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

TELMO BRUGNARA

Nprof: uma ferramenta para monitoramento de aplicações distribuídas

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Prof ª. Dr ª. Ingrid Jansch-Pôrto
Orientadora

Prof ª. Dr ª. Maria Lúcia Blanck Lisbôa
Co-orientadora

Porto Alegre, setembro de 2006.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Brugnara, Telmo

Nprof: uma ferramenta para monitoramento de aplicações distribuídas / Telmo Brugnara – Porto Alegre: PPGC da UFRGS, 2006.

100 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2006. Orientadora: Ingrid Jansch-Pôrto; Co-orientadora: Maria Lúcia Blanck Lisbôa.

1. Monitoramento de sistemas. 2. Avaliação de desempenho. 3 Depuração. 4. Linguagem Java. 5. Aplicações distribuídas. I. Jansch-Pôrto, Ingrid. II. Lisbôa, Maria Lúcia Blanck. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. José Carlos Ferraz Hennemann
Vice-Reitor: Prof. Pedro Cezar Dutra Fonseca
Pró-Reitora de Pós-Graduação: Profa. Valquiria Linck Bassani
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do PPGC: Prof. Carlos Alberto Heuser
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Agradeço, primeiramente, a meus pais, que me deram total apoio a esta e todas as demais caminhadas necessárias à realização deste projeto.

Às minhas irmãs, Laura e Lúcia, fontes de inspiração para esta jornada.

À profª. Maria Lúcia Blanck Lisbôa, co-orientadora, sempre pronta a auxiliar-me nas questões mais diversas e mesmo àquelas que escapam da área própria da orientação. Por todos esses anos de orientação, desde o trabalho de conclusão da graduação ao mestrado.

À profª. Ingrid Jansch-Pôrto, pela orientação deste trabalho de mestrado e todo auxílio, muito bem-vindo.

À Luciana Druzina, por todo apoio e confiança depositados.

Aos colegas da Procergs, por todo apoio e incentivo recebido, além de discussões sobre o assunto que auxiliaram na definição do tema pretendido.

Por fim, a todos amigos e familiares que sempre acreditaram e me apoiaram neste projeto.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 7

LISTA DE FIGURAS 8

LISTA DE TABELAS 9

RESUMO 10

ABSTRACT 11

1 INTRODUÇÃO 13

1.1 Apresentação 13
1.2 Monitores e seus componentes 13
1.2.1 Classificação de monitores 14
1.2.2 Componentes de monitores 14
1.3 *Profiling* 15
1.4 A plataforma Java 16
1.5 Monitores Java 16
1.6 Motivação e objetivos 17
1.7 Organização do trabalho 17

2 MEDIDAS DE DESEMPENHO DE APLICAÇÕES 19

2.1 Apresentação 19
2.2 Desempenho 19
2.2.1 Desempenho computacional 19
2.2.2 Uso de memória RAM 20
2.2.3 Tempo de inicialização 20
2.2.4 Escalabilidade 20
2.2.5 Desempenho percebido 20
2.3 Tipos de medidas 20
2.4 Medidas para aplicações distribuídas 21
2.5 Relacionamentos em aplicações distribuídas 21
2.6 Algumas considerações 22

3 TECNOLOGIAS DE CONSTRUÇÃO DE FERRAMENTAS PARA MONITORAMENTO 23

3.1 Tecnologias utilizadas para Java 23
3.1.1 JVMPI 23

3.1.2 JPDA ... 25
3.1.3 Alteração na JVM ... 29
3.1.4 Instrumentação de *bytecode* e código-fonte ... 29
3.1.5 Programação Orientada a Aspectos ... 29
3.1.6 JMX ... 30
3.2 Considerações ... 30

4 FERRAMENTAS DE DEPURAÇÃO, PROFILING E VISUALIZAÇÃO DE PROGRAMAS ... 33

4.1 Apresentação ... 33
4.2 Ferramentas de depuração e *profiling* ... 33
4.2.1 OptimizeIt ... 33
4.2.2 JProfiler ... 33
4.2.3 JFluid ... 34
4.2.4 AppPerfect ... 34
4.2.5 Hprof ... 34
4.2.6 DynaMetricO ... 34
4.2.7 Javiz ... 35
4.2.8 Aspect Oriented Performance Analysis Environment – AOP Profiler ... 35
4.2.9 JConsole ... 35
4.2.10 VisualGC e Jstat ... 35
4.2.11 Atlassian Profiling ... 35
4.2.12 CLRProfiler ... 36
4.2.13 VTune ... 36
4.2.14 Gprof ... 36
4.2.15 Hy+ ... 36
4.2.16 VisuSniff ... 36
4.2.17 Netgroup: Analyzer ... 37
4.2.18 JRastro ... 37
4.2.19 DCPI - DIGITAL Continuous Profiling Infrastructure ... 37
4.2.20 TAU Profiler ... 37
4.3 Ferramentas de visualização de programas ... 37
4.3.1 Jinsight ... 38
4.3.2 Javavis ... 38
4.3.3 JaVis ... 38
4.3.4 Jacot ... 38
4.4 Tabela de ferramentas ... 39
4.5 Contribuições para Nprof ... 40

5 A FERRAMENTA NPROF ... 43

5.1 Apresentação ... 43
5.2 Arquitetura da ferramenta Nprof ... 43
5.2.1 Módulo central ... 44
5.2.2 Módulo de controle de alvo ... 46
5.2.3 Módulo remoto ... 46
5.2.4 Monitores distribuídos ... 47
5.3 Monitores especializados ... 48
5.3.1 Monitores baseados em JDI ... 48
5.3.2 Monitores baseados em JMX (Java Management) ... 49
5.3.3 Monitores baseados em instrumentação ... 49

5.3.4 Tabela de Monitores ... 49
5.4 Instrumentação de *bytecode* ... 50
5.5 Sincronização de relógios ... 51
5.6 Interface gráfica ... 52
5.6.1 Visualizador de *Threads* ... 54
5.6.2 Visualizador de Memória ... 56
5.6.3 Visualizador de Objetos ... 56
5.6.4 Visualizador de Ambiente de Execução ... 57
5.6.5 Visualizador de Carga de Classes ... 58
5.6.6 Visualizador de Exceções ... 59
5.6.7 Visualizador de *Sockets* TCP ... 60
5.6.8 Visualizador de *Sockets* UDP ... 61
5.6.9 Visualizador de objetos em memória ... 61
5.6.10 Visualizador de grafo de sockets ... 62
5.6.11 Arquivos de traços para mensagens socket ... 63
5.6.12 Arquivo de traços para compilação e coleta de lixo ... 63
5.7 Modelo de extensão de Nprof ... 64

6 ESTUDOS DE CASO ... 67

6.1 Introdução ... 67
6.2 Monitoramento da aplicação Tomcat ... 67
6.2.1 Testes ... 68
6.2.2 Considerações sobre o estudo de caso ... 70
6.3 Avaliação de desempenho: aplicação RMI ... 70
6.3.1 Organização ... 71
6.3.2 Definição e metodologia dos testes ... 71
6.3.3 Testes de *threads* ... 72
6.3.4 Testes de carga de classes ... 74
6.3.5 Testes de instanciação de objetos ... 76
6.3.6 Testes de captura de exceções ... 79
6.3.7 Testes de monitores de memória, *sockets* e ambiente de execução (*runtime*) ... 81
6.3.8 Considerações sobre o estudo de caso ... 82
6.4 Monitoramento de aplicação distribuída: JavaGroups ... 82
6.4.1 Configuração dos testes ... 82
6.4.2 Configurações 1, 2 e 3 ... 84
6.4.3 Configurações 3, 4, 5, 6 e 7 ... 85
6.4.4 Considerações sobre o estudo de caso ... 90

7 CONCLUSÕES ... 91

7.1 Trabalhos Futuros ... 92

REFERÊNCIAS ... 95

LISTA DE ABREVIATURAS E SIGLAS

API	Application Program Interface
BCI	Bytecode Instrumentation
HTTP	HyperText Transfer Protocol
IP	Internet Protocol
JDK	Java Development Kit
JDWP	Java Debug Wire Protocol
JIT	Just-In-Time compilation
JMX	Java Management eXtensions
JPDA	Java Platform Debugger Architecture
JVM	Java Virtual Machine
JVMDI	Java Virtual Machine Debug Interface
JVMPI	Java Virtual Machine Profiling Interface
JVMTI	Java Virtual Machine Tool Interface
JDI	Java Debug Interface
POA	Programação Orientada a Aspectos
RAM	Random Access Memory
RMI	Remote Method Invocation
TCP	Transfer Control Protocol
UCP	Unidade Central de Processamento
UDP	User Datagram Protocol

LISTA DE FIGURAS

Figura 2.1: Comunicação entre componentes de aplicações distribuídas 22
Figura 3.1: Arquitetura da JPDA 26
Figura 5.1: Arquitetura da ferramenta Nprof 44
Figura 5.2: Arquivo de configuração do Nprof 45
Figura 5.3: Diagrama de classes de registro de dados 47
Figura 5.4: Interface gráfica da ferramenta Nprof 53
Figura 5.5: Diálogos de inicialização e de conexão com aplicação-alvo 54
Figura 5.6: Gráfico de estados de threads 55
Figura 5.7: Informações sobre *threads* em forma de tabela 55
Figura 5.8: Visualizador de uso de memória 56
Figura 5.9: Visualizador de objetos 57
Figura 5.10: Visualizador de informações de ambiente de execução 58
Figura 5.11: Visualizador de carga de classe 59
Figura 5.12: Visualizador de exceções 60
Figura 5.13: Visualizador de *sockets*: *bytes* recebidos 60
Figura 5.14: Visualizador de s*ockets* por meio de tabela 61
Figura 5.15: Visualizador de *sockets* UDP por meio de tabela 61
Figura 5.16: Visualizador de objetos em memória 62
Figura 5.17: Visualizador de grafo de *sockets* 63
Figura 5.18: Interface Monitor 64
Figura 5.19: Interface MessageListener 65
Figura 6.1: Estado de *threads* do servidor Tomcat 68
Figura 6.2: Uso de memória pelo servidor Tomcat 69
Figura 6.3: Tráfego de mensagens entre Tomcat e HTTPClient 69
Figura 6.4: Interação entre os componentes da aplicação RMI 71
Figura 6.5: Estrutura da configuração dos testes da aplicação RMI 71
Figura 6.6: Sobrecargas no uso de Nprof e do monitor de *threads* 74
Figura 6.7: Teste de sobrecarga utilizando carga de classes 76
Figura 6.8: Sobrecarga comparada do Nprof e do monitor de objetos 78
Figura 6.9: Sobrecarga relativa quanto ao monitor de objetos 78
Figura 6.10: Sobrecarga relativa à captura de exceções 80
Figura 6.11: Sobrecarga relativa do Nprof a partir de captura de exceções 80
Figura 6.12: Comparativo de sobrecarga entre monitores *socket*, memória e *runtime* .. 81
Figura 6.13: *Sockets* UDP de um nodo da aplicação JavaGroups 88
Figura 6.14: *Sockets* de aplicação-alvo com pilha de protocolos `GMS` 90

LISTA DE TABELAS

Tabela 3.1: Tipos de eventos JVMPI 25
Tabela 3.2: Eventos da JVMDI 27
Tabela 4.1: Ferramentas analisadas 39
Tabela 5.1: Monitores implementados 50
Tabela 5.2: Classes instrumentadas pelo monitor de *sockets* 51
Tabela 6.1: Tempos de *thread* e intervalos de confiança 73
Tabela 6.2: Sobrecarga relativa referente ao uso de *threads* 73
Tabela 6.3: Tempos de carga de classe e intervalos de confiança 75
Tabela 6.4: Sobrecarga relativa referente à carga de classes 75
Tabela 6.5: Tempos de instanciação de objetos e intervalos de confiança 76
Tabela 6.6: Sobrecarga relativa referente à instanciação de objetos 77
Tabela 6.7: Tempos de captura de exceções e intervalos de confiança 79
Tabela 6.8: Sobrecarga relativa referente à captura de exceções 79
Tabela 6.9: Tempos de execução dos monitores e sobrecarga relativa 81
Tabela 6.10: Configurações dos testes com JavaGroups 83
Tabela 6.11: Resultado da execução das configurações 1,2 e 3 85
Tabela 6.12: Execução das configurações 3 a 7 (1) 86
Tabela 6.13: Execução das configurações 3 a 7 (2) 87
Tabela 6.14: Tamanho médio dos pacotes e mensagens 88
Tabela 6.15: Agrupamento das execuções com e sem retransmissão (1) 89
Tabela 6.16: Agrupamento das execuções com e sem retransmissão (2) 89

RESUMO

A crescente complexidade dos programas de computador e o crescimento da carga de trabalho a qual eles são submetidos têm sido tendências recorrentes nos sistemas computacionais, em especial para sistemas distribuídos como aplicações *web* e sistemas corporativos. O aumento da carga de trabalho gera uma demanda por sistemas que façam melhor uso dos recursos computacionais disponíveis, enquanto a maior complexidade gera uma demanda por sistemas que se preocupem em minimizar o número de erros. Portanto, podem-se identificar dois objetivos a serem perseguidos pelos desenvolvedores de sistemas de *software*: melhorar o desempenho e aumentar a confiabilidade dos sistemas.

A fim de alcançar os objetivos expostos, são desenvolvidos sistemas de monitoramento para automatizar a coleta e análise de dados sobre os sistemas computacionais alvo.

O presente trabalho visa contribuir nos seguintes aspectos: na identificação dos dados relevantes para o monitoramento de aplicações distribuídas desenvolvidas para a plataforma Java; e na criação de uma ferramenta de monitoramento de aplicações distribuídas, explorando os novos recursos do JDK 1.5, bem como os recursos já disponíveis em Java, como carga dinâmica de classes e transformação de *bytecodes*.

A fim de avaliar a ferramenta proposta foram elaborados três estudos de caso: um utiliza uma aplicação existente sem necessidade de sua adaptação; outro avalia a sobrecarga da ferramenta frente a diferentes parâmetros; e o terceiro avalia o monitoramento de um sistema distribuído.

Entende-se que a ferramenta atinge o objetivo de monitoramento de aplicações distribuídas, por meio da incorporação de técnicas e APIs distintas, ao permitir: o monitoramento de uma aplicação distribuída por meio do monitoramento de diversos nodos de tal aplicação concomitantemente; e a visualização das informações coletadas de forma *online*. Adicionalmente, a coleta simultânea de dados de diferentes nodos de uma aplicação distribuída pode ser útil para a descoberta de relações entre eventos que ocorrem durante a execução de tal aplicação.

Palavras-Chave: monitor, *profiler*, desempenho, depuração, Java, computação distribuída, mensagens, *socket*.

Nprof: a monitoring tool for distributed applications

ABSTRACT

The growing complexity of software and the increasing workload to which systems have been submitted are known trends in the computing system field, especially when distributed and web systems are considered. The increasing workload generates demand for systems that can make a better use of computing resources, while the increment of system complexity demands specific actions to prevent design faults. Therefore, software engineers have two main objectives to be concerned with: optimization and dependability.

In order to accomplish these objectives, monitoring systems have been proposed to gather data from running systems so that their behavior can be analyzed. The present dissertation intends to contribute in the following domains: identifying relevant metrics for monitoring distributed Java applications; and developing a tool to monitor and profile distributed applications, using the new resources available in JDK 1 .5 as well as some already known techniques like dynamic classloading and bytecode instrumentation.

In order to evaluate the proposed tool, three test cases have been developed: one with a well known application running without modification; another for evaluating the tools' overhead in different scenarios; and a third one to evaluate a distributed application been monitored.

We understand that the proposed tool is successful in monitoring distributed applications by the use of distinct APIs and techniques because: Nprof can monitor a distributed application by monitoring different nodes of the application simultaneously; and Nprof allows the online visualization of the collected data. Also, simultaneous collection of data from different nodes of a distributed application can be useful for discovering relations among events that occur during the execution of the application.

Keywords: monitor, profiler, performance, debugging, Java, distributed computing, messages, socket.

1 INTRODUÇÃO

Este capítulo introduz a base conceitual e tecnológica pertinentes ao tema deste trabalho e apresenta suas motivações e objetivos.

1.1 Apresentação

A crescente complexidade dos programas de computador e o aumento da carga de trabalho à qual eles são submetidos têm sido uma tendência dos sistemas computacionais, em especial para sistemas distribuídos como aplicações *web* e sistemas corporativos. O aumento da carga de trabalho gera demanda por sistemas que façam o melhor uso possível dos recursos computacionais disponíveis, enquanto a maior complexidade gera uma demanda por sistemas que se preocupem em reduzir o número de erros. Portanto, dois objetivos devem ser perseguidos pelos desenvolvedores de sistemas de *software*: melhorar o desempenho e aumentar a confiabilidade dos sistemas.

Para atingir o primeiro objetivo devem-se realizar coletas de dados sobre o comportamento operacional do sistema, tais como consumo de memória e observação das atividades de *threads*. O segundo objetivo pode ser abordado a partir de observações sobre o comportamento funcional do sistema, como o rastreamento de chamadas de funções, incluindo exceções disparadas, capturadas ou não, durante o processo de execução. Com base nessas observações, caso sejam detectados problemas de desempenho ou detectados erros ou defeitos de execução, podem ser tomadas ações para solucionar tais problemas. Por erro, entende-se a ocorrência de um estado interno do produto que diverge do estado esperado; por defeito, entende-se uma divergência entre o produto desenvolvido e produto supostamente correto. Portanto, a coleta de dados sobre o comportamento de um sistema no decorrer de sua execução pode fornecer dados importantes para a tomada de ações que visem a melhoria de desempenho e o aumento da confiabilidade. Ferramentas conhecidas como monitores, que observam e registram as atividades de um sistema, são usadas há longo tempo para essa finalidade.

1.2 Monitores e seus componentes

Monitores para sistemas computacionais podem servir para diversos fins, dentre os quais podem-se elencar a depuração de sistemas, a avaliação de desempenho e a visualização ou acompanhamento de execução de sistemas computacionais. Sendo assim, monitores são considerados de grande utilidade para desenvolvedores de sistemas, analistas de desempenho ou administradores de sistema.

Um tipo específico de monitor de sistemas computacionais que, além de coletar dados, também se incumbe de traçar um perfil da aplicação analisada a partir dos dados capturados, é conhecido como *profiler* (GRAHAM, 1982). Inicialmente, os *profilers* tinham por objetivo capturar e sumarizar dados sobre a execução das rotinas de um programa: contagem de chamadas a rotinas, tempo de execução de rotinas e grafo de chamada entre rotinas. Atualmente, *profilers* englobam facilidades para análise de outros aspectos de um programa, como o uso de memória e *threads*.

1.2.1 Classificação de monitores

Monitores podem ser classificados a partir de diferentes características, como tipo de implementação, tipo de mecanismo de captura e possibilidades de visualização (JAIN 1991).

Como tipo de implementação, podem-se ter monitores em *hardware*, em *software*, em *firmware* ou híbridos, sendo que este último é composto por uma combinação dos tipos anteriores.

Monitores em *software* geralmente possuem taxas de coleta mais baixas que monitores em *hardware*, e sobrecarga maior. No entanto são mais fáceis de serem desenvolvidos e modificados, caso necessário, e ainda podem capturar informações em um grau mais alto de abstração. Monitores em *software* também apresentam um custo menor de desenvolvimento que monitores em *hardware*. Os monitores em *firmware* apresentam características intermediárias entre os monitores em *software* e *hardware*. Tal tipo de monitor possui um custo mais baixo do que os em *hardware*, porém, assim como os em *hardware*, não consegue capturar informações em níveis tão altos de abstração como os de *software*.

Quanto ao tipo de mecanismo que ativa a captura de dados podem-se classificar os monitores como ativados por eventos (também conhecidos como monitores de traços, ou *tracing*) ou por temporizador (também chamados de monitores de *sampling*). Um monitor ativado por eventos realiza captura de dados na ocorrência dos eventos específicos aos quais está relacionado. Sendo assim, a sobrecarga causada por monitores ativados por eventos tende a ser baixa, caso o evento seja raro, e alta, caso se repita freqüentemente (LILJA, 2000). Monitores temporizados são aqueles ativados em intervalos de tempo fixos; portanto, a sobrecarga desse tipo de monitor tende a ser constante (e, portanto, de maior previsibilidade). Monitores temporizados são, por este motivo, apropriados para o monitoramento de programas com eventos que ocorrem com alta freqüência.

Quanto ao tipo de visualização, podem-se ter monitores *online* e *batch* (JAIN, 1991). Monitores *online* permitem a visualização do estado do sistema de modo contínuo durante a captura dos dados. Já os monitores *batch* (ou *offline*) coletam os dados para análise posterior (ou análise pós-morte), utilizando, opcionalmente, outro sistema para a análise dos dados coletados.

1.2.2 Componentes de monitores

Para analisar um sistema computacional, muitos podem ser os aspectos de interesse para a coleta de dados. Em vista disso, cabe coletar simultaneamente diversos tipos de dados, visto que diversos eventos podem ocorrer de modo concorrente. Para tanto, um monitor deve possuir diversos componentes para coleta e organização da informação

obtida. Jain (1991) propôs a separação em camadas das diversas funções existentes em um monitor, conforme enumeração abaixo:

- *Observação*: Esta camada captura dados em pontos específicos do sistema. Normalmente, cada componente monitorado tem um observador desenvolvido especificamente para ele;
- *Coleta*: Camada que recolhe dados de vários observadores. Sistemas grandes podem possuir mais de um coletor;
- *Análise*: Realiza o estudo dos dados coletados. Pode consistir de rotinas estatísticas para sumarizar os dados, dentre outras;
- *Apresentação*: Este componente refere-se à interface com o usuário. Ela produz, por exemplo, relatórios e gráficos;
- *Interpretação*: Este componente refere-se a uma entidade inteligente (pessoa ou sistema especialista) que pode fazer uma interpretação dos resultados.
- *Console*: Este componente provê uma interface para controle de parâmetros e estados do sistema. Tecnicamente, não faz parte do monitor, porém é útil a existência de funções de monitoramento e controle em uma mesma aplicação;
- *Administração*: A entidade que toma a decisão de modificar os parâmetros ou configurações de um sistema, com base nos dados coletados, é chamada de administrador. O administrador aplica suas decisões a partir do componente de console.

Um monitor pode constituir-se de múltiplos componentes de cada camada, não havendo obrigatoriedade da presença de todas as camadas (entretanto, as camadas de observação, coleta e apresentação são obrigatórias). É interessante notar que há uma correspondência muitos-para-muitos entre camadas sucessivas. Por exemplo, um observador pode enviar dados para diversos coletores, enquanto um coletor pode receber dados de diversos observadores.

1.3 *Profiling*

Profilers são, basicamente, monitores compostos por um conjunto de observadores para coleta de dados sobre a execução de um programa, dados estes que constituem um perfil do programa analisado. Existem *profilers* para diversas plataformas e linguagens de programação, dentre elas C, C++, Java e dotNet (MICROSOFT, 2005), e mesmo *profilers* que monitoram aplicações a partir do código nativo, como é o caso de VTune (INTEL, 2005).

A principal função de um *profiler* é a de avaliar o desempenho de sistemas. Os *profilers* mais conhecidos na indústria (BORLAND, 2004) (EJ-TECHNOLOGIES, 2005) são os utilizados para auxiliar o desenvolvedor na otimização de seu sistema. Porém, a técnica de *profiling* é também muito utilizada para a otimização automatizada de código objeto (DUFOUR et al., 2002), em especial de forma conjunta com compiladores. Neste trabalho, são focados especificamente os *profilers* utilizados para análise de desempenho ou depuração.

A principal funcionalidade dos *profilers* mais conhecidos é a de verificar em quais locais (seja em funções, rotinas ou módulos) um determinado sistema gasta a maior

parte do seu tempo de processamento. Assim, o desenvolvedor pode procurar otimizar os pontos mais custosos para o programa alvo, a fim de melhorar o desempenho do sistema como um todo (WILSON; KESSELMANN, 2000). Existem duas maneiras para otimizar um programa a partir de informações de tempo de processamento: pode-se procurar otimizar as rotinas que consomem o maior tempo ou fazer com que as rotinas mais onerosas sejam chamadas com menor freqüência. Para tanto, ainda de acordo com a abordagem de Wilson e Kesselmann (2000), muitos *profilers* provêm informações sobre quais métodos chamam os métodos que gastam maior tempo de processamento, e com qual freqüência.

Profilers mais recentes possuem uma ampla gama de observadores que coletam dados para análise de desempenho. Tais observadores podem coletar dados sobre uso de memória, estado de *threads* e tempo de processamento. Alguns *profilers* possuem também funções que se caracterizam mais como de depuração, como a de visualização do uso de blocos sincronizados para verificação de *deadlocks*.

1.4 A plataforma Java

A tecnologia Java - ambiente de execução e linguagem de programação - tem conquistado bastante espaço em relação a aplicações corporativas e *web* devido à edição Enterprise (SUN MICROSYSTEMS, 2005-d). Porém, a linguagem Java tem recebido, desde seu início, críticas devido a seu fraco desempenho (WILSON; KESSELMANN, 2000) (SHIRAZI, 2000), muitas delas devido a ser uma linguagem cujos programas executam a partir de uma linguagem intermediária e não a partir de um código objeto compilado para uma arquitetura específica.

No entanto, a tecnologia Java teve vários avanços que permitiram melhorias de desempenho, tanto no que diz respeito ao ambiente de execução (JVM) quanto aos recursos da linguagem de programação e suas bibliotecas de apoio (APIs). Dentre elas, destaca-se o uso de compiladores *Just-In-Time* (JIT) e melhorias na própria máquina virtual. Além disso, a nova versão da plataforma Java (versão 5.0) (AUSTIN, 2004) inclui otimizações para o compartilhamento da memória usada pelas classes carregadas por mais de uma máquina virtual, visto que diversas instâncias da JVM podem estar em atividade em uma mesma máquina real. Isto torna possível, por exemplo, ter-se um programa em execução em uma JVM ao mesmo tempo em que outra JVM executa um *profiler* que atua sobre esse programa em uma mesma máquina real, sendo que a memória utilizada pelas classes carregadas por ambas as aplicações podem ser compartilhadas entre elas.

1.5 Monitores Java

A linguagem Java tem sido muito utilizada no desenvolvimento de sistemas dos mais variados portes e tipos: centralizados, distribuídos e aplicações *web*. Muitos monitores, com os mais variados propósitos, já foram desenvolvidos para a coleta de dados sobre programas escritos nessa linguagem. Dentre eles, alguns monitores do tipo *profilers* tornaram-se razoavelmente populares por serem bastante úteis para depurar e analisar o desempenho de sistemas; incluem-se entre eles OptimizeIt (BORLAND, 2004), JProfiler (EJ-TECHNOLOGIES, 2004) e JFluid (DMITRIEV, 2004), descritos em maior detalhe no capítulo 4. Esses *profilers*, geralmente implementados como ferramentas não intrusivas que operam diretamente sobre a máquina virtual (JVM), em geral se destinam a uma única aplicação Java centralizada, e monitoram o uso de

memória, a carga de UCP e a execução de *threads*. Porém, aplicações distribuídas não podem ser tratadas como um conjunto de aplicações isoladas, e monitoradas apenas separadamente. Elas possuem outras exigências, necessitando a coleta de informações referentes a sua natureza distribuída como, por exemplo, a observação de conexões de rede.

1.6 Motivação e objetivos

No segundo semestre de 2004, a Sun Microsystems (2004-d) lançou uma nova versão da plataforma Java, a versão 1.5.0, nela incluindo a JVMTI (JVM Tool Interface), uma nova interface de programação nativa para uso em ferramentas. Essa interface permite inspecionar o estado e controlar a execução de aplicações Java, para fins de *profiling*, depuração, monitoramento, análise de *threads,* entre outros usos. Como boa parte das ferramentas, até então existentes, se baseavam nas APIs JVMPI (JVM Profiler Interface) e JVMDI (JVM Debug Interface) - interfaces estas que se tornaram obsoletas com o lançamento de JVMTI - abriu-se a possibilidade de desenvolvimento de novas ferramentas, mais atualizadas e com mais funcionalidades.

Aliando esta nova tecnologia à constatação da necessidade de ferramentas específicas para monitoramento, este trabalho propõe:

- Identificar os dados relevantes para o monitoramento de aplicações distribuídas desenvolvidas sob a plataforma Java;
- Criar uma ferramenta de monitoramento de aplicações, explorando os novos recursos do JDK 1.5, bem como os recursos já disponíveis em Java, como carga dinâmica de classes e transformação de *bytecodes*.

A ferramenta aqui proposta, denominada Nprof, tem como propósito principal o monitoramento de aplicações distribuídas em Java. Aplicações distribuídas são aqui consideradas como um conjunto de aplicações independentemente executadas e que se comunicam por meio de mensagens. Entende-se que a coleta simultânea de dados sobre aplicações diferentes, porém conectadas, é útil para a descoberta de relações entre eventos que ocorrem durante a execução, e assim o relacionamento entre dados pode servir de suporte a desenvolvedores para a depuração e otimização de sistemas distribuídos.

1.7 Organização do trabalho

O capítulo 2 prossegue apresentando medidas de desempenho, com o objetivo de melhor direcionar a coleta de dados, e o capítulo 3 aborda as tecnologias existentes para monitoramento de aplicações para subsidiar o projeto da arquitetura da ferramenta proposta. No capítulo 4, são apresentadas algumas ferramentas existentes na área de monitoramento de sistemas centralizados e distribuídos. O capítulo 5 descreve o projeto e as funcionalidades da ferramenta proposta. No capítulo 6, encontram-se os estudos de caso relacionados à ferramenta e, no capítulo 7, as conclusões.

2 MEDIDAS DE DESEMPENHO DE APLICAÇÕES

Este capítulo apresenta diferentes aspectos de desempenho e suas medidas, com especial atenção quanto a sistemas distribuídos, e propõe um conjunto de métricas para embasar a coleta de dados para tal tipo de sistema.

2.1 Apresentação

A otimização e a depuração de programas são, há muito tempo, tarefas importantes para o funcionamento correto e satisfatório de sistemas computacionais. No que se refere ao desempenho, o desenvolvimento de sistemas computacionais para a Internet, onde são freqüentes os sistemas com muitos usuários simultâneos e necessidade de tempo de resposta relativamente baixo, requer sistemas otimizados para o cumprimento de suas tarefas. Quanto à depuração, a crescente complexidade dos sistemas computacionais, aliada à necessidade de sistemas de alta disponibilidade, faz com que seja dada maior atenção a falhas nas fases de projeto, de construção e mesmo de instalação (*deployment*). Aqui se considera que a depuração não como verificação, mas como a fase de diagnóstico de falhas de um processo de remoção de falhas (LAPRIE, 1995). A verificação procura erros, enquanto o diagnóstico de falhas busca a causa desse erro, do qual já se conhece a existência; por exemplo, qual a causa de um *deadlock* observado?

2.2 Desempenho

Pode-se analisar o desempenho de um programa de diversas formas. Desempenho computacional possui diversos aspectos. Segundo Wilson e Kesselmann (2000), para que se possa analisar o desempenho de um programa, os seguintes aspectos - caracterizados com base nestes autores - devem ser considerados.

2.2.1 Desempenho computacional

Desempenho computacional refere-se ao tempo de execução do sistema, como, por exemplo, aquele gasto para a execução de uma rotina ou de uma chamada de método. O desempenho de um programa depende muito do tempo de execução de suas rotinas; a arquitetura da máquina utilizada, a freqüência de operação do processador e a estratégia do algoritmo escolhido são fatores preponderantes no desempenho computacional resultante. Porém esses não são os únicos fatores a serem considerados.

2.2.2 Uso de memória RAM

A quantidade de memória necessária para a execução de um programa pode ser de importância fundamental para seu geral desempenho. Apesar de os sistemas operacionais disporem de mecanismos de memória virtual em disco (caso a memória principal esteja escassa), a velocidade de acesso nesse meio é muito inferior à da memória RAM. É interessante notar também que muitas vezes parte da memória principal pode não estar disponível por estar comprometida no uso simultâneo de outros programas.

