UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

DOMonitor:
um Ambiente de Monitoração de Aplicações Distribuídas Java

por

EDVAR BERGMANN ARAUJO

Dissertação submetida à avaliação,
como requisito parcial para a obtenção do grau de Mestre em
Ciência da Computação

Prof. Dr. Cláudio Fernando Resin Geyer
Orientador

Porto Alegre, agosto de 2002.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Araujo, Edvar Bergmann

DOMonitor: um Ambiente de Monitoração de Aplicações Distribuídas Java / por Edvar Bergmann Araujo. - Porto Alegre: PPGC da UFRGS, 2002.

124f.: il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2002. Orientador: Geyer, Cláudio Fernando Resin

1. Monitoração de Aplicações Distribuídas. 2. Monitoração de Programas Java. 3. Visualização. 4. Programação com Objetos Distribuídos. I. Geyer, Cláudio Fernando Resin. II. Título

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitora: Profa. Wrana Maria Panizzi

Pró-Reitor de Ensino: Prof. José Carlos Ferraz Hennemann

Pró-Reitor Adjunto de Pós-graduação: Jaime Evaldo Fensterseifer

Diretor do instituto de Informática: Prof. Philippe Olivier Alexandre Navaux

Coordenador do PPGC: Prof. Carlos Alberto Heuser

Bibliotecária-Chefe do Instituto de Informática: Beatriz Haro

Agradecimentos

Gostaria de começar agradecendo o meu orientador, Cláudio Geyer pela importante participação nesta fase da minha vida, pela forma como me orientou e pelos ensinamentos valiosos que me transmitiu, sem os quais não seria possível completar este trabalho.

Ao professor e amigo Adenauer Yamin, gostaria de agradecer pelo apoio, dedicação e paciência. Caro Adenauer, com certeza tua participação foi decisiva.

A todas as pessoas que ajudaram de alguma forma o desenvolvimento deste projeto.

A todos os meus amigos e amigas, pelos momentos de alegria, de tristeza, pelas festas, pelas conversas, enfim pela grande amizade que possuímos. Vocês são muito importantes para mim.

A minha querida namorada, Letícia, por todo carinho e compreensão nos momentos difíceis e por tudo o que representas na minha vida.

A minha maravilhosa família, que me proporcionou todas as condições para que eu esteja completando esta fase da minha vida, que vocês continuem sempre me apoiando e me amando. Eu amo muito vocês.

Finalmente, mas não em último lugar, agradeço a Deus por mais esta oportunidade de crescimento pessoal.

Sumário

Lista de Abreviaturas 7
Lista de Figuras 8
Lista de Tabelas 10
Resumo 11
Abstract 12
1 Introdução 13
1.1 Tema 13
1.2 Motivação 13
1.3 Contexto Local 15
1.3.1 DOBuilder 15
1.3.2 DEPAnalyzer 16
1.3.3 ReMMos 17
1.4 Objetivos 17
1.5 Contribuições do Autor 18
1.6 Estrutura do Texto 19
2 Programação Paralela e Distribuída com Java 20
2.1 Ambiente Orientado a Objetos 20
2.2 Independência de Plataforma 20
2.3 *MultiThreading* 21
2.4 Suporte à Rede 21
3 Monitoração e Visualização de Programas 24
3.1 Conceitos Relacionados à Monitoração de Programas 24
3.2 Pablo 25
3.3 Paradyn 26
3.4 Pajé 28
3.5 HProf 28
3.6 OptimizeIt 29
3.7 Jinsight 31
3.8 JaViz 32
3.8.1 Coleta das Informações 34
3.8.2 Pós-Processamento 35
3.8.3 Visualização 36
3.9 Deterministic Replay of Distributed Java Applications 37
3.9.1 Estrutura de Repetição 38

3.9.2 DejaVu Distribuído 41
3.9.3 Implementação e Resultados de Desempenho 42

3.10 Uma proposta de Visualização de Aplicações Distribuídas 43

4 Modelo DOMonitor 44

4.1 A Necessidade de Monitorar Programas Distribuídos 44

4.2 Objetivos da Monitoração do DOMonitor 45

4.3 Atividades de Monitoração realizadas por Software 47

4.4 Monitoração Baseada em Eventos 48

4.5 Demais Decisões do Modelo 49

4.6 Arquitetura do DOMonitor 51
4.6.1 Definição dos Parâmetros para Monitoração 52
4.6.2 Coleta de Informações 53
4.6.3 Publicação de Resultados 53

5 A Coleta de Informações no DOMonitor 55

5.1 Definição dos Parâmetros de Monitoração 55
5.1.1 Definição dos Eventos de Interesse 55
5.1.2 Definição das Opções de Filtro 60
5.1.3 Definição do Processamento das Informações 61
5.1.4 Arquivo de Configuração 62

5.2 Detecção de Eventos 63
5.2.1 Detecção das Informações Locais - JVMPI 63
5.2.2 Detecção das Informações Remotas – *daemon* EXEd 67

5.3 Coleta das Informações Locais 69
5.3.1 Execução das *Threads* 69
5.3.2 Classes 71
5.3.3 Objetos 74
5.3.4 Invocação de Métodos 75
5.3.5 Eventos de Sincronização 76
5.3.6 Formato do Arquivo de Traço Local 79

5.4 Coleta das Informações Remotas 81
5.4.1 Invocação Remota de Método 82
5.4.2 Formato do Arquivo de Traço de Comunicação 84

6 O Processamento das Informações no DOMonitor 85

6.1 Comportamento Global da Execução 85
6.1.1 Adequação dos Relógios 85
6.1.2 Ligação entre as Invocações Remotas de Métodos 88
6.1.3 Publicação dos Resultados 90

6.2 Comportamento Local 92

7 Integração com outros Ambientes 93

7.1 Integração do DOMonitor com o Pajé 93
7.1.1 Expansibilidade 93
7.1.2 Integração com o Pajé 94

7.2 Integração do DOMonitor com o ISAM ____ 97
7.2.1 O contexto do ISAM ____ 98
7.2.2 A Arquitetura ISAM ____ 100
7.2.3 O DOMonitor no *Middleware* ISAM ____ 101
7.2.4 O Escalonamento no *Middleware* ISAM ____ 101
7.2.5 A Monitoração com o DOMonitor no ISAM ____ 104

8 A Implementação do DOMonitor ____ 107

8.1 Agente de Monitoração - DOProf ____ 107
8.1.1 Funcionamento ____ 108

8.2 Análise Comparativa ____ 111

9 Conclusão ____ 115

Referências Bibliográficas ____ 118

Lista de Abreviaturas

ASCII	American Standard Code for Information Interchange
API	Application Programming Interface
CPU	Central Processing Unit
CORBA	Common Object Request Broker Adapter
DJVM	DejaVu Java Virtual Machine
EXEHDA	Execution Environment for High Distributed Applications
GPPD	Grupo de Processamento Paralelo e Distribuído
GUI	Graphical User Interface
HPF	High Performance Fortran
I/O	Input/Output
IP	Internet Protocol
ISAM	Infra-Estrutura de Suporte às Aplicações Móveis
J2SE	Java 2 Platform Standard Edition
JDBC	Java DataBase Connectivity
JIT	Just-In-Time
JVM	Java Virtual Machine
JVMPI	Java Virtual Machine Profiler Interface
MPI	Message-Passing Interface
ORB	Object Request Broker
ReMMoS	Replication Model in Mobility Systems
RMI	Remote Method Invocation
SDDF	Self-Defining Data Format
SSL	Secure Socket Layer
Splash-2	Stanford Parallel Applications for Shared Memory - 2
SvPablo	Source View Pablo
TCP/IP	Transmission Control Protocol/Internet Protocol
UDP	User Datagram Protocol
UFRGS	Universidade Federal do Rio Grande do Sul

Lista de Figuras

FIGURA 1.1 – Interface de Programação do DOBuilder ____ 16
FIGURA 2.1 – Camadas RMI ____ 23
FIGURA 3.1 – Visão da Arquitetura do Paradyn ____ 27
FIGURA 3.2 – Profiler de Memória do OptimizeIt ____ 30
FIGURA 3.3 – Diagrama de Execução da Thread ____ 31
FIGURA 3.4 – Diagrama de Referência de Objetos ____ 32
FIGURA 3.5 – Arquitetura de Módulos do JaViz ____ 33
FIGURA 3.6 – Registro de Comunicação do JaViz ____ 35
FIGURA 3.7 – Janela de Visualização do JaViz ____ 37
FIGURA 3.8 – Programa Exemplo ____ 39
FIGURA 3.9 – Escalonamento Físico de Threads ____ 39
FIGURA 3.10 – Intervalo Lógico de *Threads* ____ 40
FIGURA 4.1 – Arquitetura de Módulos do DOMonitor ____ 51
FIGURA 4.2 – Interface Gráfica do Usuário ____ 52
FIGURA 5.1 – Monitoração Completa ____ 57
FIGURA 5.2 – Monitoração de Threads ____ 58
FIGURA 5.3 – Monitoração da Sincronização ____ 59
FIGURA 5.4 – Monitoração da Comunicação ____ 59
FIGURA 5.5 – Monitoração Livre ____ 60
FIGURA 5.6 – Processamento das Informações ____ 62
FIGURA 5.7 – Layout do Arquivo de Configuração do DOMonitor ____ 63
FIGURA 5.8 – JVM implementada por software sobre um S.O. ____ 64
FIGURA 5.9 – Arquitetura de monitoração utilizando JVMPI ____ 65
FIGURA 5.10 – Eventos Monitorados não Definidos ____ 65
FIGURA 5.11 – Definição dos Eventos Monitorados ____ 66
FIGURA 5.12 – Cliente e Servidor executam no mesmo *host* ____ 68
FIGURA 5.13 – Cliente e Servidor executam em *hosts* diferentes ____ 68
FIGURA 5.14 – Tupla THR_{INI} ____ 70
FIGURA 5.15 – Tupla THR_{FIM} ____ 70
FIGURA 5.16 – Tupla CLA_{INI} ____ 71
FIGURA 5.17 – Tupla de Informações do Método ____ 73
FIGURA 5.18 – Tupla de Descarga da Classe ____ 73
FIGURA 5.19 – Tupla de Criação de Objeto ____ 74
FIGURA 5.20 – Tupla de Destruição do Objeto ____ 74
FIGURA 5.21 – Tupla de Movimentação de Objetos ____ 74
FIGURA 5.22 – Tupla de Invocação do método ____ 75
FIGURA 5.23 – Tupla de Finalização da Invocação de Método ____ 75
FIGURA 5.24 – Ciclo de Vida da *Thread* ____ 76
FIGURA 5.25 – Tupla MON_{ER} ____ 78
FIGURA 5.26 – Tupla MON_{ED} ____ 78
FIGURA 5.27 – Tupla MON_{EX} ____ 78
FIGURA 5.28 – Tupla MON_{WAT} ____ 79
FIGURA 5.29 – Tupla MON_{WD} ____ 79
FIGURA 5.30 – Layout do Arquivo de Traço Detalhado ____ 80
FIGURA 5.31 – Exemplo de Chamada RMI ____ 81
FIGURA 5.32 – Tupla de Invocação Remota de Método ____ 82
FIGURA 5.33 – Tupla de Resposta a uma chamada RMI ____ 82
FIGURA 5.34 – Exemplo de Traço de Chamada RMI ____ 83

FIGURA 5.35 – Layout do Arquivo de Traço de Comunicação ___ 84
FIGURA 6.1 – Informações relacionadas a comunicação ___ 88
FIGURA 6.2 – Formato do Arquivo de Traço Global ___ 91
FIGURA 7.1 – Hierarquia de Tipos Simplificada ___ 94
FIGURA 7.2 – Integração do DOMonitor ao Pajé ___ 95
FIGURA 7.3 – Hierarquia de tipos para visualizar aplicações distribuídas Java ___ 95
FIGURA 7.4 – Visualização de um programa Ahapascan pelo Pajé [KER 2000b] ___ 97
FIGURA 7.5 – Componentes do Ambiente ISAM ___ 99
FIGURA 7.6 – Arquitetura ISAM ___ 100
FIGURA 7.7 – Organização do Escalonamento ___ 103
FIGURA 7.8 – DOMonitor na Arquitetura ISAM ___ 104
FIGURA 8.1 – Interação entre os componentes de monitoração ___ 107
FIGURA 8.2 – Formato do Arquivo de Configuração do DOProf ___ 108
FIGURA 8.3 – Definição do tipo `Registro_Classe` ___ 109
FIGURA 8.4 – Definição do `Registro_Metodo` ___ 109

Lista de Tabelas

TABELA 5.1 – Eventos de Definição e Indefinição de cada tipo de ID 69
TABELA 5.2 – Tabela Hash de Classes 72
TABELA 5.3 – Tabela Hash de Métodos 72
TABELA 5.4 – Relação das Tuplas locais 80
TABELA 5.5 – Tuplas do Arquivo Remoto 84
TABELA 8.1 – Comparação entre os trabalhos 113

Resumo

A linguagem de programação Java vem sendo uma das escolhidas para a implementação de aplicações compostas por objetos distribuídos. Estas aplicações caracterizam-se por possuir comportamento complexo e, portanto, são mais difíceis de depurar e refinar para obter melhores desempenhos. Considerando a necessidade do desenvolvimento de uma ferramenta de monitoração para o modelo de objetos distribuídos, que colete informações mais detalhadas sobre a execução da aplicação, é apresentado neste trabalho um ambiente de monitoração de aplicações distribuídas escritas em Java, o **DOMonitor**.

Um dos objetivos do DOMonitor é obter o comportamento que a aplicação apresenta durante a execução, possibilitando a detecção de comportamentos equivocados e seu respectivo refinamento. O DOMonitor é voltado para aplicações compostas por objetos distribuídos e caracteriza-se por identificar principalmente: (i) o comportamento dinâmico das *threads*; (ii) a utilização dos métodos de sincronização; e (iii) a comunicação entre os entes distribuídos da aplicação.

O DOMonitor está fundamentado em quatro premissas: (i) ser transparente para o usuário, não exigindo anotações no código fonte; (ii) apresentar uma organização modular, e por isto ser flexível e expansível; (iii) ser portável, não exigindo nenhuma alteração na Maquina Virtual Java; e (iv) operar de forma a garantir a ordem dos eventos previstos pelo programa.

Os dados produzidos pelo DOMonitor podem ser utilizados com diversas finalidades tais como visualização da execução, escalonamento e como suporte à execução de aplicações móveis. Para comprovar esta versatilidade, foi proposta a integração do sistema a dois outros projetos, o Pajé e o ISAM. O projeto ISAM utilizará os dados monitorados para tomadas de decisão durante o curso da execução e o projeto Pajé permite a visualização gráfica das características dinâmicas de uma aplicação Java.

Palavras-Chaves: Monitoração de Aplicações Distribuídas, Monitoração de Programas Java, Visualização, Programação com Objetos Distribuídos Java.

TITLE: "DOMONITOR: A TOOL FOR MONITORING DISTRIBUTED JAVA APPLICATIONS"

Abstract

Java programming language has been a common choice for implementing distributed object applications. These applications' behavior can be classified as somewhat complex, thus they are the most difficult to debug and refine in order to achieve better performance. In such respect, in the present work is presented **DOMonitor:** a monitoring tool for the distributed object paradigm which target is to collect information about Java distributed application executions.

One of the main goals of DOMonitor is to obtain the application execution behavior, in order to accomplish the detection, and consecutively the refinement, of unexpected or erroneous behaviors. DOMonitor is designed for distributed object applications, providing mainly the identification of: (i) threads dynamic behavior; (ii) synchronization methods usage; and (iii) the communication between the application distributed objects.

DOMonitor relies on four premises: (i) be user transparent, avoiding the use of source code annotations; (ii) present a modular organization, thus being expansible and flexible; (iii) be portable, not requiring any Java Virtual Machine alteration; and (iv) operate in a way that preserve the order of program events.

Data produced by DOMonitor can be used with many purposes like execution visualization, scheduling and also as a support for mobile applications execution. This versatility is evidenced by the purpose of DOMonitor system integration with two other projects: Paje and ISAM. ISAM project will make use of the monitoring data for execution time decisions and Paje will implement the graphical visualization of the dynamic behavior of a Java application.

Keywords: Monitoring distributed applications, monitoring java programs, visualization, distributed object programming in Java.

1 Introdução

1.1 Tema

O tema desta dissertação é a monitoração da execução de aplicações distribuídas Java, sendo esta monitoração realizada com o objetivo de obter informações sobre o comportamento da execução da aplicação.

1.2 Motivação

Os últimos anos da computação têm sido marcados por duas tecnologias básicas: Orientação a Objetos e Sistemas Distribuídos. A união destas duas tecnologias deu origem ao que se conhece por **Objetos Distribuídos**. A primeira é relacionada às técnicas de projeto e desenvolvimento de software. A segunda segue lado a lado com as redes de computadores. As noções de objetos enraizadas no princípio de abstração de dados e na metáfora de troca de mensagens são suficientemente flexíveis para se adaptarem às diversas granulosidades de arquiteturas de hardware e software, oferecendo bons fundamentos para os novos desafios da computação distribuída.

A linguagem de programação **Java** oferece várias facilidades para implementação de aplicações paralelas e distribuídas. Java possui suporte para *Multithreaded* e Invocação Remota de Método (RMI – *Remote Method Invocation*) na definição da linguagem. Melhorias recentes nos componentes que integram a plataforma Java (compilador, compilador *Just-in-Time*, máquina virtual) estão efetivamente reduzindo a diferença de desempenho entre aplicações escritas em Java e aplicações escritas em outras linguagens e que se beneficiam de técnicas mais maduras de compilação e execução. Essas melhorias e a portabilidade têm ocasionado um aumento no número de aplicações distribuídas que têm sido implementadas em Java.

No modelo de programação de **Objetos Distribuídos em Java**, um objeto pode estar localizado em qualquer ponto da rede e, apesar de sua localização incerta, pode se comunicar com outros objetos através de uma invocação remota de método de forma muito semelhante às invocações locais. Aliado a isso, a orientação a objetos resolve boa parte dos problemas de reutilização, modularidade e encapsulamento desejados em quaisquer aplicações, principalmente nas de grande porte. No entanto, esse tipo de aplicação possui comportamento mais complexo, o que dificulta o processo de programação, depuração e refinamento. Com isso, alcançar melhores desempenhos nessas aplicações ainda é responsabilidade de programadores experientes [KLE 99].

A combinação de *threads* e comunicação vem sendo utilizada para programar aplicações regulares, mascarar comunicação ou latências de I/O, evitar *deadlocks* de comunicação, explorar paralelismo de memória compartilhada e implementar invocações remotas de métodos [FOS 96]. Programas deste tipo devem tratar com diversas questões, tais como: competição das *thread*s por recursos; atrasos ocasionados pela sincronização; *overhead* da troca de contexto; distribuição da carga; retardos na comunicação devido à rede [JI 98]. Desta forma, os programas distribuídos são mais difíceis de depurar e de refinar devido à existência de várias *threads* de controle e suas dependências.

As ferramentas de **Análise de Desempenho** procuram auxiliar os programadores no processo de desenvolvimento, depuração e refinamento das suas

aplicações. Ao longo da última década foram desenvolvidas diversas ferramentas de análise de desempenho para o modelo de programação paralela e distribuída [DER 99], [MAL 97], [MIL 95], [TSA 95], [ZHA 95], [REE 93]. Em essência, essas ferramentas abordam características como: a interação entre os processos; a quantidade de mensagens (ou dados) trocados entre dois nodos; tempo realizando computação útil e tempo em espera; a influência das operações de sincronização no comportamento do sistema.

O modelo de programação orientado a objeto é fundamentalmente diferente do modelo procedural. Com isso, muitas das informações sobre o comportamento da execução de programas Java são perdidas em ferramentas de desempenho convencionais. Portanto, sem uma ferramenta de monitoração específica para programas Java é difícil obter detalhes do comportamento da execução da aplicação muitas vezes essenciais a fim de melhorá-la para alcançar melhores resultados.

Atualmente, vem crescendo o interesse pela monitoração e visualização de aplicações Java. Ferramentas já desenvolvidas incluem JaViz [KAZ 2000], HyperProf [HYP 2001], ProfileViewer [PRO 2001], JProbe [JPR 2001], OptimizeIt [OPT 2001], Visual Quantify [VIS 2001], Jinsight [JIN 2000] e Hprof [VIS 2000]. O foco principal da maioria dessas ferramentas é na ocupação da memória (alocação de objetos) e na utilização da CPU (chamadas de métodos). Além disso, a maioria caracteriza o comportamento da aplicação de uma forma global, oferecendo métricas que atendem somente algumas classes de aplicações. Deste modo, não permitem uma análise mais detalhada, por exemplo: registrando informações sobre o comportamento de cada *thread* e da interação entre elas.

Das ferramentas citadas anteriormente, a única que suporta o modelo de objetos distribuídos em Java é o JaViz. Com o JaViz é possível monitorar aplicações do tipo cliente/servidor construídas utilizando RMI, no entanto, não é possível identificar os retardos causados devido ao compartilhamento por recursos. Estas deficiências tornam-se um grande empecilho para depuração de aplicações com objetos distribuídos, pois o programador não tem uma visão clara de como os vários objetos que compõe a aplicação estiveram distribuídos entre as máquinas durante a execução e nem os efeitos colaterais causados devido à sincronização.

Considerando a necessidade do desenvolvimento de uma ferramenta de monitoração para o modelo de objetos distribuídos, que colete informações mais detalhadas sobre a execução da aplicação, foi concebido o sistema de monitoração de aplicações distribuídas escritas em Java - **DOMonitor**. Os dados produzidos pelo sistema podem ser utilizados com diversas finalidades tais como visualização da execução, escalonamento e como suporte a execução de aplicações móveis. Para comprovar a versatilidade dos dados produzidos, é proposta a integração do sistema a outros projetos: Pajé [STE 99], [KER 2000a], [KER 2000b] - uma ferramenta de visualização para aplicações paralelas facilmente extensível para suportar outros modelos de programação, ISAM – uma infra-estrutura de suporte às aplicações móveis distribuídas [AUG 2001b], [AUG 2001c], e EXEHDA – um ambiente de execução para aplicações distribuídas [YAM 99], [YAM 2001b].

O tema visualização não é uma área de pesquisa recente. A visualização da execução é uma ferramenta essencial para auxiliar a depuração e o refinamento de aplicações implementadas utilizando um modelo de programação distribuída. As ferramentas de visualização procuram demonstrar graficamente o comportamento que a aplicação apresentou durante uma execução. Esses gráficos são construídos a partir do comportamento da execução obtido por um processo de monitoração, ou seja, a

execução da aplicação é monitorada e os dados obtidos por este processo são utilizados para a construção dos gráficos de visualização.

O projeto Pajé é uma ferramenta de visualização interativa que mostra a execução de aplicações paralelas. Foi projetada para representar uma grande variedade de interações entre *threads*. Um dos destaques do Pajé é a combinação de características como interatividade, escalabilidade e expansibilidade.

Já que aplicações paralelas e distribuídas são executadas por diversas JVM's cooperando através de chamadas entre objetos, é possível escolher a repartição física das JVM's e, portanto, gerenciar eficientemente os recursos oferecidos pela plataforma de execução. Além disso, inúmeras características podem possibilitar o aumento dos recursos disponíveis, superando restrições como localização física dos computadores e heterogeneidade do sistema operacional.

O projeto EXEHDA (*Execution Environment for High Distributed Applications*) tem por objetivo prover suporte para execução de aplicações distribuídas baseadas em Java. O seu escopo compreende mecanismos para escalonamento e alocação remota de objetos. Sua concepção contempla a integração com os projetos Holoparadigma [BAR 2001c] [BAR 2001d] e ISAM.

O projeto ISAM (Infra-estrutura de Suporte às Aplicações Móveis) propõe uma arquitetura de software que simplifica a tarefa de implementação de aplicações móveis distribuídas. O objetivo é conceber um ambiente de desenvolvimento e execução no qual todos os componentes, mesmo os básicos, estão comprometidos com a premissa de elevada adaptabilidade.

1.3 Contexto Local

Ainda como motivação, cabe ressaltar que a contribuição do DOMonitor está inserida nos esforços de pesquisa feitos pelo grupo OPERA do GPPD (Grupo de Processamento Paralelo e Distribuído) do Instituto de Informática da UFRGS. Três trabalhos do grupo já foram citados na motivação, o ISAM, o EXEDHA e o HoloParadigma, sendo que serão comentadas outras características destes ambientes ao longo do texto para tornar mais claro o seu funcionamento e o entendimento de algumas decisões do DOMonitor. Outro trabalho relacionado também terá algumas características exploradas mais adiante no texto. Algumas funcionalidades do trabalho proposto por Silva [SIL 2001] são usadas pelo DOMonitor, como será mais bem detalhado no capítulo 5.

Além dos esforços de pesquisa referenciados no parágrafo anterior, outros ambientes já foram desenvolvidos ou estão em fase final de desenvolvimento no grupo OPERA no contexto de objetos distribuídos.

1.3.1 DOBuilder

O DOBuilder [MAL 2001] é uma ferramenta de programação visual para o desenvolvimento de aplicações com objetos distribuídos em Java. A programação é baseada na manipulação de componentes, com geração de código em Java e execução em ambiente distribuído. No DOBuilder, fica a cargo do programador especificar de maneira visual a estrutura do seu programa e inserir o código textual para a lógica da aplicação, sendo que o ambiente se encarrega do tratamento da distribuição e da comunicação de mais baixo nível. A aplicação é representada como um grafo dirigido,

onde os nodos representam os objetos distribuídos e os arcos indicam os relacionamentos existentes entre esses objetos (figura 1.1). A ferramenta procura aproveitar as melhores características das ferramentas visuais de programação paralela e distribuída e das ferramentas de programação visual em Java (recursos de visualização para a programação concorrente e características de engenharia de *software*, respectivamente).

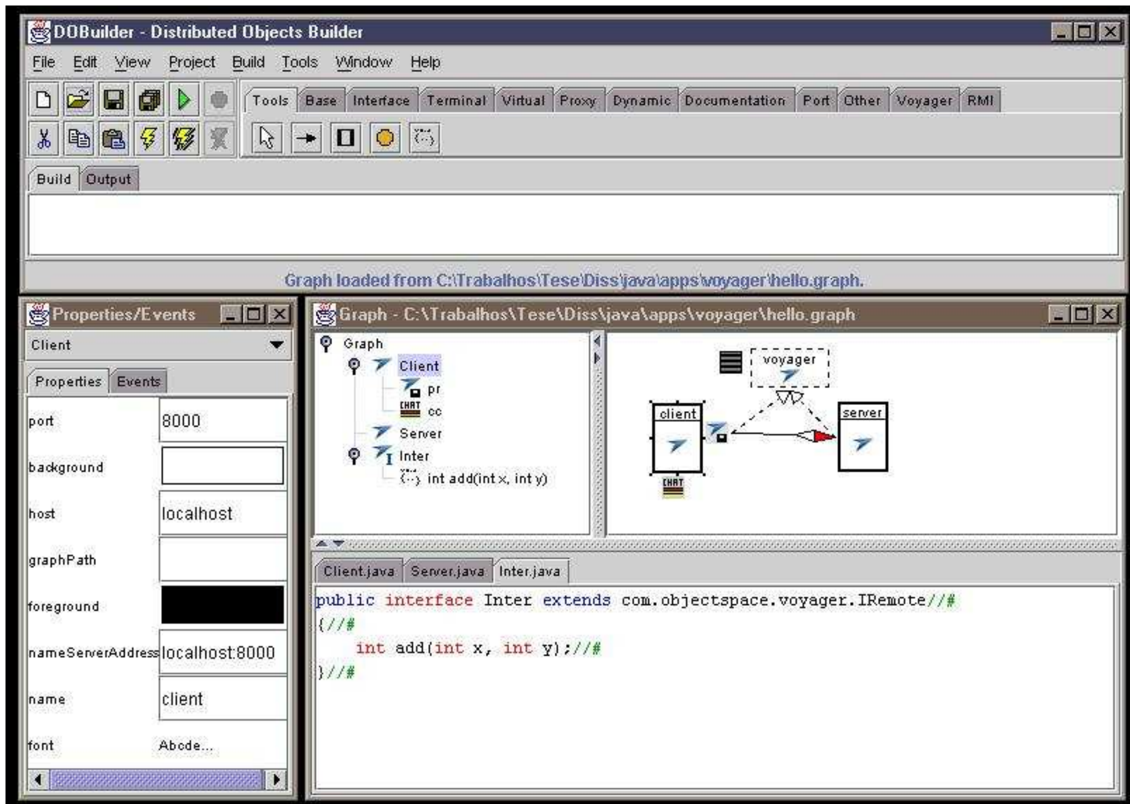


FIGURA 1.1 – Interface de Programação do DOBuilder

Malacarn [MAL 2001] relacionou entre as melhorias que poderiam ser agregadas ao ambiente, características de monitoração e visualização. Uma vez que o DOMonitor é um ambiente de monitoração para aplicações desenvolvidas em Java que não exige alterações no código fonte da aplicação, aplicações desenvolvidas no DOBuilder podem ser monitoradas pelo DOMonitor.

1.3.2 DEPAnalyzer

A análise estática é uma técnica utilizada para obter informações sobre os componentes de um programa e os seus relacionamentos [AZE 99]. Esta técnica obtém informações sobre o comportamento do programa a partir do estudo das ações descritas textualmente no código fonte, as quais representam a semântica ou o significado dos comandos da aplicação. A análise estática prove informações aproximadas, pois estas são aplicáveis a todas as possíveis execuções de um programa, sendo portanto abrangentes. No entanto, mesmo com a possível amplitude dos resultados desta análise, estas informações podem auxiliar na distribuição dos programas.

Azevedo [AZE 2001] apresenta o modelo DEPAnalyzer (*DEPendencies Analyzer*). Este modelo é um analisador estático que visa gerar informações sobre os

relacionamentos entre as classes de um programa Java. O resultado desta análise são as dependências e/ou as invocações entre os conjuntos de objetos que poderão existir dinamicamente. Estas informações podem auxiliar na distribuição dos objetos em uma arquitetura distribuída, visando minimizar as comunicações remotas e, conseqüentemente, aumentar o desempenho da aplicação.

As informações obtidas pelo DEPAnalyser (estáticas) são diferentes das informações obtidas pelo DOMonitor (dinâmicas). Como trabalho futuro, poderiam ser avaliadas as informações obtidas por ambos os ambientes e proposta uma forma de integração.

1.3.3 ReMMos

Nos últimos anos surgiram várias frentes de pesquisas para que as organizações pudessem explorar ao máximo os sistemas distribuídos. Dentre estas linhas de pesquisa, Ferrari [FER 2001] destaca os estudos sobre mobilidade e replicação em sistemas distribuídos. Através da mobilidade, a execução de um componente de software pode ser transferida de uma máquina para outra. A replicação permite que existam diversas cópias de um mesmo componente de software em execução. A replicação de componentes em sistemas distribuídos normalmente é utilizada para torná-los mais confiáveis e seguros ou para aumentar o desempenho da aplicação, uma vez que acessos remotos podem ser evitados através da localidade da réplica.

O ReMMoS (*Replication Model in Mobility Systems*) [FER 2001] é um modelo de replicação em ambientes de objetos distribuídos que permite mobilidade. O objetivo do modelo é prover um ambiente de execução para suporte ao desenvolvimento de aplicações envolvendo mobilidade explicita (a cargo do desenvolvedor) e replicação implícita (transparente). Assim, o programador não necessita preocupar-se com o gerenciamento e a consistência das cópias. Além destas características, este modelo visa propor um desempenho satisfatório para as aplicações do usuário.

Ambientes como o ReMMos podem utilizar técnicas de monitoração para: (i) identificar a carga de um determinado nodo (o que poderia motivar a mobilidade para tentar reduzir a carga de trabalho do nodo e balancear a carga total do sistema); (ii) para identificar o número de invocações remotas realizadas a um determinado objeto, que poderiam estar aumentando demasiadamente o tráfego da rede (o que poderia levar a replicação).

1.4 Objetivos

O objetivo principal deste trabalho é desenvolver uma ferramenta de monitoração para o modelo de programação de objetos distribuídos Java. Dentre os objetivos específicos, destacam-se:

- realizar um estudo sobre características das linguagens orientadas a objetos;
- realizar um estudo sobre as técnicas de monitoração;
- realizar um estudo sobre ferramentas de monitoração de programas orientados a objetos;
- aplicar estes estudos a uma linguagem específica, Java;

- realizar um estudo sobre o ambiente de execução de Java no que diz respeito a monitoração;
- modelar um ambiente de monitoração que obtenha informações sobre a execução de uma aplicação Java em uma arquitetura distribuída;
- registrar e classificar as informações obtidas durante a execução em categorias, para facilitar sua utilização;
- propor a integração dos dados de monitoração com uma ferramenta de visualização existente;
- propor a integração dos dados de monitoração com um ambiente de execução de aplicações móveis distribuídas;
- modelar e implementar um protótipo para validação do modelo proposto.

1.5 Contribuições do Autor

A realização deste trabalho desencadeou o alcance de várias contribuições, dentre as quais destacam-se:

- uma comparação entre os ambientes de monitoração desenvolvidos para a linguagem Java, suas características de funcionamento, seus pontos fortes e suas limitações;
- a concepção de um ambiente de monitoração aplicável ao modelo de programamão orientado a objeto de Java;
- a identificação das informações necessárias para registrar o comportamento dinâmico das *threads*;
- a identificação das informações necessárias para registrar os objetos (em conjunto com sua respectiva classe) criados durante a execução;
- a identificação das informações necessárias para registrar o compartilhamento por recursos, ou seja, a identificação dos objetos compartilhados e das *threads* que ficaram bloqueadas nestes objetos;
- a identificação das informações necessárias para registrar as chamadas de métodos;
- a identificação das informações necessárias para registrar a invocação remota de métodos, ou seja, a comunicação entre objetos que estejam em execução em máquinas virtuais distintas;
- a implementação de uma interface gráfica que permita ao usuário definir as características da monitoração;
- a implementação de um agente de monitoração que obtenha estas informações;
- a disponibilização das informações para diversas finalidades;
- a proposição de integração dos dados de monitoração para visualização e com um ambiente de execução de aplicações móveis distribuídas.

1.6 Estrutura do Texto

Após esta introdução, o capítulo dois apontará algumas características da linguagem de programação Java que vêm tornando esse um dos ambientes escolhidos para o desenvolvimento de aplicações no modelo de programação paralela e distribuída. A seguir, o capítulo três apresenta trabalhos relacionados à monitoração e visualização de aplicações. O capítulo quatro começa a salientar as características do modelo proposto, abordando decisões de projeto e arquitetura do DOMonitor. O capítulo cinco detalha questões relacionadas a forma como é monitorada a aplicação e os dados obtidos pelo processo. O capítulo seis discute sobre as formas de como esses dados podem ser processados de acordo com a finalidade que se pretende. O capítulo sete trata das propostas de integração dos dados monitorados com outras ferramentas. Detalhes de implementação são destacados no capítulo oito. Além disso, neste capítulo é realizada uma comparação entre este trabalho e os outros tipos de soluções apresentados para os mesmos problemas que se pretendem resolver aqui. Por fim, o capítulo nove encerra o trabalho com uma conclusão que resume as principais contribuições obtidas e aponta para trabalhos futuros.

2 Programação Paralela e Distribuída com Java

A linguagem de programação Java [SUN 2001b] apresenta várias características que possibilitam o desenvolvimento de sistemas distribuídos. Características como portabilidade, suporte à programação *multithreading*, suporte à programação distribuída (incluindo RMI e *garbage collection*), tem levado a utilização de Java em sistemas com tamanho e complexidade bem superiores aos pequenos *applets*. No entanto, grandes aplicações possuem problemas de desempenho. Mais especificamente, quanto maior a complexidade da aplicação maior será a chance dela apresentar gargalos de desempenho e mais difícil de serem encontrados esses gargalos. Neste capítulo serão expostas algumas características de Java que são particularmente interessantes em aplicações distribuídas.

2.1 Ambiente Orientado a Objetos

Java é uma linguagem Orientada a Objetos "pura" no sentido que o menor bloco de construção é a classe [NIE 97]. Uma estrutura de dados ou função não pode existir ou ser acessada em tempo de execução a não ser como um elemento de uma definição de classe. Isto resulta em um ambiente de programação com estrutura bem definida no qual todas as operações são mapeadas na representação das classes e nas transações entre elas. Isso traz vantagens para o desenvolvedor de aplicações distribuídas. Distribuir um sistema implementado em Java pode ser pensado como distribuir os objetos de forma racional e estabelecer a ligação entre eles usando o suporte de rede embutido na linguagem.

O suporte de Java para interface abstrata de objetos é outra característica valiosa para o desenvolvimento de sistemas distribuídos. A interface descreve as operações, mensagens e consultas que uma classe de objetos é capaz de servir, sem fornecer nenhuma informação sobre como essas habilidades são implementadas. Se uma classe precisa ser movida para uma máquina remota, então a implementação local da interface pode agir como um atalho, encaminhando as chamadas através da rede para a classe remota. O próprio pacote RMI usa interfaces abstratas para definir os *stubs* locais para objetos remotos.

2.2 Independência de Plataforma

Um desafio apresentado aos desenvolvedores de software é a grande variedade de plataformas de hardware disponível. Usualmente, uma rede possui diferentes tipos de hardware, com arquitetura, sistema operacional e propostas distintas. Java soluciona esse desafio habilitando a criação de programas independentes de plataforma. Um mesmo programa Java pode executar sem modificação em uma grande variedade de computadores e dispositivos. Comparado com programas compilados para hardware e sistema operacional específicos, programas escritos em Java são mais baratos e fáceis de desenvolver, administrar e manter [VEN 2000].

Java é uma linguagem interpretada, uma vez que o compilador Java gera *bytecodes* para uma máquina virtual e não código nativo da máquina. Esse *bytecode* executa na Máquina Virtual Java (JVM - *Java Virtual Machine*), uma arquitetura virtual de hardware implementada em software para a máquina "real" e seu sistema operacional.

Devido aos programas Java serem compilados para *bytecodes* independentes de qualquer arquitetura de hardware, uma aplicação Java pode ser executada em qualquer sistema, desde que exista uma JVM disponível em tal plataforma. Essa característica é particularmente importante em aplicações distribuídas que muitas vezes executam em ambientes heterogêneos. Além disso, uma vez que os elementos do sistema tenham sido especificados usando classes Java e compilados em *bytecode* Java, eles podem mover-se entre os nodos disponíveis sem a necessidade de recompilação. Isso torna possível o balanceamento de carga entre a rede.

2.3 *MultiThreading*

Java é uma linguagem *Multithreaded*, o que significa que pode haver várias *threads* executando ao mesmo tempo. Uma *thread* é um fluxo de controle separado dentro do programa. Conceitualmente, *threads* são similares a processos, exceto que, diferente dos processos, várias *threads* compartilham o mesmo espaço de endereçamento, o que significa que elas podem compartilhar variáveis e métodos. Devido a essas características, as *threads* são mais leves comparadas com os processos.

MultiThreading fornece uma alternativa para uma aplicação manipular diferentes tarefas ao mesmo tempo. Em um determinado momento, podem existir várias *threads* em execução e a menos que elas sejam explicitamente coordenadas, estarão executando métodos sem nenhuma preocupação sobre o que as outras estão fazendo. Problemas poderão surgir quando esses métodos compartilharem os mesmos dados. Se um método estiver alterando o valor de algumas variáveis ao mesmo tempo em que outro método esteja consultando estas variáveis, é possível que a *thread* que está consultando obtenha informações inconsistentes, ou seja, algumas variáveis com o conteúdo antigo e outras com o conteúdo novo. Dependendo da aplicação, esta situação pode causar um erro crítico.

O modificador ***synchronized*** exige que Java obtenha um *lock* para a classe que contém o método antes de executar o método. Somente um método pode obter o *lock* em uma classe em um dado momento, o que significa que somente um método *synchronized* pode estar executando em uma classe por vez. Isto permite que um método altere dados e deixe-os em um estado consistente antes que um método que execute concorrentemente tenha a permissão de acessar os dados. Quando o método terminar, ele libera o *lock* na classe.

A possibilidade de criar e controlar várias *threads* é especialmente importante no desenvolvimento de aplicações distribuídas, já que os agentes distribuídos são mais concorrentes do que os agentes de processos de única máquina, pois executam distribuídos pela rede.

2.4 Suporte à Rede

Aplicações distribuídas podem ser desenvolvidas em termos de dividir uma aplicação em entes individuais de computação que são distribuídos pela rede, mas que trabalham em conjunto para realizar uma determinada tarefa. Algumas motivações para esse tipo de programação são:

- Realizando computação em paralelo a partir do particionamento de uma aplicação complexa em pequenos pedaços permite a solução de problemas maiores sem a necessidade de recorrer a computadores

maiores. Ao invés disso, pode-se usar computadores menores, mais baratos e mais fáceis de serem encontrados;

- Entes de processamento redundantes em várias máquinas da rede podem ser usados por sistemas que precisam de tolerância a falhas.

Chegando-se a um consenso sobre os benefícios da integração entre o modelo de objetos e o de sistemas distribuídos, a questão principal que surge na implementação é como tornar acessíveis os métodos de um objeto que está localizado remotamente. Na maioria das vezes isto é resolvido por meio de um *middleware* que torne as invocações remotas quase transparentes e muito parecidas com as invocações locais. Esse agente gerencia a passagem de parâmetros, valores de retorno e erros que podem ocorrer durante a execução das operações remotas.

Atualmente existem várias arquiteturas de agentes de invocações (ou ORBs - *Object Request Broker*) disponíveis para Java, sendo os mais conhecidos e usados o CORBA da OMG [OMG 99], DCOM da Microsoft [MIC 2000], Java RMI da Sun [SUN 2001c] e Voyager da ObjectSpace [GLA 99]. RMI é totalmente integrado a API Java, sendo independente de processador por natureza, mas limitado a programas escritos em Java. RMI é uma ótima solução para realizar comunicação entre programas Java de diferentes máquinas [JUR 2000]. No entanto, se for necessário conectar-se com programas escritos em outras linguagens, deve-se utilizar outra tecnologia tal como CORBA.

Embora o conceito de RMI possibilite a criação de "visões" de todos os objetos de um determinado sistema a fim de comunicarem-se com os demais, RMI é normalmente utilizado em ambientes cliente/servidor tradicionais. Um único servidor de aplicação recebe conexões e requisições de vários clientes. RMI é simplesmente o mecanismo pelo qual o cliente e o servidor se comunicam. Os objetivos de RMI são:

- Integrar o modelo de objetos distribuídos em Java sem interferir na linguagem ou modelo de objetos existente;

- Tornar a interação com um objeto remoto quase tão fácil quanto a interação com um objeto local;

Com isso, é possível acessar objetos remotos da mesma forma como se acessaria um objeto local (atribuição de variáveis, passagem de argumentos, ...) e invocar métodos de objetos remotos da mesma forma que são realizadas as chamadas locais. Além disso, RMI inclui mecanismos sofisticados que permitem às invocações de métodos em objetos remotos passar objetos inteiros ou partes de objetos tanto por valor como por referência. Adicionalmente, foram definidas exceções adicionais para tratar erros da rede que podem ocorrer enquanto uma operação remota estiver em andamento [LEM 2001].

Para garantir compatibilidade com programas Java e implementações existentes e para tornar o processo tão transparente ao programador quanto possível, a comunicação entre objeto local (cliente) e objeto remoto (servidor) é implementada em uma série de camadas como mostrado na figura 2.1.

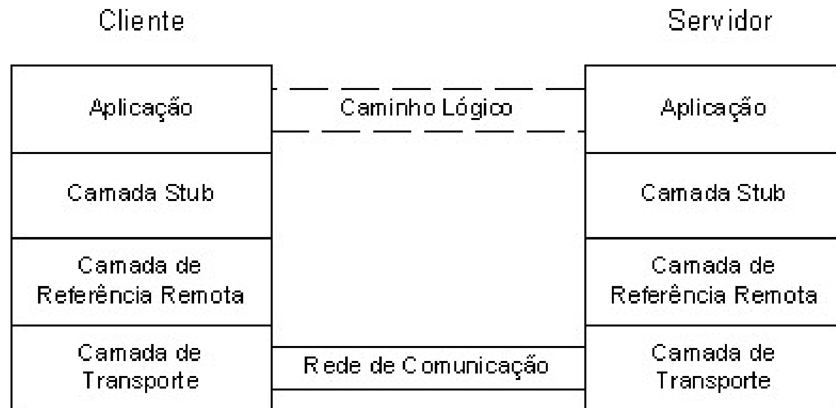


FIGURA 2.1 – Camadas RMI

Para o programador, parece que o cliente comunica-se diretamente com o servidor. Na realidade, o programa cliente invoca um método remoto usando o *stub*. O *stub* é um objeto especial que implementa as interfaces do objeto remoto, isto significa que, o *stub* possui métodos correspondentes a todos os métodos que o servidor exporta. De fato, o cliente imagina que está invocando um método no objeto remoto, mas está invocando um método equivalente no *stub*. *Stubs* são usados na máquina virtual dos clientes ao invés dos objetos e métodos reais que ficam no servidor. Quando o *stub* é invocado, ele passa a invocação para a Camada de Referência Remota.

