Relatório Técnico

Núcleo de Computação Eletrônica

Explorando a Robustez de Servidores Enterprise JavaBeans

Oliveira, C. E. T.
Leite, E. B.

NCE - 06/01

Universidade Federal do Rio de Janeiro

Explorando a Robustez de Servidores Enterprise JavaBeans

Oliveira, C.E.T, Leite, E.B.

Núcleo de Computação Eletrônica – Universidade Federal do Rio de Janeiro
E-mail: carlo@nce.ufrj.br

Resumo

Nos últimos anos o crescimento da tecnologia Java criou um mercado bastante promissor: o de servidores de aplicação. Com o aumento da oferta de produtos nesse mercado torna-se necessária uma avaliação profunda sobre um produto para se garantir que o mesmo cumpra os requisitos de um sistema. Este trabalho procura levantar métricas que possam ser usadas com o objetivo de avaliar a performance de servidores que implementam a especificação Enterprise JavaBeans, e a partir dessas métricas gerar baterias de testes que possam gerar conclusões concretas sobre a performance e a viabilidade de um servidor.

Abstract

In the last years the growth of Java technology created a promising market: the application server market. The growing number of products being offered brings the need of a deep analysis of these products in order to guarantee that they can cover all the requirements of a system. This paper discusses a few metrics that can be used in order to evaluate the performance of application servers that implement the Enterprise JavaBeans specification, and shows some tests that can be used to generate concrete conclusions about the performance and viability of an application server.

Palavras chave: Java, Enterprise JavaBeans, JNDI, JDBC, CORBA

Introdução

A evolução e o amadurecimento da arquitetura Java ocorrida nos últimos anos permitiu que a mesma passasse a ser utilizada nos ambientes de servidores. Essa evolução foi fundamental para o surgimento de um grande mercado de servidores de aplicações, onde podemos encontrar uma gama extremamente variada de produtos que podem atender aos mais variados requisitos de um sistema.

O fator crucial do surgimento desse mercado foi a padronização por parte dos criadores da linguagem Java de uma série de serviços que juntos formam a base necessária para a criação de um servidor de aplicações. É crucial que um servidor de aplicação siga os padrões sugeridos, pois será uma garantia de que o mesmo poderá atingir um grande número de desenvolvedores, que preferem se adequar aos padrões do mercado. O uso de padrões também facilita a vida dos desenvolvedores, pois eles podem desenvolver os seus sistemas sem se preocupar em se adequar a um determinado produto, dando a eles maior flexibilidade e poder na escolha das ferramentas que serão usadas para o desenvolvimento do sistema.

Neste trabalho vamos analisar os componentes da especificação Enterprise JavaBeans [1] e o papel deles na performance geral de um sistema. Essa arquitetura influencia na escolha das métricas, e para isso um conjunto de componentes foi desenvolvido para obter dados sobre a performance de um servidor, e a partir desses dados podemos realizar análises comparativas entre servidores, ajudando na tomada de decisão sobre qual produto adquirir para ser usado como base no desenvolvimento de um sistema.

Essas avaliações são importantes para que possamos validar a especificação, submetendo as suas implementações a um conjunto de testes que permitem identificar as falhas da especificação e eventuais pontos de melhoria.

Este trabalho inicialmente apresenta uma pequena introdução à especificação Enterprise JavaBeans, expondo as motivações e os principais componentes da mesma. A seguir apresentaremos os componentes da especificação que influem diretamente na performance de um servidor de aplicações, e por fim mostraremos os componentes de teste criados para realização das análises em alguns servidores de aplicação, comparando os resultados obtidos entre eles.

Introdução à Arquitetura Enterprise JavaBeans

A arquitetura Enterprise JavaBeans tem por objetivo disponibilizar aos desenvolvedores, através do uso de um *framework* específico, toda uma gama de serviços de infra-estrutura que antes precisavam ser criados e mantidos pela própria equipe de desenvolvimento da aplicação alvo, delegando a eles apenas a responsabilidade de implementar a lógica da aplicação.

