UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Análise de desempenho de sistemas distribuídos de grande porte na plataforma Java

Gleydson de Azevedo Ferreira Lima

Número de ordem PPgEE: M186
Natal, RN, fevereiro de 2007

Divisão de Serviços Técnicos

Catalogação da publicação na fonte. UFRN / Biblioteca Central Zila Mamede

Lima, Gleydson de Azevedo Ferreira.

Análise de desempenho de sistemas distribuídos de grande porte na plataforma Java / Gleydson de Azevedo Ferreira Lima - Natal, RN, 2007

81 f.: il

Orientador: João Batista Bezerra

Dissertação(Mestrado) - Universidade Federal do Rio Grande do Norte. Centro de Tecnologia. Programa de Pós-Graduação em Engenharia Elétrica.

1. Sistemas distribuídos - Dissertação. 2. Fator de desempenho - Sistemas - Dissertação. 3. Máquina virtual - Dissertação. 4. Coleta de lixo - Dissertação. 5. Hibernate - Dissertação. 6. Struts - Dissertação. I. Bezerra, João Batista. II. Título.

RN/UF/BCZM CDU 004.75(043.2)

Análise de desempenho de sistemas distribuídos de grande porte na plataforma Java

Gleydson de Azevedo Ferreira Lima

Dissertação de Mestrado aprovada em 02 de fevereiro de 2007 pela banca examinadora composta pelos seguintes membros:

Prof. Dr. João Batista Bezerra (orientador) ..... DCA/UFRN

Prof. Dr. André Maurício Campos ..... DIMAp/UFRN

Prof. Dr. Jorge Henrique Cabral Fernandes ..... CIC/UnB

Resumo

A plataforma Java vem sendo crescentemente adotada no desenvolvimento de sistemas distribuídos de alta demanda de usuários. Este tipo de aplicação é mais complexa pois necessita além de atender os requisitos funcionais cumprir os parâmetros de desempenho pré-estabelecidos. Este trabalho realiza um estudo da máquina virtual Java (JVM) abordando seus aspectos internos e explorando as políticas de coleta de lixo existentes na literatura e as usadas pela JVM. Apresenta também um conjunto de ferramentas que auxiliam na tarefa de otimizar aplicações e outras que auxiliam no monitoramento das aplicações em produção.

Diante da grande quantidade de tecnologias que se apresentam para solucionar problemas inerentes às camadas das aplicações, torna-se difícil realizar escolha daquela que possui o melhor tempo de resposta e o menor uso de memória. O trabalho apresenta um breve referencial teórico de cada uma das possíveis tecnologias e realiza testes comparativos através de uma análise estatística da variável aleatória do tempo de resposta e das atividades de coleta de lixo.

Os resultados obtidos fornecem um subsídio para engenheiros e gerentes decidirem quais tecnologias utilizarem em aplicações de grande porte através do conhecimento de como elas se comportam nestes ambientes e a quantidade de recursos que consomem. A relação entre produtividade da tecnologia e seu desempenho também é considerada como um fator importante nesta escolha.

Palavras-chave: Desempenho, Máquina Virtual, Coleta de Lixo, WEB, Negócio, Persistência, Struts, JSF, Hibernate, Comparação de Desempenho

Abstract

The Java Platform is increasingly being adopted in the development of distributed systems with high user demand. This kind of application is more complex because it needs beyond attending the functional requirements, to fulfill the pre-established performance parameters. This work makes a study on the Java Virtual Machine (JVM), approaching its internal aspects and exploring the garbage collection strategies existing in the literature and used by the JVM. It also presents a set of tools that helps in the job of optimizing applications and others that help in the monitoring of applications in the production environment.

Due to the great amount of technologies that aim to solve problems which are common to the application layer, it becomes difficult to choose the one with best time response and less memory usage. This work presents a brief introduction to each one of the possible technologies and realize comparative tests through a statistical analysis of the response time and garbage collection activity random variables.

The obtained results supply engineers and managers with a subside to decide which technologies to use in large applications through the knowledge of how they behave in their environments and the amount of resources that they consume. The relation between the productivity of the technology and its performance is also considered an important factor in this choice.

Keywords: Performance, Virtual Machine, Garbage Collection, WEB, Business, Persistence, Struts, JSF, Hibernate, Performance Comparison.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Sistemas computacionais distribuídos com altas demandas de acessos devem ter o desempenho como um fator norteador de todo o seu desenvolvimento. É necessário um controle total sob este aspecto. Todavia, este controle não é trivial, uma vez que são diversos os fatores que podem degradar o desempenho, como, por exemplo: a escolha errada de uma tecnologia de software ou a configuração não adequada do ambiente de produção às características do problema.

Os ambientes de execução dos sistemas, mais especificamente aqueles que executam em máquinas virtuais, devem estar satisfatoriamente ajustados, sob pena de não aproveitar toda a capacidade do hardware. Portanto, é necessário entender o funcionamento dessas máquinas para possibilitar o ajuste de parâmetros. Nessa otimização é de suma importância o domínio sobre ferramentas auxiliares para obter informações do ambiente de execução.

Além da adequação do ambiente, a otimização da aplicação é fator indispensável para um menor tempo de resposta e maior *throughput* no atendimento de requisições. É necessário ter domínio sobre o detalhamento do desempenho do código e sobre as escolhas de tecnologias que devem ser efetuadas durante o processo de desenvolvimento.

O fator desempenho é, em várias situações, crítico para a produção de um sistema. Nessas situações, uma corporação estabelece um contrato para estabelecer regras para obtenção desses requisitos não funcionais, os SLA (*Service Level Agreement*) [Tude 2003]. Nestes acordos, variáveis como tempos máximo e médio de resposta, quantidade de usuários, dentre outros, são estabelecidos e devem ser cumpridos. O grau de obtenção dos acordos SLA definem a qualidade operacional do sistema (*QoS - Quality Of Service*)[Tude 2003].

Os processos de engenharia de software tradicionais são focados na construção de casos de uso, considerando-se o menor custo, prazo mínimo de produção visando uma maior qualidade. Usualmente se baseiam em uma seqüência de passos que devem ser seguidos, visando à obtenção de um produto de boa qualidade funcional. Frequentemente, os resultados obtidos não levam em conta os requisitos não funcionais, resultando, em algumas situações, em um sistema com o desempenho insatisfatório.

O impacto no desempenho pode decorrer de projetos e códigos não-otimizados em relação a determinadas funcionalidades, com o agravante de que em sistemas com milhares de linhas de código, a tarefa de detectar o gargalo pode ser custosa e demorada. Assim,

a produção de código rápido é um dos fatores para o sucesso da escalabilidade de um sistema, além de outros, tais como a disponibilidade de banda de transmissão na rede e ambiente de execução otimizado.

Uma banda insuficiente ou uma arquitetura de rede deficiente constituem fatores críticos, mas cuja resolução extrapola o espaço de contexto do nó onde a aplicação é executada. Como são fatores externos, em geral podem ser resolvidos sem mudanças na aplicação. Já os ajustes no ambiente de execução exigem o conhecimento das características da aplicação e do ambiente para obter os melhores parametros de configuração.

Ressaltando-se que sistemas distribuídos com milhares de acessos simultâneos são executados, normalmente, em hardwares de grande porte com alta capacidade de processamento, as linguagens de programação modernas, focadas nestes contextos, são desenvolvidas para que o código não seja executado diretamente no hardware, e sim, por meio de instruções independentes de plataforma, através de máquinas virtuais. A utilização dessas máquinas cria um modelo de arquitetura de três níveis, como ilustrado na Figura 1.1. Nesta arquitetura, tanto a aplicação quanto a máquina virtual necessitam ser ajustadas para usar melhor os recursos disponíveis no hardware, criando assim duas áreas críticas para a análise de desempenho: a aplicação e o ambiente de execução.

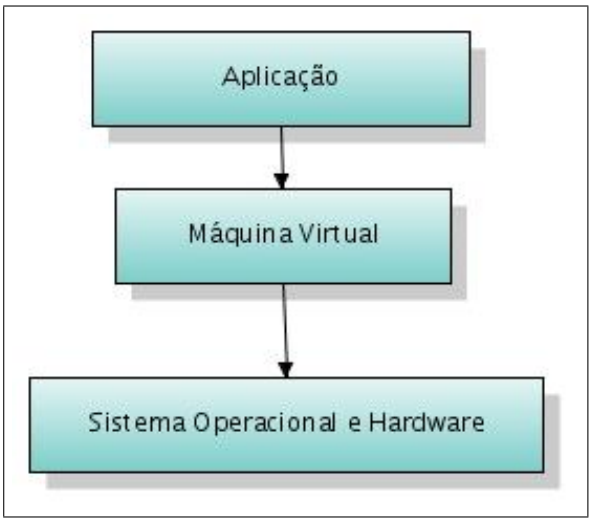

Figura 1.1: Níveis da Infra-Estrutura de Execução de um sistema em máquina virtual

A aplicação pode ser otimizada através de ferramentas que apóiam a análise do desempenho de código durante o processo de desenvolvimento e a escolha correta das tecnologias considerando o desempenho como um peso forte na escolha. Já o problema do ambiente pode ser resolvido através do ajuste das técnicas de escalonamento, sincronização, coleta de lixo, tamanhos de áreas de memória utilizadas pela máquina virtual, dentre outros.

Feitas essas considerações de ordem geral no desenvolvimento de sistemas de grande porte, este trabalho realiza um estudo dos aspectos que influenciam no desempenho de aplicações, apresentando a máquina virtual Java, seu funcionamento interno e as políticas de coletas de lixo existentes na literatura; ele aborda também um conjunto de ferramentas para auxiliar na tarefa de obtenção de informações do ambiente de execução e do detalhamento de desempenho do código. Como foco principal, compara um conjunto de tecnologias realizando testes de desempenho através de aplicações desenvolvidas para este fim, obtendo os tempos de respostas e uso de memória, possibilitando escolher aquela mais adequada para as situações de alta demanda de usuários confrontando com a relação de produtividade no desenvolvimento. As seções abaixo detalham os aspectos tratados neste trabalho.

1.1 Impacto do Ambiente de Execução

A execução de sistemas em máquina virtual possibilita a utilização de coletores de lixo. Ou seja, um módulo da máquina virtual que busca por objetos inúteis e remove-os. Porém, apesar de ser algo extremamente apreciado na programação do sistema, uma vez que ausenta do programador a tarefa de desalocar objetos, impõe problemas à escalabilidade e desempenho da aplicação.

Muitas pesquisas têm sido realizadas na busca de um coletor de lixo que tenha pouca influência na execução do código conduzindo-as a um conjunto de algoritmos que realizam esta tarefa de distintas maneiras. A escolha correta do algoritmo depende do hardware e das características da aplicação.

1.2 Ferramentas de Apoio

Analisar o funcionamento de um coletor de lixo, obter parametros da quantidade de memória utilizada e o tempo de coleta são tarefas que não podem ser realizadas sem a ajuda da máquina virtual. Neste contexto, é oportuno o uso de ferramentas que obtenham esses dados em baixo nível e os transformem em dados sintéticos que possam ser melhor analisados pelo engenheiro de software.

Diversas ferramentas são apresentadas visando o monitoramento da máquina virtual, bem como o detalhamento do desempenho do código em execução (os *profilers*).

1.3 Impacto do Design da aplicação

Obtendo-se um ambiente de execução otimizado tem-se a segurança que a aplicação possui um cenário próximo do ideal para sua execução. Porém, de pouco adianta esse aspecto favorável se a aplicação não possui desempenho compatível com os requisitos não funcionais estabelecidos em SLAs.

A otimização das aplicações é custosa e demorada se não houver métodos de obter

métricas e parametros de seu desempenho. Para tal tarefa, existem os *profilers*, ferramentas que possibilitam detectar problemas de desempenho facilmente e com alta qualidade.

Obter o controle sob o código criado pela equipe de desenvolvimento de um software é essencial, todavia é comum utilizar bibliotecas e *frameworks* de terceiros para resolver problemas inerentes das camadas da aplicação. Assim, a escolha errada de uma dessas soluções põe no sistema um componente degradante de seu desempenho e pode levar a um resultado insatisfatório.

Não são facilmente encontrados artigos que analisem essas soluções obtendo comparativos detalhados de desempenho que possibilitem ao arquiteto e o gerente do software escolherem a tecnologia mais adequada para o seu problema. Frequentemente, essa escolha é realizada através de características como produtividade, qualidade de implementação, reusabilidade, dentre outras.

Uma análise abrangente dos vários aspectos do desempenho exige o uso de uma plataforma de alta qualidade que utilize máquina virtual e possua também todo o acervo de ferramentas necessário. É necessáro também uma plataforma que possa executar em ambientes multiprocessados e com hardwares de alta capacidade, uma vez que são esses ambientes que são encontrados em grandes sistemas. Assim, toda a análise dos fatores supramencionados serão desenvolvidos através da plataforma Java [Tim Lindholm 1999].

1.4 Organização do texto

Esta dissertação está dividida da seguinte forma: o capítulo 2 aborda os conceitos da máquina virtual, o capítulo 3 apresenta ferramentas para otimização e monitoramento de aplicações, o capítulo 4 aborda os aspectos lógicos que influenciam na aplicação, apresentando as camadas e soluções que serão testadas. O capítulo 5 descreve como os testes são realizados e apresenta os resultados obtidos. O capítulo 6 conclui sobre o uso das tecnologias em ambientes de alto desempenho e sugere trabalhos futuros.

Capítulo 2

A máquina virtual e a coleta de Lixo

Em ambientes de execução de sistemas de alto desempenho em geral máquinas virtuais são empregadas. Assim, entender como essas máquinas funcionam, principalmente as técnicas de coleta de lixo é fator preponderante para possibilitar a otimização de desempenho. Considerando os objetivos desse trabalho, neste capítulo serão abordados aspectos do funcionamento interno da máquina virtual Java e suas políticas de coleta de lixo.

2.1 A máquina Virtual Java (JVM)

Desde que surgiu a linguagem Java, em 1995, a sua característica mais marcante é a independência de plataforma. Um código Java não é compilado diretamente para as instruções de uma determinada arquitetura de hardware, e sim para um formato intermediário capaz de ser interpretado por uma máquina virtual. Esta característica viabiliza a independência da aplicação em relação à plataforma.

Outras tecnologias permitem a portabilidade de código, porém, com recompilação. É o caso do padrão ANSI C [Holmes 1995], no qual é possível recompilar o mesmo código em C para diversas plataformas de hardware como mostra a Figura 2.1.

Um código compilado para uma JVM (*Java Virtual Machine*) não necessita passar por recompilações para ser executado em outra plataforma. A compilação é feita apenas uma vez e o código então pode ser executado em todas as plataformas. Essa compilação é feita em um formato intermediário denominado *bytecodes* [Tim Lindholm 1999]. Esses *bytecodes* representam um código binário pré-compilado preparado para interpretação. O binário é então interpretado para cada plataforma alvo através da JVM. A figura 2.2 ilustra esse funcionamento.

O preço pago pela flexibilização da portabilidade é que o código interpretado é, em média, 20 vezes mais lento que um código nativo compilado [Venners 1998], o que a princípio pode ser um problema. Entretanto, sabe-se que um percentual pequeno (10-20%) do código de uma aplicação representa uma parte significativa do seu desempenho [Venners 1998]. Baseado nesta constatação, pesquisas foram realizadas visando aproximar o desempenho da interpretação à execução nativa, que resultaram em avanços significativos na otimização de código em tempo de execução. Como não é viável transformar todo o código interpretado em um nativo a cada execução[Venners 1998], o foco voltou-se para a parte do código mais importante para o desempenho, denominado *hotspot*.

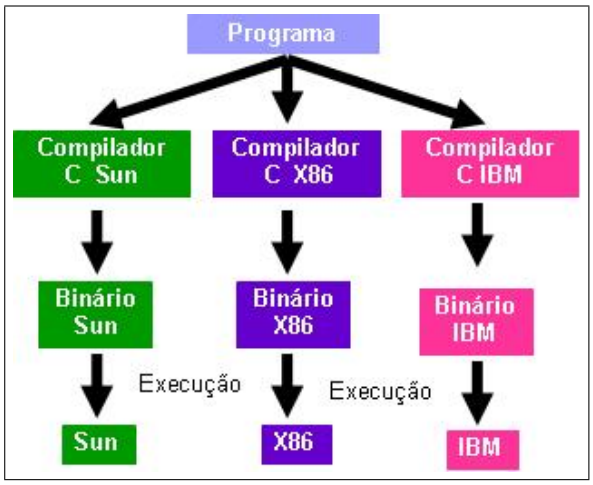

Figura 2.1: Compilação para várias plataforma em ANSI C.

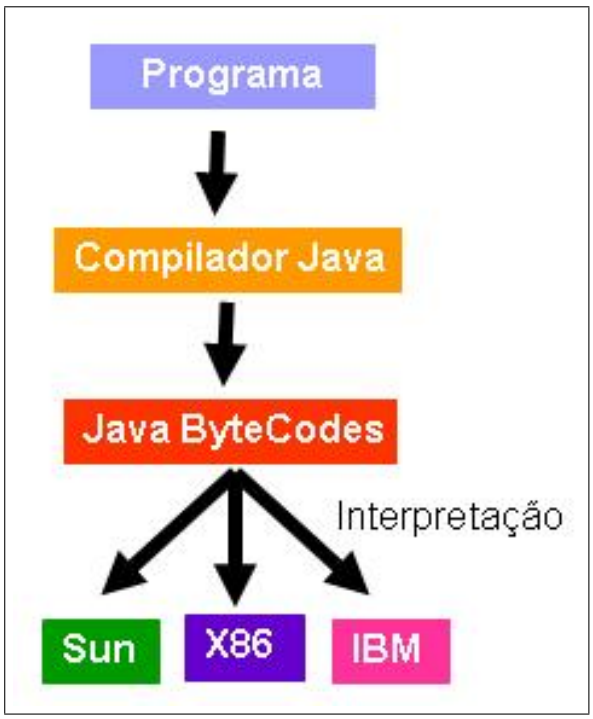

Figura 2.2: Compilação e execução na plataforma Java.

As máquinas virtuais modernas possuem compiladores em tempo de execução (*JIT - Just In-Time Compiler*) [Venners 1998] que geram códigos nativos para os *hotspots*, tornando assim, um código interpretado com desempenho semelhante ao compilado. O processo funciona da seguinte forma: ao iniciar a execução de um código o compilador JIT procura pelos *hotspots*; identificados estes fragmentos transforma-os em código nativo. Como os *hotspots* são responsáveis diretos pela maior parte do desempenho, a aplicação como um todo executa mais rápido. Além dessas características, as máquinas virtuais *hotspots* possuem outros recursos, como: [Sun Microsystems 2004*b*].

1. *Alocação rápida de memória e coleta de lixo:* A tecnologia Java HotSpot fornece uma rápida alocação de memória para objetos, e oferece uma coleta de lixo rápida e eficiente;
2. *Sincronização de Threads:* A linguagem Java permite o uso de linhas concorrentes e simultâneas, chamadas *threads*. Em servidores multiprocessados a sincronização das *threads* deve ser otimizada visando o melhor uso dos vários processadores. A tecnologia *hotspot* fornece mecanismos para escalabilidade para servidores multiprocessados com memória compartilhada.

2.2 A máquina virtual Java em Detalhes

Na especificação da máquina virtual Java [Tim Lindholm 1999] o comportamento de uma instância é descrito em termos de subsistemas, áreas de memórias, tipos de dados e instruções. Esses componentes descrevem uma arquitetura interna para uma máquina virtual Java abstrata.

A Figura 2.3 mostra um diagrama de blocos que ilustra a maioria dos subsistemas e áreas de memória descritas na especificação. O subsistema de carregamento de classes é um mecanismo com a função de carregar tipos (classes e interfaces). Cada máquina virtual também possui um motor de execução, responsável pela execução das instruções contidas nos métodos das classes carregadas.

Quando a máquina virtual executa um programa, ela necessita de memória para armazenar os diversos componentes, incluindo *bytecodes* e outras informações extraídas das classes carregadas, objetos que são instânciados pelo programa, parâmetros para os métodos, valores de retornos, variáveis locais e resultados imediatos de computações. A JVM organiza a memória em várias áreas de dados de execução.

Algumas áreas de dados são compartilhadas entre as diversas *threads* da aplicação, enquanto outras são únicas para cada *thread*. Cada instância da máquina virtual Java possui uma área de métodos e um *Heap*. Estas áreas são compartilhadas por todas as *threads* em execução. Quando a máquina virtual carrega um arquivo de uma classe, faz um *parsing* da classe e armazena esses dados na área de métodos. Enquanto o programa é executado, a máquina virtual põe todos os objetos instanciados no *heap*. A figura 2.4 [Venners 2000] ilustra essa divisão.

Quando uma *thread* é criada, ela recebe seu próprio registrador PC (*program counter*) e uma pilha de dados. Se a *thread* está executando um método Java, o valor do registrador PC indica a próxima instrução a ser executada. A pilha da *thread* Java armazena o estado

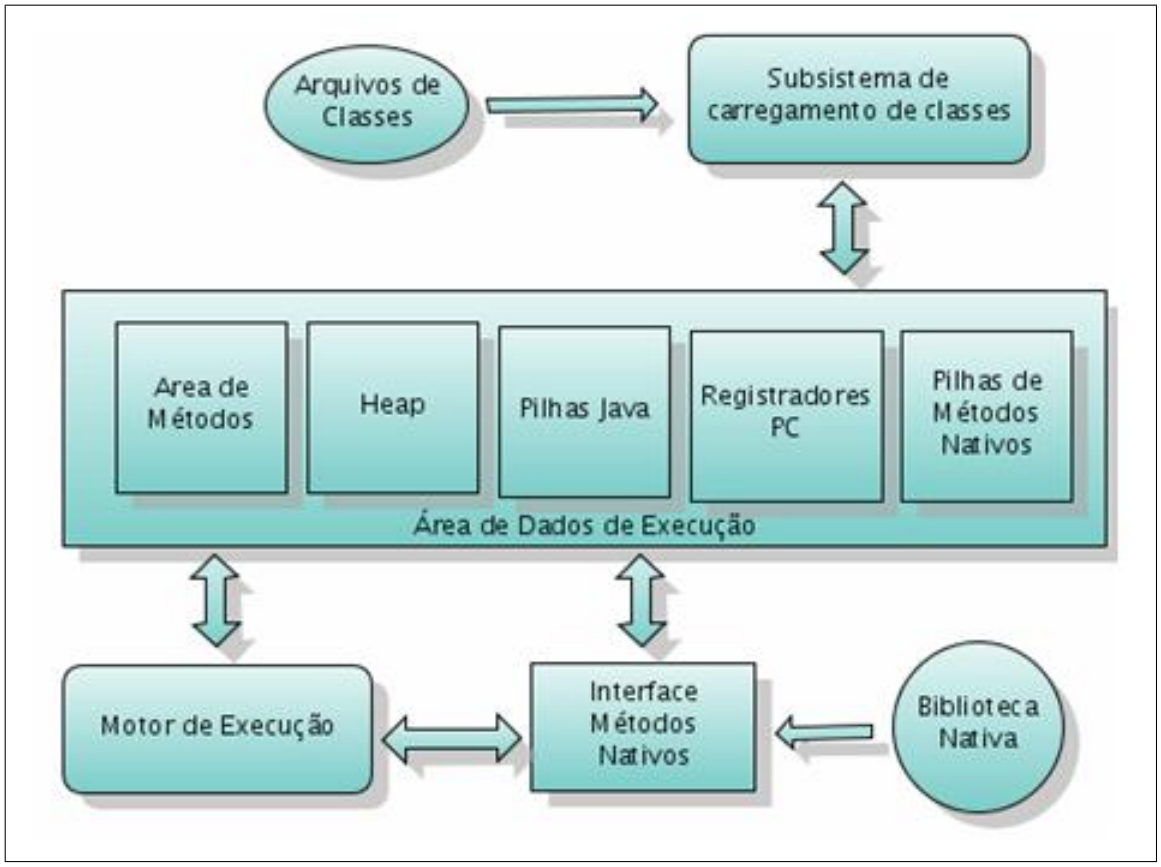

Figura 2.3: Subsistemas da máquina Virtual Java (JVM).

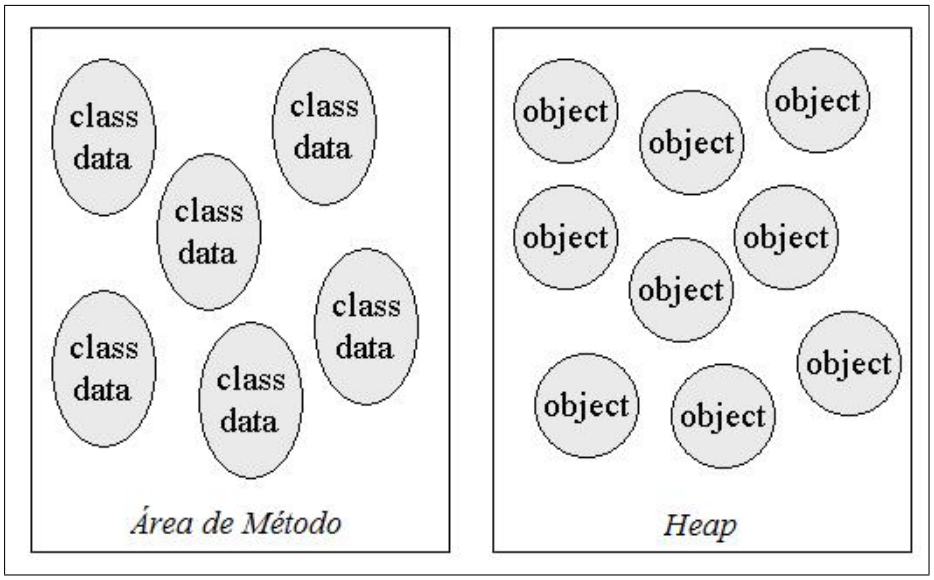

Figura 2.4: Áreas de dados e de código na JVM.

da *thread* durante a execução do método. Este estado inclui suas variáveis locais, os parâmetros, o valor de retorno e cálculos intermediários.

A pilha é composta de quadros. Um quadro de pilha contém o estado de uma invocação de um método. Quando uma *thread* invoca um método, a máquina virtual põe no topo da pilha um novo quadro para esta invocação. Quando o método é finalizado, a máquina virtual retira o quadro do topo da pilha.

A máquina virtual Java não tem registradores para armazenar dados intermediários. O conjunto de instruções usa a pilha para armazenar os valores intermediários. Essa decisão foi tomada para assegurar um conjunto compacto de instruções e permitir que a ela possa ser implementada em arquiteturas com poucos registradores [Venners 2000].

A Figura 2.5 [Venners 2000] é uma descrição gráfica das áreas de memória demandadas pela execução de cada *thread*. Estas áreas são privadas entre as várias *threads*. Nenhuma *thread* pode acessar o registrador PC ou a pilha de outra *thread*.