2.2.3 Tempo de inicialização

O tempo de inicialização pode ser um fator crítico para programas que executam em um ambiente interativo (não servidor). Em Java, pelo menos dois fatores influenciam o tempo de inicialização de um programa: a sobrecarga da máquina virtual e o mecanismo de compilação em tempo de execução. A sobrecarga da máquina virtual se deve à necessidade de carregar e inicializar diferentes componentes, antes mesmo de iniciar a execução do programa propriamente dito. O mecanismo de compilação em tempo de execução (JIT) faz com que o código intermediário Java (*bytecode*) seja incrementalmente compilado para código nativo, melhorando o desempenho de execução com o passar do tempo.

2.2.4 Escalabilidade

Escalabilidade se refere, no caso de desempenho, ao modo como um dado programa se comporta frente a diferentes cargas de trabalho. Apesar de programas que atuam como servidores estarem mais sujeitos a enfrentar diferentes cargas de trabalho (por exemplo, um site *web* que deve ter a capacidade de receber até 500 clientes simultâneos), a escalabilidade também é um fator relevante para sistemas cliente (por exemplo, um sistema de "renderização" tridimensional que deve tratar pequenas ou grandes quantidades de objetos).

2.2.5 Desempenho percebido

De relevante importância para aplicações cliente, este item se refere a forma de como o desempenho é percebido pelo usuário final de um determinado programa. Nem sempre os programas que possuem melhor desempenho são os mesmos que aparentam ter melhor desempenho. O fato de um programa mostrar resultados parciais de uma solicitação, ou apenas informar o que está sendo realizado, faz com que ele pareça possuir desempenho superior a outro programa que não considere tais fatores.

2.3 Tipos de medidas

Profilers convencionais utilizam uma série de observadores para a coleta de diferentes medidas para se obter uma visão global dos recursos computacionais utilizados por determinado programa. Dentre os tipos de medidas comumente coletados por um *profiler* encontram-se:

- *Uso de UCP*: O uso da unidade central de processamento (UCP) determina a quantidade de tempo que o sistema avaliado gasta na execução de cada rotina. Provavelmente seja esta a medida mais frequentemente obtida por *profilers*. Tipicamente, esta informação é utilizada para identificar as partes do sistema que são mais utilizadas para, ao otimizá-las, obter um ganho

maior no desempenho final do sistema como um todo (WILSON; KESSELMANN, 2000);

- *Grafo de Chamada de Rotina*: Revela a seqüência de chamadas entre rotinas de um sistema, e a freqüência de ocorrência de cada par (rotina chamadora, rotina chamada);
- *Uso de Memória*: O uso de memória por parte do sistema revela a parcela da memória principal está sendo utilizada pelo sistema monitorado;
- *Threads*: Informações coletadas sobre *threads* normalmente incluem o número de *threads* ativas e também o estado em que elas se encontram.

Considerando-se especificamente a plataforma Java, outras medidas comumente coletadas por *profilers* são:

- *Objetos*: O número de objetos alocados em memória em determinado momento ou ao longo do tempo;
- *Classes carregadas*: Determina quantas e quais são as classes utilizadas pela aplicação-alvo.

2.4 Medidas para aplicações distribuídas

O desempenho local de aplicações distribuídas pode ser medido de maneira semelhante ao desempenho de aplicações centralizadas, e os mesmos aspectos relacionados na seção 2.2 - desempenho computacional, uso de RAM, tempo de inicialização, escalabilidade e desempenho percebido - são válidos. Porém uma aplicação distribuída faz uso de trocas de mensagens para a realização da computação, o que não ocorre em aplicações centralizadas. Cada meio de comunicação (Ethernet, Wi-Fi, etc.) possui capacidade de transferência de dados e latência próprias. Como os outros fatores de desempenho dependem da velocidade do meio de comunicação, este também deve ser objeto de observação para fim de uma análise do desempenho da aplicação distribuída como um todo.

A fim de medir o uso do meio de comunicação de uma aplicação distribuída, alguns dados são importantes, como o número de mensagens enviadas e recebidas, o tamanho das mensagens e o tempo necessário para enviar e recebe-las.

2.5 Relacionamentos em aplicações distribuídas

Aplicações distribuídas podem ser organizadas na forma de cliente-servidor, e também segundo o modelo de componentes cooperantes, constituído por aplicações que executam em diferentes máquinas e que cooperam para a resolução de um determinado serviço. Em ambos os casos, existe a comunicação entre os componentes da aplicação, conforme esquematizado na figura 2.1

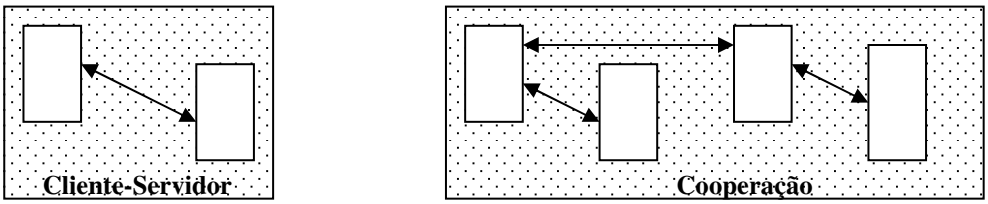


Figura 2.1: Comunicação entre componentes de aplicações distribuídas

Um dos objetivos deste trabalho é o de identificar os dados relevantes para o monitoramento de aplicações distribuídas; é importante, então, descobrir os relacionamentos existentes entre as aplicações que, de alguma forma, estão participando de uma aplicação distribuída. Uma maneira possível de identificar um relacionamento entre duas ou mais aplicações é verificar a existência de conexões entre elas.

Devido às várias possibilidades de conexão que uma aplicação pode ter com outra aplicação, torna-se necessário selecionar um subconjunto de aplicações correlacionadas para monitoramento. A este subconjunto dá-se aqui o nome *de grupo interno*. Ao subconjunto de todas as aplicações que não pertencem ao grupo interno, mas que possuem alguma relação com algum membro do grupo interno, dá-se o nome de *grupo externo*.

Para fins de medidas de desempenho de aplicações distribuídas são propostas as métricas abaixo relacionadas, considerando as aplicações-alvo (grupo interno) e suas conexões com outras aplicações (grupo externo).

- Número de aplicações-alvo (no grupo interno) e número de aplicações no grupo externo;
- Número de conexões *socket* (grupos interno e externo);
- Número de mensagens enviadas e recebidas (total, por *socket* e por aplicação);
- Número de *bytes* enviados e recebidos (total, por *socket* e por aplicação);
- Número de *bytes* nos *buffers* de leitura dos *sockets* (total, por *socket* e por aplicação).

Para cada uma destas métricas podem-se conhecer as medidas que se referem a aplicações do grupo interno ou externo. É importante notar que esses grupos são considerados dinâmicos: se uma aplicação do grupo externo passar a ser também monitorada, as conexões estabelecidas por esta aplicação serão consideradas como pertencentes ao grupo interno.

2.6 Algumas considerações

Este capítulo, ao apresentar aspectos de desempenho e medidas para análise de desempenho, forneceu subsídios para determinar e justificar as métricas a serem observadas pela ferramenta proposta neste trabalho. Com base nas métricas propostas, diferentes problemas de desempenho, ou mesmo de projeto, podem ser analisados.

Outrossim, outras medidas podem ser derivadas dessas métricas como, por exemplo, o número de *bytes* enviados entre membros do grupo interno, mas ainda não recebidos por algum de seus membros. O capítulo 5, que descreve em detalhe a ferramenta proposta, associa coletores de dados específicos a essas métricas.

3 TECNOLOGIAS DE CONSTRUÇÃO DE FERRAMENTAS PARA MONITORAMENTO

Este capítulo descreve as principais tecnologias e técnicas de implementação usadas na construção de ferramentas de monitoramento, com ênfase na plataforma Java.

3.1 Tecnologias utilizadas para Java

Um programa de usuário normalmente não tem acesso a informações de desempenho sobre a própria execução ou mesmo a informações internas ao ambiente de execução. Para a interceptação de informações, muitas vezes faz-se necessário o uso de bibliotecas que agreguem esse tipo de funcionalidade.

No caso da plataforma Java, podem-se utilizar diferentes técnicas para a obtenção desse tipo de informação. Existem tanto bibliotecas que interagem diretamente com a máquina virtual (em código nativo) como bibliotecas na linguagem Java. Como a Sun Microsystems (2005-e) põe à disposição o código-fonte de sua máquina virtual, há monitores que se valem da alteração da própria máquina virtual para a interceptação de dados. Há ainda a possibilidade de instrumentação da aplicação-alvo para a coleta de dados. Essa instrumentação pode ser realizada de duas maneiras: (i) no código-fonte, que requer que esteja disponível e seja passível de alteração o código-fonte da aplicação-alvo para a coleta de dados; (ii) e no código objeto (*bytecode*), que permite incorporar ou alterar segmentos de rotinas por meio de bibliotecas específicas. Ao longo da presente seção, serão descritas as principais técnicas de coleta de dados sobre uma aplicações em Java.

3.1.1 JVMPI

A JVMPI (Java Virtual Machine Profiling Interface) (VISWANATHAN; LIANG, 2000) é uma API em código nativo, para a linguagem C, que interage diretamente com a máquina virtual Java para o desenvolvimento de aplicações de *profiling*. A JVMPI foi introduzida na versão 1.1 da plataforma Java, mas sempre foi considerada uma API experimental. Essa API continuou sendo incluída nas versões mais recentes de Java, porém sua utilização implicava sobrecarga da aplicação monitorada e, além disso, havia problemas de compatibilidade entre a API e alguns tipos de coletores de lixo (O'HAIR, 2004). Recentemente, esta API foi descontinuada em função da criação da JVMTI, desenvolvida para agrupar as atribuições da JVMPI com as da JVMDI (seção 3.1.2.1).

A arquitetura da biblioteca JVMPI compreende um módulo servidor, que implementa as funções de *profiling* e está presente na máquina virtual Java; e um agente JVMPI, que interage com o módulo servidor por meio de chamadas da API em código nativo.

Muitas ferramentas utilizam a JVMPI para a captura de dados de uma máquina virtual em execução, dentre elas OptimizeIt (BORLAND, 2004), JProfiler (EJ-TECNOLOGIES, 2004) e Hprof (LIANG; VISWANATHAN, 1999). As versões mais recentes dessas ferramentas já utilizam a JVMTI como API padrão.

A captura de dados, a partir da JVMPI, se dá por meio da solicitação e recebimento de eventos gerados pela máquina virtual. A tabela 3.1 mostra os diferentes tipos de eventos definidos pela JVMPI.

Tais eventos devem ser solicitados pelo agente JVMPI à máquina virtual Java. Para tanto, a JVM possui um conjunto de funções para a solicitação, ativação e desativação de eventos. Além das funções relacionadas com a ativação e desativação de eventos, a JVMPI possibilita o uso das seguintes funções:

- GetCallTrace: função chamada pelo agente para obter a pilha de execução de uma *thread*;
- SuspendThread, ResumeThread: funções chamadas pelo agente JVMPI para suspender e prosseguir a execução de *threads*;
- ThreadHasRun: função usada para determinar se a *thread* entrou em estado de execução desde a última vez que foi suspensa;
- GetThreadStatus: função utilizada para determinar o estado de uma *thread*, como, por exemplo, rodando, bloqueada ou esperando por acesso a monitor.
- EnableGC, DisableGC, RunGC: funções para ativar, desativar e rodar o coletor de lixo.

A JVMPI é considerada, portanto, uma API de mão-dupla, pois permite tanto a coleta de informações de *profiling,* quanto alguns tipos de atuação na aplicação monitorada como, por exemplo, suspender/prosseguir *threads* e desabilitar/reabilitar o coletor de lixo.

Tabela 3.1: Tipos de eventos JVMPI

Nome do evento	**Descrição**
THREAD_START, THREAD_END	Eventos disparados ao início e fim da execução de *threads*.
CLASS_LOAD, CLASS_UNLOAD	Eventos disparados quando classes são carregadas ou descarregadas pela máquina virtual.
CLASS_FOR_INSTRUMENT	Evento disparado quando uma classe está pronta para ser instrumentada pelo agente de *profiling*.
METHOD_ENTER, METHOD_EXIT	Eventos disparados quando a máquina virtual inicia ou termina a execução de um método.
NEW_ARENA, DELETE_ARENA	Eventos disparados quando nova área de memória é alocada para armazenamento de objetos ou desalocada (memória *heap*).
NEW_OBJECT, DELETE_OBJECT	Eventos disparados quando objetos são criados ou liberados pelo coletor de lixo.
MOVE_OBJECT	Evento disparado quando um objeto é movido entre diferentes áreas de alocação, o que ocorre durante a execução do coletor de lixo.
COMPILED_METHOD_LOAD, COMPILED_METHOD_UNLOAD	Eventos gerados quando um método é compilado por um compilador JIT e é carregado em memória ou quando um desses métodos é desalocado.
MONITOR_CONTENDED_ENTER	Evento disparado quando uma *thread* é bloqueada ao tentar entrar em monitor que pertence a outra *thread*.
MONITOR_CONTENDED_ENTERED	Evento disparado quando uma *thread* que estava bloqueada para entrar em monitor entra no monitor.
MONITOR_CONTENDED_EXIT	Evento disparado quando uma *thread* sai de monitor e é verificado que outra *thread* está bloqueada pelo mesmo monitor.
MONITOR_WAIT	Evento disparado quando uma *thread* fica em estado de espera.
MONITOR_WAITED	Evento disparado quando uma *thread* sai do estado de espera.
HEAP_DUMP	Evento disparado para mostrar o estado das áreas de alocação *heap*.
MONITOR_DUMP	Evento disparado para mostrar o estado de todos os monitores e *threads* do sistema.

3.1.2 JPDA

A JPDA (Java Plataform Debugging Architecture) (SUN MICROSYSTEMS, 2005-b) é uma arquitetura multi-camadas para depuração de programas Java, que permite aos desenvolvedores a criação de ferramentas de depuração e monitoramento de forma padronizada. A JPDA consiste de três camadas: a camada de instrumentação da máquina virtual (camada *back-end* - JVMTI), que se encontra junto à máquina virtual

monitorada; a camada de comunicação (JDWP); e a de depuração em Java (JDI). A arquitetura da JPDA em camadas visa possibilitar uma maior flexibilidade para a implementação de ferramentas de monitoramento, e assim permitir tanto o monitoramento de aplicações Java em máquinas remotas, por intermédio do protocolo JDWP (SUN MICROSYSTEMS, 2005-b), como o monitoramento local a partir da JVMDI ou JVMTI. A figura 3.1 ilustra a arquitetura.

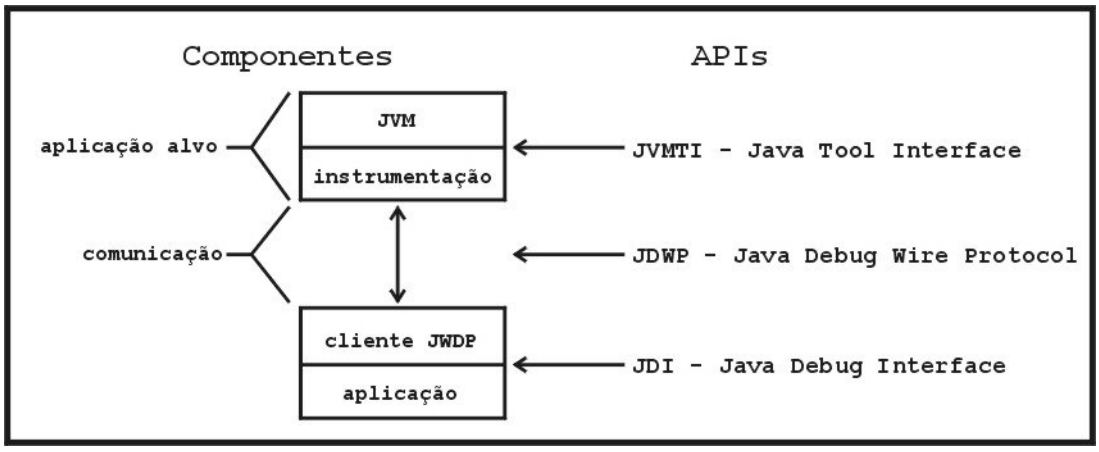


Figura 3.1: Arquitetura da JPDA, adaptado de (SUN MICROSYSTEMS, 2005-b)

Note-se que a camada de instrumentação da JVM, anteriormente denominada JVMDI, passou, a partir da versão 1.5 da plataforma Java, a se chamar de JVMTI (Java Virtual Machine Tool Interface), sendo o resultado junção das duas APIs de código nativo JVMPI (Java Virtual Machine Profiler Interface) e JVMDI (Java Virtual Machine Debug Interface).

3.1.2.1 JVMDI

A JVMDI (SUN MICROSYSTEMS, 2004-a) é uma API também de código nativo desenvolvida para a depuração de aplicações Java. Assim como a JVMPI, ela é atualmente considerada obsoleta e deverá ser removida das futuras versões da plataforma Java.

Uma das motivações para a junção das bibliotecas JVMDI e JVMPI se revela no fato de possuírem muitas funcionalidades semelhantes. Na tabela 3.2, encontram-se os principais eventos que podem ser capturados por meio da JVMDI e a informação sobre a existência, ou não, de evento similar na JVMPI.

Tabela 3.2: Eventos da JVMDI

Nome do Evento	**Evento similar na JVMPI**	**Descrição**
JVMDI_EVENT_SINGLE_STEP	Não	Evento gerado após uma *thread* executar uma instrução da máquina virtual.
JVMDI_EVENT_BREAKPOINT	Não	Evento gerado quando uma *thread* chega em um ponto da aplicação previamente designada como ponto de parada (*breakpoint*).
JVMDI_EVENT_FIELD_ACCESS	Não	Evento gerado quando determinado campo de uma classe é lido.
JVMDI_EVENT_FIELD_MODIFICATION	Não	Evento gerado quando determinado campo de uma classe é modificado.
JVMDI_EVENT_FRAME_POP	Não	Evento gerado ao término da execução de determinado método a partir de solicitação específica.
JVMDI_EVENT_METHOD_ENTRY	Sim	Evento gerado ao início da execução de determinado método.
JVMDI_EVENT_METHOD_EXIT	Sim	Evento gerado ao término da execução de determinado método.
JVMDI_EVENT_EXCEPTION	Não	Evento gerado quando uma exceção é lançada.
JVMDI_EVENT_EXCEPTION_CATCH	Não	Evento gerado quando uma exceção é capturada.
JVMDI_EVENT_THREAD_END	Sim	Evento gerado ao término da execução de uma *thread*.
JVMDI_EVENT_THREAD_START	Sim	Evento gerado quando uma *thread* é iniciada.
JVMDI_EVENT_CLASS_LOAD	Sim	Evento gerado quando uma classe é carregada pela JVM.
JVMDI_EVENT_CLASS_UNLOAD	Sim	Evento gerado quando uma classe é desalocada.
JVMDI_EVENT_CLASS_PREPARE	Sim	Evento gerado quando termina a fase de preparação de uma classe.
JVMDI_EVENT_VM_INIT	Sim	Evento gerado ao término da inicialização da máquina virtual.
JVMDI_EVENT_VM_DEATH	Sim	Evento gerado ao término da máquina virtual.

Nota-se, assim, que grande parte dos eventos capturados pela JVMDI possui equivalente na JVMPI. Os eventos que não possuem equivalentes se referem ao uso de pontos de parada (*breakpoints*), acesso e modificação de campos e lançamento e captura de exceções. Por ser uma API voltada para depuração, ela possui um maior número de funções para atuar sobre a máquina virtual monitorada.

3.1.2.2 JVMTI

A JVMTI (SUN MICROSYSTEMS, 2004-d) provê praticamente todas as funcionalidades das APIs de código nativo antecessoras, tanto da JVMPI quanto da JVMDI (O'HAIR, 2004). Ademais, ela não possui boa parte das limitações das mesmas APIs. Entretanto, algumas funcionalidades existentes na JVMPI requerem o uso da JVMTI conjugada com a técnica de instrumentação de *bytecode*. Uma vantagem da JVMTI é a de que múltiplos agentes que usem essa biblioteca podem operar simultaneamente em uma mesma JVM.

É interessante notar que, devido à arquitetura da JPDA, aplicações que utilizam a API JDI, ou o protocolo JDWP, vão, indiretamente, fazer uso da JVMTI, pois o módulo servidor (*back-end*), mesmo que implementado na própria JVM, se utiliza da interface exposta pela JVMTI.

3.1.2.3 JDWP

O Protocolo de Depuração Java (Java Debug Wire Protocol) (SUN MICROSYSTEMS, 2004-b) é utilizado para a comunicação entre a aplicação de depuração e a máquina virtual Java monitorada. Sua especificação permite que diferentes implementações de aplicações de depuração possam conectar-se a uma máquina virtual Java que tenha sido inicializada com o agente da JVMTI. e, por ser um protocolo, não restringe a linguagem de implementação da aplicação de depuração. Diferentes implementações de máquinas virtuais Java implementaram o protocolo JDWP, porém poucos projetos de depuradores que utilizam JDWP não são desenvolvidos em Java. Devido à alta complexidade da JDWP, a maioria das aplicações de depuração para Java ou utiliza-se de uma biblioteca em código nativo (seja JVMDI, JVMPI ou JVMTI) ou utiliza-se da API JDI em Java.

3.1.2.4 JDI

A Interface de Depuração Java JDI (Java Debug Interface) (SUN MICROSYSTEMS, 2004-c) é uma API de alto nível que fornece informações úteis a monitores e depuradores que necessitam do estado de execução de uma máquina virtual Java, seja ela local ou remota.

A JDI provê mecanismos para o acesso do estado da máquina virtual Java por meio de classes, *arrays*, interfaces, tipos primitivos, e instâncias de quaisquer dos tipos.

A JDI também permite o controle explícito sobre a execução de uma JVM. Ela possibilita a suspensão e reativação de *threads*, a criação de pontos de parada e pontos de inspeção (*watchpoints*). Permite também que a aplicação de depuração receba notificação de uma ampla gama de eventos, como o lançamento de exceções, carga de classes, criação e término de *threads* e chamada de métodos específicos.

Além disso, a JDI também possibilita que sejam inspecionados o estado, as variáveis locais e a pilha de execução de *threads* suspensas. Outra vantagem de se utilizar a JDI é a de desenvolver-se uma aplicação de depuração que será compatível com qualquer máquina virtual, ao contrário de uma ferramenta desenvolvida para uma biblioteca nativa, que deverá, pelo menos, ser recompilada para cada arquitetura específica. Porém, por ser uma API de alto nível, ela tende a gerar uma sobrecarga maior à aplicação-alvo.

Diversas aplicações utilizam-se da JDI para diferentes tarefas que envolvem o monitoramento de uma máquina virtual Java (OECHSLE; SCHMITT, 2002) (MEHNER, 2002). Esta foi também a API escolhida para a implementação da ferramenta Nprof.

3.1.3 Alteração na JVM

Como o código-fonte da linguagem Java é disponibilizado no *site* da Sun Microsystems, uma possibilidade para o monitoramento de uma máquina virtual Java é a de alterar a própria máquina virtual Java para a inserção de ganchos de chamada (*hooks*) ou mesmo para a própria coleta das informações relevantes. Por meio da alteração da máquina virtual, as possibilidades de monitoramento são praticamente irrestritas, porém a complexidade para alterar tal código é alta.

A alteração da máquina virtual Java traz consigo a dificuldade da distribuição: somente ambientes que possuam a máquina virtual alterada poderão executar uma ferramenta de monitoramento que faça uso de tais alterações. Ademais, existe uma restrição legal: apesar de a Sun Microsystems liberar o código-fonte de sua máquina virtual Java, caso esta seja alterada, sua licença de uso não permite a livre distribuição de tais alterações (SUN MICROSYSTEMS, 2005-c).

Existem, no entanto, aplicações de monitoramento de grande relevância que utilizam a alteração da máquina virtual para a coleta de informações, como a ferramenta JFluid (DMITRIEV, 2004) e algumas versões da ferramenta Jinsight (PAUW et al., 2001). Outras ferramentas, também desenvolvidas pela Sun, como VisualGC e jstat (2005-a), utilizam-se da instrumentação da máquina virtual para a coleta de informações.

3.1.4 Instrumentação de *bytecode* e código-fonte

A instrumentação de uma aplicação para coleta de informações sobre a execução dessa mesma aplicação é uma técnica comumente utilizada. A instrumentação pode dar-se basicamente a partir de dois pontos: do código-fonte, ao se inserir chamadas para alguma API que faça a coleta de dados; ou do código objeto (*bytecode* Java, por exemplo). Caso o número de instrumentações necessárias seja pequeno, a instrumentação manual, a partir do código-fonte, pode ser uma boa alternativa. Porém, para casos em que são necessários vários pontos de instrumentação, pode tornar-se uma tarefa muito trabalhosa para o desenvolvedor, além de ser uma tarefa suscetível a erros. A instrumentação a partir do código objeto tem também a vantagem de permitir a instrumentação de aplicativos ou bibliotecas dos quais não se possui o código-fonte.

O desenvolvimento de bibliotecas para a manipulação de código objeto Java incentivaram consideravelmente o uso da técnica de instrumentação de *bytecode*. Dentre as bibliotecas utilizadas para alteração de *bytecode* destacam-se a BCEL (APACHE, 2004) e a Javassist (CHIBA; NISHIZAWA, 2003). Tanto a ferramenta JFluid como a Nprof utilizam instrumentação de *bytecode*, juntamente com a técnica de *hotswapping*, que permite a alteração de *bytecode* para uma classe já carregada pela máquina virtual Java, visando a coleta de informações.

3.1.5 Programação Orientada a Aspectos

A técnica de programação orientada a aspectos (POA) permite a separação de interesses, ou aspectos, que representam funcionalidades auxiliares de uma aplicação, em módulos distintos (KICKZALES, 1997). Os aspectos são combinados em pontos

específicos da aplicação, denominados de pontos de junção. Os *frameworks* de POA para Java realizam esta combinação de uma das duas formas: ou fazem uma compilação de toda aplicação inserindo os aspectos nos pontos determinados, ou fazem a instrumentação dinâmica em tempo de execução. A POA se presta, de forma especial, para a inserção de código referente a aspectos não funcionais da aplicação. Algumas ferramentas têm valido-se de tecnologia de aspectos para a geração de traços (*logging*) e monitoramento de aplicações. Dentre os trabalhos relacionados com *profiling,* destacam-se os de Sato, Chiba e Tatsubori (2003), Davies et al. (2003) e Farquhar e Cannon-Brookes (2003).

3.1.6 JMX

A API de JMX para administração e monitoramento (SUN MICROSYSTEMS, 2004-e) é uma nova API para monitoramento de máquinas virtuais Java introduzida na nova versão da plataforma Java (versão 1.5). Essa API baseia-se no padrão de MBeans (*Management Beans*), que tem se tornado bastante difundido ultimamente. Dentre as informações disponibilizadas por esta nova API, encontram-se:

- informações sobre *threads* ativas e classes carregadas;
- informações do ambiente de execução, como tempo de atividade, propriedades de sistema, e parâmetros para a máquina virtual;
- informações de *thread*, como estado, estatísticas de contenção devido a acesso a monitores e pilha de execução;
- uso de memória;
- estatísticas de coleta de lixo;
- detecção de *deadlock*;
- informações sobre o sistema operacional no qual a máquina virtual está executando;

JMX (*Java Management Extensions*) é uma API para a linguagem Java, e os dados coletados por essa API estão disponíveis por meio de classes do pacote `java.lang.management`. Como essa API é recente, poucas aplicações fazem uso direto das informações específicas sobre a máquina virtual. No entanto, como ela se baseia no padrão de MBeans, espera-se que venha a se tornar bastante difundida. A aplicação JConsole (CHUNG, 2004), contida no JDK, é uma aplicação que utiliza tal API.

3.2 Considerações

Neste capítulo foram detalhadas diversas tecnologias para o monitoramento de aplicações Java. Visto que algumas dessas tecnologias são bastante recentes, poucas são as ferramentas que fazem uso delas. Desta forma, decidiu-se utilizar, na ferramenta Nprof, uma arquitetura híbrida que permite, tanto quanto possível, utilizar diferentes técnicas para a coleta de dados para o monitoramento de aplicações distribuídas.

Considerando que algumas tecnologias possuem propósitos muito semelhantes (como no caso de JVMDI e JDI) ou são refinamentos evolutivos (como JVMPI e JVMTI), optou-se por selecionar um subconjunto das tecnologias disponíveis para a implementação da ferramenta. Tais tecnologias são: JDI, JMX e instrumentação de *bytecode*.

A opção pelo uso das referidas tecnologias (e por não utilizar outras tecnologias) baseou-se nos seguintes motivos:

- JVMPI e JPDA possuem funcionalidades bastante semelhantes, portanto é desnecessário o uso de ambas as tecnologias. Decidiu-se utilizar a JPDA porque: (i) JPDA permite o desenvolvimento de ferramentas de monitoramento sem o uso de código não-Java (nativo); (ii) e porque JVMPI é uma API em vias de extinção;
- Foi verificado que o uso conjugado de tais tecnologias não acarretaria incompatibilidades entre elas;
- As tecnologias de instrumentação de *bytecodes* e de *Java Management* são relativamente novas e o uso delas seria interessante para verificar suas potencialidades;
- Não se utilizou instrumentação da máquina virtual devido: (i) à grande complexidade de instrumentação; (ii) a restrições para redistribuição da ferramenta; (iii) e por tornar a solução menos portável;
- Optou-se por utilizar tecnologias existentes para tornar a ferramenta mais compatível com as aplicações existentes e por facilitar a instalação da aplicação em ambientes novos.

4 FERRAMENTAS DE DEPURAÇÃO, PROFILING E VISUALIZAÇÃO DE PROGRAMAS

4.1 Apresentação

Diversas são as ferramentas que, de uma forma ou de outra, são utilizadas para o monitoramento de uma aplicação Java. A seguir serão descritas algumas ferramentas que serviram como base para a criação da ferramenta proposta Nprof e que possuem propósitos semelhantes a esta.

O objetivo de ferramentas de *profiling*, depuração e visualização avaliadas é o de caracterização do comportamento de determinado programa. A análise, realizada a partir das diversas ferramentas consideradas, avalia diferentes aspectos da execução de um programa. As ferramentas de *profiling* e depuração são apresentadas em separado das ferramentas de visualização, visto que estas últimas não se destinam tanto à depuração e à realização de medidas de desempenho.

4.2 Ferramentas de depuração e *profiling*

4.2.1 OptimizeIt

Inicialmente desenvolvido pela Intuitive Systems e adquirido pela Borland (2005-b), o OptimizeIt (BORLAND, 2004) é uma das ferramentas mais conhecidas para *profiling* de aplicações Java. Esta ferramenta utiliza a JVMPI para a coleta de dados e possui módulos para *profiling* de UCP, memória, estado de *threads* e contenção de monitores, além de módulos para servidores como, por exemplo, para avaliação de banco de dados (JDBC) ou de diretórios de nomes (JNDI). Além dessas funcionalidades, o OptimizeIt possui um módulo para teste de cobertura de código, permitindo identificar partes não utilizadas do código de uma aplicação.

4.2.2 JProfiler

Desenvolvido pela Ej-Technologies, JProfiler (EJ-TECHNOLOGIES, 2004) é, assim como OptimizeIt, uma ferramenta comercial para *profiling* de aplicações Java. Ela utiliza a JVMPI para a coleta de dados e possui suporte a *profiling* de UCP, *threads* e uso de memória. Além disso, JProfiler possui um módulo para inspeção de valores de quaisquer variáveis presentes na aplicação monitorada, chamado Heapwalker.