A Camada de Referência Remota utiliza um protocolo de referência remota específico que é independente do *stub* do cliente e do *skeleton* do servidor. A camada é responsável por entender o que uma referência remota significa. Em essência, a Camada de Referência Remota traduz a referência local ao *stub* em uma referência remota para o objeto no servidor e então passa os dados para a Camada de Transporte.

A Camada de Transporte manipula a real movimentação dos dados pela rede. Ela fornece configuração de conexão, gerenciamento de conexão e envio dos objetos remotos. No lado do servidor, a Camada de Transporte espera por pedidos de conexão ou por invocações. Quando recebe uma invocação, repassa a invocação para a Camada de Referência Remota no servidor. Esta converte as referências remotas enviadas pelo cliente em referências para a máquina virtual local. Então passa a requisição para o *skeleton*. O *skeleton* lê os argumentos e passa os dados para o programa servidor, o qual faz realmente a chamada do método. Se o método retornar um valor, este valor desce pelas camadas *skeleton*, referência remota e transporte no lado do servidor, atravessa a rede e então sobe pelas camadas de transporte, referência remota e *stub* no lado do cliente. Logicamente os dados fluem horizontalmente (cliente-servidor e servidor-cliente), mas o real fluxo dos dados é vertical.

Outra característica importante é a robustez de Java. Assim como programas desenvolvidos por programadores inescrupulosos, programas mal escritos por programadores bem intencionados podem destruir informações, monopolizar a utilização da CPU, ou levar o sistema a travar. A arquitetura de Java garante um certo nível de robustez ao programa prevenindo certos tipos de *bugs*, tal como corrupção de memória. Esta arquitetura garante que programas descarregados (*downloaded*) não irão inadvertidamente (ou intencionalmente) travar. Devido a Java evitar que vários tipos de *bugs* ocorram, programadores Java não precisam gastar tempo tentando encontrá-los e corrigi-los.

3 Monitoração e Visualização de Programas

A Monitoração e a Visualização da execução não são áreas de pesquisa recente. Várias ferramentas foram desenvolvidas para o modelo de programação paralela ou distribuída. Este capítulo apresenta questões relacionadas a estes temas, bem como características de alguns trabalhos relacionados.

3.1 Conceitos Relacionados à Monitoração de Programas

A programação paralela e distribuída é reconhecidamente mais complexa do que a programação seqüencial. Várias são as razões para que isso aconteça: diversas tarefas independentes, comunicação e sincronização. Essas dificuldades prejudicam todo o ciclo de desenvolvimento dos programas, refletindo negativamente na qualidade das aplicações concorrentes. Para amenizar esse problema, diversas ferramentas de apoio ao desenvolvimento têm surgido com características próprias para resolver diferentes tipos de problemas.

A depuração do desempenho é uma fase importante no ciclo de desenvolvimento de programas paralelos, uma vez que o principal motivo para se utilizar sistemas paralelos é a obtenção de alto desempenho. A depuração do desempenho geralmente inclui várias fases: monitoramento para obter dados de desempenho, análise de dados e apresentação dos mesmos ao programador, geralmente por ferramentas de visualização [STE 2001b].

O processo de monitoração consiste em obter informações relacionadas ao comportamento da execução de uma aplicação. As propostas de monitoração podem ser classificadas em **software**, **hardware** e **híbridas**. A classificação das propostas é como um espectro, com monitoração por hardware em uma extremidade e monitoração por software na outra. A principal diferença entre as duas é que as propostas de **hardware** separam a tarefa de monitoração da carga de trabalho do ambiente que está sendo monitorado, enquanto as proposta de **software** acabam compartilhando o poder computacional que seria utilizado apenas pela aplicação. Outras propostas podem estar entre as de hardware e software. Estas são chamadas de **monitoração híbrida** e contam com um suporte limitado de hardware dedicado [MEN 94].

A primeira fonte de perturbação causada pelo processo de monitoração por software é a execução de instruções adicionais. Entretanto, perturbações secundárias podem ser provenientes da ausência de otimizações de compilação e de *overhead* no sistema operacional [MAL 92]. O grau de intrusão que pode ser tolerado depende da natureza da aplicação e do resultado desejado da monitoração. Para determinar este nível, é importante entender que um *Monitor Intrusivo* perturba a execução de uma aplicação distribuída alterando de maneira arbitrária a duração dos eventos na aplicação monitorada. A alteração na duração dos eventos pode:

(a) levar a resultados incorretos;

(b) criar (ou mascarar) situações de *deadlock* quando a ordem dos eventos nas diferentes *threads* é afetada;

(c) aumentar drasticamente o tempo da execução da aplicação monitorada;

(d) tornar o entendimento de um programa distribuído uma tarefa mais difícil.

É muito importante analisar as características da aplicação que podem afetar de forma mais significativa o comportamento da aplicação como um todo, ou seja, identificar os elementos críticos do sistema, aqueles que possuem maior influência sobre o comportamento geral da aplicação. Em se tratando de aplicações paralelas e distribuídas, o principal elemento é a comunicação. A velocidade da troca de informações entre duas máquinas é muito baixa se comparada com a velocidade de processamento das máquinas. Enquanto uma mensagem é transferida, a unidade de processamento pode executar milhões de instruções. Outro elemento importante é a utilização dos recursos compartilhados pelos diversos componentes da aplicação.

Sistemas desenvolvidos com objetos distribuídos são conhecidamente mais complexos. Estes sistemas têm comportamento mais dinâmico que os sistemas procedurais tradicionais. As ferramentas de avaliação de desempenho para o modelo de programação orientado a objetos devem permitir que o usuário possa observar as mudanças na estrutura da aplicação bem como o comportamento dos seus componentes. Além disso, devem ser capazes de detectar o desbalanceamento na configuração da aplicação e sobrecarga na interação entre objetos, auxiliando desta forma para um refinamento adequado.

O restante deste capítulo descreve alguns ambientes de avaliação de desempenho de programas paralelos e distribuídos descritos na literatura, procurando destacar as soluções utilizadas e problemas existentes.

3.2 Pablo

Pablo é um conjunto de ferramentas para instrumentar programas paralelos, coletar traços de execução e analisar os resultados [REE 93]. A operação entre os vários módulos do Pablo é baseada no Pablo *Self-Defining Data Format* (SDDF), ou seja, os módulos do Pablo utilizam um formato padronizado no qual os dados são representados [AYD 97]. O padrão SDDF fornece a generalidade e a expansibilidade necessárias para representar um conjunto de métricas de desempenho e pontos de avaliação. Por exemplo, os componentes (módulos) de monitoração do Pablo produzem como saída arquivos de traço no formato SDDF e os componentes de análise recebem esses arquivos como entrada. É possível converter outros formatos para SDDF, de tal forma que arquivos de traços produzidos por outras ferramentas de monitoração possam ser analisadas usando as Ferramentas de Análise do Pablo.

Os programas que são monitorados pelo Pablo precisam ser **instrumentados**. A Biblioteca de Instrumentação consiste de uma biblioteca de traço com extensões para: traço de procedimentos, traço de estruturas de repetição, traço de troca de mensagens, traço de I/O e traço de MPI. A instrumentação para gerar traço de procedimentos, de laços de repetição e de I/O deve ser feita manualmente através da inserção de chamadas para as rotinas da biblioteca no código fonte da aplicação. Já para instrumentar as trocas de mensagens usando MPI, basta conectar a aplicação a uma biblioteca específica e nenhuma modificação no código fonte é necessária.

Depois de realizadas essas modificações no código fonte da aplicação, a mesma deve ser recompilada e link-editada com as bibliotecas de traço para estar apta a ser executada. Durante a execução da aplicação é gerado o **Arquivo de Traço**, contendo informações sobre a execução. No caso de programas paralelos com MPI, cada processo MPI produz um arquivo de traço. Os arquivos de traço dos processos devem ser agrupados utilizando um utilitário SDDF, o *MergePabloTraces*.

A análise do arquivo de traço é feita depois de finalizada a execução, ou seja, a análise sobre o traço de eventos é *post-mortem*. Essa análise é realizada com a Interface de Análise do Pablo, que fornece quatro tipos de módulos com os quais gráficos podem ser construídos: (i) Análise de Dados; (ii) Apresentação de Dados; (iii) Importação de Arquivo de Traço; e (iv) Exportação de Arquivo de Traço. O módulo de Análise dos Dados inclui transformações matemáticas simples, tais como: contador, acumulador, média, máximo, mínimo, funções trigonométricas, etc. O módulo de Apresentação dos Dados inclui: Gráficos de Barras, *Bubble Charts*, *Strip Charts*, *Contour Plots*, *Interval Plots*, Diagramas de Kiviat, *Scatter Plots* 2-D e 3-D, *Matrix Displays*, *Pie Charts* e *Polar Plots*.

O sistema de visualização do Pablo foi projetado para mostrar graficamente certos comportamentos estatísticos de um programa paralelo durante sua execução. Adicionalmente, é possível especificar transformações e apresentações de dados, conectando graficamente módulos de análise e visualização de dados e selecionando interativamente quais registros de dados devem ser processados por cada módulo. O Pablo também suporta o desenvolvimento e a adição de novos módulos de análise de dados ao seu ambiente de análise.

O *SvPablo (Source View Pablo)* é uma proposta diferente na qual, os componentes de instrumentação, biblioteca de traço e análise foram combinados em um único componente. O SvPablo é um ambiente gráfico para instrumentação do código fonte da aplicação e para procura dinâmica por dados de desempenho [DER 99]. O SvPablo suporta instrumentação interativa ou automatizada, dependendo da linguagem. A instrumentação interativa possibilita um controle detalhado, permitindo aos usuários especificar os pontos nos quais os dados devem ser capturados. Esse tipo de instrumentação é aplicado a C, Fortran 77 e Fortran 90. Instrumentação automatizada consiste no compilador ou no ambiente de execução inserir código de instrumentação no código da aplicação e é implementado apenas para HPF (*High Performance Fortran*).

3.3 Paradyn

O Paradyn [MIL 95] é uma ferramenta para avaliação de programas paralelos e distribuídos de longa duração e que automatiza grande parte da procura por gargalos de desempenho. A ferramenta é voltada para aplicações de grande porte e, por isso, os mecanismos de instrumentação, controle de programa e visualização dos dados devem considerar este grande número de componentes do programa.

O Paradyn procura diminuir o *overhead* através de uma técnica que instrumenta a aplicação dinamicamente e controla automaticamente essa instrumentação com o intuito de encontrar problemas de desempenho. Essa tecnologia permite que código de instrumentação seja inserido, alterado e removido de uma aplicação em execução. A instrumentação é inserida dinamicamente no arquivo binário (executável) do programa. Informações de desempenho são analisadas *on-line* para ajustar a quantidade de informações que devem ser coletadas: mais dados são coletados onde um problema de desempenho é esperado.

O processo tem início pela procura de problemas de mais alto nível para o programa inteiro, tais como: totais de bloqueios de sincronização, bloqueios de I/O ou retardos de memória. Somente uma quantidade pequena de instrumentação é inserida para encontrar esses problemas. Uma vez que um problema genérico foi encontrado, instrumentação é inserida para encontrar causas mais específicas. Nenhuma instrumentação detalhada é incluída para classes de problemas que não existem.

O módulo chamado **Consultor de Desempenho** serve para indicar o lugar onde deve ser inserida instrumentação. Ele possui conhecimento sobre os gargalos de desempenho e a estrutura do programa, de tal forma que pode associar gargalos com causas e partes específicas do programa. A obtenção das informações de tempo e de quantidade é baseada em dois tipos de objetos: os **contadores** são inteiros que indicam a freqüência (número de ocorrências) de algum evento e os **temporizadores** medem a quantidade de tempo gasto executando uma determinada tarefa.

O Paradyn fornece duas abstrações básicas para representação dos dados: *Metric* (métricas) e *Focus* (foco). Métricas apresentam algumas características do desempenho de um programa paralelo em função da variação do tempo. Exemplos incluem: utilização de CPU, utilização de memória, e freqüência de chamadas de procedimento. Um Foco é uma especificação de uma parte do programa em execução, expressa em termos de recursos do programa. Tipos de recursos típicos incluem objetos de sincronização, objetos de código fonte (procedimentos, blocos básicos, processos, processadores e discos). Recursos são separados em várias hierarquias diferentes, cada uma representando uma classe de objetos de uma aplicação paralela.

A arquitetura do Paradyn pode ser dividida em duas partes: a interface gráfica do usuário e os *daemons* da ferramenta (figura 3.1). A interface de usuário permite a visualização de desempenho, utiliza o Consultor de Performance para encontrar gargalos, inicia e finaliza a aplicação, e monitora o estado da aplicação. Os *daemons* da ferramenta monitoram e instrumentam os processos da aplicação. A visualização do Paradyn é durante a execução e não *post-mortem* como na maioria das ferramentas.

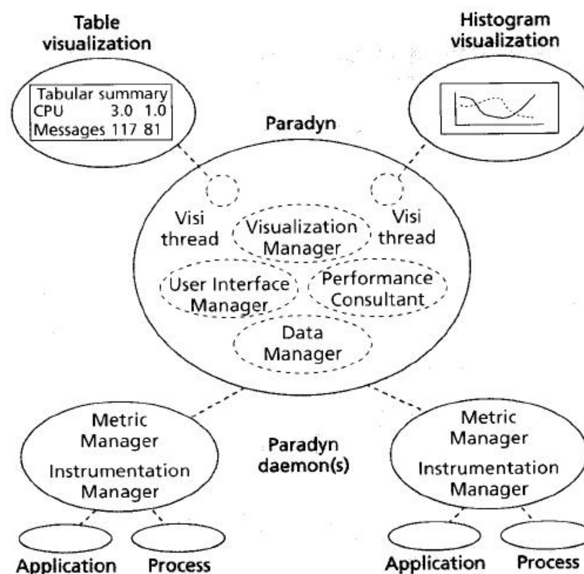


FIGURA 3.1 – Visão da Arquitetura do Paradyn

O trabalho proposto por Xu [XU 99] apresenta uma extensão ao Paradyn para instrumentar dinamicamente aplicações *multithreaded*. Xu [XU 99] aponta alguns desafios para construção de ferramentas de monitoração para aplicações *multithreaded*: (i) associar dados de desempenho com as *threads* individuais e realizar isso com um *overhead* aceitável; (ii) manipular o assincronismo que pode ocorrer em programas *multithreaded*, tal como preempção de *threads*; (iii) minimizar o custo de instrumentação quando se realizam medições por *thread* de métricas baseadas em

tempo. Mais detalhes sobre estas questões e a solução adotada nesta proposta foram descritos por Xu [XU 99].

3.4 Pajé

O Pajé é uma ferramenta de visualização interativa que mostra a execução de aplicações paralelas. O Pajé foi projetado para permitir ao programador a visualização da execução de programas paralelos compostos por um grande número de *threads* que se comunicam entre si. É possível representar uma grande variedade de interações entre as *threads*, sendo que estas podem ter vários tempos de execução em cada nodo de um sistema paralelo de memória distribuída. Com o intuito de ser extensível, a arquitetura do sistema é baseada em componentes modulares os quais comunicam-se utilizando um fluxo de dados pré-definido.

O Pajé foi inicialmente desenvolvido para possibilitar a visualização de programas desenvolvidos para o Athapascan [BRI 97]. O modelo de programação do Athapascan foi projetado para arquiteturas paralelas compostas por nodos multiprocessados com memória compartilhada e conectados através de uma rede de comunicação. O Athapascan explora dois níveis de paralelismo, o paralelismo entre nodos e o paralelismo dentro de cada um dos nodos. As principais funcionalidades do Athapascan são: criação dinâmica de *threads* local ou remotamente; compartilhamento de espaço de endereçamento entre *threads* de um mesmo nodo; comunicação por troca de mensagens entre as *threads* remotas.

Um dos maiores destaques do Pajé é combinar características como interatividade, escalabilidade e expansibilidade. A interatividade permite inspecionar todos os objetos mostrados na tela e voltar a momentos anteriores no tempo, visualizando o estado dos objetos em um instante passado. A escalabilidade é a habilidade de tratar com um grande número de *threads*. A expansibilidade é uma característica importante por lidar com a evolução da interface de programação paralela e técnicas de visualização. Além disso, a expansibilidade possibilita estender o ambiente com novas funcionalidades: o processamento de novos tipos de traços, a adição de novos gráficos, a visualização de novos modelos de programação.

Conforme citado na motivação deste trabalho, pretende-se utilizar as informações sobre o comportamento da execução obtidas com a monitoração em um processo de visualização que permita ao programador verificar a execução da sua aplicação. A ferramenta escolhida para essa integração monitoração-visualização foi o Pajé. Por esta razão, a descrição do Pajé nesta seção foi resumida, sendo que maiores detalhes da ferramenta serão descritos no capítulo 7, onde é apresentada a proposta de integração dos dados monitorados com o Pajé.

3.5 HProf

O HProf é um agente de monitoração disponibilizado com o pacote J2SE (*Java 2 Platform Standard Edition*) [SUN 2001d]. O HProf obtém os dados da execução em conjunto com a máquina virtual java através de uma interface, o JVMPI (*Java Virtual Machine Profiler Interface*). A monitoração é limitada apenas às invocações de métodos locais, ou seja, não são registradas informações relacionadas à comunicação entre objetos que executem em JVM's distintas. As informações da execução são coletadas e armazenadas em um arquivo de traço. A monitoração é

realizada utilizando uma combinação entre as técnicas de traço de eventos e amostragem. Caracteriza-se por:

- Contar as invocações a métodos locais tanto individualmente, como por par chamador-chamado;
- Fornecer informações sobre a média do tempo de execução dos métodos.

O HProf não possui suporte de visualização próprio. O arquivo de traço gerado pode ser analisado por ferramentas de visualização desenvolvidas especificamente para o HProf, tais como: **ProfileViewer** [PRO 2001] e **HyperProf** [HYP 2001]. O maior enfoque dos gráficos é na relação método chamador – método chamado.

HyperProf – possui um gráfico de visualização que demonstra um arquivo de traço como uma árvore hiperbólica, permitindo ao usuário verificar o tempo envolvido em cada método, os métodos chamados por cada método e assim por diante. As informações contempladas referem-se a chamadas de métodos locais, com ênfase principal para a relação método chamador -> método chamado.

ProfileViewer – também demonstra os relacionamentos método chamador-chamado com informações de tempo. Entretanto, é difícil determinar se um método tem tempo de execução alto ou baixo ou se o tempo de execução de um método varia baseado no contexto de sua chamada e nos parâmetros que recebe.

3.6 OptimizeIt

OptimizeIt é uma ferramenta comercial de análise de desempenho destinada a aplicações desenvolvidas em Java [OPT 2001]. Caracteriza-se por facilitar a localização de código ineficiente e utilização incorreta de memória. Permite aos usuários detectar a utilização de memória e de CPU de qualquer componente Java, mesmo quando o código fonte não estiver disponível. Além disso, possui suporte para visualização de *threads* e de detalhes sobre o mecanismo de coleta de lixo. No entanto, não tem a capacidade de traçar comunicação entre objetos executados por JVM’s distintas.

A ferramenta disponibiliza uma variedade de gráficos e relatórios com objetivo de mostrar aos usuários exatamente o que está acontecendo no código da aplicação. Existe ainda a possibilidade de visualizar o código fonte para estudar partes relevantes do mesmo. Por exemplo, quando se identifica um problema de desempenho na alocação de objetos, é possível abrir uma janela e verificar qual parte do código está gerando o problema.

A visualização dos dados de monitoração pode ser feita *on-line* ou *off-line*. A visualização *off-line* reduz o *overhead* introduzido na execução da aplicação monitorada. Além disso, permite, que a partir de um conjunto de arquivos de traço, seja possível fazer uma comparação entre o processo de desenvolvimento da aplicação.

O OptimizeIt foi escrito em Java e utiliza a versão original da máquina virtual, ou seja, não há a necessidade de utilizar uma JVM modificada ou ter de modificar o código fonte da aplicação. OptimizeIt possui dois componentes principais:

- o **módulo de monitoração**, que registra as atividades na máquina virtual Java e as envia instanteneamente a interface do usuário;

- a **interface de usuário**, que mostra os dados de desempenho obtidos e os controles para refinar a aplicação.

As características principais do OptimizeIt são:

- Começar e parar a monitoração durante a execução da aplicação;
- Suspender e reiniciar a execução do programa monitorado;
- Selecionar apenas classes de interesse;
- Fornecer uma visualização *on-line* de todas as classes usadas no programa e das instâncias alocadas;
- Destacar automaticamente linhas de código que alocam instâncias de objetos;
- Representar graficamente a atividade das *threads* para um período de amostragem;
- Remover métodos mais rápidos da visualização através de filtros.

A ferramenta caracteriza-se por destacar dois problemas principais: a utilização da memória e a utilização da CPU. O ***Profiler* de Memória** (figura 3.2) fornece informações sobre os objetos alocados durante a execução do programa Java e permite visualizar qual método é responsável pela alocação de cada objeto. Além disso, auxilia na identificação de objetos não liberados pelo coletor de lixo. Com o coletor de lixo de Java, não é tão necessário que os programadores tenham que cuidar dos objetos alocados e liberar explicitamente a memória quando ela não for mais necessária. No entanto, pode acontecer que alguns objetos mantenham referências a objetos que não sejam realmente necessários a nenhum deles.

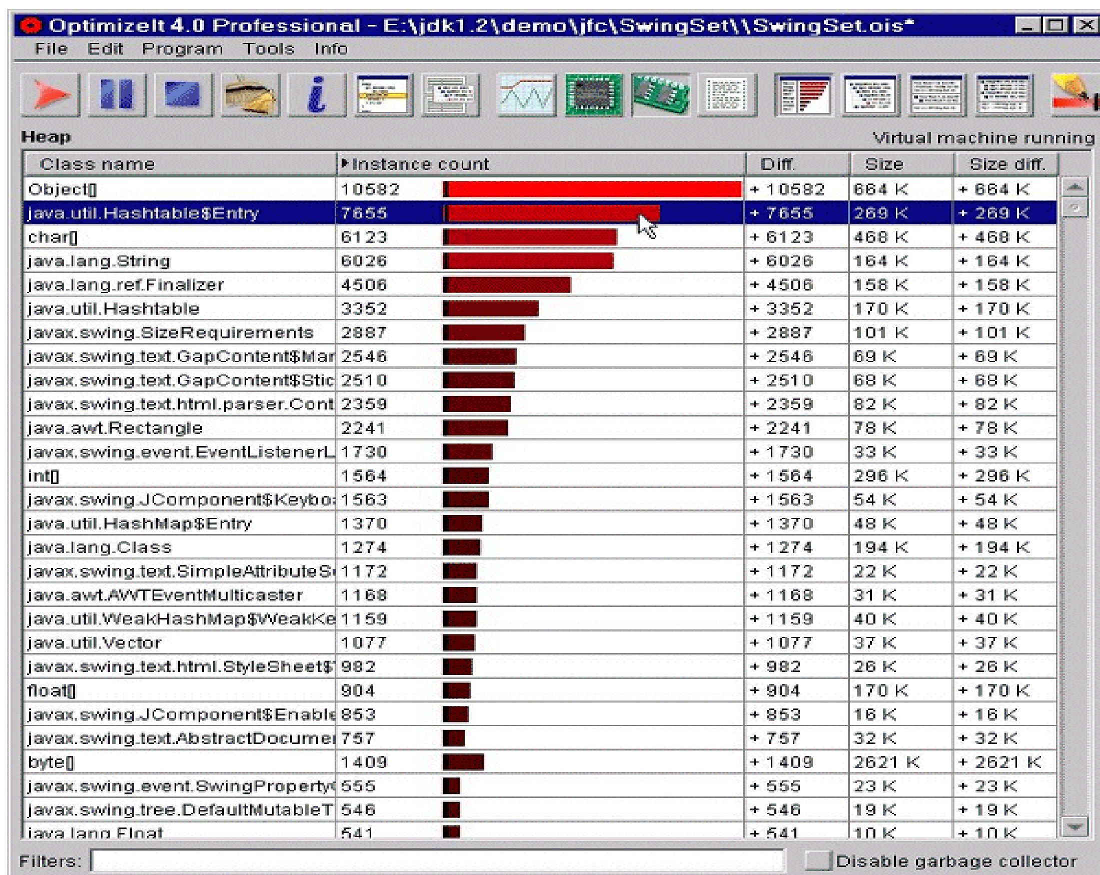


FIGURA 3.2 – Profiler de Memória do OptimizeIt

O ***Profiler* de CPU** auxilia na identificação dos métodos que consomem mais tempo durante a execução do programa. O OptimizeIt fornece uma descrição por *thread* do tempo consumido por cada método e a quantidade de ciclos de CPU utilizadas por ele. A identificação dos métodos onde são consumidos muitos ciclos de CPU, serve como ponto de partida para encontrar problemas de desempenho no programa. Isso auxilia o entendimento do que deve ser alterado no programa para melhorar o seu desempenho.

3.7 Jinsight

O Jinsight [JIN 2001] foi projetado especificamente para monitorar e visualizar o comportamento da execução de programas orientados a objetos desenvolvidos em Java. A ferramenta possui características para auxiliar na procura e solução de deficiências na memória e na procura por gargalos de desempenho em pontos como: criação de objetos, coleta de lixo, seqüência de execução, interação entre *threads*, referências a objetos. No entanto, não tem a capacidade de traçar atividades cliente/servidor entre várias JVM's.

A monitoração no Jinsight é realizada através de uma JVM modificada (jdk1.1.7/1.1.8), ou seja, a monitoração só é possível através da versão disponibilizada em conjunto com a ferramenta. O traço da execução é visualizado através de um visualizador, completamente desenvolvido em Java, que fornece diversas opções de gráficos. As várias visões permitem visualizar os elementos de um programa Java - os objetos, métodos, seqüência de chamadas, referências à objetos e threads - por vários ângulos. Além disso, as visões permitem visualizar como esses elementos agiram juntos na execução do programa.

Dentre os diversos gráficos (visões) que a ferramenta disponibiliza, há um particularmente interessante que permite a visualização do programa em termos das *threads* em execução (figura 3.3).

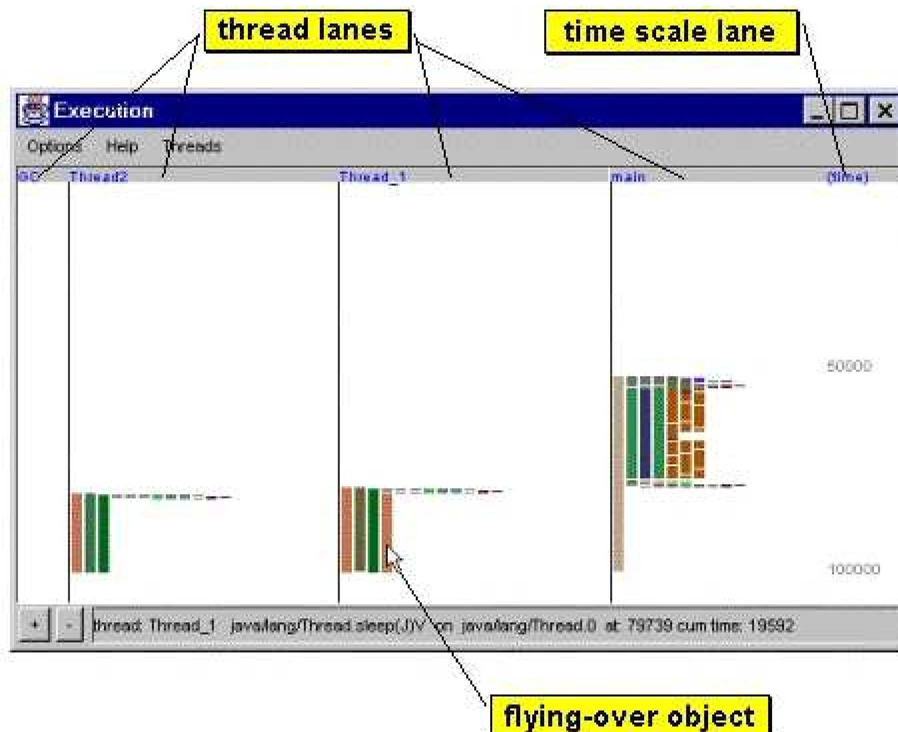


FIGURA 3.3 – Diagrama de Execução da Thread

Cada uma das *threads* é representada por uma linha vertical, identificada na sua parte superior com o nome da *thread*. A ordem cronológica da execução é observada de forma vertical, de cima para baixo. Na horizontal, da esquerda para a direita, é possível verificar a pilha de execução de cada uma das *threads*.

O Jinsight possui capacidades adicionais de extração de comportamento que permitem tratar com arquivos de traços de tamanho elevado gerados pela monitoração de programas complexos do mundo-real. A ferramenta apresenta os padrões de comportamento da execução de forma simples e compacta. Isso é feito através de uma técnica de **extração de comportamento** que obtém informações que muitas vezes são redundantes e então as reduz para uma forma fundamental. Isso permite examinar grandes áreas do espaço da execução sem preocupar-se com todos os objetos ou chamadas de métodos. A extração de comportamento simplifica a análise da execução.

O *Reference Pattern View*, mostrado na figura 3.4, usa extração de padrões e técnicas de visualização para auxiliar na exploração das estruturas de dados do programa. Isso possibilita uma rápida compreensão sobre o padrão da interconexão entre grande número de objetos e auxilia a encontrar objetos que estão mantendo referências a objetos impedindo-os de serem excluídos pelo coletor de lixo.

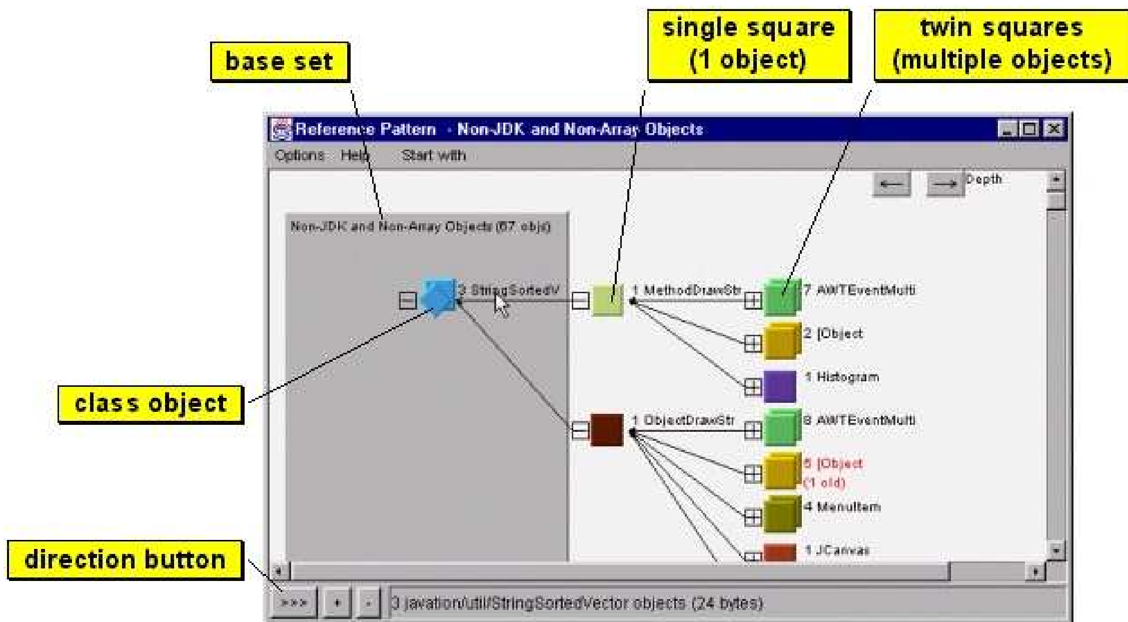


FIGURA 3.4 – Diagrama de Referência de Objetos

3.8 JaViz

JaViz [KAZ 2000] é um ambiente para análise de desempenho de aplicações distribuídas Java. Entre os seus objetivos está o de auxiliar o programador desse tipo aplicações na identificação de problemas como: métodos ineficientes e pontos do programa onde ocorre um número excessivo de invocações remotas de métodos. Para isso, o JaViz incorpora ferramentas de Visualização e Análise Estatística.

A execução de um programa Java é apresentada na forma de uma árvore de execução com os relacionamentos *caller-callee* (chamador-chamado) das chamadas de métodos do programa, com os métodos chamadores sendo superiores aos chamados. A árvore de execução representa o grafo de chamadas de um único programa, sendo ele um cliente ou um servidor. Portanto, em um ambiente cliente/servidor com `n` JVM's,

JaViz irá gerar `n` árvores de execução diferentes representando a execução de cada um dos clientes e servidores.

O desenvolvimento da ferramenta baseou-se em algumas exigências de funcionalidade e desempenho. Entre elas, destaca-se:

- a habilidade de registrar o tempo gasto para todas as chamadas de métodos executados na JVM;
- a habilidade de organizar um traço de chamadas de métodos em uma árvore de execução. Esta árvore inclui todas as *threads* do programa, com o método `run()` da *thread* aparecendo como filho do método que invocou essa *thread*.
- a habilidade de construir uma árvore de execução que comporte as invocações de métodos de outras JVM's. A árvore é organizada a partir de uma JVM e inclui todas as chamadas de métodos executadas por outras JVM's;
- a habilidade de construir corretamente cada uma das árvores para as chamadas de métodos mesmo quando algumas chamadas são omitidas na geração do traço por razões de eficiência.
- uma implementação que não interfira excessivamente o desempenho da aplicação. É permitido ao usuário controlar o conjunto de métodos que serão monitorados com o objetivo de diminuir a quantidade de dados coletados.

O JaViz é composto de três componentes principais (figura 3.5): (i) as **JVM's instrumentadas** que foram modificadas para registrar as chamadas de métodos; (ii) um conjunto de ferramentas de **Pós-Processamento** que organizam as invocações remotas de métodos e geram estatísticas; e (iii) uma ferramenta de **Visualização**. As ferramentas de Visualização e Pós-Processamento são escritas em Java, garantindo assim portabilidade

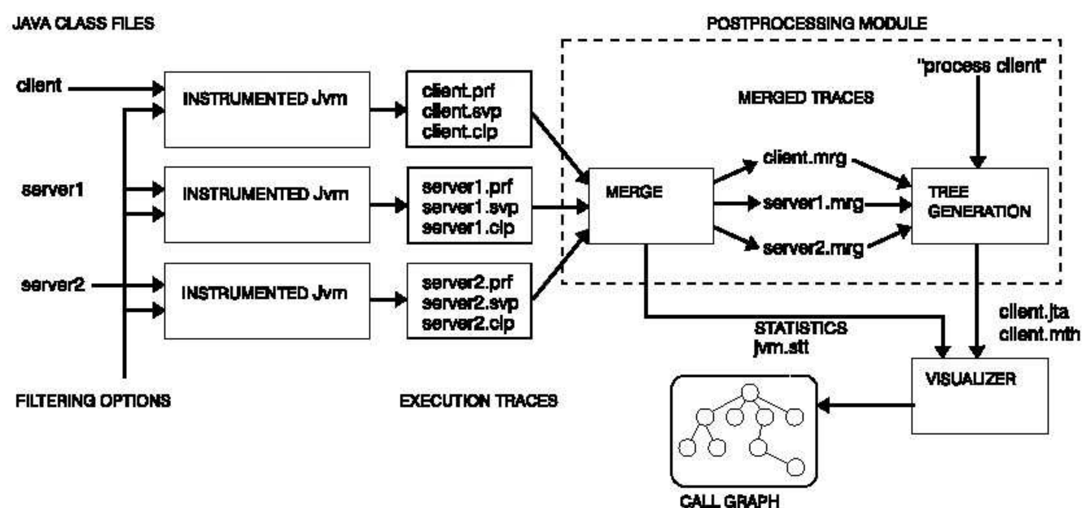


FIGURA 3.5 – Arquitetura de Módulos do JaViz

3.8.1 Coleta das Informações

A coleta das informações é realizada através de uma versão modificada da máquina virtual Java (JVM instrumentada). Quando um programa Java for executado por esta JVM, pode ocorrer a geração de até três arquivos de traço, conforme pode ser visto na figura 3.5. O arquivo **.prf** contém informações sobre o momento da chamada e da conclusão de todos os métodos executados. Os outros dois arquivos registram a interação cliente/servidor, se houver. O arquivo **.clp** contém informações sobre todas as chamadas RMI originadas na JVM, isto é, identifica informações para todos os métodos remotos invocados pela JVM. O arquivo **.svp** registra informações sobre todas as chamadas RMI recebidas, ou seja, todos os métodos remotamente invocados nesta JVM por outras JVM's. Pode-se considerar o arquivo **.clp** como *Cliente Profile* e o arquivo **.svp** como *Server Profile*. Cabe ressaltar que um servidor também pode executar funções do tipo cliente e um cliente também pode atuar como servidor para outras JVM's.

O módulo de geração de traços da JVM foi modificado para registrar todas as invocações de métodos usando *times stamps* para identificar o **momento inicial** e **final** da execução do método, com precisão de microsengundos. Adicionalmente, é armazenado um **identificador de thread** para identificar unicamente a *thread* que está executando o método. A coleta de informações é realizada pela JVM através de uma função que é chamada em todas as invocações de método. O módulo de monitoração cria um novo registro de traço para cada evento de entrada ou saída de método.

Para diminuir o *overhead*, as informações são armazenadas na memória principal e escritas para arquivos externos somente quando o buffer está cheio. Ainda com o objetivo de reduzir a quantidade de operações de I/O com o disco, o JaViz armazena no registro de trace um valor numérico para identificar um método, sendo que esse valor numérico é armazenado em um tabela em conjunto com o nome do método para futura utilização. Uma das formas adicionais de reduzir a quantidade de informações coletadas inclui a utilização de filtros. Os dois filtros disponíveis no JaViz são: (i) a exclusão dos métodos chamados pela biblioteca Java, ou seja, só são registrados os métodos de níveis superiores da biblioteca Java; (ii) especificar a lista de classes para as quais haverá geração de informações..

A coleta das informações relativas às invocações remotas de métodos é um pouco mais complexa. Inicialmente, todos os objetos a serem exportados para uma JVM remota recebem automaticamente um identificador único da JVM servidora. De maneira similar, cada método que possa ser remotamente invocado em um objeto exportado recebe um identificador único do módulo de RMI. A JVM modificada registra esses identificadores em ambos os lados, cliente e servidor, para cada método remoto invocado. Por exemplo, a instância de um objeto `Factory_Warehouse` é identificada como objeto `25` no servidor. Esse número será único no servidor de tal forma que a ferramenta de análise de desempenho sabe que toda chamada remota para o objeto `25` é sempre direcionada a instância do objeto `Factory_Warehouse`. De maneira análoga, se um método remoto `lookup` é identificado como método `5`, a combinação do objeto `25` com o método `5` identificará no servidor `Factory_Warehouse.lookup`.

A JVM modificada também armazena o nome da máquina a fim de possibilitar a identificação dos registros correspondentes nos arquivos de traço do cliente e do servidor. No lado do cliente é registrado o nome da máquina na qual foi invocado o método remoto (servidor). No lado do servidor é registrado o nome da

máquina de onde a foi originada a invocação. O número da porta da conexão TCP/IP do cliente usada para a invocação remota é armazenado nos arquivos de traço do cliente e do servidor. Ele é necessário para distinguir entre chamadas remotas realizadas ao mesmo servidor por JVM's clientes diferentes que executam em uma mesma máquina física.

Um exemplo dos registros gerados no arquivo de traço do cliente e do servidor é mostrado na figura 3.6 para uma invocação remota a `Factory_Warehouse.lookup()` originada na JVM_1 (cliente) à JVM_2 (servidor). O registro do cliente indica que é uma chamada RMI ao método `5` do objeto `25` para o servidor em execução na máquina `cslp62b.cs.umn.edu`, na qual, neste exemplo, está o método `Factory_Warehouse.lookup()`. A informação de traço também registra o identificador da *thread*, `30144624`, o número da porta da conexão TCP/IP, `4667`, e o *time stamp*, `1022150658651`.

O registro correspondente no arquivo de traço do servidor indica uma chamada recebida da máquina `cslnt3.cs.umn.edu` através da porta `4667` para o método `5` do objeto `25`. Também são armazenados o identificador da *thread*, `29740592`, e o *time stamp*, `1033151850258`. Cabe ressaltar que o valor dos *time stamps* do cliente e do servidor não são os mesmos, uma vez que em sistemas distribuídos não há a garantia dos relógios das máquinas estarem sincronizados. A ausência de sincronização requer um processamento adicional para organizar as chamadas cliente/servidor usando a ordem cronológica dos *time stamps*.

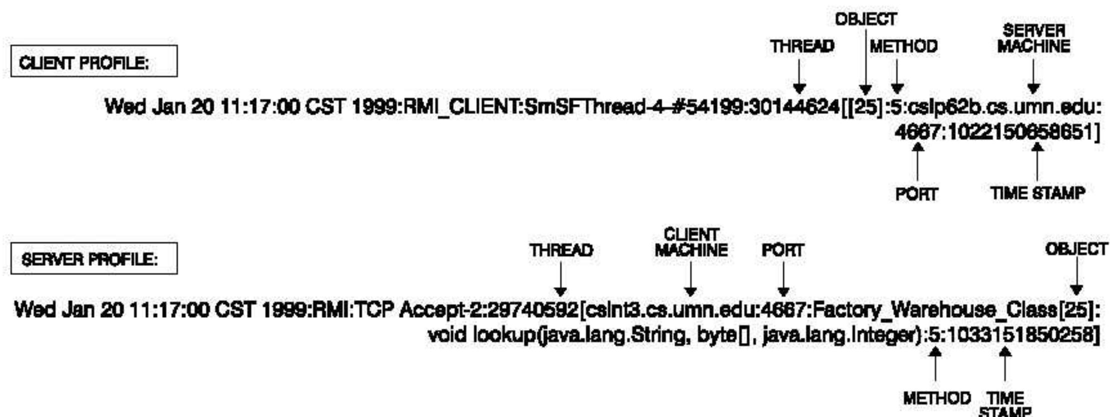


FIGURA 3.6 – Registro de Comunicação do JaViz

3.8.2 Pós-Processamento

As informações dos arquivos de traço são processadas para gerar a árvore de execução do programa Java de forma a facilitar o processo de visualização. Esse pós-processamento é feito em duas etapas: etapa de *merge* e etapa de Geração da Árvore.

Etapa de *Merge* – depois que todos os programas clientes e servidores tenham finalizado sua execução, os arquivos de traço de cada JVM envolvida na execução da aplicação distribuída estão disponíveis para serem processados. Os três arquivos de traço gerados por JVM são reunidos em um mesmo arquivo de traço contendo todas as informações de cada JVM, incluindo as atividades cliente/servidor. Esse processo cria uma ligação entre os registros das invocações de métodos remotos a partir do arquivo de traço do cliente para o arquivo de traço do servidor. No final desta

etapa, o trace resultante de cada JVM (arquivo **.mrg**) contém ponteiros para o arquivo de traço de outras JVM's, de forma que o caminho da invocação remota possa ser identificado.

Geração da Árvore – a etapa de geração da árvore analisa os arquivos de traços resultantes da etapa anterior para criar um arquivo da saída contendo a árvore de execução para um determinado cliente ou servidor. Esse arquivo é usado pelo visualizador para mostrar o grafo de chamadas. A etapa de geração utiliza os arquivos **.mrg** como entrada. Processa cada chamada de método do arquivo de traço e escreve um registro correspondente no arquivo de saída (**.jta**).

O principal objetivo da etapa de Geração da Árvore é tornar explícito o relacionamento entre método chamador – método chamado. Quando se trata de chamadas de métodos locais a mesma JVM esse relacionamento é facilmente identificado. Já para os métodos remotos o processo é um pouco mais complexo, pois o algoritmo precisa seguir a ligação entre os arquivos de traço do cliente e do servidor para incorporar ao arquivo de traço do cliente as informações correspondentes contidas no arquivo de traço do servidor. Por conseguinte, devem ser desconsiderados (excluídos) os registros de traço de métodos remotos invocados por outras JVM's.

Cada registro no arquivo **.jta** corresponde a uma chamada de método e contém informações como: (i) um ponteiro para o método antecessor (quem o chamou); (ii) um ponteiro para seu último método sucessor (o último método que chamou); (iii) número de métodos invocados pelo método atual; (iv) momento em que o método começou a executar; (v) momento em que o método finalizou a execução; (vi) o identificador da thread que executou o método; (vii) o identificador do método; (viii) o identificador da JVM na qual o método foi executado. A fim de permitir que o Visualizador mostre os nomes dos métodos, um mapeamento dos identificadores de métodos com seus nomes é realizado com o auxílio de um arquivo (**.mth**).

Geração de Estatísticas de Execução – Para facilitar a análise de desempenho do grafo de chamadas, são geradas informações estatísticas sobre cada um dos métodos do programa. Cada um dos arquivos detalhados de traço (**.prf**) são analisados para obter: o número total de vezes que um método foi chamado e informações relacionadas ao tempo de execução do método (tais como, tempo médio, mínimo e máximo e o desvio padrão). As estatísticas de todos os métodos são escritas em um arquivo (**.gtt**). As estatísticas são mostradas pelo visualizador quando um nodo é selecionado no grafo de chamadas.