2.2.1 A área de memória de alocação dinâmica (Heap)

Sempre que uma classe é instanciada ou um *array* é criado, este novo objeto é armazenado no *heap*. Como existe apenas um *heap* em uma instância da máquina virtual, todas as *threads* compartilham-no. Como duas aplicações Java executam em instâncias diferentes de máquinas virtuais, não há como o *heap* de uma interferir na da outra. Duas *threads* diferentes de uma mesma aplicação podem compartilhar e alterar dados do mesmo *heap*. Este é o motivo pelo qual é essencial a utilização da sincronização para proteger o acesso seguro *multithreading* [Hide 1999].

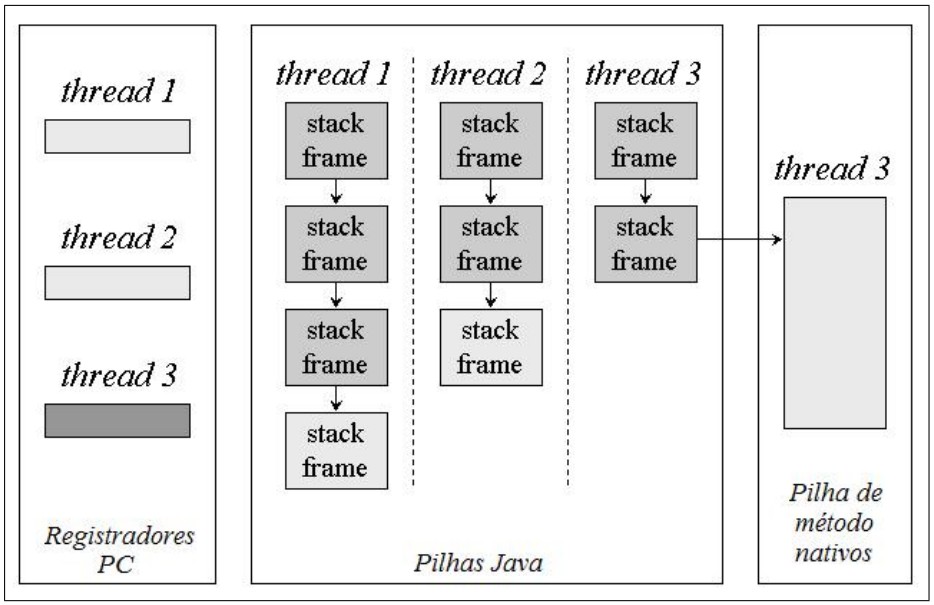

Figura 2.5: Divisão das áreas de memória por cada thread.

A máquina virtual Java tem uma instrução para alocar um novo objeto no *heap*, mas não tem instrução para liberar memória. Não é possível liberar memória explicitamente no código Java. A máquina virtual é responsável, sozinha, pela realização desta tarefa. Através da coleta de lixo (*Garbage Collection*), os objetos não utilizados pela aplicação são removidos do *heap*.

A função principal do coletor de lixo é remover os objetos que não estão sendo mais referenciados pela aplicação. Ele também deve mover os objetos no *heap* para reduzir a fragmentação [Venners 2000].

A especificação da máquina virtual Java não atribui qualquer regra para a coleta de lixo. As referências a objetos podem existir em vários lugares: pilhas, *heap*, área de métodos, pilhas de métodos nativos. A escolha da técnica de coleta de lixo será fortemente influenciada pela área de dados da máquina virtual e pode influenciar decisivamente no desempenho da aplicação [Sun Microsystems 2004*a*].

Assim como a área de métodos, a memória do *heap* não necessita ser contígua e pode ser expandida e contraída durante o progresso na execução da aplicação. A área de código de métodos pode, de fato, ficar localizada no topo do *heap*. Em outras palavras, quando a máquina virtual necessita das classes recentemente carregadas, ela pode pegá-las da mesma área onde os objetos residem. O mesmo coletor de lixo que limpa a memória de objetos sem referência também busca por classes sem referência. A JVM possibilita ao usuário especificar o tamanho inicial e o tamanho máximo do heap, normalmente, através do parâmetro -Xms e -Xmx, especificando o mínimo e máximo, respectivamente.

A escolha da técnica de coleta de lixo pode influenciar decisivamente no desempe-

nho de aplicações de uso intensivo de memória. Adiante serão discutidos os diversos algoritmos de coleta de lixo.

2.2.2 Representação dos Objetos

A especificação da JVM não define a forma na qual os objetos devem ser representados no *heap*. A representação dos objetos é uma decisão do projetista da máquina virtual. [Venners 2000]

O dado primário que deve ser representado por cada objeto diz respeito às variáveis de instância declaradas na classe do objeto e todas as suas super-classes. Dada uma referência de objeto, a máquina virtual deve ser capaz de, rapidamente, localizar os dados de instância para esta referência. É necessário também uma maneira de acessar o código desse objeto localizado na área de método. Por essa razão, a memória alocada para um objeto normalmente inclui um tipo de ponteiro para a área de método.

Um possível projeto para o *heap* seria dividí-lo em duas partes: um *pool* de manipuladores e outro de objetos. O *pool* de manipuladores possui dois componentes: um ponteiro para a instância dos dados e outro para a área de métodos. A vantagem deste esquema é que se torna fácil para a máquina virtual combater a fragmentação do *heap* [Mark S. Johnstone 1998], a ser explicado mais adiante.

2.3 Coleta de Lixo

A máquina virtual Java armazena no *heap* todos os objetos criados pela aplicação. Objetos são criados pelas instruções *newarray, anewarray e multianewarray* [Tim Lindholm 1999], mas nunca são liberados explicitamente pelo código. Como já mencionado, é a coleta de lixo que automaticamente remove objetos que não são mais referenciados pelo programa.

Esta seção não descreve a coleta de lixo oficial da máquina virtual Java, pois sua especificação não dita qualquer técnica de coleta de lixo. Na realidade nem a coleta de lixo é requerida pela especificação, mas, como não existe uma memória infinita, todas atuais implementações possuem coleta de lixo de *heaps*. O coletor de lixo é um processo que é executado em segundo plano e realiza algumas paradas na aplicação para realizar a limpeza. Por essa razão, a escolha do algoritmo é fator preponderante no desempenho do sistema. A seguir serão descritas técnicas de coleta de lixo que possibilitam a escolha do engenheiro de software na conformidade da aplicação a ser desenvolvida.

2.3.1 Algoritmos de Coleta de Lixo

Qualquer algoritmo de coleta de lixo tem duas caracteristicas básicas. Primeiro, deve detectar objetos passíveis de limpeza. Segundo, deve reaver o espaço de memória usado por objetos coletados, tornando-o disponível novamente para o programa.

A detecção de objetos passíveis de coleta é realizada pela definição de um conjunto de raízes de referências, determinando as alcançabilidades a partir dessas raízes. Um objeto é alcançável se há algum caminho de referência a partir das raízes pelo qual o programa

em execução pode acessar o objeto [Venners 2000]. É importante frisar que as raízes são sempre acessíveis para o programa. Qualquer objeto que seja alcançável pelas raizes é considerado "vivo". Objetos que não são alcançáveis a partir das raizes são consideráveis coletáveis, pois não afetam a execução do programa.

Coletores por contagem de referência

Contagem de referência foi a primeira estratégia de coleta de lixo. Nesta abordagem, um contador de referência é mantido para cada objeto no *heap*. Quando um objeto é criado e uma referência lhe é atribuída para uma variável, ao contador de referência é atribuído o valor 1. Se outra variável faz referência ao mesmo objeto, o contador de referência do objeto é incrementado. Quando a variável que referencia o objeto é destruída ou atribuida para um novo objeto, o contador de objetos é decrementado. Nessa dinâmica, quando o contador de referência do objeto atinge o valor zero é porque pode ser coletado. Quando um objeto é coletado, qualquer outro objeto que lhe faz referência tem seus contadores de referência também decrementados. Dessa maneira a coleta de um objeto pode levar em conseqüência a coleta de outros objetos [Venners 2000].

A vantagem desta abordagem é que a coleta de lixo por contagem de referência pode ser executada em pequenos intervalos de tempo intercalados com a execução do programa. Esta característica o torna adequado para ambientes de tempo real, onde o programa não pode ser interrompido por um período longo. A desvantagem é que esta abordagem não detecta ciclos, fenômeno decorrente da referência mútua entre dois objetos. Um exemplo de ciclo é um objeto pai que tem a referência para um objeto filho que o referencia de volta. Estes objetos nunca terão o contador de referência no estado zero, mesmo que eles se tornem inalcançáveis através das raízes de execução do programa. Outra desvantagem é que o contador de referência é o *overhead* de incrementar e decrementar referências a cada momento. Devido a essas desvantagens, esta técnica está em desuso. Atualmente, é difícil encontrar uma máquina virtual no mercado que adote tal algoritmo.

Coletores de Tracing

Esta estratégia de coleta armazena um grafo de referências dos objetos iniciando com um nó raiz. Objetos que são encontrados durante a varredura são marcados segundo um critério pré-estabelecido. Esta marcação é normalmente feita dentro do próprio objeto ou marcando flags em um mapa de bits separado. Depois que o *tracing* está completo, os objetos não marcados são denominados inalcançáveis e podem ser coletados [Venners 2000].

O algoritmo básico é chamado de "marca e troca". Este nome é devido às duas fases do processo de coleta de lixo. Na fase de marcação, o algoritmo percorre a árvore de referências e marca cada objeto que encontrar. Na fase de troca, objetos não marcados são liberados, e a memória resultante torna-se disponível para o programa em execução. Na máquina virtual Java, a fase de troca deve incluir a finalização de objetos, ou seja, uma chamada do método *finalize* para informar ao objeto que ele será coletado. Coletores de *tracing* representam uma categoria de coletor, os coletores de compactação e cópia são considerados coletores de *tracing*.

Coletores de Compactação

Os coletores de lixo na máquina virtual devem ter uma estratégia para combater a fragmentação do *heap*. Duas estratégias freqüentemente usadas por coletores de marca e troca são a compactação e a cópia. Ambas as abordagens movem objetos em tempo de execução para combater a fragmentação do *heap*.Os coletores de compactação movem os objetos ativos para uma extremidade do *heap*. Neste processo a outra extremidade do *heap* torna-se uma ampla área livre e contígua. Todas as referências dos objetos são alteradas de modo a referenciar a nova localização [Venners 2000].

A atualização de referências dos objetos movidos é algumas vezes feita simplesmente pela adição de um nível de indireção para a referência do objeto, ou seja, ao invés de referenciar diretamente os objetos no *heap*, a referência aponta para uma tabela de manipuladores de objetos, que por sua vez aponta para a posição atual do *heap*. Quando um objeto é movido, apenas o manipulador do objeto deve ser alterado de modo a apontar para a nova localização. Todas as referências para o objeto no programa em execução se referem ao manipulador atualizado, que não foi movido. Enquanto esta abordagem tem a vantagem de simplificar o trabalho da desfragmentação do *heap*, tem a desvantagem de adicionar um *overhead* no desempenho para todo acesso aos objetos.

Coletores de Cópia

A coleta de lixo por cópia se baseia na movimentação de todos os objetos ativos para uma nova área de memória. Enquanto os objetos são movidos para a nova área, eles são postos lado a lado, eliminando qualquer espaço livre que poderia estar entre eles na área anterior. Esta é então desalocada, tornando-se livre. A vantagem desta abordagem é que os objetos podem ser copiados à medida que eles são descobertos através do percurso dos nós raízes. Não existem fases separadas de marcação e troca. Objetos são copiados em tempo de execução para a nova área, e os ponteiros são mantidos em suas posições anteriores. Estes ponteiros permitem ao algoritmo detectar referências para objetos que foram movidos. O coletor de lixo então atribui um novo valor para o ponteiro para referenciar a nova área de memória onde o objeto foi copiado [Venners 2000].

Um algoritmo comum de coleta por cópia é o *"parar e copiar (stop and copy)"*. Neste esquema o *heap* é dividido em duas regiões. Apenas uma das duas regiões é usada ao longo do tempo. Objetos são alocados em uma das regiões até que todo o espaço da região tenha se esgotado. Neste ponto a execução do programa é interrompida e o *heap* é atravessado. Objetos ativos são copiados para a outra região à medida que são encontrados a partir da árvore de referências da raiz. Quando o procedimento finaliza, o programa retorna sua execução. A memória será então alocada na nova região do *heap* até que esta seja preenchida. Neste ponto, a execução do programa é novamente interrompida, o *heap* é percorrido e uma cópia é feita para a região original. A desvantagem desta abordagem é que do total de memória utilizado, apenas a metade está disponível para a real utilização [Venners 2000].

A Figura 2.6 ilustra a dinâmica do processo de um *heap* coletado através do algoritmo *stop and copy* . Estão ilustrados nove estados do *heap* no decorrer do tempo. No primeiro estado, a primeira metade do *heap* é o espaço não usado e a metade superior é preenchida

parcialmente de objetos. A porção do *heap* que contém objetos está representada com linhas diagonais. O segundo estado mostra a metade superior do *heap* sendo gradualmente preenchida com objetos, até ela torna-se completa como exibida no terceiro estado. Neste ponto, o coletor de lixo suspende a execução do programa e percorre o grafo de objetos começando dos nós raízes. Cada objeto ativo é copiado para a metade inferior do *heap*, obedecida a seqüência existente na metade anterior. Esta situação está ilustrada no estado 4 da figura 2.6 [Venners 2000]. O quinto estado mostra a fragmentação do *heap* depois que a coleta do lixo é finalizada. Agora a metade superior está sem uso, e a metade inferior é parcialmente preenchida com objetos, até ela tornar-se cheia, o que ocorre no estado 7. Novamente, o coletor de lixo suspende a execução do programa e percorre o grafo de objetos ativos. Neste ponto, ele copia cada objeto que encontrar para a parte superior do *heap*, como ilustrado no estado 8. O estado 9 ilustra o resultado da coleta de lixo: a metade inferior é mais uma vez um espaço inutilizado e a parte superior parcialmente preenchida com objetos. Esse processo se repete indefinidamente até a execução da aplicação terminar.

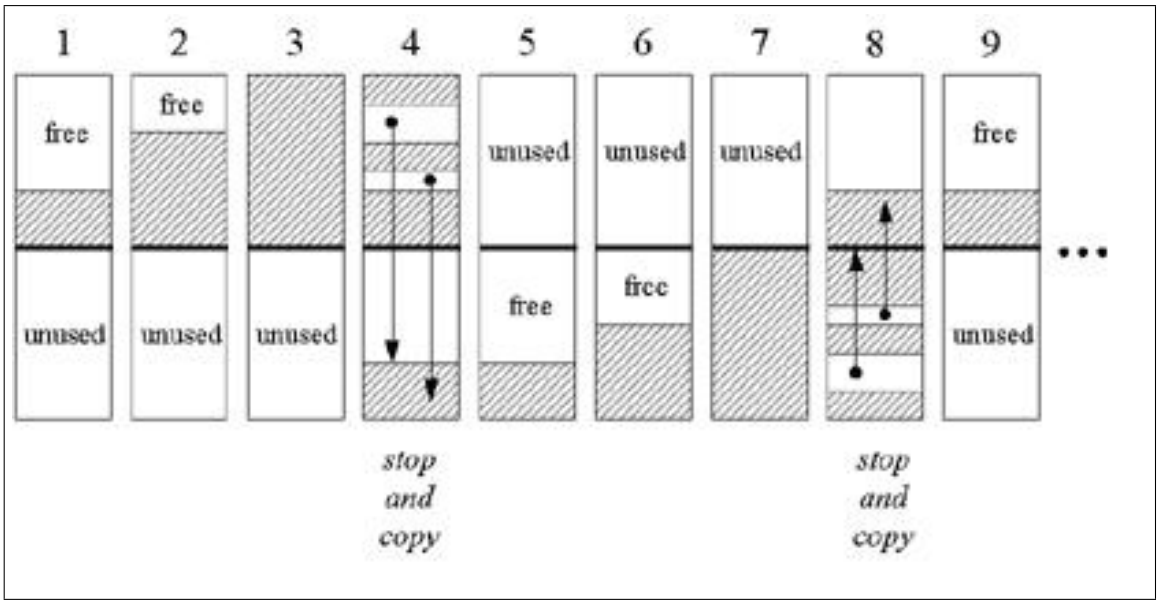

Figura 2.6: Funcionamento do Algoritmo Stop-And-Copy

Coletores de Gerações

Uma desvantagem da estatégia de coleta *"stop and copy"* é que todos os objetos devem ser copiados a cada coleta. Esta característica do algoritmo de cópia pode ser melhorada levando-se em consideração dois fatos que foram, empiricamente, observados na maioria dos programas nas mais variadas linguagens [Venners 2000]:

- A maioria dos objetos criados pelo programa tem vida curta;
- A maioria dos programas cria alguns objetos que possuem um tempo de vida longo. A maior fonte de ineficiência nos coletores de cópia é que eles gastam muito tempo na cópia os objetos de vida longa, repetidas vezes;

Diante desta constatação, a classe de coletores denominados de *coletores de gerações* resolvem o problema agrupando objetos pela idade e coletando os objetos mais novos mais freqüentemente que os objetos velhos. Nesta abordagem, o *heap* é dividido em dois ou mais *sub-heaps*, cada um contendo uma geração de objetos, de modo que a geração mais jovem é coletada mais frequentemente. Como a maioria dos objetos tem vida curta, apenas um pequeno percentual de objetos sobrevive à primeira coleta. Se um objeto sobreviver a algumas coletas como um membro da geração jovem, o objeto é promovido para a próxima geração, isto é, movido para outro *sub-heap*. Desse modo, cada geração mais velha é coletada com menor freqüência do que a geração mais jovem anterior e os objetos "maduros"(sobreviventes a múltiplas coletas) na sua geração corrente são movidos para uma geração mais velha.

A técnica de coleta de gerações pode ser aplicada ao algoritmo de marca e troca como também aos algoritmos de cópia. Neste caso, dividir o *heap* em gerações de objetos pode aumentar a eficiência do algoritmo base de coleta de lixo.

Coletores Adaptativos

O algoritmo de coleta de lixo adaptativo tem a vantagem do fato que alguns algoritmos de coleta funcionam melhor em certas situações, enquanto outros funcionam melhor em outras. Este melhor funcionamento é ditado pelas características da aplicação. A abordagem do algoritmo adaptativo troca de política de coleta durante a execução do programa, a depender do ambiente de execução.

Com a abordagem adaptativa, projetistas de implementações de máquinas virtuais não necessitam escolher apenas uma técnica de coleta de lixo. Eles podem empregar várias técnicas, cada algoritmo é escolhido de acordo com o trabalho com o qual ele se apresente mais eficiente.

Algoritmo do Trem

Uma das desvantagens em potencial das técnicas de coleta de lixo comparadas com a liberação explícita de objetos é que a coleta de lixo fornece ao programa menos controle sobre o escalonamento de CPU. É, em geral, impossível determinar exatamente quando (ou se) um coletor de lixo será invocado e quanto tempo a sua execução durará. Pelo fato dos coletores de lixo normalmente suspenderem a execução do programa enquanto fazem a busca e coleta dos objetos, pode causar longas pausas durante a execução da aplicação. Algumas pausas de coleta de lixo podem ser suficientemente longas para serem perceptíveis pelo usuário, elas podem também impedir que os programas respondam rápido o suficiente de modo a satisfazer os requisitos não funcionais estabelecidos. Se a estratégia de coleta de lixo utilizada é capaz de gerar pausas o suficiente para ser notada pelo usuário

ou para torná-la inadequada para ambientes em tempo real, o algoritmo é dito *perturbador*. Assim, para minimizar as potenciais desvantagens da coleta de lixo comparadas com a liberação explícita de objetos, um meta de projeto para os algoritmos é minimizar ou, se possível, eliminar a natureza perturbadora.

Uma abordagem para atingir coleta de lixo não perturbadora, ou pelo menos tentar atingir, é usar algoritmos de coleta de lixo incremental. Um coletor de lixo incremental é um que, ao invés de buscar e remover todos os objetos inalcançáveis a cada invocação, apenas tenta buscar e descartar uma porção desses objetos. Pelo fato de apenas uma porção do *heap* ser coletado a cada invocação espera-se menos tempo despendido nesta tarefa. Essa política de coleta de lixo ajuda a máquina virtual Java a se adaptar em ambientes de tempo real, onde se faz necessário que a aplicação seja interrompida o mínimo possível. Este tipo de coletor também é muito útil para o ambiente do usuário, pois permite eliminar ou minimizar as latências da aplicação causadas pelo coletor de lixo.

Um algoritmo incremental comum é o geracional, que durante a maioria de suas chamadas coleta apenas uma parte do *heap*. Como mencionado anteriormente, um coletor de gerações divide o *heap* em duas ou mais gerações, onde cada uma delas tem seu *sub-heap*. Através desta constatação empírica que a maioria dos objetos tem o tempo de vida curto, um coletor de gerações coleta os *sub-heaps* de gerações jovens mais frequentemente que as gerações mais velhas. O algoritmo mais famoso que implementa está política é o algoritmo do Trem.

O algoritmo do Trem [Würthinger 2005], que foi proposto inicialmente por Richard Hudson e Eliot Moss e foi usado pela máquina virtual Hotspot da Sun Microsystems [Tim Lindholm 1999], especifica uma organização para o espaço de objetos maduro e o coletor de gerações. O propósito deste algoritmo é fornecer uma coleta incremental no espaço de objetos maduros com pouco uso de tempo.

2.3.2 Coleta de Lixo e Desempenho

A máquina virtual Java 1.4.1, por exemplo, divide o *heap* em duas regiões por padrão (Região Jovem e Velha) [Goetz 2003], e utiliza os algoritmos citados anteriormente nessas regiões. É possível alterar estes algoritmos através da alteração de parametros de configuração. A figura 2.7 [Goetz 2003] ilustra os algoritmos usados em cada região e aqueles que podem ser usados em máquinas multiprocessadas.

Mesmo não sendo o objetivo do trabalho realizar testes comparativos entre algoritmos de coleta de lixo, é essencial que haja o domínio sobre cada técnica de modo a se fazer a escolha certa em cada situação particular que, se apresente na prática. Técnicas de coleta de lixo aplicadas a sistemas multiprocessados serão abordadas a seguir.

A plataforma Java permite a utilização de quatro tipos de algoritmos de coleta. Para a maioria das aplicações, com pequeno e médio porte, o algoritmo de coleta é não perturbador, o usuário não nota a sua presença que ocorre com freqüência baixa. Esse não é o caso de uma aplicação de grande porte que necessita de escalabilidade para um grande número de *threads*, processadores, *sockets* e um grande espaço de memória.

Uma constatação observada na prática [Sun Microsystems 2004*b*] é que a maioria das aplicações não podem ser perfeitamente paralelizadas. Alguma porção é sempre seqüen-

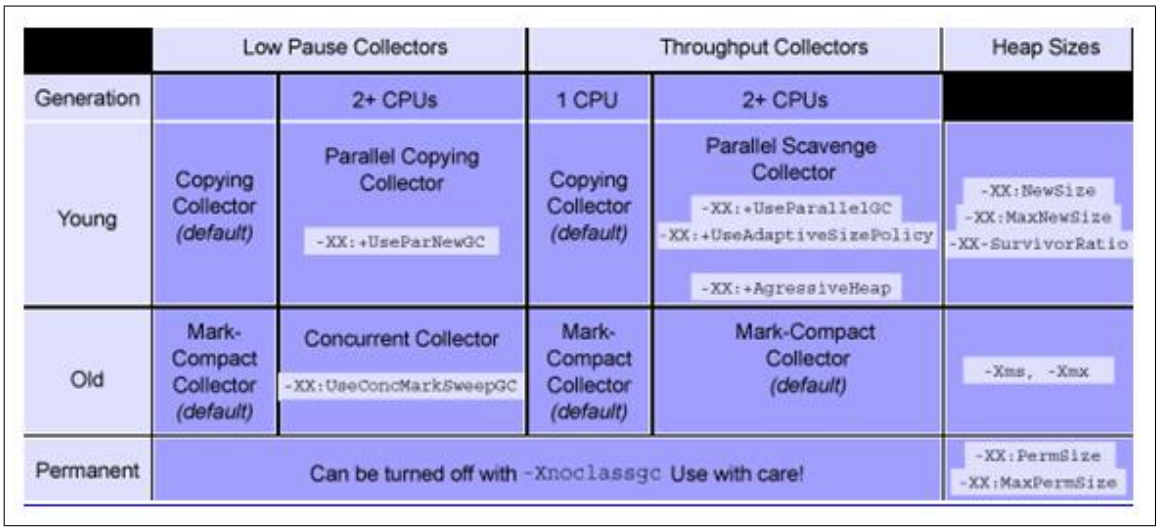

Figura 2.7: Algoritmos de Coleta de Lixo e a JVM 1.4.1.

cial e não tem como obter vantagem do paralelismo. Isto também é verdadeiro para a plataforma Java. Até a versão 1.3.1, não existia um coletor de lixo paralelo, e assim o impacto na coleta para sistemas multiprocessados crescia de acordo com o número de processadores.

O figura 2.8 [Sun Microsystems 2004*b*] mostra um sistema ideal que é perfeitamente escalável com exceção da coleta de lixo. A curva superior representa um custo de apenas 1% do tempo com coleta de lixo em um sistema com apenas um processador. Este impacto se transforma em cerca de 30% em um sistema de 32 processadores.