Entretanto, possivelmente devido ao fato de ser uma ferramenta comercial, não foram encontradas muitas informações sobre seu funcionamento interno.

4.2.3 JFluid

A ferramenta JFluid (DMITRIEV, 2004) é uma ferramenta recentemente desenvolvida pela Sun Microsystems (2005-e) e possui as funcionalidades de um *profiler* convencional: faz monitoramento do uso de UCP, memória e de estados das *threads*. Entretanto, ao invés de utilizar as APIs padrões para monitoramento, JFluid utiliza-se de alterações na própria JVM para a inserção de ganchos de chamadas e também para a instrumentação dinâmica de *bytecode*. Atualmente, tal ferramenta é também incorporada no ambiente de desenvolvimento NetBeans (SUN MICROSYSTEMS, 2005-f).

4.2.4 AppPerfect

AppPerfect DevSuite (APPPERFECT, 2006) é uma suíte de aplicativos que inclui análise de código, teste de carga (para servidores HTTP, JDBC e WebServices) e *profiler*. Dentre as funcionalidades do *profiler,* encontram-se a visualização de memória *heap* e não-*heap*, UCP, *threads* e informações sobre o sistema operacional (UCP, rede, disco e memória).

4.2.5 Hprof

Hprof (LIANG; VISWANATHAN, 1999) é uma ferramenta de *profiling* de UCP e uso de memória que é distribuído junto ao JDK da Sun Microsystems. As versões mais antigas desse *profiler* utilizavam a JVMPI para a coleta de dados, porém a que foi incluída na nova versão da plataforma Java utiliza a JVMTI (O'HAIR, 2005). Ela permite a coleta de dados tanto por amostragem quanto por traços e gera um arquivo com os dados coletados. Por ser uma ferramenta bastante conhecida, e possivelmente a primeira ferramenta de *profiling* a utilizar uma API Java padrão para a coleta de dados, diversas outras ferramentas utilizam o arquivo de saída de Hprof para análises mais elaboradas.

4.2.6 DynaMetricO

Protótipo desenvolvido a partir do projeto Sable (DUFOUR et al., 2002), essa ferramenta avalia a execução dinâmica de um programa a fim de classificá-lo de acordo com uma série de métricas. O conjunto de métricas definidas contem métricas de tamanho e estrutura do programa alvo (número de instruções de *bytecode* carregadas e executadas, número de instruções que influenciam o fluxo de controle, como *if*, *switch* e *invokeVirtual*), estrutura de dados (intensidade de uso de operações de vetores, de ponto flutuante e de ponteiros), polimorfismo (diversas métricas sobre o uso de métodos polimórficos), memória (densidade de alocação, tamanho dos objetos alocados, tempo de vida de objetos) e concorrência (número de *threads* em estado de execução, intensidade do uso de operações de *lock*).

Além de a ferramenta ser utilizada para a caracterização e classificação de programas, a proposta dos autores é a de utilizar a ferramenta para a otimização de código objeto gerado por compiladores. Para a implementação da ferramenta foi empregada a JVMPI.

4.2.7 Javiz

A ferramenta Javiz (KAZI et al., 2000) possui algumas características em comum com a ferramenta Nprof. Além de ser uma ferramenta de visualização de aplicações, ela também é uma ferramenta de *profiling* de aplicações cliente-servidor, com o objetivo de construir um grafo de chamadas global (*global call graph*) entre aplicações RMI clientes e servidoras. Apesar de ser focada em *profiling* de aplicações distribuídas, um objetivo que também norteou o projeto do Nprof, ela não coleta dados sobre o uso de conexões *socket*, nem captura informações sobre *threads*, memória ou outras informações usuais de *profiling*. Javiz utiliza a instrumentação da JVM para a captura dos dados.

4.2.8 Aspect Oriented Performance Analysis Environment – AOP Profiler

O trabalho de Davies et al. (2003) utiliza-se da tecnologia de aspectos AspectJ (ECLIPSE FOUNDATION, 2005-a), voltada para a linguagem Java, a fim de criar um aspecto para melhorar o desempenho de aplicações por meio de armazenamento temporário (*caching*) de resultados freqüentemente calculados. Para identificar quais pontos da aplicação-alvo utilizam maior parte do processamento, o aspecto também contabiliza o tempo utilizado por cada método. Apesar de ser bem específica em seu objetivo, essa ferramenta é bastante interessante do ponto de vista de explorar o potencial da tecnologia de aspectos na área de *profiling*.

4.2.9 JConsole

JConsole é uma ferramenta que acompanha a mais nova versão da plataforma Java (1.5) e faz uso da nova API de Administração e Monitoramento (SUN MICROSYSTEMS, 2004-e). Por meio da tecnologia JMX, JConsole coleta informações sobre a execução de um programa Java. Dentre os tipos de dados coletados, encontram-se dados sobre carga de classes, memória, *threads*, coleta de lixo, ambiente de execução e sistema operacional (CHUNG, 2004). Por ser baseada na tecnologia JMX, a ferramenta permite o monitoramento remoto de aplicações.

4.2.10 VisualGC e Jstat

Incluídas a partir da versão 1.4.2 do JDK da Sun Microsystems (2005-a), essas ferramentas permitem a coleta de informações sobre compilação dinâmica de classes (compilação JIT), carga de classes e principalmente sobre coleta de lixo de uma JVM. As informações referentes à coleta de lixo são muito detalhadas, e permitem identificar "gerações" de objetos, a partir do tempo em que os objetos criados permanecem na memória sem serem coletados. Para a coleta dessas informações, a Sun Microsystems fez alterações na máquina virtual. Como estas alterações não possuem documentação detalhada, apenas essas ferramentas, da própria Sun Microsystems, estão aptas a coletar tais tipos de dados.

4.2.11 Atlassian Profiling

Atlassian Profiling (FARQUHAR, CANNON-BROOKES, 2003) é um *profiler* bastante simples, que apenas coleta informações de tempo de execução de métodos. Seu uso se dá de maneira análoga a de uma API de *logging*, ou seja, a instrumentação da aplicação deve ser feita no código-fonte pelo desenvolvedor. Porém, conforme estudo de Teare (2005), é possível conjugar o uso do *profiler* com um *framework* de aspectos para que a instrumentação dos métodos seja feita de forma automática, e sem a

necessidade de disponibilizar o código-fonte. Apesar de ser uma ferramenta simples, seu uso permite explorar o uso de aspectos para a área de *profiling*.

4.2.12 CLRProfiler

O CLRProfiler (Common Language Runtime Profiler) (SOLLICH, 2003) é um *profiler* criado pela Microsoft para o monitoramento de aplicações desenvolvidas para a plataforma dotNET (MICROSOFT, 2005). Assim como Java, a plataforma dotNET é orientada a objetos e possui uma máquina virtual que se encarrega de serviços como coleta de lixo e sistema de tratamento de exceções. Suas principais funcionalidades são: visualização de informações sobre coleta de lixo (identificando quais métodos alocam que tipo de objetos, quais tipos de objetos permanecem mais tempo alocados sem serem coletados) e elaboração de grafo de chamadas entre métodos.

4.2.13 VTune

VTune (INTEL, 2005) é uma ferramenta da Intel para *profiling* de aplicações voltada para seus processadores. Ao contrário dos *profilers* tradicionais, VTune faz o monitoramento de todos os processos que estão ativos em determinada máquina e não apenas de uma aplicação específica. A ferramenta permite a coleta de informações sobre tempos de execução e grafo de chamadas. Ela também possui um módulo específico para Java, que utiliza a JVMPI.

4.2.14 Gprof

Gprof (FENLASON; STALLMAN, 1998) é um *profiler* que coleta dados sobre o tempo de processamento de cada função. Este *profiler* não está atrelado a uma linguagem específica, mas necessita da recompilação do programa alvo para a coleta de informações. Além de gerar um histograma do tempo de execução das funções, ele também cria o grafo de chamadas de funções. O Gprof utiliza uma biblioteca específica para a instrumentação de chamadas de métodos, a fim de gerar o grafo de chamada, e utiliza amostragem para contabilizar o tempo utilizado por cada função.

4.2.15 Hy+

Hy+ (CONSENS; HASAN; MENDELZON, 1994) é um sistema de visualização de informações para depuração pós-morte de aplicações paralelas e distribuídas. Uma de suas principais funcionalidades é a construção de um grafo que mostra a ordem parcial da execução das primitivas de comunicação a partir de traços armazenados em uma base de dados. O sistema possibilita a análise desses traços por meio de um sistema de consultas próprio a tal base de dados. Hy+ atua a partir da captura de traços da execução de primitivas de comunicação interprocesso da linguagem experimental Hermes (STROM et al., 1991).

4.2.16 VisuSniff

VisuSniff (OECHSLE; GRONZ; SCHÜLER, 2002) é uma ferramenta desenvolvida em Java que tem o propósito de capturar e permitir a visualização de dados enviados e recebidos por um adaptador de rede. VisuSniff permite visualizar, de maneira geral, todas as conexões entre aplicações existentes em uma máquina-alvo monitorada e as outras máquinas com as quais a máquina-alvo estabelece conexão. A ferramenta pode capturar os dados a partir do adaptador de rede ou carregar um arquivo que contenha os dados capturados por outra aplicação (como TcpDump e WinDump) para visualização.

TcpDump (JACOBSON; LERES; MCCANNE, 2002) e WinDump (RISSO et al., 2005) são ferramentas utilizadas com finalidade de captura de dados transmitidos pela rede e seu armazenamento em arquivo. VisuSniff também possui diversos mecanismos para a filtragem de dados. Para a captura de pacotes, VisuSniff utiliza a JPCAP (CHARLES; HADDAD; BITTEKER, 2005), que é uma biblioteca em Java que realiza chamadas a bibliotecas nativas para a captura dos pacotes de rede.

4.2.17 Netgroup: Analyzer

Analyzer (DEGIOANNI et al, 2005) é uma ferramenta para captura e visualização do tráfego de pacotes em uma rede desenvolvida na linguagem C++ para o sistema operacional Windows. Ela permite a visualização de cabeçalhos e conteúdo dos pacotes de forma cronológica e possui diversos modos de filtragem de pacotes. A ferramenta utiliza a biblioteca WinPcap (RISSO et al., 2005) para a captura dos pacotes.

4.2.18 JRastro

JRastro (SILVA; SCHNORR; STEIN, 2003) é uma ferramenta para o monitoramento e depuração de aplicações Java concorrentes e distribuídas. Ela permite a coleta de dados sobre o início e fim da execução de *threads*, invocações de métodos locais e uso de memória e chamadas RMI (limitado a um único cliente/servidor por máquina). Por meio de análise pós-morte, a ferramenta permite correlacionar os eventos de chamadas remotas entre cliente e servidor RMI.

4.2.19 DCPI - DIGITAL Continuous Profiling Infrastructure

O trabalho de Anderson et al. (1997) propõe uma ferramenta para *profiling* contínuo de sistemas em ambiente de produção baseado em amostragem e faz um comparativo entre diversos *profilers* existentes para a arquitetura Alpha-Digital. Dentre os *profilers* analisados, encontram-se VTune (INTEL, 2005) e Gprof (FENLASON; STALLMAN, 1998). Apesar de ser uma análise relevante por comparar a sobrecarga de diversas ferramentas, a análise se restringe a *profiling* de tempo de execução, que não se encontra entre os principais aspectos considerados no presente trabalho.

4.2.20 TAU Profiler

O *profiler* TAU (SHENDE et al, 1998) é um *profiler* que possui suporte para diversas linguagens de programação. O trabalho de Shende e Malony (2001) descreve uma variante do *profiler* para Java, utilizando a biblioteca MPI (*Message Parsing Interface*). Tal *profiler* visa o monitoramento de aplicações Java que utilizem a biblioteca MPI para computação distribuída. Dentre as atividades monitoradas pelo *profiler,* destacam-se o tempo de uso de processamento pelos diversos métodos da aplicação e o envio e recebimento de mensagens. A instrumentação da aplicação se processa em dois níveis: o *profiler* TAU utiliza a JVMPI para a captura de tempos de execução e a biblioteca MPI, que é uma biblioteca em código nativo chamada pela aplicação Java, é modificada para coletar informações sobre envio e recebimento de mensagens.

4.3 Ferramentas de visualização de programas

Ferramentas de visualização de programas possuem muitas similaridades com as de monitoramento e depuração, ou de depuração visual, conforme Oechsle e Schmitt

(2002). Entretanto, as ferramentas que são propriamente de visualização de programas têm o propósito de ajudar no entendimento do funcionamento de um programa alvo, e são muitas vezes concebidas para programadores iniciantes, e não para depuração ou análise de desempenho de um programa.

Mesmo sendo direcionada para a área de depuração e avaliação de desempenho, a ferramenta Nprof tem características que podem ser vistas como próprias de visualização, como a de visualização de estados de *threads*, que permite analisar as mudanças de estados de *threads* ao longo do tempo, e a funcionalidade de visualização de relacionamentos entre aplicações em uma rede, a partir de suas conexões de rede.

Para fins de comparação, são descritas algumas ferramentas de monitoramento destinadas à visualização de programas.

4.3.1 Jinsight

Desenvolvido pela IBM, Jinsight (PAUW et al., 2001) é uma ferramenta para visualização de aplicações Java com funcionalidades bastante semelhantes às de um *profiler*. Ela é focada na visualização de ordem e tempos de chamadas e também na descoberta de desperdícios de memória (*memory leaks*). Ao contrário de outras ferramentas de visualização, como JaVis (MEHNER, 2002) e Javavis (OECHSLE; SCHMITT, 2002), Jinsight permite a visualização de sistemas razoavelmente grandes. Para captura de dados, Jinsight utiliza tanto a instrumentação da JVM (para JVMs mais antigas) como JVMPI (para as que possuem estas API). Entretanto Jinsight só pode ser utilizado para análise pós-morte.

4.3.2 Javavis

Javavis (OECHSLE; SCHMITT, 2002) é uma ferramenta desenvolvida para auxiliar o ensino de conceitos de programação orientada a objetos com a linguagem Java. A ferramenta permite a visualização da execução de uma aplicação Java por meio de diagramas UML (Linguagem de Modelagem Unificada), capturando informações sobre *threads*, chamadas de métodos e valores de variáveis. Para a captura dos dados, Javavis utiliza a JDI.

4.3.3 JaVis

JaVis (MEHNER, 2002) é uma ferramenta para visualização e depuração de programas concorrentes Java. Como a ferramenta Javavis, ela permite a visualização de um programa Java por meio de diagramas UML e, dentre suas funcionalidades, encontram-se a detecção e visualização de *deadlocks* em aplicações a partir de traços de execução. Para a captura de dados, JaVis também utiliza a JDI.

4.3.4 Jacot

Jacot (LEROUX; RÉQUILÉ-ROMANCZUK; MINGINS, 2003) é uma ferramenta similar a Javavis e JaVis, permitindo a visualização da execução de programas por meio de diagramas UML. Sua principal funcionalidade é a depuração de *threads*, podendo identificar casos de *deadlock* e *starvation*. Também utiliza a JDI para captura de dados.

4.4 Tabela de ferramentas

A fim de sumarizar os principais aspectos avaliados nas ferramentas analisadas, apresenta-se a tabela 4.1.

Tabela 4.1: Ferramentas analisadas

Ferramenta	Propósito[1]	Tecnologia	Visualização		Apl. distribuídas
			online	*offline*	
OptimizeIt	D	JVMPI / JVMTI	X	X	lim.[2]
JProfiler	D	JVMTI	X		
JFluid	D	BCI	X	X	
AppPerfect	D	JVMTI	X		lim.
Hprof	D	JVMTI		X	
DynaMetricO	Otim / D	JVMPI		X	
Javiz	D	instrumenta JVM		X	X
AOP Profiler	Otim / D	AOP	n/a	n/a	
JConsole	D	JMX	X		
VisualGC e Jstat	D	instrumenta JVM	X		
Atlassian Profiling	D	AOP		X	
CLRProfiler	D	.net	X		
VTune	D	nativo e JVMPI	X	X	
Gprof	D	Nativo		X	
Hy+	D	Linguagem própria		X	X
VisuSniff	Rede	Nativo	X	X	X
Netgroup: Analyzer	Rede	Nativo	X	?	X
JRastro	D	JVMPI		X	X
DCPI	D	Nativo	X		
TAU Profiler	D	JVMPI	?	X	X
Jinsight	D / V	JVMPI / JVM modif.		X	
Javavis	V	JDI	X		
JaVis	V	JDI		X	
Jacot	V	JDI	X		
Nprof	D	JDI / BCI	X	logs[3]	X

[1] Cada ferramenta é classificada de acordo com seu propósito principal: D – Depuração; Otim – Otimização automatizada; Rede – Monitoramento de Rede; e V – Visualização de aplicações.

[2] Funcionalidade limitada.

[3] Nprof gera arquivos de traços que podem ser utilizados para pós-processamento, porém não possui um módulo para tal pós-processamento.

4.5 Contribuições para Nprof

Nprof propõe-se a ser uma ferramenta de monitoramento de sistemas distribuídos em Java, e possui pontos em comum com diversas das ferramentas analisadas.

JProfiler e, principalmente, OptimizeIt serviram de motivação para a proposta de Nprof. Entretanto, Nprof de início se voltou para cobrir também o monitoramento de aplicações distribuídas, o que não é o foco de tais ferramentas. JFluid também serviu de forte inspiração para o projeto da ferramenta, e os conceitos de instrumentação de *bytecode* e *hotswapping* foram aproveitados no presente trabalho. JFluid, contudo, monitora somente aplicações isoladas. AppPerfect é outra ferramenta semelhante a OptimizeIt e JProfiler. Hprof, por sua vez, restringe-se à coleta de dados sobre gasto de tempo de UCP e de memória.

Javavis, JaVis e Jacot são ferramentas de visualização de aplicações Java, e permitiram verificar possibilidades de visualização de programas em execução, mesmo que todas elas estejam voltadas para aplicações isoladas. Jinsight é também classificada como uma ferramenta de visualização para Java, porém seu foco apresenta maiores semelhanças com o de *profilers* tradicionais.

JConsole e VisualGC são ferramentas interessantes por utilizarem recursos bastante recentes da plataforma Java, e Nprof utiliza-se dos mesmos instrumentos utilizados por JConsole.

CLRProfiler, VTune, Gprof e DCPI permitiram a comparação entre *profilers* para aplicativos Java e para aplicativos escritos em outras linguagens, sejam elas para código intermediário (dotNet, como no caso de CLRProfiler) ou para código nativo. VTune e Gprof, entretanto, coletam informações apenas sobre tempo de execução de trechos da aplicação e montam grafos de chamada entre funções. Gprof, ao contrário de VTune, necessita recompilar a aplicação-alvo com a ligação de uma biblioteca de *profiling*. DCPI demonstra a possibilidade de *profiling* de sistemas em produção por longos períodos de tempo, o que pode ser interessante para depuração de aplicações em que ocorram situações excepcionais com baixa freqüência. CLRProfiler foca principalmente a análise de alocação de objetos e o monitoramento do coletor de lixo.

AOP Profiler e Atlassian Profiling demonstram o uso de POA para a área de *profiling* e, mesmo sendo voltados para aplicações isoladas, permitem perceber a facilidade do uso dessa nova abordagem. Uma proposta interessante é de estudar o desenvolvimento de uma ferramenta de *profiling* mais complexa pela composição de diferentes aspectos específicos.

DynaMetricO é uma ferramenta que atua como um *profiler* e existem planos para utilizá-la para a otimização de compiladores, porém o seu uso atual é de classificação de programas de acordo com um conjunto de métricas elaboradas pelo próprio projeto. A definição de métricas dinâmicas, ou seja, para programas em execução, foi de bastante interesse para o presente trabalho, mesmo considerando que Nprof utiliza um conjunto de métricas bastante diferente.

VisuSniff e Analyzer são sistemas muito interessantes para monitoramento de interfaces de rede. Visto que Nprof é uma feramenta voltada para sistemas distribuídos, a captura de mensagens enviadas e recebidas é de interesse do presente trabalho. Entretanto, VisuSniff e Analyzer permitem o monitoramento de mensagens por meio de apenas uma interface de rede, o que não é suficiente para o monitoramento de uma aplicação distribuída. Além disso, esses analisadores de rede capturam todos dados que

trafegam pela interface de rede analisada, e a proposta de Nprof é a de coletar apenas os dados referentes aos fluxos que compõem uma aplicação distribuída específica.

Hy+ é uma ferramenta para depuração pós-morte de aplicações distribuídas, e coleta dados para, entre outras finalidades, reconstituir a ordem parcial da computação distribuída. Tais informações são bastante relevantes para a depuração de aplicações distribuídas, mas para tanto Hy+ se utiliza de uma linguagem experimental para a coleta de dados. Nprof tem o propósito de monitorar programas existentes, sem necessidade de alteração destes, e tem o propósito adicional de permitir monitoramento *online* e não *offline* (pós-morte).

Javiz é, assim como Nprof, um sistema para monitoramento de aplicações distribuídas em Java. Entretanto, este sistema tem o propósito específico de construir um grafo global de chamadas para aplicações que utilizam RMI. Nprof tem o propósito de coletar dados de qualquer tipo de comunicação entre nodos de uma aplicação distribuída, e de correlacionar tais dados com outros tipos de dados de monitoramento. Além disso, Nprof tem o propósito de utilizar somente bibliotecas Java padrão, e Javiz utiliza a instrumentação da JVM para a coleta de dados.

JRastro é uma ferramenta que possui finalidades semelhantes a Nprof, coletando informações sobre *threads* e memória. Além disso, permite a coleta de dados de conexões de rede. Entretanto, ao contrário de Nprof, JRastro se restringe a coleta de informações referentes ao protocolo RMI.

TAU Profiler, para mpiJava, é uma ferramenta que possui propósitos bastante semelhantes aos de Nprof. Ambas propõem o monitoramento de aplicações Java distribuídas por meio de interceptação de chamadas de envio e recebimento de mensagens. Entretanto, o *profiler* TAU visa especificamente aplicações voltadas a ambientes de alto desempenho que utilizem o ambiente MPI de troca de mensagens. Além disso, o mecanismo de interceptação de mensagens também é diferente, visto que as primitivas MPI analisadas pelo *profiler* TAU se encontram em bibliotecas nativas.

5 A FERRAMENTA NPROF

5.1 Apresentação

A ferramenta Nprof, em relação a aplicações Java distribuídas, propõe-se a: (i) monitorar os nodos da aplicação-alvo isoladamente; (ii) e monitorar a comunicação entre os diferentes nodos de tal aplicação. Ao utilizar APIs padrão e técnicas como a de instrumentação de *bytecode*, a ferramenta Nprof é capaz de monitorar aplicações Java sem a necessidade de modificação ou acesso ao código-fonte da aplicação monitorada.

É importante notar que a ferramenta possui dois modos de conectar-se a aplicações-alvo: ela pode tanto iniciar a aplicação-alvo localmente ou conectar-se a uma aplicação já em execução. Este comportamento permite o uso da ferramenta para monitorar aplicações distribuídas de forma *online*, caso a sobrecarga devida ao módulo remoto não tenha um impacto excessivo sobre o desempenho da aplicação-alvo.

Em razão da adoção de APIs padrão Java, a ferramenta Nprof é independente de plataforma. Não é necessário o uso de nenhuma biblioteca de código nativo ou máquina virtual modificada. Além da conveniência e eficiência do uso de uma plataforma neutra, permitindo uma migração transparente para diferentes arquiteturas de computador, a ferramenta pode ser ativada e desativada dinamicamente, o que a torna adequada para programas que rodam em máquinas remotas. Isso é realizado pelo mecanismo de *hotswapping*, para o caso de classes instrumentadas, e por ativação e desativação dinâmica dos monitores.

5.2 Arquitetura da ferramenta Nprof

Para o funcionamento da ferramenta assume-se o uso de, no mínimo, duas máquinas virtuais Java: uma para a aplicação-alvo e outra para a ferramenta em si. Uma aplicação distribuída é aqui considerada como um conjunto de aplicações separadas, que executam em máquinas virtuais independentes, e que se comunicam via mensagens que trafegam em uma rede. A ferramenta Nprof consiste em um módulo central (Nprof Núcleo), um módulo de comunicação (conjunto de Controladores de Alvo), um módulo remoto (Nprof Remoto) e diversos módulos de visualização, como mostra a figura 5.1. Estes módulos serão detalhados nas próximas subseções.

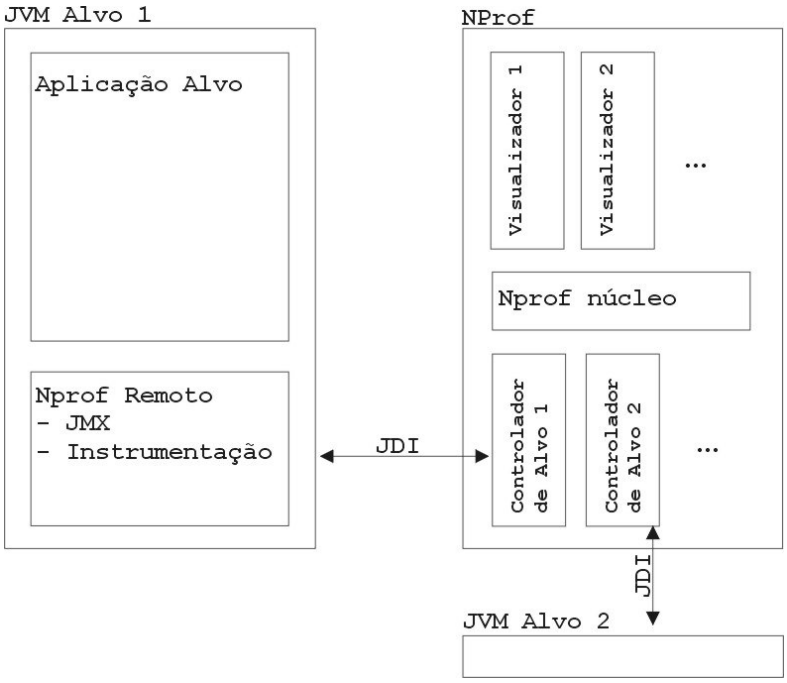


Figura 5.1: Arquitetura da ferramenta Nprof

Note-se que as JVMs podem estar executando na mesma máquina ou em máquinas distintas, que cada JVM pode executar várias *threads* concorrentemente, e que o módulo central pode controlar várias aplicações-alvo.

5.2.1 Módulo central

O módulo central (Nprof Núcleo) é responsável pela organização, sincronização e controle das informações coletadas. Ele também é responsável pela organização da interface gráfica (seção 5.6) e pelo redirecionamento dos diversos tipos de dados para seus respectivos visualizadores.

Para inicializar a ferramenta, o módulo central utiliza um arquivo de configuração que possui os principais parâmetros para a configuração dos diversos componentes. A figura 5.2 mostra um exemplo deste tipo de arquivo.

```
nprof.samplingrate=100
nprof.vm.suspend=false

nprof.classpath=c:\\proj\\nprof\\build\\classes
nprof.targetmoduleclasspath=c:\\proj\\nprof\\dist\\nprof-instrument.jar
nprof.mainclass=rmitest.RegistryInitializer
nprof.vmparams=

nprof.monitor.jmxmemorymonitor=true
nprof.monitor.threadmonitor=true
nprof.monitor.jmxruntimemonitor=true
nprof.monitor.socketmonitor=true
nprof.monitor.objectmonitor=false
nprof.monitor.gccompmonitor=true

nprof.monitor.jmxmemorymonitor.mult=2
nprof.monitor.threadmonitor.mult=1
nprof.monitor.jmxruntimemonitor.mult=5000
nprof.monitor.socketmonitor.mult=2
nprof.monitor.objectmonitor.mult=2
nprof.monitor.gccompmonitor.mult=5

nprof.monitor.outputstream=true
nprof.monitor.errorstream=true

nprof.event.classprepare=true
nprof.event.exception=true

# filtros para monitor de carga de classe
#
nprof.event.classprepare.classfilter=nprof.*


# filtros para monitor de excecoes
#
nprof.event.exception.exceptionlocation=java.lang.ClassLoader,nprof.*
nprof.event.exception.exceptionlocation=java.lang.Class
nprof.event.exception.classfilter=java.lang.RuntimeException
nprof.event.exception.excludeclass=java.lang.NoSuchMethodException
nprof.event.exception.notifycaught=true
nprof.event.exception.notifyuncaught=true
```

Figura 5.2: Arquivo de configuração do Nprof

Dentre as opções contidas no arquivo de configuração (figura 5.2), pode-se destacar:

- Taxa básica de amostragem (`nprof.samplingrate`), em milisegundos. Tal taxa representa o intervalo de tempo entre duas ativações sucessivas dos monitores;
- Parâmetros de inicialização da aplicação-alvo (`nprof.mainclass`, `nprof.classpath` e `nprof.vmparams`);
- Seleção dos monitores a serem ativados inicialmente (`nprof.monitor.`*`NomeDoMonitor`*`=[true|false]`);
- Seleção do multiplicador a ser aplicado ao tempo entre amostragens de cada monitor (`nprof.monitor.`*`NomeDoMonitor`*`.mult`). Com esta funcionalidade, permite-se que diferentes monitores possuam diferentes taxas de amostragem (por exemplo, enquanto o monitor de *threads* tem taxa de 100ms, o monitor de memória pode ter uma taxa de 200ms);

- Seleção de ativação dos monitores ativados a partir de eventos do módulo remoto (`nprof.event.`*`NomeDoMonitor`*`=[true|false]`);
- Parâmetros de filtragem dos eventos a serem gerados pela aplicação-alvo (`nprof.event.`*`NomeDoMonitor.NomeDoFiltro`*).

O módulo central também é responsável pelas funcionalidades de console (JAIN, 1991) da ferramenta, ou seja, as funcionalidades que permitem que o usuário interaja com a aplicação-alvo. Uma de suas funções oferece a possibilidade de suspender e reativar a aplicação-alvo, outra permite suspender e reativar *threads* da aplicação-alvo separadamente, e uma terceira opção permite solicitar a suspensão simultânea de todas as aplicações-alvo monitoradas em determinado momento.

5.2.2 Módulo de controle de alvo

O módulo de controle de alvo controla a conexão à JVM alvo e administra o tráfego de informação entre a aplicação-alvo e o Nprof Núcleo. Além disso, o módulo é responsável por coletar alguns tipos de dados, como o de estado de *threads*. Outros tipos de dados, que necessitam de invocação remota de métodos, são também diretamente coletados por esse módulo, como, por exemplo, a pilha de execução que contém informações sobre exceções lançadas na aplicação-alvo. A ferramenta Nprof pode conter, em determinado momento, diversos controladores de alvo ativos ao mesmo tempo (*Controlador de Alvo,* figura 5.1), uma vez que o Nprof pode estar conectado a mais de uma aplicação-alvo, e cada controlador administra a comunicação com uma única aplicação-alvo.

A conexão entre a ferramenta e a aplicação-alvo é realizada por meio do protocolo JDWP (Java Debug Wire Protocol), o protocolo padrão de depuração Java da Sun Microsystems (seção 3.1.2.3). A ferramenta utiliza a biblioteca JDI (Java Debug Interface) para realizar a transmissão dos dados coletados pelo módulo remoto (Nprof Remoto, figura 5.1). Esta biblioteca permite a captura de exceções remotas, mesmo quando essas exceções tenham sido capturadas pela própria aplicação-alvo.

5.2.3 Módulo remoto

O módulo remoto executa na mesma JVM em que se encontra a aplicação-alvo. Ele é responsável pela coleta da maior parte dos tipos de dados, incluindo uso de memória e informações sobre o ambiente de execução. A instrumentação das classes que criam e mantêm as conexões de rede também é realizada por esse módulo. Para a captura de alguns tipos de dados, como os de ambiente de execução e memória, a nova API de Administração e Monitoramento (*Monitoring and Management*), incluída na plataforma Java versão 5.0, foi utilizada.