3.8.3 Visualização

O visualizador recebe como entrada o arquivo resultante do pós-processamento (**.jta**) e mostra graficamente a árvore de execução do programa. Essa árvore é composta por nodos, sendo que cada nodo (nó) da árvore representa a chamada de um método. Cabe ressaltar que a árvore refere-se à execução de uma única máquina virtual. A árvore de execução mostra todas as chamadas de métodos que ocorreram na execução do cliente (ou servidor). Com isso, as chamadas RMI originadas de um programa cliente para serem executadas em uma outra JVM serão incluídas na árvore de execução desse cliente. Por outro lado, chamadas RMI recebidas e executadas em uma JVM são parte da execução de outra JVM (cliente) e serão excluídas da sua árvore de execução.

O visualizador consiste de duas janelas, uma com a árvore e outra com informações do nodo, como mostrado na figura 3.7. A janela da árvore mostra uma representação gráfica da árvore de execução. Inicialmente, a árvore mostra somente o nodo inicial e os seus sucessores imediatos. O usuário pode selecionar qualquer sucessor do inicial e visualizar seus sucessores, e assim por diante. Em um dado instante, a janela da árvore mostra todos os nodos abertos e seus arcos, sendo que o nodo selecionado é destacado por uma estrela. As folhas (nodos terminais) são mostradas com um sublinha (underscore) para indicar que não podem mais ser explorados. A janela de informações do nodo mostra dados estatísticos relacionados com o método que o nodo selecionado representa.

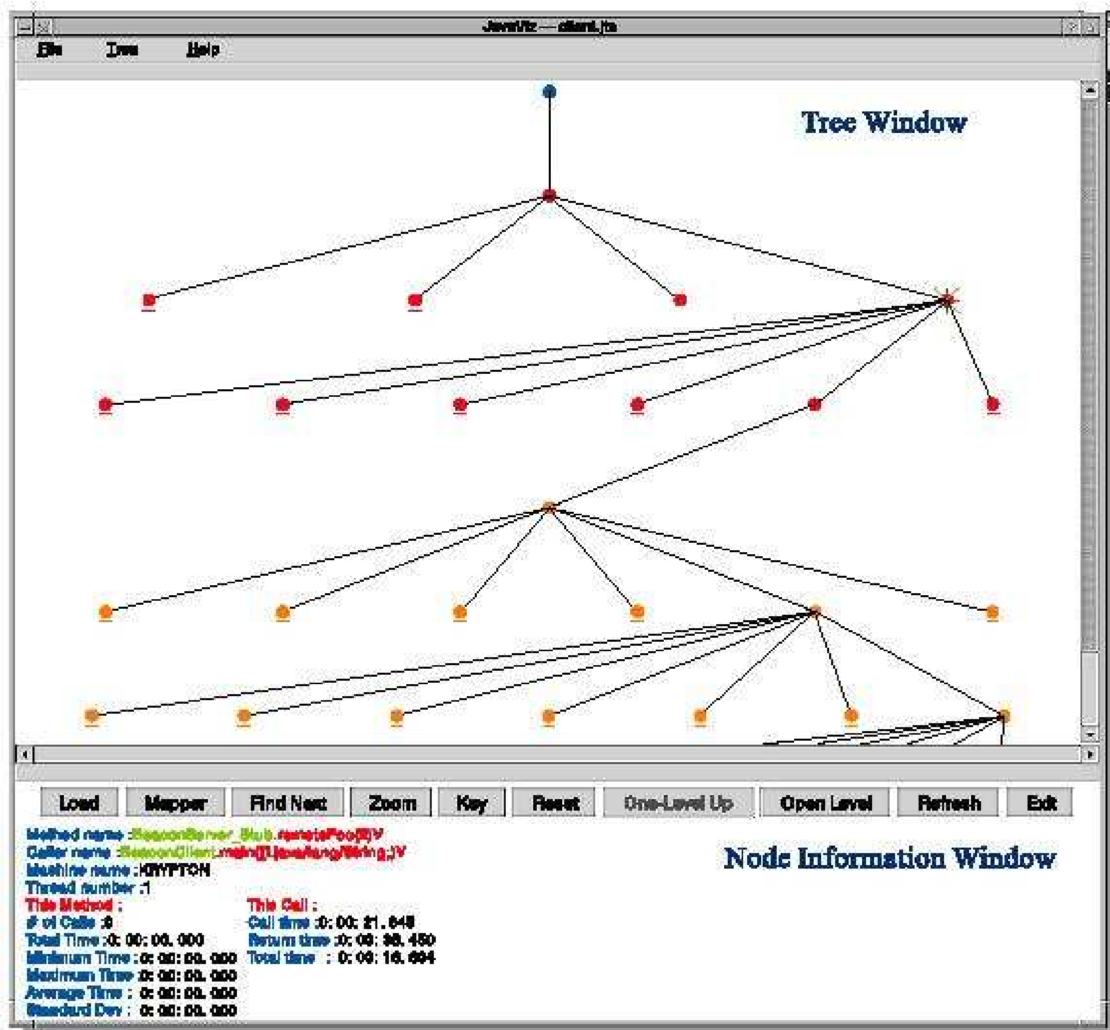


FIGURA 3.7 – Janela de Visualização do JaViz

3.9 Deterministic Replay of Distributed Java Applications

A ferramenta proposta em [CHO 2000] é uma ferramenta de ***Replay* Determinístico** (**Repetição Determinística**). Vários autores indicam que esse tipo de ferramenta auxilia no entendimento e na depuração de programas, influindo diretamente na produtividade do programador [CHO 2000], [RUS 96], [DIN 90], [LEB 87]. A ferramenta foi concebida para o modelo de Programação de Objetos Distribuídos implementados em Java.

O principal objetivo da ferramenta é transpor o problema do não-determinismo que pode ocorrer no comportamento da execução de aplicações Java devido à execução concorrente de *threads*, ao escalonamento dessas *threads* e aos retardos variáveis na rede. Esse não-determinismo torna difícil o entendimento e a depuração de aplicações *multithreaded* e distribuídas. Por exemplo, execuções repetidas de um programa são comuns durante o processo de depuração. O não-determinismo pode determinar o aparecimento de um erro em uma execução do programa e no desaparecimento do erro em outra execução do mesmo programa.

Esse trabalho é uma evolução de um sistema chamado DejaVu [CHO 98], um ambiente de Repetição Determinística de um programa java *multithreaded* em execução em uma única máquina virtual. Essa evolução, *Distributed* DejaVu, foi o resultado da integração entre novas técnicas para manipulação de eventos distribuídos com as técnicas já existentes. Como resultado obteve-se uma ferramenta de repetição determinística para aplicações *multithreaded* distribuídas desenvolvidas em Java. As técnicas de repetição foram implementadas como extensões a JVM padrão. Essa versão estendida da JVM será referenciada como DJVM.

A máquina virtual modificada, a DJVM possui dois modos de operação: (1) **Modo de Gravação**, no qual a ferramenta registra informações da execução da aplicação sobre o escalonamento lógico das threads e sobre as interações com a rede de comunicação; (2) **Modo de Repetição**, no qual a ferramenta reproduz o comportamento da execução do programa forçando o escalonamento lógico das *threads* e as interações de rede registradas.

3.9.1 Estrutura de Repetição

A reprodução da execução de um programa *multithreaded* em um hardware uniprocessado, pode ser alcançada através da obtenção das informações de escalonamento das *threads* durante a execução para forçar o mesmo escalonamento durante a repetição [RUS 96]. O escalonamento de *threads* é essencialmente uma seqüência de intervalos de tempo (*time slices*). Cada intervalo nessa seqüência contém eventos da execução de uma única *thread*. Portanto, os limites do intervalo correspondem aos pontos de troca de *threads*.

O projeto da estrutura de repetição é baseado em ***Logical Thread Schedules*** (**Escalonamento Lógico de *Threads***) e ***Logical Intervals*** (**Intervalos Lógicos**). Um dos pontos fundamentais do DejaVu distribuído é a obtenção do escalonamento lógico das *threads* [CHO 98]. Ele pode ser computado sem o auxílio do escalonador e das informações de escalonamento obtidos pelo mesmo, aqui referenciadas como **Escalonamento Físico das *Threads*** (***Physical Thread Schedule***). Com isso, o ambiente de repetição fica independente do mecanismo de escalonamento utilizado (escalonador do sistema operacional ou outro qualquer).

Para entender melhor o conceito de escalonamento lógico de *threads*, pode-se considerar um programa *multithreaded* simples, como o mostrado na figura 3.8. A *thread* `main` inicializa uma *thread* filha, T_1. Ambas podem acessar as variáveis compartilhadas `F` e `G`.

A figura 3.9 apresenta algumas instâncias de execução (escalonamento físico de *threads*) do programa exemplo em um hardware uniprocessado. A única diferença entre as instâncias de execução `(a)` e `(b)` é o momento em que a variável `J` é atualizada. Isso não afeta o comportamento da execução de um programa, pois o

acesso a uma variável local é um evento local da *thread*. O valor de `F` impresso permanece `5`. Entretanto, nas instâncias de execução `(c)` e `(d)`, o `main` imprime o valor `0` antes de T_1 atualizá-lo. Novamente, a única diferença entre `(c)` e `(d)` é a ordem de acesso as variáveis locais.

```
class Test {
  static public volatile int f = 0;
      // shared variable
  static public volatile int g = 20;
      // shared variable
  static public void main(String argv[]) {
    int j; // local variable
    MyThread t1 = new MyThread();
    t1.start();
    j = 20;
    System.out.println(``f = `` + f
      + `` j = `` + j);
  }
}

class MyThread extends Thread {
  public void run() {
    int k; // local variable
    k = 5;
    Test.f = Test.f + k;
    Test.g = Test.g - k;
  }
}
```

FIGURA 3.8 – Programa Exemplo

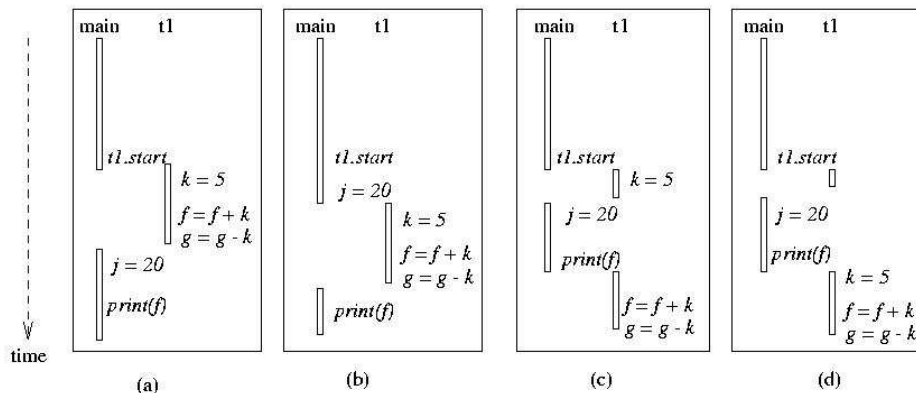


FIGURA 3.9 – Escalonamento Físico de Threads

Um comportamento de execução pode ser diferente de outro se a ordem dos acessos as variáveis compartilhadas forem diferentes. Com isso, é possível classificar escalonamentos físicos de *threads* em classes equivalentes de acordo com a ordem dos acessos às variáveis compartilhadas. No exemplo, `(a)` e `(b)` pertencem a uma classe e `(c)` e `(d)` a outra classe. O termo Escalonamento Lógico de *Threads* refere-se a todos os escalonamentos físicos que se enquadram em uma classe equivalente.

Eventos de sincronização podem afetar a ordem dos acessos a variáveis compartilhadas e, portanto, afetar o escalonamento lógico das *threads* [CHO 98]. Em [CHO 2000] o termo **Eventos Críticos** refere-se aos eventos – acessos a variáveis compartilhadas e eventos de sincronização – em que a ordem de execução pode afetar o comportamento da execução da aplicação. Um escalonamento lógico de *threads* é uma

seqüência de intervalos de eventos críticos, onde cada intervalo corresponde a eventos críticos e não-críticos executados consecutivamente em uma *thread* específica.

Um escalonamento lógico de *threads* é um conjunto ordenado de intervalos de eventos críticos, chamados **Intervalos Lógicos** ou **Intervalos Lógicos de Escalonamento** (***Logical Schedule Intervals***). Cada intervalo lógico, $\texttt{LSI}_i$, é um conjunto do máximo de eventos críticos consecutivos de uma *thread*, e pode ser representado pelo seu primeiro e último evento crítico, como $\texttt{LSI}_i$ = ($\texttt{FirstCEvent}_i$, $\texttt{LastCEvent}_i$).

A forma de capturar informações de escalonamento lógico de *threads* é baseada em um contador global (*time stamp*) compartilhado por todas as *threads* e um contador local a cada *thread*. O contador global é incrementado a cada execução de um evento crítico identificando-o unicamente. Portanto, `FirstCEvent` e `LastCEvent` podem ser representados pelos seus valores de contador global. É possível perceber que um contador é global dentro de uma DJVM. Um contador local também é incrementado a cada execução de um evento crítico. A diferença entre os dois contadores é usada para identificar o intervalo lógico de escalonamento.

O objetivo de armazenar informações sobre intervalos de escalonamento ao invés de registrar informações sobre cada evento crítico é aumentar a eficiência do mecanismo de repetição [CHO 2000]. A figura 3.10 mostra a execução de 4 *threads* e a atualização dos contadores globais e locais a cada acesso a uma das variáveis compartilhadas por cada uma das *threads*.

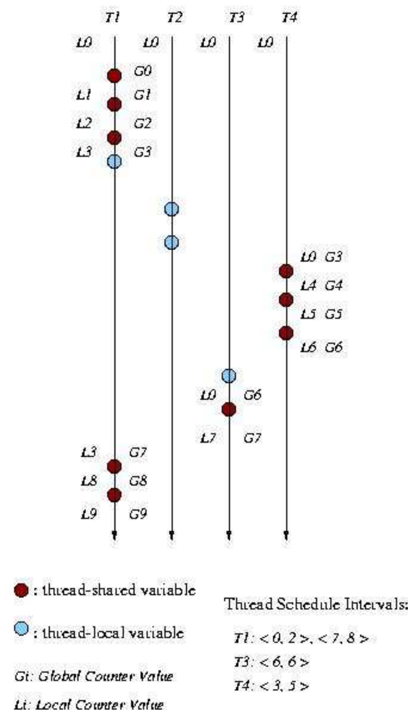


FIGURA 3.10 – Intervalo Lógico de *Threads*

No Modo de Repetição, para uma *thread* executar um intervalo de escalonamento $\texttt{LSI}_i$($\texttt{FirstCEvent}_i$, $\texttt{LastCEvent}_i$) ela deverá permanecer sem executar nenhum evento crítico até que o valor do contador global seja igual a $\texttt{FirstCEvent}_i$. Quando o valor do contador global for igual a $\texttt{FirstCEvent}_i$, a *thread* executa cada evento crítico e incrementa o valor do contador global até que o valor seja igual a $\texttt{LastCEvent}_i$. Quando o valor do contador global for igual a

$FirstCEvent_i$, a *thread* busca o próximo intervalo de escalonamento, $LSI_{i+1} = (FirstCEvent_{i+1}, LastCEvent_{i+1})$ do arquivo de traço e aguarda até que o valor do contador global seja o mesmo que $FirstCEvent_{i+1}$. Esse processo é repetido até que não existam mais intervalos de escalonamento no arquivo de traço.

3.9.2 DejaVu Distribuído

A repetição da execução de aplicações distribuídas Java é garantida através da identificação de eventos de rede como eventos críticos, pois estes podem alterar o comportamento da execução observada. O suporte na DJVM para eventos de rede assegura que os eventos de rede sejam reproduzidos na mesma ordem de execução que ocorreram originalmente (modo de gravação).

Em aplicações distribuídas, a DJVM precisa repetir o comportamento da execução como definido pela API de comunicação de rede de Java. Na sua essência, essa API é baseada em comunicação entre pontos, chamados *sockets*. A ferramenta suporta três tipos de *sockets*: (1) comunicação ponto-a-ponto ou *socket* TCP que suporta entrega confiável de pacotes; (2) datagrama ponto-a-ponto ou pacotes baseados em *sockets* UDP no qual pacotes da mensagem podem ser perdidos ou recebidos fora de ordem; (3) *socket multicast* no qual um datagrama pode ser enviado para vários *sockets* destino. No que diz respeito à repetição, *sockets multicast* são apenas um caso especial dos *sockets* UDP. O comportamento dos *sockets* UDP e TCP é diferente e, portanto, são necessárias soluções diferentes para a repetição da execução.

A) Mecanismo de Gravação e Repetição para Sockets TCP

Durante a fase de gravação é atribuído para cada DJVM um identificador único (DJVM-id). O DJVM-id permite identificar o emissor de uma mensagem ou de uma requisição de conexão. Eventos de rede são unicamente identificados por um *networkEventID*. Um *networkEvent*ID é uma tupla `<DJVMid, gCounter>`, onde DJVMid é o DJVM-id da DJVM na qual o evento de rede foi gerado e o gCounter é o valor do contador global do evento nessa DJVM.

O DejaVu distribuído possui características específicas de funcionamento de acordo com o tipo de evento de rede. A descrição feita a seguir refere-se aos eventos `connect` e `accept` utilizados para a criação de um *socket* TCP. Toda conexão *socket* pode ser unicamente identificada pelo *networkEvent*ID dos eventos `connect` e `accept` correspondentes. A chamada `connect`, executada durante uma chamada ao método `socket()` por uma *thread* cliente é um evento crítico DejaVu.

Na fase de gravação, a DJVM-cliente registra o valor do contador global no `connect` e envia a requisição de conexão de *socket* para o servidor. Quando a conexão *socket* é estabelecida, a *thread* cliente envia o *networkEvent*ID do `connect` pelo *socket* estabelecido como primeiro dado (meta-dado). Na fase de repetição, a DJVM-cliente executa o `connect` quando o valor do contador global for igual ao valor do contador global do `connect`. Além disso, é enviado o *networkEvent*ID do `connect` para o servidor como primeiro meta-dado, como na fase de gravação.

No lado do servidor, o `accept` é um evento crítico. Quando ocorre um `accept` durante a fase de gravação, a DJVM-server atualiza o contador global, aceita a conexão e recebe o *networkEvent*ID enviado pelo cliente como meta-dado do `connect` correspondente. A DJVM-server também registra a conexão *socket* estabelecida com o `accept` em um arquivo chamado *ServerSocketLog*. O *ServerSocketLog* é salvo no final

da fase de gravação e é usado para repetir deterministicamente eventos `accept` durante a fase de repetição. Cada entrada neste arquivo, chamada *ServerSocketEntry* é uma tupla `<ServerGC, ClientEventID>`, onde ServerGC é o valor do contador global na DJVM-server no evento `accept` e o ClientEventID é o *networkEvent*ID enviado pela DJVM-cliente.

Para repetir eventos `accept`, a DJVM mantém uma estrutura de dados, chamada *Connection Spool*, usada para gerenciar conexões fora de ordem. Quando um `accept` é executado por uma *thread* t_s na DJVM-server, o ambiente identifica no *ServerSocketLog* um *networkEvent*ID com valor do contador global correspondente. A DJVM-server procura no *connection pool* por um objeto *socket* já criado com o mesmo *networkEvent*ID. Se o objeto já foi criado, o `accept` é completado. Caso contrário, a DJVM-server continua armazenando no *connection pool* informações sobre as conexões fora de ordem até que receba uma requisição com o *networkEvent*ID correspondente.

B) Suporte a Sockets UDP

Um *socket* TCP é confiável, ou seja, se dados forem perdidos ou corrompidos durante a transmissão, o protocolo garante que os dados serão enviados novamente. Se os dados ou pacotes forem recebidos fora de ordem, o protocolo reorganiza-os para a ordem correta. UDP, *User Datagram Protocol*, é um protocolo alternativo que envia dados pela rede sem garantir a confiabilidade. Os pacotes, chamados datagramas, podem ser recebidos fora de ordem, duplicados ou podem ser perdidos. É responsabilidade do programador da aplicação manipular a complexidade adicional.

Para Repetição Determinística de aplicações usando UDP, a DJVM precisa garantir o mesmo comportamento de entrega de pacotes nas fases de gravação e repetição. Em outras palavras, o mecanismo de repetição deve garantir a duplicação de pacotes, perda de pacotes e entrega fora de ordem de pacotes. Detalhes desse processo em [CHO 2000].

3.9.3 Implementação e Resultados de Desempenho

A implementação da DJVM foi realizada modificando a máquina virtual java da Sun MicroSystems. Em [CHO 2000] foram apresentadas medidas de desempenho da DJVM com algumas aplicações de carga sintética (*systhetic-workload*) e com programas de *benchmark* Splash-2 (*Stanford Parallel Applications for Shared Memory*) em máquinas com arquitetura Intel e Sistema Operacional Windows NT. O *overhead* de tempo adicionado durante a fase de gravação variou entre 60% e 88% com as aplicações de carga sintética e entre 17% e 57% com os programas de *benchmark* Splash-2. O *overhead* adicionado na fase de repetição varia entre 33% e 58% com as aplicações de carga sintética e entre 32% e 66% com os programas Splash-2.

Os autores indicam que o *overhead* é menor durante a fase de repetição devido a DJVM não executar nenhuma operação de bloqueio (*lock*) como as que são necessárias para atualizar o valor do contador global durante a fase de gravação. O tamanho dos arquivos de traço variou entre 0.5kb e 36kb [CHO 98].

3.10 Uma proposta de Visualização de Aplicações Distribuídas

O objetivo do trabalho proposto por Ottogalli [OTT 2001] é auxiliar os programadores na análise da execução de suas aplicações distribuídas. A proposta inclui duas fases principais: o registro do traço de execução da aplicação e a visualização deste traço após o término da execução. O trabalho está em desenvolvimento na France Telécom no contexto do projeto Jonathan [DUM 98].

As informações de desempenho são obtidas através do registro das chamadas de métodos através do JVMPI. Eventos são registrados usando o JVMPI, acessado através da implementação de um agente de monitoração. Além de eventos relacionados ao comportamento das classes, métodos, objetos, *threads* e monitores, a ferramenta suporta o traço de eventos definidos pelo usuário (através da anotação no programa para observar eventos específicos). A utilização do JVMPI para observar eventos pode produzir um grande volume de dados. Por este motivo, a ferramenta possui uma opção de filtro que elimina o registro de classes Java padrão, o que, segundo Ottogalli reduz em 32 vezes o tamanho do traço da execução [OTT 2001].

Informações adicionais precisam ser registradas no nível do sistema operacional, para identificar a comunicação entre JVM's [OTT 2001]. Para observar a comunicação entre JVM's, é necessário identificar as ligações entre elas. A proposta usada para obter estas informações é baseada na observação das mensagens enviadas e recebidas no nível do sistema operacional. A técnica assume que serão usados *sockets* para estabelecer a conexão entre as JVM's. A identificação destes *sockets* é executada através de informações obtidas nas estruturas de dados do sistema operacional Linux.

A análise de desempenho é realizada de forma *off-line*. Os traços da execução são passados a ferramenta de visualização Pajé [STE 99]. O Pajé foi escolhido devido a sua característica de interatividade, a qual é conveniente para visualizar a execução e as comunicações entre JVM's. O Pajé precisou ser especializado para mostrar o comportamento dinâmico de aplicações distribuídas Java [OTT 2001].

4 Modelo DOMonitor

O DOMonitor surgiu da necessidade de monitorar programas distribuídos Java para obter informações mais detalhadas sobre a execução. Essas informações são posteriormente organizadas e podem ser utilizadas com diversas finalidades, ou seja, podem servir como entrada em outros projetos, tais como: Pajé, ISAM e EXEHDA. As características peculiares desses projetos exigem funcionalidade diferenciada do DOMonitor. Para o projeto Pajé, o tratamento das informações é *off-line*, enquanto que para os projetos ISAM e EXEHDA o tratamento é *on-line*.

Para construir um módulo de monitoração, várias decisões precisam ser tomadas. As próximas seções, descrevem essas decisões, com um pouco de referencial teórico sobre as técnicas existentes e comumente utilizadas mescladas com as decisões assumidas acompanhadas das respectivas razões.

4.1 A Necessidade de Monitorar Programas Distribuídos

Define-se um **programa distribuído** como sendo um conjunto de programas cooperantes para realizar uma tarefa comum. Cada programa é uma unidade determinística hábil para executar separadamente e concorrentemente com outros programas. Programas fazem duas coisas: processamento (computação) e comunicação. Processamento é a execução normal de instruções e pode ser realizado utilizando um ou mais fluxos de execução (*threads*). Comunicação é o meio pelo qual uma *thread* interage com outras. A comunicação é baseada em mensagens. Uma mensagem é a interação entre duas *threads*, a que origina os dados (emissora) e a que consume dos dados (receptora) [TSA 95].

Não há restrição sobre os programas terem de executar na mesma máquina, ou seja, nenhuma suposição é feita sobre a localização dos mesmos. Os dois extremos são: o caso onde todos executam na mesma máquina e o caso onde cada um executa em uma máquina distinta. A ferramenta proposta nesse trabalho não depende de como os programas estão fisicamente distribuídos.

Ferramentas tradicionais de monitoração não fornecem informações suficientes para tratar com os problemas de aplicações distribuídas. Essas aplicações apresentam um nível maior de complexidade e isso resulta na monitoração ser mais complexa do que a de sistemas seqüenciais. Em sistemas onde existe comunicação entre *threads*, os eventos podem ser divididos em duas classes: aqueles representando a comunicação entre as *threads* de programas diferentes e aqueles que representam atividades internas a um programa. A monitoração realizada pelo modelo proposto precisa tratar com situações inerentes aos ambientes distribuídos, tais como:

- ***Retardos Imprevisíveis na Rede de Comunicação*** – devido ao tráfego imprevisível na rede de comunicação e a distância entre os nodos, os retardos causados pela comunicação entre processadores são imprevisíveis e significativos. Com isso, a seqüência de eventos especificados em tempo de projeto pode mudar durante a execução;
- ***Não-Determinismo e Resultados que não se Repetem*** – devido aos retardos imprevisíveis na comunicação e as condições de corrida (podem ocorrer quando duas *threads* compartilham um mesmo recurso) entre processadores e *threads*, o comportamento da execução em sistemas

distribuídos é não determinístico. Com isso, pode ocorrer de uma nova execução do mesmo programa com as mesmas entradas não produzir os mesmos resultados;

- ***Referência de Relógio Global*** – os programas que fazem parte de um sistema distribuído podem estar executando concorrentemente em diferentes processadores. Cada um tem seu próprio relógio que executa independentemente do relógio dos outros. Com isso, não há uma referência global de relógio.

4.2 Objetivos da Monitoração do DOMonitor

O objetivo principal deste trabalho consiste em projetar e implementar um ambiente de monitoração para aplicações distribuídas desenvolvidas em Java. Durante a execução da aplicação no ambiente distribuído, será realizada a fase de monitoração. Nessa fase são registrados eventos relevantes como mensagens trocadas, operações de sincronização, computação realizada, e o respectivo instante em que esses eventos ocorreram. Posteriormente, na fase de processamento esses dados são organizados de forma a expressar o comportamento que a aplicação apresentou durante a sua execução. Serão ressaltadas características importantes a ambientes distribuídos como: comportamento das *threads*, compartilhamento de recursos, sincronização/bloqueio, utilização da CPU e invocações de métodos remotos (RMI).

Segundo Kazi [KAZ 2000] o desenvolvedor de aplicações distribuídas e *multithreaded* encontra 4 tipos de problemas:

- ***Distribuição da Aplicação*** – é uma característica desse tipo de aplicações que as partes que a compõem estejam dispersas em várias máquinas. Com isso, para uma avaliação efetiva de desempenho dessas aplicações, é necessário identificar os pontos onde existe um grande número de invocações remotas de métodos;

- ***Método Ineficientes*** – métodos codificados para serem flexíveis e genéricos podem causar problemas de desempenho quando utilizados com muita freqüência;

- ***Métodos de Sincronização*** – os métodos de sincronização garantem acesso mutuamente exclusivo para proteger objetos. O desempenho da aplicação pode ser degradado se a utilização desses métodos causar muita contenção;

- ***Utilização da Memória*** – a quantidade de memória utilizada na alocação de objetos pode ser determinante no desempenho de aplicações de grande porte. O problema torna-se mais crítico no momento da liberação da mesma, pois o coletor de lixo Java tende a levar mais tempo para procurar as referências a objetos em aplicações de grande porte.

Uma questão importante em projetar um ambiente de análise de desempenho é decidir para quais medidas ele será utilizado. Dentre os problemas identificados por Kazi, o DOMonitor não trata diretamente apenas do último, utilização da memória. O DOMonitor é voltado para aplicações compostas por objetos distribuídos e caracteriza-se por identificar principalmente: (i) o comportamento dinâmico das *threads*; (ii) a utilização dos métodos de sincronização; e (iii) a comunicação entre os entes distribuídos da aplicação.

(i) **O comportamento dinâmico das *threads*** que podem ser criadas ou finalizadas a qualquer momento durante a execução. Uma vez que as *threads* podem executar em momentos disjuntos, em um determinado momento poderá haver apenas uma *thread* em execução, já em um outro momento, podem haver 5 *threads* em execução nesta mesma máquina. A ferramenta foi projetada para suportar essa característica dinâmica inerente à programação distribuída, ou seja, possibilita a monitoração de mais de uma *thread* por nodo;

(ii) **A utilização dos métodos de sincronização** afeta diretamente o estado das threads. A utilização dos métodos de sincronização é imprescindível para garantir o acesso mutuamente exclusivo a algumas partes da aplicação. Se a contenção criada devido à sincronização for significativa, a mesma poderá afetar consideravelmente o desempenho da aplicação;

(iii) **A comunicação entre os entes distribuídos da aplicação.** Os objetos alocados em máquinas diferentes comunicam-se utilizando invocações remotas de métodos. Ou seja, um objeto pode chamar um método de um outro objeto mesmo que eles não estejam em execução na mesma máquina. Um aspecto crítico na análise de desempenho para esse tipo de aplicação distribuída é a identificação das partes do programa onde existe um número expressivo de invocação de métodos remotos (RMI).

Os dados de monitoração obtidos podem ser utilizados com várias finalidades. Isso porque qualquer ferramenta que necessite de informações sobre a execução de uma aplicação passa por um processo de obtenção dessas informações (monitoração). Entre as diversas utilizações que as informações de monitoração podem ter:

- **Depuração / Visualização** - permite que sejam examinados aspectos dinâmicos sobre a execução. Existem diversas razões pelas quais um programador deseja examinar aspectos dinâmicos do programa: para avaliar o desempenho, para demonstrar a presença de erros, para possibilitar a caracterização de gargalos, e assim por diante. A utilização de ferramentas que auxiliem a entender o comportamento da aplicação permite a otimização dos seus componentes, de forma a obter ganho de desempenho, dentre outros aspectos;

- **Suporte a Execução de Aplicações Móveis -** a computação móvel é uma nova proposta de paradigma computacional oriunda das tecnologias de rede sem fio e sistemas distribuídos. Uma arquitetura para sistemas móveis deve ser projetada com mobilidade, flexibilidade e adaptabilidade intrínsecas [AUG 2001a]. A monitoração serve como suporte neste contexto, fornecendo informações necessárias ao processo de tomada de decisão do ambiente de execução.

As finalidades aqui citadas não restringem a utilização das informações de monitoração em outras classes de ferramentas que igualmente necessitem de informações sobre a execução. As exigências de informação podem ser diferentes conforme a finalidade e, portanto, a intrusão causada na aplicação também poderá ser diferente. De acordo com a finalidade, a intrusão poderá inclusive ser aceita. Por exemplo, para visualização, desde que não haja troca na ordem dos eventos, o *overhead* é aceitável. No entanto, quando os dados são usados *on-line*, como para escalonamento, o *overhead* deve ser o menor possível.

Dentro do contexto do grupo de pesquisa onde o trabalho foi desenvolvido, cabe ressaltar que aplicações desenvolvidas utilizando o DOBuilder poderão ser monitoradas e posteriormente visualizadas utilizando as informações obtidas pela ferramenta. Cabe ressaltar ainda, a utilização dos dados de monitoração pelo ISAM, igualmente desenvolvido no grupo.

4.3 Atividades de Monitoração realizadas por Software

A monitoração de programas paralelos e distribuídos pode ser realizada com diferentes graus de interferência. Em uma extremidade estão as propostas por software, também chamadas de *Monitores Intrusivos*, porque eles apresentam efeitos na execução do programa que está sendo observado. Na outra extremidade estão os sistemas com grande suporte de hardware para as funções de monitoração. Tais sistemas que não afetam o comportamento dos programas sendo monitorados são chamados de *Monitores Não-Intrusivos*.

A escolha no DOMonitor foi por utilizar monitoração por software. Isso se deve principalmente pela flexibilidade inerente as propostas por software. Além disso, monitoração por hardware exige suporte adicional de hardware, ou seja, todas as máquinas do ambiente de execução teriam que possuir tal suporte, o que iria contra as características dinâmicas de um ambiente heterogêneo distribuído comumente utilizado na execução de aplicações Java.

A monitoração por software pode ocorrer em vários níveis: (a) **aplicação que é instrumentada** para gerar informações de acordo com o código de instrumentação; (b) **sistema de suporte**, que implementa todo o gerenciamento da monitoração e notificação da aplicação; (c) **sistema básico nativo**, que oferece mecanismos de monitoração no ambiente de execução com rotinas que detectam e tratam dos eventos de interesse.

O processo de **Instrumentação** consiste em alterar o código fonte da aplicação a ser monitorada, adicionando código de instrumentação em pontos específicos para gerar informações pertinentes aos eventos que interessam. Esse processo pode ser realizado de forma automática ou manual. A instrumentação automática é feita com o auxílio de um Software de Instrumentação que se encarrega de especificar no código da aplicação os códigos de instrumentação necessários. O procedimento manual é realizado com a intervenção do programador que passa a ter a responsabilidade de inserir o código de instrumentação.

A monitoração através de um **Sistema de Suporte** ou através do **Ambiente de Execução** são técnicas específicas a um determinado modelo de programação e que dispensam a necessidade de alterar o código fonte da aplicação. Nestes casos, um suporte de monitoração fornece rotinas que detectam a ocorrência de eventos sem ser necessário indicar explicitamente através de um código de instrumentação o momento em que a aplicação interrompe sua execução e passa o controle ao módulo de monitoração para que este armazene as informações necessárias.

No DOMonitor, a escolha recaiu em desenvolver um ambiente de monitoração que não exigisse do programador ter conhecimento sobre os detalhes de funcionamento do processo de monitoração e ter de alterar o código da aplicação. Com isso, a escolha foi por implementar a ferramenta em conjunto com um sistema de suporte disponível no ambiente de execução Java. Como resultado final, tem-se a monitoração sendo realizada em conjunto, pela máquina virtual e por um agente de

monitoração, como será mais bem explicado na seção 5.2. Essa decisão torna a proposta flexível e de fácil utilização, uma vez que não é necessário alterar o código fonte da aplicação ou utilizar uma versão especial da máquina virtual.

Devido ao Software de Monitoração executar em conjunto com a aplicação monitorada, ele introduz tempo de processamento e utiliza espaço de memória que poderia ser utilizado pela aplicação monitorada. Esse *overhead* ocorre principalmente pela execução de código adicional responsável pela detecção e pelo processamento de eventos. Esta intrusão pode alterar a duração da aplicação, mas não deve alterar a ordem dos eventos. Monitoração excessiva perturba a aplicação monitorada, mas monitoração reduzida pode não fornecer detalhes suficientes. Portanto, permanece o dilema entre a perturbação causada e o comportamento correto sobre a execução. Com isso, o problema encontrado na monitoração por software é determinar como controlar e prever o nível de intrusão (perturbação) mantendo o registro das informações suficientes. Em sistemas distribuídos, a intrusão é aceitável enquanto a atividade de monitoração não alterar a ordem dos eventos [TSA 95].

Na concepção do DOMonitor, considera-se que o sistema de monitoração deve aproveitar as particularidades das arquiteturas de hardware e de software tanto na coleta quanto no registro das informações disponibilizadas. Por essa razão, os processos são otimizados quanto à manipulação dos tipos de variáveis, estrutura de dados, dimensões das informações coletadas, operações de Entrada/Saída, etc. O capítulo 5 detalha melhor essas questões.

4.4 Monitoração Baseada em Eventos

Existem várias técnicas para avaliar um sistema de computador. A escolha depende do tipo de análise desejada e do nível na qual ela é executada. A técnica usada para a coleta dos dados define o modo de monitoração: ***Event Trace*** **(Traço de Eventos)** ou ***Sampling*** **(Amostragem)** [MEN 94]. **Traço de Eventos** é uma técnica baseada na ocorrência de eventos. **Amostragem** é uma técnica baseada em amostras, sendo que a execução da aplicação é interrompida periodicamente para obter informações sobre a situação atual da execução. A técnica baseada no Traço de Eventos fornece informações mais precisas que Amostragem, uma vez que não são perdidas etapas da execução.

Amostragem é uma técnica estatística que, ao invés de examinar todos os dados, analisa somente parte deles, utilizando-se de amostras. A partir dessas amostras é possível estimar alguns parâmetros que caracterizam o todo. Na avaliação de sistemas de computador, essa técnica apresenta a vantagem de produzir menos dados, portanto simplificando a análise. Amostragem coleta informações sobre o sistema em instantes de tempo específicos, ao invés de ser disparada pela ocorrência de eventos. A amostra é acionada por interrupções de tempo, baseadas no *clock* do hardware. O *overhead* introduzido por um monitor de amostragem depende de dois fatores: o número de variáveis medidas em cada amostra e o intervalo entre as amostras [TSA 95].

Verifica-se uma clara relação entre o *overhead* e a precisão dos resultados. Com a habilidade de especificar ambos os fatores, um monitor por amostragem tem condições de controlar seu *overhead*. A precisão das informações obtidas através de amostragem é determinada pela freqüência de amostras. Uma freqüência de amostras muito alta fornece informações mais detalhadas, mas apresenta uma maior perturbação na execução do programa. Erros também podem ser introduzidos devido a interrupções mascaradas. Por exemplo, se algumas rotinas do sistema operacional não puderem ser

interrompidas, a sua contribuição para a utilização da CPU não será considerada por um Monitor por Amostragem.

Uma vez que um **Evento** é uma mudança no estado do sistema, uma forma de coletar dados sobre certas atividades do sistema é capturar todos os eventos associados e registrá-los na mesma ordem na qual eles ocorreram. Em um monitor de software com a técnica de Traço de Eventos, após a detecção de um evento, é chamada uma rotina apropriada que gera um registro contendo uma referência de tempo e dados relevantes ao evento. Nesse tipo de monitoração, o comportamento da aplicação é descrito como uma série de eventos que ocorreram. Deve-se cuidar a forma como são registrados os eventos, o impacto no desempenho e o efeito colateral. A monitoração no DOMonitor é baseada em eventos. Os eventos podem ser divididos em locais (as *threads* origem e destino estão executando na mesma JVM) e remotos (as *threads* origem e destino estão executando em JVM's diferentes).

Uma vez que a execução é baseada em traço de eventos, é necessário determinar: **o que são os eventos? quais são os eventos?** De alguma forma, o monitor precisa ter conhecimento sobre o que são os eventos e quais eventos deseja-se monitorar. Uma forma de resolver esse problema é exigir que o programador identifique esses pontos através de construções específicas. Outra possibilidade é ter um software que realize essa inserção de forma automática, mas ainda assim, modificando o código fonte da aplicação. Portanto, uma das formas é ter a **Aplicação Instrumentada**, como é o caso do Pablo [PAB 2000].

Outra forma é através de **Instrumentação direta**, como é o caso do JaViz [KAZ 2000]. No JaViz, a JVM foi modificada para capturar diretamente a ocorrência dos eventos. Com isso, o ambiente fornece rotinas de sistema modificadas, que além de executar suas funções originais, registram a ocorrência dos eventos. Essa técnica é similar a usada pelo Athapascan, que utiliza uma versão instrumentada da biblioteca de execução para obter o traço de eventos [BRI 97].

Outra possibilidade, como é o caso do DOMonitor, é o ambiente de suporte possuir uma interface onde é possível definir quais são os eventos de interesse, sem alterar a aplicação origem em uma única linha. O **sistema básico** (ambiente de execução) fornece um mecanismo que permite o desenvolvimento de um monitor por software baseado em eventos sem a necessidade de alterar o ambiente de execução ou a aplicação.

Quando a taxa de eventos torna-se muito alta, a rotina de monitoração é executada com muita freqüência, o que introduz um grande *overhead* no processo de medição. Dependendo dos eventos selecionados e a taxa de ocorrência dos eventos, o *overhead* pode alcançar níveis inaceitáveis. Portanto, os eventos monitorados devem ser significativos e de interesse dos programadores, isto é, eventos que sejam indicativos do comportamento da aplicação [BAT 95].

4.5 Demais Decisões do Modelo

Posterior as decisões abordadas anteriormente, é preciso detalhar outras características. O ponto de maior importância da monitoração é manter o *overhead* tão baixo quanto possível armazenando informações suficientes para a finalidade desejada. A monitoração no DOMonitor procura diminuir o *overhead* e facilitar a tarefa de monitoração da seguinte forma:

- ***Organização Modular*** – os módulos oferecem as propriedades flexibilidade e expansibilidade, as quais permitem a adição e/ou modificação de funcionalidades. Essas características no DOMonitor são alcançadas através da separação da tarefa de detectar e coletar os eventos da tarefa de analisar e mostrar as informações. Portanto, ferramentas que utilizem as informações sobre a execução não precisam preocupar-se em como elas são coletadas, mas somente com a interpretação e apresentação delas aos usuários. Desta forma, o sistema de monitoramento mantém independência da aplicação que monitora, e as informações de saída podem ser utilizadas por outras ferramentas;

- ***Monitoração Transparente*** – simplifica a tarefa de programação, pois a funcionalidade é implementada na camada de suporte do ambiente de execução, sem exigir o envolvimento do programador e sem a necessidade de modificações no código fonte da aplicação. Por questões de flexibilidade, o ambiente possui uma interface que permite ao programador especificar *"o que monitorar"* (ao invés de *"como monitorar")*. Desta forma, o programador especifica somente os eventos de interesse, deixando que o DOMonitor gerencie a monitoração destes eventos;

- ***Portabilidade*** – o DOMonitor foi projetado para ser completamente independente da implementação da máquina virtual Java, ou seja, funcionará em conjunto com qualquer máquina virtual que implemente as mesmas funcionalidades que a máquina virtual padrão. Os módulos do DOMonitor foram desenvolvidos em Java e C, a fim de alcançar portabilidade. Além disso, o processo de monitoração não altera o código fonte da aplicação monitorada e não exige nenhum processo de recompilação;

- ***Minimizar o Overhead de Monitoração*** – o processo de monitoração sempre interfere na execução da aplicação, porém esta interferência deve ser mínima. O *overhead* causado pela monitoração é previsível e não altera a ordem dos eventos. O DOMonitor, tenta minimizar a intrusão através de um projeto cuidadoso das estruturas de dados e algoritmos internos, os quais tem como situação foco o caso onde não são considerados os eventos relacionados a API Java. O grau de intrusão que pode ser tolerado depende da natureza da aplicação e do resultado desejado da monitoração. Quando desabilitado, o agente de monitoração não interfere na execução da aplicação;

- ***Controlar a Velocidade e a Ocupação de Memória*** – para minimizar o *overhead* os eventos devem ser detectados e as informações armazenadas na velocidade do processador. Isso requer uma memória de alta velocidade. De acordo com o número de eventos de interesse e com a freqüência de ocorrência destes, a quantidade de dados coletados normalmente será grande, ocupando todo o espaço de memória rapidamente. As soluções adotadas para tratar desse problema são: restringir os dados coletados durante a monitoração de forma que somente os eventos de interesse sejam registrados; dos eventos de interesse registrados, filtrar os dados obtidos, extraindo somente as informações que interessam; gravar o conteúdo da memória em

dispositivos secundários de armazenamento somente quando atingir a capacidade da memória;

- ***Monitoração do Código da Aplicação*** – é possível evitar (funcionamento padrão) a coleta dos métodos da biblioteca Java (API Java). Com isso, não são registrados no traço os eventos relacionados a API Java. Sendo assim, o enfoque são as invocações de métodos de objetos das classes e subclasses da aplicação do usuário.

4.6 Arquitetura do DOMonitor

A arquitetura do DOMonitor é estruturada em módulos. A figura 4.1 apresenta a estrutura modular, e o contexto de monitoração e uso do DOMonitor. A definição dos eventos de interesse e a escolha pela ferramenta que utilizará os dados são realizadas na interface gráfica da ferramenta (representada na figura pela inscrição GUI). A detecção e o registro dos eventos são de responsabilidade da máquina virtual em conjunto com uma biblioteca de ligação dinâmica do DOMonitor.

