descreve-se o ambiente de execução dos testes e após discute-se esses testes com mais detalhes. Concluindo o capítulo, são apresentadas avaliações do desempenho dos resultados computacionais obtidos.

5.1 Ambiente de execução

Para efetuar os testes de desempenho da execução do programa com o agente de rastreamento foi utilizado um computador *Pentium 4* a 2.8 GHz, com 768 Mbytes de memória RAM e 512 Kbytes de memória cache. Nesse computador está instalado o sistema operacional GNU/Linux (distribuição Gentoo) com kernel versão 2.6.5 e com a versão do Java 1.5.0_04.

5.2 Testes realizados

Os testes realizados buscaram verificar a intrusão e a quantidade de eventos rastreados na depuração de um programa Java com a utilização do agente de rastreamento. O programa rastreado realiza a criação e aguarda o fim da execução de 10 *threads*. Com objetivo de gerar uma grande quantidade de eventos cada *thread* realiza os passos a seguir 4500 vezes. No primeiro momento as *threads* criam e inicializam (com dados constantes) uma matriz de 128 por 128 e um vetor de 128 inteiros. No segundo momento as *threads* chamam um método para realização de um cálculo nos elementos dessas variáveis e depois há uma chamada a um método que gera uma exceção (interrompendo sua execução) e a lança para o método chamador.

Esse programa foi escolhido para a execução dos testes pelo motivo de possuir um tempo de execução constante, pois não possui grandes problemas de sincronização, apesar de possuir uma certa quantidade de sincronismo pelas classes do sistema. Isso foi um problema com outros programas testados, como por exemplo o programa "jantar dos filósofos", em que os resultados dos tempos não eram determinísticos. Além desse motivo, foi escolhido esse programa para os testes, pois são gerados todos os possíveis eventos de rastreamento do agente.

O agente suporta diferentes configurações, o que motivou a execução de uma gama de testes. Foi possível selecionar quais eventos seriam rastreados, para a observação da intrusão do rastreamento desses eventos. Além disso, foi possível intercalar os testes com a seleção de todas as classes e métodos do programa, e com a seleção de apenas as classes e métodos do usuário.

Para cada tipo de teste foram realizadas 20 execuções do programa. Os resultados considerados foram obtidos através da média aritmética das amostras obtidas.

5.3 Resultados obtidos

Nesta seção, apresentam-se os resultados obtidos nos testes realizados com o rastreamento do programa mencionado na seção anterior com o agente desenvolvido neste trabalho. Além disso, é realizada uma análise desses resultados.

A tabela 5.1 apresenta para cada teste realizado com a sua seleção as quantidades de eventos registrados, o tempo de execução, a porcentagem de aumento para o tempo sem o agente e o tempo por evento obtido. Os tempos de execução são representados em segundos na tabela e os tempos por evento são em microsegundos. O tempo por evento é obtido através da subtração do tempo no teste fazendo a seleção de determinados eventos com o tempo obtido no teste sem eventos, e esse resultado é dividido com o número de eventos encontrados.

Tabela 5.1: Resultados obtidos nos diferentes testes realizados.

Seleções	**Quant.de ev. registrados**	**Tempo na exec.**	**Increm%**	**μs/evento**
Sem o agente	-	**17,965**	-	-
Sem eventos	0	18,16	1,08	-
Monitores	1570	18,251	1,59	57,96
Memória	103407	19,467	8,36	12,64
Métodos do Usuário	315055	18,417	2,52	0,81
Mon+Mem+MetUser	508108	18,817	4,74	1,29
Mon+Mem+MetUser+ +Exceção	614010	181,187	908	265,5
Todos métodos	339969	18,718	4,19	1,64
Mon+Mem+TodosMet	623961	20,107	11,9	3,12
Mon+Mem+TodosMet+ +Exceção	728509	178,433	893	220

A partir da análise da tabela pode-se constatar o bom rendimento do agente de rastreamento, mas não para os casos das ocorrências de exceções. Em um primeiro momento, pensou-se que o registro nesses eventos poderia ter afetado o desempenho. Entretanto, com uma observação e análise mais profundas na implementação desse rastreamento, verificou-se o problema na própria função *NotifyFramePop* fornecida pela JVMTI. Essa função habilita o monitoramento da saída de um *frame* da pilha de execução através do evento `JVMTI_EVENT_FRAME_POP`. Nesse evento, é possível verificar se o *frame* foi finalizado ou foi interrompido. Caso tenha sido interrompido é registrado essa interrupção com objetivo de finalizar o método na posterior visualização. Isso é realizado para não ocorrer os problemas mencionados no capítulo anterior (seção 4.3.2.3). Dessa forma, com a importância desse evento para

a visualização foi necessário a utilização da função *NotifyFramePop*, não se tendo elaborado uma solução alternativa.