Devido à arquitetura de carga de classes da plataforma Java (LIANG; BRACHA, 1998), as classes do módulo remoto foram empacotadas em dois grupos: o pacote `nprof-target` e o pacote `nprof-instrumentation`. Este último contém as classes usadas para instrumentar a própria API Java. Portanto, este pacote necessita ser carregado pelo carregador de classes *Bootstrap,* de modo que esteja presente no momento em que as classes da API Java forem carregadas. O pacote `nprof-target` contém as classes que são necessárias para realizar atividades na máquina alvo, como a criação de uma *thread* para captura de informações da API de Management, e deve, como em aplicações quaisquer, ser carregado pelo carregador de classes padrão do sistema.

O módulo remoto funciona basicamente da seguinte maneira: ao invés de inicializar uma máquina virtual com a classe principal da aplicação-alvo, inicia-se uma máquina virtual com a classe `nprof.target.TargetStarter`. Esta classe, por sua vez, inicia a *thread* `TargetMonitor`, que é responsável por ativar os monitores baseados em JMX e "serializar" os dados coletados por estes. A classe `TargetStarter`, então, inicia a aplicação-alvo de fato. Os dados capturados a partir da *thread* `TargetMonitor` são, de acordo com a taxa de amostragem definida no arquivo de configuração, coletados pelos monitores que se encontram na JVM do Nprof Núcleo, e esta coleta é realizada por meio da JDI.

Idealmente, o módulo remoto deve ser tão pequeno quanto possível, para minimizar a sobrecarga causada à aplicação-alvo e, como conseqüência, provocar resultados imprecisos. Além da sobrecarga realizada pelas tarefas realizadas pelo módulo remoto, a instrumentação de classes da API Java gera também sobrecarga. Como vários tipos de dados não podem ser capturados de forma direta utilizando alguma API padrão Java, o módulo remoto utiliza uma *thread* para monitorar e capturar informações de administração (*management*), e realiza também o tratamento inicial e empacotamento dos dados capturados via instrumentação de *bytecode*. Entretanto, com o objetivo de reduzir a sobrecarga causada pelo módulo remoto, são utilizados diferentes monitores especializados para cada categoria de coleta de dados. Estes monitores podem ser ativados somente quando necessário, e também desativados tão logo termine a captura de dados.

5.2.4 Monitores distribuídos

O projeto de Nprof permite que a informação capturada por um monitor possa ser tratada por um conjunto de diferentes componentes de análise e apresentação. Por exemplo, as quantidades de *bytes* enviados e recebidos por meio de *sockets* podem ser mostradas como gráficos de apenas uma aplicação-alvo, de forma agregada ou serem salvas em um arquivo texto para análise posterior.

Para permitir que diversos analisadores recebam dados de um único observador, um modelo de envio e recebimento de eventos foi adotado, seguindo o padrão de projeto *Observer* (GAMMA et al., 1995). Cada componente de análise registra-se no coletor agregado requisitando certo tipo de dados coletados e pode requisitar dados de apenas uma aplicação-alvo ou de todas, para análise de dados agregados. Cada tipo de dado coletado é armazenado em um objeto de uma classe apropriada, segundo a hierarquia de classes mostrada na figura 5.3.

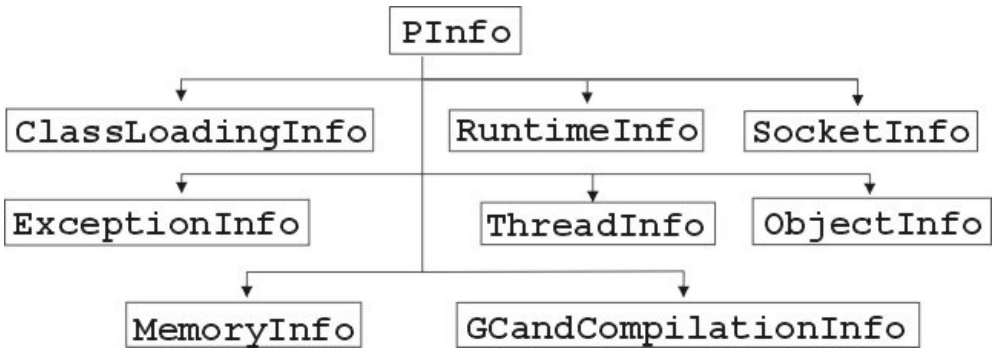


Figura 5.3: Diagrama de classes de registro de dados

Note-se que cada tipo especializa o tipo básico `Pinfo`. Os outros tipos são:

- *ClassLoadingInfo*: contém dados sobre classes carregadas;
- *ExceptionInfo*: contém dados sobre exceções, capturadas pela aplicação-alvo ou não;
- *MemoryInfo*: contém dados sobre o uso de memória *heap* e não *heap*;
- *ObjectInfo*: contém dados sobre o número de objetos instanciados para cada classe;
- *RuntimeInfo*: contém dados sobre o ambiente de execução Java, sistema operacional e máquina da aplicação-alvo;
- *SocketInfo*: contém dados sobre o uso de conexões *socket*;
- *ThreadInfo*: contém dados sobre estados de *threads*.;
- *GCandCompilationInfo*: contém dados sobre coleta de lixo e compilação JIT.

5.3 Monitores especializados

Basicamente, cada monitor da ferramenta Nprof é responsável por coletar um dos tipos de dados descritos na seção anterior.

Esses monitores usam tecnologias que variam de JMX (Java Management Extensions) à instrumentação de *bytecode*, e eles adotam técnicas de coleta por amostragem e traços. No primeiro caso, a ferramenta inspeciona a aplicação-alvo a intervalos regulares de tempo, enquanto no segundo ela atua sob demanda de eventos gerados pela aplicação. Os monitores podem utilizar apenas uma tecnologia de implementação e uma técnica de amostragem ou uma combinação delas. A seguir, são detalhadas as tecnologias utilizadas.

5.3.1 Monitores baseados em JDI

Todos os monitores utilizam a biblioteca JDI, direta ou indiretamente, para atuarem junto com a aplicação-alvo. Porém, faz-se uma distinção entre os monitores que utilizam somente a biblioteca JDI (pacotes como `com.sun.jdi`) e os que utilizam outras tecnologias para captura de dados. Os monitores JDI podem requisitar dados ou apenas receber eventos originados a partir da aplicação-alvo. O monitor de *thread* é um exemplo de monitor JDI que atua por amostragem: ele inspeciona os estados das *threads* em intervalos regulares de tempo.

O monitor de exceções é um monitor de recebimento de eventos JDI que recebe dados sobre exceções ocorridas na aplicação-alvo, tenham elas sido capturadas pela aplicação-alvo ou não. Este monitor permite ao usuário da ferramenta Nprof solicitar os dados referentes à pilha de execução existente no momento em que a exceção foi gerada. O monitor de carga de classes baseia-se também no recebimento de eventos gerados pela aplicação-alvo. Tanto o monitor de carga de classes quanto o de exceções possuem opções de configuração para filtrar eventos de carga de classes e de exceções, respectivamente.

5.3.2 Monitores baseados em JMX (Java Management)

Monitores JMX baseiam-se na nova versão da API Java (versão 5) para a coleta de dados. Esta nova API (pacote `java.lang.management`) provê diversas informações sobre uma JVM em execução, como informações de memória, *threads* e ambiente.

O monitor de memória é um monitor baseado em JMX que utiliza amostragem para capturar dados sobre o uso de memória *heap* e *não heap*. Há também o monitor de ambiente de execução, que captura várias informações sobre o sistema alvo como o número de processadores, nome e versão do sistema operacional, *classpath* do ambiente Java, diretórios de usuário e temporários, argumentos de inicialização da aplicação-alvo, entre outros.

Além desses, foi desenvolvido um monitor para a coleta de informações de compilação e coleta de lixo. Tal monitor coleta dados sobre o tempo gasto pela máquina virtual na coleta de lixo e na compilação em tempo de execução (compilação JIT).

5.3.3 Monitores baseados em instrumentação

Monitores baseados em instrumentação de *bytecode* (*Bytecode Instrumentation* - BCI) são aqueles que usam alteração de *bytecode* para coletar dados sobre chamadas de métodos e seus parâmetros. Atualmente existem dois monitores de instrumentação: o de objetos e o de *sockets*.

O de objetos é usado para coletar o número de instanciações de cada classe. Baseando-se no modelo proposto por Shirazi (2000), esse monitor instrumenta o construtor da classe `Object` para contar o número de objetos instanciados para cada classe, uma vez que a instanciação de qualquer objeto chama, em algum momento, o construtor da classe `Object`. Entretanto este método não contabiliza instanciações de *arrays*.

O monitor de *sockets* é basicamente um monitor de instrumentação. Ele instrumenta um conjunto de métodos em algumas classes relacionadas às conexões de *socket* para coletar informações sobre endereços de rede, *bytes* trafegados, associação de portas e outros. O monitor de *sockets* permite a coleta de dados tanto UDP quanto TCP. Outros detalhes sobre a instrumentação de *bytecode* encontram-se na seção 5.4.

Além de ser um monitor de instrumentação de *bytecode*, o monitor de conexões *socket* utiliza-se da técnica de amostragem para coletar dados sobre o tamanho de *buffer* de leitura nos *sockets* ao chamar o método `available()` da classe `SocketInputStream`.

5.3.4 Tabela de Monitores

A tabela 5.1 resume as técnicas e tecnologias adotadas em cada um dos monitores implementados.

Tabela 5.1: Monitores implementados

Monitor	**Técnica**	**Tecnologia**
Ambiente de Execução	Amostragem	JMX
Carga de Classes	Traços	JDI
Compilação e Coleta de Lixo	Amostragem	JMX
Exceções	Traços	JDI
Memória	Amostragem	JMX
Objetos	Traços	BCI
Sockets	Traços e Amostragem	BCI
Threads	Amostragem	JDI

5.4 Instrumentação de *bytecode*

Como mencionado, para coletar informações sobre conexões *socket,* é necessária a instrumentação de algumas classes da API Java. Essa técnica é também utilizada pelo *profiler* JFluid (DMITRIEV, 2004) para a captura de outros tipos de informação; o autor defende que a instrumentação de *bytecode* (BCI) tem a vantagem de ser flexível, porque virtualmente qualquer tipo de informação pode ser capturada com essa técnica. As classes instrumentadas pelo monitor de *sockets* (pacote `java.net`) são: `Socket`, `ServerSocket`, `DatagramSocket`, `SocketOutputStream`, `SocketInputStream`, `SocketImpl`, `SocksSocketImpl` e `PlainSocketImpl`, conforme resumido na tabela 5.2. Ao instrumentar essas classes, a ferramenta pode coletar informações durante as ações de recebimento e envio de mensagens e estabelecimento de conexões. Note-se que as classes `SocksSocketImpl` e `PlainSocketImpl` não são classes públicas da API Java, no entanto a instrumentação de alguns de seus métodos é útil para a interceptação de informações sobre conexões *socket*.

A ferramenta Nprof utiliza o mecanismo de *hotswapping*, disponibilizado pela JDI, para instrumentar dinamicamente classes que se encontram em uma JVM remota (não a que é utilizada pelo Nprof Núcleo). Dessa maneira é possível diminuir a sobrecarga na máquina alvo, uma vez que o código instrumentado só está ativo quando for realizada a coleta de dados. Uma vez terminada a coleta, as classes instrumentadas podem ter seus códigos-objeto originais restabelecidos. As aplicações-alvo devem, entretanto, ser inicializadas com parâmetros especiais para ativação do modo de depuração que, por si só, causam sobrecarga, mesmo não excessiva. Em seu trabalho, Sato (2003) verifica uma sobrecarga inferior a 5%.

Tabela 5.2: Classes instrumentadas pelo monitor de *sockets*

Classe	**Método**	**Descrição**
Socket	*construtor*	Cria o *socket*
	Bind	Amarra *socket* a uma porta local
	connect	Conecta a outro *socket*
	Close	Fecha *socket*
SocketOutputStream	Write	Escreve *bytes* no *socket*
SocketInputStream	Read	Lê *bytes* do *socket*
SocketImpl	Bind	Amarra *socket* a uma porta local
	accept	Escuta por conexão *socket*
ServerSocket	*construtor*	Cria *socket* servidor
	accept	Permite novas conexões externas
	bind	Amarra *socket* a uma porta local
PlainSocketImpl	bind	Amarra *socket* a uma porta local
	connect	Conecta a outro *socket*
	close	Fecha *socket*
SocksSocketImpl	connect	Conecta a outro *socket*
	close	Fecha *socket*
DatagramSocket	bind	Amarra *socket* a uma porta local
	send	Envia um pacote UDP
	receive	Recebe um pacote UDP

A informação coletada é armazenada para posterior leitura pelo módulo de comunicação do Nprof. Para a instrumentação da API Java foi utilizada a ferramenta Javassist (CHIBA; NISHIZAWA, 2003). Uma vez que a instrumentação realizada nessas classes consiste apenas na inserção de chamada de métodos dentro do corpo dos métodos já existentes, não há a criação ou remoção de novos métodos ou campos ou alteração da assinatura de métodos ou campos. Assim, há compatibilidade total entre as versões original e instrumentada das classes.

5.5 Sincronização de relógios

A fim de permitir a ordenação dos eventos capturados pelas aplicações-alvo com eventos ocorridos na máquina em que o monitor executa e também a ordenação de eventos entre as diferentes aplicações monitoradas, desenvolveu-se um módulo para a sincronização de relógios entre a aplicação-alvo (que contém o módulo Nprof Remoto) e a ferramenta de monitoramento (Nprof Núcleo). Na verdade, os relógios existentes nas máquinas não são alterados (e, portanto, não há uma sincronização de fato), apenas é calculada a diferença entre os relógios das máquinas em questão. Tal estratégia é conhecida como sincronização *offline* (ASHTON, 1987 apud TRINDADE et al., 2006) e é também utilizada em outros trabalhos do grupo de tolerância à falhas (TRINDADE et al., 2006).

Uma vez que se conheça, para todas as aplicações-alvo que executam simultaneamente, a diferença entre o relógio da aplicação-alvo e do monitor, pode-se utilizar o relógio do monitor como referência para o cálculo da diferença entre relógios de quaisquer aplicações-alvo alvo que estejam executando simultaneamente.

Entretanto, devido à proposta de apenas utilizar-se APIs padrão, foram adotadas, por conseguinte, apenas protocolos utilizados por estas APIs (no caso o JDWP). Sendo assim, a API JDI foi usada para uma chamada remota a um método que retorna o *timestamp* da máquina monitorada. Para a realização do cálculo da diferença entre relógios é efetuada uma consulta ao relógio da máquina monitora antes e após a chamada remota que retorna o valor do relógio da máquina remota. Após o retorno, determina-se o tempo decorrido para a chamada do método remoto. Partindo do pressuposto de que o tempo da solicitação do relógio remoto e o tempo de retorno da informação do método remoto são semelhantes, e que o tempo necessário para a captura do relógio local é desprezível, diminui-se da diferença entre o relógio remoto e o do monitor a metade do tempo necessário à chamada remota.

Embora o método aqui utilizado não seja o ideal, por fornecer resultados aproximados, acredita-se ser uma boa alternativa a partir dos pressupostos definidos e sem agregar grande complexidade ao sistema.

5.6 Interface gráfica

A ferramenta Nprof possui uma interface gráfica para a interação com o usuário. Tal interface contém botões para acionar funcionalidades de inicialização de aplicações a serem monitoradas e de conexão com aplicações já iniciadas. A interface pode conter também alguns visualizadores, que são ativados de acordo com os tipos de dados coletados na máquina alvo.

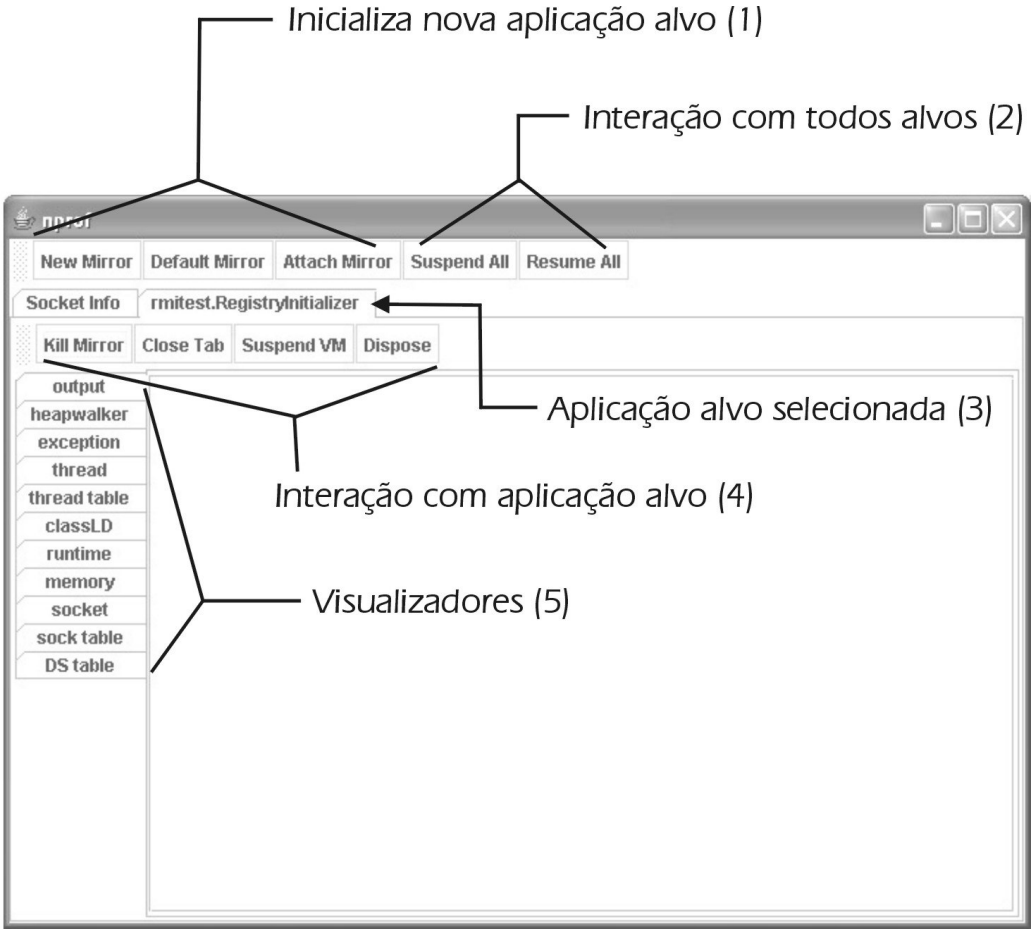


Figura 5.4: Interface gráfica da ferramenta Nprof

A figura 5.4 mostra a interface principal da ferramenta. Esta interface está organizada da seguinte forma: os botões em (1) inicializam ou efetuam a conexão com uma nova aplicação-alvo; os botões em (2) permitem a suspensão e prosseguimento de todas as aplicações-alvo ativas; as abas em (3) indicam as aplicações-alvo monitoradas; (4) contém botões para interação com a aplicação-alvo selecionada (suspender, prosseguir, terminar aplicação e terminar monitoramento) e (5) mostra os visualizadores disponíveis para a aplicação selecionada.

Devido ao uso da JDI, é possível a conexão com uma aplicação-alvo de duas formas: inicializando localmente uma nova aplicação ou conectando-se a uma aplicação já existente, podendo esta estar localizada na mesma máquina ou em uma máquina remota, desde que a aplicação tenha sido inicializada com parâmetros específicos. A figura 5.5 mostra as caixas de diálogo (*dialog box*) para a inicialização de uma aplicação-alvo ou conexão com uma aplicação já existente.

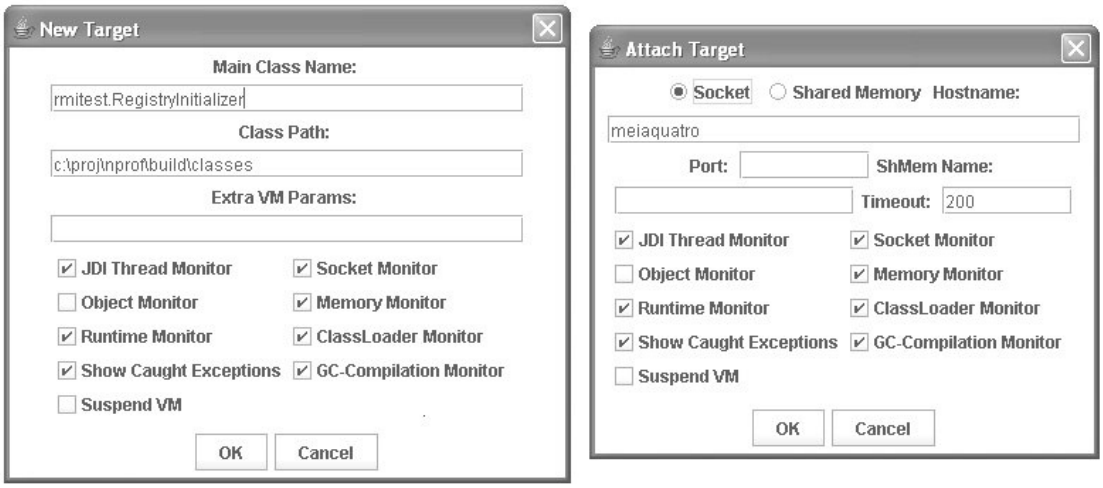


Figura 5.5: Diálogos de inicialização e de conexão com aplicação-alvo

A parte superior dos diálogos da figura 5.5 contém as definições necessárias para a conexão com a aplicação-alvo. A parte inferior é comum a ambos e permite ao usuário selecionar os monitores que ele deseja ativar no monitoramento da aplicação-alvo.

As seções seguintes descrevem as funcionalidades dos diversos tipos de visualizadores.

5.6.1 Visualizador de *Threads*

O visualizador de *threads* possui dois modos de visualização: um em forma de gráfico (figura 5.6), que mostra a evolução dos estados das *threads* conforme uma linha de tempo, e outro em tabela (figura 5.7), que contém informações sobre o estado atual das *threads*. Na realidade eles podem ser considerados dois visualizadores independentes, visto que cada um deles recebe separadamente os dados do módulo Nprof Núcleo. As figuras apresentadas refletem a execução de uma aplicação de demonstração da API de componentes gráficos da plataforma Java (`java2d.Java2Demo`).

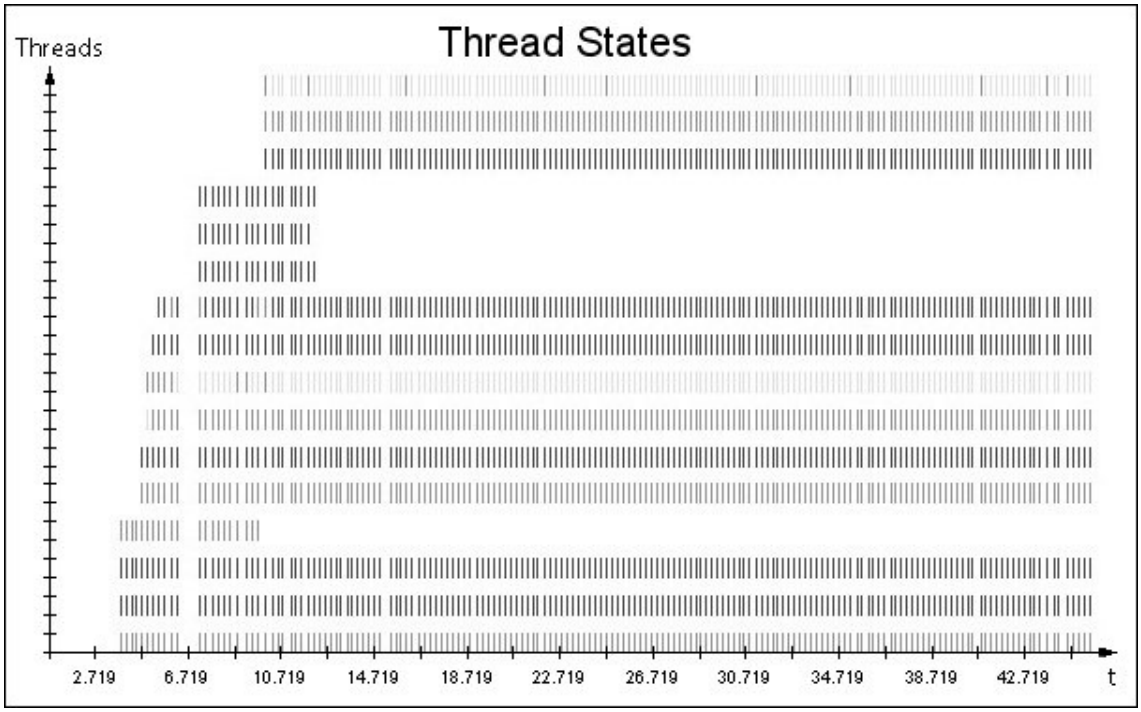


Figura 5.6: Gráfico de estados de threads

Na figura 5.6, cada linha horizontal representa uma *thread*, e a linha de tempo está na escala de milisegundos. Barras verdes denotam *threads* em execução (estado *runable*); barras amarelas, o estado de adormecimento (*sleeping*); barras vermelhas, estado de espera (*waiting*). *Threads* esperando para entrar em monitor (*blocked*) aparecem em rosa.

Já a figura 5.7 mostra uma captura de tela da ferramenta Nprof com a tabela de *threads*, que contém o identificador da *thread*, nome, estado atual, tamanho da pilha de chamadas (FrameCount), número de suspensões executadas (SuspendCount) e botões para o usuário suspender e resumir a execução das *threads* (colunas suspend e resume).

ID	Name	State	FrameCount	SuspendCount	suspend	resume
19	Image Fetcher 3	!! DEAD !!	-1	0	suspend	resume
20	Image Fetcher 1	!! DEAD !!	-1	0	suspend	resume
21	Image Fetcher 2	!! DEAD !!	-1	0	suspend	resume
22	Intro	!! DEAD !!	-1	0	suspend	resume
25	Intro	!! DEAD !!	-1	0	suspend	resume
7	Signal Dispatcher	RUNNING	-1	0	suspend	resume
11	AWT-Windows	RUNNING	-1	0	suspend	resume
14	TargetRemoteInvoke	RUNNING	2	1	suspend	resume
23	DestroyJavaVM	RUNNING	-1	0	suspend	resume
15	TargetMonitor	SLEEPING	-1	0	suspend	resume
26	MemoryMonitor	SLEEPING	-1	0	suspend	resume
27	PerformanceMonitor	SLEEPING	-1	0	suspend	resume
28	Arcs_Curves.Arcs Demo	RUNNING	3	1	suspend	resume
29	Arcs_Curves.BezierAnim Demo	RUNNING	3	1	suspend	resume
30	Arcs_Curves.Ellipses Demo	RUNNING	3	1	suspend	resume
2	Reference Handler	WAIT	-1	0	suspend	resume
3	Finalizer	WAIT	-1	0	suspend	resume
16	Java2D Disposer	WAIT	-1	0	suspend	resume
17	AWT-Shutdown	WAIT	-1	0	suspend	resume
18	AWT-EventQueue-0	WAIT	-1	0	suspend	resume
24	TimerQueue	WAIT	-1	0	suspend	resume
32	ToggleIcon	!! DEAD !!	-1			

Figura 5.7: Informações sobre *threads* em forma de tabela

O tamanho da pilha de execução (coluna FrameCount) representa a quantidade de chamadas a métodos realizadas pela *thread*, porém esta informação só está disponível para *threads* que foram suspensas por meio da JDI, ou seja, quando o valor da coluna SuspendCount é maior que um. Assim, quando a *thread* não está suspensa (SuspendCount = 0) a coluna FrameCount assume valor "–1" (informação não disponível).

Ao utilizar a aplicação `Java2Demo` é possível ver o efeito da suspensão de determinadas *threads* (`Arcs_Curves.ArcsDemo`, `Arcs_Curves.BezierAnimDemo` e `Arcs_Curves.EllipsesDemo`), que ocasionaram o congelamento de três animações que ocorriam simultaneamente. Note-se que as três linhas que identificam tais *threads* indicam que a contagem de suspensão (coluna SuspendCount) das mesmas *threads* é positiva.

5.6.2 Visualizador de Memória

O visualizador de memória apresenta a quantidade de memória utilizada ao longo do tempo. Para tanto, é diferenciado o uso de memória de alocação de objetos (*heap*) e memória permanente (não-*heap*), onde são armazenados, por exemplo, os códigos executáveis das classes (*bytecodes*). A figura 5.8 mostra um exemplo de visualização de memória.

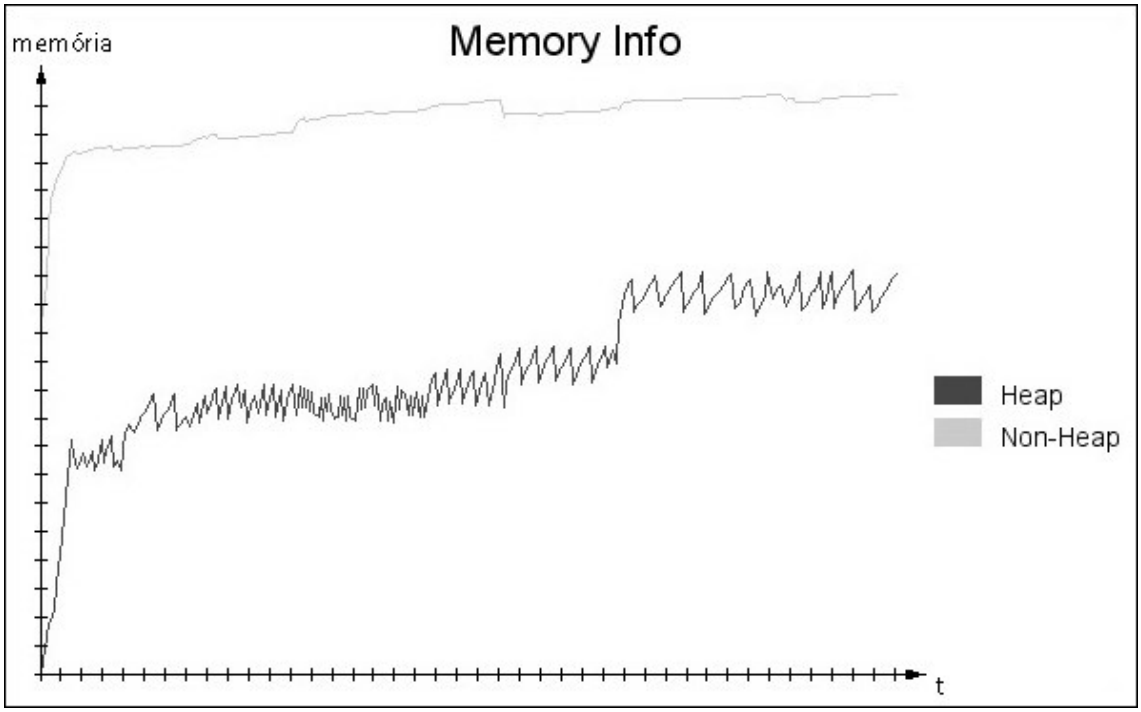
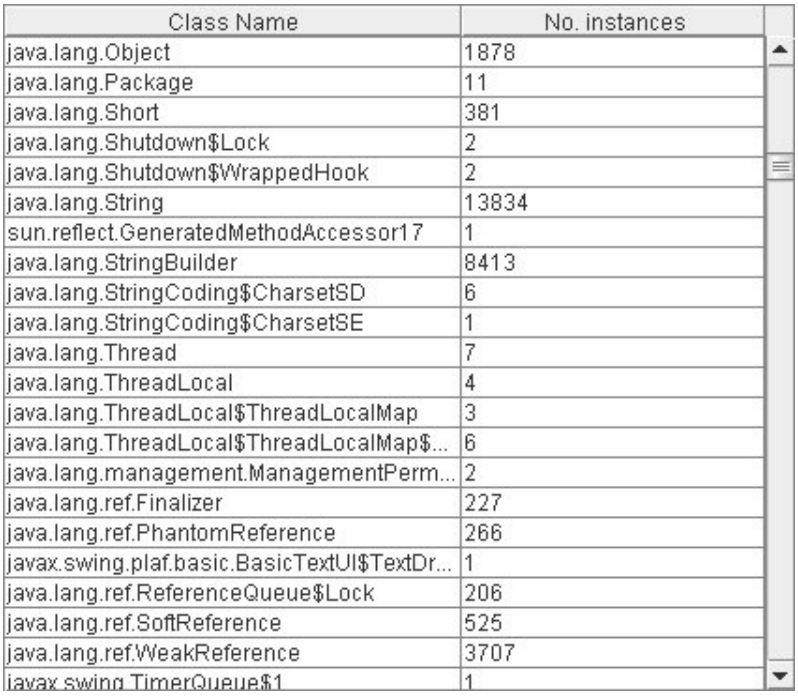


Figura 5.8: Visualizador de uso de memória

5.6.3 Visualizador de Objetos

O visualizador de objetos baseia-se nas informações obtidas pelo monitor de objetos, que utiliza instrumentação de *bytecode* para a coleta do número de instanciações realizadas para cada classe Java. Note-se que a instanciação de *arrays*, bem como a alocação de variáveis primitivas da linguagem Java não são contabilizadas por tal monitor. A figura 5.9 apresenta uma captura de tela de um exemplo de uso do visualizador de objetos para a execução da aplicação `Java2Demo`.