O coletor padrão (Figura 2.8) resolve para a maioria das situaçõe práticas de baixa demanda. No entanto, em aplicações com altas cargas, é preciso usar coletores alternativos, são eles:

- **throughput collector:** Indicado quando se deseja incrementar o desempenho da aplicação executando em ambientes com um grande número de processadores. O coletor de lixo padrão é realizado em uma *thread*, e leva um tempo de execução puramente serial, o coletor *throughput* usa várias *threads* para realizar pequenas coletas reduzindo assim o tempo total de coleta. Uma situação típica de uso é quando a aplicação tem um grande número de *threads* alocando objetos.
- **concorrent low pause collector:** indicado quando se pretende obter uma menor pausa por coleta e dispõe-se de recursos para compartilhar entre o coletor de lixo e a aplicação em execução. Tipicamente, é indicado para aplicações que tem um conjunto grande de objetos de longa vida e executam em máquinas com dois ou mais processadores. Resultados otimizados podem ser observados com este coletor nessas condições. [Sun Microsystems 2004*b*]
- **incremental:** também conhecido como algoritmo do trem 2.3.1, muito utilizado

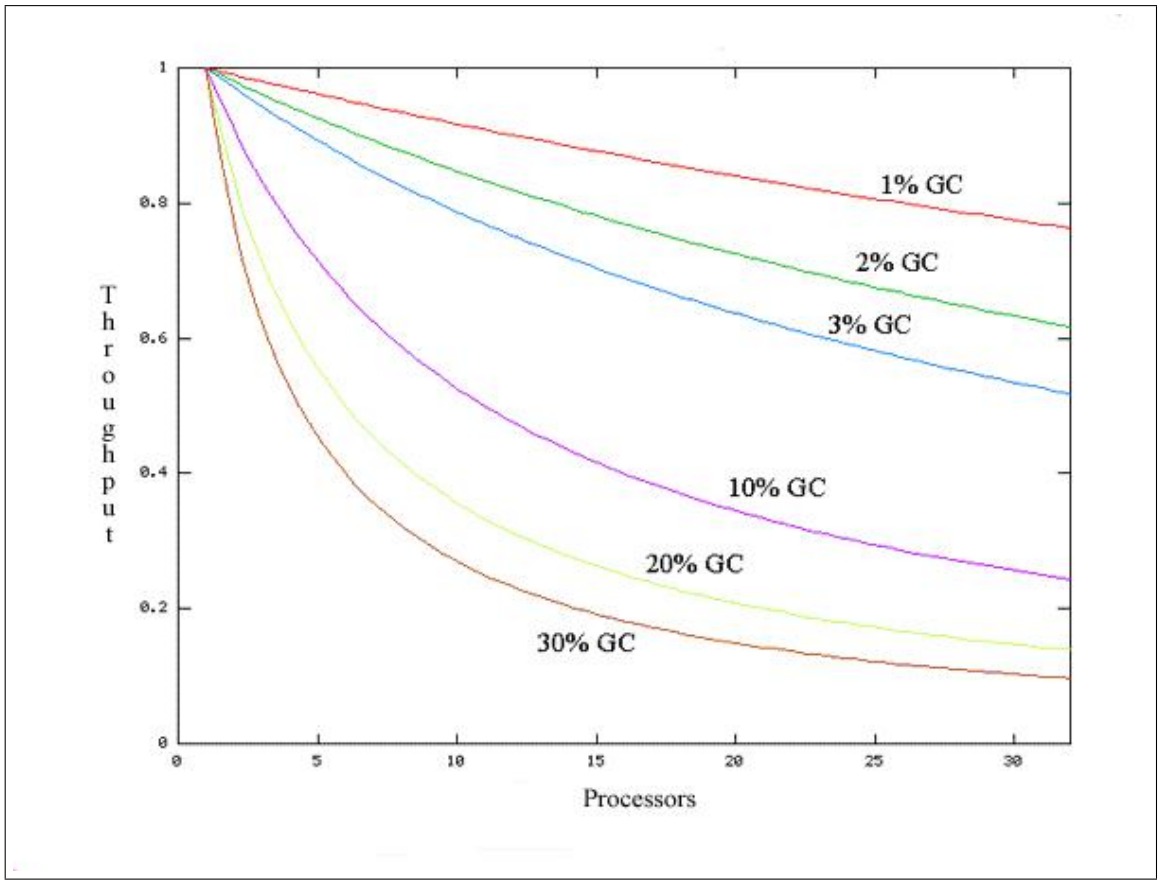

Figura 2.8: Impacto da coleta de lixo x quantidade de processadores

nas JVMs 1.4, este algoritmo está em desuso em ambientes multiprocessados por sua característica sequencial.

2.4 Memory Leak

O *memory leak* é um tipo particular de problema que decorre do consumo não intencional de memória em razão de uma falha do programa na liberação de memória que não é mais necessária. O *memory leak* pode diminuir o desempenho de um sistema uma vez que diminui a quantidade de memória disponível para execução [Satish Chandra Gupta 2005].

Em Java, a coleta de lixo simplifica o gerenciamento de memória e elimina problemas típicos de memória, como a liberação explícita. Entretanto, contrariando o que muitos programadores acreditam, é possível ter *memory leaks* em Java. Este problema ocorre em decorrência de erros do programador que mantém, por um tempo indefinido, referências para objetos sem utilização. Dessa forma, os objetos são categorizados como ativos e não removidos pelo coletor.

A figura 2.9 [Patrick 2001] ilustra um típico uso de memória no decorrer do tempo. No gráfico, a cada 5 minutos o uso de memória é medido. Note que a cada coleta de lixo a quantidade de memória diminui bruscamente. No entanto, há um incremento de uso de memória após a segunda coleta. No decorrer do tempo, o coletor de lixo vai conseguindo liberar cada vez menos memória até que todo o espaço de memória é utilizado por objetos não coletáveis.

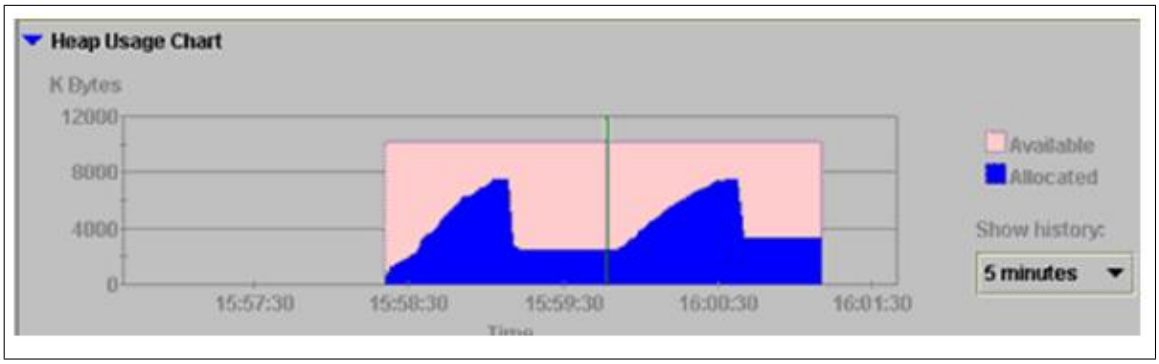

Figura 2.9: Comportamento de Memory-Leak

Este defeito é ocasionado por um erro de programação e sua detecção não é trivial sem o auxilio de ferramentas adequadas. As ferramentas de *profiler*, que serão abordadas no próximo capítulo, são capazes de detectar o *memory leak* e detectar suas causas, permitindo assim que o programador possa fazer a resolução através da alteração de código visando a correção da referência persistente. Não é possível sanar o problema somente pelo uso de ferramentas ou por uma abordagem automática da máquina virtual.

Neste capítulo, através de uma revisão sobre o detalhamento do funcionamento da JVM ficou claro o quanto é importante o domínio do engenheiro sob seus aspectos funcionais e dos algoritmos de coleta de lixo. Faz-se necessário também a utilização de ferramentas para a extração de informações da máquina virtual a respeito de vários aspectos do ambiente de execução, incluindo o uso de memória e coleta de lixo. O próximo capítulo aborda as ferramentas para este propósito.

Capítulo 3

Ferramentas de análise e monitoramento de aplicações

A tarefa de analisar o desempenho de aplicações não deve ser realizada de maneira empírica ou desenvolvida pelo programador. Devem ser empregadas ferramentas que auxiliam o desenvolvimento da aplicação, detectando fragmentos de códigos que consomem maior tempo ou memória para se obter um resultado confiável e profissional.

Faz-se necessário também, depois de passar pelo teste de desempenho no desenvolvimento, acompanhar o comportamento da aplicação em produção, quando os usuários estarão usando a plena carga. Essas ferramentas, denominados de Gerenciamento de Aplicações serão o objeto de estudo deste capítulo.

3.1 JVMTI (Java Virtual Machine Tools Interface)

Uma das ferramentas mais importantes para obter dados relevantes da execução de um programa na JVM é a JVMTI (Java Virtual Machine Tools Interface) [2004 n.d.]. Ela é uma interface de programação para ser usada por ferramentas de desenvolvimento visando obter dados relativos à execução da aplicação que a máquina virtual hospeda, além de fornecer uma maneira de inspecionar o estado e controlar a execução de aplicações. Nem todas as implementações da JVM possuem essa interface e não há a obrigatoriedade da implementação deste componente [Tim Lindholm 1999].

JVMTI é uma interface de duas vias. O cliente do JVMTI, também chamado de agente, pode ser notificado de ocorrências de interesse através de eventos. JVMTI pode consultar e controlar a aplicação através de várias funções, sejam em respostas a eventos ou independentes deles.

Os agentes são executados no mesmo processo da máquina virtual e comunicam-se diretamente com ela executando a aplicação a ser analisada. A comunicação é feita através da interface nativa do JVMTI que, dentro do processo, permite o máximo controle com uma mínima intrusão. Normalmente os agentes são compactos e podem ser controlados por um processo separado como serão os casos dos *profilers* que serão analisados mais a frente neste capítulo.

Os agentes são usualmente escritos em linguagens nativas como C. Para eles, são disponibilizadas um conjunto de funções que permitem extrair a situação atual da execução

de um software, inclusive os dados de desempenho. A interface JVMTI permite obter informações acerca dos seguintes aspectos:

- Gerenciamento de Memória;
- *Threads*;
- *Heap*;
- Variáveis, métodos, objetos, atributos;
- Eventos, *Breakpoints*, dentre outros.

3.2 Profiler

A análise de desempenho (também conhecida como analise dinâmica de desempenho) é a investigação do comportamento de um programa através da sua execução, em aposto com a análise estática do código, realizada através de inspeção do programador. A meta geral de uma análise de desempenho é determinar que partes do programa otimizar visando o incremento da velocidade ou diminuição de uso de memória.

Um *profiler* é uma ferramenta de análise de desempenho que mede o comportamento do programa enquanto ele é executado, particularmente a freqüência e duração das chamadas de métodos ou funções. A saída gerada por um *profiler* são as estatísticas obtidas em cada chamada de método, tais como: uso de memória, tempo de processador, uso de disco, dentre outros. A essa saída, denomina-se *profile*. Os *profilers* usam uma variedade de técnicas para coletar dados, incluindo interrupção de processador , instrumentação de código, *hooks* de sistemas operacional e contador de desempenho (*performance counters*) [ATOM 1994] [Whaley 2000].

Fica evidente que um *profiler* é uma ferramenta poderosa quando se pretende fazer a análise de desempenho de um modo dinâmico, isto é, a investigação do comportamento de uma aplicação durante sua execução, visando identificar, por exemplo, partes do código que demandam otimização. Também fica claro que ferramentas de análise de desempenho são importantes para se entender o comportamento do programa objeto da análise. Arquitetos de software necessitam dessas ferramentas para analisar como os programas iriam funcionar em uma dada arquitetura, bem como desenvolvedores necessitam dessas ferramentas para analisar seus programas e identificar pontos críticos do código. O mesmo ocorre com projetistas de compiladores que frequentemente as usam para identificar porções críticas de códigos e para analisar o escalonamento de instrução (*instruction scheduling*) ou algoritmos de predição de blocos (*branch prediction*) [ATOM 1994].

Alguns profilers operam por amostragem, através da sondagem do registrador PC (*Program Counter*) [Whaley 2000] realizadas por interrupções de sistema regulares. Esta abordagem é menos exata e específica, mas permite que o programa em análise execute próximo de seu desempenho máximo, ou seja, influencia pouco na execução.

Outros *profilers* instrumentam o programa alvo através da adição de instruções para coletar as informações requeridas. O uso da instrumentação do programa pode causar mudanças em seu desempenho, tornando a execução do código mais lenta, devido ao impacto da coleta dos dados.

3.2.1 Uso de *Profilers* em Java

Os programas desenvolvidos na plataforma Java não necessitam usar profilers por amostragem nem por instrumentação direta de código. Estas técnicas são utilizadas em *profilers* tradicionais focados em linguagens de código nativo, isto é, que executam diretamente no hardware. Como descrito na seção 3.1, um programa em Java executa em uma máquina virtual que pode fornecer os dados sobre o ambiente de execução através do JVMTI (*Java Virtual Machine Tools Interface*). É possível configurar a máquina virtual para informar os eventos e as estatísticas de execução para um agente, conforme ilustrado na figura 3.1.

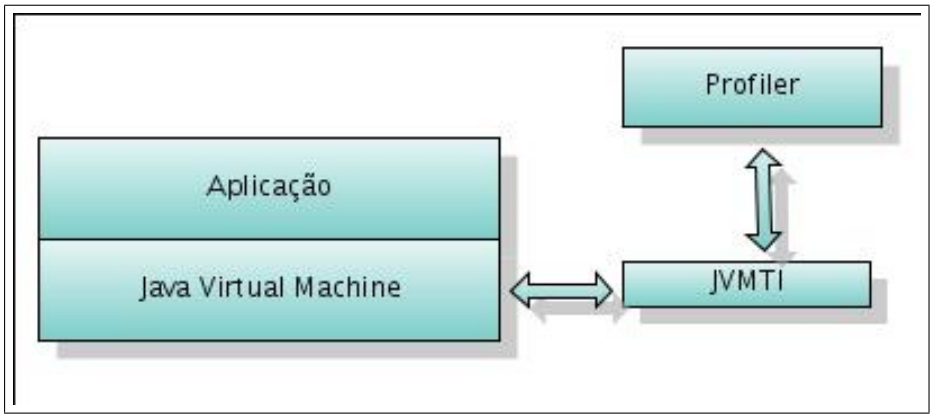

Figura 3.1: Profiler, JVM e o JVMTI

Assim, durante a execução num intervalo de tempo em particular, o profiler capta os dados informados pela JVMTI e os armazena para, posteriormente, apresentar os resultados sumarizados para o usuário.

NetBeans Profiler

O *NetBeans profiler* é um produto livre desenvolvido com financiamento da Sun MicroSystems que possui as seguintes funcionalidades [NetBeans 2006].

- **Profiling de baixo overhead:** tem como característica a possibilidade de execução de um subconjunto do código, e como o resto do código não é instrumentado, ele é executado na velocidade máxima. O subconjunto que é analisado e o tipo de *profiling* (CPU, memória, etc) pode ser modificado durante do momento de execução
- **Profiling de desempenho de CPU:** Pode-se analisar a aplicação inteira, ou selecionar um subconjunto do código ou um método raiz. Por exemplo, é possível definir o método *doGet* em sua Servlet [Eric Jendrock 2006] como a raiz, e então analisar apenas o código do método e os demais que são chamados por ele. O resto do

código, incluindo o código de seu servidor Web, executa em velocidade total. *Profiling* seletivo baseado em métodos raízes é uma boa maneira de reduzir o *overhead* de desempenho e obter resultados reais para aplicações com *multithreading* pesada sob circunstâncias realísticas.

- **Profiling de Memória:** possibilita apenas a análise de eventos de criação de objetos, que impõe menos *overhead* e permite detectar problemas com número excessivo de alocação de objetos. Se isto não for suficiente, é possível gravar as criações de objetos e todos os outros eventos. Isto permite visualizar o número, idade, caminhos de alocação, e outras características dos objetos que estão no momento em memória. Esta funcionalidade também pode ajudar a detectar *memory leaks.*
- **Debuging de *Memory Leak*:** *memory leaks* não são tão raros em aplicações Java muito grandes. Com a maioria das ferramentas existentes, levam-se horas para ter certeza que o comportamento é realmente decorrente de um *memory leak*, e então identificar qual é o objeto causador. No *NetBeans profiler* é feita a análise do objeto que está sendo criado e referenciado imediatamente. Pode-se então verificar onde estes objetos são alocados e quem os referencia para identificar o causador do memory leak.
- **Theads Profiling:** através dessa funcionalidade pode-se observar as *threads* e sua atividade durante a execução da aplicação. Este *profiling* oferece duas visões das *threads*: A linha de tempo, mostrando a atividade das *threads*, passo a passo, e seus detalhes, permitindo selecionar uma *thread* em particular e analisar seu comportamento.

As figuras 3.2, 3.3 e 3.4 [NetBeans 2006] ilustram, respectivamente, dados obtidos relativos à utilização de memória, quantidade de threads e detalhamento da execução do código.

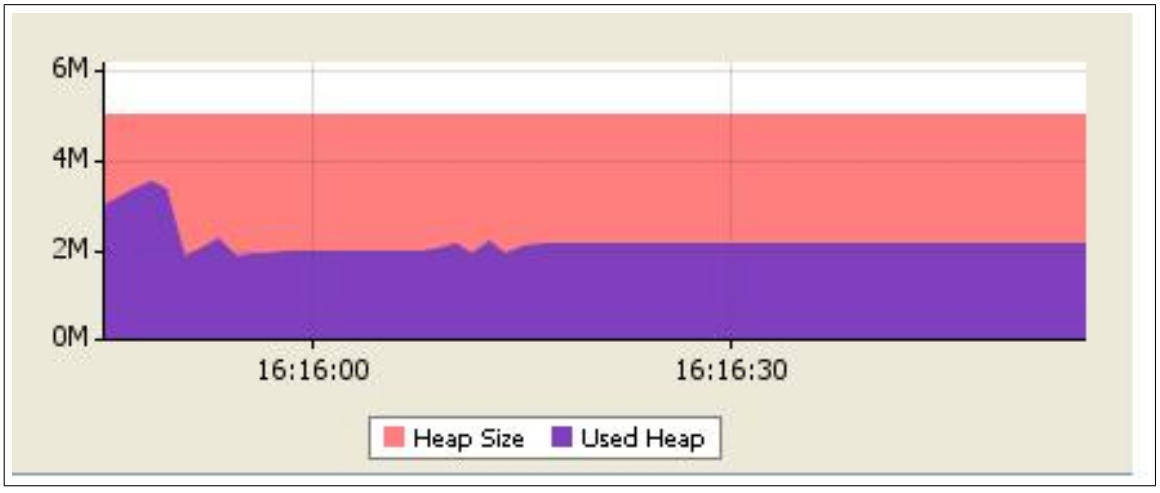

Figura 3.2: Dados da utilização do Heap pelo NetBeans Profiler

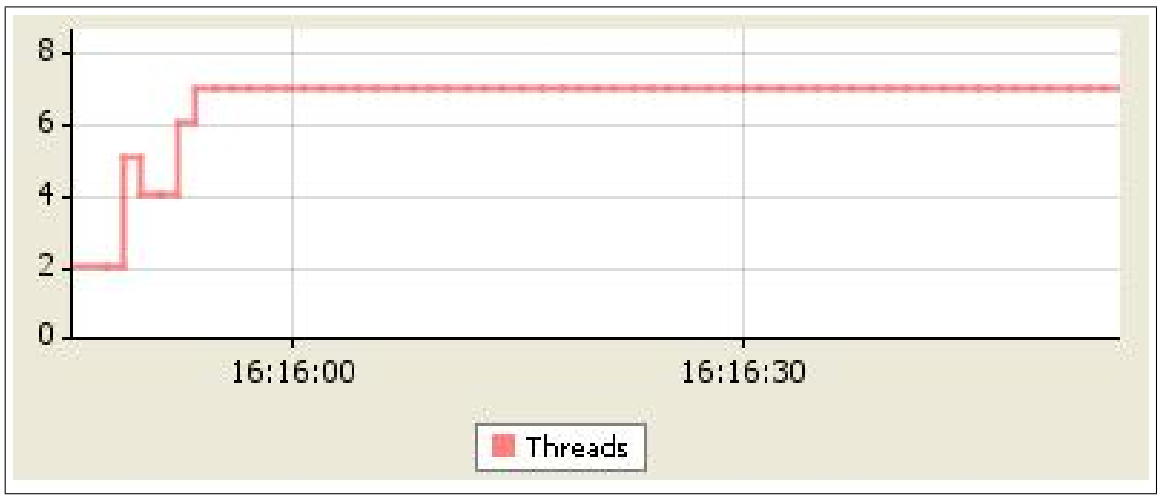

Figura 3.3: Quantidade de threads por tempo de execução da aplicação.

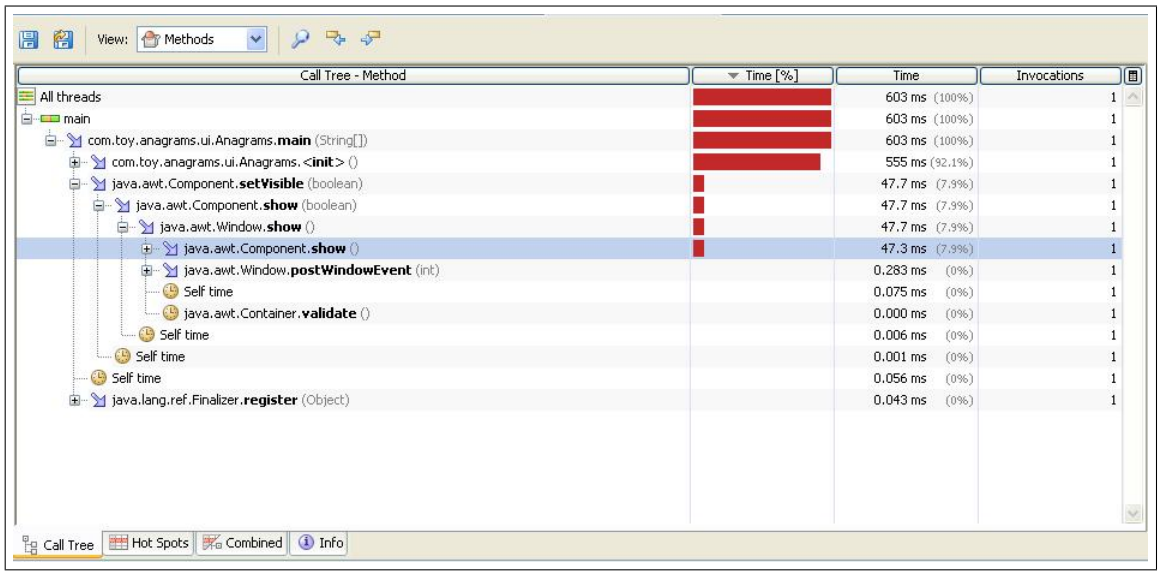

Figura 3.4: Desempenho detalhado de um código em termos de métodos e classes.

3.3 Gerenciamento e Monitoramento de Aplicações

Semelhante a outros sistemas, tais como sub-estações elétricas ou usinas nucleares, sistemas de software necessitam de monitoramento para analisar se o comportamento em produção condiz com o comportamento esperado. Através desta operação é possível detectar problemas que frequentemente não se mostram nas etapas de desenvolvimento, além de permitir o acompanhamento da execução da aplicação. Em Java, foi definida uma padronização visando esta funcionalidade, o gerenciamento de aplicações: o JMX (*Java Management Extension*) [Fleury 2002].

A especificação JMX [Sun Microsystems 2002] define uma arquitetura, um padrão de projeto, a API e serviços para o gerenciamento e monitoramento de aplicações Java pela rede. Através desta tecnologia é possível gerenciar aplicações Java inclusive através de protocolos padrões, como o SNMP (*Simple Network Simple Managment*) [Tanembaum 2003].

A máquina virtual Java 6.0, que é a utilizada neste trabalho, possui um conjunto de objetos JMX que instrumentam um conjunto de dados da máquina virtual em execução, tais como: memória, *threads*, classes carregadas, dentre outros. Uma das ferramentas de grande utilização é JConsole [Chung 2004], distribuída juntamente com o JDK (*Java Development Kit*). As figuras 3.5 e 3.6 [Chung 2004] ilustram dados obtidos através do JConsole.

A diferença entre os *profilers* e as aplicações de gerenciamento é que estas não necessitam de um processo obtendo informações diretamente da máquina virtual através do JVMTI, por exemplo. Não é necessário ter acesso ao processo para por um agente extrator de dados em execução. O gerenciamento deve ser realizado remotamente e com o mínimo de impacto no desempenho, uma vez que a aplicação está em produção. Estas ferramentas, denotam um importante papel na detecção de possíveis problemas em produção.

Note que com essas duas categorias de ferramentas é possível detectar problemas de desempenho durante o desenvolvimento, através dos profilers, como também monitorar o comportamento da aplicação em produção, através das ferramentas de gerenciamento.

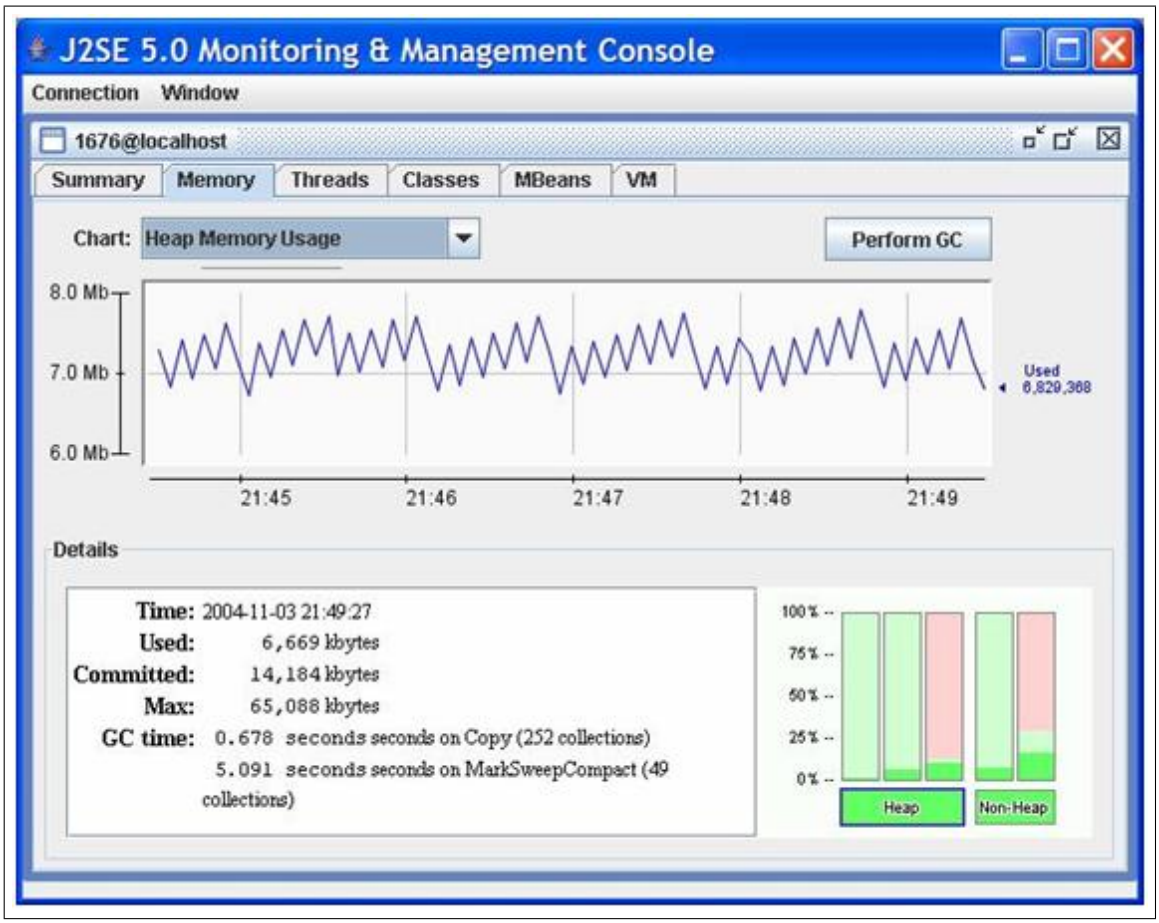

Figura 3.5: Uso de Memória da máquina virtual obtida pelo JConsole.