Outro dado constatado, na observação dessa tabela, é o melhor desempenho do agente com a seleção de classes e métodos para o rastreamento. Isso ocorre porque existe uma grande quantidade de informações a serem gravadas no caso da seleção de todas as informações (classes e métodos). Entretanto, nos testes com exceções não se obteve um bom rendimento, como visto na tabela. Isso é pelo motivo de cada evento `JVMTI_EVENT_FRAME_POP` gerado necessitar a obtenção do nome do método interrompido para realizar a seleção, pois esse evento retorna apenas um identificador interno do método que não é usado pelo agente. A partir disso, a seleção torna-se cara para o programa testado, pois esse programa gera uma grande quantidade de exceções, necessitando de seleções a cada evento gerado. Até o momento não foi encontrada outra solução para a seleção na ocorrência de exceções.

Concluindo-se a análise do desempenho do agente a partir dos resultados obtidos pode-se afirmar que o agente teve um bom rendimento, mas não para o caso da ocorrência de uma grande quantidade de exceções. Entretanto, isso não afeta na avaliação do agente, pois o baixo rendimento só ocorre em casos especiais.

Capítulo 6

Conclusão

Este trabalho teve como objetivo o desenvolvimento de uma ferramenta que auxilia a depuração de programas Java paralelos e distribuídos. Para isso, desenvolveu-se um agente de rastreamento que realiza o monitoramento e registro dos principais eventos para posterior visualização. Esse agente tem como características ser transparente ao desenvolvedor, não precisar da modificação da JVM e também, ser configurável. Na implementação do agente foi utilizado o módulo JVMTI, que recupera os eventos gerados pela JVM e a biblioteca *libRastro*, que registra esses eventos em arquivos de rastros. Com a geração do rastros, são realizadas conversões dessas informações para três tipos de visualizações no Pajé, por *threads*, por objetos e por objetos classificados pela suas respectivas classes.

A ferramenta desenvolvida possibilita o rastreamento de diversos eventos. Entre esses eventos destacam-se o rastreamento de *threads* (criação e destruição), a chamada e o retorno de métodos com a visualização da ocorrência de exceções, o rastreamento dos monitores com as *threads* bloqueadas e liberadas e o rastreamento da memória com a visualização do estado da memória em um determinado tempo e o estado do coletor de lixo. Além disso, esse agente permite a visualização de diferentes JVMs, mas não a interação entre elas. Com esses eventos obteve-se resultados satisfatórios para a depuração e controle de programas.

O agente de rastreamento torna-se muito intrusivo na execução da aplicação com a utilização da função *NotifyFramePop* disponibilizada pela JVMTI, na ocorrência de diversas exceções. Portanto, como trabalho futuro pretende-se verificar

uma alternativa para o rastreamento de exceções.

Outra proposta é concluir o rastreamento das comunicações RMI. Isso poderá ser realizado através da identificação dos métodos e *threads* que fazem as comunicações[1], e com a tentativa da identificação das JVMs do cliente e do servidor. Além disso, é desejável o rastreamento de outras formas de comunicações utilizadas no desenvolvimento de programas paralelos e distribuídos (por exemplo *sockets*), com objetivo de auxiliar na sua depuração.

[1]A JVMTI não disponibiliza informações sobre esses eventos, tornando essa tarefa mais complexa.

Referências Bibliográficas

[BOR 2004] BORLAND. **JBuilder**. http://www.borland.com/us/products/jbuilder/index.html.

[BOR 2004a] BORLAND. **Borland Optimizeit Suite**. http://www.borland.com/optimizeit.

[COH 2004] COHA, J.; SEIDMAN, D. **Bytecode Instrumentation – Making it Simple**. http://www.hp.com/products1/unix/java/pdfs/bytecode.pdf.