Class Name	No. instances
java.lang.Object	1878
java.lang.Package	11
java.lang.Short	381
java.lang.Shutdown$Lock	2
java.lang.Shutdown$WrappedHook	2
java.lang.String	13834
sun.reflect.GeneratedMethodAccessor17	1
java.lang.StringBuilder	8413
java.lang.StringCoding$CharsetSD	6
java.lang.StringCoding$CharsetSE	1
java.lang.Thread	7
java.lang.ThreadLocal	4
java.lang.ThreadLocal$ThreadLocalMap	3
java.lang.ThreadLocal$ThreadLocalMap$...	6
java.lang.management.ManagementPerm...	2
java.lang.ref.Finalizer	227
java.lang.ref.PhantomReference	266
javax.swing.plaf.basic.BasicTextUI$TextDr...	1
java.lang.ref.ReferenceQueue$Lock	206
java.lang.ref.SoftReference	525
java.lang.ref.WeakReference	3707
javax.swing.TimerQueue$1	1

Figura 5.9: Visualizador de objetos

5.6.4 Visualizador de Ambiente de Execução

O visualizador de ambiente de execução baseia-se nas informações coletadas a respeito dos parâmetros de configuração do ambiente de execução Java e do sistema operacional sobre o qual a aplicação executa. Este visualizador mostra as informações capturadas por meio da API de Administração e Monitoramento e por consultas às propriedades de sistema (`System.getProperties()`). A figura 5.10 mostra um exemplo das informações capturadas por este monitor, quando executada a aplicação Java2Demo.

Name	Value
Architecture	x86
BootClassPath	C:\apps\Java\jre1.5.0\lib\rt.jar;C:\apps\Java\jre1.5.0\lib\i18n.jar;C:\apps\Java\...
ClassPath	c:\proj\nprof\build\classes;C:\apps\Java\jdk1.5.0\demo\jfc\Java2D\Java2De...
Compilation	HotSpot Client Compiler
GCBean[0]	Copy
GCBean[1]	MarkSweepCompact
Input Arguments	-Xbootclasspath/a:c:\proj\nprof\dist\nprof-instrument.jar -Xdebug -Xrunjdwp:t...
Instance Name	4060@meiaquatro
LibraryPath	C:\apps\Java\jre1.5.0\bin;.;C:\WINDOWS\System32;C:\WINDOWS;C:\apps\te...
No. Porcessors	1
OS Version	5.1
SpecName	Java Virtual Machine Specification
SpecVendor	Sun Microsystems Inc.
SpecVersion	1.0
Start Time	Sun Jul 03 19:29:30 BRT 2005
VmName	Java HotSpot(TM) Client VM
VmVendor	Sun Microsystems Inc.
VmVersion	1.5.0-b64
awt.toolkit	sun.awt.windows.WToolkit
file.encoding	Cp1252
file.encoding.pkg	sun.io
file.separator	\
java.awt.graphicsenv	sun.awt.Win32GraphicsEnvironment
java.awt.printerjob	sun.awt.windows.WPrinterJob
java.class.path	c:\proj\nprof\build\classes;C:\apps\Java\jdk1.5.0\demo\jfc\Java2D\Java2De...
java.class.version	49.0
java.endorsed.dirs	C:\apps\Java\jre1.5.0\lib\endorsed
java.ext.dirs	C:\apps\Java\jre1.5.0\lib\ext
java.home	C:\apps\Java\jre1.5.0

Figura 5.10: Visualizador de informações de ambiente de execução

5.6.5 Visualizador de Carga de Classes

O visualizador de carga de classes mostra em formato de tabela as informações geradas pela JDI quando são solicitados eventos referentes à carga de classes. As informações contidas nos eventos são: o nome da classe, o momento em que a carga da classe foi efetivada (em milisegundos) e o carregador de classes (*Classloader*) responsável pela carga da classe. A figura 5.11 mostra um exemplo do uso deste visualizador ao monitorar-se a aplicação `Java2Demo`.

Millis	Class	ClassLoader	In...
9390	javax.swing.JTabbedPane$Pa...		
9390	javax.swing.JPanel$AccessibI...		
9390	javax.accessibility.AccessibleB...		
9390	javax.accessibility.AccessibleS...		
9390	javax.accessibility.AccessibleS...		
9390	javax.swing.JTabbedPane$Acc...		
9406	java2d.DemoGroup	instance of sun.misc.Launcher$AppClassLoader(id=159)	
9406	java2d.DemoPanel	instance of sun.misc.Launcher$AppClassLoader(id=159)	
9422	java.awt.print.Printable		
9453	java2d.Surface	instance of sun.misc.Launcher$AppClassLoader(id=159)	
9453	java2d.AnimatingSurface	instance of sun.misc.Launcher$AppClassLoader(id=159)	
9453	java2d.demos.Arcs_Curves.Arcs	instance of sun.misc.Launcher$AppClassLoader(id=159)	
9484	java2d.Tools	instance of sun.misc.Launcher$AppClassLoader(id=159)	
9484	javax.swing.ImageIcon		
9484	javax.swing.ImageIcon$1		
9484	java2d.Tools$ToggleIcon	instance of sun.misc.Launcher$AppClassLoader(id=159)	
9484	javax.swing.plaf.metal.MetalTo...		
9484	javax.swing.plaf.metal.MetalBo...		
9484	javax.swing.JToolBar		
9484	javax.swing.JToolBar$DefaultT...		

Figura 5.11: Visualizador de carga de classe

É interessante notar que a figura 5.11 permite verificar que os carregadores de classe utilizados pela aplicação-alvo operam conforme sua especificação (LIANG; BRACHA, 1998): as classes da API Java (por exemplo, classes `javax.*`) são carregadas pelo carregador de classes *Bootstrap* (que é representado pela coluna *ClassLoader* vazia, ou seja, para tais classes é utilizado um carregador interno à máquina virtual), enquanto as classes da aplicação-alvo (`java2d.*`) são carregadas por um carregador de classes de sistema (no caso da classe `sun.misc.Launcher$AppClassLoader`).

5.6.6 Visualizador de Exceções

O visualizador de exceções, assim como o de carga de classes, cria uma tabela a partir de eventos JDI redirecionados para ele. Os campos da tabela gerada são: tempo em que ocorreu a exceção (em milissegundos); classe da exceção; descrição da exceção; local de ocorrência da exceção (classe e método); local de captura (caso tenha sido capturada); e *thread* em que ocorreu a exceção.

Entretanto, além de organizar os eventos em forma de tabela, o visualizador cria também um painel auxiliar (parte inferior da figura 5.12) para mostrar informações detalhadas sobre a pilha de execução de uma exceção selecionada pelo usuário. No momento que o usuário clica em uma linha da tabela (cada linha corresponde a uma exceção), o visualizador realiza uma chamada a um método remoto, via JDI. Esta chamada busca, na máquina virtual remota, a pilha de execução da *thread* que executava no momento em que a exceção fora gerada, o que é um mecanismo útil para a depuração de exceções. A figura 5.12 mostra um exemplo do uso do visualizador de exceções para a aplicação `Java2Demo`. As exceções em questão são referentes ao mecanismo de "serialização" Java. Note-se, contudo, que elas são exceções que foram efetivamente capturadas por meio da primitiva `catch` (do bloco `try/catch`), mas mesmo as exceções capturadas podem ser muito relevantes para a depuração de um sistema.

millis	Class	Message	Location	Catch...	Thread	Threa...	Except...
1281	class java.lang.NoSuchFieldException (...	serialPersistentFields	java.lang...	java.io...	Target...	11	insta...
1421	class java.lang.NoSuchFieldException (...	serialVersionUID	java.lang...	java.io...	Target...	11	insta...
1421	class java.lang.NoSuchFieldException (...	serialPersistentFields	java.lang...	java.io...	Target...	11	insta...
1437	class java.lang.NoSuchFieldException (...	serialVersionUID	java.lang...	java.io...	Target...	11	insta...
1453	class java.lang.NoSuchFieldException (...	serialPersistentFields	java.lang...	java.io...	Target...	11	insta...
1468	class java.lang.NoSuchFieldException (...	serialVersionUID	java.lang...	java.io...	Target...	11	insta...
1468	class java.lang.NoSuchFieldException (...	serialPersistentFields	java.lang...	java.io...	Target...	11	insta...
1484	class java.lang.NoSuchFieldException (...	serialVersionUID	java.lang...	java.io...	Target...	11	insta...
1484	class java.lang.NoSuchFieldException (...	serialPersistentFields	java.lang...	java.io...	Target...	11	insta...
1515	class java.lang.NoSuchFieldException (...	serialVersionUID	java.lang...	java.io...	Target...	11	insta...

```
class java.lang.NoSuchFieldException (no class loader): serialPersistentFields (@Wed Dec 31 21:00:0
   location: java.lang.Class.getDeclaredField(java.lang.String)+31
   catch location: java.io.ObjectStreamClass.getDeclaredSerialFields(java.lang.Class)+42
   thread: TargetMonitor:class nprof.target.TargetMonitor (loaded by instance of sun.misc.Launcher$
StackTrace:
   java.lang.Class.getDeclaredField(unknown)
   java.io.ObjectStreamClass.getDeclaredSerialFields(unknown)
   java.io.ObjectStreamClass.getSerialFields(unknown)
   java.io.ObjectStreamClass.access$700(unknown)
   java.io.ObjectStreamClass$2.run(unknown)
```

Figura 5.12: Visualizador de exceções

5.6.7 Visualizador de *Sockets* TCP

O visualizador de *sockets*, assim como o de *threads*, permite a visualização dos dados tanto por meio de gráfico como por meio de tabela. Os gráficos gerados mostram a evolução temporal do valor acumulado de dados enviados e recebidos, tanto para número de *bytes* quanto para número de mensagens. A figura 5.13 apresenta um exemplo do uso do visualizador de *sockets* em gráfico para uma aplicação teste de RMI.

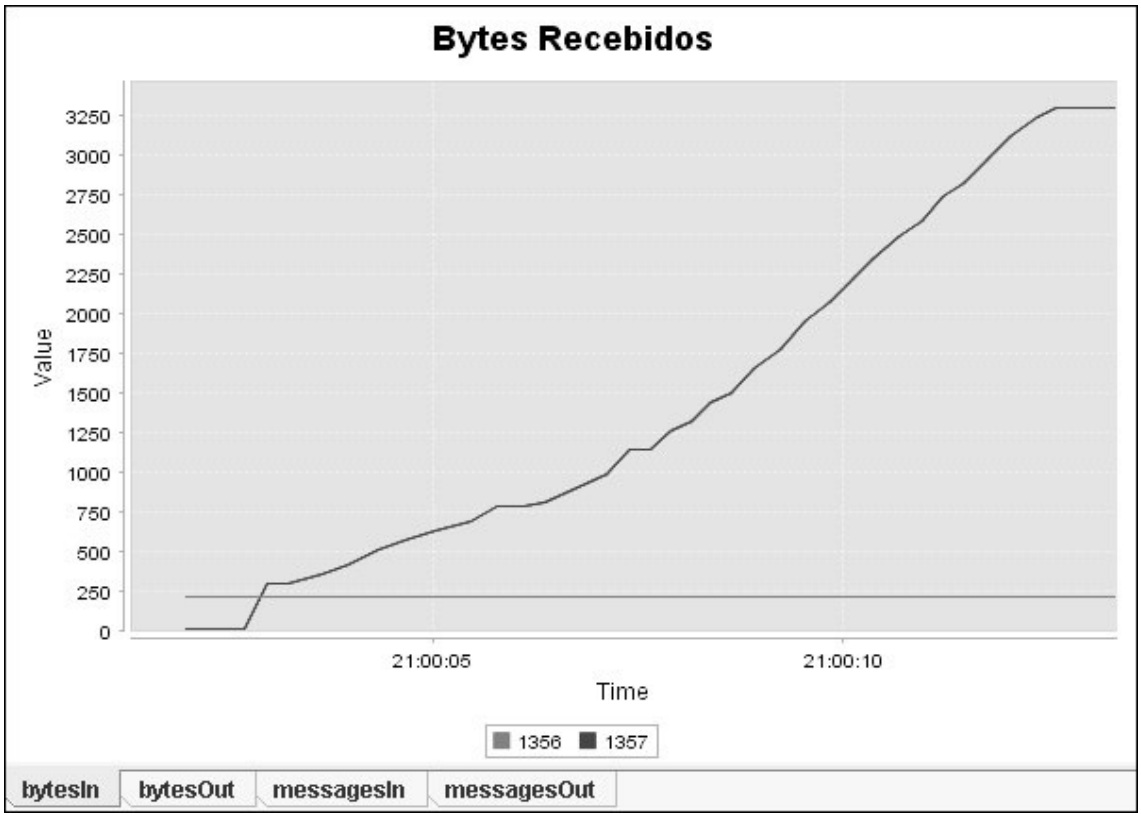


Figura 5.13: Visualizador de *sockets*: *bytes* recebidos

Na figura 5.13 se encontra o valor acumulado do número de *bytes* recebidos para cada *socket* (identificados pelo número da porta local - 1356 e 1357). De forma análoga, o visualizador gera gráficos semelhantes para número de *bytes* enviados, número de mensagens enviadas e número de mensagens recebidas.

Como opção aos gráficos, o visualizador gera também tabelas, conforme a captura de tela representada na figura 5.14. Além das informações utilizadas pelo módulo gráfico do visualizador, a tabela informa, para cada *socket*, se tal *socket* é do tipo servidor (que aceita conexões, coluna "Server?"), o tempo necessário para o estabelecimento da conexão (aplicável para conexões clientes, coluna "Conn time"), porta remota (coluna "Port"), porta local (coluna "LPort"), endereço IP remoto e local (colunas "Addr" e "LAddr", respectivamente). Além de tais informações, a tabela contém o valor somado de *bytes* e mensagens enviadas e recebidas para todos os *sockets* da aplicação (1ª linha).

Socket	Bytes Out	Bytes In	Msgs Out	Msgs In	Server?	Conn time	Port	LPort	Addr	LAddr
Total	21117	7982	179	282						
0	38	249	2	3	true	0	1183	1099	127.0.0.1	127.0.0.1
1	472	302	3	2	false	47	1182	1184	10.1.1.5	10.1.1.5
2	38	252	2	3	true	0	1187	1099	127.0.0.1	127.0.0.1
3	472	302	3	2	false	0	1186	1188	10.1.1.5	10.1.1.5
4	216	91	3	6	true	0	1189	1099	127.0.0.1	0.0.0.0
5	19881	6786	166	266	true	0	1191	1099	127.0.0.1	127.0.0.1

Figura 5.14: Visualizador de s*ockets* por meio de tabela

5.6.8 Visualizador de *Sockets* UDP

O visualizador de *sockets* UDP (Datagrama) é semelhante ao de *sockets* TCP, permitindo a visualização em forma de tabela de informações sobre número de *bytes* e pacotes recebidos e enviados. Para exemplificar o uso de tal visualizador, foi executada uma aplicação teste do pacote JGroups (BAN, 1998), que utiliza intensamente UDP para a implementação de comunicação *multicast* confiável. A figura 5.15 apresenta a tabela gerada a partir da execução de duas instâncias da aplicação `org.jgroups.demos.Draw`, do pacote do JGroups. As colunas "Bytes Out", "Bytes In", "Port", "LPort", "Addr" e "LAddr" são análogas às colunas do visualizador de *sockets* TCP. As colunas "Pckt Out" e "Pckt In" referem-se ao número de pacotes enviados e recebidos, respectivamente.

DatagSock	Bytes Out	Bytes In	Pckt Out	Pckt In	Port	LPort	Addr	LAddr
Total	783718	124304	307	769				
0	737000	0	10	0	9	1453	10.1.1.5	0.0.0.0
1	0	46257	0	293	1455	45566	10.1.1.5	0.0.0.0
2	46257	0	293	0	45566	1455	228.8.8.8	0.0.0.0
3	0	76106	0	452	1459	45566	10.1.1.5	0.0.0.0
4	461	1941	4	24	1458	1454	10.1.1.5	10.1.1.5

Figura 5.15: Visualizador de *sockets* UDP por meio de tabela

5.6.9 Visualizador de objetos em memória

O visualizador de objetos em memória foi desenvolvido com base em visualizadores de estado de variáveis existentes em ferramentas de depuração, como o existente nos ambientes de desenvolvimento JBuilder (BORLAND, 2005-a) e Eclipse (ECLIPSE FOUNDATION, 2005-b), e em *profilers* como o JProfiler. Tal visualizador, ao contrário dos anteriores, coleta informações apenas a partir da interação com o usuário da ferramenta Nprof, e, portanto, não está atrelado à coleta de dados de nenhum monitor

específico. A partir da solicitação do usuário, o visualizador faz requisições JDI à JVM alvo para obter campos de objetos e valores de variáveis.

Para a coleta de tais dados, a aplicação-alvo deve estar em estado suspenso. O visualizador solicita, então, referências de objetos remotos às *threads* ativas da aplicação. Para cada *thread* (que deve estar suspensa), o visualizador pode solicitar sua pilha de execução, gerando uma visualização em forma de árvore. Dentro da pilha podem ser verificados quais variáveis estão ativas e seus valores atuais. É importante salientar que, de cada referência a um objeto remoto, podem-se capturar as propriedades referentes a tal objeto; propriedades estas que podem, por sua vez, ser também objetos. Sendo assim, a navegação pela árvore de objetos remotos pode ser bastante extensa.

Para exemplificar o uso do visualizador, a figura 5.16 o exibe a partir da execução da aplicação-alvo `org.jgroups.demos.Draw`, do pacote JGroups. Nas duas primeiras linhas, encontram-se duas *threads* ativas da aplicação e, nas quatro linhas subseqüentes, a pilha de execução de uma das *threads*.

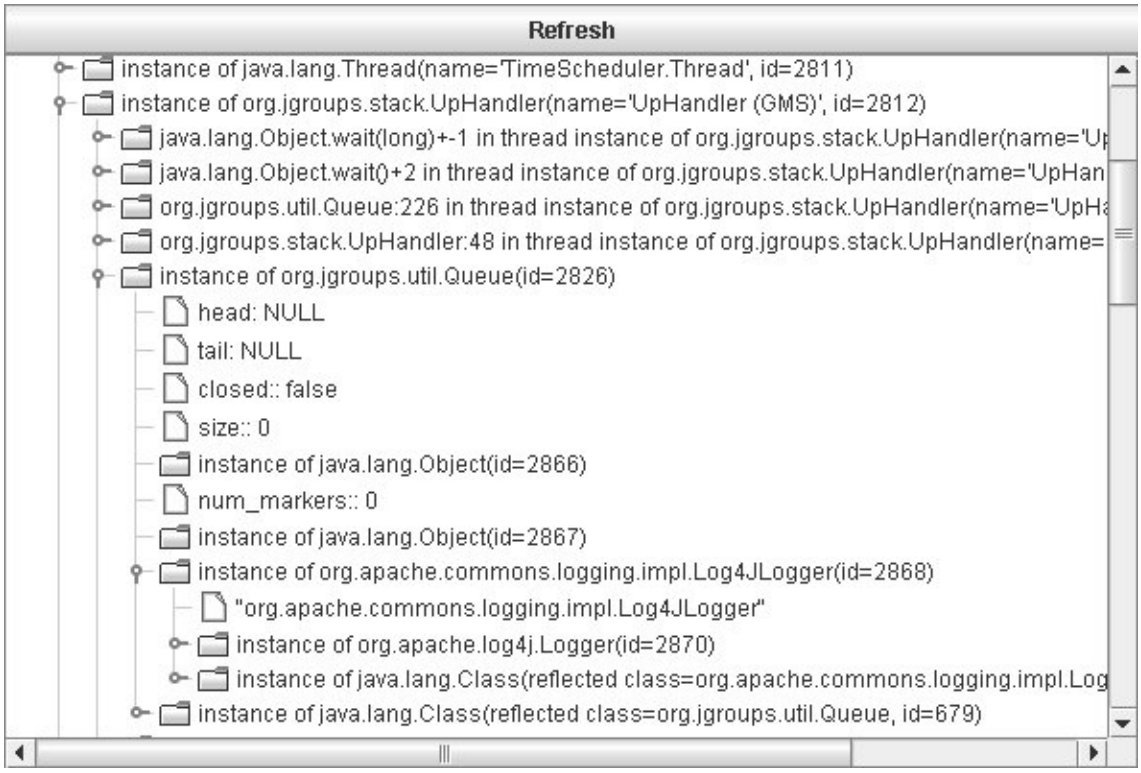


Figura 5.16: Visualizador de objetos em memória

5.6.10 Visualizador de grafo de sockets

O visualizador de grafo de *sockets* utiliza os dados coletados pelo monitor de *socket*s para criar um grafo que relaciona as diferentes aplicações-alvo monitoradas entre si, desde que estas mantenham conexões *socket* umas com as outras. Um exemplo de tal visualizador pode ser visto na figura 5.17, que é o grafo gerado a partir das conexões entre aplicações que se comunicam por meio de RMI. Cada nodo do grafo representa uma aplicação-alvo e a porta TCP utilizada pela mesma aplicação. Cada aresta representa um *socket* TCP.

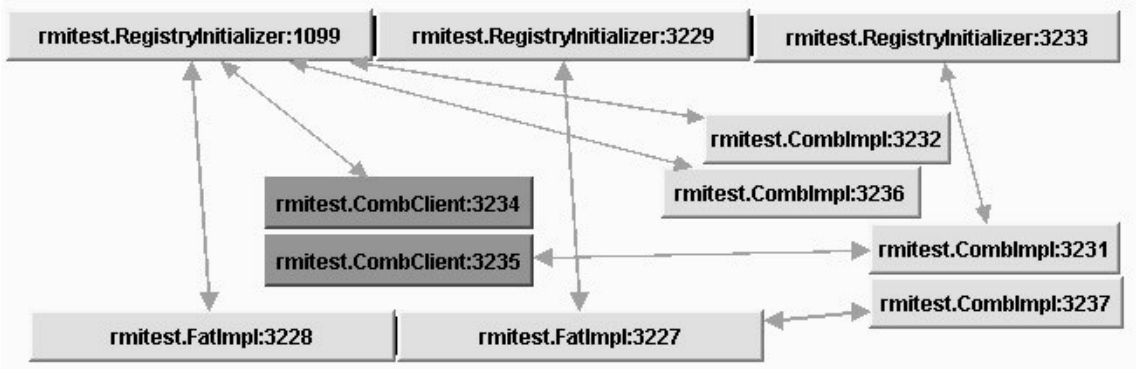


Figura 5.17: Visualizador de grafo de *sockets*

5.6.11 Arquivos de traços para mensagens socket

Algumas das informações capturadas por determinados monitores da ferramenta não possuem visualizadores correspondentes. Apesar de não permitirem a visualização *online* das informações coletadas, arquivos de traços são muito úteis para análises pós-morte de aplicações. Este é o caso dos arquivos de traços gerados a partir das mensagens enviadas e recebidas por *sockets*. Tais mensagens são armazenadas em um arquivo de traços na máquina correspondente à JVM monitorada. Para cada JVM monitorada é criado um arquivo de traços, mesmo que múltiplas JVMs estejam executando na mesma máquina real. O arquivo de traços para mensagens de *socket*s (sejam elas UDP ou TCP) é gerado no seguinte formato:

```
timestamp; porta local; IP remoto; porta remota; protocolo; sentido; mensagem
```

em que:

- Timestamp: indica o milisegundo em que ocorreu o evento de envio ou recebimento de mensagem;
- Porta local: indica a porta pela qual a mensagem foi enviada ou recebida;
- IP remoto: identifica o endereço IP da máquina remota com a qual a JVM monitorada está conectada;
- Porta remota: especifica a porta utilizada pela aplicação remota;
- Protocolo: indica se é uma mensagem TCP ou UDP;
- Sentido: indica se a mensagem está sendo recebida ou enviada;
- Mensagem: corresponde ao conteúdo da mensagem propriamente dita.

Ao final do arquivo são registradas informações referentes à diferença apurada entre os relógios das máquinas. Assim, é possível, por meio de pós-processamento de todos os arquivos de traços das JVMs envolvidas, recriar a ordem parcial entre todas as mensagens enviadas e recebidas em uma aplicação distribuída.

5.6.12 Arquivo de traços para compilação e coleta de lixo

Além do arquivo para traços de mensagens de rede, a ferramenta gera outro com as informações capturadas sobre compilação em tempo de execução (JIT) e sobre coleta de lixo. Ambos os tipos de informação são capturadas pelo uso da API JMX. As informações capturadas incluem:

- tempo acumulado utilizado para compilação JIT;
- número de vezes que o coletor de lixo foi ativado;
- tempo acumulado utilizado para a coleta de lixo.

5.7 Modelo de extensão de Nprof

Nprof foi desenvolvido com um modelo de extensão definido. O modelo de extensão aplica-se a monitores e visualizadores, e permite que sejam agregadas à ferramenta novas classes para monitores e visualizadores, desde que estas cumpram o contrato estabelecido pelas interfaces às quais devem implementar.

Na implementação de novos monitores, estes devem implementar a interface Java `nprof.monitor.Monitor`. Tal interface é mostrada na figura 5.18.

```
public interface Monitor {
    boolean require(int i);
    void init(MirrorManager mirrorm);
    void refreshActiveTargetSettings();
    void refreshInactiveTargetSettings();
    void setPrecisionMultiplier(int i);
    int getPrecisionMultiplier();
    void captureInfo();
    boolean errorOnCapture();
}
```

Figura 5.18: Interface Monitor

Como `Monitor` uma interface, todos seus métodos devem ser implementados, mesmo que o corpo da implementação seja, de fato, vazio. Entretanto, a alguns métodos deve ser reservada maior importância. O método `require(int)` verifica quais são as funcionalidades que ele necessita estarem presentes na aplicação monitorada (basicamente se ele necessita efetuar chamadas a métodos remotos). O método `init()` permite a inicialização dos recursos a serem utilizados pelo monitor e passa por parâmetro a referência da representação da máquina alvo monitorada (`MirrorManager`). O método `captureInfo()` é o responsável pela captura dos dados efetivamente, e é ativado de acordo com a taxa de amostragem determinada para tal monitor. Para que os dados capturados sejam enviados aos visualizadores, o método `captureInfo()` deve realizar uma chamada ao método `log()` da classe `MessageMediator` e passar os dados coletados encapsulados em uma subclasse de `PInfo` (figura 5.3).

Assim como para a implementação de novos monitores, novos visualizadores devem implementar uma interface específica, no caso a `MessageListener`, representada na figura 5.19.

```
public interface MessageListener {
   public boolean message(String component, String message);
   public boolean message(MirrorManager mm, String component,
                          PInfo     info);
}
```

Figura 5.19: Interface MessageListener

Os visualizadores, implementando a interface `MessageListener`, têm seus métodos `message()` invocados a cada vez que um novo dado é coletado, e podem assim atualizar a interface com o usuário. É por meio de ambas as interfaces que a classe `MessageMediator` consegue correlacionar a coleta de dados dos diferentes monitores aos diferentes visualizadores dispensando a referência direta a qualquer classe concreta.

6 ESTUDOS DE CASO

6.1 Introdução

Este capítulo apresenta alguns estudos de casos realizados com a ferramenta Nprof a fim de avaliar a eficiência, desempenho e utilidade da ferramenta. O primeiro estudo de caso (seção 6.2) considera as possibilidades de uso da ferramenta em uma aplicação Java largamente utilizada na indústria, o servidor *web* e *container* Servlet/JSP Tomcat (APACHE, 2005). O segundo (seção 6.3), baseado em estudo anterior (BRUGNARA; CECHIN; LISBÔA, 2005), avalia o desempenho da ferramenta e a sobrecarga causada pela ativação de seus diversos monitores frente a uma aplicação distribuída RMI. O terceiro e último estudo de caso (seção 6.4) examina as funcionalidades da ferramenta Nprof frente a um cenário de aplicação distribuída, utilizando diferentes níveis de confiança de entrega por meio do uso do *framework* de comunicação de grupos JavaGroups (BAN, 1998).

Cabe salientar que não é de interesse traçar o perfil dos casos abordados em múltiplas execuções, mas sim observar as características e propriedades da ferramenta Nprof.

6.2 Monitoramento da aplicação Tomcat

O servidor *web* e *container* Servlet/JSP Tomcat (APACHE, 2005) é largamente utilizado como servidor de aplicações web. Devido a sua reputação de conformidade com as especificações JSP e Servlet, o Tomcat é considerado como implementação referência de tais especificações (SUN MICROSYSTEMS, 2005-g).

Visto que Nprof tem como objetivo ser útil para o monitoramento de aplicações Java existentes e mesmo aplicações de relativa complexidade, o *container* Tomcat foi escolhido como uma aplicação-alvo para avaliar a efetividade da ferramenta Nprof. Além disso, por ser uma aplicação servidora, o Tomcat permite o teste das funcionalidades da ferramenta Nprof quanto ao uso de *sockets*.

6.2.1 Testes

A fim de simular[4] uma aplicação cliente-servidor, foi desenvolvida uma aplicação teste que realiza requisições HTTP, denominada HTTPClient. Tal aplicação atua da seguinte forma: cria 10 *threads* para realizar requisições em paralelo, e cada *thread* realiza 20 requisições da página inicial do servidor Tomcat. A fim de avaliar diferentes aspectos da execução do servidor Tomcat, foram ativados os monitores de *threads*, *sockets* e memória na aplicação-alvo Tomcat. Note-se que todos os gráficos possuem uma linha de tempo relativa ao inicio do monitoramento, tornando possível, assim, relacionar eventos de diferentes naturezas. A figura 6.1 apresenta o gráfico relativo ao uso de *threads* por parte do Tomcat.