Esta arquitetura define o que chamamos de *Component Transaction Monitors*. Nesse modelo o programador desenvolve todo o conjunto de componentes necessários para satisfazer a lógica da aplicação e depois instala esses componentes em um *container*, que é a peça do servidor responsável por gerenciar todos os componentes instalados pelos desenvolvedores. Características adicionais que não estão relacionadas com a lógica da aplicação são definidas no momento da instalação desses componentes.

A especificação define dois tipos de Enterprise JavaBeans: *session beans* e *entity beans*. *Session*

beans são usados para se representar a execução de casos de uso [6], ou seja, eles são usados para se implementar os cenários de interação entre o usuário do sistema e a aplicação. *Session beans* podem ser *statefull* ou *stateless*; *statefull session beans* mantém o seu estado interno entre invocações de métodos. Existe uma instância de um *statefull session bean* para cada cliente do *bean*, e o servidor é responsável por gerenciar esses *beans* de forma a melhor aproveitar os recursos de memória do sistema. Um *stateless session bean* não mantém o seu estado interno entre invocações de métodos, o que permite ao servidor realizar diversas otimizações que resultam em um melhor uso dos recursos da máquina. Como o estado não é mantido entre chamadas uma mesma instância de um *session bean* pode ser usada para atender as requisições de diferentes clientes.

Entity beans são usados para representar os dados do sistema e o comportamento intrínseco a eles. O servidor é responsável por manter a integridade dos dados dos *entity beans*, bem como garantir a persistência desses dados em meio de armazenamento secundário, para que os mesmos possam ser recuperados posteriormente.

Existem duas abordagens para o controle da persistência de um *entity bean*: o *container* notifica o *entity bean* quando o mesmo precisa persistir o seu estado. O *bean* então executa seu próprio código que irá gravar o seu estado em algum meio de persistência. A isso chamamos de *bean managed persistence*; o *container* pode por conta própria persistir o estado do *entity bean*. O servidor deve prover ferramentas para que o desenvolvedor do *entity bean* possa configurar como a persistência será feita. A isso chamamos de *container managed persistence*. A grande maioria dos servidores existente no mercado usa como mecanismo de persistência bancos de dados relacionais. Cabe ao servidor prover ferramentas que permitam a realização do mapeamento dos objetos para as tabelas de um banco de dados.

Para gerar um Enterprise JavaBean o desenvolvedor precisa criar quatro peças básicas: a *home interface*, a *remote interface*, a classe que implementa os métodos da *remote interface* e um arquivo XML [9] que descreve as características do EJB.

A *home interface* define os métodos responsáveis pelo controle do ciclo de vida dos EJBs. Através dele podemos criar, localizar e destruir instâncias de EJBs. A *home interface* de um *session bean* deve conter apenas métodos para criar instâncias; a de um *entity bean* deve conter métodos para criar e localizar instâncias.

A *remote interface* é responsável por informar quais são os métodos do Enterprise JavaBean que serão expostos ao cliente. Para cada método existente na *remote interface* deve existir outro com a mesma assinatura na classe responsável por implementar os métodos do Enterprise Bean. Esse método existente nessa classe contém a implementação da lógica necessária para o funcionamento da aplicação.

O arquivo XML tem por objetivo informar o servidor das características que o Enterprise Bean assumirá durante a operação do servidor como, por exemplo, campos a serem persistidos, características transacionais dos métodos e controle de segurança.

O servidor é então responsável por gerar as classes que implementam a *home* e a *remote interface*, seguindo as informações descritas no arquivo XML fornecido pelo desenvolvedor do Enterprise Bean. A implementação da *remote interface* gerada pelo servidor simplesmente repassa as chamadas para os métodos correspondentes na classe que realmente contém a implementação da lógica. A implementação gerada pelo servidor acrescenta apenas código que

reflete as características descritas no XML, como controle de transação e de segurança.