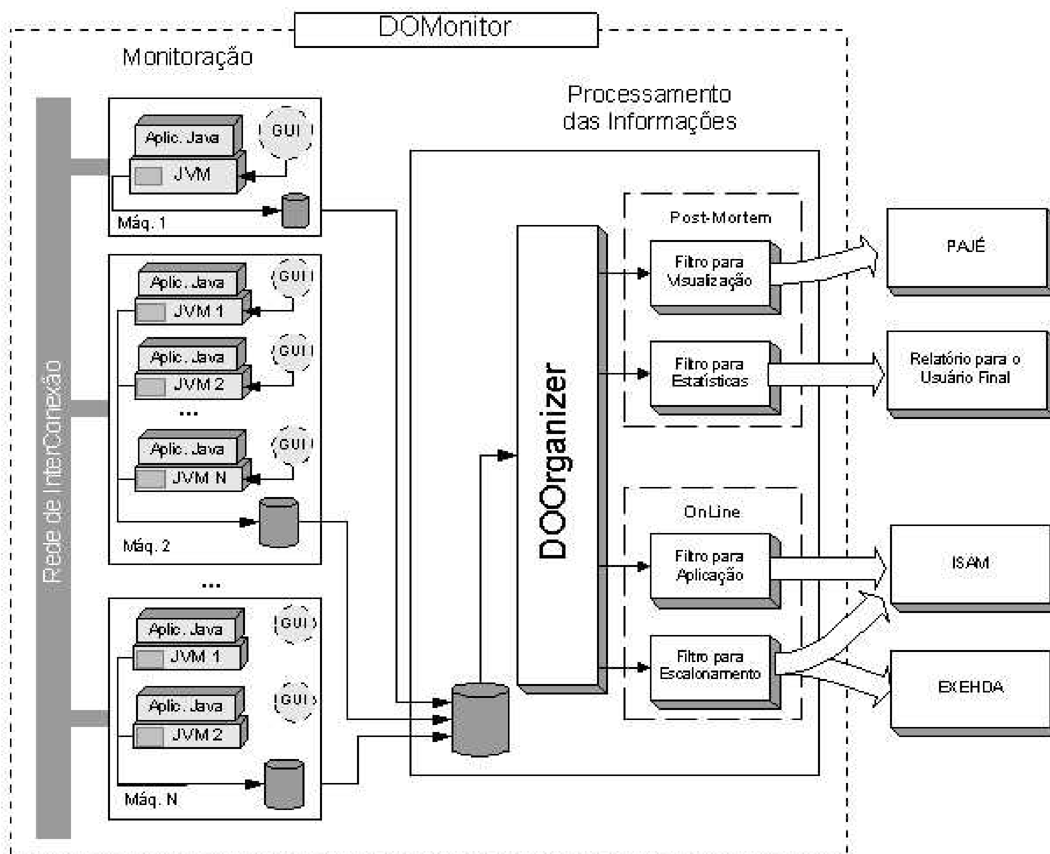


FIGURA 4.1 – Arquitetura de Módulos do DOMonitor

Uma característica que deve ser levada em consideração é a interferência que a ferramenta de monitoração pode ocasionar. Assim, outra decisão de projeto é em relação ao momento em que ocorre o uso das informações monitoradas. Devido às características distintas dos projetos Pajé, ISAM e EXEHDA, o DOMonitor permite o uso de informações em dois momentos: *off-line* – após a execução, e *on-line* – durante a execução. No caso do projeto Pajé, para uma visualização integrada do que ocorreu nos diferentes processadores, se faz necessário um processo de composição (*merge*) das

informações coletadas. No caso do projeto ISAM, grande parte das informações requeridas são individualizadas por processador.

Descrições sucintas sobre os módulos são realizadas nas próximas seções. Informações detalhadas sobre o funcionamento são apresentadas no capítulo 5 (Coleta das Informações), capítulo 6 (Processamento das Informações) e capítulo 7 (Integração com outros ambientes). Detalhes adicionais ainda poderão ser esclarecidos no capítulo 8 (Implementação).

4.6.1 Definição dos Parâmetros para Monitoração

A definição das funcionalidades e do comportamento do mecanismo de monitoração é realizada através da interface gráfica da ferramenta (figura 4.2). Através da interface, o usuário pode definir quais são os **eventos de interesse** que deseja monitorar. Em mais alto nível, o usuário pode determinar qual o enfoque de monitoração que deseja de acordo com a categoria de aplicação que será monitorada, ou mesmo, permite preocupar-se somente com um determinado comportamento. Para cada uma das categorias, são selecionados automaticamente os eventos essenciais para determinar o comportamento. Além disso, o usuário pode selecionar eventos adicionais conforme critério próprio. Isso torna a escolha dos eventos de interesse mais genérica e aplicável.

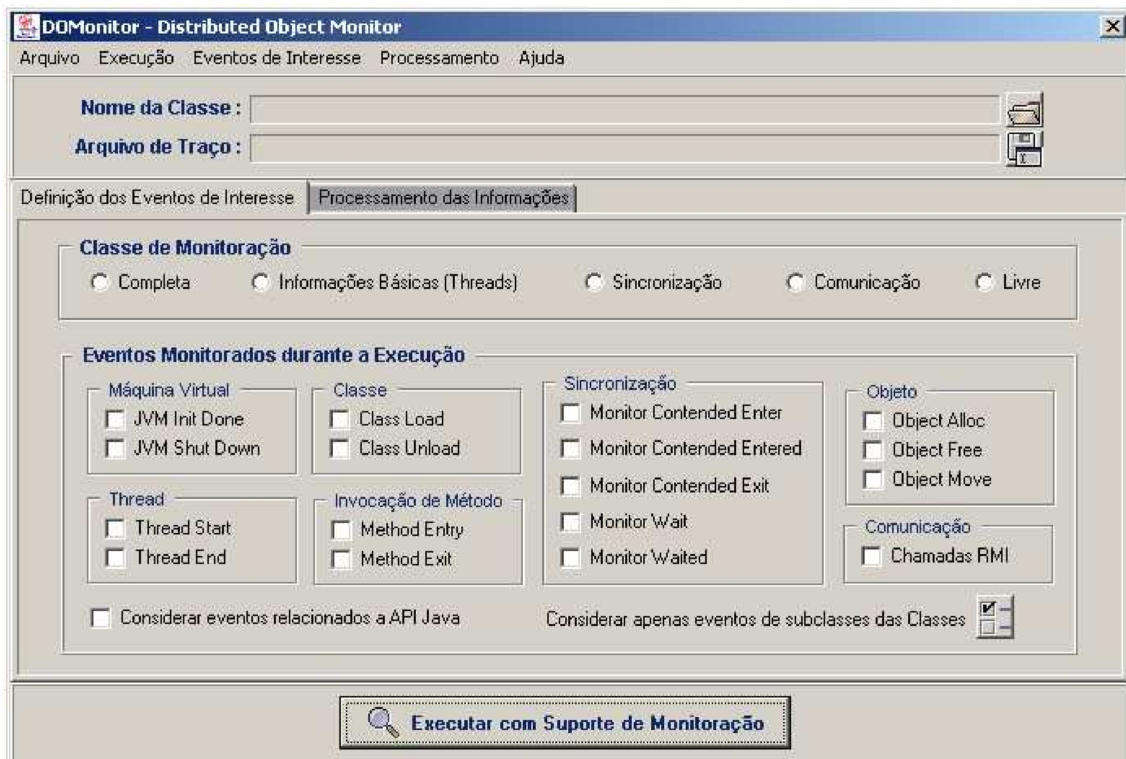


FIGURA 4.2 – Interface Gráfica do Usuário

As **opções de filtro** também são definidas a partir da interface gráfica. A opção padrão é por eliminar todos os eventos gerados por classes que pertencem a API Java. Com isso, o enfoque de monitoração fica restrito as classes da aplicação do usuário. Outra opção de filtro possibilita que o usuário limite a monitoração a apenas algumas classes de seu interesse.

Outra preocupação do usuário é com a definição da **finalidade dos dados de monitoração**. A opção padrão é por utilizar os dados de monitoração após a aplicação ter sido totalmente executada. No entanto, se o processamento precisar ser realizado de forma *On-Line*, deverá obrigatoriamente definir qual a ferramenta destino, para que seja possível carregar as funcionalidades específicas à ferramenta escolhida. Dependendo destas funcionalidades, informações adicionais são requisitas, tais como: se a publicação de resultados será realizada em intervalos pré-definidos de tempo, ou se o agente de monitoração ficará aguardando por uma requisição da ferramenta.

A ferramenta prepara a linha de comando que será executada no sistema operacional, sendo permitido ao usuário adicionar e/ou remover características a essa linha. Antes da aplicação começar a executar, duas tarefas importantes são realizadas:

- A JVM é inicializada com suporte de monitoração. Isso é realizado através da carga do agente de monitoração juntamente com seus respectivos parâmetros;
- É inicializado o suporte de monitoração para aplicações que utilizam RMI. Esse processo consiste em inicializar uma classe que irá se sobrepor a classe original. Essa nova classe será responsável por coletar os eventos relacionados à comunicação remota entre objetos, além de executar sua tarefa normal.

4.6.2 Coleta de Informações

O módulo de **Coleta das Informações** do DOMonitor, o DOProf, é responsável por registrar informações relacionadas aos eventos gerados durante a execução da aplicação. Este processo tem início no instante em que a máquina virtual Java é carregada. Para que a aplicação seja monitorada, o DOProf precisa ser carregado em conjunto com a JVM. Após ativo, o DOProf informa a máquina virtual sobre os eventos de interesse para os quais deseja receber notificação.

Cada vez que a JVM detecta a ocorrência de um evento de interesse, o controle é transferido para o DOProf, o qual registra a ocorrência do evento armazenando as informações relevantes. O controle retorna então para o ponto de onde a execução da aplicação do usuário parou.

Em um ambiente cliente/servidor com *n* JVM's diferentes (clientes ou servidores), serão gerados *n* diferentes arquivos com informações representando o perfil de execução de cada cliente e servidor. Cada um dos arquivos corresponde ao traço de eventos que ocorreram durante a execução da aplicação em um determinada JVM. Este traço é composto por diversos tipos de eventos, cada um com informações específicas conforme o formato definido pelo DOMonitor.

4.6.3 Publicação de Resultados

No módulo de **Processamento das Informações**, as informações geradas pelas JVM's são processadas para definirem o comportamento da aplicação. Uma característica que deve ser considerada é a interferência que esse processo pode ocasionar na execução da aplicação. Inicialmente, duas características precisam ser consideradas: (i) se o processamento das informações será realizado *On-Line* (durante a execução) ou *Post-Motern* também conhecido como Pós-Processamento (após o

término da execução); (ii) se é necessário reproduzir o comportamento global da aplicação, ou se as informações locais a cada máquina são suficientes.

A necessidade de informações relacionadas ao comportamento da aplicação é determinada de acordo com o tipo de ferramenta. Dependendo da ferramenta que utilizará os resultados obtidos, será suficiente obter o comportamento local da aplicação ou será necessária uma composição dos eventos para obter o comportamento global. Assim sendo, a necessidade de informações é determinada de acordo com a classe de ferramenta que utilizará os resultados, sendo que algumas classes exigem comportamento global, como as ferramentas de visualização e para outras classes é suficiente informações locais a cada máquina, como é o caso dos escalonadores.

O processamento do comportamento global da aplicação é normalmente realizado após o término da execução (*post-mortem*). O objetivo principal de aguardar o término da execução é diminuir a interferência que o processo certamente causaria no comportamento da aplicação. Esse processamento é realizado de forma centralizada, sendo que cada máquina envia suas informações a uma máquina central que realiza todo o processamento. Em sistemas distribuídos, a falta de um relógio global impede que haja ordenação entre os eventos. A ordenação de eventos é de importância essencial na monitoração e principalmente na depuração dos programas. Sem um conhecimento mínimo das relações de causalidade entre os eventos, a análise dos resultados fica extremamente prejudicada. No DOMonitor, a ordenação dos eventos de JVM's diferentes é realizada utilizando um método estático para correção de relógios [MAI 95].

Existem ferramentas que necessitam de informações durante a execução, normalmente para permitir um acompanhamento imediato ou uma possível tomada de decisão. Assim sendo, esse processo deve ser feito *on-line*, o que aumenta a interferência na aplicação. No entanto, para algumas classes de ferramentas que necessitam de informações imediatas, como é o caso dos escalonadores, são suficientes as informações locais. Com isso, tem-se uma abordagem distribuída no que diz respeito ao processamento das informações, ou seja, cada uma das máquinas realiza seu próprio processamento independente das demais.

Os dados armazenados são processados no final de um período de coleta por um módulo separado, chamado **filtro**. Esse filtro é definido de acordo com o formato de entrada da ferramenta que utilizará as informações da execução. O DOMonitor é extensível neste ponto uma vez que permite a geração de dados para diversas utilizações, sendo necessário apenas o desenvolvimento de um novo filtro, que execute uma organização das informações conforme a necessidade da ferramenta. Portanto, esses dados poderão abastecer diversas classes de ferramentas que necessitem de informações mais detalhadas sobre a execução da aplicação. Propostas de integração com outros ambientes são apresentadas no capítulo 7.

5 A Coleta de Informações no DOMonitor

Monitoração é o processo de registrar a ocorrência de eventos específicos durante a execução do programa com o objetivo de obter informações de tempo de execução que não poderiam ser obtidas a partir da análise do código do programa [TSA 95]. As principais aplicações destas informações têm sido: teste e depuração de programas, escalonamento dinâmico de tarefas, análise de desempenho e otimização de programas. Mais recentemente, o contexto de execução das aplicações móveis também é monitorado para permitir a adaptação das mesmas ao seu complexo ambiente de execução [AUG 2001c]. Este capítulo descreve em detalhes o processo de monitoração do DOMonitor.

5.1 Definição dos Parâmetros de Monitoração

A definição dos parâmetros de monitoração é realizada por intermédio de uma interface gráfica com o objetivo de tornar esta tarefa o mais simples possível e com menores chances de erro. As informações selecionadas na interface são armazenadas em um arquivo de configuração do DOMonitor. A interface gráfica é necessária apenas até que seja disparado o processo de execução da aplicação, momento em que ela pode ser encerrada para liberar espaço de memória e utilização do processador.

O processo de parametrização da monitoração envolve obrigatoriamente a definição da **classe "executável"** que se quer monitorar. Uma vez que tenha sido escolhida a classe que será executada, automaticamente a ferramenta sugere um nome para o **arquivo de traço** que será gerado durante a execução.

5.1.1 Definição dos Eventos de Interesse

A monitoração no DOMonitor é baseada na ocorrência de eventos. A monitoração de todos os eventos pode ocasionar em uma grande quantidade de dados sendo coletados, o que exigiria um considerável tempo de CPU. Segundo Reed [REE 98], deve existir a preocupação com a quantidade das informações armazenadas, principalmente em sistemas paralelos e distribuídos com vários processadores onde a freqüência de ocorrência de eventos tende a ser maior. Por estas razões, é importante que os eventos monitorados (**eventos de interesse**) sejam cuidadosamente selecionados. De fato, o *overhead* causado pelo monitor é diretamente proporcional à variedade de eventos interceptados e a freqüência com que eles ocorrem.

A JVM disponibiliza uma série de eventos para que o agente de monitoração os escolha de acordo com suas necessidades. Dentre os eventos disponibilizados, foram selecionados os eventos necessários para os propósitos do DOMonitor. Estes eventos foram divididos em categorias para auxiliar o usuário no entendimento da relação entre eles, conforme pode ser observado na interface gráfica do DOMonitor (figura 5.1).

A categoria **Máquina Virtual** contém dois eventos de interesse: um que indica o momento em que a máquina virtual foi inicializada e outro que indica o momento em que a máquina virtual vai ser descarregada. Esta categoria será sempre considerada uma **categoria de interesse**, pois é partir da geração destes eventos que o agente de monitoração realiza algumas tarefas de suporte como será detalhado na seção 5.2.

A categoria ***Thread***, pertencem os eventos relacionados ao ciclo de vida da *thread*. Um dos eventos diz respeito ao início da execução da *thread* e outro é gerado no momento em que a *thread* finaliza sua execução. Todos os eventos das demais categorias são gerados no contexto da execução de uma *thread* específica. Portanto, todas as demais categorias dependem da categoria *Thread*. Além disso, os eventos desta categoria são de suma importância para ser possível associar os custos de um determinado evento a uma *thread* específica.

A categoria **Classe** engloba dois eventos que indicam o momento em que uma classe foi carregada ou descarregada pela máquina virtual. Esta categoria é de importância fundamental para o funcionamento das opções de filtro do DOMonitor, pois é a partir dos eventos desta categoria que são modificadas as estruturas de dados do DOMonitor para eliminar chamadas a API Java ou para considerar apenas as classes selecionadas a partir do filtro de classes (seção 5.3.2).

A categoria **Objeto** engloba os eventos relacionados à criação e destruição de objetos durante a execução da aplicação. Uma vez que quando se define um objeto, ele pertence a uma determinada classe, existe uma dependência direta da categoria objeto em relação à categoria Classe. Com isto, para que a JVM possa criar um objeto de uma determinada classe, primeiro esta classe precisa ser carregada na JVM (seção 5.3.3).

A categoria **Método** contém dois eventos relacionados à invocação de métodos: um evento que indica o início da execução de um método e outro que indica a conclusão. Existe uma dependência direta desta categoria com as categorias Objeto e Classe. As informações relacionadas aos métodos são disponibilizadas ao agente de monitoração no momento em que a classe a qual o método pertence foi carregada pela máquina virtual (seção 5.3.2). Além disso, quando um método é invocado, o agente de monitoração recebe o identificador do objeto no qual foi chamado o método.

A categoria **Sincronização** engloba diversos eventos relacionados às atividades sincronização. Os três primeiros eventos são gerados a partir da utilização de métodos ***synchronized***. Os demais identificam bloqueios ocasionados por chamadas a wait e sleep. Esta categoria depende da categoria Objeto, pois a contenção por recursos é normalmente associada a objetos compartilhados.

A categoria **Comunicação** é a responsável por obter às invocações de métodos remotos. Esta categoria depende das categorias Classes, Objetos e Métodos. A partir das informações disponibilizadas por estas categorias é que será possível identificar todas as informações relacionadas à chamada remota.

Retomando uma das decisões assumidas para o modelo, a de **Monitoração Transparente**, ficou definido que é permitido ao usuário a abstração de ***O que Monitorar***, sem que o usuário envolva-se em ***Como Monitorar***. Para alcançar esta propriedade, a ferramenta disponibiliza através de sua interface gráfica a opção por escolher **Classes de Monitoração**.

As classes de monitoração são definidas de acordo com algumas características de execução das aplicações desenvolvidas para o modelo de objetos distribuídos. Para cada uma destas classes, existe um conjunto pré-definido de **eventos de interesse** que serão monitorados. Esse conjunto é determinado de acordo com a dependência entre as informações geradas por eventos diferentes e não pode ser alterado para evitar que sejam perdidas informações essenciais. Além dos eventos pré-definidos, o usuário pode selecionar eventos adicionais de acordo com critério próprio.

Uma vez que o usuário vai alternando entre as classes de monitoração, é possível verificar na interface os eventos de interesse que serão monitorados durante a execução para obter todas as informações necessárias à classe correspondente. As Classes de Monitoração disponíveis são:

A) Completa;

B) Informações Básicas (*threads*);

C) Sincronização;

D) Comunicação;

E) Livre;

A) Completa

A classe de monitoração **completa** considera eventos de todas as categorias, sendo escolhida quando se deseja obter informações detalhadas sobre as diversas características da aplicação. Com ela são obtidas informações relacionadas ao comportamento dinâmico das *threads* (no que diz respeito ao seu ciclo de vida e a contenção por recursos), as classes carregadas e aos objetos criados durante a execução, as chamadas de métodos locais e as invocações remotas de métodos. A partir da figura 5.1 é possível verificar os eventos que serão monitorados.

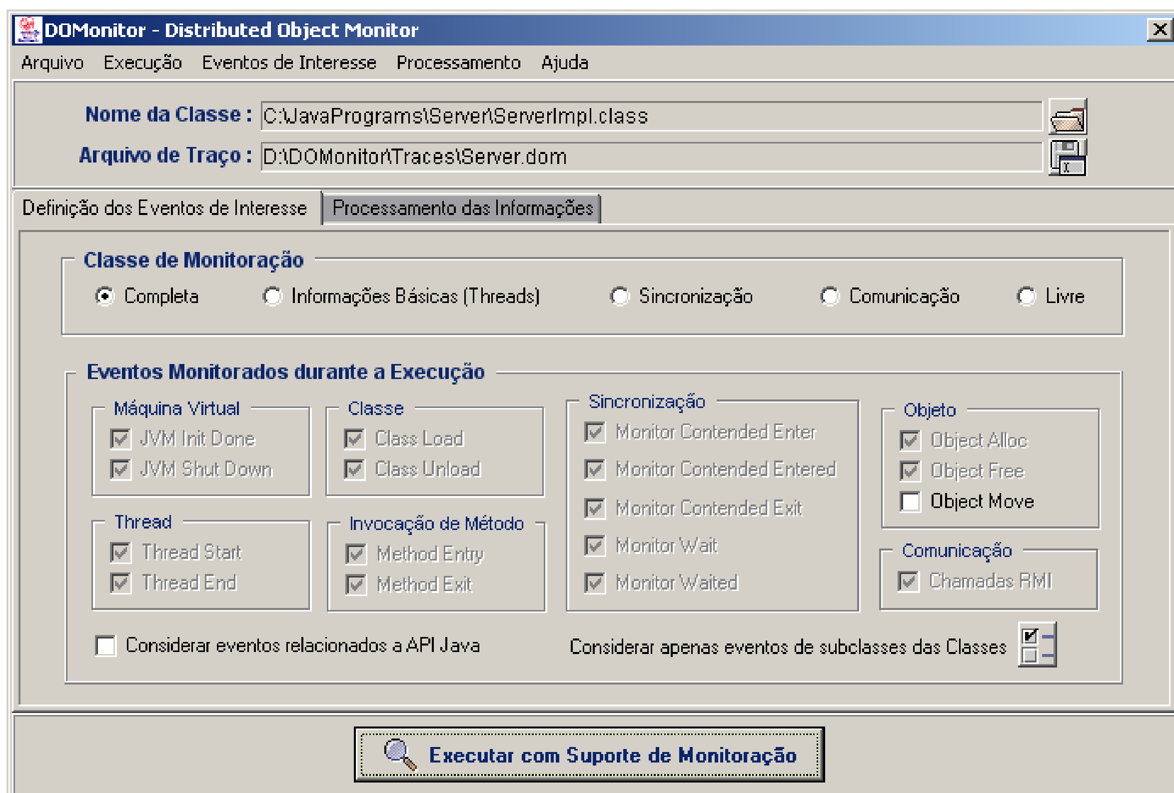


FIGURA 5.1 – Monitoração Completa

B) Informações Básicas (Threads)

Nesta classe de monitoração são considerados eventos de apenas duas categorias: *Máquina Virtual* e *Threads* (figura 5.2). As informações obtidas por esta classe referem-se apenas ao ciclo de vida das *threads* que compõe a aplicação. São geradas informações referentes à criação das *threads* e à destruição das mesmas. O

overhead gerado pela ferramenta nesta classe é praticamente inexistente, uma vez que o número de eventos monitorados é muito pequeno e que a freqüência com que estes eventos ocorrem é baixa.

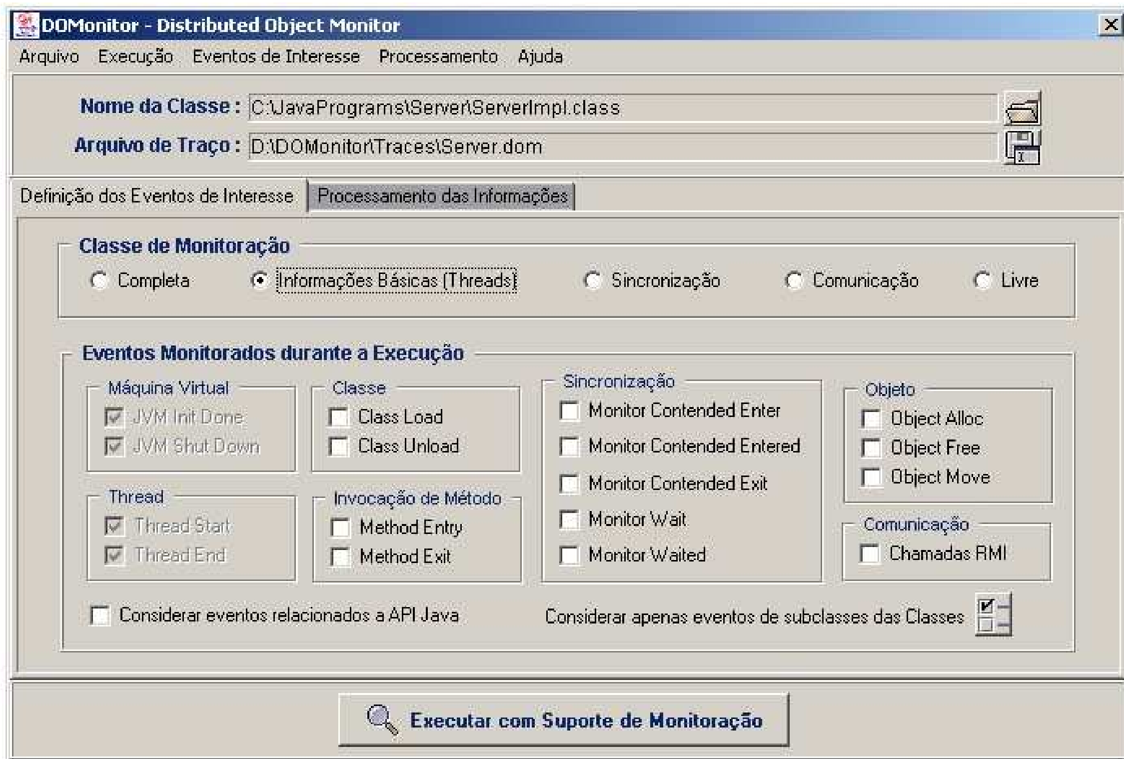


FIGURA 5.2 – Monitoração de Threads

C) Sincronização

É possível perceber que essa classe de monitoração é uma especialização da classe de Informações Básicas. Além das categorias *Máquina Virtual* e *Threads*, são selecionadas diversas outras categorias (figura 5.3). As informações obtidas com esta classe permitem a identificação das *threads* que irão executar de forma concorrente para alcançar a finalização de uma tarefa. Essas *threads* poderão compartilhar recursos, e fazer isso de maneira organizada utilizando sincronização. É possível identificar os objetos de classes da aplicação do usuário que foram criados e os métodos que foram chamados. É possível verificar também os objetos onde houve contenção por recursos.

Os três primeiros eventos da categoria sincronização são relacionados aos métodos sincronizados (*synchronized*) e os outros dois eventos são relacionados aos bloqueios gerados por chamadas `wait` e que foram posteriormente liberados por chamadas a `notify` ou `notifyAll`. Alem dos bloqueios gerados por `wait`, através dos dois últimos eventos da categoria sincronização é possível identificar as chamadas `sleep`.

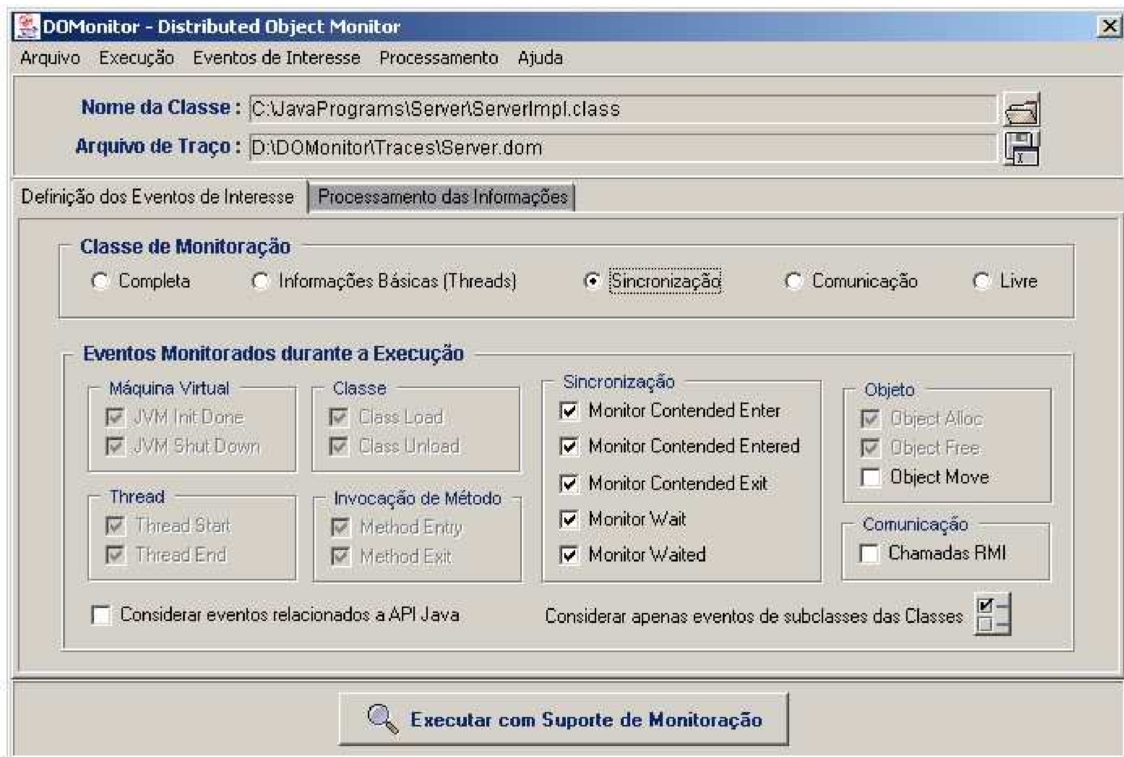


FIGURA 5.3 – Monitoração da Sincronização

D) Comunicação

Nesta classe de monitoração são obtidas informações relacionadas à interação entre objetos que executem em máquinas virtuais diferentes. Os eventos de interesse monitorados podem ser observados na figura 5.4.

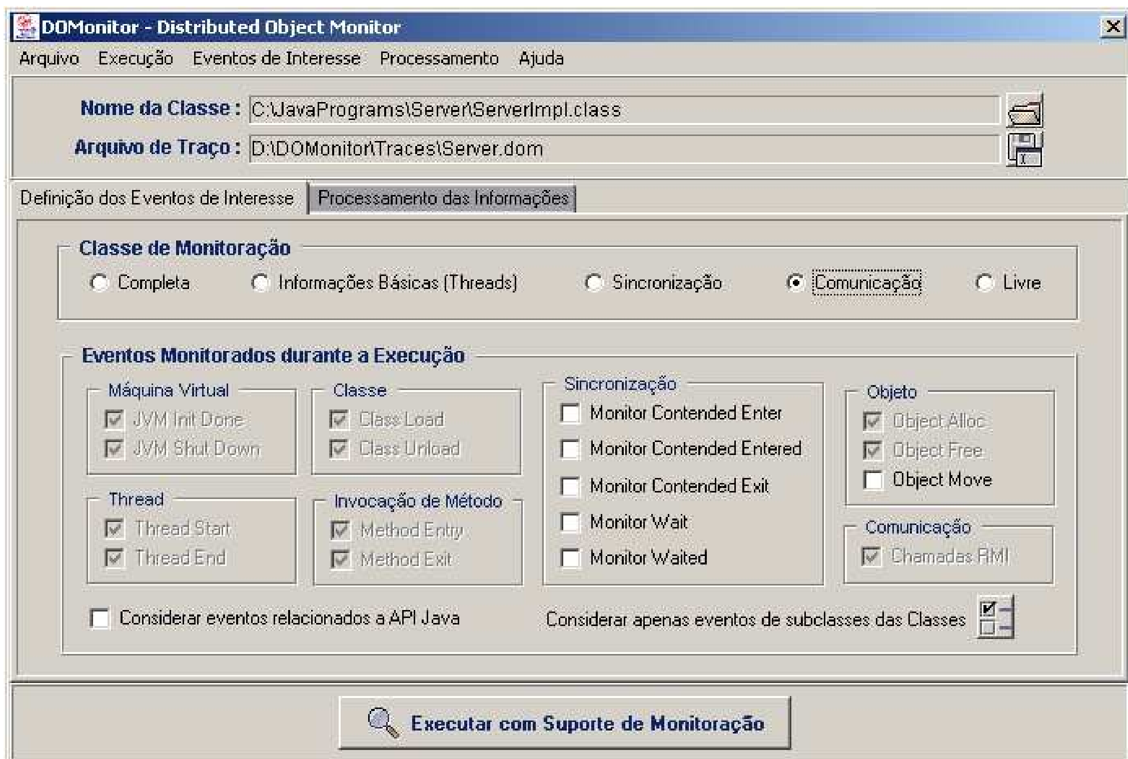


FIGURA 5.4 – Monitoração da Comunicação

Informações obtidas nesta classe incluem o comportamento dinâmico das *threads*, as classes e objetos da aplicação que foram criados, as chamadas de métodos e principalmente as invocações de métodos de objetos remotos. Ou seja, a partir destes eventos de interesse é possível traçar a interação entre clientes e servidores através de chamadas RMI.

E) Livre

Esta classe de monitoração foi disponibilizada para permitir uma maior flexibilidade ao usuário, uma vez que este seleciona os eventos de interesse sem nenhuma restrição por parte do DOMonitor. Para isto, o usuário precisaria ter um conhecimento prévio sobre os eventos disponíveis e a relação entre estes eventos, para não deixar de fora eventos imprescindíveis para o tipo de informações que deseja obter.

Os únicos eventos previamente selecionados nesta classe são os da categoria *Máquina Virtual* (figura 5.5). Com apenas estes eventos selecionados, o *overhead* gerado pela ferramenta é inexistente, pois o agente de monitoração foi carregado mas não recebe a notificação de nenhum evento. A escolha por apenas estes dois eventos parece não ter sentido. No entanto, esta classe será utilizada quando o objetivo for monitorar aplicações para o ISAM. O agente de monitoração será inicializado e ficará aguardando requisições do ISAM por meio de chamadas de funções, como será detalhado no capítulo 7.

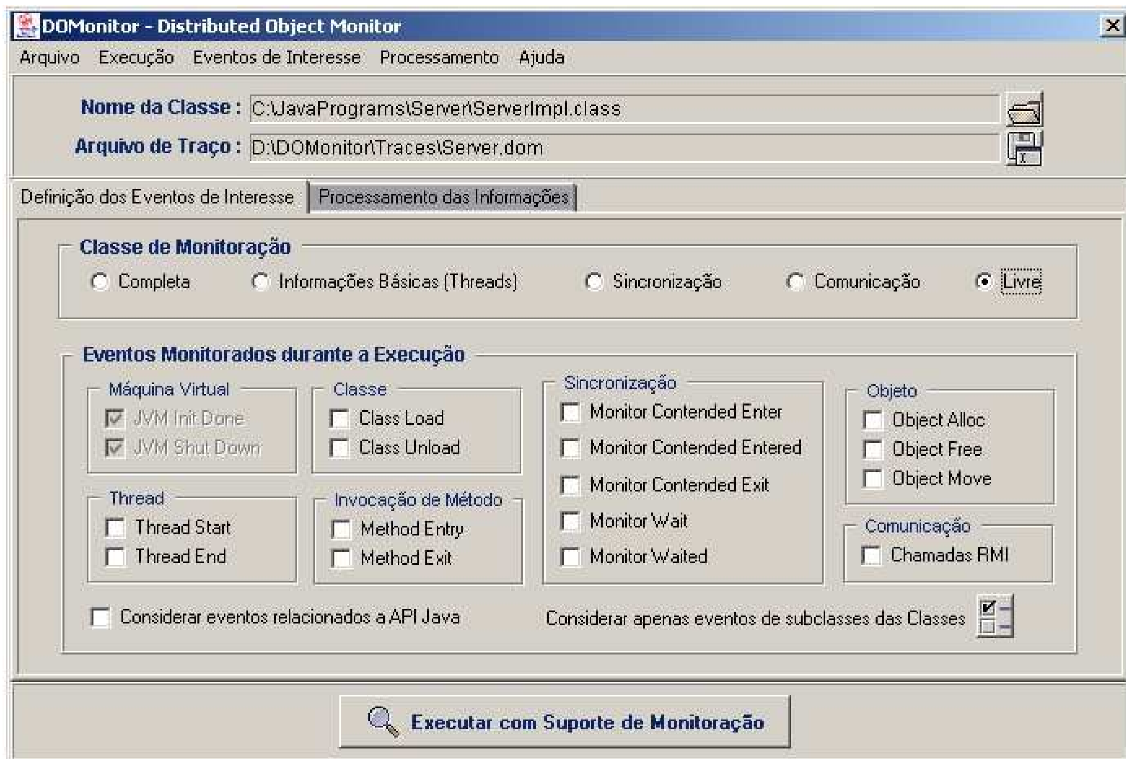


FIGURA 5.5 – Monitoração Livre

5.1.2 Definição das Opções de Filtro

A introdução de mecanismos para filtrar e selecionar as informações coletadas habilita que o usuário limite significativamente a quantidade de dados coletados e direcione seu foco para as partes mais importantes da atividade do sistema monitorado. Além disso, opções selecionadas a partir da interface gráfica permitem ao

usuário especificar uma política, definindo grupos ou níveis de objetos aos quais se tem interesse em um determinado processo de monitoração.

Java é uma linguagem Orientada a Objetos "pura" no sentido que o menor bloco de construção é a classe. Os programas Java são construídos definindo classes, criando objetos, sendo que cada objeto pertence a uma determinada classe. Cada objeto implementa uma série de métodos para realizar determinadas ações. Sem dúvida, os eventos relacionados a estes três componentes da orientação a objetos (classes, objetos e métodos) são os que possuem a maior freqüência de ocorrência. Destes eventos, é possível afirmar que os relacionados à invocação de métodos são os que ocorrem com maior freqüência.

Das classes de monitoração apresentadas na seção anterior, pode-se verificar que estas três categorias (Classes, Objetos e Métodos) são categorias de interesse na maioria delas. Uma vez que o número de eventos dessas categorias pode ser muito grande, a quantidade de dados gerados para uma aplicação, dependendo do seu porte, pode ser enorme. Portanto, para reduzir significativamente a quantidade de dados registrados pode-se eliminar o registro de eventos relacionados à API Java. Segundo [OTT 2001], se forem eliminados os eventos relacionados à API Java, o tamanho do arquivo de traço é reduzido em mais de 30 vezes.

Os algoritmos de manipulação de informações e as estruturas de dados do DOMonitor foram projetados especificamente para esta forma de monitoração. Ou seja, o DOMonitor é otimizado para registrar os eventos relacionados ao código do usuário, desprezando os eventos relacionados à API Java. Uma outra importante opção de filtro, permite que os usuários escolham uma lista de classes das quais serão geradas informações. Isso é bastante útil quando se tem um conjunto de classes já desenvolvidas e otimizadas que são usadas em uma aplicação em desenvolvimento. Das classes já desenvolvidas, o usuário não deseja registrar informações pois já conhece seu funcionamento e já refinou seu desempenho. A partir desta opção de filtro, o usuário seleciona apenas as classes das quais tem dúvidas sobre seu comportamento e desempenho, facilitando assim sua compreensão sobre os dados obtidos e diminuindo o *overhead* de monitoração.

5.1.3 Definição do Processamento das Informações

Após definir as características de monitoração, o usuário precisa definir as características relacionadas ao processamento e utilização das informações. Tais definições também são realizadas através da interface gráfica (figura 5.6).

A primeira decisão é quanto ao momento em que os dados da monitoração serão processados, se *off-line* (após o término da execução) ou se *on-line* (durante a execução). Após escolher pelo processamento *off-line* o usuário seleciona a ferramenta que utilizará os dados de monitoração, como por exemplo, o Pajé. Além disso, é preciso definir se existe a necessidade de comportamento global, o que no caso do Pajé, é necessário.

No caso da definição pelo comportamento global, os arquivos de traço de todas as JVM's que fazem parte de execução precisam ser reunidos em um único arquivo de traço contendo todas as informações. O processamento será realizado de forma centralizada. Em um dos nodos que compõem o ambiente de execução, é necessário definir que este é o nodo de referência. A máquina centralizadora será a que possuir informações na janela *Relação de Máquinas* (figura 5.6). Essa relação contém o

nome de todas as máquinas que farão parte da execução da aplicação distribuída bem como o nome do arquivo de traço gerado por cada uma delas. Essas informações são imprescindíveis para determinar o comportamento global da execução.

Os arquivos de traço que forem gerados em outros nodos, deverão ser enviados a este nodo central. Para tanto, nas outras máquinas precisa ser definida apenas a informação *Enviar os arquivos de traço para a Máquina*. Com isso, após a execução da aplicação Java nesta máquina, os arquivos de traço gerados serão enviados ao nodo central para que o processamento possa ser realizado.

No caso de processamento *on-line* é necessário definir a ferramenta destino, por exemplo, ISAM, e a forma como os dados serão disponibilizados. Existem duas opções: a publicação dos dados ser realizada periodicamente, de acordo com o tempo estabelecido; ou a publicação de dados a partir de requisições que a ferramenta destino fará ao DOMonitor por meio de uma chamada de função, como é o caso do ISAM.

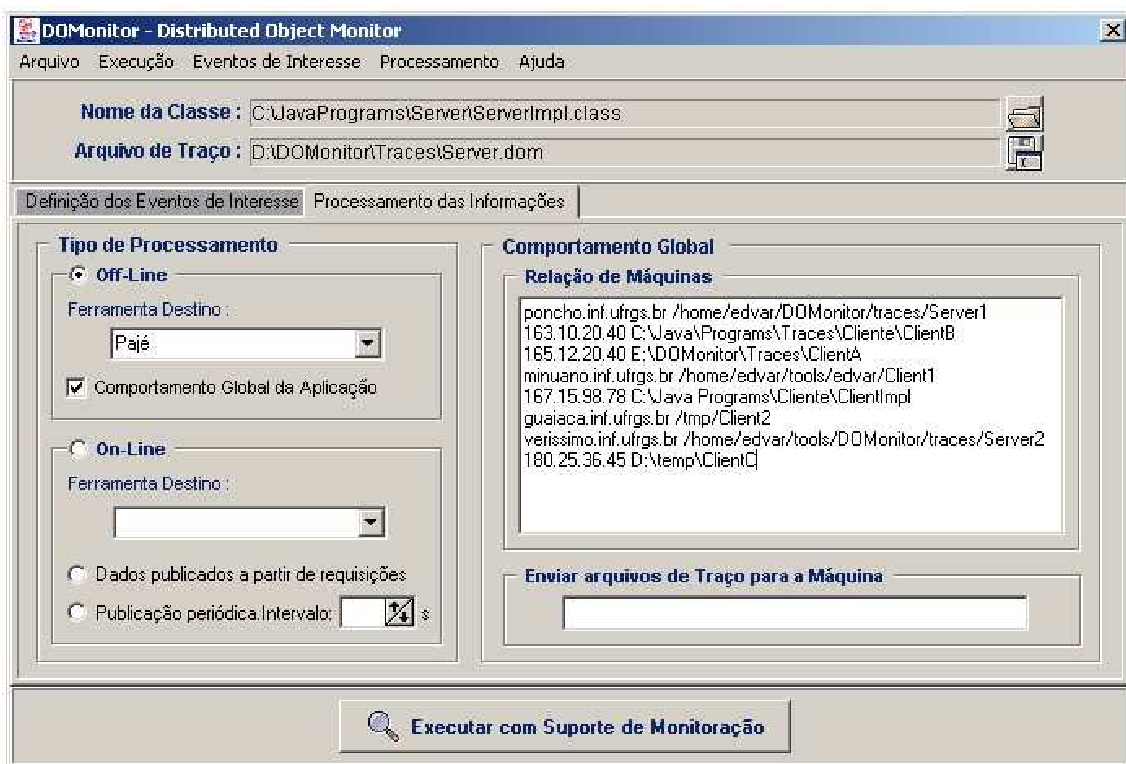


FIGURA 5.6 – Processamento das Informações

5.1.4 Arquivo de Configuração

As definições selecionadas na interface gráfica são gravadas em um arquivo de configuração com as Informações Gerais da Execução do DOMonitor (DOMonitor.conf). O conteúdo deste arquivo é utilizado pelo agente de monitoração do DOMonitor, o DOProf, logo que este é carregado. É a partir destas informações que o DOProf configura seus parâmetros de execução.

Isto permite que execuções subseqüentes com as mesmas características sejam realizadas sem a necessidade de invocar a interface. Além disso, para tornar o ambiente ainda mais genérico, esse arquivo poderá ser modificado por um editor de textos qualquer desde que sejam respeitados o formato e as informações necessárias. Com isso, a monitoração não fica restrita a máquinas onde seja possível executar a interface gráfica.