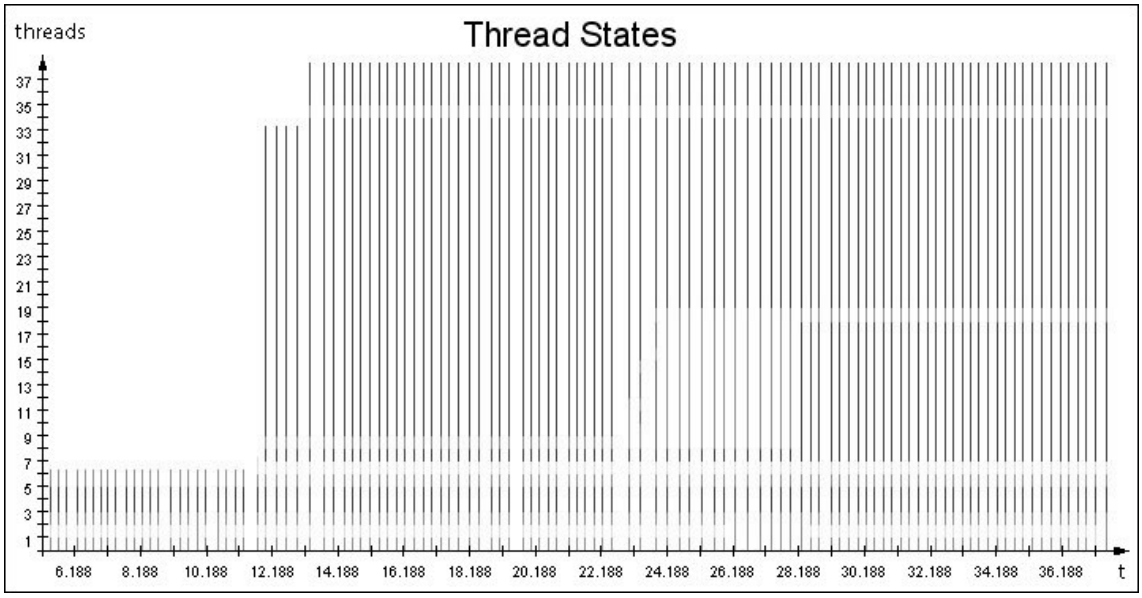


Figura 6.1: Estado de *threads* do servidor Tomcat

Na figura 6.1 percebe-se que, por volta do 11º segundo de execução (entre os milisegundos 10.188 e 12.188), há a criação de várias *threads*, e que tais *threads* estão no estado de espera (em vermelho). Posteriormente identificou-se que esse comportamento se deve à configuração do servidor Tomcat, que, por padrão, inicia 25 *threads* para atender às requisições HTTP.

Na mesma figura percebe-se também que, por volta do 23º segundo de execução algumas *threads* mudam de estado, passando do estado de espera (vermelho) para o estado de execução (verde). Tal transição está associada ao início das requisições HTTP por parte da aplicação cliente HTTPClient. Pode-se ainda notar que, no período inicial dessa transição, algumas *threads* assumiram o estado de espera por recurso (em cor rosa), devido ao uso de primitivas de sincronização internas ao Tomcat. Após o término das requisições, que ocorre próximo ao 28º segundo, a maior parte das *threads* envolvidas retorna ao estado de espera.

Para avaliar a relação do uso de memória por parte da aplicação Tomcat com o número de *threads* ativas, utilizou-se o monitor de memória. A figura 6.2 indica o uso de memória pela aplicação Tomcat ao longo do tempo.

---

[4] Os testes foram realizados em um único computador, com a seguinte configuração: Processador Athlon 64 2800+, 512 Mb RAM, Java 1.5.0_04 e Tomcat 5.5.9

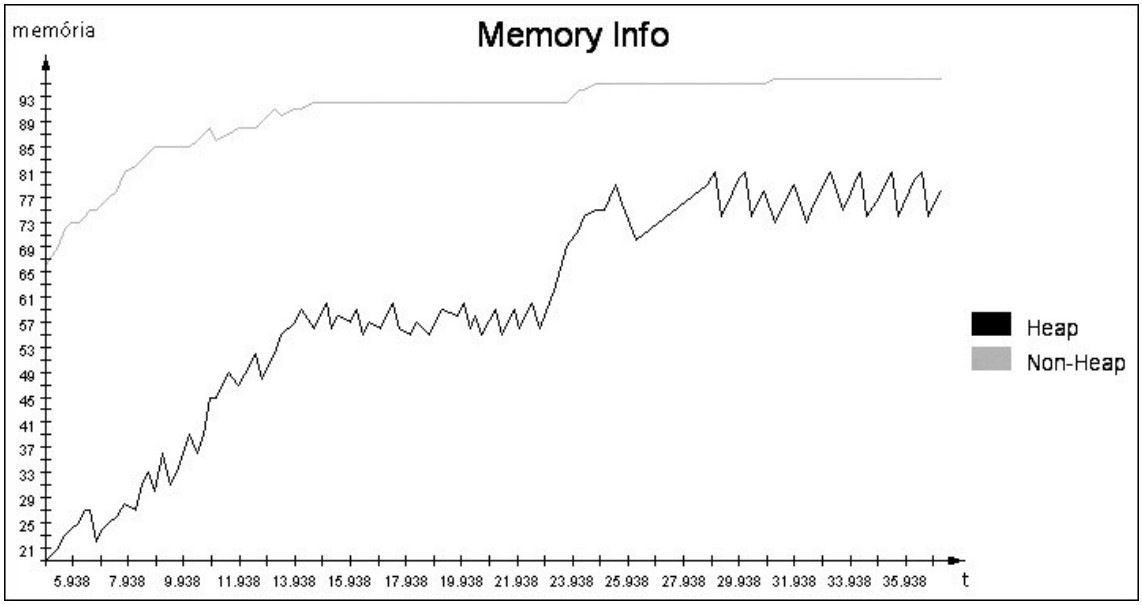


Figura 6.2: Uso de memória pelo servidor Tomcat

Assim como para o gráfico de *threads*, percebe-se um incremento abrupto no uso da memória *heap* em torno do 22º segundo, fato que está relacionado com o início das requisições HTTP por parte da aplicação de teste HTTPClient.

Utilizou-se também o monitor de *sockets*, a fim de verificar as características das conexões *socket* realizadas e a quantidade de dados trafegados. Com os dados sobre o tráfego de mensagens coletados, elaborou-se o gráfico ilustrado na figura 6.3. Os dados são referentes aos valores acumulados de mensagens enviadas e recebidas.

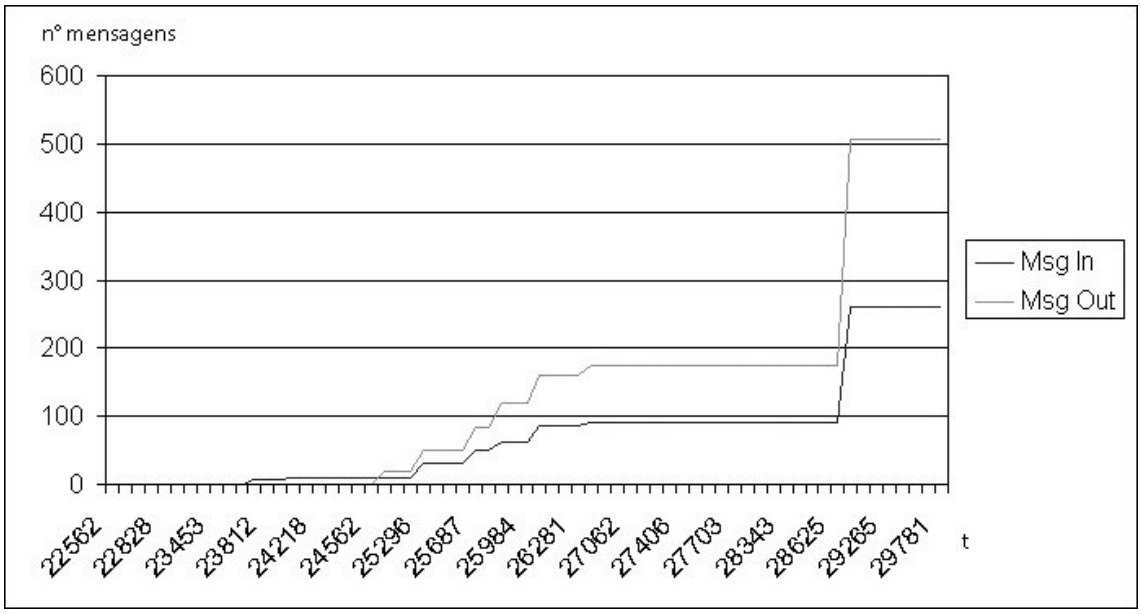


Figura 6.3: Tráfego de mensagens entre Tomcat e HTTPClient

Confirmando a expectativa, os dados coletados pelo monitor de *sockets* permitem identificar que mensagens começam a ser enviadas e recebidas pela aplicação Tomcat no 23º segundo de execução. A análise dos dados permite perceber ainda que, durante a maior parte do tempo de execução, a razão entre a quantidade de mensagens enviadas e recebidas tende a dois. A explicação para tal fenômeno é que, enquanto a mensagem de

solicitação HTTP do presente estudo de caso é bastante curta (em média, o tamanho da mensagem é de 197 *bytes*), a mensagem de retorno correspondente é relativamente grande (em média, tem 4664 *bytes*). Portanto, o retorno à solicitação HTTP foi, em geral, fragmentado em dois pacotes TCP.

6.2.2 Considerações sobre o estudo de caso

O presente estudo de caso tem o propósito principal de ilustrar as funcionalidades da ferramenta Nprof frente a uma aplicação de largo uso na indústria. A partir dos aspectos observados, considera-se que os testes realizados foram significativos na avaliação de um caso de uso de uma aplicação-alvo real. Mesmo utilizando apenas uma parte dos monitores disponíveis, foi possível obter um perfil razoável do modo de operação da aplicação-alvo.

6.3 Avaliação de desempenho: aplicação RMI

Para avaliar a sobrecarga da ferramenta, enquanto captura os diferentes tipos de informações, foi desenvolvida uma pequena aplicação distribuída de teste. Esta aplicação é de caráter matemático e tem por objetivo calcular as combinações de *n* elementos escolhidos *p* a *p*. Como o único propósito da aplicação-alvo desenvolvida é o de avaliar a sobrecarga da ferramenta, os resultados das computações realizadas pela aplicação-alvo foram desprezados.

A estrutura da aplicação teste é formada por quatro módulos: o módulo de serviço de nomes, `rmitest.RegistryInitializer`, que em Java RMI é conhecido como *registry*; dois módulos servidores, `rmitest.FatImpl` (`FAT`) e `rmitest.CombImpl` (`COMB`), que implementam respectivamente a função fatorial e o cálculo da combinação propriamente dita; e o módulo cliente, `rmitest.CombClient`, responsável por requisitar o serviço.

Na aplicação, o módulo cliente requisita ao servidor `COMB` o cálculo do número de combinações simples a partir dos valores naturais *n* e *p*, em que *n* representa a cardinalidade de um dado conjunto *X* e *p,* a cardinalidade de um subconjunto de *X*. A fórmula da quantidade de combinações é dada por:

$$C_n^p = \frac{n!}{p!(n-p)!}$$

Por sua vez, esse servidor requisita ao servidor `FAT` (`rmitest.FatImpl`) a computação dos três fatoriais: *n!*, *p!* e *(n-p)!*. Quando o módulo servidor `COMB` recebe as respostas das três requisições, ele calcula o número de combinações e o retorna ao cliente.

A figura 6.4 é um exemplo de grafo gerado dinamicamente pela ferramenta Nprof para a aplicação descrita anteriormente. Ela apresenta a interação via *sockets* dos diversos módulos da aplicação.

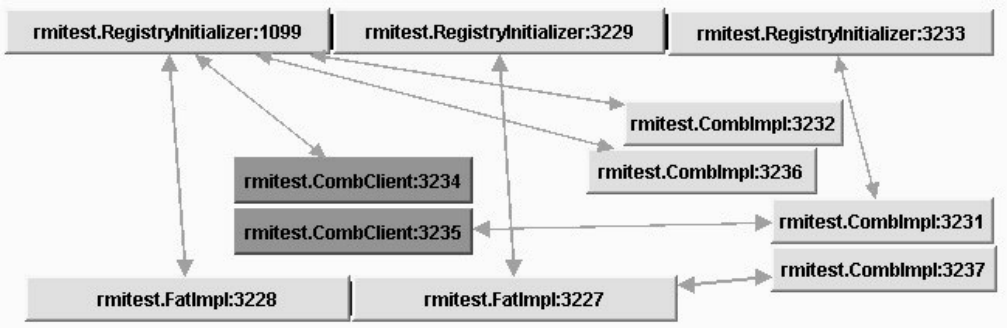


Figura 6.4: Interação entre os componentes da aplicação RMI

Cada vértice no grafo representa uma porta *socket* aberta. No vértice existem etiquetas com o nome do módulo ao qual pertencem, seguido pelo número da porta. Cada aresta é uma conexão *socket*. Note-se que uma porta *socket* servidora (uma porta *socket* que aceita conexões) pode ter muitas conexões. Este é o caso do vértice identificado por `RegistryInitializer:1099`.

6.3.1 Organização

Para a realização dos testes, foram utilizadas três máquinas idênticas conectadas a uma rede Fast Ethernet (100Mbps) do seguinte modo: na primeira máquina é executada a ferramenta Nprof (módulo Nprof Núcleo); na segunda máquina, o módulo monitorado por Nprof (`CombClient`), bem como o módulo Nprof Remoto; e na terceira máquina, os módulos servidores (`COMB` e `FAT`) e o servidor de nomes (`RegistryInitializer`). A figura 6.5 mostra esta organização:

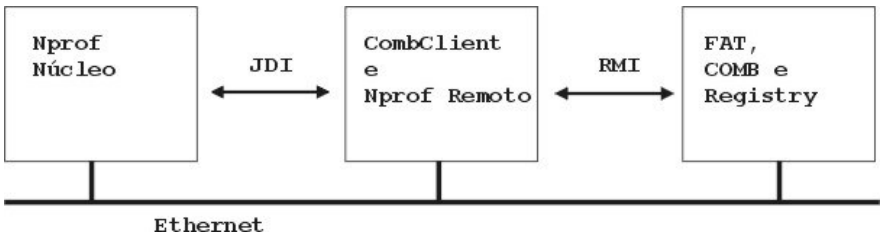


Figura 6.5: Estrutura da configuração dos testes da aplicação RMI

As três máquinas usadas são do modelo Dell Optiplex GX270 com a seguinte configuração, idêntica para as três: Pentium IV 2.8GHz, 248 Mb RAM, Ethernet Intel PRO 1000 MT (rodando a 100Mbps), Windows XP Professional SP2, Java HotSpot(TM) Client VM (build 1.5.0_01-b08, mixed mode, sharing) e HD IDE 20Gb: WDC WD400BB-75JA1.

6.3.2 Definição e metodologia dos testes

Para identificar o impacto de cada monitor no desempenho da aplicação, e levando-se em consideração que a ferramenta Nprof pode ativar diversos monitores simultaneamente, elaborou-se um conjunto de testes em que foi utilizado um monitor por vez, garantindo uma análise isolada de cada tipo de monitoramento. Para testar a sobrecarga dos monitores, foi gerada uma carga de trabalho correspondente ao aspecto analisado pelo monitor. Foram geradas cargas para avaliar o monitor de *threads*, o monitor de carga de classes, o monitor de instanciação de objetos e o monitor de exceções. Assim, pôde-se analisar o impacto do aumento de carga sobre cada monitor.

A metodologia utilizada para a análise de sobrecarga dos monitores para os quais foi gerada carga de trabalho é a seguinte: definem-se diferentes níveis de sobrecarga, com intervalos constantes de incremento de carga de trabalho. Para cada nível de carga de trabalho, mede-se o tempo gasto em três cenários: (i) sem o uso do Nprof; (ii) com o uso do Nprof, porém sem o monitor específico ativo; (iii) e com Nprof e monitor ativos. Para cada uma das medidas é calculado o intervalo de confiança ao nível de 95% e o coeficiente de variação do intervalo de confiança.

Para cada um dos testes também são definidas medidas de sobrecarga, calculadas para cada um dos níveis de carga de trabalho. Tais medidas são definidas conforme segue:

- **nprof** – é a sobrecarga temporal da aplicação instrumentada com o Nprof em relação à aplicação não instrumentada. Fórmula:

$$nprof = \frac{Tcn - Tsn}{Tsn}$$

- **nprof+mon** – sobrecarga temporal da aplicação instrumentada com o Nprof e o monitor específico em relação à aplicação não instrumentada. Fórmula:

$$nprofmon = \frac{Tnm - Tsn}{Tsn}$$

- **mon** – sobrecarga temporal da aplicação instrumentada com o Nprof e o monitor específico em relação a aplicação instrumentada apenas com o Nprof. Corresponde à sobrecarga do monitor, sem considerar a carga do Nprof. Fórmula:

$$mon = \frac{Tnm - Tcn}{Tcn}$$

- **(nprof+mon)-mon** – diferença entre a sobrecarga do uso do Nprof junto com monitor em relação à sobrecarga causada pelo monitor isoladamente. Esta coluna corresponde à sobrecarga imposta pelo Nprof. Entretanto, diferentemente da coluna nprof, aqui aparece o efeito da interação entre o Nprof e o monitor específico. Fórmula:

$$nprof\,2 = nprofmon - mon$$

Não foram realizados testes de carga com os monitores de *socket*, memória e *runtime*. No caso dos monitores de memória e *runth ime*, a sobrecarga é mínima e não há um meio direto de geração de carga.

A aplicação usada nos testes solicitou a computação do cálculo da combinação de *n* valores combinados *p* a *p*, usando 10 pares fixos de valores, repetida por 10 vezes a operação com cada par, totalizando 100 cálculos de combinação por execução.

6.3.3 Testes de *threads*

Para efetuar as medições de avaliação do monitor de *threads*, foram realizados três conjuntos de testes, conforme a metodologia explícita na seção 6.3.2: uma medição de referência (sem o uso de Nprof); uma segunda medição com o Nprof ativo e nenhum monitor ativado; e uma terceira medição em que o Nprof e o monitor de *threads* estavam

ativos. Cada conjunto de testes foi executado para um determinado número de *threads* que variavam de 0 a 2000, em incrementos de 400. Tais *threads* consistiam unicamente de um laço infinito sem processamento algum. Cada configuração de teste foi executada 20 vezes. Destas, foram excluídos o maior e o menor valor. Na tabela 6.1 estão listadas as médias dos tempos de execução obtidos e seus intervalos de confiança ao índice de 95%.

Tabela 6.1: Tempos de *thread* e intervalos de confiança

	sem nprof			com nprof			nprof + mon		
No. Threads	Média	IC	IC %	Média	IC	IC %	Média	IC	IC %
0	5327	±43	0,8%	5533	±172	3,1%	5516	±167	3,0%
400	5403	±33	0,6%	5634	±193	3,4%	5674	±204	3,6%
800	5492	±41	0,7%	5792	±165	2,8%	5836	±179	3,1%
1200	5601	±52	0,9%	5963	±194	3,3%	6015	±171	2,8%
1600	5736	±38	0,7%	6178	±165	2,7%	6380	±183	2,9%
2000	5867	±39	0,7%	6560	±175	2,7%	6780	±334	4,9%

Na tabela 6.1, os conjuntos de colunas sem nprof, com nprof e nprof+mon correspondem às medidas em cada um dos tipos de configuração. As colunas das Médias representam as médias aritméticas dos tempos considerados, em milisegundos; os valores das colunas IC representam os intervalos de confiança obtidos referente ao índice de 95%, também em milisegundos; e as colunas IC% representam os coeficientes de variação dos intervalos de confiança (IC/Média).

A partir de tais valores, foi possível o cálculo da sobrecarga relativa devido ao uso da ferramenta Nprof. Os valores de sobrecarga relativa foram calculados conforme estabelecido na seção 6.3.2 e encontram-se na tabela 6.2.

Tabela 6.2: Sobrecarga relativa referente ao uso de *threads*

Num. Threads	nprof	nprof+mon	mon	(nprof+mon)-mon
0	4%	4%	0%	4%
400	4%	5%	1%	4%
800	5%	6%	1%	5%
1200	6%	7%	1%	7%
1600	8%	11%	3%	8%
2000	12%	16%	3%	12%

É interessante notar que a diferença entre as medições de sobrecarga do Nprof isoladamente (coluna nprof) e aquelas calculadas na coluna (nprof+mon)−mon resulta em valores muito semelhantes. Com isso, verifica-se que a interação entre o Nprof e o monitor de *threads* não acarreta um custo significativo ao desempenho da aplicação instrumentada.

Na figura 6.6 são apresentados os gráficos relativos às três primeiras colunas de sobrecarga da tabela 6.2. Nesta figura, percebe-se que a sobrecarga imposta pelo Nprof é

mais significativa do que aquela causada pelo monitor de *threads*. Isso pode ser identificado pelos valores das curvas nprof e nprof+mon quando comparadas com a curva mon, que apresenta valores relativamente menores do que as primeiras.

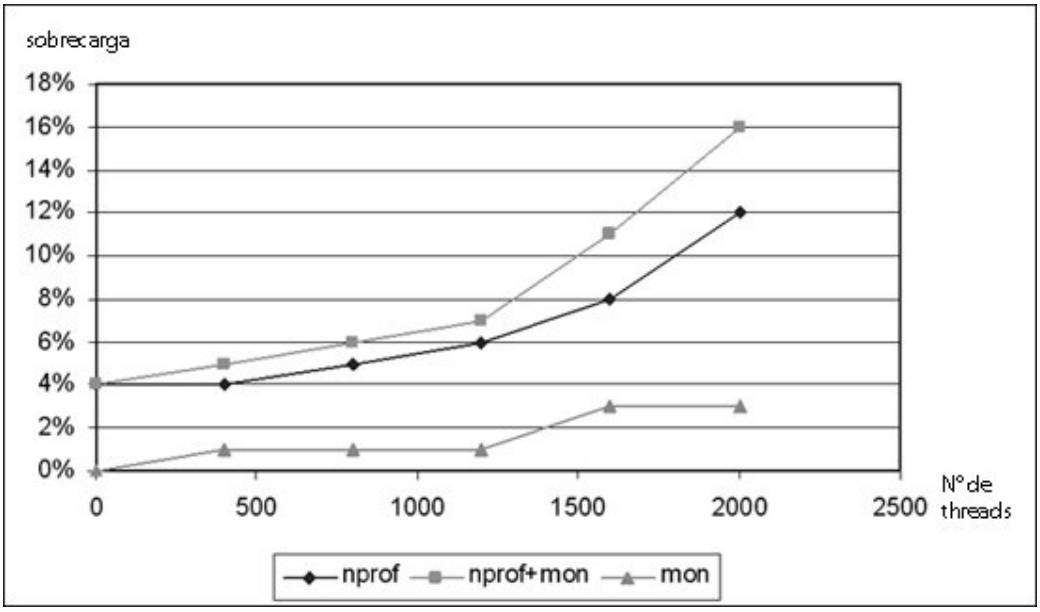


Figura 6.6: Sobrecargas no uso de Nprof e do monitor de *threads*

6.3.4 Testes de carga de classes

De acordo com a metodologia proposta na seção 6.3.2, foi desenvolvida uma carga de teste relacionada com a de classes, de maneira a avaliar o impacto do monitor de carga de classes no desempenho das aplicações. Desta forma, decidiu-se por forçar a carga de até 500 classes. Para isso, levando-se em consideração o mecanismo de carga de classes, no qual uma classe, uma vez carregada, não pode ser diretamente "descarregada", foram criadas 500 classes idênticas (com o mesmo código objeto – *bytecode*) com tamanho de 1,3 *Kbytes* cada. É interessante notar que, por não haver "descarga" de classes, aquelas que foram carregadas permanecem ocupando a memória até o fim da execução do teste (ao contrário dos objetos).

Para os testes com o monitor de carga de classes foram também realizados três conjuntos de testes, conforme definido na seção 6.3.2. Cada conjunto de testes foi executado com um número determinado de cargas de classes, o que corresponde a uma configuração. O número de classes carregadas em cada configuração variou entre 0 a 500 com incremento de 100. Cada configuração de teste foi rodada 20 vezes e, destas, foram excluídas a maior e a menor medida. Assim como na tabela 6.1, na tabela 6.3 estão listados os tempos de execução, em milisegundos; ao lado, o intervalo de confiança obtido referente ao índice de 95% (IC), também em milisegundos; e, ao lado deste, o coeficiente de variação do intervalo de confiança (IC %).

Tabela 6.3: Tempos de carga de classe e intervalos de confiança

	sem nprof			com nprof			nprof + mon		
No. Classes	Média	IC	IC %	Média	IC	IC %	Média	IC	IC %
0	5318	±30	0,6%	5540	±210	3,8%	9242	±215	2,3%
100	5602	±80	1,4%	6219	±290	4,7%	10187	±200	2,0%
200	5843	±75	1,3%	6654	±256	3,8%	10793	±347	3,2%
300	6047	±77	1,3%	7188	±286	4,0%	11624	±292	2,5%
400	6247	±93	1,5%	7862	±388	4,9%	12299	±269	2,2%
500	6485	±157	2,4%	8412	±296	3,5%	13147	±376	2,9%

Os títulos das colunas da tabela 6.3 correspondem aos definidos na seção 6.3.2. A partir dos valores encontrados, foi possível o cálculo da sobrecarga relativa referente ao uso da ferramenta Nprof. Os valores de sobrecarga relativa encontram-se na tabela 6.4.

Tabela 6.4: Sobrecarga relativa referente à carga de classes

Num. Classes	nprof	nprof+mon	mon	(nprof+mon)-mon	diff-nprof
0	4%	74%	67%	7%	3%
100	11%	82%	64%	18%	7%
200	14%	85%	62%	23%	9%
300	19%	92%	62%	31%	12%
400	26%	97%	56%	40%	15%
500	30%	103%	56%	46%	17%

O significado de cada coluna da tabela 6.4 segue a metodologia exposta na seção 6.3.2, porém a coluna diff-nprof foi acrescentada. A coluna diff-nprof representa a diferença entre os dois modos de cálculo da sobrecarga do monitor isolado, ou seja, a diferença entre a coluna (nprof+mon)-mon e a coluna nprof.

Nota-se que, mesmo sem o uso de uma carga forçada de classes (tabela 6.4, Num classes = 0), há uma sobrecarga expressiva ao utilizar-se o Nprof com o monitor. Essa sobrecarga deve-se à carga das classes da própria aplicação-alvo e da carga das classes da API Java que a aplicação utiliza. O número de classes carregadas pela aplicação nessa situação é de aproximadamente 360. Na figura 6.7 estão representadas graficamente as três primeiras colunas da tabela 6.4, que contêm os valores numéricos de sobrecarga referente à carga de classes.

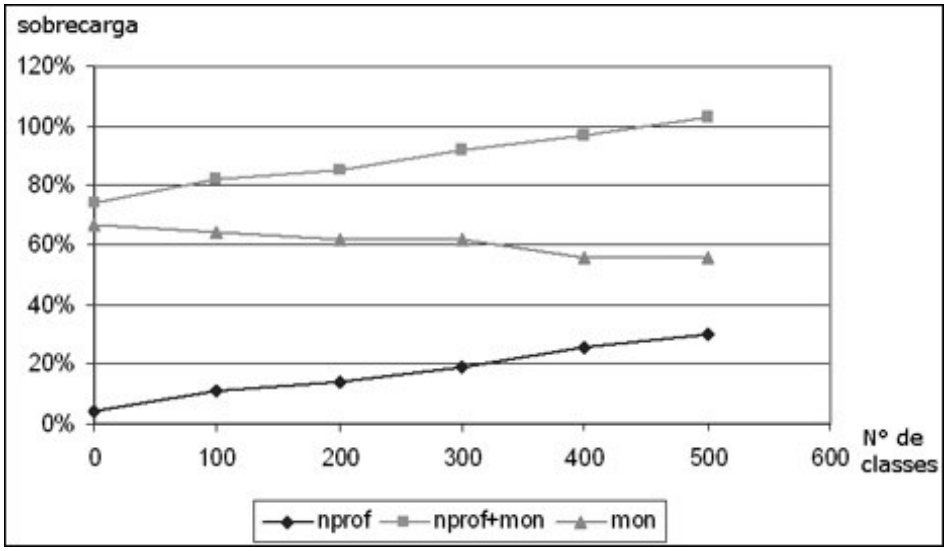


Figura 6.7: Teste de sobrecarga utilizando carga de classes

A partir das informações da tabela e do gráfico pode-se concluir o seguinte:

- A sobrecarga causada pela ferramenta tem uma relação aparentemente linear com o número de classes carregadas. Isso pode ser verificado pelo índice de correlação linear de 0,99396 entre a sobrecarga (coluna nprof da tabela 6.4) e o número de classes carregadas;
- Ao aumentar o número de classes carregadas, a sobrecarga aumentou principalmente devido à ação do Nprof, uma vez que a variação da sobrecarga do monitor isolado (coluna mon) com o número de classes carregadas não apresenta crescimento.

6.3.5 Testes de instanciação de objetos

Para a avaliação da sobrecarga do monitor de instanciação de objetos sobre a aplicação-alvo, os testes de tal monitor também foram separados em três conjuntos, conforme definido na seção 6.3.2. Cada conjunto de testes foi configurado e executado para um número específico de instanciações de objetos que variavam de 0 a 400.000, com incrementos de 80.000. Assim como nas configurações anteriores, os conjuntos de teste foram formados por 20 execuções, deles foram removidos o maior e o menor valor para formar a amostra. A tabela 6.5 contém os tempos de execução e respectivos intervalos de confiança. Os tempos estão em milisegundos.

Tabela 6.5: Tempos de instanciação de objetos e intervalos de confiança

	sem nprof			com nprof			nprof + mon		
Num. Obj	Média	IC	IC %	média	IC	IC %	média	IC	IC %
0	5319	±34	0,6%	5554	±180	3,2%	5707	±104	1,8%
160.000	5376	±47	0,9%	5634	±210	3,7%	7344	-	-
240.000	5399	±53	1,0%	5638	±198	3,5%	8266	-	-
320.000	5422	±42	0,8%	5652	±192	3,4%	8926	±181	2,0%
400.000	5433	±62	1,1%	5669	±161	2,8%	9725	±237	2,4%

Pode-se notar que os valores de intervalo de confiança para os valores de instanciações de 160.000 e 240.000 (tabela 6.5) estão nulos. Isso é devido a uma instabilidade na execução de código nativo da máquina virtual Java utilizada. Para tais casos, a "serialização" de objetos na JVM alvo para transferência à JVM do Nprof provoca um erro fatal, quando utilizado o Nprof e o monitor conjugados. Assim, apenas duas medidas foram consideradas para o caso de 160.000 objetos e uma para o caso de 240.000.

A partir dos valores encontrados, realizou-se o cálculo da sobrecarga relativa referente ao uso da ferramenta Nprof. Seus valores encontram-se na tabela 6.6.

Tabela 6.6: Sobrecarga relativa referente à instanciação de objetos

Num Obj.	nprof	nprof+mon	mon	(nprof+mon)-mon	diff-nprof
0	4%	7%	3%	5%	0%
160.000	5%	37%	30%	6%	1%
240.000	4%	53%	47%	6%	2%
320.000	4%	65%	58%	7%	2%
400.000	4%	79%	72%	7%	3%

A definição das colunas de sobrecarga segue o padrão definido na seção 6.3.2, porém contém também a coluna diff-nprof, definida da mesma forma que a coluna de mesmo nome da tabela 6.4. Na figura 6.8 estão representados, graficamente, os valores das três primeiras colunas de sobrecarga. Da observação da figura, detecta-se uma relação linearmente crescente entre a sobrecarga do monitor (curvas nprof+mon e mon) e o número de objetos instanciados, enquanto a sobrecarga imposta pelo Nprof não aumenta com o número de objetos instanciados, ficando em torno de 4%. A relação linear é confirmada pelo índice de correlação linear de 0,999095 e 0,998862, para as sobrecargas nprof+mon e mon, respectivamente.

A correlação verificada é devida à técnica de instrumentação que o monitor utiliza para capturar a informação da aplicação-alvo. O monitor é o responsável pela instrumentação da classe `Object` e essa instrumentação acarreta uma sobrecarga à instanciação de cada objeto. Assim tem-se uma relação linear entre o número de objetos instanciados e a sobrecarga causada pelo monitor, conjugado ou não com o Nprof, e a sobrecarga.