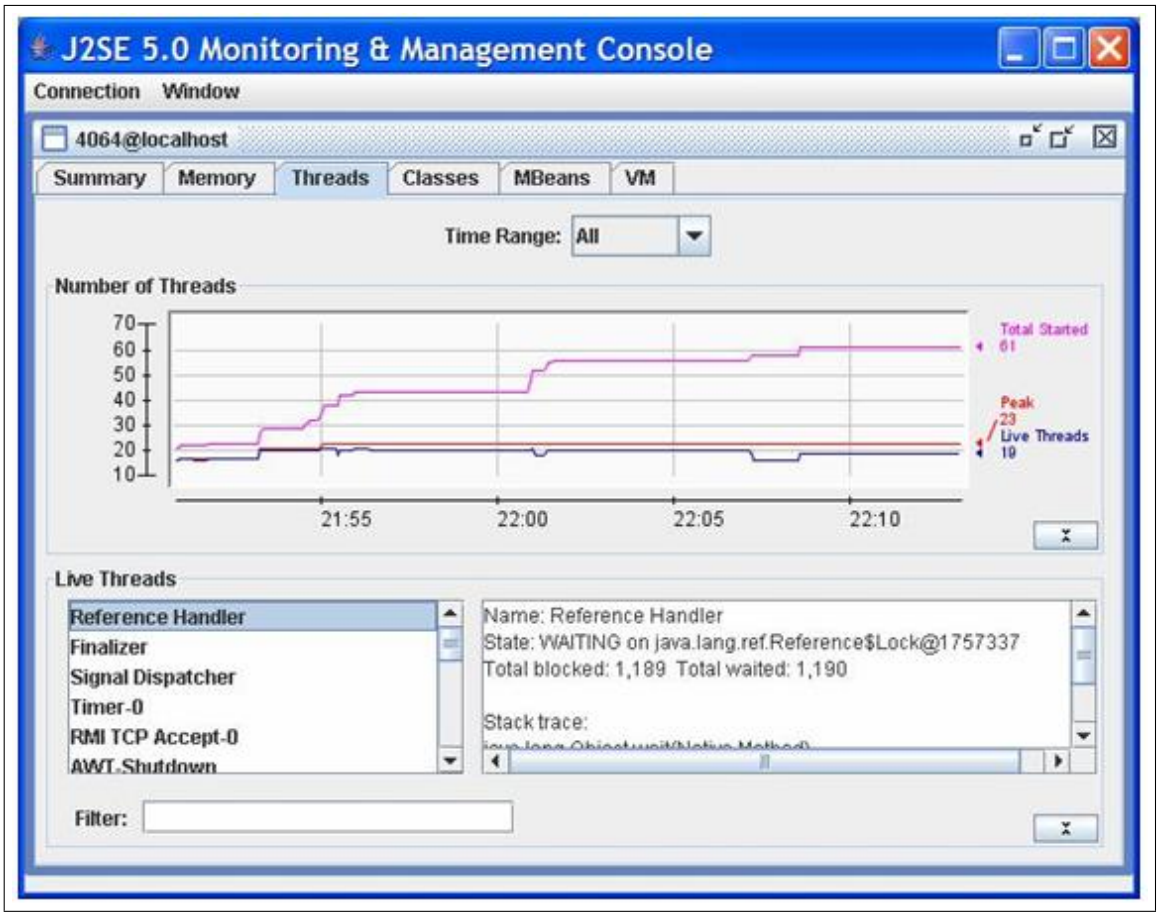

Figura 3.6: Gerenciamento de Threads através do JConsole.

Capítulo 4

Soluções e Camadas no contexto da aplicação

Nos capítulos anteriores, foram analisados os fatores do ambiente que influenciam no desempenho de um sistema visando um melhor aproveitamento dos recursos disponíveis através da adaptação do ambiente ao tipo de aplicação desejada. O estudo também contemplou a descrição de algumas ferramentas que possibilitam fazer a análise de desempenho do código da aplicação (*profiler*) e o monitoramento de ambientes de produção ilustrando a ferramenta JConsole, com ênfase no controle e detecção de problemas de desempenho.

Apesar da otimização da infra-estrutura ser essencial, em muitas situações, de nada adianta um ambiente de execução otimizado se a aplicação não o é, haja vista que seu desempenho influencia decisivamente no grau de eficência do sistema como um todo.

No cenário atual, há disponibilidade de diversos frameworks e tecnologias de qualidade, de modo que o desenvolvedor utiliza soluções já consolidadas para servir como base no desenvolvimento de suas aplicações. Estas tecnologias evidenciam suas diferenças nas APIs *(Aplication Programing Interface)* [Reese 2000] e no modelo de programação. Entretanto, existe pouco material na literatura que possibilite avaliar o desempenho dessas tecnologias.

De acordo com a arquitetura a ser utilizada num dado projeto de software, a avaliação de desempenho pode ser direcionada para uma ou mais camadas, conforme o interesse do desenvolvedor. Nessa perspectiva e considerando que é usual nos projetos modernos a arquitetura em três camadas [John Crupi 2004], a seguir serão descritas as características de cada um dos níveis dessa arquitetura, destacando-se as tecnologias Java disponíveis para cada um deles.

- **Camada de Persistência**: Responsável pela recuperação, gravação e atualização de objetos em bancos de dados relacionais. Nesta área destacam-se as tecnologias Hibernate [Chistian Bauer 2005], EJB3 [Rima Patel Sriganesh 2006] e JDBC [Reese 2000].
- **Camada de Visualização ou Apresentação:** Responsável pela geração da interface com o usuário, captação e manipulação dos dados e controle de navegação. Destacam-se, tecnologicamente, os *frameworks* Struts[Husted 2004*b*] e JavaServer Faces [Husted 2004*a*].

- **Camada de Negócio:** Responsável pelo processamento da lógica de negócio. Normalmente é ativada pela camada de visualização, interagindo com os objetos recuperados ou criados pela camada de persistência. Destacam-se as tecnologias EJB [Rima Patel Sriganesh 2006] e Web Services [Cerami 2002*b*].

A Figura 4.1, representa um gráfico de relacionamento destas três camadas supramencionadas:

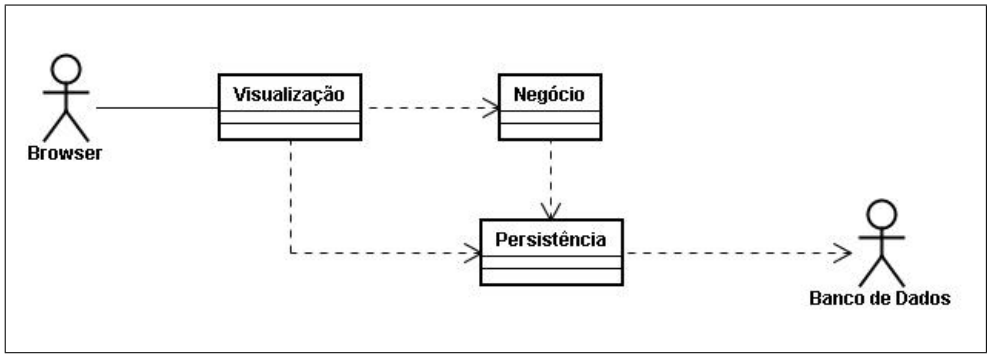

Figura 4.1: Modelo de uma arquitetura em 3 camadas

Como existem várias tecnologias para tratar dos aspectos inerentes de cada camada, uma escolha inadequada de alguma delas, bem como uma má utilização do ambiente, pode ter um impacto global significativo no desempenho de uma aplicação particular. Assim, para cada uma dessas camadas serão examinadas algumas soluções possíveis, utilizando tecnologias ou projetos distintos. Para cada uma das tecnologias a serem analisadas, espera que a relação produtividade no desenvolvimento e desempenho sejam antagônicas, ou seja, quanto maior a produtividade menor o desempenho. Assim, ter-se-á qual tecnologia espera-se ser mais rápida ou mais lenta. Este capítulo introduz todas essas tecnologias ilustrando seu modo de funcionamento. No próximo capítulo, os testes comparativos serão realizados.

4.1 Camada de Mapeamento Objeto Relacional

Historicamente, os bancos de dados relacionais se destacaram com uma posição chave no mercado. A simplicidade de seu modelo e a base matemática fornecida pela álgebra relacional [F. Henry Korth 2006] forneceu a este produto a confiabilidade e robustez necessárias para dominar o mercado [Group n.d.].

Com a criação e popularização das linguagens orientadas a objetos, rapidamente os desenvolvedores começaram a notar a flexibilidade deste paradigma de programação. Características como reusabilidade e extensibilidade mostraram-se mais evidentes e fortes. A construção das aplicações evoluiram, mas, a forma de sua comunicação com as bases

de dados relacionais não tiveram o mesmo desenvolvimento, tornando a porção do código que acessa o banco de dados semelhante ao paradigma das linguagens estruturadas.

Para proporcionar uma melhoria na comunicação entre linguagens orientada a objetos e bancos de dados, introduziram-se os sistemas gerenciadores de bancos de dados (SGBD) orientado a objetos [Bancilhon 1992]. Nesta abordagem, a persistência de objetos era automática e não havia necessidade de uso da linguagem SQL, por exemplo. Os SGBDs orientados a objetos não se popularizaram em sistemas para corporações devido a ausência de mecanismos eficientes para consulta de dados [Bancilhon 1992], principalmente pela falta de uma base matemática como a álgebra relacional.

Como os bancos de dados orientado a objetos foram descartados do cenário exposto, surgiu a demanda de melhorar a integração entre aplicações desenvolvidas orientada a objetos e banco de dados relacionais. A solução de mapeamento objeto relacional (MOR) foi a que mais se destacou sendo incorporada em vários sistemas. Nesta solução, há uma porção da aplicação que cuida de transformar objetos em dados relacionais e dados relacionais em objetos. Hoje em dia, tecnologias de mapeamento objeto relacional fazem essa tarefa de maneira automática mediante configuração do programador.

Neste contexto, e dentro da perspectiva deste trabalho serão abordados três soluções: a primeira solução é a abordagem de acesso a banco de dados sem utilizar mapeamento objeto relacional, o JDBC. A segunda é a restruturação do código em JDBC para se adequar aos padrões de MOR. Finalmente, a tecnologia Hibernate, que faz o mapeamento objeto relacional automático, será analisada.

4.1.1 JDBC

Na plataforma Java a primeira maneira de interagir com bancos de dados foi utilizando a API JDBC (*Java Database Connectivity*)[Reese 2000]. Esta API, apesar de utilizar uma linguagem orientada a objetos, tem no seu código uma pequena semelhança a uma abordagem estruturada. O exemplo abaixo ilustra um código para recuperar os dados de uma tabela utilizando JDBC:

```
Statement st =  conexao.createStatement();
ResultSet rs =
st.executeQuery("SELECT * FROM ALUNO");
while ( rs.next() ) {
    String valorColuna1 = rs.getString("NOME");
    String valorColuna2 = rs.getString("MATRICULA");
}
```

O código de acesso aos dados do banco, normalmente está distribuído em toda a aplicação e em grande quantidade. A abordagem de acesso através do JDBC gera uma grande replicação de código e uma abordagem semi-estruturada. Visando melhorar a organização desse código, os engenheiros de software começaram a trabalhar em padrões de projetos que pudessem manter a utilização destas APIs, porém, aumentando a qualidade do código

através de mapeamento objeto relacional. Assim, surgiram alguns padrões, dentre eles o DAO (*Data Access Object*) [John Crupi 2004]

4.1.2 Data Access Object

O padrão de projeto DAO propõe-se a abstrair e encapsular todo o acesso ao banco de dados. Ele gerencia as consultas, recuperação e gravação de objetos no banco de dados, retornando, a quem o invocou, objetos ao invés de linhas (tuplas) de banco de dados. A listagem de código abaixo ilustra um código semelhante ao da abordagem JDBC porém com mapeamento objeto relacional.

```
public Collection<Aluno> findAllAlunos() {
    Statement st = conexao.createStatement();
    ResultSet rs = st.executeQuery("SELECT * FROM ALUNO");
    ArrayList<Aluno> resultado = new ArrayList<Aluno>();
    while ( rs.next() ) {
        Aluno aluno = new Aluno();
        aluno.setNome(rs.getString("NOME"));
        aluno.setMatricula(rs.getString("MATRICULA"));
        resultado.add(aluno);
    }
    return resultado;
}
```

Note que, pelo fato do DAO ser um padrão de projeto, não houve alteração na tecnologia, apenas uma re-estruturação do código seguindo uma nova filosofia. Nesta abordagem, os dados recuperados do banco de dados são transformados em objetos e então retornados através de uma coleção de objetos.

É importante ressaltar que essa nova abordagem na comunicação com o banco de dados representa várias vantagens sob a ótica da engenharia de software, pois além de diminuir a dependência com a estrutura da base de dados, centraliza todo o acesso aos dados nas classes DAO. Porém, comparado com a abordagem JDBC, implica a criação de um objeto para cada linha retornada da base de dados, gerando mais demanda para a coleta de lixo em tempo de execução.

A estruturação do projeto com DAOs tem vantagens na ótica da engenharia de software. Normalmente, são criadas classes para gerenciar a persistência das tabelas em objetos, entendendo-se por persistência [John Crupi 2004] o conjunto de operações para mapear a tabela do banco de dados em objetos da aplicação. O padrão de projeto DAO segue a filosofia do mapeamento objeto relacional, gerando assim, na sua estrutura mais simples, classes de mapeamento que tratam das operações de **C**reate, **R**ecovery, **U**pdate e **D**elete (CRUD).

Os métodos de persistência são implementados para cada tabela no banco de dados. Por exemplo, um sistema com 30 tabelas, resulta em 30 classes de persistência com 4

métodos, no mínimo, para cada um. Estipulando uma média de 10 linhas de código para cada método, resulta em 30 x 10 x 4 = 1200 linhas de código.

A primeira vista o uso de DAO implica em uma grande quantidade de linhas de código, entretanto, na abordagem JDBC no mínimo as mesmas 1200 linhas precisam ser desenvolvidas, porém, com uma grande probabilidade de serem replicadas e consequentemente ultrapassarem facilmente esse número. Para equacionar estes problemas, que afetam diretamente a produtividade de projetos, surgiram tecnologias que se propõem a realizar este mapeamento de forma automática, bastando configurar certos comportamentos.

4.1.3 Hibernate

A tecnologia Hibernate [Chistian Bauer 2005] tem como propósito mapear objetos Java em tabelas de banco de dados, visando, no exemplo, eliminar as 1200 linhas necessárias para a persistência, tornando este processo automaticamente gerenciado. Para isso, é necessário incluir anotações [David Flanagan 2004] na classe informando como esta persistência será realizada. No contexto do código a seguir está configurada a persistência para a classe *Aluno*.

```
@Entity
    @Table(name="tbl_aluno")
    public class Aluno  {
    @Id
    private int id;
    @Column(name="desc_aluno")
    private String nome;
    @Column(name="matricula_aluno")
    private String matricula;
}
```

A persistência das classes com Hibernate é estabelecida quando este é inicializado, ocasião em que as classes configuradas como persistentes e suas anotações inspecionadas (@Entity, @Table, etc). A partir destas anotações, é possível saber, por exemplo, que os objetos da classe *Aluno* são persistidos na tabela *tbl_aluno*, que a chave primária é o atributo *id* e que *nome* e *matricula* são armazenados nas colunas *desc_aluno* e *matricula_aluno*.

Com a posse dessas informações, o *framework* é capaz de realizar toda a persistência de maneira automática, sem a necessidade de desenvolvimento de código JDBC. Na realidade, o código SQL é gerado dinamicamente pelo Hibernate. Assim, é retirado do desenvolvedor essa carga de trabalho. Em consequência, a produtividade no desenvolvimento de aplicações que utilizam banco de dados é significativamente incrementada, principalmente em projetos onde há muitas tabelas. Porém, a geração de código SQL automático tem um custo de execução, pois adiciona um *overhead* na consulta.

Estas abordagens (JDBC, DAO e Hibernate) têm grau crescente de produtividade e decrescente de desempenho. Na grande maioria dos casos, em aplicações que não são altamente demandandas, a perda do desempenho é baixa comparada com o custo benefício para o projeto. Para comparar esse desempenho, uma aplicação foi desenvolvida para ser testada sob condições de carga elevada. A seção a seguir descreve esta aplicação.

4.1.4 Aplicação para comparação de desempenho

Para testar as três soluções supramencionadas, um banco de dados foi criado no PostgreSQL (`www.postgresql.org`) com a estrutura de tabelas ilustrada na figura 4.2:

Figura 4.2: Modelo Relacional da Aplicação de Teste da Camada de Mapeamento.

Cada arquitetura, em conformidade com seu modelo de desenvolvimento, irá recuperar uma venda, o funcionário e a empresa associados. Ou seja, a consulta de seleção irá envolver as três tabelas. Esta consulta será a mesma para as três soluções, o que irá variar é a maneira de recuperar os resultados peculiares de cada solução.

A base de dados se encontra previamente preenchida com dez vendas, cada uma com um funcionário e empresa diferentes. Cada requisição deve recuperar todas as vendas. A instância do problema em questão é ter diversos usuários, ao mesmo tempo e repetidas vezes, recuperando dez vendas utilizando cada uma das soluções.

Para que o teste seja fiel é importante que fatores externos não constribuam para degradar o desempenho. O custo de abertura e fechamento de conexões com o banco de dados é um desses fatores. Se para cada requisição, uma conexão com a base de dados fosse aberta e no final fechada, há um forte impacto no tempo devido a autenticação da base de dados. Para equacionar este problema, foi usado um *pool* de conexões fornecido pelo servidor de aplicações JBoss (`www.jboss.org`), ambiente de execução de todos os testes, onde os seguintes parâmetros foram configurados para o *pool* de conexões:

```
<local-tx-datasource>
   <jndi-name>dsMestrado</jndi-name>
   <connection-url>jdbc:postgresql://host:5432/mestrado</connection-url>
   <driver-class>org.postgresql.Driver</driver-class>
   <user-name>usuario</user-name>
   <password>senha</password>
```

```
    <min-pool-size>1</min-pool-size>
    <max-pool-size>50</max-pool-size>
    <blocking-timeout-millis>5000</blocking-timeout-millis>
    <idle-timeout-minutes>1</idle-timeout-minutes>
</local-tx-datasource>
```

O *pool* de conexões deve gerenciar a abertura e fechamento de conexões variando de 1 a 50. Uma requisição usa uma conexão e devolve para o *pool* que em seguida fornece aquela conexão para outra requisição. Caso o número limite de conexões em uso seja atingido e uma amostra necessitar de conexão, ela espera o tempo parametrizado acima como *blocking-timeout-millis*. Caso este tempo seja excedido, a requisição é descartada uma vez que um erro é emitido. Assim, há 50 conexões compartilhadas entre todas as solicitações processadas. O mesmo *pool* de conexões será utilizado para todas as soluções testadas.

4.2 Camada de Visualização

A camada de visualização com a grande popularização da Internet tem sido representada por tecnologias Web. Através da internet os mais variados tipos de aplicações são desenvolvidas hoje em dia e com grande sucesso comercial. Neste cenário, a plataforma Java Enterprise Edition (JEE) [Eric Jendrock 2006] tem um papel essencial como uma das líderes no desenvolvimento de soluções corporativas. Entre as tecnologias que circundam o contexto do JEE, destacam-se Servlets, JSP [Eric Jendrock 2006], Struts[Husted 2004*b*] e JSF[Husted 2004*a*].

As Servlets surgiram em 1997 com o objetivo de competir com os pesados programas CGIs (Common Gateway Inteface) [Eric Jendrock 2006] que eram utilizados para geração de conteúdo WEB naquela época. As Servlets são classes Java que executam em contêineres e são capazes de processar requisições WEB. A grande desvantagem desta tecnologia é a mistura de código HTML dentro de código Java, tornando a separação de visualização e controle uma árdua tarefa. As vantagens residem também no fato dela ser uma tecnologia de baixo nível, ou seja, ao passo que exige mais código do programador, também permite um maior desempenho.

Visando solucionar o problema da mistura de código HTML e Java dentro da classe, surgiu, em 1998, a tecnologia JSP (JavaServer Pages). JSP permite realizar o oposto, utilizar código Java dentro de páginas HTML, realizando assim uma melhor separação, tendo em vista que, o código HTML é, em geral, maior que o código Java. JSP não representa uma tecnologia totalmente diferente de Servlets, pelo contrário, é apenas um nível de abstração, já que toda página JSP, depois de passar por um processo de compilação, torna-se uma Servlet. Isso significa dizer que apesar de JSP tornar o desenvolvimento mais produtivo para o programador, ao fim do processo, ele vira uma Servlet. Esse processo de transformação é realizado apenas no primeiro acesso, não onerando significativamente o desempenho. Do segundo acesso em diante, a JSP é ignorada e a servlet é acessada diretamente.

- Exemplo de uma Servlet:

```
public class Servlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res ) {
        PrintWriter out = res.getWriter();
        out.println("A raiz de 4 é : "  + Math.sqrt(4));
    }
}
```

- Exemplo de uma JSP:

```
<%= "A raiz de 4 é " + Math.sqrt(4) %>
```

No exemplo do código acima, quando a página JSP é acessada pela primeira vez, ela é traduzida para a Servlet exibida. Teoricamente, os dois devem ter desempenhos semelhantes, uma vez que são equivalentes.

Estas duas tecnologias, Servlets e JSP, representam a base fundamental do processamento de solicitações WEB utilizando a plataforma Java. Todavia, uma aplicação WEB moderna necessita de muitos requisitos, tais como:

- **Captação dos dados de formulários:** Os dados informados pelos usuários em formulários WEB devem ser facilmente tratáveis de acordo com os mais variados tipos inseridos (números, datas, CPFs, CEPs, etc)
- **Validação de dados:** Confrontar a entrada de dados dos usuários com um modelo de validação que informa se as informações estão semânticamente válidas ou não.
- **Geração de conteúdo:** A tecnologia deve ser capaz de gerar conteúdo para os mais variados tipos de dispositivos. Nas aplicações atuais, existem diversos canais de apresentação, tais como celulares, notebooks, PDA (Personal Digital Assistante), dentre outros.
- **Produtividade com desenvolvimento baseado em componentes:** o desenvolvimento *desktop* é baseado na filosofia de agrupar componentes de interface para montar visualizações de alta qualidade. As aplicações WEB também necessitam dessa característica visando equiparar, em produtividade, às aplicações *desktop*.

Existem vários outros requisitos que competem às aplicações WEB modernas. As tecnologias Servlet e JSP, apesar de permitirem atender esses requisitos de alguma maneira, não os atende de forma satisfatória na relação custo-benefício. A solução para tal problema foi criar tecnologias de mais alto nível que possibilitasse atender aos requisitos abstraindo as tecnologias base (Servlets e JSP). Tais como: Struts, WebWork [Inc. 2005], Velocity [Project 2005] e mais recentemente o JavaServer Faces.

A Figura 4.3 mostra um relacionamento hierárquico entre as tecnologias mencionadas. Quanto mais baixa a camada da solução, mais rapidamente o código é executado pela JVM, tendo assim sua execução global mais rápida. Todavia, também possui mais linhas de código escritas pelo desenvolver levando a uma menor produtividade por ocasião do desenvolvimento. Isto é, o fator produtividade da tecnologia caminha na direção contrária do desempenho, quanto mais acima estiver a tecnologia, maior é a abstração e produtividade para o desenvolver, mas menor será o desempenho na velocidade de execução.

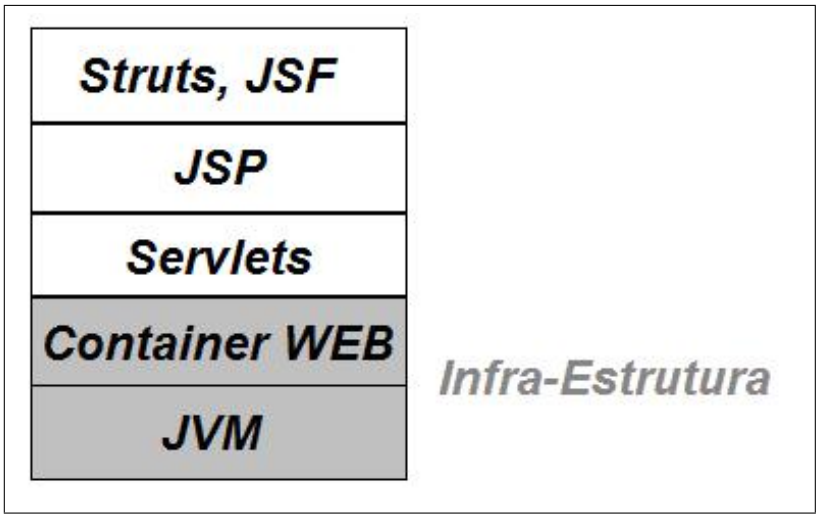

Figura 4.3: Hierarquia das tecnologias envolvidas na camada WEB

O propósito deste trabalho é quantificar essa perda de desempenho através de testes de estresse para o mesmo problema desenvolvido com cada uma das tecnologias: JSP, Struts e JSF. Porém, antes da elaboração dos testes, que serão abordados no próximo capítulo, é preciso detalhar como funcionam as tecnologias Struts e JSF, visando proporcionar uma melhor análise dos resultados.

4.2.1 Struts

Struts é um dos frameworks mais conceituados e maduros do mercado. Um dos primeiros a introduzir o conceito de desenvolvimento MVC (*Model-View-Controller*) [John Crupi 2004], no qual o modelo é composto pelos objetos de negócio e do domínio do problema, a visualização é composta dos códigos das páginas e o controle contém os fluxos de navegação e captação de dados do usuário. O Struts é um projeto Open-Source e usado em muitos sistemas corporativos.[Eric Jendrock 2006]

O funcionamento (figura 4.4)[Souza 2004] baseia-se no processamento de requisição através de classes de ações (*Action*). Cada solicitação que é enviada pelo browser tem seus dados preenchidos e validados em uma classe que representa os dados do formulário gerados pelo cliente (*ActionForm*). Esse objeto, com os dados oriundos do cliente, é repassado para a classe, que realiza as ações sobre os dados, invocando a camada de negócio. O resultado do processamento então é enviado para uma visualização utilizando JSP ou Tag Libraries [Eric Jendrock 2006] (componentes Web de visualização).

É visível que a solicitação processada através de Struts possui uma alta qualidade em termos de modelagem e manutenção. A divisão clara entre Modelo, Visualização e Controle (MVC) torna o código mais adaptável, modular e extensível. A questão que se coloca é o quanto essa organização traz de impacto no desempenho. A resposta a este

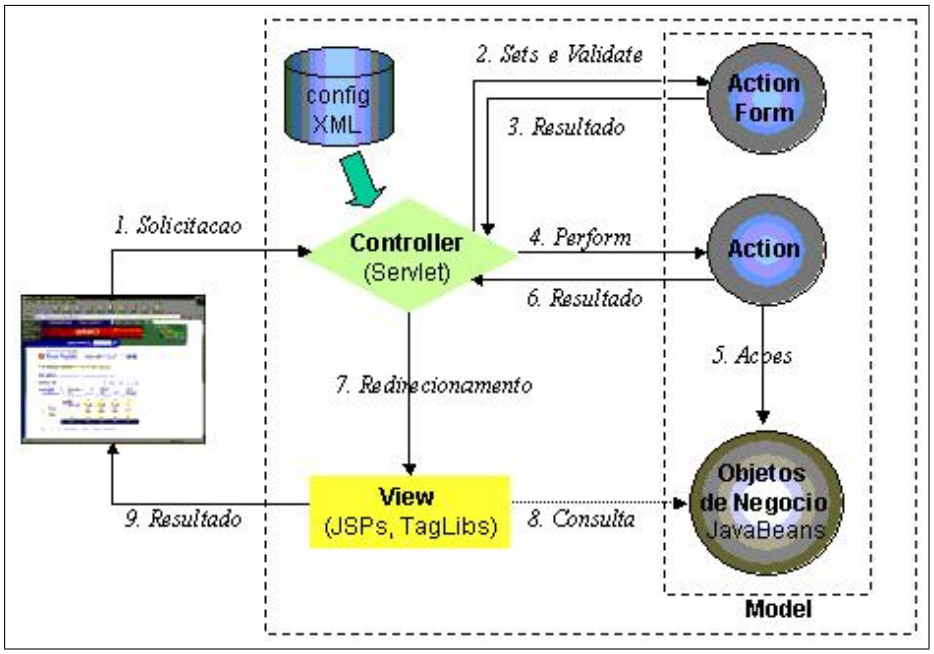

Figura 4.4: Fluxo de processamento de uma requisição com Struts.

questionamento será aborada no capítulo 5.