DOMonitor – Distributed Object Monitor
Informações Gerais da Execução

#Nome da Classe
C:\JavaPrograms\Server\ServerImpl.class

#Arquivo de Traço
D:\DOMonitor\Traces\Server.dom

#Classe de Monitoração
1 – Monitoração Completa

#Eventos Adicionais

#Considerar eventos da API Java
N

#Considerar eventos das Classes

#Tipo de Processamento
Off-line

#Ferramenta Destino
Pajé

#Comportamento Global
S

#Dados Publicados através de requisições

#Publicação Periódica. Intervalo

#Relação de Máquinas
poncho.inf.ufrgs.br /home/edvar/DOMonitor/traces/Server1
163.10.20.40 C:\Java\Programs\Traces\Cliente\ClientB
165.12.20.40 E:\DOMonitor\Traces\ClientA
minuano.inf.ufrgs.br /home/edvar/tools/edvar/Client1
167.15.98.78 C:\Java Programs\Cliente\ClientImpl
guaiaca.inf.ufrgs.br /tmp/Client2
verissimo.inf.ufrgs.br /home/edvar/tools/DOMonitor/traces/Server2
180.25.36.45 D:\temp\ClientC

#Enviar arquivos de traço para a máquina

FIGURA 5.7 – Layout do Arquivo de Configuração do DOMonitor

5.2 Detecção de Eventos

Para alcançar os objetivos propostos e considerando as características almejadas, a execução da aplicação Java, ou melhor, as ações (eventos) executadas em seqüência ou de forma concorrente devem ser registradas. As informações necessárias para descrever a execução da aplicação serão então organizadas em duas categorias: **informações locais** (as *threads* origem e destino estão executando na mesma JVM) e **informações remotas** (as *threads* origem e destino estão executando em JVM's diferentes).

5.2.1 Detecção das Informações Locais - JVMPI

A detecção dos eventos locais é realizada através de um agente de monitoração construído usando o JVMPI [SUN 2001a]. O JVMPI (*Java Virtual Machine Profiler Interface*) é um mecanismo portável e genérico que permite o desenvolvimento de agentes de monitoração para atuar em conjunto com a JVM. Com

isso, o agente de monitoração obtém informações sobre a execução através de um conjunto de eventos gerados durante a execução da aplicação na máquina virtual. Dentre as características do JVMPI, algumas motivaram a sua escolha:

- ***Propósito Geral*** – os agentes de monitoração não precisam de nenhuma instrumentação na máquina virtual. Além disso, o JVMPI possibilita que sejam monitorados métodos compilados por compiladores JIT;
- ***Portabilidade*** – o JVMPI faz parte do pacote Java e foi projetado para ser completamente independente da implementação da máquina virtual Java;
- ***Não-Intrusiva*** – quando a monitoração estiver desabilitada a JVM faz somente um teste de condição para cada evento disponível pelo JVMPI. A maioria dos eventos ocorre em momentos nos quais se pode tolerar o *overhead* de um teste adicional [VIS 2000]. Como resultado, a JVM pode ser distribuída com suporte de monitoração.

O coração da plataforma Java é a máquina virtual Java, considerada um computador abstrato. A principal função da JVM é carregar os arquivos de classes (.class) e executar os *bytecodes* que eles contém. A JVM é composta por dois módulos principais: ***class loader*** e ***execution engine*** (figura 5.8). O módulo *class loader* é responsável por carregar as classes tanto do programa quanto da API Java (apenas as realmente necessárias). Os *bytecodes* são executados pela *execution engine* [SUN 2001a].

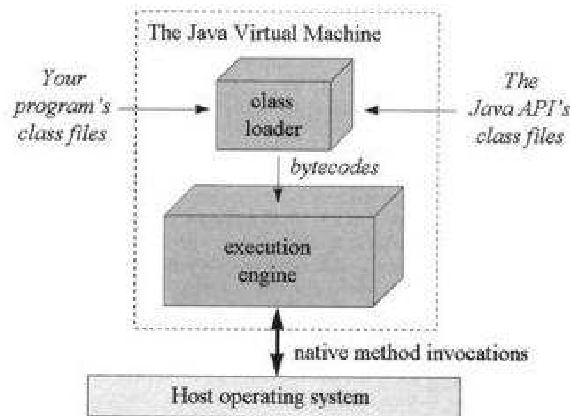


FIGURA 5.8 – JVM implementada por software sobre um S.O.

Dentro da JVM, mais especificamente na *execution engine*, está o módulo responsável por detectar os eventos (monitoração no ambiente de execução). No agente de monitoração (DOProf) fica toda a parte de controle, armazenamento e definição dos eventos de interesse, entre outras atividades. O JVMPI é apenas uma interface que permite a comunicação entre o DOProf e a JVM. Isso é feito a partir de um conjunto de funções que são disponibilizadas e de estruturas de dados bem definidas. A figura 5.9 mostra a arquitetura geral de um sistema baseado em JVMPI.

O DOProf é inicializado durante o processo de inicialização da JVM, antes do começo da execução da aplicação. O DOProf executa no mesmo processo que a JVM. Como as chamadas às funções são no mesmo processo, a intrusão por parte da ferramenta de monitoração é mínima. No entanto, a implementação do DOProf foi cuidadosa em tratar com questões relativas a *threads* e *locks* para prevenir corrupção de

dados (das informações geradas) e *deadlocks* (que podem inclusive travar a JVM e a aplicação em execução).

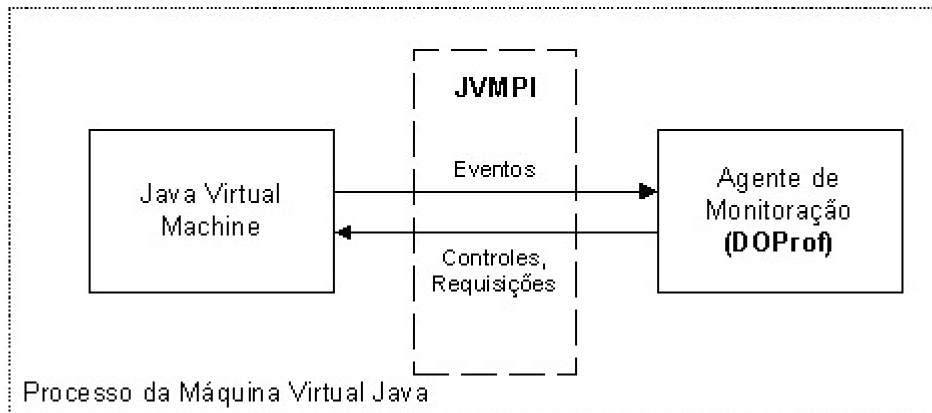


FIGURA 5.9 – Arquitetura de monitoração utilizando JVMPI

A ferramenta que usará os resultados da monitoração é um processo distinto, muitas vezes executado após o término da execução da aplicação. Isto é realizado, principalmente, para evitar a interferência com a aplicação. Viswanathan [VIS 2000] afirma que é possível escrever agentes de monitoração compactos que delegam tarefas que consomem muitos recursos para etapas posteriores de processamento, de tal forma que executar o DOProf no mesmo processo que a máquina virtual não interfira excessivamente com as informações da execução.

O suporte de monitoração está presente em todas as implementações atuais da JVM. Este suporte permanece inativo até que seja explicitamente ativado. O primeiro passo para a ativação deste suporte é a carga do agente de monitoração que é realizada pela JVM. Após a carga, está inicializada a forma pela qual o DOProf irá interagir com a máquina virtual para obter os eventos gerados na execução da aplicação. Inicialmente, a monitoração de todos os eventos disponibilizados pela JVM está desabilitada (figura 5.10). Portanto, o DOProf estará carregado em conjunto com a JVM, mas não será notificado sobre a ocorrência de evento algum.

FIGURA 5.10 – Eventos Monitorados não Definidos

A máquina virtual tem a possibilidade de monitorar um conjunto finito de eventos (figura 5.10). A variedade de eventos disponibilizados é suficiente para a maioria das ferramentas que precisam obter dados sobre a execução de uma aplicação

Java [VIS 2000]. Para mais detalhes sobre os eventos disponibilizados pela JVM, consultar [VIS 2000] e [SUN 2001a]. Dependendo do tipo de monitoração desejada, o DOMonitor instrui a máquina virtual a enviar somente os eventos relevantes. O DOProf precisa então, utilizando-se de uma função específica da JMVPI, informar a JVM sobre os eventos para os quais deseja receber notificação (figura 5.11).

FIGURA 5.11 – Definição dos Eventos Monitorados

O DOProf pode habilitar e desabilitar a notificação de eventos a qualquer momento durante a execução, ou seja, a determinação dos eventos de interesse é realizada dinamicamente e pode ser alterada conforme conveniência do DOProf. Quando a JVM detecta a ocorrência de um evento, verifica se este é um evento de interesse do DOProf. Em caso positivo, notifica o DOProf sobre a ocorrência do evento enviando informações associadas a ele. Caso contrário, o agente de monitoração não recebe nenhum tipo de notificação sobre a ocorrência do evento. As informações que serão registradas para cada tipo de evento estão descritas na seção 5.3.

A questão da perturbação da aplicação decorrente da presença da ferramenta de monitoração deve ser considerada. Embora a intrusão possa ser reduzida utilizando-se uma implementação cuidadosa, ela não pode ser eliminada. A maior causa da perturbação é relacionada à geração e gravação do traço de eventos em disco. Perturbações secundárias podem ser geradas devido aos retardos acumulados de cada geração de evento.

A escolha por alguns eventos de interesse dentre os vários eventos que podem ser monitorados pela máquina virtual já é uma decisão tomada com bastante cuidado para monitorar realmente apenas os tipos de eventos imprescindíveis. Para reduzir a quantidade de informações geradas, algumas ocorrências de determinados eventos podem ser desprezadas, ou seja, nenhuma informação ser registrada no traço de eventos. Por exemplo, quando se opta pela notificação do evento *Class Load* o DOProf passa a receber informações sobre todas as classes carregadas pela JVM. Grande parte das classes carregadas é da API Java, portanto, apenas desprezando o registro das informações dos eventos relacionados à API Java obtém-se uma diminuição enorme no tamanho do arquivo de traço e conseqüentemente no *overhead*.

Uma vez que operações de disco consomem tempo de processamento é importante minimizar o número de acessos a disco. Isso é realizado com o auxílio de áreas locais de armazenamento (buffers). No entanto, como o tamanho do buffer é limitado, os acessos a disco não são completamente eliminados, pois os buffers

precisam ser descarregados em disco quando se tornarem cheios. Essa alternativa reduz bastante o número de gravações em disco, reduzindo com isso significativamente a perturbação na aplicação. Além disso, a descarga dos buffers em disco é realizada por uma *thread* específica de baixa prioridade.

Para reduzir ainda mais o tamanho do arquivo de traço, o tipo e a quantidade de informações nele armazenadas foi cuidadosamente planejada. Por exemplo, considerando o evento com maior taxa de ocorrência, as invocações de métodos, ao invés de armazenar o nome do método (normalmente grande, algumas vezes ultrapassando os 100 bytes) cada vez que ele for executado é armazenado apenas um *identificador de método* com 4 bytes. A associação do identificador do método a seu nome é feita durante o processamento das informações.

5.2.2 Detecção das Informações Remotas – *daemon* EXEd

A monitoração das invocações remotas de métodos exige informações adicionais comparadas às registradas para chamadas de métodos locais. No caso das invocações remotas, é necessário obter no mínimo a JVM onde o método foi realmente executado. Uma vez que na JVM-server pode haver diversas *threads* em execução, é desejável saber também por qual das *threads* foi executado o método. Portanto, deseja-se saber que um determinado método de um objeto remoto qualquer foi executado em um nodo por uma *thread*.

Para isso, além das informações relacionadas à chamada do método que podem ser obtidas através do JVMPI, é necessário registrar a conexão criada entre as JVM’s para efetivamente proceder com a chamada RMI. Desta forma, é utilizado um processo adicional para registrar informações essenciais para uma futura reconstrução das invocações remotas de métodos. Esse processo é utilizado devido à grande dificuldade de obter tais informações na versão atual do JVMPI [SUN 2001a]. Nada impede que, no caso de novas versões do JVMPI com eventos desenvolvidos especificamente para esse objetivo, essas informações passem a ser também registradas com o JVMPI.

Invocações remotas são sempre transportadas através da camada de rede fornecida pelo sistema operacional, independente de o cliente e o servidor estarem executando na mesmo *host* ou em *hosts* diferentes. Como conseqüência, o suporte de rede deve estar ativo mesmo em um *host* isolado (*standalone*) no qual se deseja a funcionalidade de invocações remotas de métodos. A figura 5.12 mostra o fluxo das informações entre o objeto cliente e o objeto servidor quando as JVM’s estão em execução no mesmo *host* [HAR 97].

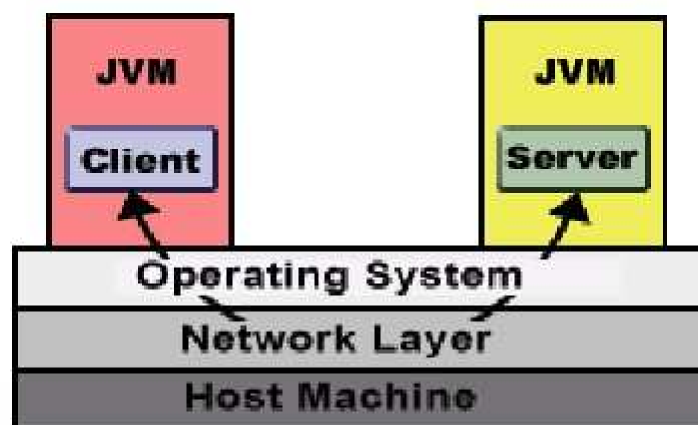


FIGURA 5.12 – Cliente e Servidor executam no mesmo *host*

Considerar as chamadas entre JVM's diferentes como chamadas remotas tem a vantagem de desvincular o ambiente de execução da arquitetura disponível para execução. A vinculação do que foi executado em uma determinada máquina será realizada considerando a relação JVM/máquina. Isso permite identificar a quantidade de tráfego de rede que seria gerado se em uma outra execução as duas JVM's fossem dispostas em máquinas distintas.

Na figura 5.13, é demonstrada a situação onde as JVM's estão executando em *hosts* diferentes. Para a perspectiva do programador, as duas situações são idênticas. No entanto, normalmente o sistema operacional irá otimizar as comunicações que ocorrem entre processos que residam no mesmo *host*.

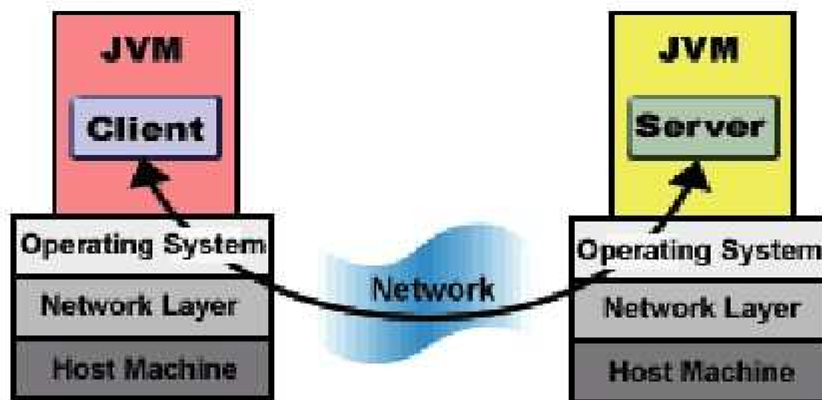


FIGURA 5.13 – Cliente e Servidor executam em *hosts* diferentes

As informações sobre as atividades entre cliente e servidor são obtidas através de uma extensão ao *daemon* EXEd. O *daemon* EXEd [SIL 2001] é composto por dois mecanismos principais: um primeiro para dar suporte à instanciação remota de objetos Java e um segundo responsável por coletar informações dinâmicas sobre a execução da aplicação distribuída. Também integram a proposta mecanismos para otimização nas comunicações e para a construção de perfis de comunicação inter-objetos. Os mecanismos são integrados com a API RMI de Java, buscando preservar a natureza da Orientação a Objetos na construção de aplicações distribuídas, e conseqüentemente a compatibilidade com a semântica nativa da linguagem Java.

A integração com a API RMI se dá pela utilização de objetos da classe proposta *VSocket*, a qual oferece, para o nível da aplicação, a abstração de uma conexão orientada a byte (*stream*) como esperado pelo nível de RMI – cuja implementação padrão está baseada em *sockets* TCP/IP. Cada objeto *VSocket* possui um *EXEid,* requisitado junto ao *EXEd*, estando portanto habilitado a utilizar as primitivas de comunicação disponibilizadas pelo mesmo. Desse modo, é possível aos objetos *VSocket* exercerem o papel de redirecionadores da comunicação executada no nível RMI para o *EXEd*. A partir do interfaceamento das primitivas de troca de mensagens, o *VSocket* fornece a semântica de uma conexão orientada a bytes, permitindo a instrumentação da quantidade de dados trocada entre os objetos. O DOMonitor utiliza esta proposta e faz pequenas extensões para permitir que seja obtida a *thread* na qual o objeto foi executado. A inicialização deste processo é realizada em conjunto com a inicialização do agente de monitoração.

5.3 Coleta das Informações Locais

As informações locais são detectadas e registradas através do DOProf que interage com o ambiente de execução (máquina virtual) por meio do JVMPI. Uma vez que a execução é monitorada de forma independente por cada JVM e as informações coletadas (traço de eventos) são armazenadas em arquivos distintos, todos os eventos registrados dentro de um mesmo arquivo de traço estão relacionados à mesma máquina virtual. Com isso, não é necessário registrar em qual JVM os eventos foram executados.

O JVMPI se refere às entidades na JVM utilizando ID's. Cada ID possui um **Evento de Definição** e um **Evento de Indefinição**. Um evento de definição fornece informações relacionadas a um ID. Um ID é válido até que ocorra o evento de indefinição, sendo que este valor pode ser reusado posteriormente para qualquer tipo de entidade. A tabela 5.1 apresenta o evento de definição e o evento de indefinição para cada um dos ID's que são tratados pelo DOMonitor.

TABELA 5.1 – Eventos de Definição e Indefinição de cada tipo de ID

ID	Evento de Definição	Evento de Indefinição
T_{ID}	Thread Start	Thread End
O_{ID}	Object Alloc	Object Free
C_{ID}	Class Load	Class Unload
M_{ID}	Defining Class Load	Defining Class Unload

Quando ocorre o evento de definição, é armazenado o T_{INI}, o ID e a descrição. Quando ocorrem os demais eventos relacionados ao ID, é gravado apenas o ID e o T(tempo). No evento de indefinição, são gravados o ID e o T_{FIM}. Assim, tem-se a relação ID $\leftrightarrow$ Descrição, em cada instante de tempo. Isso evita que seja armazenada a descrição a cada ocorrência de eventos relacionados a um determinado ID.

Cada um dos registros (linhas) do arquivo de trace é uma tupla que segue um dos formatos que serão apresentados no decorrer deste capítulo. Com isso, o formato das informações coletadas fica definido de acordo com as tuplas que foram originadas e armazenadas nesse arquivo.

Dos eventos de interesse monitorados, apenas dois não possuem informações específicas e não irão produzir linhas no traço de eventos da execução. Tais eventos são gerados apenas uma vez e são eles: `JVM InitDone` e `JVM ShutDown`. O evento `JVM InitDone` é usado pelo DOProf para inicializar suas estruturas de dados e criar *threads* do ambiente de monitoração. O evento `JVM ShutDown` é usado para descarregar para os arquivos de traço as informações que ainda estejam em buffers na memória, liberar todas as estruturas de dados utilizadas e encerrar as *threads*. Tais procedimentos são mais bem detalhados no capítulo 8. Todos os outros eventos geram informações específicas que serão registradas no traço de eventos. As próximas seções detalham as informações armazenadas.

5.3.1 Execução das *Threads*

Em contraste aos programas seqüenciais, os quais possuem um único fluxo de controle, sistemas distribuídos possuem vários fluxos de controle (*threads*). Essas *threads* podem ser criadas ou destruídas a qualquer instante durante a execução da

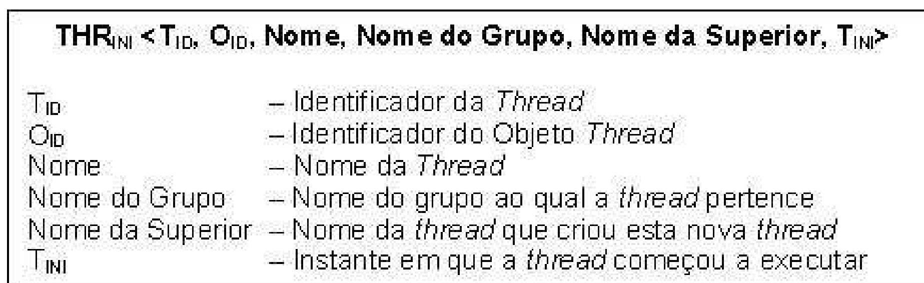
aplicação. A monitoração do comportamento dinâmico das *threads* obtém informações relativas ao ciclo de vida de cada uma delas.

A coleta das informações relativas às *threads* é realizada baseada em dois eventos de interesse previamente especificados junto a máquina virtual: **Evento de Criação da Thread** (`Thread Start`) e **Evento de Destruição da Thread** (`Thread End`). As informações obtidas com esses eventos identificam as *threads* existentes na aplicação e o seu comportamento dinâmico. Estas informações são armazenadas em tuplas, uma relacionada ao início da execução da *thread* (THR_{INI}) e outra relacionada ao fim da execução da *thread* (THR_{FIM}).

A execução de aplicações Java normalmente começa com a invocação do método `main()`. No entanto, o ambiente de execução cria *threads* auxiliares de sistema não disparadas pela aplicação do usuário. O DOMonitor desconsidera todas as *threads* da JVM, preocupando-se apenas com as *threads* da aplicação.

Quando o DOProf recebe a notificação de uma nova *thread*, verifica se essa é uma *thread* de sistema ou da aplicação. No caso de ser uma *thread* da aplicação é criada uma área de armazenamento local, que é utilizada para armazenar o traço de eventos desta *thread*. Essa área de armazenamento é chamada neste trabalho de **Buffer Local de Thread**. Após criado o buffer, ele já começa recebendo os dados de identificação da *thread*, ou seja, uma tupla THR_{INI} (figura 5.14).

Eventos são enviados na mesma *thread* em que eles foram gerados. Por exemplo, um evento relacionado a uma chamada de método é enviado na mesma *thread* na qual o método foi invocado. Todos os eventos gerados por uma *thread* são armazenados no Buffer Local da Thread. Os armazenamentos locais são reunidos ou descarregados para disco em momentos específicos ou quando se tornarem cheios.

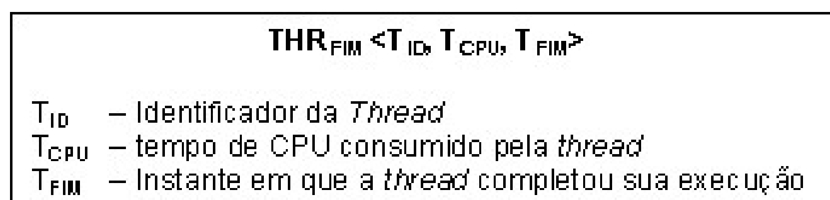


THR_{INI} <T_{ID}, O_{ID}, Nome, Nome do Grupo, Nome da Superior, T_{INI}>

T_{ID} – Identificador da *Thread*
O_{ID} – Identificador do Objeto *Thread*
Nome – Nome da *Thread*
Nome do Grupo – Nome do grupo ao qual a *thread* pertence
Nome da Superior – Nome da *thread* que criou esta nova *thread*
T_{INI} – Instante em que a *thread* começou a executar

FIGURA 5.14 – Tupla THR_{INI}

O Buffer Local da *Thread* permanece válido durante todo o ciclo de vida da *thread*. No momento em que ocorre o evento de destruição da *thread*, o buffer recebe as informações relacionadas ao evento e as armazena em uma tupla do tipo THR_{FIM} (figura 5.15). Com isto, o buffer local da *thread* possui informações sobre todos os eventos relacionados a esta *thread*. Este buffer será então descarregado em disco e esta área de memória liberada.



THR_{FIM} <T_{ID}, T_{CPU}, T_{FIM}>

T_{ID} – Identificador da *Thread*
T_{CPU} – tempo de CPU consumido pela *thread*
T_{FIM} – Instante em que a *thread* completou sua execução

FIGURA 5.15 – Tupla THR_{FIM}

5.3.2 Classes

Quando o DOProf recebe a notificação de que uma classe foi carregada, a primeira ação é verificar se essa classe faz parte das classes para as quais devem ser registradas informações, ou seja, são testadas as condições para verificar se a classe carregada é uma **Classe de Interesse**. A opção padrão é por não registrar informações relacionadas às classes da API Java. Adicionalmente, o usuário pode, a partir da interface gráfica, ter selecionado a monitoração de apenas algumas classes da aplicação. Para as classes que satisfizerem as condições, serão registradas informações sobre sua carga, possibilitando assim que informações relacionadas à criação dos objetos dessa classe sejam também registradas.

O evento gerado quando uma classe é carregada na JVM é o `Class Load`. As informações relativas a esse evento incluem dados da classe carregada e de todos os métodos dessa classe. A partir destas informações, é gerada uma Tupla CLA_{INI} (figura 5.16), contendo as informações relacionadas à classe, um registro na **Tabela Hash de Classes** e tantos registros na **Tabela Hash de Métodos** quantos forem o número de métodos desta classe.

CLA_{INI} <C_{ID}, T_{ID}, Nome da Classe, Nome do Fonte, T_{INI}>

C_{ID}	– Identificador da Classe
T_{ID}	– Identificador da *Thread* que gerou a carga da classe
Nome da Classe	– Nome da Classe
Nome do Fonte	– Nome do Arquivo Fonte em que está definida a classe
T_{INI}	– Instante em que a classe foi carregada

FIGURA 5.16 – Tupla CLA_{INI}

Após ter sido registrada a tupla relacionada à carga da classe, é inserido um registro na **Tabela Hash de Classes** (tabela 5.2) contendo o C_{ID}, o número de métodos e os M_{ID} de todos os métodos da classe. Uma classe permanece nessa estrutura até ocorrer o evento de indefinição correspondente. Portanto, esta tabela é modificada a cada vez que uma classe de interesse é carregada pela JVM ou descarregada da mesma. Esta tabela serve como suporte para a identificação das informações que devem ser geradas no traço de eventos, ou seja, serão armazenados eventos somente relacionados às classes que estiverem nesta tabela.

A tabela 5.2 exemplifica como ficaria a Tabela Hash de Classes após três classes de interesse terem sido carregadas. Esta tabela armazena o identificador da classe, o número de métodos que a classe possui e uma lista com todos os identificadores de métodos da classe. A chave de procura da tabela é o C_{ID} para permitir consultas rápidas nesta tabela. Detalhes adicionais sobre a utilização da tabela também são expostos na seção 5.3.3.

TABELA 5.2 – Tabela Hash de Classes

C_{ID}	N	Identificadores dos Métodos
…		
70501248	2	2356, 189564
…		
693857	5	158569, 1058, 785412, 54870025, 65824585
…		
5310	3	145875, 745896, 14528742
…		

Uma vez que a definição dos métodos de uma determinada classe se dá no momento da definição da mesma, as informações sobre os métodos são construídas no momento do evento `Class Load`. Na **Tabela Hash de Métodos** é inserido um registro para cada um dos métodos com informações de identificação e um campo `booleano` que indica se o método foi invocado. Os métodos são acessados e unicamente identificados pelo seu identificador. O valor do *flag* `inv` é inicialmente `false` para todos os métodos e passa para `true` somente para os métodos invocados. Essa tabela é maior que a tabela Hash de Classes.

TABELA 5.3 – Tabela Hash de Métodos

M_{ID}	Nome do Método	Linha	Inv
…			
54870025	desenhaReta	2	true
…			
2356	getSaldo	25	false
…			
158569	desenhaCirculo	89	false
…			
65824585	desenhaQuadrado	12	true
…			
14528742			false
…			
1058	desenhaRetângulo	203	false
…			
785412	desenhaLosângulo	125	false
…			
189564	setSaldo	38	true
…			
145875			false
…			
745896			false

Quando o DOProf recebe a notificação do evento de indefinição das classes, o `Class UnLoad`, o agente verifica na Tabela Hash de Classes se esta é uma classe de interesse. Em caso afirmativo, esta linha da tabela de classes deverá ser excluída. No entanto, o evento de indefinição da Classe também é o evento de invalidação de todos os métodos desta classe. Com isso, no momento em que uma classe vai ser descarregada

da memória os identificadores de método também serão invalidados. Por isso, deverão ser excluídas todas as linhas da Tabela de Métodos correspondentes a métodos implementados pela classe. Esse é o objetivo de, na tabela de Classes, ser armazenado o identificador de todos os métodos, para que a pesquisa na tabela de métodos seja realizada de forma rápida, através do acesso à chave principal.

Para cada um dos métodos implementados pela classe, verifica-se se este foi invocado ou não (consulta ao *flag* `inv`). Para um método que foi invocado são armazenadas as informações correspondentes no buffer local da *thread*, seguindo as informações da tupla MET_{INF} (figura 5.17). Para os métodos que não foram invocados, nenhuma informação é registrada.

Quando se define uma classe, é definido um conjunto de métodos para esta classe. Muitos dos métodos definidos pela classe podem não ter sido invocados durante a execução da aplicação monitorada. O *flag* `inv` é usado para diminuir a quantidade de informações geradas no arquivo de traço. Com ele, é possível armazenar informações relacionadas apenas aos métodos que foram invocados durante a execução.

As informações armazenadas na tupla MET_{INF} possuem informações relacionadas a um identificador de método para uma determinada vigência. Pois, uma vez que o evento que tem a maior taxa de ocorrência são os relacionados a execução de métodos, no registro de invocação de método armazena-se somente o identificador e em outro registro, armazena-se as informações relativa ao método.

$MET_{INF} < M_{ID}, C_{ID}, \text{Nome do Método}, \text{Linha}, T_{INI}, T_{FIM} >$

M_{ID} – Identificador do Método
C_{ID} – Identificador da Classe que implementa o método
Nome do Método – Nome do Método
Linha – Número da linha onde começa o código do método
T_{INI} – Instante em que a definição do método foi realizada
T_{FIM} – Instante em que o método foi descarregado

FIGURA 5.17 – Tupla de Informações do Método

Por fim, após gravar os registros correspondentes às informações dos métodos e liberar a memória utilizada na Tabela de Métodos, são armazenadas as informações relativas à descarga da classe em uma tupla CLA_{FIM} (figura 5.18). O último procedimento realizado é excluir esta linha da Tabela de Classes.

Este processo é que permite o controle da geração de traço apenas para classes que não sejam da API Java e para classes adicionais informadas pelo usuário. Caso contrário não seria possível identificar quando deveriam ser geradas informações relacionadas à carga e descarga de classes. Além disso, esse processo é fundamental para controlar os eventos relacionados a objetos e às chamadas de métodos, como ficará claro logo a seguir.

$CLA_{FIM} < C_{ID}, T_{ID}, T_{FIM} >$

C_{ID} – Identificador da Classe
T_{ID} – Identificador da *Thread*
T_{FIM} – Instante em que a classe foi descarregada

FIGURA 5.18 – Tupla de Descarga da Classe

5.3.3 Objetos

Os objetos em Java são instâncias de uma determinada classe. Estas instâncias são criadas dentro do programa antes de serem utilizadas. Esta criação gera um evento, o evento `Object Alloc`. Quando o DOProf recebe a notificação deste tipo de eventos, a primeira ação realizada é verificar se a classe deste objeto é uma classe de interesse, ou seja, se o C_{ID} relacionado ao objeto que está sendo criado está na Tabela de Classes. Em caso afirmativo, são armazenadas as informações relacionadas à criação do objeto, seguindo o formato da tupla OBJ_{INI} (figura 5.19). Caso contrário, nenhuma informação é registrada. Com isso, é possível verificar que só são armazenadas informações a objetos que sejam instâncias de classes de interesse.

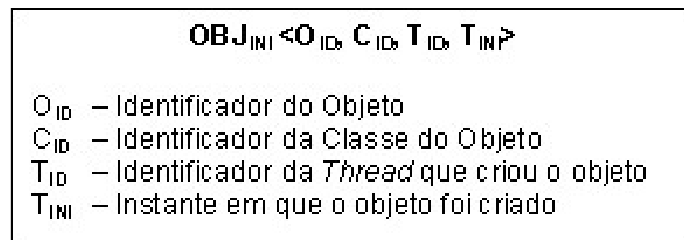


FIGURA 5.19 – Tupla de Criação de Objeto

No momento que um objeto é destruído, a máquina virtual notifica o DOProf sobre a ocorrência do evento `Object Free`. Este evento corresponde à invalidação do O_{ID} associado ao objeto. As informações relacionadas a este evento são armazenadas na tupla OBJ_{FIM} (figura 5.20).

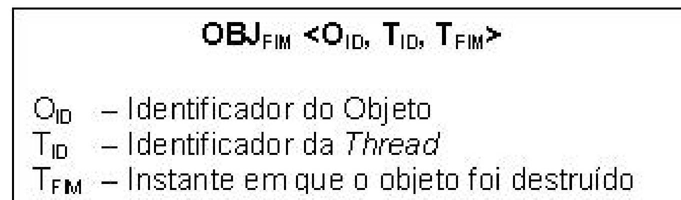


FIGURA 5.20 – Tupla de Destruição do Objeto

O evento `Object Move` é monitorado eventualmente para suportar a movimentação dos objetos pelas áreas de memória de Java. Essa movimentação ocasiona a mudança no O_{ID} do objeto. As informações relacionadas a este evento são registradas, seguindo o formato da tupla OBJ_{MOV} (figura 5.21).

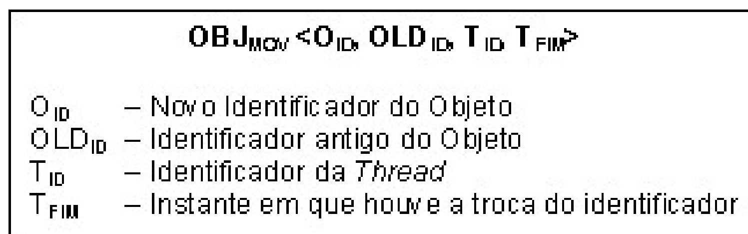


FIGURA 5.21 – Tupla de Movimentação de Objetos

5.3.4 Invocação de Métodos

As informações sobre a invocação de métodos são coletadas com dois objetivos principais: (i) saber qual método está em execução; (ii) identificar o método que causou comunicação entre as máquinas. A execução de métodos é monitorada também para saber quais métodos consomem mais tempo dentro da *thread*, e por conseqüência, quais métodos devem ser otimizados para melhorar o desempenho.

As chamadas de métodos na aplicação Java fazem com que o DOProf receba a notificação de ocorrência do evento `Method Entry`. O agente de monitoração precisa ser extremamente ágil na manipulação deste evento, pois este é o que ocorre com maior freqüência durante a execução da aplicação Java. Por outro lado, uma vez que a taxa de ocorrência é bastante elevada, o armazenamento de informações sobre a invocação de todos os métodos poderia elevar consideravelmente o tamanho do arquivo de traço e por conseqüência aumentar o número de acessos a disco e o *overhead*.

A solução adotada no DOProf é registrar informações apenas para os métodos de interesse. São métodos de interesse todos que estiverem na Tabela de Métodos. A Tabela de Métodos é utilizada justamente para agilizar a pesquisa pelos métodos de interesse. Na situação ideal, duas instruções são necessárias para verificar se um método é de interesse ou não. Uma primeira instrução para gerar a chave de acesso à Tabela de Métodos, e uma segunda instrução para verificar a existência do M_{ID} na tabela. Em caso afirmativo, é atribuído o valor *booleano* `true` para o campo `inv` da tabela. Além disso, são armazenadas informações relacionadas à invocação do método segundo o formato da tupla MET_{INI} (figura 5.22). Invocações de um mesmo método por threads diferentes são distinguidas uma da outra pelo identificador da thread que é único em um determinado momento. Caso contrário, nenhuma informação é gerada.

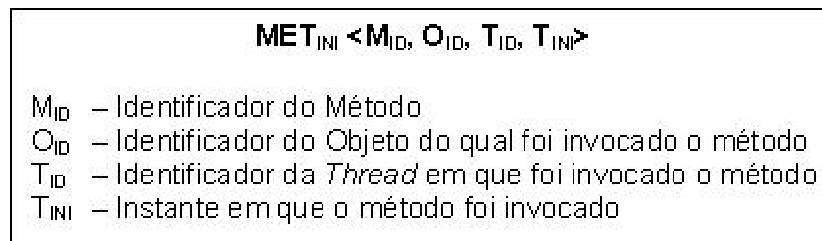


FIGURA 5.22 – Tupla de Invocação do método

O evento `Method Exit` é gerado toda a vez que um método terminou sua execução. As mesmas condições válidas para o evento `Method Entry` são verificadas antes de serem armazenadas informações sobre este evento. Quando armazenadas informações, elas seguem o formato da tupla MET_{FIM} (figura 5.23).

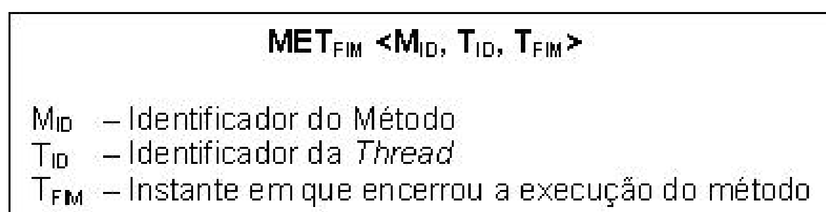


FIGURA 5.23 – Tupla de Finalização da Invocação de Método

5.3.5 Eventos de Sincronização

No modelo de programação de objetos distribuídos, as aplicações são normalmente desenvolvidas utilizando *threads* para obter desempenho. Um ponto a ser cuidadosamente observado nesse tipo de aplicações é comportamento das *threads* e o compartilhamento por recursos.

A) Estados de Execução das Threads

Em um determinado momento, uma *thread* pode estar em um entre vários estados, conforme pode ser observado na figura 5.24 [DEI 2001]. As *threads* trocam de estado em diversas situações, tais como: (i) esperando por I/O; (ii) escalonamento do ambiente de execução (*time sharing*); (iii) chamada a um método *synchronized*; (iv) chamada explícita de alguma primitiva de sincronização; e (v) chamada ao método `start`; (vi) chamada ao método `stop` ou finalização do método `run`.

Logo que uma *thread* é criada ela está no **Estado de Criada**. Mantém-se nesse estado até que o método `start()` seja invocado, quando passa para o **Estado de Pronta**. A *thread* de maior prioridade entre as que estão em **Estado de Prontas** entra no **Estado de Execução**, momento no qual o sistema atribui um processador para a *thread*. A *thread* entra no **Estado de Morta** quando seu método `run()` é completado ou quando o método `stop()` é chamado.

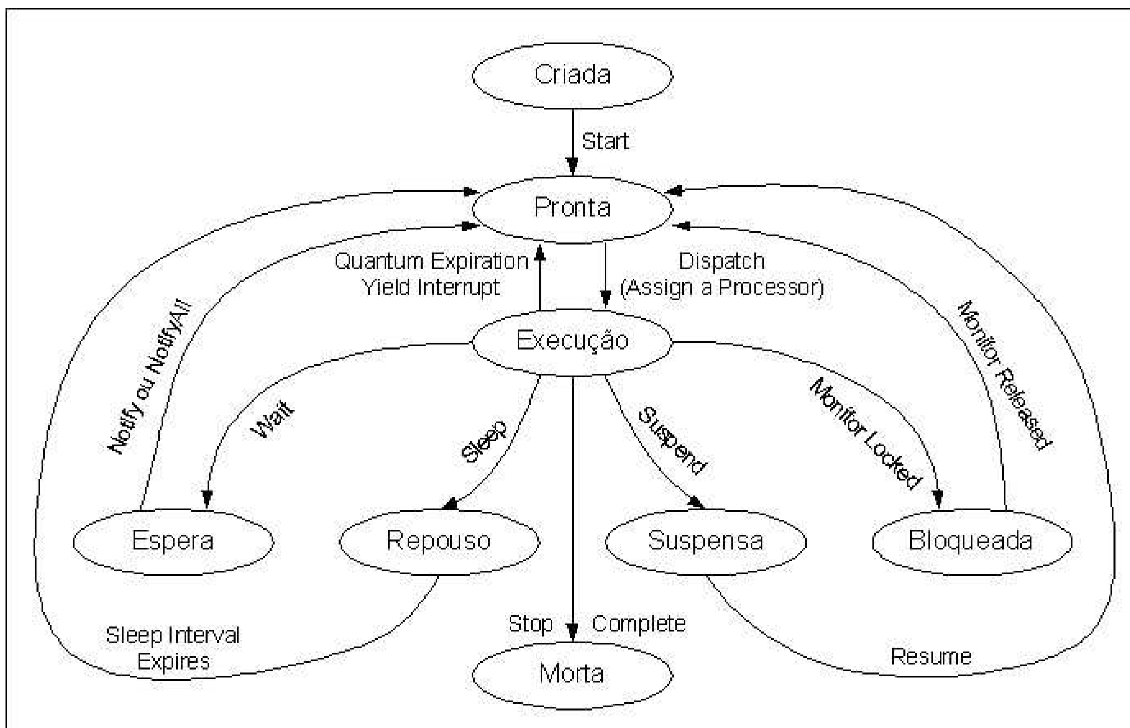


FIGURA 5.24 – Ciclo de Vida da *Thread*

Quando uma *thread* em execução chama o método `sleep()`, ela entra no **Estado de Repouso**. Uma *thread* em repouso retorna ao **Estado de Pronta** após o tempo de repouso terminar. Quando uma *thread* **em Execução** chama `wait()` a *thread* entra no **Estado em Espera**, onde ela permanece em uma fila associada com um objeto particular no qual o `wait` foi chamado. Uma das *threads* da fila de espera de um determinado objeto retorna ao **Estado de Pronta** no momento em que é realizada uma chamada `notify` por outra *thread* associada com o objeto. Todas as *threads* da fila de

espera de um objeto retornam ao **Estado de Pronta** se for chamado `NotifyAll` por outra *thread* associada ao objeto. Quando uma *thread* em execução chama o método `suspend()` ela entra no **Estado Suspensa**. A *thread* sai desse estado quando for chamado o método `resume()`.

Java utiliza monitores para executar sincronização. Todo objeto com métodos ***synchronized*** é um monitor. O monitor restringe a somente uma *thread* por vez executar um método *synchronized* em um objeto. Objetos monitores enfileiram todas as *threads* que estão esperando para entrar no objeto e executar um método synchronized. Isso é realizado bloqueando o objeto quando um método *synchronized* é chamado. Se existirem vários métodos *synchronized* somente um deles poderá estar ativo em um objeto por vez.

Uma *thread* entra na fila de espera por um objeto se a *thread* chamou um método *synchronized* de um objeto enquanto outra *thread* já está executando em um método *synchronized* deste mesmo objeto. Neste momento, a *thread* que chamou o método entra no **Estado de Bloqueada**. Quando um método *synchronized* acabar de executar, o *lock* no objeto é liberado e o monitor deixa a *thread* com maior prioridade entrar. Esta *thread* retorna então para o **Estado de Pronta** [DEI 2001].

O DOMonitor contribui com a identificação de vários desses estados. A ferramenta gera dados que permitem a obtenção dos momentos em que as *threads* foram **Criadas**, tornaram-se **Prontas** e passaram ao **Estado de Mortas**. Além disso, é possível obter também os **Bloqueios** ocorridos devido ao compartilhamento por recursos e os momentos em que estiveram em **Estado de Espera** ou em **Estado de Repouso**.

B) Sincronização

A maioria dos problemas de desempenho em programas *multithreaded* como competição por recursos, problemas de escalonamento e sincronização resultam em um excessivo tempo em espera. Por exemplo, se *threads* aguardam freqüentemente por uma *thread* **B**, então, melhorar o desempenho da *thread* **B** é como aumentar a eficiência de toda a aplicação. O agente de monitoração pode identificar os monitores que estão causando contenção devido ao acesso por várias *threads*. É útil saber, por exemplo, que duas threads, T_1 e T_2, repetidamente tentam entrar no monitor associado com uma instância da classe `C`.

Os eventos de sincronização permitem a obtenção de informações relacionadas ao momento em que uma *thread* foi bloqueada, ao momento em que saiu deste estado e ao total de tempo que a *thread* esteve no estado de bloqueada. As informações obtidas pelo DOProf são armazenadas em cinco tuplas, de acordo com o evento que foi gerado.

Toda vez que uma *thread* chamar um método de um objeto monitor e este monitor já estiver adquirido (*acquired*) por outra *thread*, o DOProf recebe o evento `Monitor Contended Enter`. Este evento indica que duas *threads* estão competindo por um mesmo recurso, um objeto compartilhado, e uma *thread* ficará bloqueada até que outra libere o monitor. A ocorrência em demasia deste evento indica ao programador que sua aplicação pode estar apresentando problemas de desempenho devido ao compartilhamento de recursos. As informações relacionadas a este evento são registradas, seguindo o formato da tupla MON_{ER} (figura 5.25).