[FOS 95] FOSTER, I. **Designing and building parallel programs - concepts and tools for parallel software engineering**. USA: Addison-Wesley, 1995.

[Kel 2004] Kelly O'Hair. **The JVMPI Transition to JVMTI**. http://java.sun.com/developer/technicalArticles/Programming/jvmpitransition/.

[Kel 2004a] Kelly O'Hair. **HPROF**: A Heap/CPU Profiling Tool in J2SE 5.0. http://java.sun.com/developer/technicalArticles/Programming/HPROF.html.

[LIA 99] LIANG, S. **Java native interface**: programmer's guide and specification. [S.l.]: pub-AW, 1999.

[SIL 2004] SILVA, G. J. da et al. **Biblioteca de Rastreamento libRastro**. http://www.inf.ufsm.br/lsc/libRastro.

[SIL 2002] SILVA, G. J. da; STEIN, B. Uma Biblioteca Genérica de Geração de Rastros de Execução para Visualização de Programas. **Anais do I Simpósio de Informática da Região Centro, Santa Maria**, 2002.

[SIL 2003] SILVA, G. J. da; STEIN, B. Geração de Rastros de Execução para Visualização de Programas Java Distribuídos. **Trabalho de Graduação. Universidade Federal de Santa Maria**, 2003.

[STE 2000] STEIN, B.; KERGOMMEAUX, J. C. de; BERNARD, P. É. Pajé, an interactive and visual tool for tuning multi-threaded parallel applications. **Parallel Computing**, v.26, n.10, Aug. 2000.

[STE 99] STEIN, B. **Visualisation interactive et extensible de programmes parallèles à base de processus légers**. 1999. Thèse de doctorat en informatique — Université Joseph Fourier, France.

[Sun 99] Sun Microsystems. **Java Remote Method Invocation Specification**. Revision 1.8 Java 2 SDK, version 1.4.ed. [S.l.: s.n.], 1999.

[Sun 2004] Sun Microsystems. **Java Virtual Machine Profiler Interface**. `http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html`.

[Sun 2004a] Sun Microsystems. **Java Virtual Machine Tool Interface**. `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html`.

[Sun 2004b] Sun Microsystems. **NetBeans**. `http://www.netbeans.org/`.

[TAN 2002] TANENBAUM, A. S.; van STEEN, M. **Distributed Systems. Principles and Paradigms**. [S.l.]: Prentice, 2002.

[TAN 97] TANENBAUM, A. S.; WOODHULL, A. S. **Operating Systems. Design and Implementation**. 2.ed. [S.l.]: Prentice-Hall, 1997.

[The 2004] The Eclipse Foundation. **Eclipse**. http://www.eclipse.org/.

[WIE 2005] WIEDENHOFT, G. R.; STEIN, B. Rastreamento e visualização de programas Java usando JVMTI. **Anais da Quinta Escola Regional de Alto Desempenho, Canoas, RS, Brasil**, 2005.

Apêndice A

Manual de instalação e utilização da *JRastro*

A instalação e utilização da biblioteca *JRastro* é fácil e simples. Na seção seguinte apresenta-se o requisito principal à utilização da *JRastro*. Após, descrevem-se a instalação (seção A.2) e a utilização (seção A.3) dessa biblioteca. Finalizando este manual, a seção A.4 contém os eventos e as funções disponibilizados pela JVMTI.

A.1 Requisito

O requisito principal para a utilização do agente é que esteja instalado no sistema a versão 1.5 ou superior da JDK. Pois, versões anteriores não possuem o módulo JVMTI, não sendo compatíveis com a biblioteca *JRastro*.

A.2 Instalação

A biblioteca *JRastro* pode ser encontrada em `http://www.inf.ufsm.br/lsc/libjRastro`. Após o *download* da *JRastro* é necessário alterar as variáveis `JDK_HOME`, `JRASTRO_DIR` e `JRASTRO_DIR_BIN` do arquivo *Makefile*, descritas a seguir:

- A variável `JDK_HOME` deve ser alterada para conter o caminho correto da instalação da JDK.
- A variável `JRASTRO_DIR` contém o local que serão instalados os arquivos auxiliares da biblioteca.

- A variável `JRASTRO_DIR_BIN` indica o local onde serão instalados os binários e programas auxiliares. O valor dessa variável deve estar ou ser colocado na variável de ambiente *PATH*.