4.2.2 JavaServer Faces - JSF

A tecnologia JSF (JavaServer Faces) surgiu como uma evolução dos frameworks de processamento de requisições baseado em ações, para partir para um modelo de processamento inspirado nas aplicações para ambientes *desktop*. Segundo a Sun Microsystems, líder da especificação da tecnologia, a tecnologia JSF pode ser caracterizada da seguinte maneira:[Husted 2004*a*]

"Simplifica o desenvolvimento de interfaces gráficas para aplicações Java servidoras. Desenvolvedores de variados níveis podem rapidamente construir aplicações web através de: montagem de componentes reusáveis de interface gráfica; conexão desses componentes com fontes de dados; e pela escrita de eventos gerados pelo cliente e processados pelo lado servidor".

Projetada para ser flexível, a tecnologia utiliza componentes padrões de interfaces gráficas e conceitos de camada WEB sem limitar os desenvolvedores a uma linguagem particular de marcação, protocolo ou dispositivo cliente. Os componentes de interfaces gráficas encapsulam a funcionalidade do componente, não a apresentação visual, habilitando-os para serem apresentados para vários dispositivos diferentes.

Tendo a facilidade de uso como meta primária, a arquitetura claramente define a separação entre lógica da aplicação e apresentação enquanto permite conectar facilmente a

camada de apresentação com o código da aplicação através do processamento de eventos. Diferentemente das tecnologias anteriores, o JSF recupera o formulário enviado pelo cliente através de componentes reusáveis de interfaces gráficas, como mostra a figura 4.5 [Husted 2004*a*]:

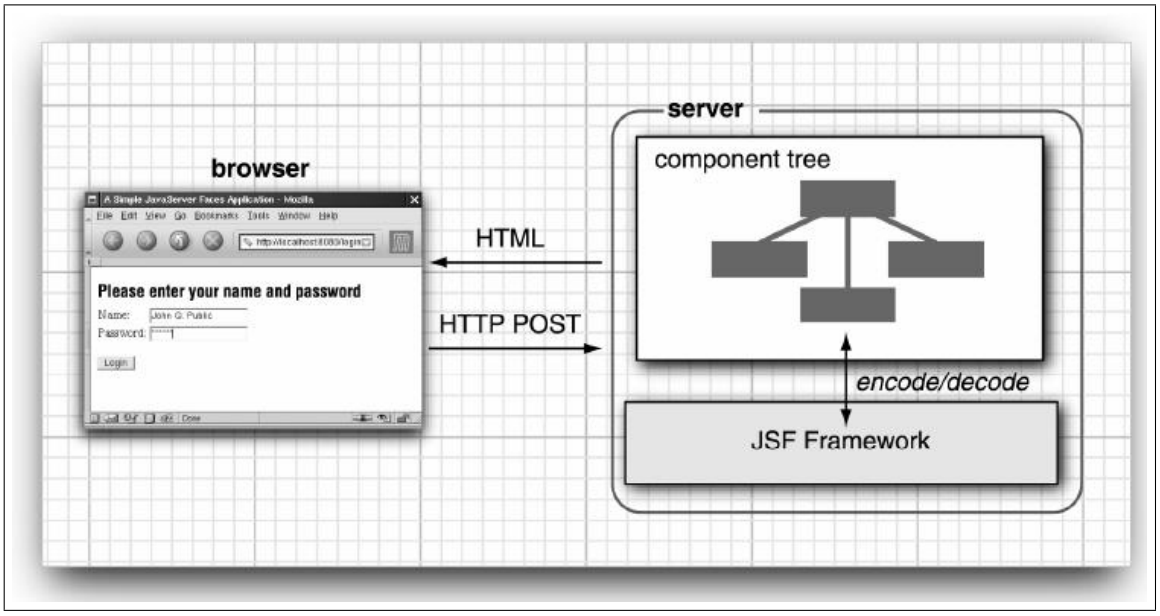

Figura 4.5: Fase de Restauração de Visualização do JSF

A fase de restauração da visualização, constrói o formulário enviado pelo cliente em uma árvore de componentes que é utilizada para o restante do processamento. O processamento de uma solicitação JSF passa por diversas fases, como ilustrado na figura 4.6 [Husted 2004*a*]:

Uma solicitação depois de enviada para o servidor, passa primeiramente pela restauração da árvore de componentes, informa os valores digitados pelo usuário em cada componente correspondente, realiza as validações dos dados, atualiza os dados da camada de modelo, chama a lógica de negócio e, por fim, monta a resposta na linguagem de marcação utilizada (normalmente HTML). Durante todo esse clico é possível fazer processamento de eventos. Este processo é ilustrado pela figura 4.6 [Husted 2004*a*].

Os diversos serviços providos pela tecnologia a tornam extremamente atrativa para o uso em projetos. No entanto, é preciso avaliar como todos esses recursos se comportam em um ambiente submetido a sobrecarga, comparando com as tecnologias Struts e JSP. Essas são questões de interesse do trabalho, ou seja, quantificar o aumento de utilização de recursos para servir como base comparativa ao aumento da produtividade.

A literatura da área é forte em comparativos destes frameworks no tocante a métricas de engenharia de software, tais como: extensibilidade, modularidade, abstrações, dentre outros. [Eric Jendrock 2006] No entanto, poucos artigos comparam o desempenho destas

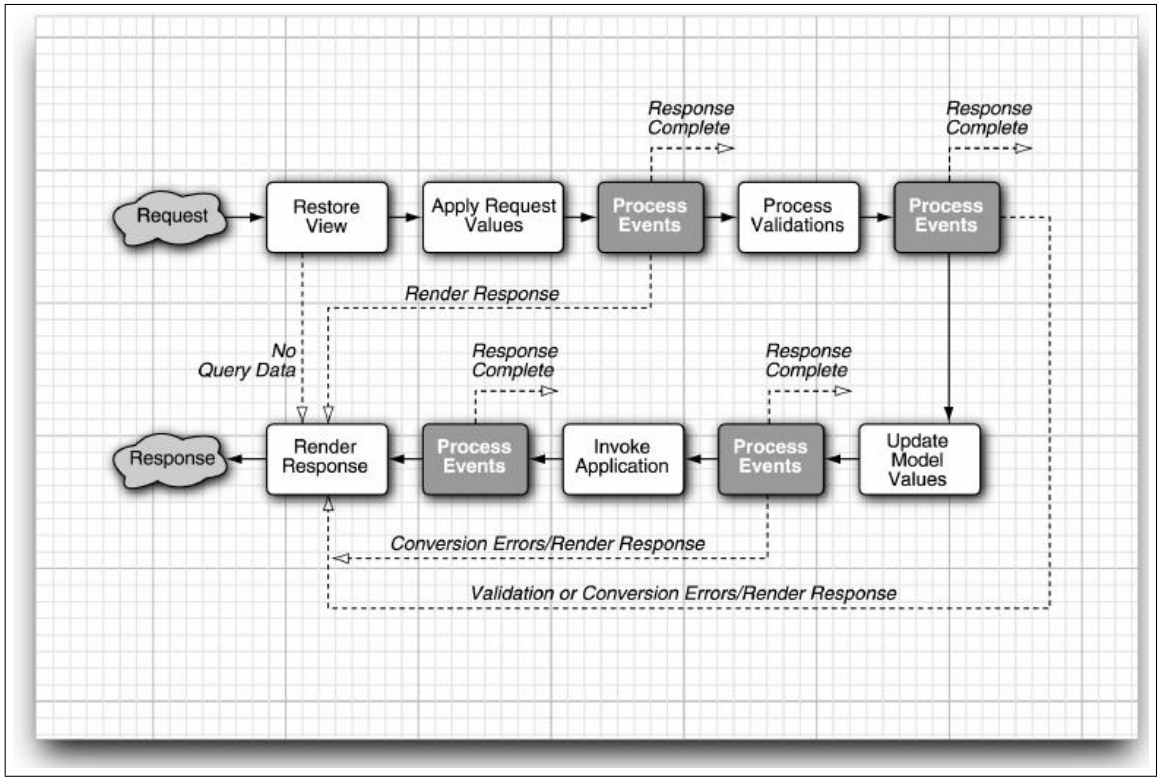

Figura 4.6: Ciclo de Vida de um processamento JSF

Tabela 4.1: Relatório de Funcionários - Teste da Camada de Visualização.

Id	*CPF*	*Nome*	*Endereo*	*Empresa*
1	CPF Gerado	Nome Aleatório	Endereço Aleatório	Empresa Aleatória
2	CPF Gerado	Nome Aleatório	Endereço Aleatório	Empresa Aleatória
100	CPF Gerado	Nome Aleatório	Endereço Aleatório	Empresa Aleatória

tecnologias. Há de se esperar que as tecnologias tenham o desempenho decrescente com relação a sofistação dos seus serviços. Nesta linha, é de se concluir que Servlets, JSP, Struts e JavaServer Faces possuem desempenho em ordem decrescente. Entretanto, é necessário avaliar a medida desse decrescimento, sendo esse um dos principais focos do trabalho. Para se obter tais resultados, um mesmo problema será desenvolvido utilizando as três diferentes tecnologias (JSP, Struts e JSF). Estas aplicações serão submetidas a grandes demandas e os resultados serão coletados. A seção a seguir analisa a aplicação desenvolvida.

4.2.3 Aplicação para comparação de Desempenho

O teste de uma aplicação Web deve caracterizar como ela se comporta na geração dos dados da resposta, por exemplo, um relatório. Para simular N usuários submetendo seguidamente as solicitações, será usada a ferramenta de teste (apresentada no capítulo 5) enviando os dados através de uma requisição *HTTP GET* conforme figura 4.7.

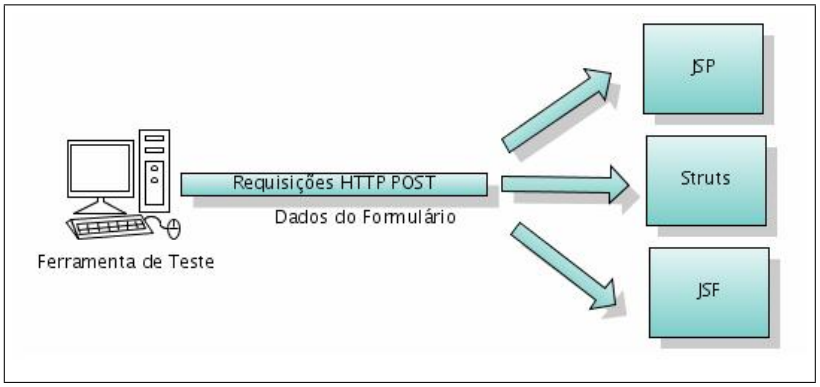

Figura 4.7: Modelo de testes para as tecnologias da camada WEB.

Para analisar a geração de conteúdo através de cada uma das tecnologias será gerada uma lista de 100 funcionários, de acordo com a tabela 4.1.

Considerando-se que cada uma das três tecnologias tem sua metodologia particular de gerar listas dinâmicas, para simular um ambiente real, os dados serão gerados dinamicamente a cada requisição. Ou seja, em cada evento serão seguidos os seguintes passos:

1. Criação, de maneira uniforme, de 100 objetos de funcionários;

2. Montagem da lista para os 100 objetos criados no passo anterior usando a particularidade de cada tecnologia;

Ao final do teste, será obtido um parâmetro comparativo de desempenho (descrito no capítulo 5) na geração de conteúdo em cada uma das tecnologias de interesse, possibilitando comparar os resultados e concluir sobre aquela que possui melhor desempenho.

4.3 Camada de Negócio

A camada de apresentação trata dos aspectos de controle e apresentação dos dados inseridos e apresentados ao usuário. É a partir desta camada que os dados são captados e enviados à camada de negócio, que realiza o processamento desses dados através da manipulação da camada de persistência. Assim, é nesta camada que encontra-se a "inteligência"do processamento dos dados.

Em um modelo de três camadas, é possível representar a interação entre eles conforme ilustrado na figura 4.8.

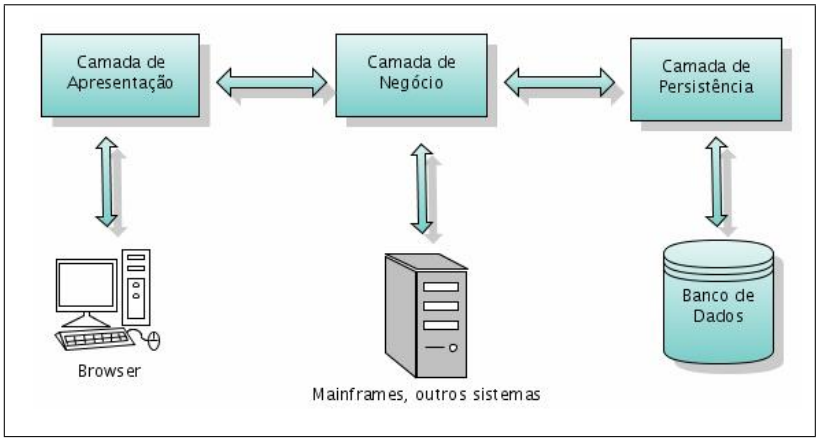

Figura 4.8: Interação entre as camadas de apresentação, persistência e a de negócio

A camada de negócio normalmente é ativada pela camada de apresentação e comunica-se com os dados e outros sistemas (*mainframes*, ERPs (*Enterprise Resource Planning*), dentre outros). Devido a esta característica, esta camada depende de vários serviços para realizar suas funcionalidades com robustez, tais como: transações, gerenciamento de recursos, *multithreading*, segurança, dentre outros.

Vários softwares surgiram para atender às necessidades de serviços desta camada, especialmente o serviço transacional. Os *TP Monitors* (*Transaction Processor Monitors*) [Browne n.d.] foram os primeiros nesta linha, eles permitiam que um código em execução tivesse a transação automaticamente gerenciada. O maior representante destes softwares foram os produtos para *mainframes* na linguagem COBOL (IBM CICS, por exemplo) [Browne n.d.], que são até hoje largamente utilizados em processamento corporativo.

Com o advento da tecnologia orientada a objetos, as linguagens de terceira geração, como o COBOL, vêm perdendo espaço para os servidores de aplicações: softwares que fornecem serviços para as aplicações orientada a objetos.

4.3.1 Enterprise JavaBeans

Dentre as tecnologias para prover serviços para a camada de negócio destaca-se a EJB - Enterprise JavaBeans [Rima Patel Sriganesh 2006]. Esta tecnologia permite desenvolver componentes que são auto-gerenciáveis em vários aspectos: transações, segurança, recursos, dentre outros. Tem largo uso em corporações e substitui o código mainframe executando em TP Monitors.

Os componentes EJB são distribuídos, ou seja, permitem que sejam invocados por softwares que executam em outros espaços de endereçamento ou até em outros nós da rede. Essa característica possibilita que não apenas uma camada WEB possa invocar a camada de negócio, mas que outras aplicações também o façam diretamente. Assim, uma aplicação para Desktop, por exemplo, poderia chamar o mesmo componente EJB que é invocado pela camada de visualização da aplicação WEB, gerando uma centralização do código e da manutenção, como mostrado na figura 4.9 [Eric Jendrock 2006].

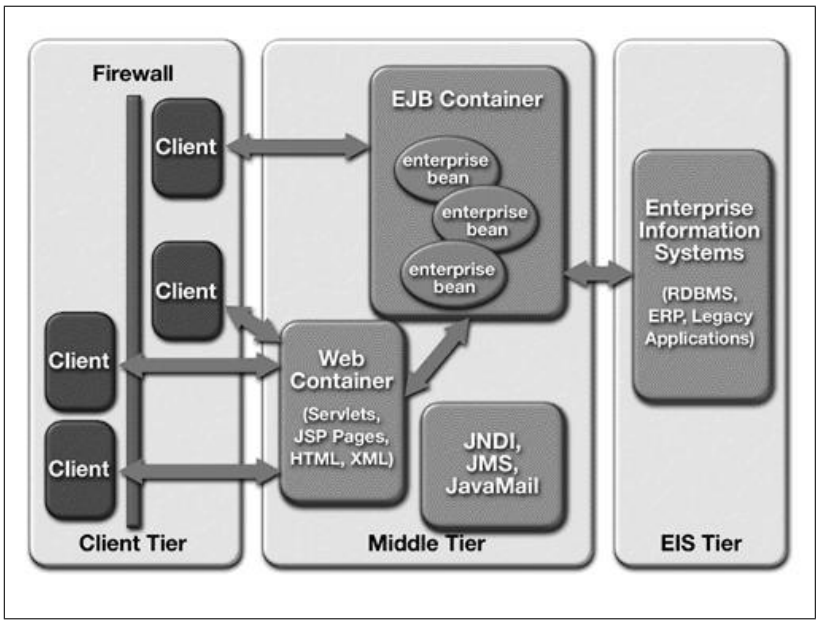

Figura 4.9: Interação entre EJB e diversos clientes

As chamadas dos componentes EJB podem ser realizadas inclusive por outros sistemas desenvolvidos na linguagem Java. Assim, há uma invocação da lógica de negócio de um sistema a partir de um outro sistema. Esta característica é denominada de **interoperabilidade de aplicações**.

A interoperabilidade de serviços providos pela camada de negócio vem crescendo fortemente na indústria, principalmente em interações B2B (*Business-to-Business*) [Cerami 2002*a*]. Isto possibilita, por exemplo, uma empresa solicitar serviços providos por outras empresas. Esse modelo de consumir e fornecer serviços deu origem a um novo modelo de arquitetura denominado SOA (*Service-Oriented Architecture*) [Cerami 2002*a*].

4.3.2 Web Services

Na arquitetura orientada a serviços a mesma implementação nunca está replicada em dois sistemas, o que implica em, um sistema invocar operações do outro quando delas necessita. Porém, a grande barreira para a maior integração entre corporações é o fator tecnológico, o que leva a se questionar como garantir que as tecnologias dos dois sistemas sejam compatíveis. Nesta vertente, padrões de protocolos binários não trouxeram a solução esperada, como o CORBA [Browne n.d.]. Padrões que utilizavam apenas texto foram os mais aceitos e largamente utilizados. Dentre eles, destacam-se os documentos baseados em XML (*eXtensible Markup Language*) [Newcomer 2002] a partir do qual surgiu a tecnologia para possibilitar invocação de operações inter-sistemas: **o Web Service**.

Web Services [Newcomer 2002] são aplicações XML mapeadas em programas, objetos, banco de dados ou um conjunto de funções de negócio. Usando um documento XML criado em forma de uma mensagem, um programa envia uma requisição para um Web Service através da rede, e, opcionalmente, recebe uma resposta, também na forma de um documento XML. Os padrões de Web Services definem o formato das mensagens, especificando a interface que a mensagem deve ser enviada, descrevendo convenções para o mapeamento do conteúdo em termos de entrada e saída, e definem mecanismos para publicação e descoberta de serviços.

Os Enterprise JavaBeans podem ser chamados através de Web Services como definido na especificação J2EE 1.4. Assim, são mantidas as características da transacionalidade e permite-se que qualquer sistema faça essa invocação.

4.3.3 Descrição das chamadas com EJB e Web Services

A análise de desempenho da camada de negócio será feita com base em três soluções de codificação da mesma funcionalidade. O problema em questão deve ser independente de outras camadas para não prover mais um fator de imprevisibilidade. Ou seja, não haverá comunicação com banco de dados ou nenhum outro recurso. Assim como no teste realizado com a camada de persistência, será utilizada uma Servlet que receberá as requisições da aplicação de teste e chamará uma das três soluções propostas. A figura 4.10 ilustra esta situação.

A ferramenta de testes executará o conjunto de requisições definidas para cada uma das três soluções. A primeira é um JavaBean que não utiliza qualquer serviço, exceto pela simples classe que contém o processamento da lógica de negócio invocada pela Servlet. A segunda solução de Web Service é a mesma implementação da primeira solução, exceto que agora é invocada através de uma mensagem XML. A terceira solução (EJB) também representa a mesma da primeira solução, exceto por utilizar o serviço de transação provido

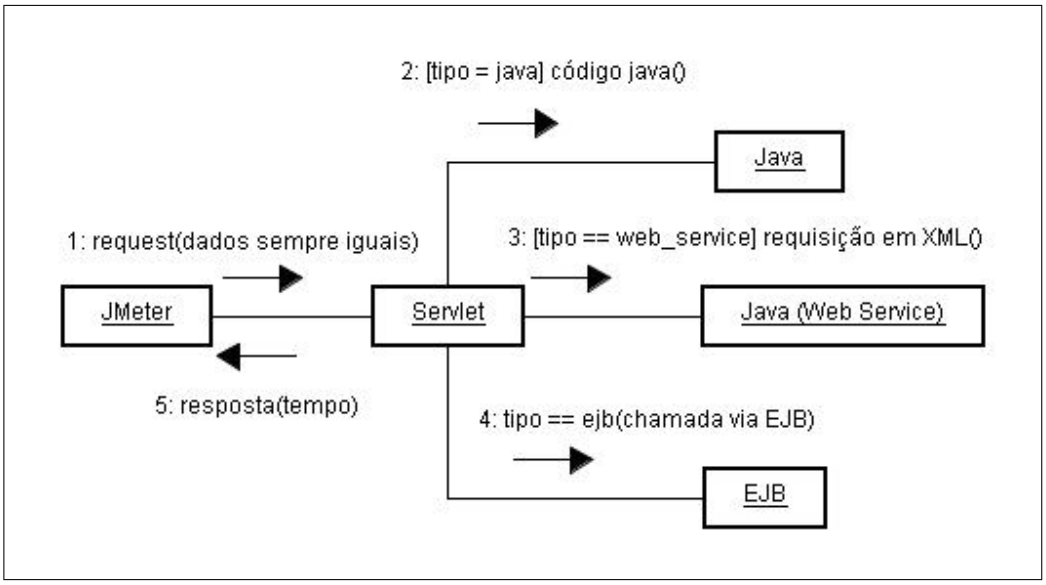

Figura 4.10: Diagrama de colaboração do teste da camada de negócio

pela tecnologia EJB.

No teste Java será considerando o modelo de referência por ser essa implementação a maneira mais simples e, teoricamente, o de melhor desempenho. Considerando a figura 4.10, a mensagem 2 não possui nenhum intermediário, é realizado direto e nativamente na máquina virtual, sem intermediários. Já a mensagem 3 e 4 são interceptadas por serviços de middleware, para prover as funcionalidades desejadas.

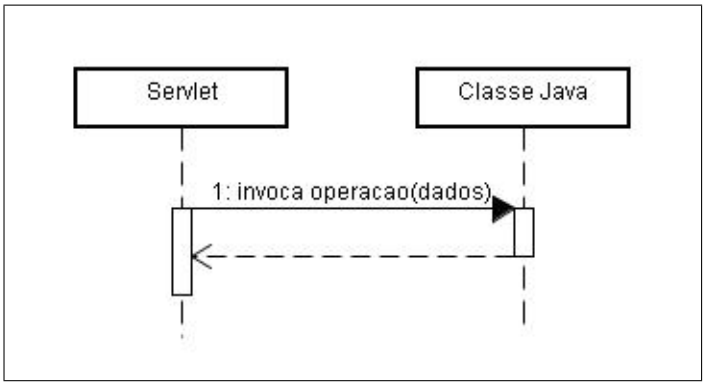

Figura 4.11: Invocação de uma classe Java

O teste do Web Service, representado pela mensagem 3 na figura 4.10, necessita passar

pelos passos ilustrados na figura 4.12 para completar sua solicitação.

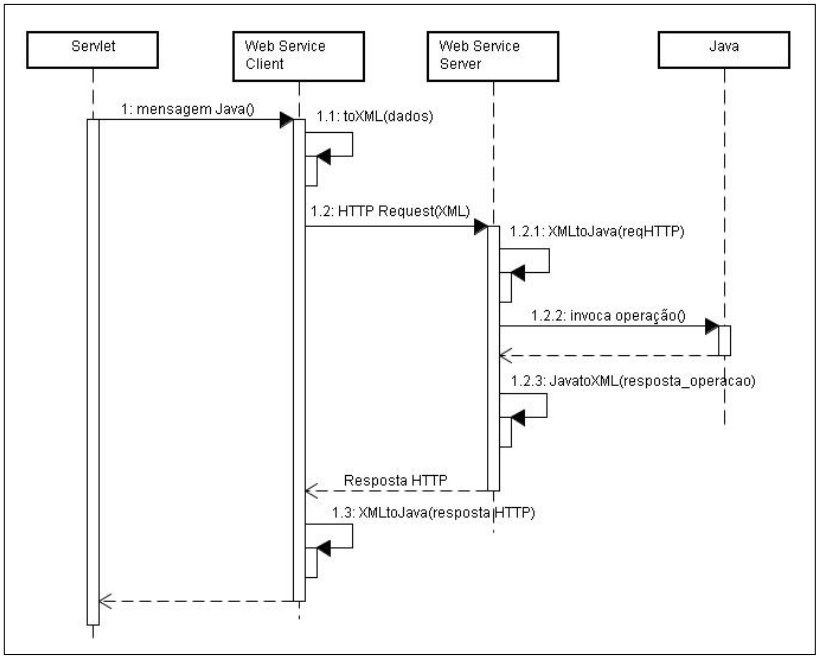

Figura 4.12: Chamada de um Web Services

A utilização de Web Services implica em transformar os dados da requisição (que são recebidas da aplicação de teste) e transforma-los em XML (1.1) . Os dados transformados em XML são enviados para um serviço Web através de uma requisição HTTP (1.2). Uma outra classe, que recebe as requisições HTTP, transforma o conteúdo novamente para Java (1.2.1) e invoca a operação desejada (1.2.2). O resultado da operação é novamente transformado em uma resposta em XML (1.2.3) e é retornado na solicitação HTTP. Os dados em XML são novamente transformados em Java (1.3) e enviados para a Servlet.

Com esse mecanismo, de utilização do documento XML como um intermediário, é possível obter a característica da independência da tecnologia. Porém, comparado com a solução Java tradicional, há o custo da transformação de Java em XML (*Marshal*) e a transformação de XML em Java (*Unmarshal*) [Eric Jendrock 2006]. Esse custo, o tempo adicional da infra-estrutura dos serviços de Web Services, é o que será medido no capítulo 5.

Após a requisição ser recebida pela Servlet, ela invocará o serviço que estará na mesma máquina, não tendo assim um custo de comunicação. O custo computado é a utilização dos protocolos de Web Services.

A outra tecnologia em questão é a EJB (*Enterprise JavaBeans*). Semelhante a Web Services, EJB tem intermediários que realizam a interceptação da chamada provendo serviços de middleware. A Figura 4.13 ilustra o fluxo da invocação EJB.