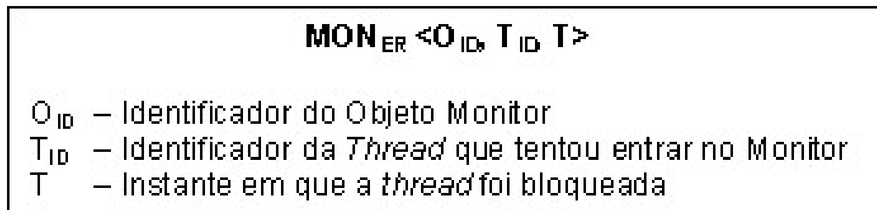


FIGURA 5.25 – Tupla MON_{ER}

Quando uma *thread* libera um objeto monitor e existem outras *threads* esperando para entrar no monitor, o ambiente de execução Java libera uma destas *threads* que estão na fila de espera por este objeto para que ela entre em execução. O evento gerado por uma *thread* que entra em um objeto monitor após esperar que este monitor fosse liberado (*released*) por outra *thread* é o `Monitor Contended Entered`. As informações armazenadas para este evento seguem o formato da tupla MON_{ED} (figura 5.26).

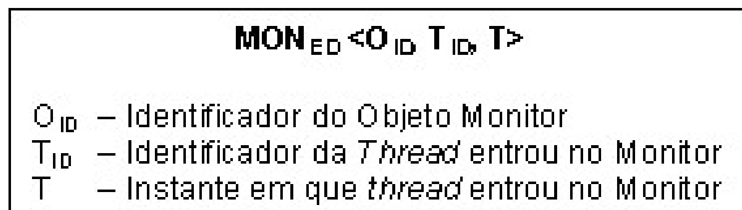


FIGURA 5.26 – Tupla MON_{ED}

O evento `Monitor Contended Exit` é gerado quando uma *thread* sai de um monitor e existe uma outra *thread* esperando para adquirir (*acquire*) o mesmo monitor. As informações relacionadas a este evento são armazenadas no formato da tupla MON_{EX} (figura 5.27).

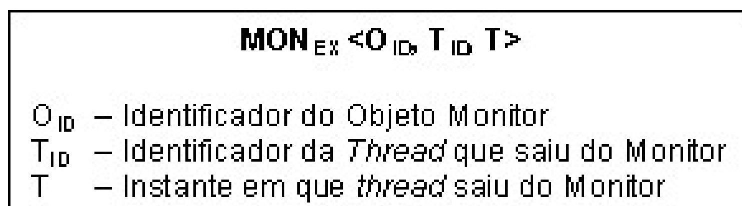


FIGURA 5.27 – Tupla MON_{EX}

Uma *thread* também entra na fila de espera por um objeto se chamar `wait()` enquanto estiver dentro do objeto. No entanto, é importante distinguir *threads* que foram bloqueadas devido ao monitor estar ocupado daquelas que explicitamente chamaram `wait()`. Após a finalização de um método *synchronized*, *threads* que estavam bloqueadas devido ao monitor estar ocupado podem tentar entrar no objeto. *Threads* que explicitamente invocaram `wait()` são liberadas para continuar somente quando forem notificadas via `notify()` ou `notifyAll()`.

Quando uma *thread* chama `wait()` em um objeto o DOProf recebe a notificação de um evento `Monitor Wait`. As informações relacionadas a este evento são armazenadas em uma tupla do tipo MON_{WAT} (figura 5.28). Este evento também é gerado quando uma *thread* invocar o método `sleep()`. A distinção entre as duas

situações `wait()` e `sleep()` é realizada a partir do campo Time. Quando for uma chamada `sleep()` o campo Time recebe `NULL`.

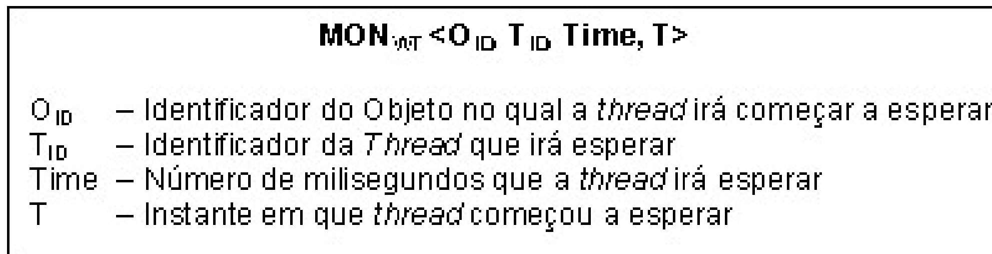


FIGURA 5.28 – Tupla MON_{WAT}

Quando uma *thread* para de esperar em um objeto, o DOProf recebe a notificação de um evento `Monitor Waited`. No caso do campo Time ser `NULL`, indica que a *thread* estava no modo de `sleep()`. As informações geradas seguem o formato da tupla MON_{WD} (figura 5.29).

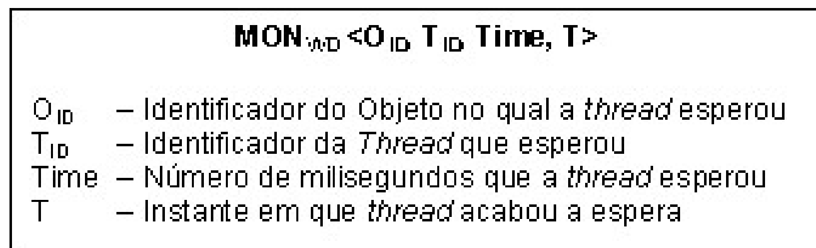


FIGURA 5.29 – Tupla MON_{WD}

5.3.6 Formato do Arquivo de Traço Local

O arquivo de traço local é composto por um conjunto de tuplas usadas para representar os eventos que ocorreram durante a execução. É armazenado um caractere no início de cada linha para identificar a que tipo de tupla aquele registro se refere, conforme tabela 5.4.

TABELA 5.4 – Relação das Tuplas locais

Nome	Descrição	Char
THR_{INI}	Registro gerado a cada nova *thread* da aplicação que é criada	T
THR_{FIM}	Registro gerado que identifica o final da execução de uma *thread*	H
CLA_{INI}	Registro gerando quando uma classe de interesse é carregada	C
CLA_{FIM}	Registro gerado quando uma classe de interesse é descarregada	L
OBJ_{INI}	Registro gerado quando um objeto é instanciado	O
OBJ_{FIM}	Registro gerado quando um objeto é destruído	J
OBJ_{MOV}	Registro gerado quando um objeto troca de identificador	N
MET_{INF}	Registro gerado com informações relacionadas ao método	M
MET_{INI}	Registro gerado quando um método é invocado	E
MET_{FIM}	Registro gerado quando é finalizada a invocação de um método	F
MON_{ER}	Registro gerado quando uma *thread* tenta entrar em um monitor já adquirido por outra *thread*	R
MON_{ED}	Registro gerado quando uma *thread* entra em um monitor depois de esperar que o monitor fosse liberado	B
MON_{EX}	Registro gerado quando uma *thread* sai de um monitor e existem *threads* esperando para entrar neste monitor	X
MON_{WT}	Registro gerado quando uma *thread* aguarda em um objeto	W
MON_{WD}	Registro gerado quando uma *thread* acaba de aguardar em um objeto	D

O arquivo de traço contendo as informações sobre os eventos locais é um conjunto ordenado de tuplas. O layout do arquivo de traço detalhado pode ser observado na figura 5.30.

DOMonitor – Distributed Object Monitor
Arquivo de Traço Detalhado
Nome do Host – Identificador da JVM

'T', T_{ID}, O_{ID}, Nome, Nome do Grupo, Nome da Superior, T_{INI}
'H', T_{ID}, T_{CPU}, T_{FIM}
'C', C_{ID}, T_{ID}, Nome da Classe, Nome do Fonte, T_{INI}
'L', C_{ID}, T_{ID}, T_{FIM}
'O', O_{ID}, C_{ID}, T_{ID}, T_{INI}
'J', O_{ID}, T_{ID}, T_{FIM}
'N', M_{ID}, C_{ID}, Nome do Método, Linha, T_{INI}, T_{FIM}
'M', M_{ID}, O_{ID}, T_{ID}, T_{INI}
'E', M_{ID}, T_{ID}, T_{FIM}
'F', O_{ID}, OLD_{ID}, T_{ID}, T_{FIM}
'R', O_{ID}, T_{ID}, T
'B', O_{ID}, T_{ID}, T
'X', O_{ID}, T_{ID}, T
'W', O_{ID}, T_{ID}, Time, T
'D', O_{ID}, T_{ID}, Time, T

FIGURA 5.30 – Layout do Arquivo de Traço Detalhado

5.4 Coleta das Informações Remotas

Nas aplicações distribuídas os objetos encontram-se espalhados pelo ambiente de execução, ou seja, executam em várias máquinas. A interface RMI permite que objetos Java instanciados em diferentes máquinas possam se comunicar. Mais formalmente, um objeto remoto é um objeto em que métodos podem ser chamados por uma JVM diferente daquela onde o objeto está instanciado, geralmente por uma JVM de um computador diferente. Cada objeto remoto implementa uma ou mais interfaces remotas que especificam quais dos seus métodos podem ser invocados pelo sistema estrangeiro (cliente). Os detalhes sobre a realização da conexão entre os nodos e a transferência dos dados ficam ocultos nas classes RMI.

Um dos diferenciais do DOMonitor é a capacidade de monitorar a execução de atividades que envolvam várias JVM's. No exemplo da figura 5.31, a $\texttt{JVM}_1$ e a $\texttt{JVM}_2$ são duas máquinas virtuais. Neste caso em especial, a $\texttt{JVM}_2$ é o servidor e a $\texttt{JVM}_1$ é o cliente. A instância de `BankSystemServer` é um objeto remoto que está disponível para a $\texttt{JVM}_1$. O mecanismo de RMI permite que o cliente, uma instância de `BankUser` obtenha uma referência à instância remota de `BankSystemServer`.

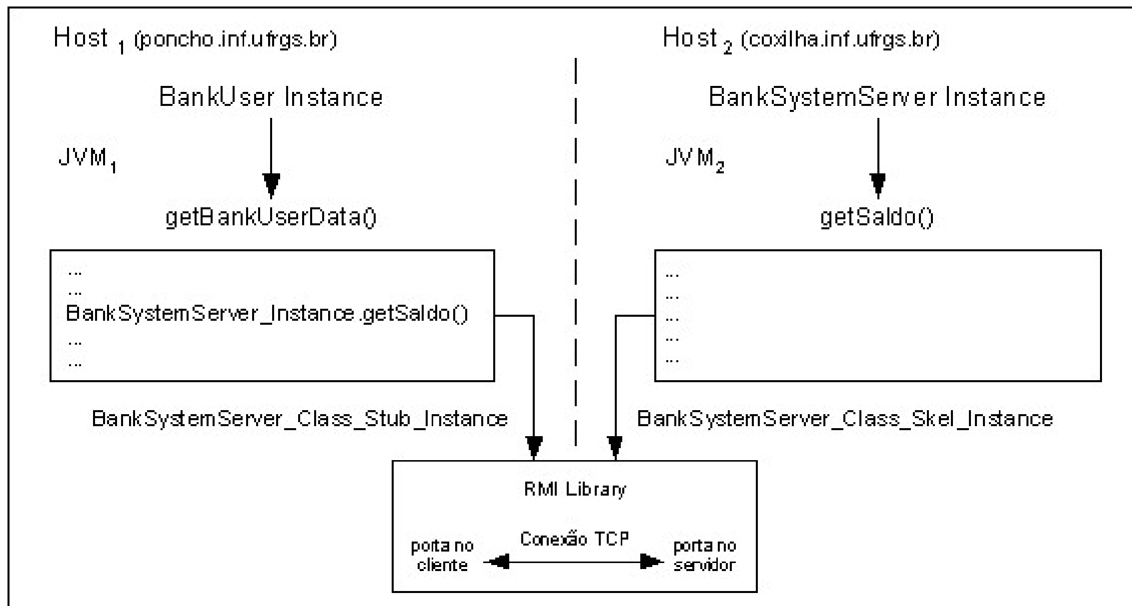


FIGURA 5.31 – Exemplo de Chamada RMI

A $\texttt{JVM}_1$ executa o método `getBankUserData()` dentro do objeto `BankUser`. O método `getBankUserData()` invoca `BankSystemServer_Instance.getSaldo()` onde o objeto `BankSystemServer_Instance` contém a referência remota do objeto `BankSystemServer`. Internamente, esta chamada é direcionada a `BankSystemServer_Class_Stub.getSaldo`, onde a instância de `BankSystemServer_Class_Stub` é a visão do objeto remoto do lado do cliente. O objeto *stub* estabelece uma conexão TCP/IP (*Transmission Control Protocol/Internet Protocol*) com o a visão do lado do servidor, uma instância do `BankSystemServer_Class_Skel`. O objeto *stub* no lado do cliente comunica-se com o objeto *skel* (*skeleton*) no lado do servidor através do protocolo de serialização de

objetos para invocação remota. O objeto *skel* invoca o método remoto `getSaldo()` na instância remota do `BankSystemServer` e retorna o resultado para o *stub*.

5.4.1 Invocação Remota de Método

O endereço de uma máquina poderia ser suficiente para identificá-la. No entanto, aplicações modernas podem realizar diferentes tarefas ao mesmo tempo. Por exemplo, requisições de *email* precisam ser separadas das requisições de *ftp*, as quais precisam ser separadas do tráfico *web*. Isso é resolvido com a utilização de portas. Cada computador com um endereço IP possui um grande número de portas lógicas (65535 para ser mais preciso). Com isso, quando dados são enviados para uma máquina, deve ser informado o endereço IP e o número da porta. Desta forma, os dados tem seu destino identificado unicamente [HAR 97].

A mesma situação ocorre quando se trata dos objetos distribuídos. Cada objeto que comunica-se com outro é unicamente identificado pelo IP da máquina na qual está executando e pelo número da porta lógica associada a este objeto. Ou seja, todas as mensagens enviadas para o par <Endereço IP, Porta Lógica> são direcionados para o objeto. Isto é garantido, uma vez que a classe da qual um objeto remoto é descendente possui um construtor que cria o objeto e determina para ele um número de porta em tempo de execução. Portanto, quando esse objeto é registrado, é armazenado seu *host* e número de porta. Quando o cliente necessita uma referência ao objeto remoto, realiza uma consulta ao registro, que retorna a referência completa, inclusive com o número da porta.

Quando um método é remotamente invocado, é gerada a Tupla de Requisição do cliente, chamada COM_{CLI} (figura 5.32).

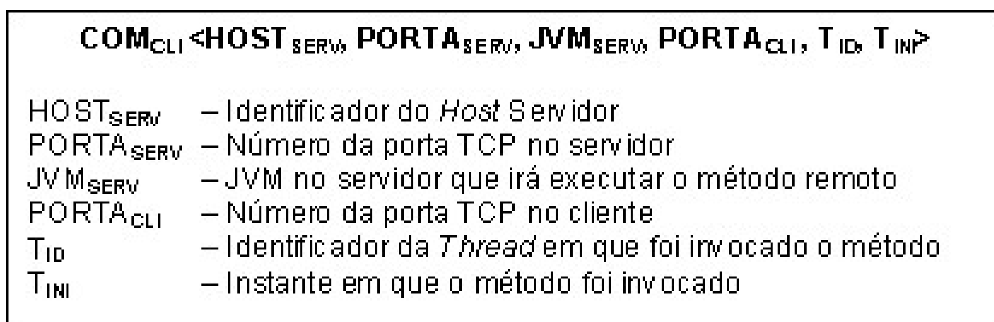


FIGURA 5.32 – Tupla de Invocação Remota de Método

Quando a resposta a uma invocação remota de método for enviada, é gerada a tupla de Resposta a Invocação Remota de Método, COM_{SER} (figura 5.33).

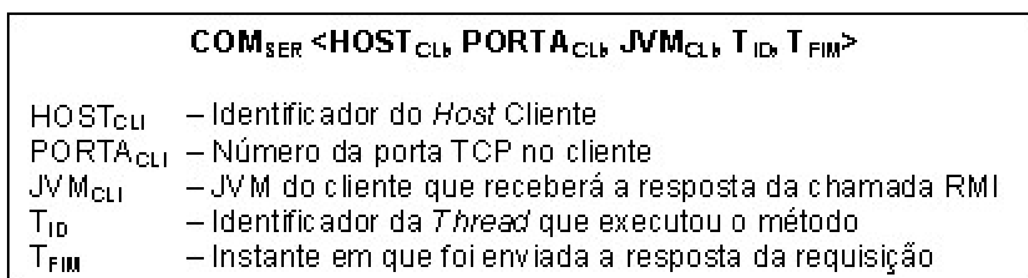


FIGURA 5.33 – Tupla de Resposta a uma chamada RMI

Segundo Harold [HAR 97], para enviar dados de uma máquina para outra, basta identificar o *Host* destino e o número da porta TCP/IP. No entanto, como pode ser observado nas tuplas de comunicação (figura 5.32 e 5.33) o DOProf armazena também a JVM destino e a *thread* que gerou a comunicação. A informação da JVM é necessária para identificar em qual arquivo de traço de comunicação estará o registro correspondente à resposta da invocação remota. A informação da *thread* é armazenada para que seja possível associar a troca de dados a uma determinada *thread* da aplicação.

Cabe ressaltar ainda, a razão pela qual o número da porta TCP/IP do cliente é armazenada no registro COM_{CLI}. Esta informação é de importância fundamental para a ligação entre a invocação remota de método e sua resposta. O número da porta TCP/IP do cliente é necessária para relacionar corretamente a invocação remota à sua resposta, principalmente na situação em que duas *threads* em execução na mesma JVM invocam o mesmo método remoto no servidor quase ao mesmo tempo.

No exemplo da figura 5.34, duas *threads* em execução na JVM_C invocam o mesmo método remoto no mesmo objeto remoto da JVM_S. Pode acontecer, da chamada realizada pelo *thread* TC_2 ser recebida no servidor antes invocação feita pela *thread* TC_1 (a invocação realizada por TC_1 foi realizada antes da invocação realizada por TC_2). Ou seja, a segunda chamada pode chegar ao servidor antes da primeira. Com isto, não é possível fazer a ligação das invocações remotas apenas com o par <Host, JVM>. O número da porta de conexão TCP/IP do cliente é usado para resolver essa ambigüidade.

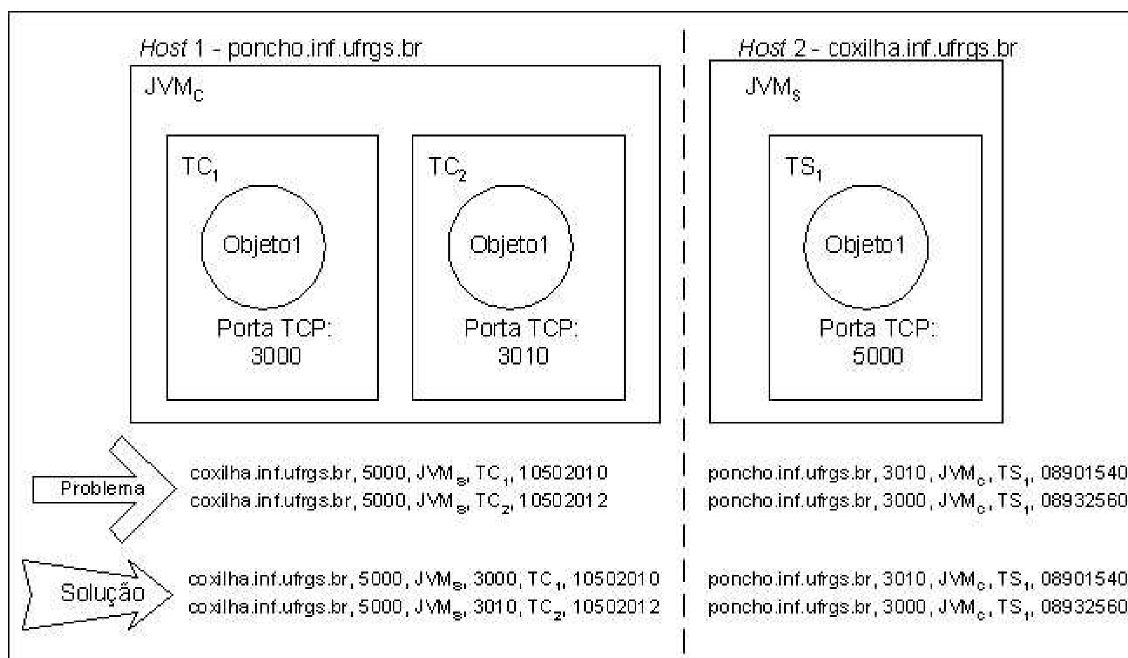


FIGURA 5.34 – Exemplo de Traço de Chamada RMI

As informações de relógio armazenadas pelo cliente e pelo servidor não são as mesmas, uma vez que em sistemas distribuídos não há a garantida de estarem sincronizados. O DOMonitor não exige que os relógios estejam sincronizados nas várias máquinas que podem compor o ambiente distribuído heterogêneo. Essa decisão de projeto responde a uma situação importante em aplicações distribuídas, particularmente aquelas com servidores compartilhados, pois pode não ser possível realizar a sincronização ou podem ser gerenciadas separadamente e portanto, difícil de sincronizar.

A ausência de relógios sincronizados requer um processamento adicional, realizado com conseqüências mínimas. Uma conseqüência dos relógios sem sincronismo é a aparente ausência de continuidade de tempo quando está se traçando a execução entre chamadas RMI. O relógio do servidor pode preceder o relógio do cliente, ou ser maior que o tempo da resposta à invocação armazenado no cliente. Essa discrepância pode ser resolvida organizando os tempos para criar um “tempo global plausivo”. Isso é resolvido pelo processo de adequação de relógios (capítulo 6).

5.4.2 Formato do Arquivo de Traço de Comunicação

TABELA 5.5 – Tuplas do Arquivo Remoto

Nome	Descrição	Char
COM_{CLI}	Registro gerado pelo cliente em uma invocação remota de método	I
COM_{SER}	Registro gerado pelo servidor em resposta a uma invocação remota	S

O *layout* do arquivo de comunicação gerado a partir das invocações remotas de métodos pode ser observado na figura 5.35.

DOMonitor – Distributed Object Monitor
Arquivo de Traço de Comunicação
Nome do Host – Identificador da JVM

‘I’, $HOST_{SERV}$, $PORTA_{SERV}$, JVM_{SERV}, $PORTA_{CLI}$, T_{ID}, T_{INI}
‘S’, $HOST_{CLI}$, $PORTA_{CLI}$, JVM_{CLI}, T_{ID}, T_{FIM}

FIGURA 5.35 – Layout do Arquivo de Traço de Comunicação

6 O Processamento das Informações no DOMonitor

O módulo de Processamento das Informações tem comportamento diferenciado de acordo com a finalidade dos dados de monitoração. Por exemplo, se os dados de monitoração forem usados para visualizar o comportamento da execução, é necessário que as informações sejam organizadas de tal forma que reflitam o **comportamento global da execução**. Já se os dados de monitoração forem usados para fornecer informações ao escalonador, é suficiente o **comportamento local**, ou seja, a exigência de informações é por máquina.

6.1 Comportamento Global da Execução

O comportamento global da execução da aplicação é obtido através da composição do comportamento local de cada uma das JVM's que participaram da execução. Esta composição é necessária de acordo com a finalidade para a qual os dados de monitoração serão utilizados. Uma das finalidades que exigem esta composição é a visualização. Este processo é realizado após o término da execução da aplicação (*Post-Mortem*) para não interferir no ambiente de execução.

Para a obtenção do comportamento global, o processamento das informações pode ser dividido em três etapas: adequação dos relógios, ligação entre as invocações remotas e publicação dos resultados. Esses dados publicados podem então passar por um processo de filtro para serem utilizados por uma determinada ferramenta. Com isso, a coleta e o processamento das informações ficam independentes da utilização dos resultados.

O processamento das informações será realizado na máquina em que o arquivo principal está armazenado. Para isso, os arquivos que estiverem fisicamente em outras máquinas, devem ser enviados à máquina que irá realizar o processamento.

6.1.1 Adequação dos Relógios

O modelo físico de um ambiente heterogêneo distribuído consiste de vários computadores que estão geograficamente separados, mas que estão conectados por uma rede de comunicação. Todos os nodos são autônomos e se comunicam com os outros através da rede de comunicação. Nestes ambientes, não existe um relógio global comum a todos os nodos, cada nodo possui seu próprio relógio.

Quando um evento `e` ocorre antes de um evento `f`, pode-se dizer que o tempo registrado na ocorrência do evento `e` é anterior ao tempo da ocorrência do evento `f`. Isso é verdadeiro quando os tempos dos eventos `e` e `f` forem medidos pelo mesmo relógio. No entanto, se os tempos dos diferentes eventos forem medidos por relógios distintos, e já que relógios distintos podem ter tempos diferentes, não se pode garantir que um evento ocorreu antes de outro apenas observando os tempos dos eventos. Existem algumas técnicas para ordenar eventos em ambientes distribuídos [JAL 94]: **Ordenação Parcial dos Eventos**, **Relógios Lógicos** e **Ordenação Total dos Eventos.**

Ordenação Parcial dos Eventos – como cada nodo da rede possui seu próprio relógio, eventos de um determinado nodo podem ser ordenados usando o relógio do nodo. O problema ocorre quando se tenta definir uma ordem relativa entre eventos de diferentes nodos. No esquema proposto por Lamport [LAM 78], um evento

a, executado por um processo é considerado como tendo “acontecido antes” (→) de outro evento b, executado pelo mesmo processo, se o evento a ocorre antes do evento b na seqüência dos eventos executados pelo processo, Neste caso, $a \rightarrow b$.

Em aplicações distribuídas, essa relação é construída a partir da premissa de que o recebimento de uma mensagem não pode ocorrer antes do seu envio. Isto é, pode-se garantir que o envio de uma mensagem “ocorre antes” do recebimento da mesma por um outro processo. A definição formal descrita em [LAM 78] descreve que uma relação → em um conjunto de eventos de um sistema distribuído é a menor relação que satisfaça uma entre as três condições: (1) se a e b são eventos executados no mesmo processo, e a é executado antes de b, então $a \rightarrow b$; (2) se a é o emissor de uma mensagem e b é o receptor dessa mensagem, então $a \rightarrow b$; (3) se $a \rightarrow b$ e $b \rightarrow c$, então $a \rightarrow c$. Dois eventos distintos são dito concorrentes se nem $a \rightarrow b$ e nem $b \rightarrow a$.

Relógios Lógicos – nesta técnica, cada *host* tem um relógio lógico, de tal forma que o tempo dos eventos a partir destes relógios sejam consistentes com a relação →. Um relógio lógico é uma forma de atribuir um número a um evento. O valor do relógio não tem relação com o relógio físico. O relógio lógico C_i, para um processo P_i é uma função que atribui um valor $C_i(a)$ para um evento a de um processo P_i. Como é condição que a relação → seja mantida, então, para quaisquer eventos a, b, se $a \rightarrow b$, então $C(a) < C(b)$.

A implementação desta técnica é baseada no envio de *timestamps* com as mensagens. Quando uma mensagem é enviada por um processo P_i, o *timestamp* do evento de emissão é incluído na mensagem m (T_m) e é verificado pelo receptor. Existem duas condições que um sistema de relógio lógico precisa satisfazer: (1) cada processo P_i incrementa um contador local (C_i) entre dois eventos sucessivos; (2) após receber uma mensagem m, um processo P_j atribui a C_j um valor maior ou igual ao seu valor atual e maior do que T_m.

Ordenação Total – relógios lógicos podem ser usados para realizar uma ordenação total entre eventos de um sistema distribuído. Um método simples é ordenar os eventos pelos *timestamps*. Uma vez que um sistema de relógio lógico é composto por diversos relógios lógicos, podem haver eventos com *timestamps* iguais. Uma solução é utilizar uma ordem de precedência de acordo com o processo. Com isso, se dois eventos possuírem o mesmo *timestamp*, o evento do processo com ordem menor na relação de precedência é considerado como tendo “ocorrido antes”. Portanto, esta ordenação total depende do sistema de relógio lógico e na relação de ordenação (precedência) dos processos.

A ordenação de eventos é de importância essencial na monitoração e principalmente na depuração dos programas. Sem um conhecimento mínimo das relações de causalidade entre os eventos, a análise dos resultados fica extremamente prejudicada. A solução para o problema de obter uma ordenação causal dos eventos na ausência de um relógio global sincronizado pode ser alcançada através de vários algoritmos ([LAM 78], [FID 91], [FOW 90]). Algumas ferramentas implementam soluções alternativas, como é o caso do MODIMOS.

O MODIMOS (*Managed Object-based Distributed Monitoring Systems*) é um sistema experimental para visualização do comportamento dinâmico de aplicações distribuídas compostas por objetos heterogêneos [ZIE 95]. O sistema propõe-se a monitorar e visualizar aplicações construídas por vários componentes cooperantes, escritos em diferentes ambientes de computação. Isso é alcançado devido à arquitetura

multi-camada da ferramenta, onde as camadas inferiores relacionam-se com um ambiente específico e comunicam-se com as superiores utilizando uma interface padrão.

O MODIMOS implementou uma solução alternativa para o problema de ausência de um relógio global sincronizado. A estrutura hierárquica das aplicações monitoradas e a semântica dos eventos introduzem uma relação de ordem parcial entre conjuntos de eventos monitorados, p. ex., o evento de *destruição* de um objeto **O** não pode ser considerado sem que exista o evento de *criação*. Com isso, foram criadas regras de acordo com a semântica dos eventos.

A ordenação parcial dos eventos possibilita a verificação da relação de ordem e de causa-conseqüência. Mas a partir deste tipo de ordenação perde-se a medida precisa sobre a duração dos eventos. Por exemplo, este tipo de ordenação não permite que seja verificado o tempo que uma mensagem demorou no percurso entre a máquina origem e o seu destino.

Sem uma referência global de tempo não é possível identificar se um evento `e` aconteceu antes ou depois de um evento `f` que ocorreu em outro processo. Além disso, não é possível medir o intervalo de tempo real entre a ocorrência dos evento `e` e `f`. Se for suficiente a relação de ordem, pode-se usar uma das técnicas apresentadas anteriormente. No entanto, quando se está avaliando o desempenho da implementação de uma aplicação distribuída é necessário medir o intervalo de tempo real entre os eventos, e para tal, a utilização de relógios lógicos não é apropriada.

A medição precisa de tempo é crucial para o sucesso da análise de desempenho baseada em eventos [MAI 95]. Para que estes dados possam ser visualizados de maneira satisfatória é necessário que se obtenha a duração dos eventos. Por exemplo, é desejável que possa ser realizada uma estimativa do intervalo de tempo entre o envio e o recebimento de uma mensagem. As técnicas citadas anteriormente não conseguem estimar a duração dos eventos estabelecendo uma visão clara da concorrência entre as diversas *threads* que compõe a aplicação. O DOMonitor utiliza um método estático, proposto por Maillet [MAI 95], para estimar uma referência de tempo global precisa.

Métodos estáticos têm sido propostos na literatura com o objetivo de estimar uma referência de tempo global. Estes métodos são apropriados para análise de desempenho uma vez que eles não perturbam a aplicação analisada. No entanto, estes algoritmos devem coletar algumas amostras de tempo a partir das quais o tempo global será calculado [MAI 95].

A solução apresentada por Maillet [MAI 95] consiste em coletar amostras de tempo antes da aplicação começar a executar e outras amostras após o término da execução da aplicação. É realizada uma composição entre as amostras obtidas durante estas duas fazes de sincronização de forma que os coeficientes de correção podem ser obtidos. Devido ao coeficiente de correção ser conhecido somente após o término da execução da aplicação, não é possível realizar uma correção *on-line*. Isto não chega a ser uma desvantagem quando o tempo global é usado para análises *Post-Mortem*.

Com isto, o tempo de um evento pode ser obtido usando o relógio físico do processador e, posteriormente, uma correção será aplicada neste tempo para expressá-lo em uma base de tempo comum. O relógio local de um dos processadores é escolhido como referência. A precisão do tempo global obtido é independente das variações dos retardos de comunicação. Além disso, a estimativa não exige a troca de mensagens adicionais durante a execução da aplicação monitorada.

6.1.2 Ligação entre as Invocações Remotas de Métodos

Nesta etapa, as invocações de métodos que foram executados por outras JVM's são processadas para determinar a comunicação existente entre as JVM's que compuseram o ambiente de execução da aplicação distribuída. Para entender melhor como é realizada essa associação, consideremos um cenário simples onde existe uma única JVM cliente interagindo com uma JVM servidora. Em uma aplicação composta por uma única *thread*, várias chamadas realizadas pelo cliente para o mesmo método de um objeto remoto no servidor podem ser facilmente vinculadas a partir da porta de comunicação utilizada pelo cliente e pelo servidor. Os registros com a mesma porta TCP de ambos os lados podem ser associados seguindo a ordem cronológica. No entanto, como Java é um ambiente *multithreaded* não basta saber para qual JVM foi enviada uma mensagem. É necessário obter exatamente qual *thread* gerou a comunicação e por qual motivo esta comunicação foi gerada.

O processo de ligação entre as invocações remotas precisa identificar exatamente a origem e o destino da troca de dados. Para tanto, é necessário identificar o *host*, a JVM e a *thread* que originaram cada uma das comunicações. Além disso, é importante a identificação de quais trocas de dados são requisições (invocações remotas) ou respostas (resposta a uma chamada RMI).

As informações disponíveis para a construção destas ligações estão armazenadas nos arquivos de traço de comunicação de cada JVM. Cabe ressaltar que as informações contidas nestes arquivos seguem o formato da tupla de invocação remota de método COM_{CLI} (figura 5.32) ou de resposta a uma invocação COM_{SER} (figura 5.33). É possível perceber que não há no registro COM_{CLI} a identificação total da origem da mensagem (*host*, JVM, *thread*) e do destino da mensagem (*host*, JVM, *thread*). O *host* e a JVM de origem são obtidos a partir do arquivo de traço de comunicação que contém os registros. A *thread* de origem e o par *host* e JVM destino constam da tupla COM_{CLI}. Portanto, a informação da *thread* para a qual a mensagem foi enviada precisa ser obtida. Esta informação pode ser obtida no registro COM_{SER} que corresponde à invocação em questão.

De forma análoga, a comunicação gerada para responder à invocação remota também precisa ter seu destino bem definido, de tal forma que seja possível identificar que a resposta é para uma *thread* específica. As informações resultantes são armazenadas em uma tupla COM_{REG} com as informações conforme pode ser observado na figura 6.1.

$COM_{REG} <T_{ID}, HOST_{DEST}, JVM_{DEST}, T_{DEST}, Tipo, T>$

T_{ID}	– Identificador da *thread* que gerou a comunicação
$HOST_{DEST}$	– *Host* para o qual a mensagem foi enviada
JVM_{DEST}	– JVM para a qual a mensagem foi enviada
T_{DEST}	– Identificador da *thread* para a qual a mensagem foi enviada
Tipo	– Tipo da mensagem: 'C' indica uma invocação remota. 'S' indica a resposta de uma invocação remota.
T	– Instante em que o método foi invocado

FIGURA 6.1 – Informações relacionadas a comunicação

O resultado do processamento das invocações remotas é armazenado em um arquivo **.dpc**, que será posteriormente utilizado para a composição de todas as

informações em um único arquivo de traço. Os passos para realizar esta ligação são detalhados a seguir:

1. O processo começa a ser executado para uma determinada máquina virtual. Seu arquivo de traço de comunicação é percorrido à procura de tuplas COM_{CLI}, identificadas pelo caractere ‘I’;
2. Quando uma tupla do tipo COM_{CLI} é encontrada, é necessário obter o registro COM_{SER} correspondente. Esta procura é realizada da seguinte forma:
   - 2.1. É identificado o arquivo de traço de comunicação da JVM correspondente a partir do nome do *host* e da JVM do servidor de acordo com os campos $HOST_{SERV}$ e JVM_{SERV} da tupla COM_{CLI};
   - 2.2. O arquivo selecionado é percorrido à procura de uma tupla COM_{SER}, identificada pelo caractere ‘S’. As seguintes condições devem ser satisfeitas para obter a correspondência:
     - 2.2.1. A informação de tempo da tupla COM_{SER} deve ser maior ou igual que a informação de tempo do registro COM_{CLI};
     - 2.2.2. O campo $HOST_{CLI}$ da tupla COM_{SER} deve ser igual ao *host* do cliente do qual se está processando o arquivo de comunicação;
     - 2.2.3. O campo JVM_{CLI} da tupla COM_{SER} deve ser igual a JVM cliente da qual se está processando o arquivo de comunicação;
     - 2.2.4. O campo $PORTA_{CLI}$ da tupla COM_{SER} deve ser igual ao campo $PORTA_{CLI}$ do registro COM_{CLI};
     - 2.2.5. Este registro não pode ter sido utilizado ainda. Ou seja, não são válidos os registros COM_{SER} que tenham o caractere ‘U’ no final da linha (inicialmente nenhum registro possui o *flag* ‘U’);
   - 2.3. Quando é encontrado o registro COM_{SER} correspondente, é gerado no arquivo **.dpc** desta JVM o registro correspondendo à resposta da invocação remota de método, já contendo o identificador da *thread* para qual a resposta foi enviada. Este identificador é obtido através do campo T_{ID} do registro COM_{CLI};
   - 2.4. O registro COM_{SER} é marcado como utilizado através do caractere ‘U’ (isto garante que cada registro seja utilizado apenas uma vez);
3. É gerado no arquivo **.dpc** o registro indicando uma invocação de método.

Estes passos são repetidos para todas as JVM’s. Cabe ressaltar que no momento em que é gerado o registro correspondendo a uma invocação remota de método no arquivo **.dpc** da JVM que atuou como cliente na invocação, também é gerado o registro de resposta esta invocação no arquivo **.dpc** da JVM que atuou como servidor. Com isso, no final do processamento dos arquivos de comunicação de todas as JVM’s, tem-se nos arquivos **.dpc** registros ordenados cronologicamente relacionados a todas as invocações remotas e a todas as respostas de requisições.

6.1.3 Publicação dos Resultados

Nesta etapa, é realizada uma composição entre as informações dos arquivos de traço detalhado (**.dom**) e dos arquivos de comunicação processados (**.dpc**) de todas as JVM's. A partir desta composição, tem-se em um único arquivo de traço informações sobre todos os eventos que ocorreram durante a execução da aplicação, possibilitando desta forma a obtenção do comportamento global.

O arquivo de traço resultante deste processo recebe o nome de **Arquivo de Traço Global** (**.dtg**) e contém os eventos ordenados em ordem cronológica de ocorrência. Cada evento precisa ser unicamente identificado com relação ao local em que ele ocorreu, ou seja, uma vez que cada evento foi executado por uma *thread*, em execução em uma máquina virtual que por sua vez está em execução em um *host* físico, cada evento que ocorreu durante a execução de uma aplicação Java tem que ter seu custo atribuído a estes componentes do ambiente de execução. Uma vez que neste arquivo de traço existirão eventos de todos as JVM's e *hosts* do ambiente de execução distribuído, cada um dos eventos deve ser unicamente identificado por um identificador de *host* e JVM.

Com o objetivo de reduzir o arquivo resultante, cada par <*host*, JVM> será representado por um ID. Este ID será definido no cabeçalho do arquivo de traço global da seguinte forma: **ID <*Host*, JVM>**, sendo ID um número inteiro qualquer e as informações de *host* e JVM são obtidas a partir do cabeçalho do arquivo de traço detalhado que se está obtendo informações. Por exemplo, quando o arquivo de traço de uma JVM_1 que executou fisicamente no *host* poncho.inf.ufrgs.br começar a ser percorrido, será criada a definição de um ID tal como: 5 <poncho.inf.ufrgs.br, JVM_1>.

As informações contidas no arquivo de traço global seguem o mesmo formato das tuplas apresentadas anteriormente apenas com a inclusão do identificador para representar o *host* e a JVM. Com isso, cada uma das linhas do arquivo de traço global terá o caractere inicial que identifica o tipo de evento, o ID para representar o par <*host*, JVM> onde ocorreu o evento e restante das informações conforme definido anteriormente. O formato do arquivo resultante deste processo é mostrado na figura 6.2.

Após este processo, o Arquivo de Traço Global contém eventos representando o comportamento global da aplicação. Desta forma, tais informações podem ser utilizadas como entrada para outras ferramentas, dentre as quais pode-se citar as ferramentas de depuração e visualização. Portanto, uma vez que se deseje utilizar as informações obtidas pelo DOMonitor basta implementar um módulo que transforme o formato de dados gerado pela monitoração para o formato de dados de entrada da ferramenta em questão. Com isso a coleta e o processamento ficam independentes da utilização dos dados de monitoração.

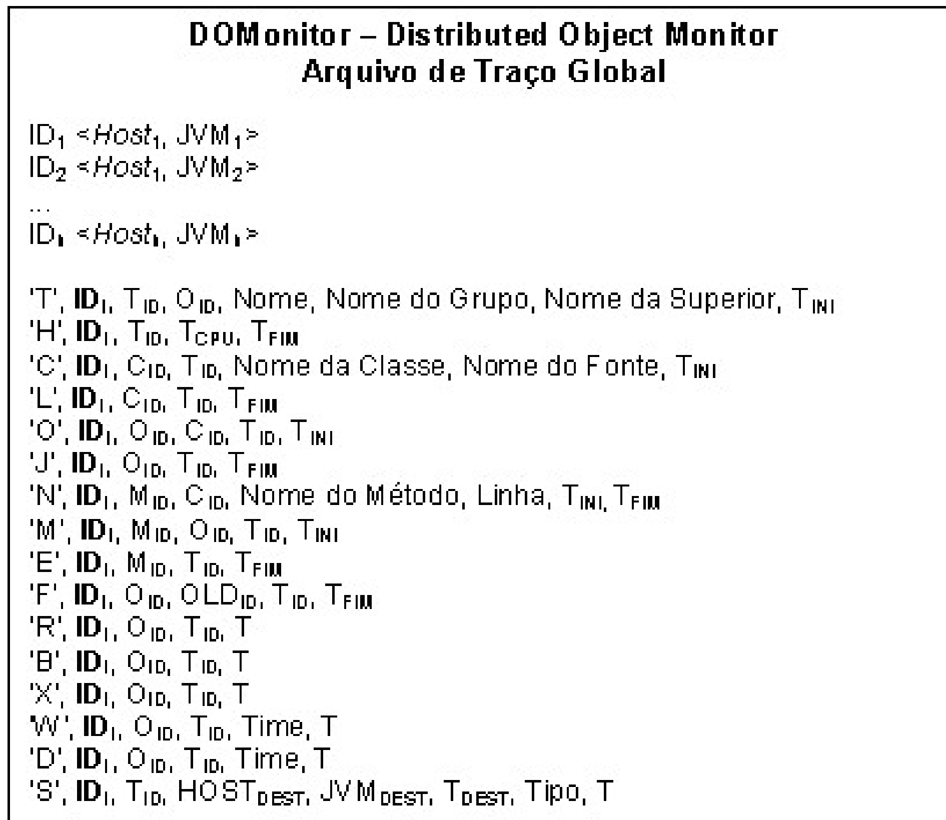


FIGURA 6.2 – Formato do Arquivo de Traço Global

Além da utilização das informações contidas no arquivo de traço por outras ferramentas, é possível construir um processo de análise para obter determinadas informações sobre estes dados produzidos. Dentre as informações que podem ser obtidas, pode-se citar:

- O número de *threads* criadas pela aplicação, o instante em que cada uma delas começou e terminou de executar, o tempo que a *thread* existiu no sistema, a quantidade de tempo que ela esteve no estado de pronta ou o tempo que ela ficou bloqueada;
- É possível obter a relação método chamador-chamado, pois exceto em um caso especial envolvendo a criação de uma nova *thread*, um método que chama métodos filhos completa sua execução somente depois que todos os métodos filhos chamados tenham completado. Ou seja, uma vez que no arquivo de traço global os métodos aparecem ordenados cronologicamente, um método que chama outros métodos engloba o tempo de execução de todos os filhos com seu próprio tempo de execução;
- O tempo de execução dos métodos, a *thread* que o executou, de qual objeto foi realizada a chamada e de que classe tal objeto foi definido;
- É possível obter o estado das *threads* (em execução, bloqueada, em espera, finalizada). Com essas informações, usuários podem facilmente diferenciar o tempo que a *thread* gastou realizando computação *versus* esperando por outras *threads*. Além disso, permite observar o grau pelo qual diferentes *threads* são sincronizadas, e em quais monitores houve maior contenção por recurso;

- Através do identificador da *thread* e da relação de ordem entre os eventos, é possível identificar qual foi o método remotamente invocado gerando a transmissão/recepção de dados, de que objeto esse método foi invocado e em mais alto nível, de que classe é esse objeto;
- É possível obter o tempo decorrido entre uma invocação remota e o seu retorno, uma vez que, após a adequação dos relógios, ambos os tempos são medidos utilizando uma mesma referência de relógio.

6.2 Comportamento Local

O comportamento local da execução de uma aplicação Java refere-se aos eventos que ocorreram locais a uma determinada máquina virtual Java. Neste tipo de processamento é considerado apenas o Arquivo Detalhado de Traço da JVM da qual se quer obter as informações. Uma vez que o comportamento desejado é referente a uma JVM, não há a necessidade de realizar adequação de relógio e nem a ligação das invocações remotas.