Com as alterações no arquivo *Makefile*, os processos de compilação, instalação e desinstalação estão prontos para serem executados. Esses processos são descritos abaixo:

- A compilação da biblioteca *JRastro* ocorre com a execução do comando:
  - $ make
- Na instalação da biblioteca *JRastro* executa-se o comando como usuário root:
  - # make install
- Para desinstalar a biblioteca *JRastro*, execute como root:
  - # make uninstall

A.3 Utilização

A biblioteca *JRastro* está pronta para ser utilizada, após compilação e instalação. Para realizar o rastreamento de um programa Java (por exemplo *MeuPrograma*) através da biblioteca *JRastro* tem-se duas possibilidades:

- `java -agentlib:jRastro=sEventos,sMetodos MeuPrograma`

  A opção *-agentlib* na execução do programa *MeuPrograma*, faz com que a máquina virtual carregue a biblioteca *JRastro*. A biblioteca será carregada com os parâmetros *sEventos e sMetodos*, que são dois arquivos de configurações para o agente de rastreamento. O arquivo *sEventos* deve conter a seleção de quais eventos se deseja rastrear, e o arquivo *sMetodos* contém a seleção das classes e métodos que serão rastreados, conforme descritos na seção 4.2. Essa opção indica que a biblioteca *JRastro* será carregada do diretório padrão das

bibliotecas (colocada no momento da instalação), ou de um dos diretórios da variável de ambiente `LD_LIBRARY_PATH`.

- `java -agentpath:/usr/libjRastro.so=sEventos,sMetodos MeuPrograma`

  A opção *-agentpath* nessa linha de execução carrega a biblioteca *JRastro* do caminho específico "`/usr/`" com os parâmetros *sEventos e sMetodos.*

Após a execução do programa, os arquivos de rastros gerados devem ser convertidos para o formato do visualizador Pajé. Essa conversão pode ser realizada por um dos três conversores fornecidos com a *JRastro.* Esses conversores recebem como parâmetros um arquivo contendo as informações dos ajustes dos relógios das máquinas[1], ou um arquivo vazio, caso o programa tenha sido executado em uma máquina, e os rastros gerados durante o rastreamento. Abaixo são apresentadas as três conversões possíveis:

- # jRastro_read arq rastro-*

  Essa conversão gera o arquivo no formato do Pajé, chamado "rastros.trace", contendo as informações para a visualização por *threads.*

- # jRastro_readObj arq rastro-*

  Essa conversão gera o arquivo "rastrosObj.trace", que contém as informações para a visualização por *objetos.*

- # jRastro_readObjClass arq rastro-*

  Essa conversão tem como saída o arquivo "rastrosObjClass.trace" contendo as informações para a visualização por *objetos classificados pelas classes.*

A.4 Eventos e funções da JVMTI

A JVMTI dispõem de uma grande quantidade de eventos e funções para o monitoramento de programas Java. A tabela A.1 apresenta os eventos disponibilizados

[1] Arquivo gerado com a utilização da biblioteca *libRastro* antes e depois da execução do programa.

pela JVMTI, incluindo os eventos usados neste trabalho (eventos em destaque na tabela).

Tabela A.1: Eventos disponibilizados pela JVMTI.

Breakpoint	Method Entry
Class File Load Hook	Method Exit
Class Load	**Monitor Contended Enter**
Class Prepare	**Monitor Contended Entered**
Compiled Method Load	**Monitor Wait**
Compiled Method Unload	**Monitor Waited**
Data Dump Request	Native Method Bind
Dynamic Code Generated	**Object Free**
Exception	Single Step
Exception Catch	**Thread End**
Field Access	**Thread Start**
Field Modification	VM Death Event
Frame Pop	**VM Initialization Event**
Garbage Collection Finish	VM Object Allocation
Garbage Collection Start	VM Start Event

A tabela A.2 contém a classificação das funções disponíveis pela JVMTI.

Tabela A.2: Classificação das funções da JVMTI.

Memory Management	Field
Thread	Method
Thread Group	Raw Monitor
Stack Frame	JNI Function Interception
Heap	Event Management
Local Variable	Extension Mechanism
Breakpoint	Capability
Watched Field	Timers
Class	System Properties
Object	General