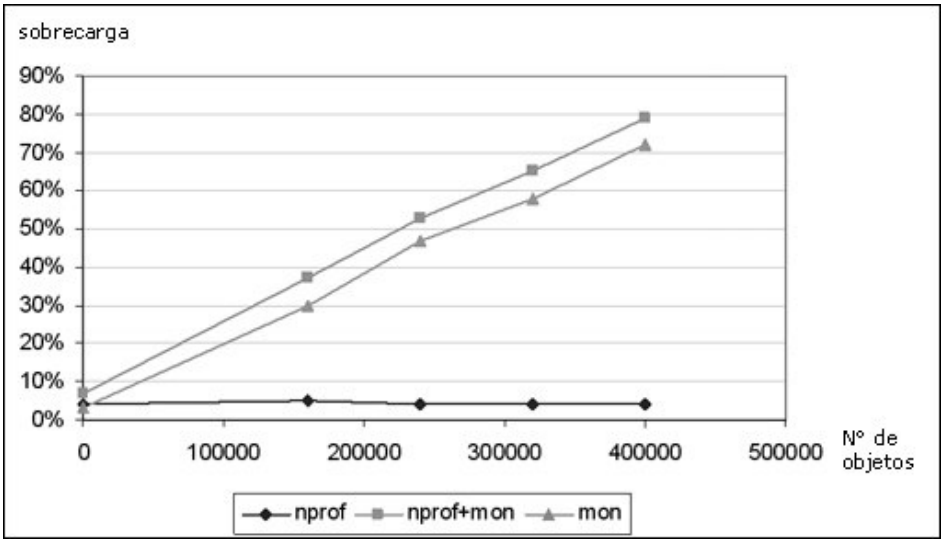


Figura 6.8: Sobrecarga comparada do Nprof e do monitor de objetos

Na figura 6.9 apresenta-se a comparação da sobrecarga gerada pelo Nprof quando operando sem monitores (coluna nprof na tabela 6.6) com aquela obtida quando o Nprof opera com monitores (coluna (nprof+mon)−mon na tabela 6.6). A diferença entre as sobrecargas destas duas situações (coluna diff-nprof na tabela 6.6) está relacionada com a interação entre o Nprof e o monitor, que só aparece quando o Nprof e o monitor estão operando conjuntamente.

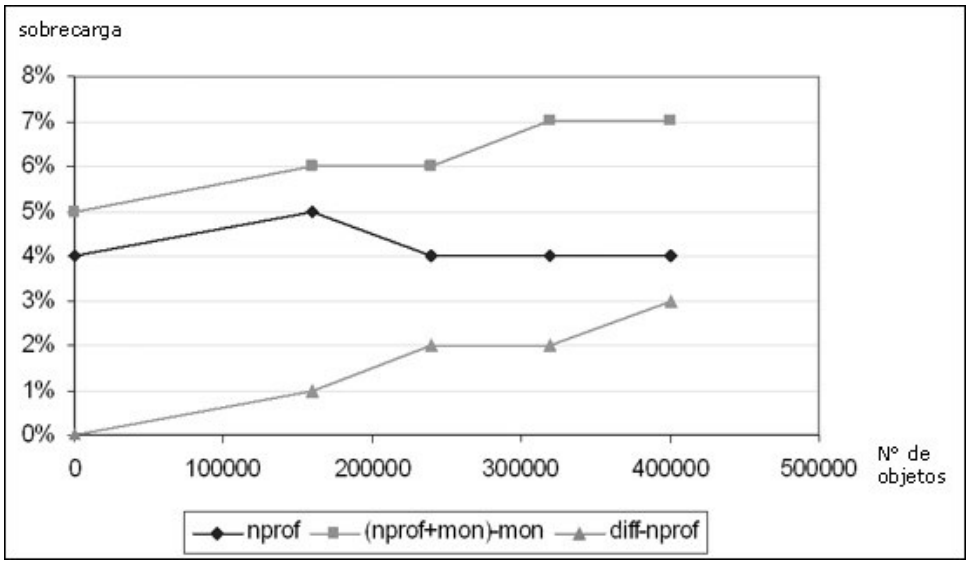


Figura 6.9: Sobrecarga relativa quanto ao monitor de objetos

A análise da figura 6.9 confirma a existência de um custo associado à interação entre o Nprof e o monitor: a curva nprof praticamente não se altera enquanto a curva (nprof+mon)−mon é crescente, indicando consumo de recursos adicionais aos consumidos pelo Nprof e monitor propriamente ditos (consumo este relacionado à interação entre Nprof e monitor).

Mesmo quando operando juntos, a sobrecarga imposta pelo Nprof é mínima, se comparada com aquela imposta pelo monitor. Além disso, a diferença cresce com o aumento do número de objetos, variando de 5 vezes, quando são feitas 160.000

instanciações de objetos, até mais de 10 vezes, quando se chega a 400.000 instanciações.

É importante notar que a sobrecarga imposta pela aplicação de teste à ferramenta se dá pela instanciação de objetos e não há mecanismo que mantenha tais objetos em memória. Portanto, para um número de 160.000 objetos instanciados, não significa, necessariamente, que estejam todos em memória, uma vez que, periodicamente, os objetos não mais utilizados são eliminados pelo coletor de lixo.

6.3.6 Testes de captura de exceções

As medidas da sobrecarga do monitor de capturas de exceções foram realizadas mediante a geração de certo número de exceções. Estas, por sua vez, foram capturadas pela própria aplicação teste.

De forma semelhante às medições efetuadas sobre os outros monitores, foram realizados três conjuntos de testes, correspondendo aos diferentes cenários definidos na seção 6.3.2. Cada conjunto de testes foi executado com um número específico de capturas de exceções. Este número variou entre 0 a 400 capturas de exceções a intervalos de 80 capturas de exceções. Cada configuração foi rodada 20 vezes, tendo sido excluídos o maior e menor valor. Na tabela 6.7 estão listados os tempos de execução e intervalos de confiança ao índice de 95%. Os tempos estão em milisegundos.

Tabela 6.7: Tempos de captura de exceções e intervalos de confiança

	sem nprof			com nprof			nprof + mon		
No. Exc.	Média	IC	IC %	Média	IC	IC %	Média	IC	IC %
0	5328	±43,20	0,8%	5571	±215,74	3,9%	7581	±393,85	5,2%
80	5325	±44,13	0,8%	5548	±135,83	2,4%	8685	±450,27	5,2%
160	5326	±40,48	0,8%	5565	±181,22	3,3%	10067	±568,49	5,6%
240	5327	±30,10	0,6%	5555	±166,47	3,0%	11345	±430,34	3,8%
320	5330	±44,72	0,8%	5565	±192,17	3,5%	12542	±513,19	4,1%
400	5332	±29,00	0,5%	5580	±188,61	3,4%	13827	±432,74	3,1%

A definição das colunas da tabela 6.7 segue o mesmo critério definido na seção 6.3.2. Na tabela 6.8, encontram-se os valores de sobrecargas relativas, conforme modelo das tabelas 6.4 e 6.6.

Tabela 6.8: Sobrecarga relativa referente à captura de exceções

Num Exc.	nprof	nprof+mon	mon	(nprof+mon)-mon	diff-nprof
0	5%	42%	36%	6%	2%
80	4%	63%	57%	7%	2%
160	4%	89%	81%	8%	4%
240	4%	113%	104%	9%	4%
320	4%	135%	125%	10%	6%
400	5%	159%	148%	12%	7%

As colunas nprof, nprof+mon e (nprof+mon)−mon da tabela 6.8 estão representados graficamente na figura 6.10.

Pela inspeção da figura 6.10, identifica-se uma possível relação linear entre a sobrecarga imposta pelo monitor (curvas nprof+mon e mon). Esta possibilidade é confirmada pelo coeficiente de correlação linear entre a sobrecarga nprof+mon e o número de capturas de exceções, no valor de 0,999331, e entre a sobrecarga de mon e o número de capturas de exceções, 0,999456. Por outro lado, o custo de Nprof é aproximadamente constante, com o valor de 4%.

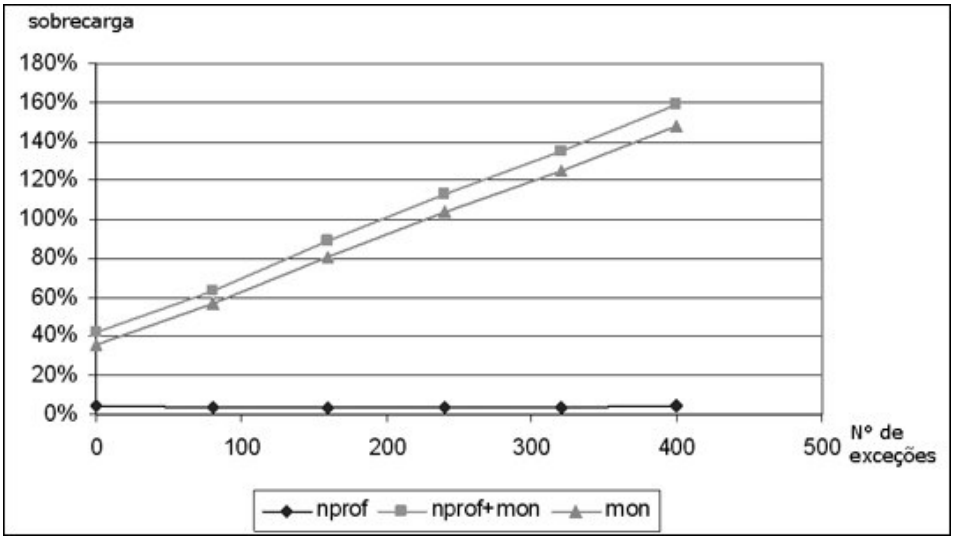


Figura 6.10: Sobrecarga relativa à captura de exceções

Entretanto, quando o Nprof está rodando junto com o monitor de capturas de exceções, o custo do Nprof cresce, provavelmente devido à interação entre eles. Este efeito pode ser visualizado na figura 6.11, em que são apresentadas as curvas de nprof, com os valores medidos sem a existência de um monitor, e (nprof+mon)-mon, quando surgem os efeitos da interação entre Nprof e o monitor.

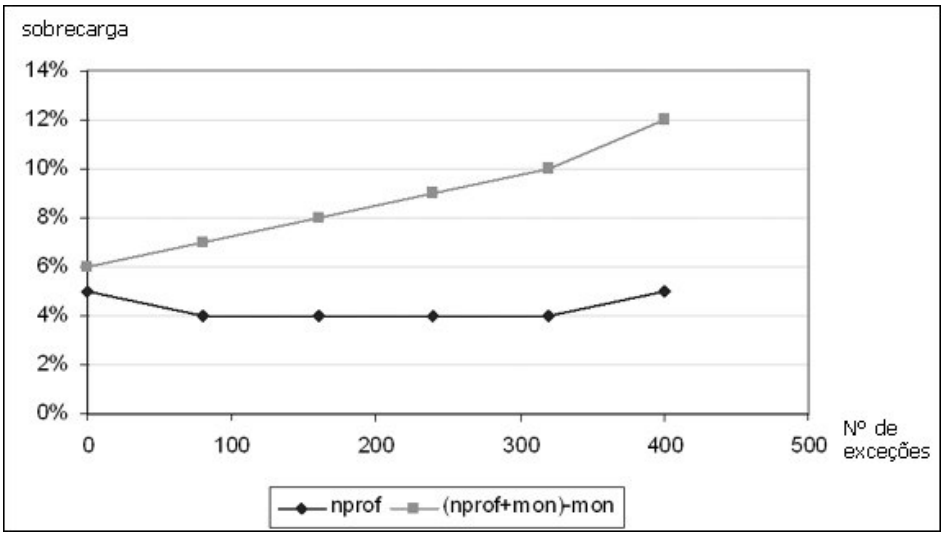


Figura 6.11: Sobrecarga relativa do Nprof a partir de captura de exceções

O custo medido da sobrecarga do monitor de exceções é alto, mesmo quando não foram acrescentadas exceções como carga de teste (primeira linha na tabela 6.8). Esta característica deve ser observada principalmente quando a aplicação a ser monitorada

for de tempo real, a fim de verificar se as exigências temporais são atendidas. Tal sobrecarga ocorre, entretanto, devido às exceções ocorridas na própria aplicação-alvo e nas respectivas classes da API Java que a aplicação utiliza, principalmente durante a fase de inicialização da aplicação-alvo. O número de exceções lançadas pela JVM-alvo na situação em que não é gerada explicitamente nenhuma exceção pela aplicação de teste é de aproximadamente 140.

Finalmente, é necessário considerar-se que o custo do monitor de capturas de exceções também sofre influência do número de exceções geradas pela aplicação-alvo. Este, por sua vez, depende do perfil da aplicação a ser monitorada.

6.3.7 Testes de monitores de memória, *sockets* e ambiente de execução (*runtime*)

Para verificar a sobrecarga causada pelos monitores de memória, *sockets* e ambiente de execução, foram realizados testes com cada um deles sem o Nprof, com o Nprof, porém sem o monitor ativo, e com o Nprof e monitor. Cada teste foi executado 20 vezes, e da amostra foram excluídos o maior e o menor valores. A tabela 6.9 contém os tempos de execução e da sobrecarga relativa. Os tempos estão em milisegundos. A figura 6.12 compara a sobrecarga causada por cada um dos monitores.

Tabela 6.9: Tempos de execução dos monitores e sobrecarga relativa

	Tempos			Sobrecarga		
	sem nprof	com nprof	nprof+mon	nprof	nprof+mon	mon
Socket	5340	5526	5510	3%	3%	0%
Memória	5327	5561	5531	4%	4%	0%
Runtime	5334	5589	5627	5%	5%	1%

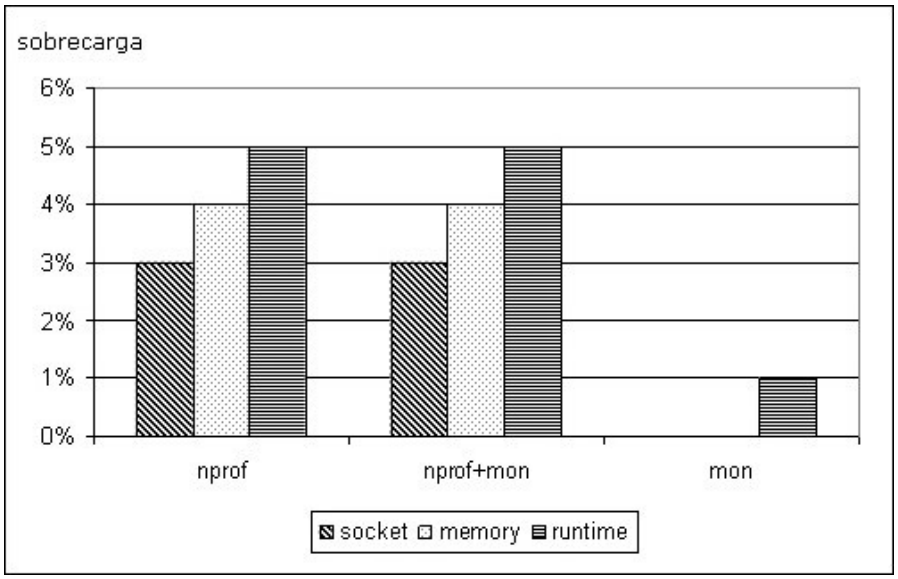


Figura 6.12: Comparativo de sobrecarga entre monitores *socket*, memória e *runtime*

Nota-se, a partir da tabela 6.9 e da figura 6.12, que a sobrecarga causada é baixa, e que a sobrecarga causada pela ativação dos monitores é praticamente desprezível.

6.3.8 Considerações sobre o estudo de caso

As medições das sobrecargas devidas aos diversos monitores avaliados permitem algumas constatações. Em primeiro, os monitores responsáveis por provocar a maior sobrecarga no sistema são os de classes e exceções. Esses monitores possuem em comum o fato de serem baseados em *tracing*: recebem mensagens contendo eventos ocorridos na aplicação-alvo. Para esses monitores também se verificou uma alta sobrecarga, mesmo quando não aplicada uma carga específica relacionada ao monitor. Isso se deve ao lançamento de exceções e a carga de classes inerente à própria aplicação-alvo.

Outra constatação refere-se aos outros dois monitores para os quais se gerou uma carga sintética: o de *threads* e o de objetos. Esses apresentaram uma sobrecarga pequena para valores baixos de carga aplicada. É interessante notar que ambos os monitores se utilizam da técnica de amostragem. Nota-se também uma semelhança entre a sobrecarga imposta pelos monitores de *socket*, memória e *runtime* (para os quais não foi gerada carga específica), e aquela dos monitores de *threads* e objetos, quando medidos (mesmo com monitores ativados) sem a aplicação de carga de trabalho específica: tais casos possuem uma sobrecarga baixa (entre 4% e 7%).

6.4 Monitoramento de aplicação distribuída: JavaGroups

Visto que uma das principais características da ferramenta Nprof é o monitoramento de aplicações distribuídas por meio de interceptação de chamadas *socket*, optou-se por realizar um estudo de caso com base em um *framework* para comunicação *multicast* confiável, o JavaGroups (BAN, 1998). Tal *framework* permite a criação de canais de comunicação *multicast* confiável que podem apresentar diferentes níveis de confiança de entrega, dependendo da configuração realizada. Visto que JavaGroups utiliza *multicast*, as mensagens enviadas através dos canais de comunicação são, em última instância, mensagens UDP, as quais são capturadas pelo monitor de *sockets*.

JavaGroups permite a configuração do nível de confiança de entrega das mensagens enviadas por meio da utilização de uma pilha configurável de sub-protocolos, que possuem funções específicas como: fragmentação de mensagens (FRAG); respostas negativas de confirmação (NACK); protocolo de *membership* (GMS); dentre outros. Tal pilha de protocolos é configurada no momento de criação do canal de comunicação JavaGroups, que se responsabiliza pela configuração e inicialização de cada um dos sub-protocolos da pilha.

Por permitir uma configuração de protocolos relativamente complexa, pode ser difícil perceber o funcionamento exato de uma determinada configuração de pilha de protocolos, e como tal conjunto pode garantir a entrega ou *membership*. Um dos objetivos do presente estudo de caso é verificar a influência dos parâmetros e pilhas de protocolo escolhidos na execução da aplicação-alvo.

6.4.1 Configuração dos testes

Para realizar testes com o *framework* JavaGroups, criou-se uma aplicação teste que utiliza os canais de comunicação criados para enviar e receber mensagens de tamanhos fixos. Cada instância da aplicação-alvo envia um determinado número de mensagens *multicast* e, após o término do envio das mensagens, espera um tempo de 5 segundos a fim de receber possíveis mensagens retardatárias e termina sua execução. A fim de

verificar possíveis diferenças no comportamento da aplicação-alvo, criaram-se diferentes configurações a partir da variação de alguns parâmetros de execução. Tais parâmetros são:

- pilha de protocolos: foram utilizadas três pilhas de protocolos diferentes, de diferentes níveis de complexidade;
- número de mensagens: 100 ou 1000 mensagens;
- tamanho das mensagens: 100 ou 1000 *bytes*;
- espera entre envios sucessivos de mensagens: 0 ou 20 milisegundos.

As pilhas completas de protocolos utilizados nos testes são:

UDP: apenas UDP.

VERIFY: UDP:PING:MERGE2:FD_SOCK:VERIFY_SUSPECT – permite a detecção de falhas no envio de mensagens e verificação de nodos suspeitos de falhas.

GMS: UDP:PING:MERGE2:FD_SOCK:VERIFY_SUSPECT:pbcast.NAKACK :UNICAST:pbcast.STABLE:FRAG:CAUSAL:pbcast.GMS – permitem retransmissão, ordenamento causal e *membership*, entre outros.

A utilização de tais pilhas de protocolos se deve ao fato de serem pilhas de protocolos utilizadas em aplicações de demonstração distribuídas junto com o *framework* JavaGroups. Isso porque muitos dos subprotocolos dependem de funcionalidades existentes em outros subprotocolos; portanto, a configuração de uma pilha de protocolos JavaGroups não é uma tarefa trivial.

Verificou-se, por meio de testes iniciais com diferentes pilhas de protocolos, que a variação do número e tamanho das mensagens, dentro dos limites estabelecidos, não traria grandes contribuições na análise das propriedades das diferentes pilhas de protocolos. Decidiu-se, portanto, por centralizar os casos de teste na variação do tempo de espera, entre mensagens sucessivas (0 ou 20 ms).

Os três primeiros conjuntos de testes, cada um correspondendo a uma configuração, foram realizados com a pilha de protocolos UDP. Variaram-se os parâmetros de número de mensagens, tamanho de mensagens e tempo de espera entre envios sucessivos de mensagens. Nas quatro últimas configurações, modificaram-se apenas a pilha de protocolos utilizada e o tempo de espera entre envios sucessivos de mensagens. A tabela 6.10 resume as configurações e suas combinações de parâmetros.

Tabela 6.10: Configurações dos testes com JavaGroups

Configuração	Protocolos	Num Msg	Tam Msg	Tempo espera
1	UDP	1000	100	20
2	UDP	1000	100	0
3	UDP	100	1000	20
4	VERIFY	100	1000	20
5	VERIFY	100	1000	0
6	GMS	100	1000	20
7	GMS	100	1000	0

A aplicação distribuída foi executada em três máquinas distintas, enquanto uma quarta executava a ferramenta Nprof. Para que as instâncias da aplicação distribuída começassem suas execuções simultaneamente, elas foram inicializadas em estado suspenso e então reiniciadas pela aplicação Nprof. Para a realização dos testes foram utilizados computadores Pentium IV com 1,8GHz e 512Mb RAM interligados por uma rede Fast Ethernet e sistema operacional Windows XP SP2 com máquina virtual Java 1.5.0.

Cada configuração teve um mínimo de três execuções, tendo cada uma coletado dados de três instâncias da aplicação distribuída, totalizando um mínimo de nove medidas para cada uma das configurações. Para cada execução, os dados coletados foram:

- Tempo de execução;
- Número de mensagens enviadas;
- Número de *bytes* enviados;
- Número de mensagens recebidas; e
- Número de *bytes* recebidos.

O objetivo dos testes realizados é o de verificar a influência dos parâmetros escolhidos (pilha de protocolos, número e tamanho de mensagens e tempo de espera) por meio da análise dos dados coletados. Dentre as possíveis influências, destacam-se:

- Influência do número e tamanho das mensagens no tempo de execução;
- Influência da pilha de protocolos no número de mensagens enviadas;
- Influência da pilha de protocolos no tamanho das mensagens enviadas.

Ao executar os primeiros testes com a aplicação distribuída JavaGroups, verificou-se que, em diversos casos, nem todas as mensagens eram recebidas pelos outros membros da aplicação (sem o uso de protocolo de entrega confiável). Decidiu-se, assim, criar um temporizador entre o envio de mensagens sucessivas, que poderia estar ativado (com tempo de espera em 20 milisegundos) ou desligado, conforme explícito nos parâmetros na tabela 6.10 (tempo de espera), a fim de procurar minimizar as perdas de pacotes. Em todos os casos, o tempo de 20 milisegundos foi suficiente para evitar a perda de pacotes. Ao mesmo tempo, as perdas acabaram por ser benéficas aos testes por permitirem a avaliação dos protocolos de recuperação de falhas do JavaGroups.

6.4.2 Configurações 1, 2 e 3

As configurações 1, 2 e 3 possuem a propriedade comum de utilizar apenas o protocolo UDP. Entretanto, as configurações 1 e 3 utilizam temporizadores de 20 milisegundos enquanto a configuração 2, não. As configurações 1 e 2 utilizam 1000 mensagens de 100 *bytes* enquanto a configuração 3 utiliza 100 mensagens de 1000 *bytes*. Para cada configuração foram tomadas nove medidas. A tabela 6.11 mostra os resultados obtidos.

Tabela 6.11: Resultado da execução das configurações 1,2 e 3

Conf	Tempo (ms)	IC 90%	Msg out	Msg in	IC 90%	Tx perda	Msg size
1	41307	± 161	1000	3000	-	0,00%	146
2	9814	± 84	1000	2453	± 74	18,24%	146
3	12778	± 90	100	300	-	0,00%	1046

Na tabela 6.11, a primeira coluna (Tempo-ms) apresenta o tempo médio de execução das instâncias da aplicação distribuída, e a segunda (IC 90%), o intervalo de confiança, com nível de significância de 90%, para a média do tempo de execução. Note-se que o tempo de execução sofre grande variação devido ao uso do temporizador, quando ativado (20 milisegundos entre duas mensagens sucessivas). A quarta (Msg out) e quinta (Msg in) colunas indicam o número de mensagens enviadas e recebidas, respectivamente. A sexta coluna (IC 90%) mostra o intervalo de confiança, com nível de significância de 90%, considerando o número médio de mensagens recebidas em relação à média amostral (coluna 5). A sétima coluna (Tx perda) apresenta a taxa de perda de mensagens e a oitava (Msg size), o tamanho médio das mensagens enviadas e recebidas, como coletadas pelo Nprof.

Note-se que, por pertencerem a um grupo que utiliza um canal de comunicação *multicast*, as aplicações acabam por receber as próprias mensagens que enviam. Portanto, se cada aplicação envia 1000 mensagens, é esperado que cada uma receba 3000 mensagens (visto que três instâncias da aplicação executam concorrentemente). A partir dos dados coletados, foram possíveis as seguintes observações:

- A perda de pacotes ocorreu somente na configuração 2, a única que não utilizou temporizador entre o envio de mensagens sucessivas. Tal fato se verifica porque o número de mensagens recebidas é menor que o triplo do número de mensagens enviadas;
- O tamanho apurado das mensagens enviadas e recebidas foi maior do que o tamanho original na aplicação teste. Isso ocorreu porque, ao utilizar os canais *multicast* do JavaGroups, estes incluem informações em todos os pacotes enviados dependendo da pilha de protocolos configurada, ou seja, a pilha de protocolos utilizada gerou sobrecarga de tráfego na rede. Notou-se que o incremento no tamanho das mensagens foi invariavelmente de 46 *bytes* para cada mensagem;
- Ao avaliar as configurações 1 e 3, se verificou que, para tais casos, o uso dos canais de comunicação do JavaGroups não gera mensagens adicionais, como possíveis mensagens para confirmação de envio ou descoberta de grupo, visto que o número de mensagens recebidas foi invariavelmente de três vezes o número de mensagens enviadas por cada um dos nodos da aplicação-alvo isoladamente.

6.4.3 Configurações 3, 4, 5, 6 e 7

As configurações 3 a 7 diferem, entre si, em pontos distintos dos do grupo de configurações anterior. Para este grupo, o número de mensagens enviadas por cada nodo e os tamanhos das mensagens são constantes, sendo 100 o número de mensagens e 1000 *bytes* seus tamanhos. Para as configurações do presente grupo, os parâmetros variáveis são: a pilha de protocolos (três pilhas diferentes) e o tempo de espera entre envios

sucessivos de mensagens (0 ou 20 milisegundos). A tabela 6.12 apresenta as médias dos dados coletados para tempo de execução dos testes, número de mensagens enviadas e recebidas e respectivos intervalos de confiança para as médias obtidas. Para as configurações 3, 4, 5 e 6 foram tomadas nove medidas. Para a configuração 7 foram consideradas 15 medidas, em função das particularidades ocorridas em tal configuração, explicadas mais adiante.

Tabela 6.12: Execução das configurações 3 a 7 (1)

Conf	milis	Prot.	tempo	IC 90%	IC/méd	msg out	IC 90%	IC/méd	msg in	IC 90%	IC/méd
3	20	UDP	12778	± 90	0,71%	100,0	-	-	300,0	-	-
4	20	VERIFY	13159	± 136	1,03%	100,0	-	-	300,0	-	-
5	0	VERIFY	9804	± 72	0,73%	100,0	-	-	281,6	± 3,37	1,20%
6	20	GMS	14828	± 389	2,62%	116,7	± 2,28	1,95%	328,5	± 1,53	0,47%
7	0	GMS	15312	± 3293	21,51%	122,2	± 6,44	5,27%	321,7	± 0,95	0,30%

Na tabela 6.12 as três primeiras colunas apresentam os parâmetros de execução dos testes. Note-se que a terceira coluna, que indica a pilha de protocolos utilizada, mostra apenas o protocolo de mais alto nível utilizado. Na quarta coluna (tempo) se encontra o tempo de execução médio dos nodos da aplicação distribuída. Na quinta coluna (IC 90%), o intervalo de confiança de 90% para a média do tempo de execução, e na sexta coluna (IC/méd), o coeficiente de variação do intervalo de confiança, ou seja, a relação percentual entre o intervalo de confiança e a média do tempo de execução. A sétima coluna (msg out) apresenta o número médio de mensagens enviadas e as colunas 8 e 9 (IC 90% e IC / tot) mostram o intervalo de confiança para a média de mensagens enviadas e seu coeficiente de variação. A décima coluna (msg in) indica o número médio de mensagens recebidas por nodo da aplicação distribuída, e as colunas 11 e 12 (IC 90% e IC/méd), o intervalo de confiança e a relação entre o intervalo de confiança e a média amostral (ou coeficiente de variação do intervalo de confiança), respectivamente.

A primeira observação que pode ser feita quanto aos dados coletados é que, pelo uso de pilhas de protocolos mais complexas, o tempo de execução médio aumenta, como é de se esperar. Isso se constata comparando-se as configurações 3,4 e 6, pois todas elas possuem o mesmo tempo de espera entre envios sucessivos de mensagens (de 20ms). Ao usar a pilha de protocolos `UDP`, tem-se o tempo médio de 12778ms. Com a pilha `VERIFY`, o tempo médio passa para 13159ms, um incremento de 3%. Com a pilha `GMS`, porém, o tempo médio passa para 15312ms, um incremento de 12,7% sobre o tempo médio utilizado com a pilha `VERIFY`.

Uma segunda observação sobre a diferença de tempo entre as configurações 6 e 7 - ambas utilizando pilha `GMS`, porém a última sem temporização entre envios de mensagens - é que a configuração 7 gasta mais tempo, em média, que a configuração com temporização. Em tal observação deve-se considerar que o intervalo de confiança para o tempo de execução da configuração 7 é muito superior ao das demais. Foi constatado que esse comportamento deveu-se ao fato de que, na configuração 7, o mecanismo de recuperação de falhas da pilha de protocolos `GMS` foi ativado, e então foi realizada a retransmissão dos pacotes considerados perdidos. Como essa situação de detecção de perda de pacotes e retransmissão não ocorreu nas outras execuções, explica-se o grande intervalo de confiança nas colunas 5 e 6. De fato, a ativação do mecanismo

de retransmissão motivou medições extras da configuração 7 para melhor avaliar tal fenômeno.

Outra observação diz respeito à pilha de protocolos `GMS`. Ao passo que com as outras pilhas de protocolos só foi enviada a quantidade de mensagens estipulada pela aplicação (100), ao utilizar a pilha de protocolos `GMS` notou-se que foram enviados mais pacotes do que os especificados. Isso decorre da utilização de protocolos com funcionalidades adicionais, como o de *membership*, para o qual são necessárias mensagens extras para a formação de grupo e eleição de coordenador.

Ainda outra observação relevante diz respeito à média e ao intervalo de confiança do número de mensagens enviadas na configuração 7. Nota-se que o intervalo de confiança é amplo, e a média de mensagens enviadas maior que na configuração 6. Isso pode ser explicado pelo mesmo motivo do tempo médio de execução dessa configuração ser alto: o mecanismo de recuperação de falhas da configuração 7 realizou o reenvio das mensagens perdidas. E, conforme foi constatado, a retransmissão de mensagens perdidas não ocorreu em todas as execuções da configuração 7 (o que justifica a alta variabilidade de tempo de execução e de mensagens enviadas de tal configuração). Note-se, entretanto, que o número de mensagens recebidas e seu intervalo de confiança não diferem muito entre as configurações 6 e 7, o que pode indicar que a retransmissão de pacotes obteve sucesso.