O comportamento local pode ser obtido de forma *off-line* ou de forma *on-line*. O processamento *off-line* realizado para determinar o comportamento local é igual ao descrito na seção 6.1.3. Os eventos são ordenados cronologicamente e gravados no arquivo de saída. O formato do arquivo é o mesmo demonstrado na figura 6.2 com exceção da tupla identificada pelo caractere ‘S’ que não está presente neste tipo de processamento.

O comportamento *on-line* é determinado de acordo com a necessidade de informações da ferramenta destino. Normalmente, a quantidade de informações requisitadas neste tipo de processamento é bem menor do que no processamento *off-line*. Além disto, a velocidade com que estas informações são disponibilizadas deve ser maior para não inviabilizar a sua utilização. Por isto, na maioria das situações, as informações podem ser obtidas diretamente a partir dos Buffers de Armazenamento Local do DOMonitor.

A disponibilização destas informações pode ocorrer de forma periódica, de acordo com um intervalo de tempo pré-determinado ou a partir de requisições explícitas por meio de uma função disponibilizada pelo DOMonitor para tal fim, como é o caso do ISAM. A forma como o ISAM requisita as informações ao DOMonitor é detalhada na seção 7.2.5.

7 Integração com outros Ambientes

O comportamento da execução de uma aplicação obtido pelo DOMonitor pode ser utilizado por diversas classes de ferramentas que necessitem de informações mais detalhadas sobre a execução da aplicação. Neste capítulo propõe-se a integração de tais informações com duas ferramentas existentes: o Pajé (Visualização) e o ISAM (Infra-Estrutura de Suporte às Aplicações Móveis). Esta integração poderá ser utilizada para consolidar as informações geradas pelo DOMonitor.

7.1 Integração do DOMonitor com o Pajé

Certas características de Java, como múltiplas threads e eventos de rede, introduzem o não-determinismo no comportamento da execução da aplicação. Observando o comportamento geral do programa, o usuário tem maior facilidade em entender o funcionamento de sua aplicação, o relacionamento entre *threads* e a concorrência de eventos, agilizando a tarefa de depuração.

A **Visualização da Execução** é uma ferramenta essencial para auxiliar na depuração e refinamento de aplicações implementadas usando o modelo de programação de objetos distribuídos. Essas aplicações executam em diversos nodos, sendo que as *threads* em execução no mesmo nodo comunicam-se usando primitivas de sincronização e as *threads* em execução em nodos diferentes comunicam-se usando invocação remota de métodos. Portanto, para visualizar esse tipo de aplicações a ferramenta deverá ser escalável e suportar um modelo de programação de dois níveis.

O Pajé é uma ferramenta de visualização interativa que mostra a execução de aplicações paralelas. [STE 99] Foi projetado para suportar um grande número de *threads* que se comunicam entre si e que podem ter ciclos de vida disjuntos. Além disso, é capaz de representar uma grande variedade de interações entre as *threads.* Um dos maiores destaques do Pajé é combinar características como interatividade, escalabilidade e expansibilidade [KER 2000a] e [KER 2000b].

A **interatividade** permite inspecionar todos os objetos mostrados na tela e voltar a momentos anteriores no tempo, visualizando o estado dos objetos em um instante passado. A **escalabilidade** provê habilidade de tratar com um grande número de *threads*. A **expansibilidade** é uma característica importante por lidar com a evolução da interface de programação paralela e técnicas de visualização. Além disso, a expansibilidade possibilita estender o ambiente com novas funcionalidades: o processamento de novos tipos de traços, a adição de novos gráficos, a visualização de novos modelos de programação.

Todas essas características são extremamente importantes, no entanto, a expansibilidade é neste ponto destacada por permitir que uma ferramenta desenvolvida inicialmente para o modelo de programação paralela possa ser estendida para o modelo de Programação de Objetos Distribuídos Java.

7.1.1 Expansibilidade

Várias características do Pajé foram projetadas com o objetivo de alcançar a expansibilidade: arquitetura modular, a flexibilidade dos módulos de visualização e a generalidade do módulo de simulação.

Arquitetura Modular - a arquitetura do Pajé é baseada em componentes modulares que se comunicam utilizando um fluxo de dados pré-definido. A maioria desses componentes é independente da semântica do modelo de programação paralela. Portanto, a ferramenta de visualização pode ser aproveitada para outros modelos de programação, sem exigir o conhecimento interno e nem modificações em nenhum dos componentes do Pajé. Além disso, essa estrutura facilita a adição de novos módulos.

Flexibilidade dos Módulos de Visualização - Os componentes de visualização do Pajé não possuem dependência com nenhum modelo de programação paralelo. Para alcançar esse objetivo, o Pajé recebe inicialmente como entrada os tipos de objetos que serão visualizados, as relações entre eles e a forma como esses objetos devem ser visualizados (figura 7.1). As únicas restrições são: a natureza hierárquica dos tipos de relações entre os objetos visualizados e a habilidade de colocar cada um desses objetos na escala de tempo da visualização. Portanto, a definição da hierarquia de tipos pode ser realizada de acordo com o modelo de programação.

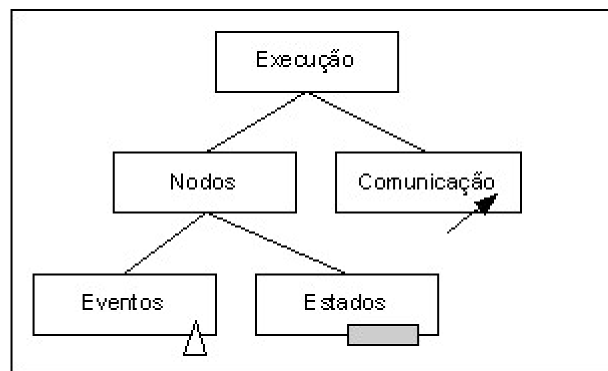


FIGURA 7.1 – Hierarquia de Tipos Simplificada

Generalidade do Pajé - o Pajé permite a definição de *o quê?* visualizar e *como* os objetos visualizados devem ser representados pelo Pajé. Ao invés de ser computada pelo componente de *simulação* projetado para um modelo de programação específico, a hierarquia de tipos dos objetos visualizados pode ser definida por desenvolvedores de ferramentas de tal forma que sejam repassados para o componente de *simulação* do Pajé. O *simulador* usa essas definições para construir uma nova árvore de tipos usada para relacionar os objetos a serem visualizados [STE 2001a].

7.1.2 Integração com o Pajé

O Pajé constrói a representação gráfica do comportamento do programa a partir dos arquivos de traço gerados durante a execução do programa. Usuários ou desenvolvedores de ferramentas podem adaptar o Pajé conforme suas necessidades sem ter de conhecer o funcionamento interno da ferramenta ou ter de alterar qualquer um dos seus componentes. Ou seja, a ferramenta é parametrizável, sendo possível definir *O Quê?* representar e *Como?* representar. Isso pode ser feito através da definição da hierarquia de tipos de objetos que serão visualizados, bem como, a forma como esses objetos devem ser visualizados [STE 2001a]. O formato do arquivo de traço utilizado pelo Pajé foi definido em [STE 2001a] e inclui quatro tipos diferentes de informação:

1. A descrição do formato das instruções genéricas;
2. Instruções genéricas, com a definição da hierarquia dos tipos de objetos que serão visualizados;

3. A definição do formato dos eventos do arquivo de traço;
4. Um conjunto de eventos registrados, de acordo com o formato definido, para serem utilizados na construção do gráfico de acordo com a hierarquia de tipos.

A integração dos dados de monitoração do DOMonitor ao Pajé é realizada com o auxílio de um filtro específico ao Pajé. Este filtro organiza os dados de monitoração em um formato compreensível pela ferramenta. Informações necessárias para que o Pajé possa demonstrar o comportamento da execução de uma aplicação Java incluem: (i) o formato dos eventos; (ii) os eventos gerados durante a execução da aplicação Java; (iii) a hierarquia de tipos utilizada pelo Pajé no momento de gerar os gráficos para visualização. A figura 7.2 mostra a integração entre as ferramentas de monitoração e visualização.

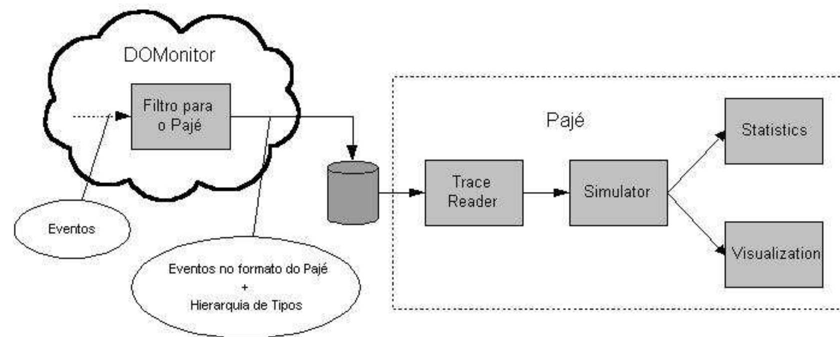


FIGURA 7.2 – Integração do DOMonitor ao Pajé

A) Hierarquia de Tipos

Para especializar o Pajé para o modelo de programação de objetos distribuídos Java é necessário definir o tipo de objetos que serão visualizados e a relação hierárquica entre estes objetos. Esta descrição é inserida pelo filtro do DOMonitor no começo do arquivo de traço usado como entrada para o processo de visualização. A hierarquia de tipos proposta para visualizar aplicações distribuídas Java monitoradas pelo DOMonitor é mostrada na figura 7.3.

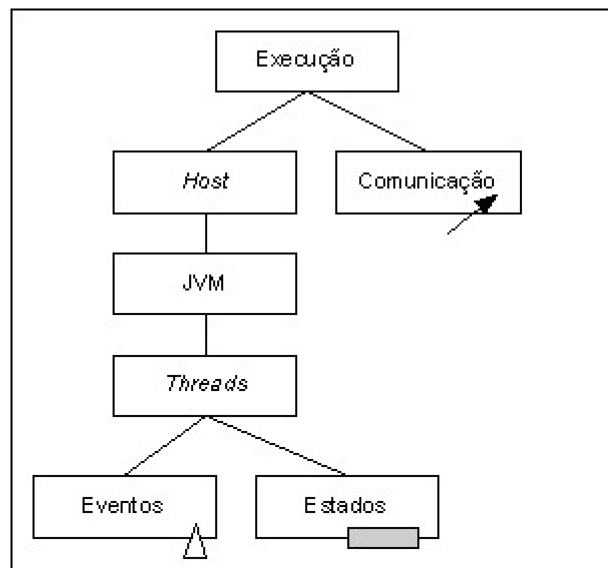


FIGURA 7.3 – Hierarquia de tipos para visualizar aplicações distribuídas Java

A hierarquia de tipos usada no Pajé pode ser relacionada a uma árvore, onde as folhas são chamadas **entidades** e nodos intermediários são **containeres**. Entidades são objetos elementares que podem ser visualizados, tais como: eventos, estados de *threads* ou comunicações. Containeres são objetos de nível mais alto, que podem conter entidades ou containeres de nível mais baixo (figura 7.3).

Uma chamada de função é disponibilizada para criar novos containeres e funções podem ser usadas para criar novos tipos de entidades, as quais podem ser eventos, estados, *links* e variáveis. Um **evento** é uma entidade que representa uma ação instantânea. **Estados** representam os comportamentos de interesse do container. Um ***link*** representa uma forma de conexão entre um container origem e um destino. Uma **variável** armazena a evolução histórica de valores sucessivos de dados associados com um container. As funções que podem ser usadas para definir novos tipos de containeres e entidades estão definidas em [STE 2001a].

B) Formato dos Eventos

O arquivo de traço resultante do processo de filtro é composto por eventos de diversos tipos, cada um com suas informações específicas. O Pajé trata desta característica permitindo que seja definido no início do arquivo de traço o formato dos eventos. A definição do formato dos eventos contém o nome do tipo do evento e o nome e tipo de cada um dos seus campos. Portanto, através do filtro do DOMonitor é definido o formato de cada um dos eventos que devem ser visualizados pelo Pajé. A definição segue o seguinte formato:

- todas as linhas devem começar com um caracter '`%`';
- a definição de um evento começa com uma linha `%EventDef` e termina com uma linha `%EndEventDef`;
- A linha `%EventDef` contém o nome e o número do evento. O nome é usado para identificar o tipo de um evento. O número do evento é usado para identificar o evento na segunda parte do arquivo de traço. Este número é definido pelo usuário;
- Os campos de um evento são definidos entre as linhas `%EventDef` e `%EndEventDef`, um campo por linha, definido pelo nome do campo e o seu tipo.

Após estas definições, estão os eventos gerados pelo DOMonitor durante a execução da aplicação. A correspondência de um evento com sua definição é feita por meio de um número, que seve ser único para cada descrição de evento. Com isto, a segunda parte do arquivo de traço contém um evento por linha, com os campos separados por espaços, sendo a primeira informação o número que identifica o tipo do evento, e logo após tem-se as informações relacionadas aos outros campos de acordo com a ordem pré-estabelecida. Na versão atual, o primeiro evento do arquivo de traço deve possuir os campos "StartTime" e "EndTime" do tipo "date", contendo respectivamente, a informação de tempo do primeiro e do último evento do arquivo de traço.

C) Visualização

Visualização de Threads no Pajé - A visualização da atividade de nodos *multithreaded* é realizada em um diagrama que representa os estados e as comunicações das *threads* (figura 7.4). O eixo horizontal representa o tempo enquanto threads são mostradas ao longo do eixo vertical, agrupadas por nodo. O espaço alocado para cada

nodo do sistema paralelo é ajustado dinamicamente ao número de threads sendo executadas nesse nodo. Comunicações são representadas por setas enquanto o estado das threads é mostrado por retângulos. Cores são utilizadas para indicar o tipo de comunicação ou a atividade de uma thread. O Pajé trata da escalabilidade através da utilização de filtros.

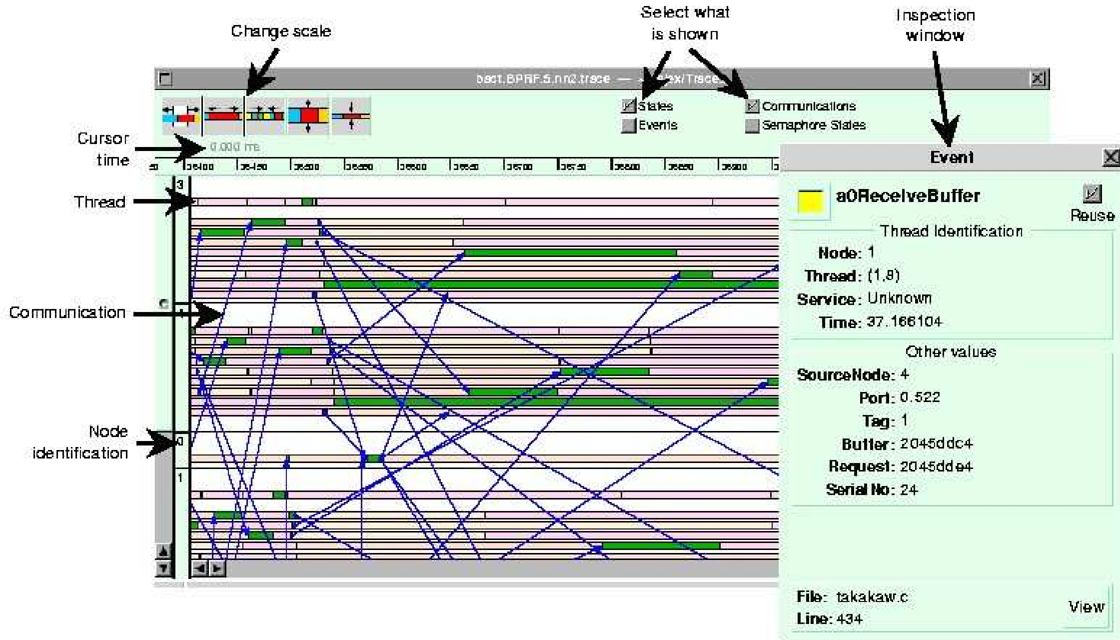


FIGURA 7.4 – Visualização de um programa Ahapascan pelo Pajé [KER 2000b]

Visualização de Sincronização – os estados dos semáforos e dos *locks* são representado no Pajé em um diagrama Espaço-Tempo (figura 7.4). Cada estado é associado a uma cor e um retângulo desta cor é mostrado na posição correspondente ao período de tempo em que este estado ocorreu. O Pajé reconhece três estados diferentes para um semáforo: (i) quando existe alguma *thread* bloqueada no semáforo; (ii) quando uma *thread* irá bloquear-se no semáforo se realizar uma operação ‘P’; (iii) quando uma *thread* puder realizar uma operação ‘P’ sem bloquear-se.

Quanto aos *locks*, se uma *thread* T_1 possuir o *lock*, e uma *thread* T_2 tentar obter o *lock*, T_2 ficará bloqueada até que T_1 libere o *lock*. Baseado nesse comportamento, uma forma de representar *locks* em um diagrama espaço-tempo é associar cada *lock* a uma cor, e um retângulo dessa cor é desenhado próximo a *thread* que o possui. Além disso, para facilitar a identificação das *threads* que estão bloqueadas esperando por um *lock*, um retângulo diferente com a cor do *lock* é mostrado próximo a essas *threads*. Essas representações podem ser utilizadas para demonstrar as atividades de sincronização de uma aplicação Java.

Pretende-se que com a integração dos dados de monitoração com uma ferramenta de visualização seja possível visualizar as aplicações desenvolvidas usando o DOBuilder.

7.2 Integração do DOMonitor com o ISAM

A computação móvel é uma nova proposta de paradigma computacional oriunda das tecnologias de rede sem fio e sistemas distribuídos. Nela, o usuário, portando dispositivos móveis como *palmtops* e *notebooks*, tem acesso a uma infra-

estrutura compartilhada e independente da sua localização física. Isto disponibiliza uma comunicação flexível entre as pessoas e um acesso aos serviços de rede. Observa-se que a crescente introdução de facilidades de comunicação tem deslocado as aplicações da computação móvel de uma perspectiva de uso pessoal (atual) para outras mais avançadas e de uso corporativo, como as aplicações móveis distribuídas [AUG 2001c].

7.2.1 O contexto do ISAM

A produção de software no ambiente móvel é complexa. Seus componentes são variáveis no tempo e no espaço em termos de conectividade, portabilidade e mobilidade. O desafio que se apresenta é projetar aplicações móveis distribuídas cujos níveis de serviço e disponibilidade de recursos são imprevisíveis. Existem, portanto, requerimentos emergindo para uma nova classe de aplicações projetadas especificamente para este ambiente dinâmico.

Esta nova classe de aplicações tem sido referenciada de muitas formas: *environment-aware, network-aware, resource-aware, context-aware applications*. Porém, todas têm um conceito embutido: adaptação. A diferença está no grau de adaptabilidade e nos recursos que são objetos da adaptação. Nesta perspectiva, a adequação do processo de monitoramento de recursos é central para a qualidade das decisões de adaptação tomadas.

Segundo Satyanarayanan [SAT 96], mobilidade exige adaptabilidade. Isto significa que os sistemas devem ter consciência da localização e da situação onde estão inseridos, e devem tirar vantagem desta informação para configurar-se dinamicamente de um modo distribuído. O foco da complexidade na implementação de aplicações móveis com comportamento adaptativo está no fato que os componentes distribuídos das mesmas sofrem influência dos diversos ambientes onde estão inseridos.

Sistemas distribuídos tradicionais são construídos com suposições sobre a infra-estrutura física de execução, como conectividade permanente e disponibilidade dos recursos necessários. Porém, essas suposições não são válidas nos sistemas móveis. Isto impede o uso direto das soluções adotadas pelos sistemas distribuídos, as quais podem ser altamente ineficientes devido à variabilidade freqüente da conexão à rede e da disponibilidade de recursos e serviços.

A computação móvel abrange uma faixa de cenários, os quais têm diferentes requerimentos no sistema de suporte. Uma categorização geral distingue entre dois cenários:

- **infra-estruturado** - cenário composto pela presença de uma rede fixa onde alguns *hosts*, referenciados como estações-base, constituem os pontos de acesso para os *hosts* móveis.
- ***ad-hoc*** - cenário dinâmico composto somente por *hosts* móveis (sem o suporte dado por uma rede fixa). A topologia resultante é altamente variável, constituída a partir das intersecções das áreas (células) de abrangência dos *hosts* móveis.

Considera-se que, para o desenvolvimento de aplicações distribuídas mais avançadas, é necessário que os *hosts* móveis usufruam a infra-estrutura da rede fixa existente e possam se beneficiar de ambientes como o oferecido pela Internet. Desta forma, o modelo de rede adotado é o de uma rede móvel infra-estruturada. Este modelo

é refletido nos elementos básicos do ambiente de execução do sistema ISAM [AUG 2001a] e [BAR 2001c] apresentado na figura 7.5. Estes elementos são:

- HoloBase - é o ponto inicial de contato do *host* móvel com os serviços ISAM residentes na parte fixa da rede. Possui as funções de identificação, autenticação e de ativação das ações básicas do sistema;
- HoloCélula - denota a área de atuação de uma HoloBase, e é composta pela mesma e por HoloSítios;
- HoloSítio - são os nodos do sistema responsáveis pela execução da aplicação móvel distribuída propriamente dita. Nestes também são processados serviços de gerenciamento ISAM;
- HoloSítioMóvel - são os nodos móveis do sistema, responsáveis por serviços de interface com o usuário, e por algumas funções de monitoramento de recursos ISAM;
- HoloHome - é um ponto de referência único por usuário móvel no âmbito de toda rede. Está associado a um HoloSítio registrado para tal na arquitetura.

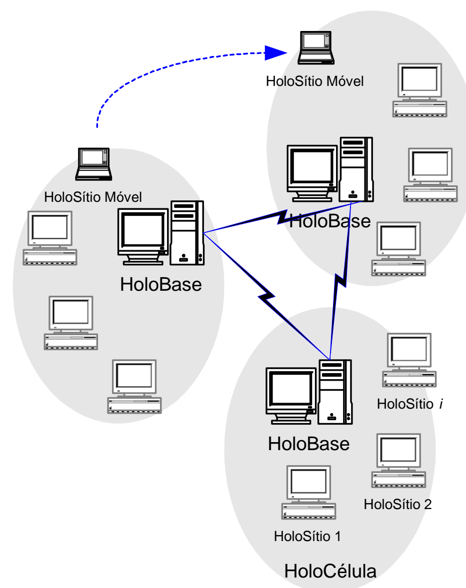


FIGURA 7.5 – Componentes do Ambiente ISAM

A crescente disponibilidade de facilidades de comunicação tem deslocado as aplicações da computação móvel de uma perspectiva de uso pessoal para outras mais avançadas de uso corporativo.

Parece, portanto, ser necessário definir uma nova arquitetura para sistemas móveis, projetada com mobilidade, flexibilidade e adaptabilidade intrínsecas. Com esta motivação, está em desenvolvimento o projeto ISAM (Infra-estrutura de Suporte às Aplicações Móveis Distribuídas). O DOMonitor atende uma parte significativa das informações necessárias ao processo de tomada de decisão no ISAM. O contexto em que se insere o DOMonitor está caracterizado na seção 7.2.3, a seguir.

7.2.2 A Arquitetura ISAM

O principal desafio no tratamento da adaptação é a sua complexidade. Desta forma, a arquitetura ISAM foi organizada em módulos independentes, abordando cada um os vários aspectos envolvidos no comportamento adaptativo da aplicação. A arquitetura proposta é organizada em camadas com níveis diferenciados de abstração e está direcionada para a busca da manutenibilidade da qualidade de serviços oferecida ao usuário móvel através do conceito de adaptação [AUG 2001b].

No ISAM, o sistema se adapta para fornecer qualidade, enquanto que a aplicação se adapta para manter a qualidade dentro da expectativa do usuário móvel. Uma visão organizacional desta arquitetura é apresentada na figura 7.6. Salientam-se dois pontos. Primeiro, a adaptação permeia todo o sistema, por isto está colocada em destaque na representação do ISAM. Segundo, o escalonador é o "core" da arquitetura. As decisões de adaptação, tanto da aplicação quanto do escalonador, são baseadas no perfil do comportamento de três entidades: o usuário móvel, a aplicação e o sistema em si, os quais compõem o contexto de execução.

A camada superior (SUP) da arquitetura é composta pela aplicação móvel distribuída. A construção desta aplicação baseia-se nas abstrações do Holoparadigma [BAR 2001a] e [BAR 2001b], as quais permitem expressar mobilidade, acrescidas de novas abstrações para expressar adaptabilidade (ISAMadapt).

Por sua vez, a camada inferior (INF) é composta pelas tecnologias dos sistemas distribuídos existentes, tais como sistemas operacionais nativos e a Máquina Virtual Java. Além disso, a arquitetura ISAM foi concebida de forma a não excluir os mecanismos de adaptação nativos das tecnologias que compõem esta camada e, assumindo-se a existência de uma rede infra-estruturada (rede fixa + rede móvel) que forneça o suporte necessário para o acesso aos recursos da rede em escala global.

O DOMonitor tem sua atuação na camada intermediária do *middleware* ISAM. A mesma será caracterizada na seção 7.2.5.

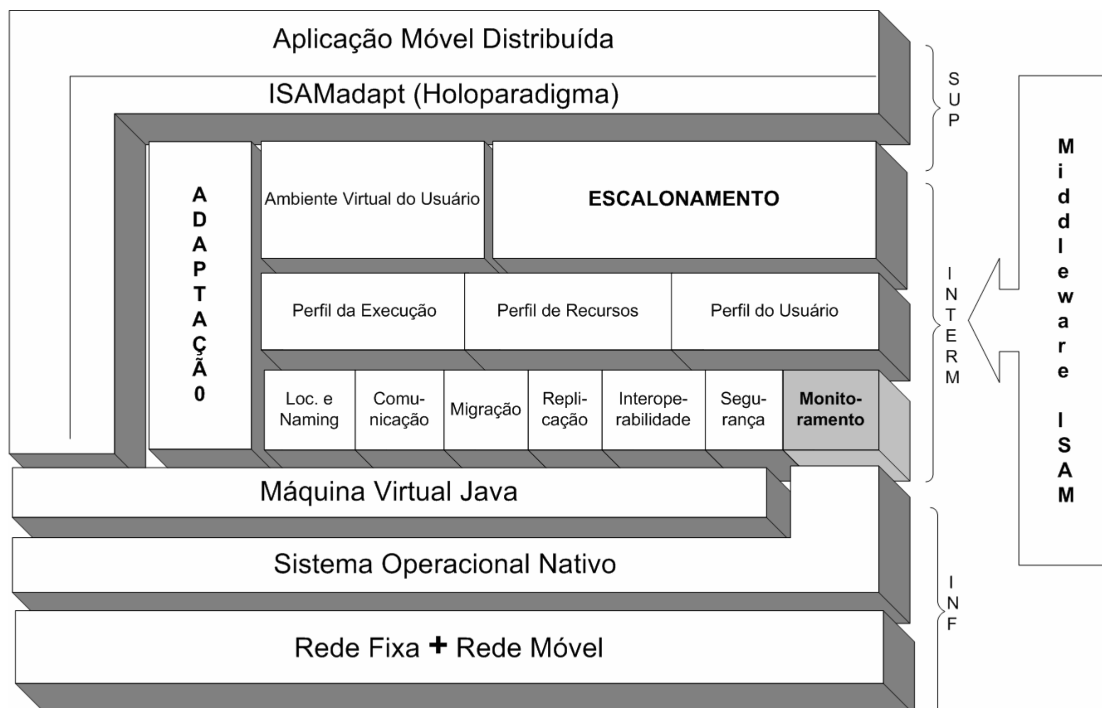


FIGURA 7.6 – Arquitetura ISAM

7.2.3 O DOMonitor no *Middleware* ISAM

A) A Camada Intermediária - Primeiro Nível

A camada intermediária (INTERM) é o núcleo funcional da arquitetura ISAM, sendo fornecida em três níveis de abstração. O primeiro nível é composto por dois módulos de serviço à aplicação: Escalonamento e Ambiente Virtual do Usuário. O escalonamento, por sua vez, é o componente-chave da adaptação na arquitetura ISAM. O Ambiente Virtual do Usuário (AVU) compõe-se dos elementos que integram a interface de interação do usuário móvel com o sistema. O ISAM suporta a exploração de aplicações contextualizadas (adaptadas aos recursos, serviços e localização corrente) e individualizadas (adaptadas aos interesses e preferências do usuário móvel). O desafio da adaptabilidade é suportar os usuários em diferentes localizações com diferentes sistemas de interação que demandam diferentes sistemas de apresentação dentro dos limites da mobilidade. Este módulo deve caracterizar, selecionar e apresentar as informações de acordo com as necessidades e o contexto em que o usuário se encontra. Para realizar estas tarefas, o sistema se baseia num modelo de uso onde as informações sobre o ambiente de trabalho, preferências, padrões de uso, padrões de movimento físico e hardware do usuário são dinamicamente monitoradas e integram o Perfil do Usuário e da Aplicação.

B) A Camada Intermediária - Segundo Nível

No projeto da arquitetura ISAM busca-se um conceito flexível de adaptação que está relacionado ao contexto em que a aplicação está inserida. A mobilidade física introduz a possibilidade de movimentação do usuário durante a execução de uma aplicação. Desta forma, a localização corrente do usuário determina o contexto de execução, definido como "toda informação, relevante para a aplicação, que pode ser usada para definir seu comportamento" [AUG 2001a]. Numa análise preliminar, o contexto é determinado através de informações de quem, onde, quando, o que está sendo realizado e com o que está sendo realizado. Obter essas informações é a tarefa do módulo de monitoramento, que atua tanto na parte móvel quanto na parte fixa da rede.

As informações que dirigem as decisões do escalonador e dão suporte à aplicação para sua decisão de adaptação (ISAMadapt) são advindas de três fontes: perfil da execução, perfil dos recursos e perfil do usuário e da aplicação.

C) A Camada Intermediária - Terceiro Nível

No terceiro nível da camada intermediária estão os serviços básicos do ambiente de execução ISAM, e é neste nível que atua o DOMonitor.

O módulo monitoramento do ISAM, do qual o DOMonitor é parte integrante, obtém informações do acompanhamento das aplicações executadas pelo usuário, em um dado tempo e em um dado local, com determinados parâmetros. O que permite determinar a evolução histórica e quantitativa das entidades monitoradas. A interpretação destas informações estabelece o perfil do usuário e das aplicações. Desta forma, as aplicações móveis ISAM poderão se adaptar à dimensão pessoal, além das dimensões temporal e espacial [AUG 2001a].

7.2.4 O Escalonamento no *Middleware* ISAM

No ISAM, as aplicações solicitam direta ou indiretamente recursos do escalonador. Algumas podem especificar uma determinada necessidade de qualidade de serviço (QoS), outras podem aceitar o “melhor-possível” nos níveis de serviço [MAH

99]. Assim, o módulo de escalonamento do *middleware* tem a estratégia de trabalhar com diferentes políticas de gerenciamento para diferentes aplicações, usuários e/ou domínios de execução. Neste caso, as estratégias de adaptação exigem do mecanismo de escalonamento o tratamento de problemas de otimização utilizando critérios múltiplos [YAM 2001a].

Como forma de reduzir a complexidade do escalonamento, e prover comportamento adaptativo nas diferentes dimensões e níveis de adaptação possíveis na computação móvel, o ISAM adota a estratégia denominada **Adaptação Colaborativa Multinível**, onde:

- o sistema (escalonador) é responsável por determinadas adaptações, normalmente relativas ao desempenho e gerência de recursos do ambiente;
- a aplicação é responsável por decisões de adaptações específicas de seu domínio e situações de uso;
- ambos, são responsáveis por uma negociação de decisões de adaptação.

Neste sentido, um problema de pesquisa em andamento, é estabelecer o nível de cooperação requerido entre projetistas de sistemas e projetistas de aplicações para criar protocolos aceitáveis para ambos os grupos [AUG 02].

Por outro lado, o ISAM gerencia diversas aplicações que concorrem na utilização dos recursos. A otimização da execução de um subconjunto do total de aplicações não pode comprometer o nível mínimo de QoS necessário para o restante. Desta forma, o escalonador precisa trabalhar com uma visão global das execuções em andamento [VAH 99]. Este é um dos requisitos, que faz com que a Gerência do Monitoramento integre as informações provenientes dos DOMonitors existentes nos HoloSítios do ambiente de execução. Nas próximas seções, serão descritos os princípios utilizados na modelagem do escalonamento no ISAM.

A) Organização Física do Escalonamento

A forma como é organizada a distribuição dos equipamentos afeta diretamente todos os serviços do *middleware*, e naturalmente o escalonamento. A organização adotada no ISAM é a **celular hierárquica** (vide figura 7.5). Nesta, os equipamentos pertencentes a uma mesma célula comunicam-se diretamente (utilizando uma organização plana). Nas comunicações com o exterior um equipamento específico atua como fronteira. A característica hierárquica faculta que uma célula possa recursivamente conter outras.

Esta organização atende a necessidade de confinamento de contexto inerente ao modelo de programação adotado para o ISAM, o Holoparadigma [BAR 2001b]. Este modelo é baseado em eventos associados a escopos. Outrossim, a organização celular hierárquica é orgânica com o Holoparadigma, o qual trabalha com o conceito de entes (entidade de modelagem). O conceito de entes também contempla a possibilidade de agrupamento hierárquico. Tal associação se mostra oportuna ao mapeamento e/ou à alocação dinâmica de tarefas.

Em função das características do software e do hardware, o escalonamento no *middleware* ISAM utiliza uma organização fisicamente distribuída e cooperativa representada na figura 7.7. Como principais premissas passíveis de serem atingidas por esta organização tem-se: tolerância-a-falhas, escalabilidade, autonomia dos domínios administrativos (HoloCélulas) e suporte a múltiplas políticas de escalonamento.

A proposta está baseada em dois escalonadores: (*i*) **EscWAN** e (*ii*) **EscEnte**:

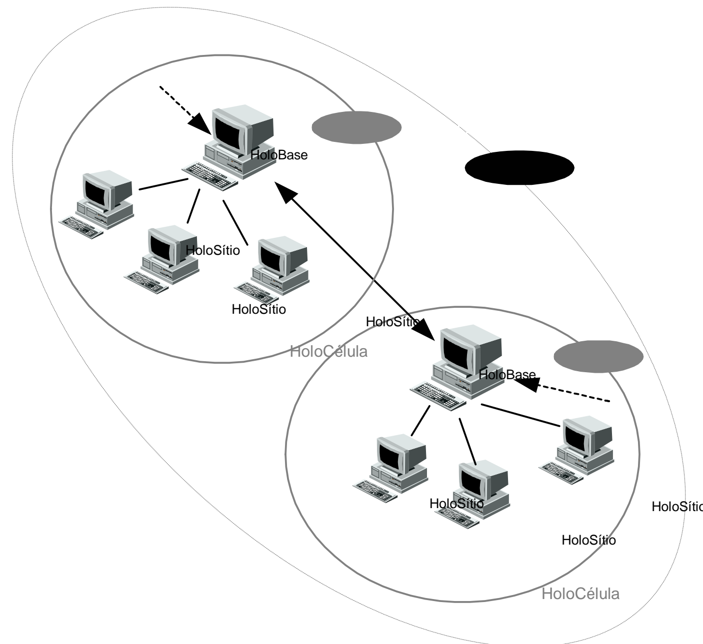


FIGURA 7.7 – Organização do Escalonamento

EscWAN: fica localizado no nodo HoloBase, com atuação entre as HoloCélulas. O mesmo tem atribuições no gerenciamento global da arquitetura, tais como:

- localizar recursos (hardware e software) mais próximos, para reduzir custos de comunicação;
- decidir quando e onde replicar serviços e/ou componentes de software (entes);
- decidir quando e para onde migrar os componentes de software;
- instanciar o Ambiente Virtual do Usuário nos HoloSítios. Esta instanciação é feita sob duas óticas: (*i*) balanceamento de carga - neste caso é escolhido o nodo menos carregado, (*ii*) aspectos de afinidade da aplicação - exigência de memória, bases de dados, etc.;
- disponibilizar antecipadamente, por usuário, a demanda de componentes das aplicações e dos dados;
- repassar ao escalonador EscEnt a carga de trabalho (componentes de software) proveniente de outras HoloBases.

Pelas suas atribuições, além da consideração de custos de comunicação e balanceamento de carga, o escalonador EscWAN atua de forma intensiva sobre aspectos de replicação e migração.

EscEnte: também existente em todas as HoloBases, tem atribuições no gerenciamento interno da HoloCélula, tais como:

- efetuar o mapeamento dos componentes da aplicação nos HoloSítios da HoloCélula. Os critérios utilizados também são balanceamento de carga e de afinidade funcionais;

- dar suporte aos procedimentos de adaptação colaborativa multinível com a aplicação.

Uma estratégia do escalonador EscEnte é associar o contexto (a HoloCélula, as aplicações e os usuários) a grupos de escalonamento [CAV 98], onde cada grupo pode definir políticas especificas de balanceamento de carga. Cabe ao mecanismo de escalonamento gerir a evolução da execução das aplicações dos diferentes grupos segundo as políticas por eles selecionadas.

7.2.5 A Monitoração com o DOMonitor no ISAM

No ISAM o escalonamento é alimentado por métricas oriundas de dois níveis distintos:

- no nível de sistema tem-se dois tipos principais de métricas: (*i*) uma para caracterização do *workload* dos *hosts* e (*ii*) outra para construção de perfis de comunicação entre objetos. Os mecanismos para capturar os perfis de comunicação entre os objetos são integrados com a API RMI de Java, preservando a compatibilidade com a semântica nativa da linguagem [SIL 2001];
- no nível de aplicação a monitoração é feita utilizando o DOMonitor [ARA 2001].

A relação do DOMonitor com o ISAM está retratada na figura 7.8. Na mesma estão destacados os principais componentes do contexto de monitoramento, quais sejam:

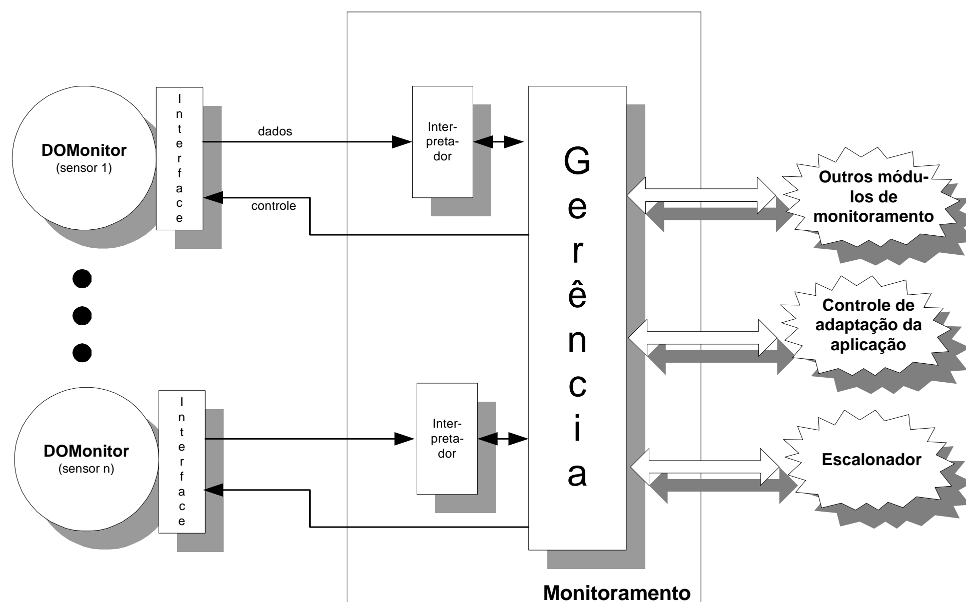


FIGURA 7.8 – DOMonitor na Arquitetura ISAM

A) Sensor

O sensor coleta eventos pertinentes aos códigos Java das aplicações em execução. Este código poderá corresponder a uma aplicação do usuário, bem como a algum mecanismo de controle da arquitetura ISAM. Os sensores são ativados e programados pela Gerência do monitoramento.

A ativação do sensor é realizada a partir de uma chamada de função. O DOMonitor disponibiliza uma função chamada `AtivaSensor` com a seguinte estrutura:

```
boolean AtivaSensor()
```

onde o valor de retorno indica que a ativação foi bem sucedida.

Assim como existe uma função para ativar o sensor, o DOMonitor disponibiliza uma função específica para desativá-lo. A função `DesativaSensor` tem a seguinte estrutura:

```
boolean DesativaSensor()
```

onde o valor de retorno indica que a desativação foi bem sucedida.

Este processo de ativar e desativar o sensor é disponibilizado pelo DOMonitor para flexibilizar sua utilização. Com isto, desde que corretamente inicializado, o DOMonitor pode permanecer inativo (sem causar *overhead*) até que seja explicitamente ativado. Após obter a funcionalidade desejada pode ser novamente desativado pelo ISAM. Este ciclo ativação/desativação pode ser executado quantas vezes forem necessárias.

B) Interface de Controle do Sensor

Através desta interface é programada a atuação do DOMonitor. Os principais parâmetros que a gerência do monitoramento negocia com a mesma são:

- que eventos monitorar;
- a periodicidade de atuação do DOMonitor.

Estes parâmetros são provenientes tanto da aplicação, no momento que esta registra o seu interesse específico de monitoramento, como do escalonador que caracteriza as informações necessárias para administração geral da arquitetura. Esta interface também é responsável por funções de translação das informações de baixo nível oriundas do DOMonitor, para as padronizadas para o ISAM.

A definição de **quais eventos monitorar** pode ser feita dinamicamente através de duas funções disponibilizadas pelo DOMonitor, uma para habilitar e outra para desabilitar a monitoração de um determinado evento. As fuções possuem a seguinte estrutura:

```
boolean HabilitaMonitoracaoEvento(int tipo_evento, char *nome)
boolean DesabilitaMonitoracaoEvento(int tipo_evento, char *nome)
```

onde `tipo_evento` representa o evento que deseja-se habilitar/desabilitar; `nome` é um argumento opcional utilizado quando se deseja habilitar/desabilitar um ente específico dentro da máquina virtual; o valor de retorno indica se a operação foi bem sucedida.

A **requisição das informações de monitoração** é realizada através de uma chamada a função `RequisitaInformacao`, que possui a seguinte estrutura:

```
boolean RequisitaInformacao(int tipo_evento, char *nome, char *infos[])
```

onde `tipo_evento` é um argumento que representa o evento do qual se deseja obter informações. Para ober informações sobre todos os eventos, informa-se `0`; `nome` é um argumento opcional utilizado quando se deseja obter informações sobre um ente específico dentro da máquina virtual; no terceiro argumento, `infos`, são

disponibilizadas as informações requisitadas; o valor de retorno indica se a operação foi bem sucedida.

C) Interpretador

A cada sensor (instância de visão do DOMonitor) é associado um interpretador. Os parâmetros do interpretador são especificados pelos mecanismos de controle da adaptação tanto da aplicação como do escalonador. Seu principal objetivo é parametrizar as informações repassadas à gerência do monitoramento.

D) Gerência do monitoramento

A gerência do monitoramento coordena as informações coletadas em determinado HoloSítio. Como algumas decisões de escalonamento e conseqüentemente de adaptação dependem do estado global da execução, e esta ocorre de forma distribuída as Gerências do Monitoramento nos diversos HoloSítios, quando necessário, trocam informações através de um protocolo de comunicações. Dentre as atribuições da Gerência do Monitoramento, temos:

- registrar o interesse de monitoramento das aplicações em execução no HoloSítio;
- registrar o interesse de monitoramento dos escalonadores EscWAN e EscENTE;
- aglutinar as informações já parametrizadas de diversos sensores;
- repassar para os escalonadores as informações de monitoramento de seu interesse;
- repassar para a aplicação as informações de monitoramento de seu interesse;
- comunicar-se, quando necessário, com as outras Gerências do Monitoramento.

No que diz respeito aos dados monitorados, o fato das aplicações serem distribuídas, com componentes de software em diferentes contextos de execução, exige heurísticas aglutinadoras para determinadas tomadas de decisão. Os escalonadores EscWAN e EscENTE são responsáveis por aglutinar as informações de monitoramento que se inter-relacionam, sejam estas provenientes do monitoramento de um ou diversos HoloSítios.

8 A Implementação do DOMonitor

Este capítulo apresenta alguns detalhes de implementação e funcionamento do DOMonitor, mais especificamente do agente de monitoração do DOMonitor, o DOProf. A segunda parte do capítulo apresenta uma comparação entre as características e funcionalidades do DOMonitor e os trabalhos relacionados apresentados no capítulo 3.

8.1 Agente de Monitoração - DOProf

O DOProf é implementado como uma biblioteca de ligação dinâmica (.dll no Windows, .so no Unix). Esta biblioteca deve ser carregada pela JVM para inicializar o suporte de monitoração, ou seja, só serão coletadas informações das JVM's que carregarem a biblioteca DOProf. A implementação desta biblioteca utiliza a JVMPI (*Java Virtual Machine Profiler Interface*) [VIS 2000], [SUN 2001a]. A JVMPI fornece um mecanismo para o DOProf interagir com a JVM e registrar os dados da execução.