Tratando-se de uma tecnologia de objetos distribuídos, um componente EJB sempre

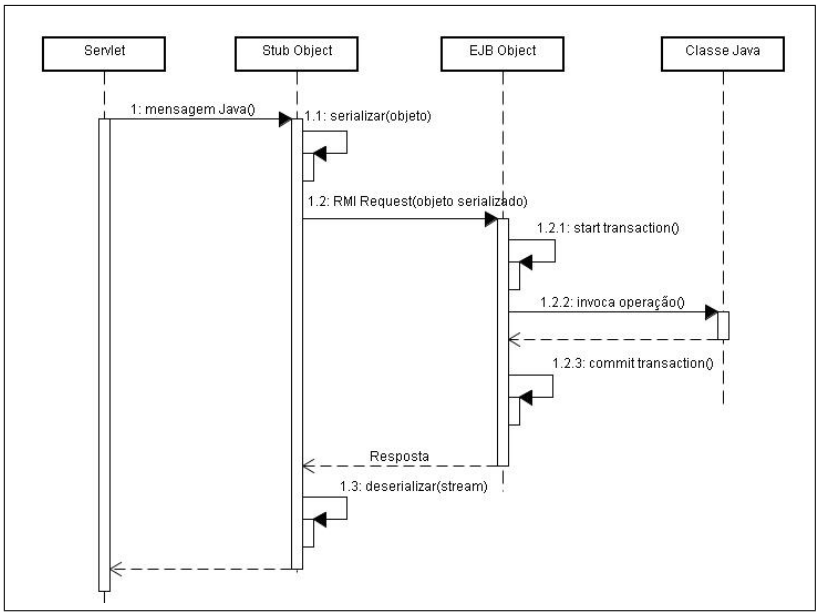

Figura 4.13: Fluxo de chamada de um EJB

possui um objeto representante no cliente e outro no servidor. Neste caso, assim como no teste anterior, ambos estão na mesma máquina.

A Servlet repassa os dados do objeto para o Stub [Rima Patel Sriganesh 2006] (o representante do lado do cliente), este serializa [1] o objeto (1.1) e o envia para o objeto representante no servidor: o EJB Object [Rima Patel Sriganesh 2006]. Esta classe, de posse dos dados enviados pelo Stub, inicia uma transação (1.2.1) e invoca a operação desejada (1.2.2). Ao final da operação, confirma a transação (1.2.3) e envia a resposta em um objeto serializado para a Servlet.

4.3.4 Aplicação para comparação de Desempenho

O propósito do teste não é comparar o desempenho da solução EJB e a solução Web Services, já que em muitas arquiteturas o Web Service é utilizado para chamar os EJBs, fazendo uma junção dos fluxos anteriores. A tecnologia EJB foca-se em fornecer serviços para lógica de negócio e a tecnologia Web Services foca-se em fornecer um mecanismo portável para ser invocada por outras tecnologias e dispositivos.

A questão chave a ser respondida é: considerando um código Java que realiza uma determinada função, qual o impacto no desempenho se este código fosse invocado através de Web Services e através de EJB. Considerando T1 o incremento de tempo do Web Services e T2 o incremente de tempo do EJB, poderemos concluir que a utilização das

[1] Serialização é o processo de copiar o objeto para arquivos, sockets de rede. É intensamente usado para copiar objetos em sistemas distribuídos

duas tecnologias levaria um incremento de tempo T3 = T1 + T2. Para medir T1 e T2 será realizado um teste utilizando as três tecnologias, que terá seus resultados apresentados no próximo capítulo.

Para testar o desempenho das três soluções, uma aplicação que simula o processamento de uma lógica de negócio foi desenvolvida. O problema consiste em encontrar, dado um conjunto de clientes, aqueles que são devedores. Como não deve haver acesso ao banco de dados, pelos motivos expostos anteriormente, um cliente é definido como devedor se o identificador dele é divisível por três.

Visando enviar um conjunto significativo de dados que passarão pelas classes intermediárias, cada requisição envia 100 clientes (sempre os mesmos) para serem consultados. A classe cliente é exibida na listagem a seguir.

```
public class Cliente {

    private int id;
    private String nome;
    private Long cpf;
    private String endereco;
    // métodos sets e gets
}
```

A classe que busca os clientes devedores recebe uma lista de clientes e apenas percorre-os, retornando aqueles que têm seu *id* divisível por três. Essa lógica é apenas para realizar alguma computação.

```
import java.util.ArrayList; import java.util.Collection;

public class Devedores {
    public Collection<Cliente> processaDevedores(Collection<Cliente> clientes) {
        ArrayList<Cliente> devedores = new ArrayList<Cliente>();
        for (Cliente c : clientes) {
            if (c.getId() % 3 == 0) {
                devedores.add(c);
            }
        }
        return devedores;
    }
}
```

A maneira de utilizar essa funcionalidade através da tecnologia EJB como também invocá-la através de Web Services foge do foco do trabalho. Assim, não será descrito os passos para tornar o código acima em um EJB ou em um Web Services, para este

propósito, consulte as referências [Rima Patel Sriganesh 2006] e [David Chappell 2002]. O foco será em analisar os resultados obtidos a seguir utilizando a ferramenta de testes nos capítulos subseqüentes. Os códigos fontes dos testes estão em anexo no trabalho.

Este capítulo descreveu as camadas pelo qual as aplicações são subdivididas e explorou tecnologias e soluções inerentes de cada camada. Agora, conhecendo-se as características de cada tecnologia, o foco volta-se a utilizar as aplicações de testes desenvolvidas e descritas neste capítulo para obter os dados comparativos de desempenho.

Capítulo 5

Testes e Análise de Desempenho

Apresentadas as questões relativas ao desempenho do ambiente de execução das tecnologias que são usadas nas camadas que modelam a aplicação, é chegado o momento de se realizar um comparativo de desempenho entre algumas tecnologias de cada camada. Antes de iniciar-se esta abordagem, uma descrição de como os testes serão realizados e a metodologia de análise dos resultados.

A intenção é simular um ambiente estressado, com várias requisições de oriundas de um grande número de usuários. Para este propósito, será usada a ferramenta do grupo Apache JMeter (`jakarta.apache.org/jmeter/`).

O JMeter permite simular uma certa quantidade de usuários, cada um realizando N requisições seqüencialmente. Para cada usuário, a ferramenta inicia uma nova *thread* que age independentemente das outras. Por exemplo, considerando-se um único usuário realizando 10 requisições seguidamente, a próxima requisição só é realizada depois da chegada da anterior. Ao longo do tempo, os eventos associados com o envio de requisições se comportam como ilustrado na figura 5.1. O intervalo de tempo entre envio de requisições consecutivas é denominado *round trip time*, RTT.

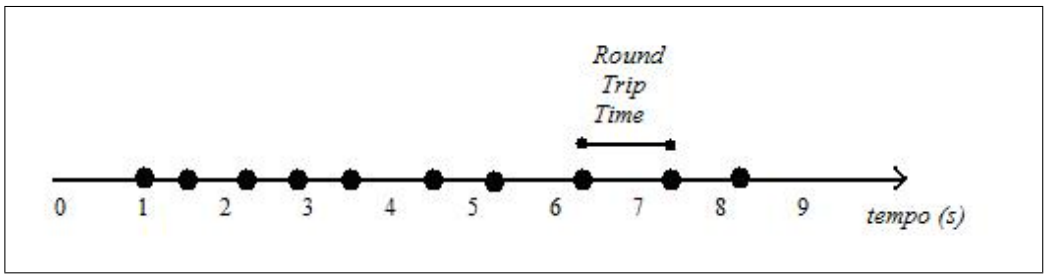

Figura 5.1: Computação de um Round Trip Time

Realizando os testes somente com um usuário, ou seja, uma *thread*, o processo é seqüencial e não sobrecarrega o servidor. Para saturá-lo, o número de usuários configurados deve ser grande o suficiente para permitir uma demanda intensiva dos recursos do servidor (processador, memória, disco, etc). É nestas condições que observa-se o maior

impacto da coleta de lixo e a escalabilidade da aplicação ao crescimento de usuários. Considerando vários usuários, resulta em várias retas da figura 5.1 em paralelo, *threads* agindo independentemente solicitando requisições e captando os *round trip times*. Ao final do experimento, considerando um teste com 50 usuários e 2000 requisições cada um, temos 100 mil amostras da variável aleatório RTT.

O problema que se apresenta inicialmente á a caracterização da estatística relacionada ao processo aleatório associado à variável aleatório RTT e assim obter parâmetros tais como média e desvio padrão que servirão de referência para os resultados a serem fornecidos pela ferramenta de testes, o JMeter.

A primeira vista, considerando a modelagem do experimento, o processo aleatório associado a RTT parece ter a natureza *poissoniana*, pois se asemelha ao funcionamento de um protocolo de rede do tipo *stop-and-wait* [Tanembaum 2003]. Entretanto, uma análise mais cuidadosa permite concluir que além da parcela de tempo relativa ao funcionamento do protocolo de rede, RTT inclui outros tempo associados ao processamento em cada um dos servidores (ferramenta de testes, de banco de dados e da aplicação sob análise) que podem modificar a natureza de RTT como um processo de *Poisson*.

Para se investigar a natureza estatística de RTT foram realizados testes preliminares para obtenção de sua função de densidade de probabilidade e compará-la com funções bem conhecidas no estudo de processos estocásticos tais como *Poisson, Gaussiano e Student*. Os resultados obtidos estão na figura 5.2 e levam ao afastamento e conclusão de que não se dispõe de um modelo teórico para o processo estocástico associado a RTT. Apesar disso, é possível obter a partir de sua densidade de probabilidade a média e mediana, que poderão servir de referência para a análise dos dados a serem fornecidos pela ferramenta JMeter.

5.1 Infra-estrutura para realização dos Testes

Os experimentos foram realizados em um servidor Pentium 4 *Xeon Hyper-threading* com um processador de 3 GHz e 1 GB de memória para abrigar a aplicação a ser analisada. Já o gerador de requisições, o JMeter, será executado em uma máquina similar, conectado ao servidor através de uma porta *Gigabit Ethernet* de um *switch*. O servidor de banco de dados, utilizados nos experimentos da camada de persistência, tem também configuração semelhante ao servidor das aplicações. A figura 5.3 ilustra a arquitetura utilizada na realização dos testes.

O gerador de solicitações é uma aplicação capaz de gerar uma quantidade predefinida de requisições, através da configuração dos seguintes parâmetros:

- Quantidade de usuários (*threads*)
- Tempo de inicio da *thread*
- Quantidade de requisições

O número de *threads* determina a quantidade de usuários que demandam a aplicação de forma concorrente. Cada *thread* realiza o teste independente uma da outra e permite um paralelismo na geração da amostra. O tempo de início da *thread* controla a fatia de tempo

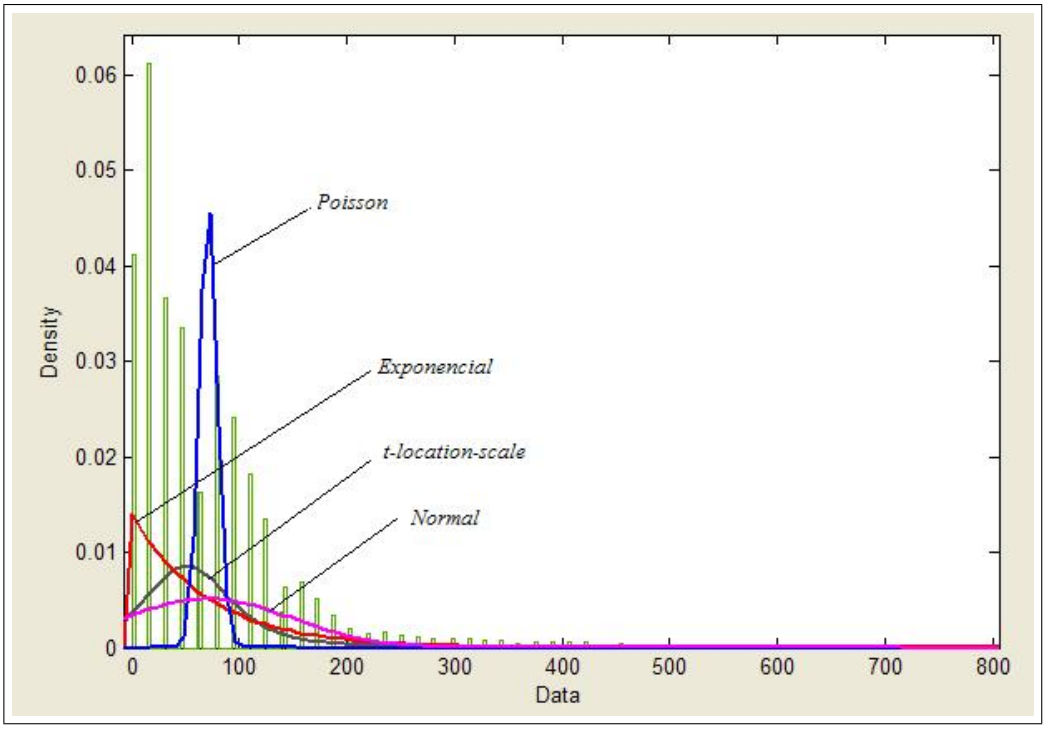

Figura 5.2: Comparação entre a distribuição obtida e as teóricas conhecidas

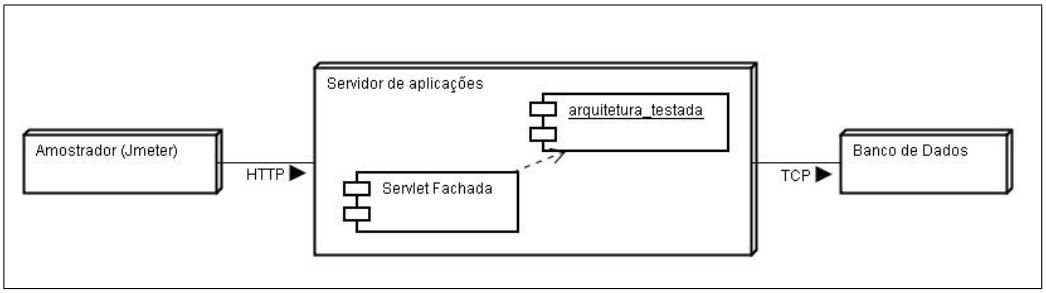

Figura 5.3: Máquinas envolvidas no teste

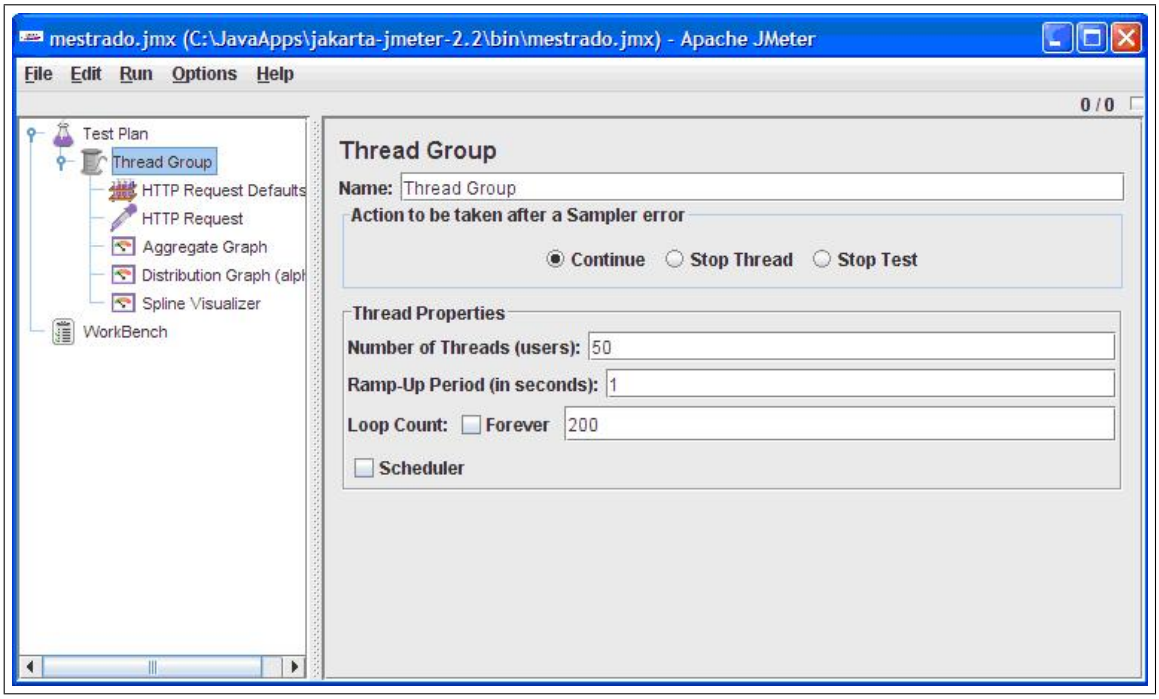

Figura 5.4: Tela da aplicação de Teste - JMeter

que o JMeter deve aguardar para iniciar cada *thread* consecutivamente. Por exemplo, se estiver como 1s, como na figura 5.4, a cada 1s uma nova *thread* será iniciada. Esse tempo é usado para permitir uma graduação na sobrecarga do teste e não sobrecarregar o servidor logo de imediato. O total de requisições (*loop count*) informa quantas requisições que cada *thread* vai gerar.

Sendo U a quantidade de *threads*(usuários) e N o número de amostras geradas por cada uma, todas as threads competem pela interface de rede para enviar mensagens e recebe suas respostas. A tarefa de cada *thread* pode ser sintetizada pelo algoritmo a seguir:

```
while ( qtd_amostras < U ) {
    t0 = hora_atual;
    gerar_amostra;
    aguarda_resultado;
    t1 = tempo_resposta
    rtt = t1 - t0; // round trip time
    armazena rtt;
}
```

Para cada requisição tem-se além do tempo de resposta o status do sucesso. Devido ao uso do protocolo HTTP, não há possibilidade de perda de mensagens, pelo fato do TCP

possuir garantia de entrega. Para cada resposta, o código *HTTP RESPONSE CODE* é utilizado para determinar o sucesso ou falha do evento. Caso, por algum motivo não previsto, a requisição não seja realizada com sucesso, o response code será 500 (*INTERNAL SERVER ERROR*). Dessa forma, o erro é controlado, caso alguma requisição gere erro, ela pode ser descartada ou o teste refeito.

5.2 Resultados dos Experimentos

O capítulo anterior descreveu as tecnologias que seriam testadas para cada camada. Cada tecnologia será submetida a 50 usuários simultâneos cada um realizando 2000 requisições. Ou seja, para cada experimento são realizadas 100 mil requisições. Este número mostrou-se suficiente para extressar o servidor.

Como foi visto no capítulo 2, quanto maior a quantidade de coletas de lixo mais intrusiva ela é no desempenho da aplicação. Por este motivo, além da análise dos tempos de resposta, o uso de memória demandado por cada experimento também será obtido para permitir conhecer qual tecnologia consome mais recursos de armazenamento. Para coletar esses dados, o servidor deve ser inicializado com a opção *-verbose:gc*, que habilita a máquina virtual Java imprimir cada coleta que ela realiza, informando a quantidade de bytes liberados.

Um exemplo de saída com a opção *-verbose:gc* ativada seria ([GC 73969K->62134K(129792K), 0.0060170 secs]) correspondendo a, respectivamente:

- Memória ocupada no momento da coleta
- Memória ocupada após a coleta
- Tamanho total do Heap
- Tempo dispendido na coleta

Durante a execução dos testes estas saídas serão todas capturadas para posterior análise do uso de memória. O *heap* foi configurado para um máximo de 1 GB e mínimo de 128 Mb.

5.2.1 Resultados da Camada de Persistência

Pelas diretrizes estabelecidas no capítulo 4, os testes da camada de persistência compararam a aplicação desenvolvida com JDBC, uma utilizando o padrão de projeto DAO e outra com Hibernate. O resultado esperado era que o desempenho fosse decrescente nessa ordem: JDBC, DAO e Hibernate. Os gráficos a seguir ilustram os resultados obtidos.

A Figura 5.5 ilustra a distribuição dos tempos de resposta submetidas às requisições. Note que uma parcela significativa do tempo de resposta é próxima a zero. O uso do *heap* neste teste não se mostrou muito intenso, como apresentado nas figuras 5.6 e 5.7. O tempo médio de resposta foi 22,28 milisegundos.

O teste com o padrão de projeto DAO é apenas uma pequena variação da aplicação com JDBC de forma a ser mais orientada a objetos. Por esse motivo, esperava-se um uso maior do *heap* e um desempenho um pouco menor pelo custo de criação de mais objetos.