A tabela 6.13 traz outros dados relativos às execuções das configurações 3 a 7. De forma análoga à tabela 6.12, ela mostra o número médio de *bytes* enviados e recebidos, e seus respectivos intervalos de confiança.

Tabela 6.13: Execução das configurações 3 a 7 (2)

Conf	Milis	Prot	Bytes out	IC 90%	IC %	Bytes in	IC 90%	IC %
3	20	UDP	104600,0	-	-	313800,0	-	-
4	20	VERIFY	104600,0	-	-	313800,0	-	-
5	0	VERIFY	104600,0	-	-	294523,7	± 3449,98	1,17%
6	20	GMS	129158,2	± 1285,09	0,99%	383911,0	± 1386,13	0,36%
7	0	GMS	135067,2	± 4965,82	3,68%	380878,6	± 1485,56	0,39%

A exemplo do ocorrido na tabela 6.12, verifica-se que, na configuração 7 da tabela 6.13, o intervalo de confiança para a média de *bytes* enviados é alto, o que é reflexo também da retransmissão de pacotes ocorrida em algumas execuções. Note-se que o número de *bytes* recebidos na configuração 7 está mais próximo do obtido na configuração 6, na qual não há perda de pacotes, revelando indício de que a retransmissão de pacotes da pilha de pacotes `GMS` foi eficaz.

Uma observação interessante diz respeito ao tamanho médio das mensagens enviadas, levando-se em conta as diferentes configurações utilizadas. Nota-se que nas configurações 3, 4 e 5, as mensagens tem tamanho fixo de 1046 *bytes* e a sobrecarga imposta pela pilha de protocolos utilizada é, portanto, de 46 *bytes*. Já nas configurações 6 e 7 o tamanho médio das mensagens enviadas é maior, e, como conseqüência, a sobrecarga também é maior. A tabela 6.14 mostra o tamanho médio das mensagens e a sobrecarga média por mensagem. Tal fato se deve a prováveis dados inseridos nas mensagens enviadas a fim de garantir entrega, detecção de defeitos ou *membership* para os protocolos superiores da pilha `GMS`.

Tabela 6.14: Tamanho médio dos pacotes e mensagens

Conf	Tam msg	bytes out	msg out	bytes / msg	sobrecarga / msg
3	1000	104600,0	100,0	1046,0	46,0
4	1000	104600,0	100,0	1046,0	46,0
5	1000	104600,0	100,0	1046,0	46,0
6	1000	129158,2	116,7	1107,0	107,0
7	1000	135067,2	122,2	1105,8	105,8

Outra observação relevante, que se refere tanto à tabela 6.12 quanto à tabela 6.13 é que, ao contrário das pilhas de protocolos `UDP` e `VERIFY`, nas pilhas de protocolos `GMS` (configurações 6 e 7), o número de mensagens recebidas esperado não é, necessariamente, o triplo do número de mensagens enviadas (como seria de se esperar para mensagens enviadas por *multicast* para um grupo de 3 nodos). Isso ocorre porque os protocolos de mais alto nível de tais configurações não usam somente mensagens *multicast*. Uma possibilidade é de que o protocolo de retransmissão de mensagens ou algum protocolo de verificação de defeitos envie mensagens *unicast*. Esta diferença entre o número de mensagens e o triplo de mensagens enviadas pode ser notada na figura 6.13, que é uma captura de tela da ferramenta Nprof para um nodo da aplicação com pilha de protocolos `GMS`.

DatagSock	Bytes Out	Bytes In	Pckt Out	Pckt In	Port	LPort	Addr	LAddr
Total	127249	383478	112	322				
0	0	126579	0	103	1175	7600	143.54.12.228	0.0.0.0
1	126579	0	103	0	7600	1175	228.8.8.8	0.0.0.0
2	224	450	3	3	1178	1174	143.54.12.114	143.54.12.228
3	0	126786	0	104	1179	7600	143.54.12.114	0.0.0.0
4	0	128593	0	108	1144	7600	143.54.12.48	0.0.0.0
5	446	1070	6	4	1143	1174	143.54.12.48	143.54.12.228

Figura 6.13: *Sockets* UDP de um nodo da aplicação JavaGroups

A figura 6.13 apresenta uma captura de tela do monitoramento de *sockets* realizado pela ferramenta Nprof em um dos nodos da aplicação distribuída. Em tal figura, é possível identificar todos os *sockets* UDP abertos pela aplicação-alvo (numerados de 0 a 5) após o término de uma execução da aplicação distribuída, utilizando a pilha de protocolos `GMS`. Por meio do endereço remoto de cada socket (8ª coluna - Addr), é possível identificar se é um *socket unicast* ou *multicast* (*sockets multicast* usam a faixa de endereços IP de 224.0.0.0 a 239.255.255.255). Sendo assim, apenas o *socket* identificado como 1 é um *socket multicast*. Entretanto o endereço IP de tal *socket*, 228.8.8.8, é utilizado apenas para envio de mensagens; as mensagens *multicast* recebidas vêm com o endereço da máquina que de fato enviou a mensagem. Analisando os *sockets* que foram utilizados para enviar mensagens, identifica-se, então, que há mensagens que não são enviadas para o endereço *multicast* do grupo, mas para endereços *unicast* de membros específicos do grupo, como no caso dos *sockets* 2 e 5 (Figura 6.13, coluna *Bytes Out* e *Pckt Out*). Tais sockets (2 e 5) são, portanto, *sockets unicast* entre os membros do grupo e os *sockets* identificados pelos números 0, 3 e 4 identificam os canais *multicast* dos quais foram recebidas mensagens.

Como dito anteriormente, foi verificado, para a configuração 7, que em parte das execuções houve retransmissão de mensagens devido ao mecanismo de entrega confiável. A fim de identificarem-se mais claramente as diferenças entre os casos de retransmissão e não retransmissão para as execuções de tal configuração, agruparam-se os casos em dois grupos, um com e outro sem retransmissão, como consta na tabela

6.15. Note-se que em apenas quatro medidas identificou-se a situação de retransmissão de pacotes (coluna Nº exec), enquanto em 11 execuções tal retransmissão não foi identificada. O intervalo de confiança calculado se baseia na quantidade de medidas tomadas. Para melhor comparação entre os grupos definidos, incluiu-se a informação de desvio padrão (coluna Desvio).

Tabela 6.15: Agrupamento das execuções com e sem retransmissão (1)

	Nº exec	Tempo	Desvio	IC 90	IC %	msg out	Desvio	IC 90	IC %
Total (g1+g2)	15	15311,53	± 7242	± 3293	21,51%	122,15	± 14,2	± 6,44	5,27%
grupo 1	11	11108,64	± 754	± 412	3,71%	113,89	± 5,7	± 3,11	2,73%
grupo 2	4	26869,50	± 56	± 65	0,24%	140,75	± 7,1	± 8,40	5,96%
g2 / g1	-	142%	-	-	-	24%	-	-	-

Na tabela 6.15, o grupo 1 indica as execuções que não efetuaram retransmissão de mensagens e o grupo 2 as que efetuaram. O grupo 1 é constituído por 11 execuções e o grupo 2, de quatro (coluna Nº exec). Note-se que, apesar de a configuração 7 possuir alta variabilidade para tempo de execução e número de mensagens enviadas (tabela 6.12), tal variabilidade reduz-se bastante para as execuções agrupadas, em especial quanto ao fator tempo. Isso indica que as execuções dentro de um mesmo grupo comportam-se de maneira semelhante, caracterizando dois grupos distintos (grupos 1/*sem* e 2/*com* retransmissão). É também interessante notar o incremento no tempo de execução do grupo 2 em relação ao grupo 1: um aumento de 141%. No número de mensagens enviadas verificou-se, também, incremento; neste caso menor, de 24%. De fato, é de se esperar que, em execuções em que haja retransmissão de mensagens, tanto o tempo quanto o número de mensagens enviadas aumentem.

Na tabela 6.16 encontram-se dados sobre as execuções da configuração 7 agrupadas pelo mesmo critério da tabela 6.15 (*com* e *sem* retransmissão), porém contendo os dados referentes a mensagens recebidas. Note-se que o número de mensagens recebidas de ambos os grupos é bastante similar (o grupo 2 apresenta um decréscimo no número de mensagens recebidas de apenas 0,53%), o que é um forte indício de que pelo menos grande parte das mensagens retransmitidas foram recebidas.

Tabela 6.16: Agrupamento das execuções com e sem retransmissão (2)

	Nº exec	msg in	Desvio	IC 90	IC %
Total (g1+g2)	15	321,69	± 2,10	± 0,95	0,30%
grupo 1	11	322,22	± 1,99	± 1,09	0,34%
grupo 2	4	320,50	± 2,08	± 2,45	0,76%
g2 / g1	-	-0,53%	-	-	-

A possibilidade da retransmissão de pacotes, quando do uso da pilha de pacotes GMS, pode ainda ser confirmada pela figura 6.14, que apresenta uma captura de tela da interface Nprof de uma execução em que os tempos de execução de membros da aplicação-alvo foram bem superiores à média (pertencentes ao equivalente do grupo 2 da tabela 6.15).

DatagSock	Bytes Out	Bytes In	Pckt Out	Pckt In	Port	LPort	Addr	LAddr
Total	132217	382184	141	320				
0	0	126579	0	103	1169	7600	143.54.1...	0.0.0.0
1	3611	5857	28	8	1139	1168	143.54.1...	143.54.1...
2	126579	0	103	0	7600	1169	228.8.8.8	0.0.0.0
3	0	119298	0	100	1175	7600	143.54.1...	0.0.0.0
4	2027	8209	10	7	1174	1168	143.54.1...	143.54.1...
5	0	122241	0	102	1140	7600	143.54.1...	0.0.0.0

Figura 6.14: *Sockets* de aplicação-alvo com pilha de protocolos `GMS`

A figura 6.14 mostra os *sockets* existentes para um nodo da aplicação-alvo após sua execução. Nota-se que o número de pacotes enviados (103) pelo nodo selecionado para o endereço *multicast* (socket 2) é bem próximo do definido para a configuração (100), porém há um número considerável de pacotes (28 do socket 1 mais 10 do socket 4, totalizando 38) que são pacotes enviados a nodos específicos da aplicação distribuída. Visto que o somatório de pacotes enviados (141) é bem superior à média das execuções de tal configuração, reforça-se a idéia de que se trata de uma situação excepcional, como a de retransmissão de pacotes após a detecção de erro de transmissão.

6.4.4 Considerações sobre o estudo de caso

O estudo de caso sobre uma aplicação distribuída utilizando JavaGroups conseguiu atingir os objetivos propostos: a partir do monitoramento da aplicação distribuída com a ferramenta Nprof foi possível identificar relações entre as configurações das pilhas de protocolos (e demais parâmetros) e as informações de execução coletadas como, por exemplo, a relação entre as configurações de protocolos e o número de mensagens enviadas, bem como a influência da pilha de protocolos no tamanho das mensagens enviadas. De modo geral, reconheceu-se que as pilhas de protocolos mais complexas geravam sobrecarga temporal, e também que o uso da pilha de protocolos identificada pelo protocolo `GMS` faz com que não somente mais mensagens sejam enviadas, como também que o tamanho médio das mensagens seja maior do que ao utilizar-se pilhas de protocolos mais simples.

Como resultado não esperado, o monitoramento da aplicação distribuída permitiu que fosse identificada a ativação dos mecanismos de detecção de defeitos e retransmissão de mensagens do *toolkit* JavaGroups, e quantificar a sobrecarga gerada por tal mecanismo. Tal fenômeno ocorreu devido à perda de mensagens UDP, não previstas na definição dos testes.

Entendeu-se que tal experimento é uma aproximação razoável de aplicações reais utilizadas na academia e na indústria, e que experimentos semelhantes podem ser empregados em aplicações existentes para avaliar o funcionamento e a sobrecarga de diferentes protocolos. Por meio do uso da ferramenta Nprof, o usuário poderia perceber se sua aplicação, no que tange a comunicação de *sockets*, condiz com a especificação quanto ao número e propriedades dos *sockets* que são abertos, qual o volume de tráfego de tais *sockets*, para assim obter embasamento para ações corretivas e de otimização sobre sua aplicação.

7 CONCLUSÕES

O monitoramento de sistemas computacionais possui diversas utilidades para os desenvolvedores e administradores de sistemas, tais como depuração, otimização, auditoria, segurança, entre outros. Destacou-se, no presente trabalho, o uso de monitoramento para fins de otimização e depuração de sistemas, normalmente aplicados na fase de desenvolvimento de sistemas.

Um dos objetivos do trabalho é a exploração de técnicas para o monitoramento de aplicações Java, em especial para aplicações distribuídas. Para tanto, foi desenvolvida a ferramenta Nprof, que se utiliza das novas APIs existentes na versão 1.5 do ambiente Java e permite o monitoramento de aplicações distribuídas ou isoladas. Entretanto, além de buscar as funcionalidades presentes nas novas APIs, procurou-se manter práticas já consolidadas para o monitoramento de aplicações Java.

A ferramenta utiliza uma arquitetura de camadas extensíveis de coleta e visualização de informações. Diversos coletores e visualizadores foram desenvolvidos para a ferramenta. Cada coletor (ou monitor) tem a responsabilidade de coletar um tipo específico de dados da aplicação-alvo, enquanto cada visualizador permite a visualização dos dados capturados por um ou mais monitores. Pode-se destacar, de forma especial, o monitor de *sockets*, que permite a coleta de dados sobre envio e recebimento de mensagens e que se considera de grande importância para o monitoramento de aplicações distribuídas por permitir analisar como os diferentes nodos de uma aplicação distribuída se comunicam.

A fim de validar a ferramenta, foram propostos três estudos de caso. Cada estudo de caso teve um propósito específico, porém é interessante observar que qualquer dos estudos de caso pode se assemelhar com uma aplicação real e mesmo existente em uso na indústria ou na academia.

O primeiro estudo de caso tratou justamente de verificar as possibilidades de monitoramento de uma aplicação de largo uso no meio comercial: a aplicação Tomcat, *container* de aplicações *web* dinâmicas. Por meio da ferramenta Nprof, foi possível perceber, por exemplo, o comportamento dos estados das *threads* da aplicação e sua relação com a criação de *sockets* para atender as requisições HTTP.

O segundo estudo de caso centrou-se na análise da sobrecarga gerada pelos diversos componentes da ferramenta Nprof, que podem ser ativados de forma conjunta ou individual. Em tal estudo de caso não apenas foi testada a sobrecarga da ferramenta com os diversos monitores, como também foi criada uma carga sintética para avaliar o

comportamento de tais monitores. Esses testes foram realizados com diferentes cargas de trabalho para melhor avaliar a sobrecarga sofrida pela ferramenta. Ao se realizarem testes com diferentes conjuntos de monitores ativados em cada momento, foi possível também verificar a influência de cada monitor na sobrecarga causada à aplicação-alvo.

O terceiro estudo de caso foi realizado para avaliar o monitoramento de uma aplicação distribuída a partir do Nprof. Para tanto utilizou-se o *toolkit* JavaGroups de comunicação de grupos e, a partir deste, foi desenvolvida uma aplicação de teste para envio e recebimento de mensagens. Tal estudo de caso verificou a possibilidade de uso da ferramenta Nprof para monitoramento de aplicações distribuídas, uma vez que foi possível correlacionar informações dos diversos nodos em tempo de execução. Além disso, o estudo de caso possibilitou a avaliação de desempenho e funcionalidade do próprio *toolkit* JavaGroups, ao se utilizarem diversas configurações de teste.

Em resumo, entende-se que a ferramenta atingiu os principais objetivos propostos: possibilita o monitoramento de aplicações e a visualização de informações *online*, permite o monitoramento de uma aplicação distribuída ao possibilitar o monitoramento de diversos nodos concomitantemente, e utiliza-se das novas APIs Java de monitoramento, depuração e instrumentação. Deve-se notar, também, que muitos dos dados coletados pelos diferentes monitores inclusos na ferramenta são de possível uso para aplicações existentes, distribuídas ou não, e que tais aplicações podem utilizar o Nprof sem requerer qualquer alteração em seu código executável. Além disso, entende-se que a presente ferramenta Nprof inova no que diz respeito a seu monitor de *sockets*, na captura de dados recebidos e transmitidos via rede, sejam estes transmitidos via TCP ou UDP, possibilitando também a correlação de tais dados com os dos outros monitores da ferramenta Nprof.

7.1 Trabalhos Futuros

No decorrer do presente trabalho algumas propostas de melhorias e novas funcionalidades não foram desenvolvidas. Tais propostas caberiam como trabalhos futuros do atual projeto. Dentre os possíveis trabalhos, destacam-se alguns:

- Melhoria da visualização do sistema distribuído em relação as mensagens trafegadas entre os nodos, incluindo, por exemplo, a relação entre mensagens e seus respectivos tempos de envio e recebimento;
- Atualmente a ferramenta cria um *log* das mensagens recebidas e enviadas em cada nodo. Uma proposta é a criação de um componente para a reconstrução da ordem parcial de envio de mensagens entre nodos a partir dos arquivos de *logs* existentes em cada nodo e também das informações de diferenças de relógios entre os nodos. Tal componente poderia ser um módulo de processamento pós-morte ou ainda incorporado na própria ferramenta como um módulo *online* de visualização. A ferramenta LOrd (TRINDADE et al., 2006) possui uma proposta semelhante, e uma integração de tal ferramenta com Nprof seria bastante interessante;
- Pesquisar as APIs da futura versão da plataforma Java, atualmente em versão beta, com relação às de monitoramento e depuração. Dentre as novidades de tal versão, encontram-se a possibilidade de uso de múltiplas instrumentações de *bytecode* (BCI) concomitantes (SUN MICROSYSTEMS, 2006), como também a possibilidade de um módulo

JVMTI conectar-se a uma JVM sem a necessidade de esta ter sido inicializada com parâmetros especiais.

REFERÊNCIAS

ANDERSON, J. M. et al. Continuous Profiling: Where Have All the Cycles Gone? **ACM SIGOPS Operating Systems Review**, New York, v. 31, n. 5, p. 1-14, 1997.

APACHE SOFTWARE FOUNDATION. **Byte Code Engineering Library**. Disponível em: <http://jakarta.apache.org/bcel/>. Acesso em: maio 2004.

APACHE SOFTWARE FOUNDATION. **The Jakarta Site - Apache Tomcat**. Disponível em: <http://jakarta.apache.org/tomcat/>. Acesso em: ago. 2005.

APPPERFECT. **AppPerfect Java Profiler Data Sheet**. Disponível em: <http://www.appperfect.com/products/devsuite/jp.html>. Acesso em: jan. 2006.

ASHTON, P. **Algorithms for off-line clock synchronization**. [S.l.]: Department of Computer Sciences, University of Canterbury, 1987. (TR COSC 12/952).

AUSTIN, C. **J2SE 5.0 in a Nutshell**. Disponível em: <http://java.sun.com/developer/technicalArticles/releases/j2se15/>. Acesso em: maio 2004.

BAN, B. **Design and Implementation of a Reliable Group Communication Toolkit for Java**. Disponível em: <http://www.cs.cornell.edu/home/bba/papers.html>. Acesso em: out. 2005.

BORLAND. **Borland: Optimizeit Enterprise Suíte**. Disponível em: <http://www.borland.com/us/products/optimizeit/>. Acesso em: nov. 2004.

BORLAND. **Borland: JBuilder**. Disponível em: <http://www.borland.com/jbuilder>. Acesso em: jun. 2005 (a).

BORLAND. **Borland WorldWide**. Disponível em: <http://www.borland.com/>. Acesso em: jun. 2005 (b).

BRUGNARA, T.; CECHIN, S. L.; LISBÔA, M. L. B. Uma ferramenta para análise de desempenho de aplicações distribuídas. In: WPERFORMANCE, 4.; CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 25., 2005, São Leopoldo. **A Universidade da Computação**: um agente de inovação e desenvolvimento: anais. Porto Alegre: SBC, 2005. 1 CD-ROM.

CONSENS, M. P.; HASAN, M. Z.; MENDELZON, A. O. Visualizing and Querying Distributed Event Traces with Hy+. In: INTERNATIONAL CONFERENCE ON APPLICATIONS OF DATABASES, ADB, 1., 1994, Vadstena, Sweden.

Proceedings… Berlin: Springer, 1994. p123-141 (Lecture Notes in Computer Science, v. 819)

CHARLES, P.; HADDAD, J.; BITTEKER S. **Network Packet Capture Facility for Java**. Disponível em: <http://sourceforge.net/projects/jpcap>. Acesso em: maio 2005.

CHIBA, S.; NISHIZAWA, M. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In: INTERNATIONAL CONFERENCE ON GENERATIVE PROGRAMMING AND COMPONENT ENGINEERING, GPCE, 2., 2003, Erfurt, Germany. **Proceedings…** Berlin: Springer, 2003. p.364-376. (Lecture Notes in Computer Science, v. 2830).

CHUNG, M. **Using JConsole to Monitor Applications**. Disponível em: <http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html>. Acesso em: dez. 2004.

DAVIES, J. et al. An Aspect Oriented Performance Analysis Environment. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, 2., 2003, Boston, Massachusetts. **Proceedings…** [S.l.: s.n.], 2003. p. 17-21.

DEGIOANNI, L. et al. **Netgroup:** Analyzer: a public domain protocol analyzer. Disponível em: <http://analyzer.polito.it/>. Acesso em: maio 2005.

DMITRIEV, M. Profiling Java applications using code hotswapping and dynamic call graph revelation. In: INTERNATIONAL WORKSHOP ON SOFTWARE AND PERFORMANCE, WOSP, 4., 2004, Redwood Shores, California, USA. **Proceedings…** [S.l.: s.n.], 2004.

DUFOUR, B. et al. **Dynamic Metrics for Compiler Developers**. [S.l.]: McGill University, School of Computer Science, 2002. (Sable Technical Report SABLE-TR-2002-11).

ECLIPSE FOUNDATION. **The AspectJ Porject**. Disponível em: <http://eclipse.org/aspectj/>. Acesso em: maio 2005 (a).

ECLIPSE FOUNDATION. **Eclipse.org Main Page**. Disponível em: <http://www.eclipse.org/>. Acesso em: jun. 2005 (b).

EJ-TECHNOLOGIES. **Java Profiler – Jprofiler**. Disponível em: <http://www.ej-technologies.com/products/jprofiler/overview.html>. Acesso em: nov. 2004.

FARQUHAR, S.; CANNON-BROOKES, M. **Atlassian Profiling**. Disponível em: <http://opensource.atlassian.com/profiling/>. Acesso em: jul. 2005.

FENLASON, J.; STALLMAN, R. **The GNU Profiler**. Disponível em: <http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html> Acesso em: fev. 2005.

GAMMA, E. et al. **Design Patterns:** Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995.

GRAHAM, S.; KESSLER, P.; MCKUSICK, M. Gprof: A Call Graph Execution Profiler. **ACM SIGPLAN Notices**, New York, v.17, n.6, p120-126, Apr. 1982. Trabalho apresentado no SIGPLAN Symposium on Compiler Construction ,1982.

INTEL. **Intel VTune Performance Analyzer**. Disponível em: <http://www.intel.com/cd/software/products/asmo-na/eng/vtune/vpa/>. Acesso em fev. 2005.

JACOBSON V.; LERES C.; MCCANNE, S. **Tcpdump - dump traffic on a network**. Disponível em: <http://www.tcpdump.org/tcpdump_man.html>. Acesso em: jun. 2002.

JAIN, R. **The Art of Computer Systems Performance Analysis:** techniques for experimental design, measurement, simulation, and modeling. New York: John Wiley, 1991.

KAZI, I. et al. JaViz: A client/server Java profiling tool. **IBM Systems Journal,** New York, v. 39, n. 1, 2000.

KICKZALES, G. et al. Aspect Oriented Programming. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, ECOOP, 11., 1997, Jyväskylä, Finland. **Proceedings…** Finland: Springer, 1997. (Lecture Notes in Computer Science, v. 1241).

LAPRIE, J. Dependability - Its Attributes, Impairments and Means. In: RANDELL, B. et al. (Ed.). **Predictably Dependable Computing Systems**. Berlin: Springer-Verlag, 1995.

LEROUX, H. et al. A tool to dynamically visualize the execution of concurrent Java programs. In: INTERNATIONAL CONFERENCE ON PRINCIPLES AND PRACTICE OF PROGRAMMING IN JAVA, 2., 2003, Kilkenny City, Ireland. **Proceedings…** [S.l.: s.n.], 2003. p. 201–206. (ACM International Conference Proceeding Series, v. 42).

LIANG, S.; BRACHA, G. Dynamic class loading in the Java virtual machine. In: OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES AND APPLICATIONS, OOPSLA, 1998, Vancouver, Canada. **Proceedings…** [S.l.: s.n.], 1998, p. 36-44. (ACM SIGPLAN Notices, v. 33, n. 10).

LIANG, S.; VISWANATHAN, D. Comprehensive Profiling Support in the Java Virtual Machine. In: USENIX CONFERENCE ON OBJECT-ORIENTED TECHNOLOGIES AND SYSTEMS, COOTS, 5. **Proceedings…** [S.l.: s.n.], 1999.

LILJA, D. J. **Measuring Computer performance:** A practitioner's guide. Cambridge: Cambridge University, 2000.

MEHNER, K. JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs. In: DIEHL, S. (Ed.). **Software Visualization**. [S.l.]: Springer-Verlag, 2002. p.163-175. (Lecture Notes in Computer Science, v. 2269).

MICROSOFT. **Microsoft .NET Homepage**. Disponível em: <http://www.microsoft.com/net/>. Acesso em: abr. 2005.

OECHSLE, R.; GRONZ, O.; SCHÜLER, M. VisuSniff: A Tool for the Visualization of Network Traffic. In PROGRAM VISUALIZATION WORKSHOP, 2., 2002, HornstrupCentret, Dinamarca. **Proceedings…** [S.l.: s.n.], 2002. p. 118-124.

OECHSLE, R.; SCHMITT, T. JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI). In: DIEHL, S. (Ed.). **Software Visualization**. [S.l.]: Springer-Verlag, 2002. p. 176-190. (Lecture Notes in Computer Science, v. 2269).

O'HAIR, K. **The JVMPI Transition to JVMTI**. Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/jvmpitransition/>. Acesso em: jul. 2004.

O'HAIR, K. **Hprof:** A Heap/CPU Profiling Tool in J2SE 5.0. Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/HPROF.html>. Acesso em: fev. 2005.

PAUW, W. et al. Visualizing the Execution of Java Programs. In: DIEHL, S. (Ed.): **Software Visualization**. [S.l.]: Springer-Verlag, 2002. p. 151-162. (Lecture Notes in Computer Science, v. 2269).

RISSO, F. et al. **WinDump:** tcpdump for Windows. Disponível em: <http://www.winpcap.org/windump/>. Acesso em: jun. 2005.

SATO, Y.; CHIBA, S.; TATSUBORI, M. A Selective, Just-In-Time Aspect Weaver. In: INTERNATIONAL CONFERENCE ON GENERATIVE PROGRAMMING AND COMPONENT ENGINEERING, GPCE, 2., 2003, Erfurt, Germany. **Proceedings…** Berlin: Springer, 2003. p. 189-208. (Lecture Notes in Computer Science, v. 2830).

SHENDE, S.; MALONY, A. D. Integration and Application of the TAU Performance System in Parallel Java Environments. In: JOINT ACM-ISCOPR CONFERENCE ON JAVA GRANDE, 2001, Palo Alto, California, United States. **Proceedings…** [S.l.: s.n.], 2001. p. 87–96.

SHENDE S. et al. Portable Profiling and Tracing for Parallel, Scientific Applications using C++. In: SIGMETRICS SYMPOSIUM ON PARALLEL AND DISTRIBUTED TOOLS, 1998, Welches, Oregon, United States. **Proceedings…** [S.l.: s.n.], 1998. p. 134-145.

SHIRAZI, J. **Java Performance Tunning**. Beijing: O'Reilly & Associates, 2000.

SILVA, G. J.; SCHNORR, L. M.; STEIN, B. de O. JRastro: A Trace Agent for Debugging Multithreaded and Distributed Java Programs. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, SBAC-PAD, 15., 2003, São Paulo, Brazil. **Proceedings…** [S.l.: s.n.], 2003.

SOLLICH, P. **CLRProfiler**. Disponível em: <http://www.microsoft.com/downloads/details.aspx?FamilyId=86CE6052-D7F4-4AEB-9B7A-94635BEEBDDA&displaylang=en>. Acesso em: out. 2003.

STROM R. E. et al. **HERMES:** A Language for Distributed Computing. Englewood Cliffs: Prentice Hall, 1991.

SUN MICROSYSTEMS. **JavaTM Virtual Machine Debug Interface Reference**. Disponível em: <http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jvmdi-spec.html>. Acesso em: dez. 2004 (a).

SUN MICROSYSTEMS. **Java Debug Wire Protocol**. Disponível em: <http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdwp-spec.html>. Acesso em: out. 2004 (b).

SUN MICROSYSTEMS. **Java Debug Interface**. Disponível em: <http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdi/>. Acesso em: out. 2004 (c).

SUN MICROSYSTEMS. **JVM Tool Interface**. Disponível em: <http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/>. Acesso em: out. 2004 (d).

SUN MICROSYSTEMS. **Monitoring and Management for the Java Platform**. Disponível em: <http://java.sun.com/j2se/1.5.0/docs/guide/management/>. Acesso em: nov. 2004 (e).

SUN MICROSYSTEMS. **Code sample:** jvmstat 3.0. Disponível em: <http://java.sun.com/performance/jvmstat/>. Acesso em: set. 2005 (a).

SUN MICROSYSTEMS. **Java Platform Debugger Architecture (JPDA)**. Disponível em: <http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jpda.html>. Acesso em: fev. 2005 (b).

SUN MICROSYSTEMS. **Java Research License**. Disponível em: <http://java.net/jrl.html>. Acesso em: jun 2005 (c).

SUN MICROSYSTEMS. **Java 2 Platform, Enterprise Edition (J2EE)**. Disponível em: <http://java.sun.com/j2ee/>. Acesso em: jun. 2005 (d).

SUN MICROSYSTEMS. **Sun Microsystems**. Disponível em: <http://www.sun.com/>. Acesso em: jun. 2005 (e).

SUN MICROSYSTEMS. **Welcome to NetBeans**. Disponível em: <http://www.netbeans.org/>. Acesso em: jun. 2005 (f).

SUN MICROSYSTEMS. **JavaServer Pages Technology - Implementations & Specifications**. Disponível em: <http://java.sun.com/products/jsp/reference/api/>. Acesso em: ago. 2005 (g).

SUN MICROSYSTEMS. **Core Java Technology Features in Mustang**. Disponível em: <http://java.sun.com/developer/technicalArticles/J2SE/Desktop/Mustang_build39.html>. Acesso em: fev. 2006.

TEARE, D. **Quick Start Guide to Enterprise AOP with Aspectwerkz 2.0**. Disponível em: <http://dev2dev.bea.com/pub/a/2005/04/enterprise_aop.html>. Acesso em: jul. 2005.

TRINDADE, J. M. F.; JACQUES-SILVA, G.; DREBES, R. J.; WEBER, T. S.; JANSCH-PORTO, I. Off-line Synchronization of Distributed Logs in Fault Injection

Test Campaigns. In: IEEE LATIN-AMERICAN TEST WORKSHOP, 7., 2006, Buenos Aires. **Proceedings…** Porto Alegre: Evangraf, 2006. p. 137-142.

VISWANATHAN, D.; LIANG, S. Java Virtual Machine Profiler Interface. **IBM Systems Journal**, New York, v. 39, n. 1, 2000.

WILSON, S.; KESSELMANN, J. **Java Platform Performance:** Strategies and Tatics. Boston: Addison-Wesley, 2000.