A figura 8.1 demonstra a interação entre a máquina virtual e o agente de monitoração do DOMonitor, o DOProf. O agente de monitoração é independente da implementação da máquina virtual. O DOProf é carregado no mesmo processo do sistema operacional em que executa a JVM. O DOProf e a JVM interagem através da chamada de funções. Uma vez que estas chamadas de funções são executadas no mesmo processo, a intrusão por parte do agente de monitoração fica minimizada.

A utilização de chamadas de funções é uma boa proposta para uma interface binária eficiente entre o agente de monitoração e diferentes implementações de máquina virtual. O envio de eventos através de chamadas a funções provoca um *overhead* adicional comparado à instrumentação direta a máquina virtual. No entanto, a maioria dos eventos são enviados em situações onde a JVM pode tolerar o custo adicional de uma chamada de função [VIS 2000]. Além disso, essa proposta fica independente da implementação da máquina virtual, enquanto que instrumentando a máquina virtual torna a proposta dependente da plataforma.

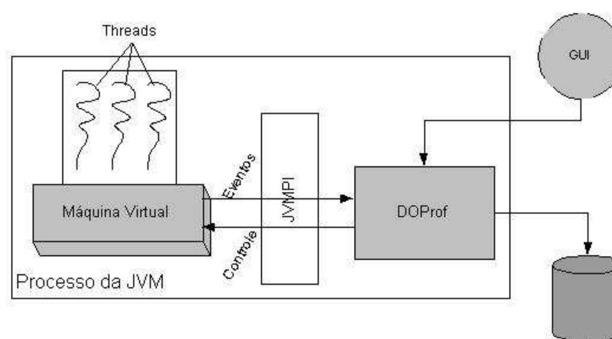


FIGURA 8.1 – Interação entre os componentes de monitoração

Para que o DOProf seja carregado em conjunto com a máquina virtual, deve ser utilizada uma opção especial, `-Xrun`, conforme mostra a linha de comando a seguir:

>java –XrunDOProf ClassName

onde DOProf é o nome da biblioteca de ligação dinâmica que contém o agente de monitoração que será carregado em conjunto com a JVM e ClassName é o nome da

aplicação Java que será executada. Cabe ressaltar que não é necessária uma versão da máquina virtual especialmente instrumentada para suportar a coleta de eventos.

8.1.1 Funcionamento

A JVM carrega o DOProf quando é inicializada e procura por um ponto de entrada específico. O ponto de inicialização entre o DOProf e a JVM é através da chamada a função: `JVM_OnLoad(JavaVM * jvm, char *options, void *reserved)`, a partir da qual obtém-se um ponteiro para uma instância da JVM, a string com os parâmetros digitados na linha de comando. Se a carga for bem sucedida e a inicialização foi realizada, a função retorna um status de OK. Caso contrário, o status é de erro.

O próximo passo realizado entre a máquina virtual e o DOProf é a inicialização de um ponteiro para a tabela de funções da JVMPI. Essa tabela consiste de dois conjuntos de funções: um implementado pelo agente de monitoração e outro implementado pela máquina virtual. Isto é feito através de uma chamada `GetEnv` no ponteiro `JavaVM`. A estrutura JVMPI define a interface de chamada de funções entre o agente de monitoração e a JVM.

A máquina virtual Java implementa um conjunto de funções para permitir ao agente de monitoração configurar parâmetros e obter mais informações a respeito da notificação de eventos. A máquina virtual faz chamadas a funções para informar ao DOProf sobre os vários eventos que ocorrem durante a execução da aplicação Java. O agente recebe os eventos e realiza chamadas na máquina virtual para proceder requisições de controle ou para obter mais informações em resposta a eventos específicos.

O DOProf precisa informar a máquina virtual sobre os eventos para os quais deseja receber notificação. Isto é feito através de uma função do JVMPI chamada `EnableEvent`. Esta função deve ser chamada tantas vezes quantos forem os eventos de interesse definidos para a seção de monitoração. Com isso, é habilitada a notificação para todos os eventos de interesse definidos para a classe de monitoração escolhida através da interface gráfica do DOMonitor.

O primeiro evento de interesse do DOMonitor que é gerado pela máquina virtual, é o `JVM Init Done`. Este evento usado pelo DOProf para inicializar suas estruturas de dados e criar *threads* do ambiente de monitoração.

O DOProf utiliza diversas constantes para alocação de suas estruturas de dados. Estas constantes podem ser definidas no arquivo DOProf.conf, que deve estar do mesmo diretório da biblioteca de ligação dinâmica DOProf. O formato do arquivo pode ser observado na figura 8.2.

Arquivo de Configuração do DOProf

```
#MAX_NRO_CLASSES 200
#MAX_NRO_METODOS 2000
#MAX_NOME_METODO 50
#TAMANHO_BUFFER_THREAD 5000
#LIMITE_OCUPACAO_BUFFER 80
#TAMANHO_BUFFER_GLOBAL 3
```

FIGURA 8.2 – Formato do Arquivo de Configuração do DOProf

A Tabela Hash de Classes foi implementada como um vetor de NRO_MAX_CLASSES posições. A função hash utilizada na versão atual do DOProf é extremamente simples, sendo que o endereço é calculado a partir do resto da divisão do identificador da classe C_{ID} por NRO_MAX_CLASSES. A partir da chave calculada, é possível acessar o elemento do vetor onde serão armazenadas e/ou consultadas as informações relativas a uma classe. Cada registro da tabela de classes é um elemento do tipo **Registro_Classe**. A definição do tipo Registro_Classe pode ser observada na figura 8.4.

```
Tipo Registro_Classe = registro
       CID: long int;
       Nro_metodos: byte;
       MID: Ponteiro para vetor de identificadores de método;
       P: Ponteiro para Registro_Classe;
  Fim_Registro;
```

FIGURA 8.3 – Definição do tipo Registro_Classe

Cada variável definida a partir deste tipo, ocupará 13 bytes na memória, sendo 4 bytes referentes ao campo C_{ID}, 1 byte referente ao campo Nro_metodos, 4 bytes para o ponteiro M_{ID} e 4 bytes para o ponteiro P. A tabela de classes ocupará o número de bytes definidos para cada elemento do tipo Registro_Classe (13 bytes) multiplicado pela constante NRO_MAX_CLASSES. No caso da constante ter sido definida como 200 (situação atual dos testes), a estrutura ocupa 13 x 200 = 2600 bytes.

O restante da ocupação é dinâmica, sendo alocada e liberada memória de acordo com a inserção e a remoção de registros da tabela. O ponteiro M_{ID} contém a referência de memória para um vetor definido dinamicamente no momento em que é inserido um registro na tabela de classes. Este vetor tem tantas posições quantos forem o número de métodos da classe. Cada elemento deste vetor é do tipo long int, e portanto, ele ocupara 4 bytes * o número de métodos da classe. O ponteiro P é utilizado para controlar as colisões que podem vir a ocorrer. Portanto, se o endereço calculado para um determinado C_{ID} já estiver sendo ocupado por outro C_{ID}, será alocada memória para um elemento do tipo Registro_Classe onde será inserido o elemento. O ponteiro P irá conter a referência a esta área de memória.

Cada elemento da Tabela Hash de Métodos é do tipo Registro_Metodo. O registro é definido, conforme a figura 8.5.

```
Tipo Registro_Metodo = registro
       MID: long int;
       Nome: string[MAX_NOME_METODO];
       Linha: int;
       Invocado: boolean;
       P: Ponteiro para Registro_Metodo;
  Fim_Registro;
```

FIGURA 8.4 – Definição do Registro_Metodo

Em uma situação hipotética em que a constante MAX_NOME_METODO tem o valor 50 e a constante MAX_NRO_METODOS tem o valor 2000, a tabela de métodos ocupará (4 + 50 + 2 + 1 + 4) * 2000 = 122000 bytes.

Os valores determinados para as constantes afetam de forma direta a quantidade de memória utilizada pelo agente de monitoração bem como ao *overhead* gerado pela ferramenta. Algumas considerações e cuidados devem ser considerados no momento de definir as seguintes constantes:

- `MAX_NRO_CLASSES` e `MAX_NRO_METODOS` – para um valor pequeno demais a tendência é aumentarem as colisões e com isso a alocação dinâmica de memória e o *overhead* de processamento. No entanto, diminui a quantidade de memória alocada estaticamente. Para valores grandes, a tendência é que exista um número menor de colisões e conseqüentemente diminua a alocação dinâmica de memória e *overhead* de processamento. No entanto, a quantidade de memória alocada e não utilizada pode ser maior;
- `MAX_NOME_METODO` – esta constante é utilizada para definir o tamanho do campo que armazena o nome dos métodos. Um valor muito grande para esta constante, implica em maior quantidade de memória alocada para a tabela de métodos. Isto porque, cada registro da tabela de métodos tem um campo definido de acordo com esta constante. Um valor muito pequeno para esta constante, pode ocasionar em o nome de determinados métodos ser truncado.

As tabelas de Métodos e de Classes são manipuladas apenas pela *thread* principal do DOProf, a *thread* responsável pelo recebimento das notificações e geração dos registros. A *thread* principal do agente de monitoração é notificada toda vez que houver a ocorrência de um evento de interesse. Esta notificação é realizada pela JVM a partir da chamada a função `NotifyEvent`. Essa função recebe como parâmetro informações relacionadas ao evento do qual está recebendo a notificação de ocorrência, em uma estrutura do tipo `JVMPI_Event`.

A estrutura `JVMPI_Event` contém **um inteiro indicando o tipo do evento**, **o identificador da *thread* que executou o evento** e outras **informações específicas ao evento**. As informações específicas são representadas como uma união de estruturas específicas aos eventos. Além destas informações, o DOProf obtém uma informação de tempo, cuja precisão é em milisegundos. Mais detalhes sobre as informações específicas a cada um dos eventos podem ser obtidos na documentação do JVMPI [SUN 2001a]. Das informações recebidas da JVM, o DOProf registra para cada um dos eventos de interesse as informações pertinentes conforme o formado das tuplas apresentadas no capítulo 5. As informações são registradas no Buffer Local da *Thread*.

Para cada nova *thread* criada na aplicação que está sendo monitorada, é criada uma área de armazenamento local, o **Buffer Local da *Thread***. A criação deste buffer é realizada a partir de uma função do JVMPI, chamada `SetThreadLocalStorage`. O tamanho deste buffer é definido de acordo com a constante `TAMANHO_BUFFER_THREAD`.

O armazenamento das informações em disco é realizado por uma *thread* auxiliar de menor prioridade, chamada *Thread* de I/O. Esta *thread* é criada através da chamada a função `CreateSystemThread`. Esta *thread* obtém as informações do **Buffer Global de I/O** e armazena no arquivo de traço detalhado. O tamanho do buffer global de I/O é definido de acordo com o resultado do produto entre as constantes `TAMANHO_BUFFER_GLOBAL` e `TAMANHO_BUFFER_THREAD`. Este buffer é

logicamente dividido em blocos, sendo que cada bloco possui o tamanho do TAMANHO_BUFFER_THREAD.

É definido um vetor de caracteres com o número de elementos correspondente à constante TAMANHO_BUFFER_GLOBAL. Esse vetor é utilizado para indicar quais áreas lógicas do Buffer Global de I/O estão ocupadas em um determinado momento. Uma vez que esta estrutura é atualizada e consultada por duas *threads*, a *Thread* de I/O e a *Thread* Principal, as atualizações a este vetor devem ser sincronizadas.

As informações do Buffer Local da *Thread* são descarregadas para o Buffer Global de I/O em quatro situações: (i) a *thread* correspondente ao buffer completou sua execução; (ii) a aplicação acabou de executar; (iii) foi ultrapassado o percentual de ocupação do buffer definido pelo índice LIMITE_OCUPACAO_BUFFER; (iv) o DOProf recebeu uma requisição através da chamada a função RequisitaInformacao;

A razão do Buffer Global de I/O ser dividido em blocos lógicos, cada um do tamanho do Buffer Local da *Thread*, é reduzir a probabilidade de ocorrência de contenção por recursos. Isso porque, mesmo que a *Thread* de I/O esteja descarregando informações em disco, pode ser necessário descarregar o Buffer de Local de outras *Treads*. Através da existência de vários blocos lógicos, é possível perceber que pode haver no buffer Global de I/O informações relacionadas a diversas *threads* ao mesmo tempo.

A ocupação das áreas do Buffer Global é verificada através do vetor de ocupação. A *thread* verifica que uma área está livre, obtém o *lock* sobre o vetor, atualiza esta posição para 'G' gravação e libera o *lock*. Depois grava todas as informações e obtém o *lock* novamente para atualizar a posição para 'O' ocupado. Então o *lock* é novamente liberado. A *thread* de I/O atualiza uma posição deste vetor com o valor 'L' livre quando termina de descarregar um determinado bloco lógico em disco. Após isto, procura por blocos lógicos que tenham a situação 'O' para descarregar em disco. Se não encontrar nenhuma informação a descarregar, a *thread* entre em modo de espera e vai ser despertada quando o Buffer Global de I/O possuir informações a serem descarregadas.

O evento JVM Shut Down é o último evento gerado e é usado para liberar todas as estruturas de dados utilizadas e encerrar as *threads*.

8.2 Análise Comparativa

DOMonitor x Paradyn: a proposta do projeto Paradyn [MIL 95] é fornecer uma ferramenta para avaliação de desempenho de programas paralelos e distribuídos de longa duração e automatizar boa parte da procura por gargalos de desempenho. O Paradyn procura diminuir o *overhead* através de uma técnica que instrumenta a aplicação dinamicamente e controla automaticamente essa instrumentação com o intuito de encontrar problemas de desempenho. O Paradyn pode identificar o tipo e a localização dos gargalos de desempenho que consomem muitos ciclos de CPU. O DOMonitor é semelhante ao Paradyn no que diz respeito à forma transparente como a monitoração é realizada. No entanto, o Paradyn originalmente não suporta aplicações *MultiThreaded* e também não foi projetado para o modelo de Programação de Objetos Distribuídos.

DOMonitor x Ferramentas Comerciais e Outras: OptimizeIt [OPT 2001], JProbe [JPR 2001], Jinsight [JIN 2000] e Visual Quantify [VIS 2001] são ferramentas de análise de desempenho comerciais. Apresentam gráficos bem elaborados com ênfase principalmente na utilização de memória, na utilização do processador e nas chamadas de métodos.

O HProf é um agente de monitoração embutido no pacote J2SE [SUN 2001d]. O HProf não obtém informações detalhadas o suficiente para detectar problemas de desempenho em grandes aplicações. O volume do arquivo de traço gerado pelo HProf é bem superior ao volume gerado pelo DOMonitor. Isso se deve a maior quantidade de eventos de interesse monitorados e pela ausência de filtros.

Estas ferramentas de monitoração para Java não podem ser usadas convenientemente em aplicações distribuídas. Elas fornecem informações cumulativas, sem suporte para aplicações distribuídas (monitoração da comunicação). Embora essas informações sejam úteis para exibir problemas de desempenho de aplicações seqüenciais Java, tem pouca utilização para auxiliar na identificação das origens dos gargalos de desempenho de aplicações distribuídas.

HyperProf e ProfileViewer são ferramentas de visualização que demonstram o comportamento da aplicação de acordo com os dados de monitorações obtidos pelo HProf. Informações contempladas referem-se a chamadas de métodos locais, com ênfase principal para a relação método chamador - método chamado. Nenhuma das duas ferramentas suporta diretamente aplicações *MultiThreaded* nem aplicações do tipo Cliente/Servidor.

A principal diferença entre a monitoração realizada por essas ferramentas e a monitoração do DOMonitor é o enfoque do tipo de problemas a serem destacados. Enquanto essas ferramentas apresentam como ponto forte o gerenciamento da memória, relação entre chamadas de métodos, o DOMonitor preocupa-se com questões relacionadas à concorrência e à distribuição como comportamento das threads, contenção causada devido à sincronização e comunicação (atividades cliente/servidor usando RMI).

DOMonitor x JaViz: JaViz é uma ferramenta de Visualização que suporta a monitoração de aplicações distribuídas [KAZ 2000]. Essa monitoração é realizada utilizando uma versão instrumentada da máquina virtual. Isso restringe a utilização da ferramenta às plataformas para as quais existem versões desta máquina virtual instrumentada. Além disso, a ferramenta não trata uma característica muito importante no contexto de aplicações *multithreaded*, que é a contenção causada devido à utilização de métodos de sincronização (competição por recursos). O DOMonitor difere do JaViz em dois dos aspectos descritos acima: o suporte de monitoração utiliza qualquer versão de máquina virtual e identifica a sincronização entre as *threads*.

O suporte de visualização do JaViz é composto por um gráfico que demonstra o comportamento da aplicação como uma árvore da execução. Não fornece uma visualização global da execução ou do ambiente no qual ela executou. As maiores limitações deste grafo são: o grafo compreende a execução de um programa, sendo este um cliente ou um servidor; não é possível identificar claramente, quanto tempo à aplicação realmente esteve em execução e a relação entre o balanceamento de carga do ambiente de execução.

Existe semelhança entre as opções de filtro utilizadas pelo DOMonitor e pelo JaViz. A diferença principal é que o Javiz coleta informações apenas do nível mais

alto das chamadas à API Java, enquanto o DOMonitor elimina todos os eventos relacionados à API Java.

DOMonitor x DejaVu: DejaVu – é uma ferramenta de repetição determinística [CHO 2000]. Este processo de repetição é baseado em uma monitoração exaustiva do comportamento das *threads*. O usuário não precisa alterar o código fonte da aplicação para utilizar a ferramenta. No entanto, a máquina virtual utilizada pelo DejaVu é uma versão própria de máquina virtual o que limita a utilização da ferramenta às plataformas para as quais exista esta versão modificada.

A monitoração do DOMonitor necessita de menor nível de informações, pois a finalidade dos dados é diferente. Assim como o DOMonitor, o DejaVu distribuído também foi concebido para o modelo de programação de objetos distribuídos java.

DOMonitor x France Telécom: O trabalho desenvolvido por Ottogalli [OTT 2001] é uma ferramenta de análise de desempenho para aplicações distribuídas Java. Este trabalho ainda está em fase de desenvolvimento na France Telécom. Os arquivos de traço são obtidos sem nenhuma modificação na aplicação monitorada ou na JVM. Os registros de traço incluem informações coletadas no nível da aplicação através do JVMPI bem como informações obtidas no nível do sistema operacional. As informações relacionadas à comunicação entre JVM's são obtidas a partir de estruturas de dados do sistema operacional Linux.

A monitoração realizada pelo DOMonitor possuir diversas semelhanças com a monitoração realizada pela ferramenta de Ottogalli. A maior diferença é na forma como é obtida a comunicação entre objetos que executem em JVM's diferentes. Enquanto a ferramenta em desenvolvimento por Ottogalli obtém as informações sobre a comunicação a nível do sistema operacional, o DOMonitor obtém tais informações através de uma extensão do *daemon* EXEd, que é totalmente integrado a API RMI de Java.

A tabela 8.1 apresenta uma comparação entre algumas características dos trabalhos relacionados.

TABELA 8.1 – Comparação entre os trabalhos

Nome	Características Principais
Pablo	- Programas devem ser instrumentados manualmente - Suporte de monitoração próprio - Processamento *Post-Mortem* - Visualização gráfica do comportamento da execução - Baseado em processos - Programas paralelos e distribuídos - Não suporta orientação a objetos
Paradyn	- Instrumentação automatizada e dinâmica - Suporte de monitoração próprio - Processamento *On-Line* - Visualização gráfica do comportamento da execução - Baseado em processos - Programas paralelos e distribuídos - Não suporta orientação a objetos

HProf	- Baseado no JVMPI - Utiliza versão padrão da JVM - Suporte de monitoração próprio - Informações sobre a quantidade e a duração das chamadas de métodos - Processamento *off-line* - Não suporta aplicações distribuídas
OptimizeIT	- Baseado no JVMPI - Utiliza versão padrão da JVM - Suporte de monitoração próprio - Processamento *off-line* e *on-line* - Suporte de visualização próprio - Obém contençao por recursos - Não suporta aplicações distribuídas
JInsight	- Versão modificada da máquina virtual - Suporte de monitoração próprio - Processamento *off-line* - Suporte de visualização próprio - Obém contençao por recursos - Não suporta aplicações distribuídas
JaViz	- Versão modificada da JVM - Suporte de monitoração próprio - Processamento *off-line* - Suporte de visualização próprio - Não obtém contençao por recursos - Suporta aplicações distribuídas
Dist. DejaVu	- Versão modificada da máquina virtual - Suporte de monitoração próprio - Não possui suporte de visualização - Fornece repetição determinística da execução - Suporta aplicações *multithreaded* - Suporta aplicações distribuídas
France Telécom	- Baseado no JVMPI - Utiliza versão padrão da JVM - Suporte de monitoração próprio - Processamento *off-line* - Suporte de visualização através de outra ferramenta - Suporta aplicações distribuídas - Informações relacionadas a comunicação obtidas através do sistema operacional Linux
DOMonitor	- Interface gráfica para parametrizar o suporte de monitoração - Baseado no JVMPI - Utiliza versão padrão da JVM - Suporte de monitoração próprio - Portabilidade - Processamento *off-line* e *on-line* - Suporte de visualização através de outra ferramenta - Suporta aplicações distribuídas - Informações relacionadas a comunicação obtidas através do mecanismo de suporte da API RMI

9 Conclusão

Assim como o paradigma orientado a objetos tradicional foi e ainda é de importância fundamental para os ambientes centralizados, os objetos distribuídos estão se estabelecendo como uma das tecnologias mais importantes para o desenvolvimento de aplicações em rede. O número cada vez mais crescente de aplicações, publicações e projetos nesta área atesta esta tendência e é mais um incentivo a que se continue nesse caminho. Na medida em que o tamanho, as necessidades de reutilização e encapsulamento das aplicações crescem, o desenvolvimento com objetos distribuídos se torna ainda mais útil e necessário.

A linguagem de programação Java apresenta várias características que possibilitam o desenvolvimento de sistemas distribuídos. Características como, portabilidade, suporte à programação *multithreading*, suporte à programação distribuída (incluindo RMI e *garbage collection*), tem levado à utilização de Java em sistemas distribuídos. No entanto, quanto maior a complexidade da aplicação maior será a chance dela apresentar gargalos de desempenho e mais difícil de serem encontrados esses gargalos. Os aspectos não determinísticos inerentes à execução de uma aplicação distribuída aumentam significativamente a complexidade do processo de depuração de programas.

As ferramentas de análise de desempenho procuram auxiliar os programadores no processo de desenvolvimento, depuração e refinamento das suas aplicações. Tendo por referência este contexto, foi concebido DOMonitor, apresentado neste trabalho. O DOMonitor é uma ferramenta de monitoração específica para o modelo de objetos distribuídos Java cujo objetivo principal é obter informações sobre a execução da aplicação.

As características principais tratadas pelo DOMonitor dizem respeito à concorrência e à distribuição, ou seja, o forte da ferramenta é obter a concorrência entre as *threads*, seu comportamento (no que diz respeito a tempo realizando computação útil, tempo bloqueada, tempo comunicando) e quanto à distribuição, ou seja, a interação entre as *threads* (através de invocações de métodos remotos - RMI).

Algumas destas características vêm sendo pouco tratadas por outras ferramentas de análise de desempenho para Java. O foco principal da maioria dessas ferramentas é na ocupação da memória (alocação de objetos) e na utilização da CPU (chamadas de métodos). Além disso, a maioria não permite uma análise mais detalhada da aplicação, deixando de fora o registro de informações sobre o comportamento de cada *thread* e da interação entre elas.

O projeto do DOMonitor foi baseado em uma série de premissas, dentre elas merecem destaque: (i) a organização modular; (ii) monitoração transparente; (iii) portabilidade.

Os **módulos** oferecem as propriedades flexibilidade e expansibilidade, as quais permitem a adição e/ou modificação de funcionalidades. Estas características foram alcançadas através da separação da tarefa de detectar e coletar os eventos da tarefa de analisar e mostrar as informações. Com isto, ferramentas que utilizem as informações sobre a execução não precisam preocupar-se em como elas são coletadas, mas somente com a interpretação e apresentação delas aos usuários. Desta forma, o sistema de monitoramento mantém independência da aplicação que monitora e as informações de saída podem ser utilizadas por outras ferramentas.

A ***Monitoração Transparente*** simplifica a tarefa de programação, pois uma vez que o suporte de monitoração é implementado na camada de suporte do ambiente de execução, não exige o envolvimento do programador e nem a necessidade de modificações no código fonte da aplicação. Por questões de flexibilidade, o ambiente possui uma interface que permite ao programador especificar *"o que monitorar"* (ao invés de *"como monitorar")*. Desta forma, o programador especifica somente os eventos de interesse, deixando que o DOMonitor gerencie a monitoração destes eventos.

O DOMonitor foi projetado para ser completamente **independente da implementação da máquina virtual Java**, ou seja, funcionará em conjunto com qualquer máquina virtual que implemente as mesmas funcionalidades que a máquina virtual padrão.

Outra decisão do DOMonitor foi a de não registrar todos os eventos gerados durante a execução da aplicação. Um registro total possui diversas desvantagens: (i) maior intrusão, pois para todos os eventos será gerado o registro de algumas informações; (ii) maior quantidade de memória exigida; (iii) maior quantidade de I/O; (iv) o resultado final (arquivo de traço) com muitas informações e pouca objetividade.

Uma vez que o DOMonitor é voltado para aplicações compostas por objetos distribuídos, os eventos de interesse foram cuidadosamente selecionados para caracterizar principalmente: (i) o comportamento dinâmico das *threads*; (ii) a utilização dos métodos de sincronização; e (iii) a comunicação entre os entes distribuídos da aplicação.

As informações coletadas durante a execução da aplicação são disponibilizadas em um arquivo de saída contendo o traço de eventos. Este traço é um conjunto ordenado de todos os eventos de interesse que ocorreram durante a execução da aplicação. Estes eventos são armazenados em um formato bem-definido para tornar o processo de monitoração independente das ferramentas que utilizam as informações geradas.

Os dados produzidos pelo sistema podem ser utilizados com diversas finalidades tais como visualização da execução, escalonamento e como suporte à execução de aplicações móveis. Para comprovar esta versatilidade, foi proposta a integração do sistema a outros projetos: Pajé e ISAM

Futuros estudos terão por alvo caracterizar resultados de uso do DOMonitor nos contextos do projeto ISAM, o qual utilizará os dados monitorados para tomadas de decisão durante o curso da execução e do projeto Pajé para permitir a visualização gráfica das características dinâmicas de uma aplicação Java.

Dentre outros trabalhos futuros, cabe destacar:

- A realização de diversas medidas relacionadas à interferência do DOMonitor em cada uma das classes de monitoração apresentadas. Estas medidas podem auxiliar também na identificação de características de implementação que poderão ser melhor exploradas para tornar o agente de monitoração o menos intrusivo possível;
- A implementação do algoritmo de adequação de relógio proposto por Maillet [MAI 95] para obter o comportamento global da execução e tornar possível a visualização gráfica das informações da execução utilizando o Pajé;

- A integração total dos dados de monitoração com o Pajé alcançam outro objetivo importante que é o de possibilitar que aplicações desenvolvidas utilizando o DOBuilder possam ter sua execução visualizada;
- O desenvolvimento de novas funcionalidades de acordo com as necessidades que surgirão durante a implementação do ambiente de execução do ISAM;
- A extensão do DOMonitor para suportar comunicações realizadas com outros *middlewares* de comunicação que não Java RMI, tais como CORBA e Voyager.

Referências Bibliográficas

[ARA 2001] ARAUJO, E. B.; AUGUSTIN, I.; YAMIN, A.; SILVA, L.; GEYER, C. Uma Proposta de Monitoração para Visualização de Aplicações Distribuídas Java. In: JORNADAS CHILENAS DE COMPUTACIÓN, 2001, Punta Arenas. **Anales**. Punta Arenas: Universidad de Magallanes, 2001.

[AUG 2002] AUGUSTIN, I.; YAMIN, A.; Barbosa, J.; GEYER, C. Towards a Taxonomy for Mobile Applications with Adaptive Behavior. In: INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED COMPUTING AND NETWORKS, Innsbruck, Austria. **Proceedings...** Innsbruck: Iasted Press, 2002,

[AUG 2001a] AUGUSTIN, I.; YAMIN, A.; BARBOSA, J.; GEYER, C. Requisitos para o Projeto de Aplicações Móveis Distribuídas. In: CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN, 7., 2001, El Calafate, Santa Cruz, Argentina. **Anales**. [S.l.:s.n.], 2001.

[AUG 2001b] AUGUSTIN, I.; YAMIN, A.; SILVA JR, E.; BARBOSA, J.; SILVA, L.; GEYER, C. Explorando Adaptação como um Framework de Projeto para Sistemas Móveis Distribuídos. In: WORKSHOP DE COMUNICAÇÃO SEM FIO E COMPUTAÇÃO MÓVEL, 3., Recife, Brasil. **Anais...** [S.l: s.n.], 2001.

[AUG 2001c] AUGUSTIN, I. **O Acesso aos Dados no Contexto da Computação Móvel**. 2001. 148p. Exame de Qualificação (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[AYD 97] AYDT, R. **The Pablo Self-Defining Data Format**. Urbana, IL, USA: University of Illinois at Urbana-Champaign, Department of Computer Science, 1997. Disponível em: <http://wotug.kent.ac.uk/parallel/performance/tools/pablo>. Acesso em: nov. 2000.

[AZE 2001] AZEVEDO, S. C. **DEPAnalyzer: um Modelo de Análise Estática de Dependências para Programas Orientados a Objeto.** 2001. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[AZE 99] AZEVEDO, S. C.; BARBOSA, J.; GEYER, C. Automatização da Análise Global no modelo GRANLOG. In: CONFERENCIA LATINO AMERICANA DE INFORMÁTICA, 25., 1999, Asuncion, Paraguay. **Memórias.** Asuncion: Universidad Autonoma de Asunción, 1999. v. 1, p. 601-612.

[BAR 2001a] BARBOSA, J.; YAMIN, A.; VARGAS, P.; FERRARI, D.; SCHAEFFER, E.; GEYER, C. Using Mobility and Blackboards to Support a Multiparadigm Model Oriented to Distributed Processing. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 13., 2001. **Proceedings…** Brasília: UNB, 2001. p. 187-194.

[BAR 2001b] BARBOSA, J.; GEYER, C. F. R. Integrating Logic Blackboards and Multiple Paradigm for Distributed Software Development. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS, 2001, Las Vegas. **Proceedings...** Las Vegas: CSREA, 2001. v. 2, p.808-814.

[BAR 2001c] BARBOSA, Jorge L. V.; GEYER, C. F. R. Uma Linguagem Multiparadigma Orientada ao Desenvolvimento de Software Distribuído. In: SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO, 5., 2001, Curitiba. **Anais...**. [S.l: s.n.], 2001.

[BAR 2001d] BARBOSA, J. et al. HoloJava: Translating a Distributed Multiparadigm Language into Java. In: CONFERENCIA LATINOAMERICANA DE INFORMÁTICA, CLEI, 27., 2001, Mèrida. **[Artículos]**. Mèrida: Universidad de los Andes, 2001.

[BAT 95] BATES, P. C. Debugging heterogeneous distributed systems using event-based models of behavior. **ACM Transactions on Computer Systems**, New York, v. 13, n. 1, p. 1-31, 1995.

[BRI 97] BRIAT, J. et al. Athapascan runtime: efficiency for irregular problems. In: INTERNATIONAL EURO-PAR CONFERENCE ON PARALLEL PROCESSING, EURO-PAR, 3., 1997, Passan, Gx. **Parallel processing:** proceedings. Berlin: Springer-Verlag, c 1997. p. 591-600. (Lecture Notes in Computer Science, v. 1300).

[CAV 98] CAVALHEIRO, G. G. H.; DENNEULIN, Y.; ROCH, J.-L. A General Modular Specification for Distributed Schedulers. In: INTERNATIONAL EURO-PAR CONFERENCE ON PARALLEL PROCESSING, EURO-PAR, 1998, Southampton. **Parallel processing:** proceedings. Berlin: Springer-Verlag, 1998.

[CHO 2000] CHOI, J.-D. et al. Deterministic Replay of Distributed Java Applications. In: INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, IPDPS, 14., 2000, Cancun, Mx. **Proceedings…** Los Alamitos: IEEE Computer Society, 2000.

[CHO 98] CHOI, J.-D.; SRINIVASAN, H. Deterministic replay of java multithreaded applications. In: ACM SIGMETRICS SYMPOSIUM ON PARALLEL AND DISTRIBUTED TOOLS, 1998. **Proceedings…** [S.l.:s.n.], 1998. p. 48-59.

[DEI 2001] DEITEL, P. J.; DEITEL, H. M. **Java how to program**. 4th ed. [S.l.]: Prentice Hall, 2001.

[DER 99] DEROSE, L.; REED, D. A. SvPablo: A Multi-Language Architecture-Independent Performance Analysis System. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, ICPP, 1999, Fukushima, Japan. **Proceedings…** [S.l.: s.n.], 1999.

[DIN 90] DINNING, A.; SCHONBERG, E. An Empirical comparison of monitoring algorithms for access anomaly detection. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2., 1990, Seattle, Washington. **Proceedings…** New York: ACM Press, 1990. p. 1-10.

[DUM 98] DUMANT, B. et al. Jonathan: an open distributed processing environment in java. In: MIDDLEWARE INTERNATIONAL CONFERENCE ON DISTRIBUTED SYSTEMS PLATFORMS AND OPEN DISTRIBUTED PROCESSING, 1998. **Proceedings…** [S.l: s.n.], 1998.

[FER 2001] FERRARI, D. N. **Um Modelo de Replicação em Ambientes que Suportam Mobilidade.** 2001. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[FID 91] FIDGE, C.J. Logical Time in Distributed Computing Systems. **IEEE Computer**, Los Alamitos, v.24, n.8, Aug. 1991.

[FOW 90] FOWLER, J.; ZWAENEPOEL, W. Causal Distributed Breakpoints. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, ICDCS, 10., 1990. **Proceedings…** [S.l.: s.n.], 1990.

[FOS 96] FOSTER, I. et al. The nexus approach to integrating multithreading and communications. **Journal of Parallel and Distributed Computing**, [S.l.], v. 37, n. 1, p. 70-82, Aug. 1996.

[GLA 99] GLASS, G. **Overview of Voyager:** ObjectSpace's Product Family for State-of-the-Art Distributed Computing. ObjectSpace, 1999. Disponível em: <http://www.objectspace.com/.../VoyagerOverview.pdf>. Acesso em: set. 2000.

[HAR 97] HAROLD, E. R. **Java Network Programming**. Sebastopol: O'Reilly, 1997.

[HYP 2001] HYPERPROF (v. 1.3) – Java Profile Browser. Disponível em: <http://www.physics.orst.edu/~bulatov/HyperProf>. Acesso em: mar. 2001.

[JAL 94] JALOTE, Pankaj. **Fault Tolerance in Distributed Systems**. New Jersey: Prentice-Hall, 1994.

[JI 98] JI, M. et al. Performance Measurements for Multithreaded Programs. **ACM SIGMETRICS Performance Evaluation Review**, New York, v. 26, n.1, p.161-170, June 1998.

[JIN 2000] JINSIGHT. Disponível em: <http://www.alphaworks.ibm.com/formula/ jinsight>. Acesso em: nov. 2000.

[JPR 2001] JPROBE Profiler. Disponível em: <http://www.in-gmbh.de/english/tools/ java/jprobe.htm>. Acesso em: jan. 2001.

[JUR 2000] JURIC, M. B. et al. Java 2 Distributed Object Middleware Performance Analysis and Optimizations. **ACM SIGPLAN Notices on Programming Languages**, New York, v. 35, n. 8, p. 31-40, Aug. 2000.

[KAZ 2000] KAZI, I. H. et al. JaViz: A client/server Java profiling tool. **IBM System Journal**, New York, v. 39, n. 1, p. 96-117, Mar. 2000.

[KER 2000a] KERGOMMEAUX, J. C.; STEIN, B. de O. Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. **Parallel Computing**, Netherlands, v. 26, n. 10, p. 1253-1274, Aug. 2000.

[KER 2000b] KERGOMMEAUX, J. C.; STEIN, B. de O. Pajé, an Extensible Environment for Visualizing Multi-Threaded Programs Execution. In: INTERNATIONAL EURO-PAR CONFERENCE ON PARALLEL PROCESSING, EURO-PAR, 6., 2000. **Parallel processing:** proceedings. Berlin: Springer-Verlag, c 2000. p. 133-140.

[KLE 99] KLEMM, R. Practical Guideline for boosting java server performance**.** In: ACM JAVA GRANDE CONFERENCE, St. Francisco, USA. **Proceedings…** [S.l.: s.n.], 1999.

[LAM 78] LAMPORT, L. Time, Clocks and the Ordering of Events in a Distributed Systems. **Communications of the ACM**, New York, v. 21, p. 558-565, July 1978.

[LEB 87] LEBLANC, T. J.; MELLOR-CRUMMY, J. M. Debugging parallel programs with instant replay. **IEEE Transactions on Computers**, Los Alamitos, v. 36, n.4, p. 471-481, Apr. 1987.

[LEM 2001] LEMAY, L.; PERKINS, C. **Aprenda em 21 dias Java 2**. Rio de Janeiro: Campus, 2001.

[MAH 99] MAHESWARAN, M. Quality of Service Driven Resource Management Algorithms for Network Computing. In: THE INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS, PDPTA, 1999. **Proceedings…** [S.l: s.n.], 1999.

[MAI 95] MAILLET, É.; TRON, C. On Efficiently Implementing Global Time for Performance Evaluation on Multiprocessor Systems. **Journal of Parallel and Distributed Computing**, [S.l.], v. 28, p. 84-93, July 1995.

[MAL 2001] MALACARNE, J. **Ambiente Visual para Programação Distribuída em Java**. 2001. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[MAL 97] MALACARNE, J. **Implementação de um Ambiente Gráfico para o Desenvolvimento de Aplicações Distribuídas**. 1997, 91p. Projeto de Diplomação (Bacharelado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[MAL 92] MALONY, A. D. et al. Performance measurement intrusion and perturbation analysis. **IEEE Transactions on Parallel and Distributed Systems**, Los Alamitos, v. 3, n.4, p. 433-450, 1992.

[MEN 94] MENASCÉ, D. A. et al. **Capacity Planning and Performance Modeling**. New Jersey: Prentice-Hall., 1994.

[MIC 2000] MICROSOFT CORPORATION. **DCOM**. Disponível em: <http://www.microsoft.com/com/tech/dcom.asp>. Acesso em: ago. 2000.

[MIL 95] MILLER, B. P. et al. The Paradyn parallel performance measurement tool. **IEEE Computer,** Los Alamitos, v. 28, n. 11, p. 37-46, Nov. 1995.

[NIE 97] NIEMEYER, P.; PECK, J. **Exploring Java**. 2nd ed. Sebastopol: O'Reilly, 1997.

[OMG 99] OMG - Object Management Group. **CORBA - Common Object Request Broker Architecture.** Disponível em: <http://www.omg.org/>. Acesso em: jul. 1999.

[OPT 2001] OPTIMIZEIT! The Ultimate Java Performance Profiler. Disponível em: <http://www.optimizeit.com>. Acesso em: mar. 2001.

[OTT 2001] OTTOGALLI, F. G. et al. Visualization of distributed applications for performance debugging. In: ICCS TOOLS AND ENVIRONMENTS FOR PARALLEL AND DISTRIBUTED PROGRAMMING, 2001, San Francisco, USA. **Proceedings…** [S.l: s.n.], 2001.

[PAB 2000] PABLO Research Group. Disponível em: <http://www-pablo.cs.uiuc.edu>. Acesso em: abr. 2001.

[PRO 2001] PROFILEVIEWER. Disponível em: <http://www.inetmi.com/~gwhi/ProfileViewer/ProfileViewer.html>. Acesso em: maio 2001.

[REE 98] REED, D. A et al. Performance Analysis of Parallel Systems: Approaches and Open Problems. In: JOINT SYMPOSIUM ON PARALLEL PROCESSING, JSPP, 1998. **Proceedings…** [S.l: s.n.], 1998. p. 239-256.

[REE 93] REED, D. A. et al. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In: SCALABLE PARALLEL LIBRARIES CONFERENCE, 1993. **Proceedings…** Los Alamitos: IEEE Computer Society Press, 1993. p. 104-113.

[RUS 96] RUSSINOVICH, M.; COGSWELL, B. Replay for concurrent non-deterministic shared-memory applications. In: ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGES AND IMPLEMENTATION, PLDI, 1996. **Proceedings…** [S.l: s.n.], 1996. p. 258-266.

[SAT 96] SATYANARAYANAN, M. Fundamental Challenges in Mobile Computing. In: ACM SYMPOSIUM. ON PRINCIPLES OF DISTRIBUTED COMPUTING, 15., 1996. **Proceedings…** [S.l: s.n.], 1996.

[SIL 2001] SILVA, L.; YAMIN, A.; AUGUSTIN, I.; ARAUJO, E.; GEYER, C. Mecanismos de Suporte ao Escalonamento em Sistemas com Objetos Distribuídos Java. In: CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN, 7., 2001, El Calafate, Santa Cruz, Argentina. **Anales**. [S.l: s.n.], 2001.

[STE 2001a] STEIN, B. de O. **Pajé trace file format**. Grenoble: Université Joseph Rourier, 2001. Technical Report.

[STE 2001b] STEIN, B. de O. Depuração de Programas Paralelos. In: ESCOLA REGIONAL DE ALTO DESEMPENHO, ERAD, 1., 2001, Gramado. **Anais...** Porto Alegre: SBC, 2001.

[STE 99] STEIN, B. de O. **Visualization interactive et extensible de programmes paraléles à base de processus légers.** 1999. PhD Thesis – Université Joseph Rourier, Grenoble. Disponível em: <http://www-mediatheque.imag.fr>. Acesso em: mar. 2001.

[SUN 2001a] SUN MICROSYSTEMS. **Documentação da JVMPI – Java Virtual Machine Profiler Interface**. Disponível em: <http://www.javasoft.com/products/jdk/1.3/docs/guide/jvmpi/jvmpi.html>. Acesso em: out. 2001.

[SUN 2001b] SUN MICROSYSTEMS. **Documentação da linguagem Java**. Disponível em: <http://java.sun.com/products/jdk/1.3/download-docs.html>. Acesso em: jul. 2001.

[SUN 2001c] SUN MICROSYSTEMS. **Documentação de RMI**. Disponível em: <http://java.sun.com/docs/books/tutorial/rmi/index.html>. Acesso em: mar. 2001.

[SUN 2001d] SUN MICROSYSTEMS. **Java 2 SDK Standard Edition v. 1.3**. Disponível em: <http://java.sun.com/products/jdk/1.3>. Acesso em: jan. 2001.

[TSA 95] TSAI, J.; YANG, S. **Monitoring and Debugging of Distributed Real-Time Systems**. Los Alamitos: IEEE Computer Society Press, 1995.

[VAH 99] VAHDAT, A. Toward Wide-Area Resource Allocation. In: THE INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS, PDPTA, 1999. **Proceedings…** [S.l: s.n.], 1999. p. 930-936.

[VEN 2000] VENNERS, Bill. **Inside the Java 2 Virtual Machine**. 2nd ed. [S.l.]: McGraw-Hill, 2000.

[VIS 2001] VISUAL QUANTIFY. Disponível em: <http://sys-com.com/java/reviews/quantify/index.html>. Acesso em: jan. 2001.

[VIS 2000] VISWANATHAN, D.; LIANG, S. Java Virtual Machine Profiler Interface. **IBM System Journal**, New York, v. 39, n. 1, p. 82-95, Mar. 2000.

[XU 99] XU, Zhichen et al. Dynamic Instrumentation of Threaded Applications. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 1999. **Proceedings…** [S.l: s.n.], 1999.

[YAM 2001a] YAMIN, A.; AUGUSTIN, I.; BARBOSA, J.. SILVA, L.; GEYER, C. Explorando o Escalonamento no Desempenho de Aplicações Móveis Distribuídas. In: WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, WSCAD, 2., 2001. **Anais...**. Brasília: UNB, 2001.

[YAM 2001b] YAMIN, A. Escalonamento em Sistemas Paralelos e Distribuídos. In: ESCOLA REGIONAL DE ALTO DESEMPENHO, ERAD, 1., 2001, Gramado. **Anais...** Porto Alegre: SBC, 2001.

[YAM 99] YAMIN, A. C. **An Execution Environment for Multiparadim Models**. 1999. PHD Proposal (PPGC) – Instituto de Informática, UFRGS, Porto Alegre.

[ZHA 95] ZHAO, Q.; STASKO, J. **Visualizing the Execution of Threads-based Parallel Programs**. Atlanta, USA: Georgia Institute of Technology, 1995. (Technical Report GIT-GVU-95-01)

[ZIE 95] ZIELINSKI, K.; LAURENTOWSKI, A. A Tool for Monitoring Software-Heterogeneous Distributed Object Applications. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, ICDCS, 15., Vancouver, Canada, 1995. **Proceedings…** Los Alamitos: IEEE Computer Society Press, 1995.