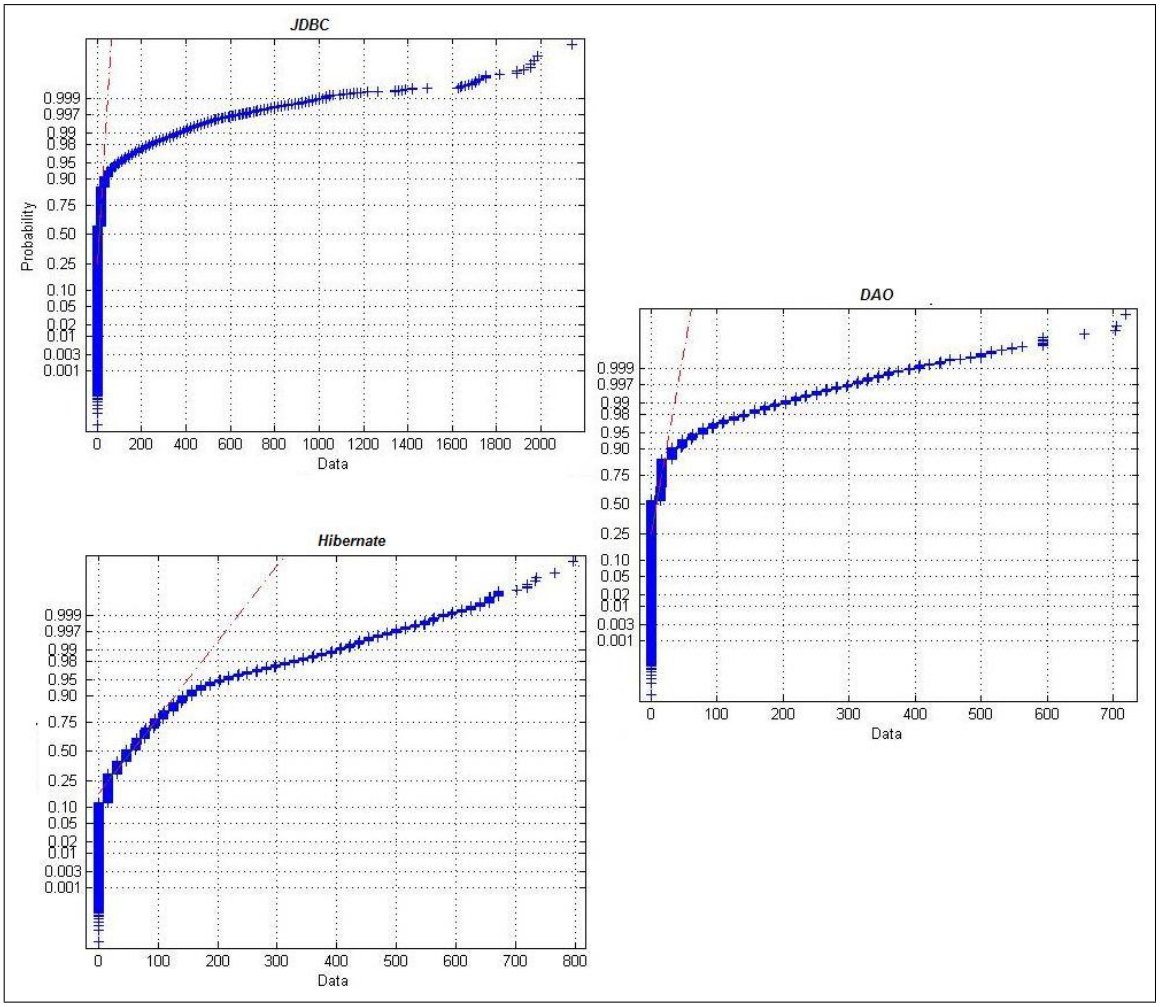

Figura 5.5: Distribuição de Probabilidade do tempo de Resposta do teste de JDBC

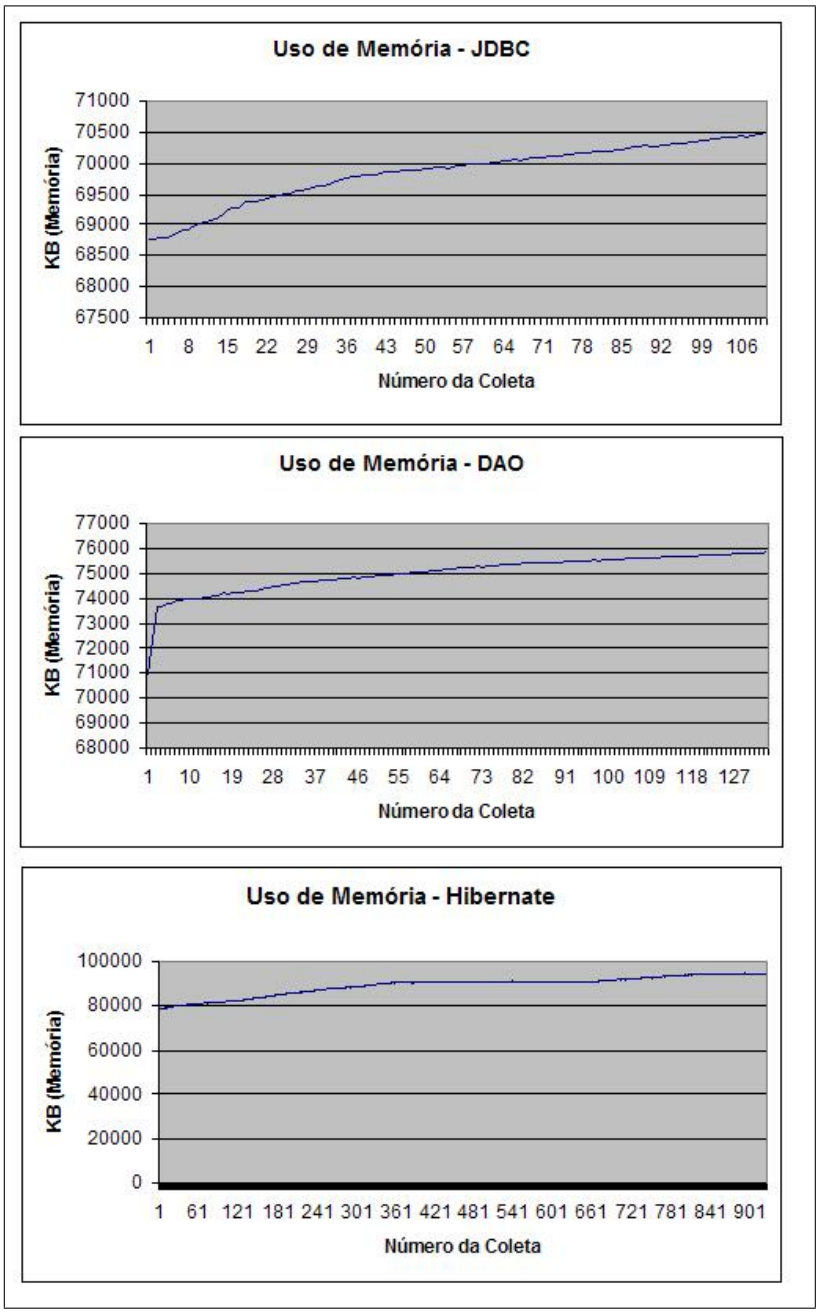

Figura 5.6: Uso de Memória das três tecnologias da camada de mapeamento

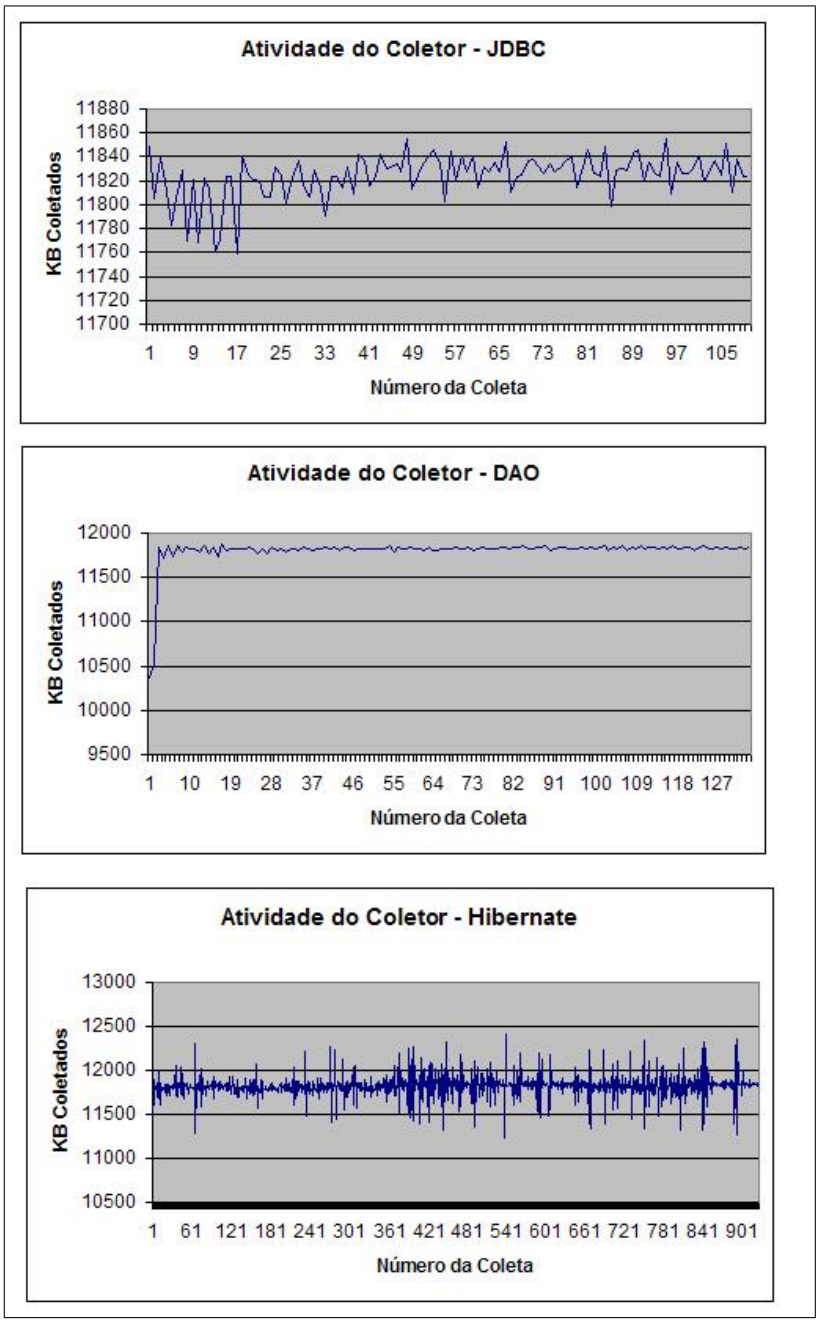

Figura 5.7: Atividade do Coletor de Lixo no experimento das três tecnologias da camada de mapeamento

Tabela 5.1: Quadro Comparativo do Desempenho da Camada de Persistência

Solução	Média	Mediana	Desvio Padrão	Tempo(min)	Thoughput	Memória
JDBC	22,26	0	79,92	1,86	892(req/s)	1,2 GB
DAO	16,71	0	38,68	1,75	949,71(req/s)	1,5 GB
Hibernate	71,17	47	77,99	3,25	511(req/s)	10,45 GB

A figura 5.5 ilustra a distribuição da resposta deste teste. A média do tempo de resposta foi, surpreendentemente, menor que o do anterior, 16,17 milisegundos. A explicação para tal fato pode ser encontrada no fato da coleta de lixo ser menos custosa do que a destruição de quadros (*frames*) de pilha. O teste com JDBC usa intensamente dados na pilha já o DAO mantém apenas a referência ao objeto na pilha e põe todos os dados recuperados no *heap*. O uso de memória deste teste pode ser ilustrado na figura 5.6 e 5.7.

É de se esperar que o Hibernate possua um desempenho inferior ao JDBC e ao DAO. No entanto, a quantidade de recursos e facilidade de programação é incomparável. Apesar de ter levado um tempo em média 3 vezes maior que o JDBC e 4 vezes maior que o DAO, o Hibernate ainda sim mostrou-se extremamente eficiente com um tempo médio de resposta de 71,17 milisegundos. O uso de memória (figura 5.6) também foi maior, realizando 913 coletas de lixo enquanto JDBC e DAO realizaram 129 e 151 respectivamente. A atividade do coletor foi também mais intensa e ilustrada na figura 5.7. A tabela 5.1 sintetiza a situação dos 3 testes.

5.2.2 Resultados da Camada de Visualização

As tecnologias JSP, Struts e JSF surgiram cronologicamente nesta ordem. Ao passo que novos recursos eram demandados pelo mercado as tecnologias foram cada vez consumindo mais recursos para poder solucioná-los com qualidade. Esta afirmação é amparada pelos testes realizados com essas três soluções cujo resultado será exibido a seguir.

Conforme discutido no capítulo o teste é focado na geração de conteúdo. O mesmo conteúdo é gerado pelas três soluções, cada uma de sua maneira peculiar (consultar anexo para detalhes na implementação).

A Figura 5.8 ilustra a distribuição do tempo de resposta deste teste. Em média, o tempo de resposta foi 95,63 milisegundos. As figuras 5.9 e 5.10 ilustram o uso da memória e o trabalho do coletor, respectivamente.

O mesmo teste utilizando a tecnologia Struts resultou em um tempo médio de 160,14 milisegundos, 67% mais lento que a aplicação em JSP. Já o comportamento do crescimento da memória ocupada e da atividade do coletor mostraram-se semelhantes à aplicação em JSP, exceto pelo fato da quantidade de coletas ter sido maior: 1642 contra 465. A figura 5.8 ilustra a distribuição do tempo de resposta, a 5.9 e 5.10 ilustram o crescimento de memória e atividades do coletor, respectivamente.

De acordo com o capítulo 4 a tecnologia JSF é desenvolvida com uma nova ótica de agrupamento de componentes, tratamento de eventos, dentre outros. Ao passo que permite várias vantagens ao engenheiro de software consome mais recursos. O tempo

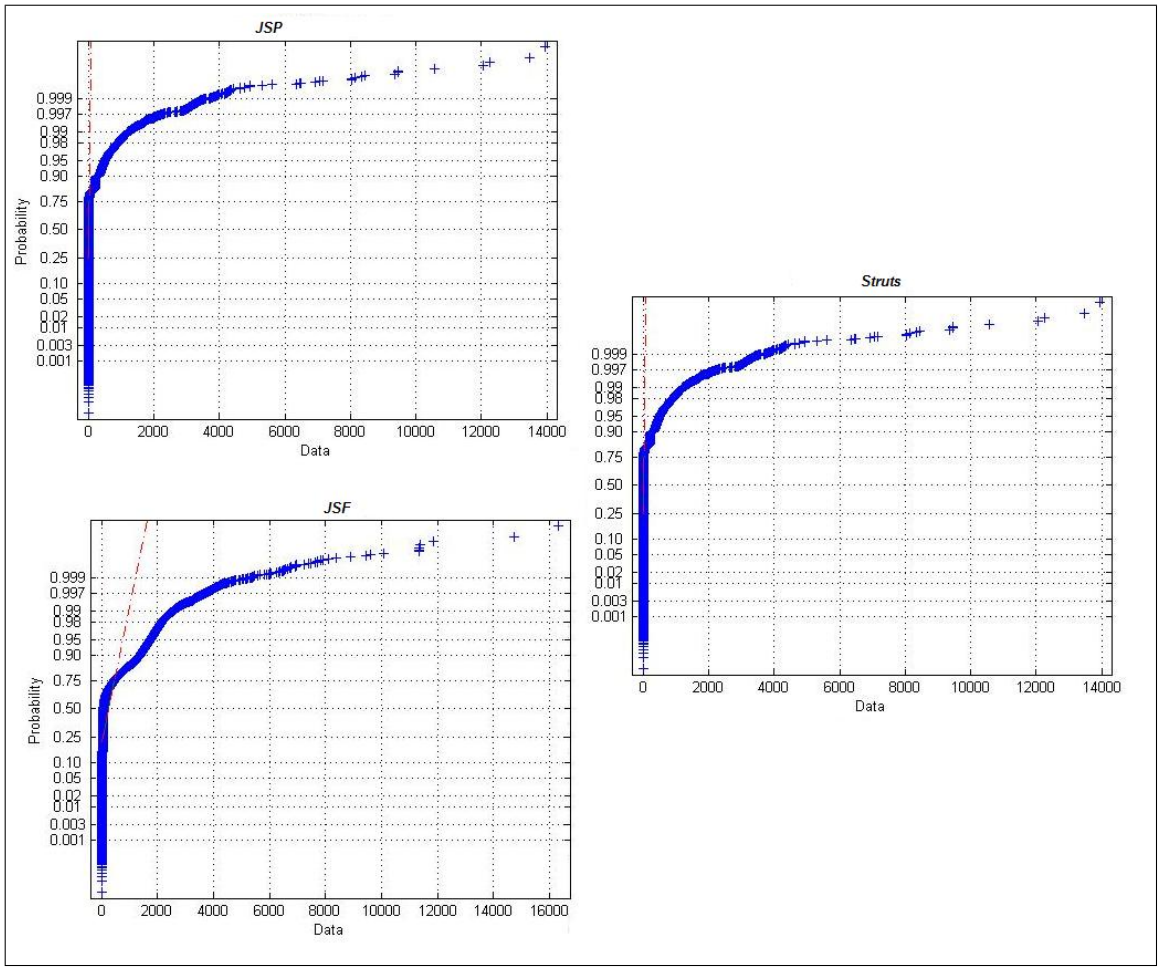

Figura 5.8: Distribuição de Probabilidade dos Experimentos da camada de visualização.

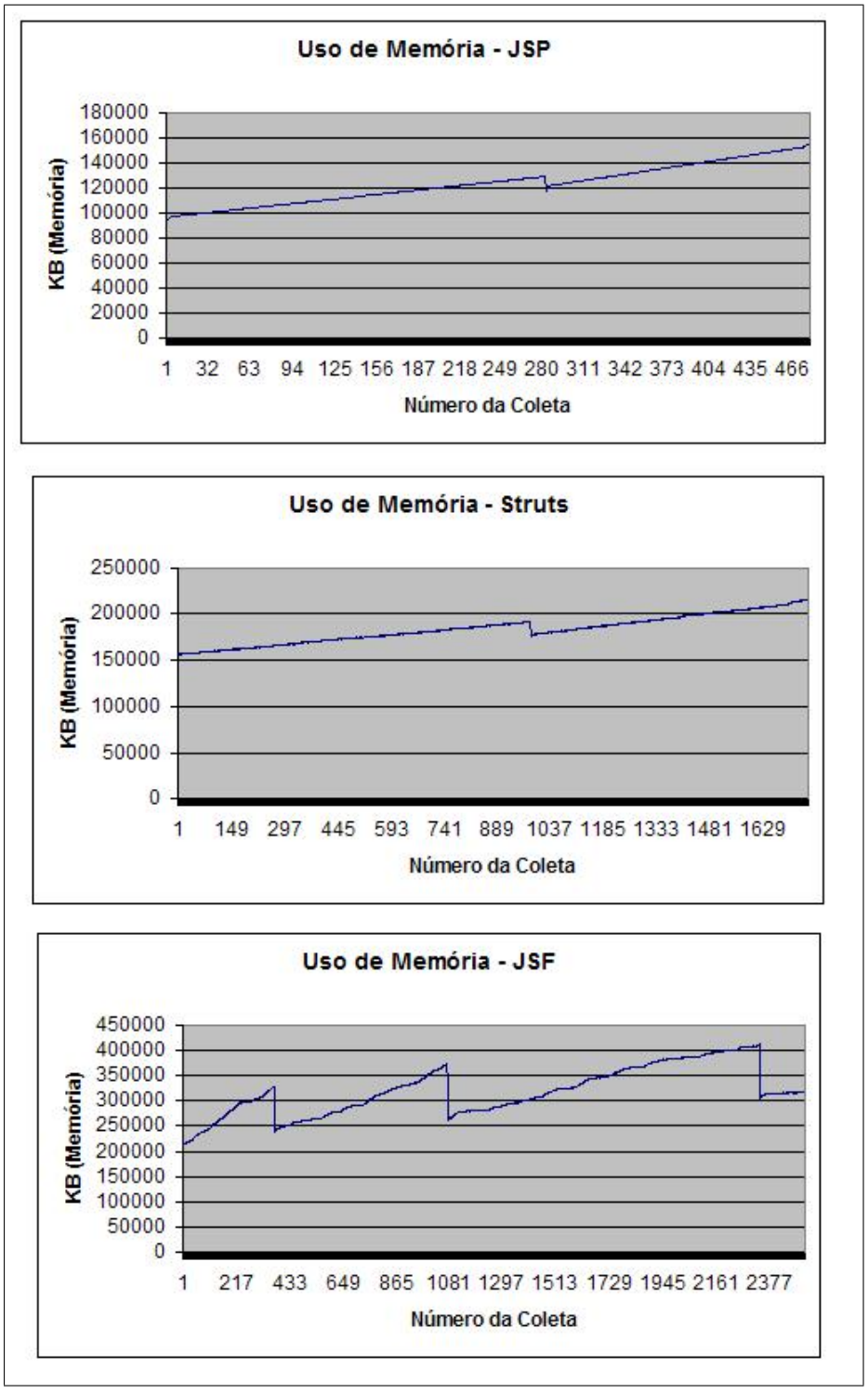

Figura 5.9: Uso de Memória dos experimentos da camada de visualização.

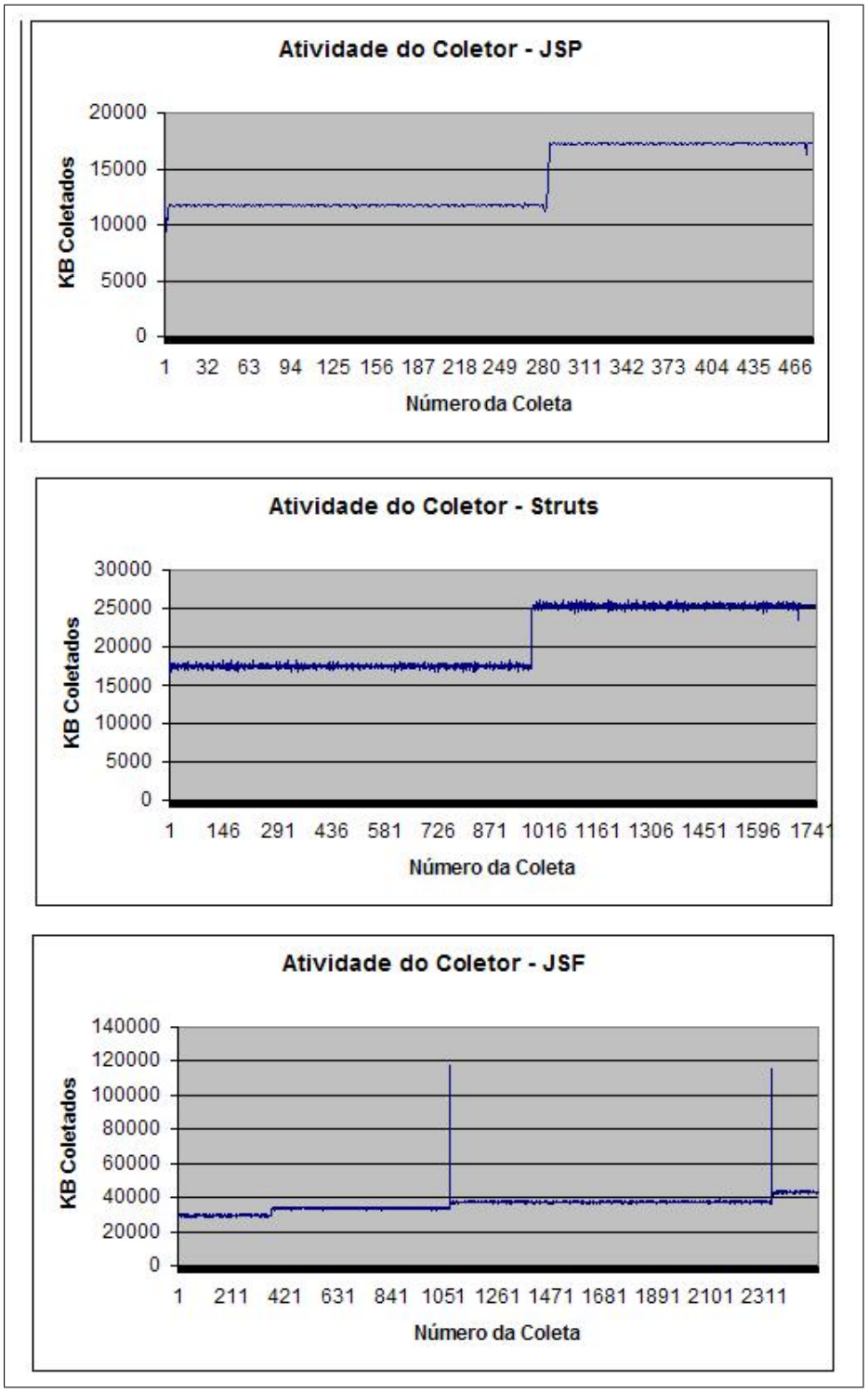

Figura 5.10: Coleta de Lixo dos Experimentos da camada de Visualização.

Tabela 5.2: Quadro Comparativo do Desempenho da Camada de Visualização

Solução	Média	Mediana	Desvio Padrão	Tempo(min)	Thoughput	Memória
JSP	95,63	15	318,05	4,35	380(req/s)	6,38 GB
Struts	160,14	63	198,64	5,35	311(req/s)	34 GB
JSF	386,26	63	653,26	13,16	126(req/s)	85,65 GB

médio do testes com a tecnologia JSF foi de 386,26 milisegundos, 400% mais lento que a JSP e 140% mais lento que Struts. Esse consumo maior de recursos também é refletido no uso da memória e nas atividades do coletor, ilustrados na figura 5.9 e 5.10. Note que o crescimento de memória é bem acentuado sendo necessária grandes coletas (*Full Garbage Collection*) gerando abismos no gráfico. Essas grandes coletas são retratadas na figura 5.10 através dos pontos destacados no gráfico pelo seu valor maior. A tabela 5.2 sintetiza os resultados dos testes da camada de visualização.

5.2.3 Resultados da Camada de Negócio

Na camada de négócio, conforme mencionado no capítulo 4, será testado uma lógica de negócio para procurar clientes devedores em uma lista. A classe Java que realiza esse componente é a forma mais simples de realizar-se esta tarefa, a essa classe chama-se de JavaBean. Este mesmo JavaBean é acesso através de duas outras tecnologias, a EJB, que fornece serviços de transação, gerência de recursos, dentre outros, e Web Services que permite a chamada a partir de outros sistemas utilizando mensagem em XML.

A Figura 5.11 ilustra a distribuição do tempo de resposta do processo utilizando JavaBean tendo uma média de 11.79 milisegundos. O uso de memória e as atividades do coletor foram constantes porém intensa, uma vez que a cada requisição um JavaBean era criado para atendê-la. As figuras 5.12 e 5.13 ilustram o crescimento da memória e as atividades do coletor, respectivamente.

Conforme ilustrado pela figura 4.13 a adoção de EJB inclui diversos serviços a um custo da inclusão de intermediários nas chamadas. Essa inclusão de fato trouxe uma queda no desempenho, no entanto, abaixo do esperado. O tempo médio de resposta do teste em EJB foi 16,40 milisegundos, apenas 39% mais lento que o JavaBean. Adicionalmente, o EJB mostrou-se extremamente estável no comportamento da memória, crescendo inicialmente para atender a chegada das requisições inicialmente, e mantendo-se estável. Este comportamento da memória é ilustrado nas figuras 5.12 e 5.13.

A tecnologia Web Services adota padrões baseados em texto (XML) para permitir a interoperabilidade entre tecnologias e dispositivos. Ao passo que permite essa característica cria um processamento extra na geração e recuperação de XML. Essa característica mostrou-se extremamente problemática no quesito desempenho fazendo com que o teste de Web Services tivesse um tempo médio de 1124 milisegundos, ou seja, 1.124 segundos para o atendimento de cada requisição. Este custo, além do intenso uso de memória que os documentos XML representam (figuras 5.12 e 5.13) é devido a abertura de conexões TCP para cada solicitação.

As figuras 5.12 e 5.13 ilustram o abundante uso de memória feito por essa aplicação.

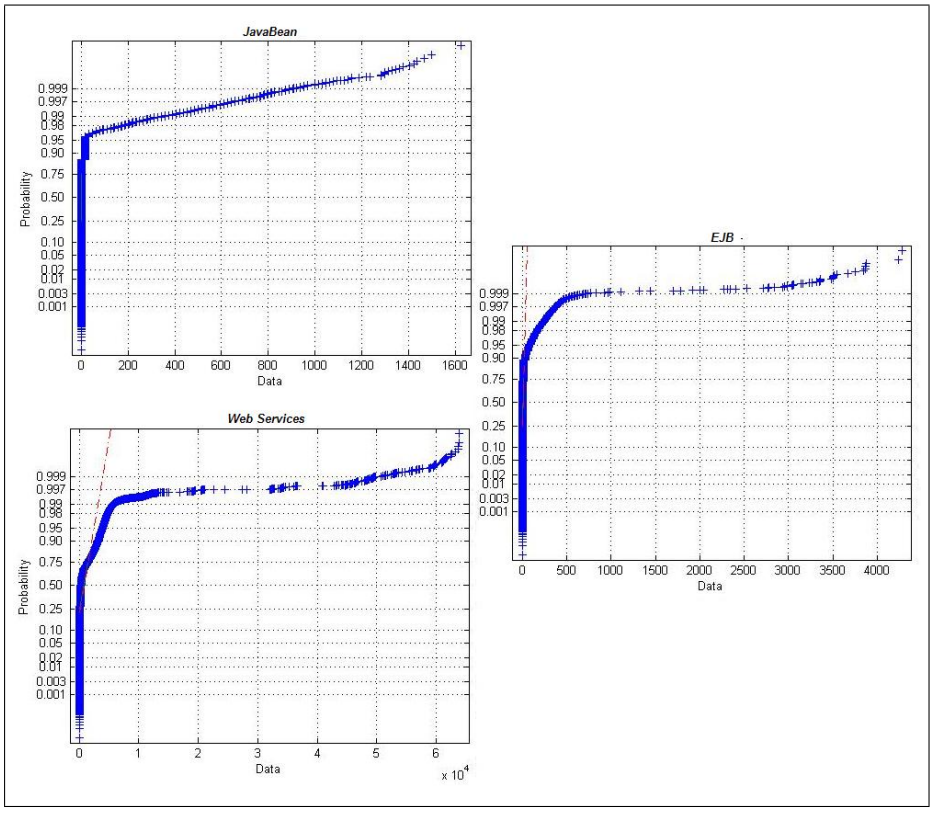

Figura 5.11: Distribuição de Probabilidade do tempo de Resposta do Experimento da Camada de Negócio.

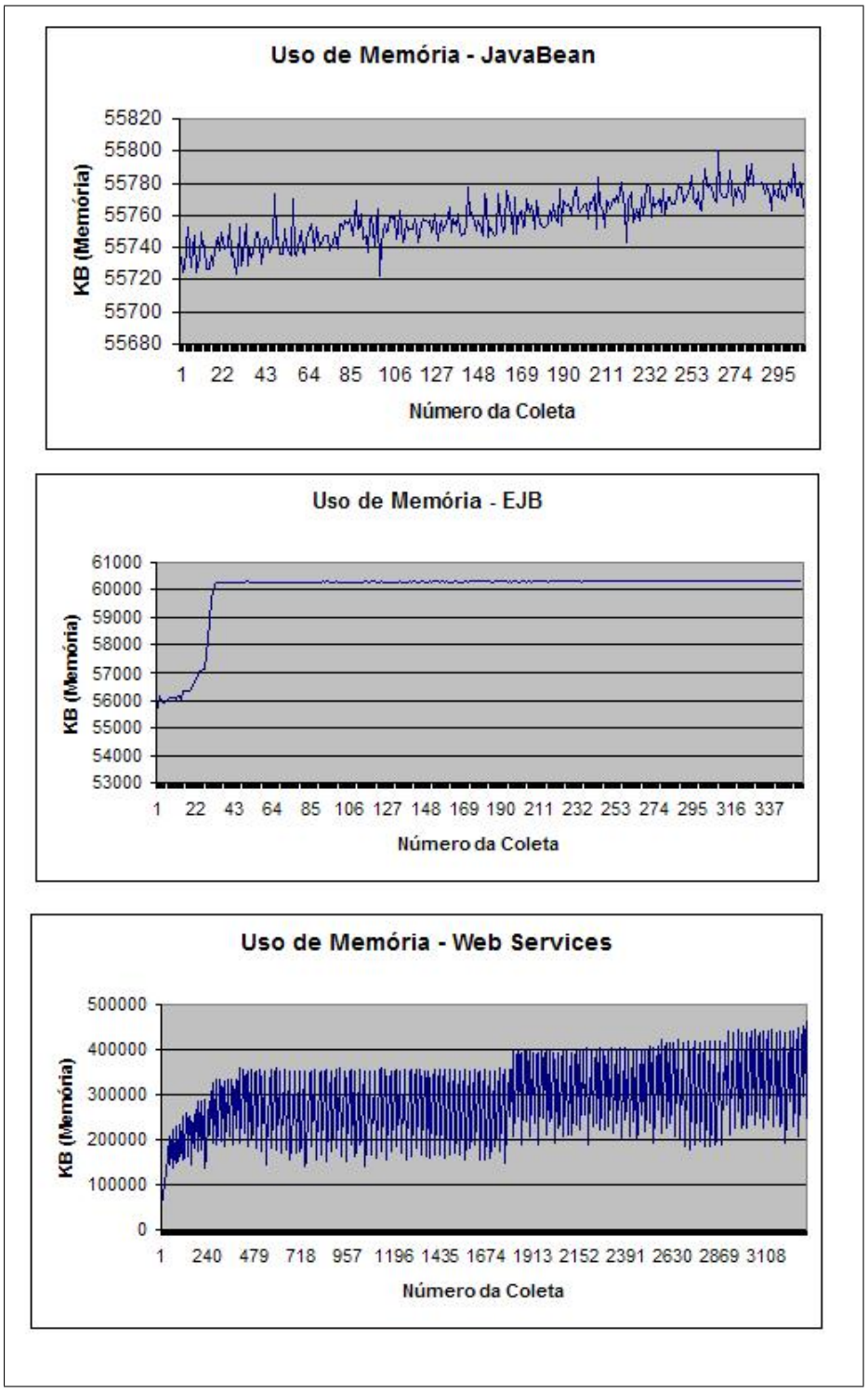

Figura 5.12: Uso de memória dos experimentos da camada de negócio

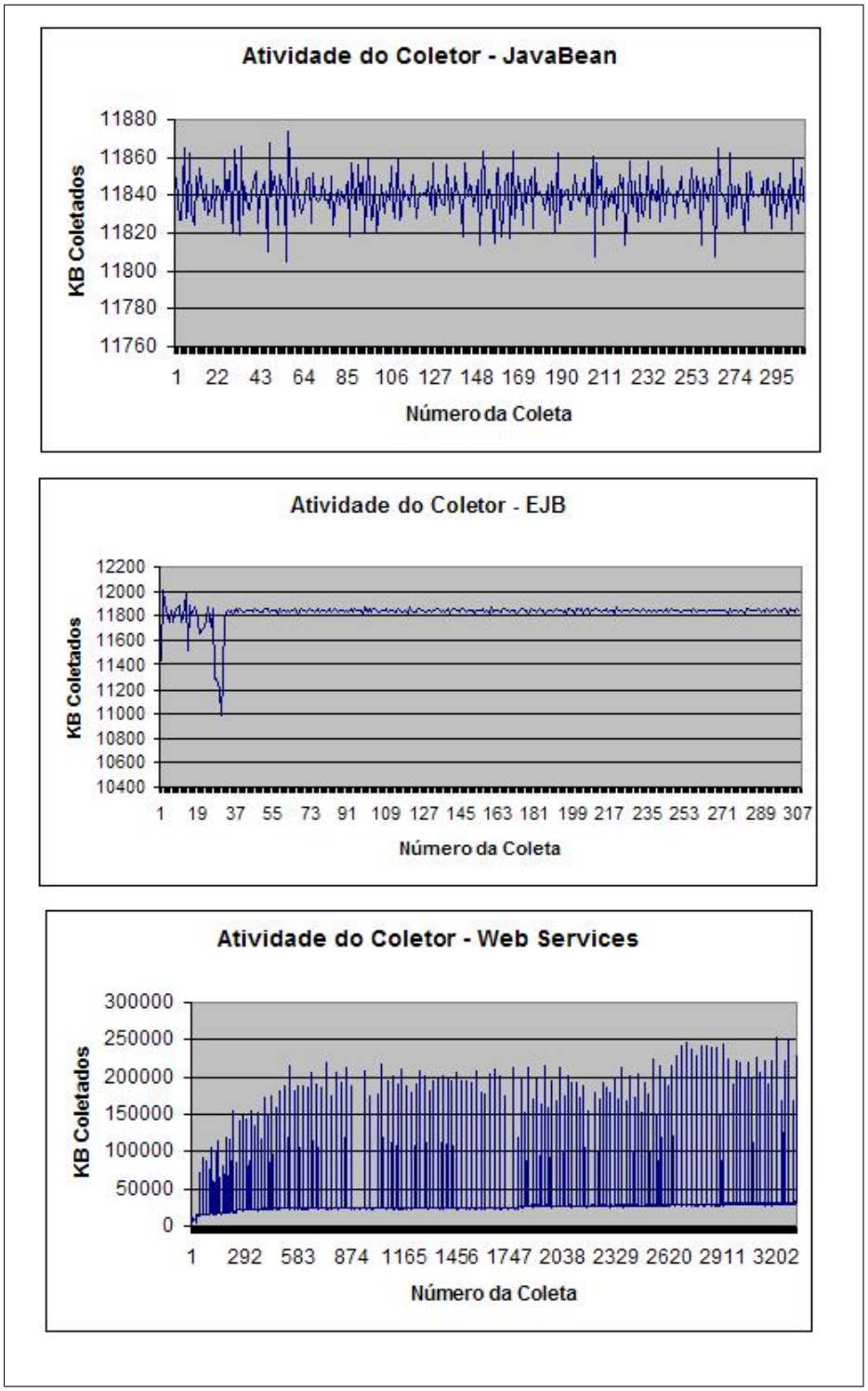

Figura 5.13: Coleta de Lixo nos experimentos da camada de negócio.

Tabela 5.3: Quadro Comparativo do Desempenho da Camada de Negócio

Solução	Média	Mediana	Desvio Padrão	Tempo(min)	Thoughput	Memória
JavaBean	11,72	0	69,89	1,38	1192(req/s)	3,37 GB
EJB	16,43	0	96,05	1,53	1093(req/s)	4 GB
Web Services	1124,00	266	2964	37	31,94(req/s)	101 GB

Tabela 5.4: Versões dos softwares utilizados em todos os testes

Software	Descrição	Versão
JVM	Máquina Virtual Java	1.6
JBoss	Servidor de Aplicações	4.0.4GA
Hibernate	Framework de Mapeamento Objeto Relacional	3.2
JBossWS	Implementação de Web Services no Servidor JBoss	1.4
Struts	Framework MVC de Visualização	1.2
Apache MyFaces	Implementação da tecnologia JavaServer Faces	1.1.5
EJB	Versão da tecnologia EJB	2.1

O uso de memória cresceu vertiginosamente e o coletor apresentou atividade intensa, inclusive realizando grandes coletas, o que impacta significativamente no desempenho. Ao todo, o coletor agiu cerca de 3100 vezes, um número bem elevado comparando com as 306 do JavaBean e 346 do EJB. A tabela 5.3 sintetiza os resultados dos testes da camada de negócio.

Conforme introduzido neste trabalho e comprovado através dos testes realizados, uma escolha incorreta da tecnologia pode por o projeto em risco no atendimento dos requisitos não-funcionais. A análise deve ser realizada camada a camada, analisando a aceitabilidade da perda do desempenho com o ganho de produtividade. Gerentes e gestores de software devem ter em mente que uma escolha de tecnologias de baixo nível trarão mais desempenho e também mais custos com mão-de-obra devido a menor produtividade. Ao passo que, a adoção de tecnologias mais nova, como JSF, conduz a gastos maiores em infra-estrutura de hardware. Essa relação tem que ser cuidadosamente analisada para um decisão correta. A tabela 5.4 ilustra as versões de todos os softwares utilizados nos testes.

Capítulo 6

Conclusões

A análise de desempenho através de gráficos e números concretos torna a decisão de qual tecnologia utilizar clara e metodológica. Muito se discute que tecnologia X é mais rápida que a tecnologia Y, no entanto, de maneira impírica e sem considerar fatores como uso de memória ou tempos de respostas em um ambiente estressado. Este trabalho apresentou o desempenho sob outra ótica: através de dados comparativos de tempos de resposta de uso de memória realizados em três tecnologias para cada uma das camadas da arquitetura da figura 4.1.

Utilizando arquiteturas de máquinas virtuais o ambiente de execução exerce um fator preponderante no impacto de desempenho. Este trabalho analisou a arquitetura da máquina virtual visando o domínio pelo engenheiro de seu funcionamento e focou principalmente nas técnicas de coleta de lixo e seu impacto no desempenho. A coleta de lixo mostrou-se efetivamente importante, mostrando-se intrusiva em vários testes que usaram memória intensamente. A tabela 6.1 ilustra o tempo dispendido nos testes que mais usaram memória, destacando-se o grande impacto da coleta com relação ao tempo geral. A mudança de algoritmos de coleta de lixo nesses casos certamente levaria a uma variação neste tempo.

Os *profilers* são capazes de detectar o consumo de memória e o desempenho já em tempo de desenvolvimento, não prolongando o problema no projeto. Já as ferramentas de monitoramento, como o JConsole, são utilizadas para monitar esse comportamento em tempo de produção.

O uso da ferramenta JMeter facilitou a realização dos testes obtendo os tempos de respostas com o número de usuários e requisições parametrizados. Um grande tempo foi despendido na construção das aplicações de testes e tratamento dos dados. Os dados gerados pelo JMeter necessitaram ser tratados e transportados para o *Matlab* para geração dos gráficos de probabilidade. O mesmo problema ocorreu com os dados de uso de memória, necessitando serem tratados para visualização sintética.

Tabela 6.1: Tempo Gasto pelo Coletor em cada teste

Teste	Tempo Coleta	Tempo Total	Percentual	Total Coletado (Gb)
Hibernate	6.23s	195s	3	10,45
JSF	64.5s	789s	8.1	85,65
Web Services	19.73mins	37.67min	52	101,53

6.1 Testes Comparativos

Na camada de persistência, o Hibernate mostrou-se um *framework* bem robusto contraposto a sua alta abstração no desenvolvimento. O tempo de resposta manteve-se baixo, 50% menor que 47 milisegundos (mediana) (tabela 5.1 e o desvio padrão foi menor que o do teste com JDBC. No entanto, se o projeto exige um altíssimo desempenho, o uso de Hibernate deve ser bem analisado. Nestes casos, a melhor solução seria a utilização de DAO. A surpresa neste teste foi o desempenho do DAO superior ao JDBC pelos motivos expostos no capítulo anterior, seção 5.2.1.

Na camada de visualização, os testes se adequaram ao comportamento esperado. O *Struts* mostrou-se um framework mais escalável que o *JavaServer Faces* e com o uso de memória mais moderado. O gráfico de coletas do Struts (figura 5.10) mostrou-se semelhante ao de JSP (figura 5.10) concluindo-se que ele não impõe um uso maior de memória mesmo sendo um framework de alto nível. Já o JavaServer Faces mostrou-se um grande consumidor de recursos. O tempo de resposta foi 140% maior que o Struts e a memória coletada foi de 87.714 Mb em JSF contra 35.650 do Struts, também 140% a mais. Com esses dados pode-se concluir que JSF deve ser cuidaosamente analisado para ser usado em um projeto de alto desempenho. Já *Struts* é mais indicado quando comparado com JSF, no entanto, a solução mais eficiente é a utilização de JSP ou Servlets diretamente, que apresentam um tempo de resposta baixíssimo e um consumo de memória menor (6533.83 Mb).

Na camada de negócio, o JavaBean mostrou-se mais rápido como o esperado. A tecnologia EJB destacou-se pelo bem comportado e constante uso de memória, gerando uma atividade constante no coletor, não demandando nenhuma grande coleta. Neste ponto EJB mostrou que efetivamente é para ser usado em processamento de grande porte e transacional competindo com *TP Monitors* COBOL. Seu tempo de resposta foi bem próximo ao JavaBean tendo a mesma mediana (inferior a 1 milisegundos). O destaque negativo ficou por conta da tecnologia Web Services, uma vez que mostrou-se lenta e com consumo de memória excessivo. Durante o experimento, a aplicação demandou a coleta de 103971,76 Mb e teve um tempo de resposta médio alto (1124) e a mediana de 244 milisegundos. Para o processamento de grande porte Web Service não é a solução mais indicada.

6.2 Trabalhos Futuros

Uma análise mais detalhada da mudança dos algoritmos de coleta de lixo nos testes de alto consumo (Hibernate, Struts, JSF e Web Services) obtendo gráficos comparativos e testando-os em máquinas multiprocesadas poderia ser encarado como um trabalho futuro, bem como a inclusão de novos frameworks e tecnologias para complementar os comparativos. Variar também implementações da mesma tecnologia como o *JSF Reference Implementation e o* Apache MyFaces traria um comparativo de implementações diferentes da mesma tecnologia.

Referências Bibliográficas

2004 (n.d.), *JVMTM Tool Interface*, Sun Microsystems, Inc.

ATOM (1994), Atom: A system for building customized program analysis tools, Programming Language Design and Implementation, Programming Language Design and Implementation.

Bancilhon, Francois (1992), *Building an Object-Oriented Database System*, 1ª edição, Academic Press.

Browne, Christopher B. (n.d.), 'Transaction processing monitors', linuxfinances.info - Disponível em `http://www.cbbrowne.com/info/tpmonitor.html`.

Cerami, Ethan (2002*a*), *Web Services Essentials - Distributed Applications with XML-RPC, SOAP, UDDI and WSDL*, 1ª edição, O'Reilly.

Cerami, Ethan (2002*b*), *WebService Essentials*, 1ª edição, O'Reilly.

Chistian Bauer, Gaving King (2005), *Hibernate em Ação*, Editora Ciência Moderna.

Chung, Mandy (2004), 'Using jconsole to monitor applications', *Sun Developer Network* .

David Chappell, Tyler Jewell (2002), *Java Web Services*, Vol. 1, O'Reilly.

David Flanagan, Brett Mclaughlin (2004), *Java 1.5 Tiger - A Developer's Notebook*, 1ª edição, Editora Campus.

Eric Jendrock, Jennifer Ball, Debbie Carson Ian Evans Scott Fordin Kim Haase (2006), *The Java EE 5 Tutorial*, Sun Microsystems, Inc.

F. Henry Korth, Silberchartz Abraham, S. Sudarshan (2006), *Sistema de Banco de Dados*, 1ª edição, Editora Campus.

Fleury, Mark (2002), *JMX Managing J2EE with Java Managment Extensions*, 1ª edição, SAMS.

Goetz, Brian (2003), Garbage collection in the hotspot jvm, Relatório técnico, Quiotix Corp - `http://www-128.ibm.com/developerworks/library/j-jtp11253/`.

Group, Gartner (n.d.), 'Gartner says worldwide relational database market increased 8 percent in 2005', 2006 Press Releases.

Hide, Paul (1999), *Java Thread Programming*, SAMS.

Holmes, Steve (1995), 'C programming', University of Strathclyde Computer Centre - Disponível em `http://www.imada.sdu.dk/~svalle/courses/dm14-2005/mirror/c/`.

Husted, Ted (2004*a*), *Mastering JavaServer Faces*, 1ª edição, John Wiley Consumer.

Husted, Ted (2004*b*), *Struts em Ação*, 1ª edição, Ciência Moderna.

Inc., OpenSynphony (2005), 'Webwork documentation', Disponível em `http://www.opensymphony.com/webwork/wikidocs/WebWork.html`.

John Crupi, Dan Malks, Deepak Alur (2004), *Core J2EE Patterns - As melhores práticas e estratégias de Design*, 2ª edição, Editora Campus.

Mark S. Johnstone, Paul R. Wilson (1998), 'The memory fragmentation problem: Solved?', *ISMM'98 - Disponível em `http://www.acm.org/pubs/citations/proceedings/plan/286860/p26-johnstone/`* pp. 26–36.

NetBeans, Inc (2006), 'Netbeans profiler features', Internet.

Newcomer, Eric (2002), *Understanding Web Services - XML, WSDL, SOAP e UDDI*, 1ª edição, ADDISON WESLEY (PEAR).

Patrick, Jim (2001), Handling memory leaks in java programs, Relatório técnico, IBM Developer Works - Disponível em `http://www-128.ibm.com/developerworks/java/library/j-leaks/`.

Project, Apache Jakarta (2005), 'What is velocity?', Site.

Reese, George (2000), *Database Programming with JDBC and Java*, 2ª edição, OREILLY e ASSOC.

Rima Patel Sriganesh, Gerald Brose, Micah Silverman (2006), *Mastering Enterprise JavaBeans 3.0*, 4ª edição, The Server Side Company.

Satish Chandra Gupta, Rajeev Palanki (2005), Java memory leaks catch me if you can, Relatório técnico, IBM Developer Works.

Souza, Welington B. (2004), 'Struts framework', JEEBrasil - Disponível `http://www.jeebrasil.com.br/mostrar/32`.

Sun Microsystems, Inc (2002), *Java Management Extensions Instrumentation and Agent Specification, v1.2*, Java Comunity Press, Santa Clara, CA 95054 USA.

Sun Microsystems, Inc. (2004*a*), 'J2se 5.0 performance white paper', *Sun Developer Network(SDN) Disponível em `http://java.sun.com/performance/reference/whitepapers/5.0_performance.html`*.

Sun Microsystems, Inc (2004*b*), *Tuning Garbage Collection with the 5.0 Java Virtual Machine*, Sun Microsystems, Inc - Disponível em `http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html`.

Tanembaum, Andrew S. (2003), *Redes de Computadores*, 1ª edição, Editora Campus.

Tim Lindholm, Frank Yellin (1999), *The Java Virtual Machine Specification*, ADDISON WESLEY.

Tude, Eduardo (2003), Service level agreement (sla), Tutorial, Teleco (www.teleco.com.br).

Venners, Bill (1998), The hotspot virtual machine how hotspot can improve java program performance and designs, Relatório técnico, Artima Developer - Disponível em `http://www.artima.com/designtechniques/hotspot.html`.

Venners, Bill (2000), *Inside The Java 2.0 Virtual Machine*, OSBORNE MCGRAW-HIL.

Whaley, John (2000), A portable sampling-based profiler for java virtual machines, *em* 'Proceedings of the ACM 2000 conference on Java Grande', ACM, ACM Press, San Francisco, California, United States, pp. 78 – 87.

Würthinger, Thomas (2005), 'Incremental garbage collection: The train algorithm', *Seminar Garbage Collection - Disponível em* *`http://www.ssw.uni-linz.ac.at/Teaching/Lectures/Sem/2005/Reports/Wuerthinger.pdf`*.

Apêndice A

Aplicação de Teste - Camada de Mapeamento

A.1 Teste de JDBC

```
public class JDBCVendas {

 public static DataSource ds;

 public void getVendas() throws Exception {

   Connection con = null;
   if (ds == null) {
      InitialContext ic = new InitialContext();
      ds = (DataSource) ic.lookup("java:/dsMestrado");
   }

   con = ds.getConnection();

   String sql = "select this_.id_venda as id_venda, this_.data as data, " +
   "this_.valor as valor, this_.id_funcionario as id_funcionario, " +
   "funcionari2_.id_funcionario as id_funcionario, funcionari2_.cpf as cpf," +
   " funcionari2_.nome as nome, funcionari2_.endereco as endereco, " +
   "funcionari2_.id_empresa as id_empresa, empresa3_.id_empresa as id_empresa, " +
   "empresa3_.endereco as endereco_empresa, empresa3_.razao_social as razao_social" +
   " from venda this_ inner join funcionario funcionari2_ on " +
   "this_.id_funcionario=funcionari2_.id_funcionario " +
   "left outer join empresa empresa3_ on funcionari2_.id_empresa=empresa3_.id_empresa";

   Statement st = con.createStatement();
   ResultSet rs = st.executeQuery(sql);

   while (rs.next()) {
```

```
    // dados da venda
    int idVenda = rs.getInt("ID_VENDA");
    Date data = rs.getDate("DATA");
    double valor = rs.getDouble("VALOR");
    // dados do funcionario
    int idFuncionario = rs.getInt("ID_FUNCIONARIO");
    String nomeFunc = rs.getString("NOME");
    Long cpf = rs.getLong("CPF");
    String endereco = rs.getString("ENDERECO");
    // empresa
    String empresa = rs.getString("razao_social");
    int idEmpresa = rs.getInt("ID_EMPRESA");
    String enderecoEmp = rs.getString("ENDERECO_EMPRESA");

    }

    rs.close();
    st.close();

    con.close();
  }
}
```

A.2 DAO

```
public class VendaDAO {

  public static DataSource ds;

  public Collection<Venda> getVendas() throws Exception {

    Connection con = null;
    if ( ds == null ) {
        InitialContext ic = new InitialContext();
        ds = (DataSource) ic.lookup("java:/dsMestrado");
    }

    con = ds.getConnection();

    Statement st = con.createStatement();
    String sql = "select this_.id_venda as id_venda, this_.data as data, " +
    "this_.valor as valor, this_.id_funcionario as id_funcionario, " +
    "funcionari2_.id_funcionario as id_funcionario, funcionari2_.cpf as cpf," +
    " funcionari2_.nome as nome, funcionari2_.endereco as endereco, " +
```

```
    "funcionari2_.id_empresa as id_empresa, empresa3_.id_empresa as id_empresa, " +
    "empresa3_.endereco as endereco_empresa, empresa3_.razao_social as razao_social" +
    " from venda this_ inner join funcionario funcionari2_ on " +
    "this_.id_funcionario=funcionari2_.id_funcionario " +
    "left outer join empresa empresa3_ on funcionari2_.id_empresa=empresa3_.id_empresa";

    ResultSet rs = st.executeQuery(sql);

    ArrayList<Venda> lista = new ArrayList<Venda>();
    while (rs.next()) {

        Venda v = new Venda();
        Funcionario f = new Funcionario();
        Empresa e = new Empresa();

        // dados da venda
        v.setData(rs.getTimestamp("DATA"));
        v.setValor(rs.getDouble("VALOR"));
        v.setIdVenda(rs.getInt("ID_VENDA"));

        // dados do funcionario
        f.setCpf(rs.getLong("CPF"));
        f.setIdFuncionario(rs.getInt("ID_FUNCIONARIO"));
        f.setNome(rs.getString("NOME"));
        f.setEndereco(rs.getString("ENDERECO"));
        f.setEmpresa(e);

        v.setFuncionario(f);

        e.setEndereco(rs.getString("ENDERECO_EMPRESA"));
        e.setIdEmpresa(rs.getInt("ID_EMPRESA"));
        e.setRazaoSocial(rs.getString("RAZAO_SOCIAL"));

        lista.add(v);

    }
    st.close();

    con.close();

    return lista;
  }
}
```

A.3 Hibernate

A.3.1 Classe

```
public class VendaHibernate {

    public Collection<Venda> getVendas() throws Exception {

        GenericDAO dao = new GenericDAO();
        Collection<Venda> vendas =  dao.findAll(Venda.class);
        dao.close();

        return vendas;

    }
}
```

A.3.2 Configuração

```
<?xml version="1.0" encoding="utf-8"?> <!DOCTYPE
hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration
DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory name="java:comp/env/hibernate/SessionFactory">
        <property name="hibernate.bytecode.use_reflection_optimizer">false</property>
        <property name="connection.datasource">java:/dsMestrado</property>
        <property name="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</prope
        <property name="hibernate.show_sql">false</property>
        <mapping class="mapeamento.dominio.Venda" />
        <mapping class="mapeamento.dominio.Empresa" />
        <mapping class="mapeamento.dominio.Funcionario" />
    </session-factory>
</hibernate-configuration>
```

Apêndice B

Aplicação de Teste - Camada de Visualização

B.1 Teste de JSP

```
<h1>Lista Funcionários</h1>

<c:set var="lista"
value="<%= mestrado.web.GeraListaFuncionarios.getFuncionarios() %>"/>

<table>
    <c:forEach items="${lista}" var="f">
        <tr>
            <td> ${f.cpf} </td>
            <td> ${f.nome} </td>
            <td> ${f.endereco} </td>
            <td> ${f.empresa.razaoSocial} </td>
        </tr>
    </c:forEach>
</table>
```

B.2 Struts

B.2.1 Action

```
public class ListaFuncionarioAction extends Action {

    @Override
    public ActionForward execute(ActionMapping map, ActionForm arg1,
     HttpServletRequest req, HttpServletResponse arg3) throws Exception {

        req.setAttribute("funcionarios", GeraListaFuncionarios.getFuncionarios());
```

```
        return map.findForward("lista");
    }

}
```

B.2.2 JSP

```
<h1>Lista Funcionários</h1>

<table>
    <logic:iterate name="funcionarios" id="funcionario">
        <tr>
            <td> <bean:write name="funcionario" property="cpf"/> </td>
            <td> <bean:write name="funcionario" property="nome"/> </td>
            <td> <bean:write name="funcionario" property="endereco"/> </td>
            <td> <bean:write name="funcionario" property="empresa.razaoSocial"/> </td>
        </tr>
    </logic:iterate>
</table>
```

B.3 JSF

B.3.1 Managed Bean

```
public class FuncionarioBean {

    public ArrayList<Funcionario> getAll() {
        return GeraListaFuncionarios.getFuncionarios();
    }

}
```

B.3.2 Visualização

```
<f:view>

    <h1>Lista Funcionários</h1>

    <h:dataTable value="#{funcionario.all}" var="f">
        <h:column>
```

```
            <h:outputText value="#{f.cpf}"/>
        </h:column>
        <h:column>
            <h:outputText value="#{f.nome}"/>
        </h:column>
        <h:column>
            <h:outputText value="#{f.endereco}"/>
        </h:column>
        <h:column>
            <h:outputText value="#{f.empresa.razaoSocial}"/>
        </h:column>
    </h:dataTable>

</f:view>
```

Apêndice C

Aplicação de Teste - Camada de Negócio

C.1 JavaBean

```
 public class Devedores {

    public int processaDevedores(Cliente[] clientes) {

        ArrayList<Cliente> devedores = new ArrayList<Cliente>();

        for (Cliente c : clientes) {

            if (c.getId() % 3 == 0) {
                devedores.add(c);
            }
        }

        return devedores.size();

    }

}
```

C.2 EJB

```
public class DevedoresBean extends Devedores implements SessionBean
{

    public void setSessionContext(SessionContext arg0)
    throws EJBException, RemoteException {
    }

    public void ejbRemove() throws EJBException, RemoteException {
```

```
    }

    public void ejbActivate() throws EJBException, RemoteException {
    }

    public void ejbPassivate() throws EJBException, RemoteException {
    }

    public void ejbCreate() {

    }

}

public interface DevedoresHome extends EJBHome {

    public DevedoresEJB create() throws CreateException, RemoteException;

}


public interface DevedoresEJB extends EJBObject {

    public int processaDevedores(Cliente[] clientes)
            throws RemoteException;

}
```

C.2.1 Configuração

```
<ejb-jar>
    <description>
        <![CDATA[]]>
    </description>
    <display-name>Mestrado</display-name>
    <enterprise-beans>
        <session>
            <description>
                <![CDATA[]]>
            </description>
            <ejb-name>DevedoresBean</ejb-name>
            <home>mestrado.negocio.ejb.DevedoresHome</home>
            <remote>mestrado.negocio.ejb.DevedoresEJB</remote>
            <ejb-class>mestrado.negocio.ejb.DevedoresBean</ejb-class>
            <session-type>Stateless</session-type>
```

```
            <transaction-type>Container</transaction-type>
        </session>
    </enterprise-beans>
</ejb-jar>
```

C.3 Web Service

Para expor uma classe como Web Service o Apache Axis foi utilizado, não foi necessário criar nenhuma classe para que isso seja feito. Apenas arquivos de configurações. O cliente de acesso ao Web Service também foi gerado.