Exploração das Arquiteturas x Impacto na Performance e Robustez do Sistema

Toda a arquitetura Enterprise JavaBeans é implementada através do conceito de componentes, responsáveis por prover serviços. Uma aplicação pode ser construída através da união de diversos componentes que podem ser obtidos de fornecedores variados, ou desenvolvidos por conta própria.

Esta característica de componentização da arquitetura pode trazer problemas de performance se as interfaces de interação entre esses componentes não forem padronizadas da forma correta.

Para analisarmos o comportamento da interação desses componentes é preciso testar o funcionamento da arquitetura, tentando extrair métricas que possam avaliar o impacto que esses componentes podem ter no desempenho geral de um servidor.

Alguns componentes podem influir na performance e na robustez do sistema. A seguir vamos analisá-los.

Componentes que afetam a performance

Esses componentes influem no tempo em que a aplicação leva para responder às requisições dos clientes. Geralmente envolvem recursos que eventualmente precisam interagir com os clientes da aplicação. Nessa categoria podemos listar os serviços *JNDI* e *JDBC*.

JNDI (Java Naming and Directory Interface)

O *JNDI* [3] é uma interface de programação para acessar serviços de diretório, a partir do qual podemos ter acesso a serviços expostos na forma de objetos. Todo servidor Enterprise JavaBeans disponibiliza as referências aos *enterprise beans* instalados através de um serviço *JNDI*.

Dependendo da abordagem do serviço *JNDI*, a busca por referências no serviço pode causar um impacto na performance. Serviços baseados em *CORBA* [8] costumam ter uma resolução de nomes mais lenta do que a de serviços implementados em *TCP*, pois no primeiro o provedor do objeto não é conhecido no momento da resolução, devido à transparência de localização de objetos promovida pela especificação *CORBA*. Nas implementações *TCP*, a localização do servidor é previamente conhecida.

JDBC (Java DataBase Conectivity)

O *JDBC* [7] é o padrão Java para acesso a bancos de dados relacionais. Geralmente o fabricante do servidor de banco de dados fornece uma implementação das interfaces *JDBC* para que uma aplicação Java possa acessá-lo.

Esse componente causa um grande impacto na performance do sistema, pois depende explicitamente de como essas interfaces de acesso foram implementadas, bem como a performance geral do servidor de banco de dados em si. Podemos ter um servidor extremamente rápido, mas se os mecanismos de acesso aos dados do banco não forem eficientes, a performance geral será degradada. É papel do servidor tentar amenizar os impactos de performance que as interfaces *JDBC* podem trazer ao sistema.

Componentes que afetam a robustez

Esses componentes influem na forma como o servidor utiliza os recursos da máquina, otimizando ao máximo a utilização desses recursos, evitando que o sistema como um todo sofra um aumento nos tempos de resposta das requisições. Nessa categoria podemos listar o *pooling* de recursos e o *clustering* de servidores.

Pooling de recursos

O *pooling* de recursos é talvez um dos mecanismos eficientes, e por isso um dos mais utilizados para o gerenciamento de recursos. Ele é usado para otimizar o uso de recursos que sejam escassos ou cuja criação é muito custosa.

O *pool* é iniciado com uma quantidade de recursos. Toda vez que um recurso for requerido ele é obtido do *pool*, e o mesmo fica alocado até que o requisitante devolva o recurso ao *pool*. Se não existirem recursos disponíveis, mais um recurso é criado e associado ao *pool* e este é retornado ao requisitante. Costuma-se especificar a quantidade máxima de recursos que podem estar no *pool*, evitando que ocorram problemas em função do excesso de recursos criados.

Se essa quantidade máxima de recursos no *pool* for atingida o requisitante pode assumir um estado de espera até que um recurso esteja disponível ou até que o tempo de espera pelo recurso se esgote, gerando uma condição de erro. Essa condição pode ser gerada imediatamente, dependendo da necessidade.

O mecanismo de *pooling* promove um melhor compartilhamento de recursos, partindo da premissa que o recurso é passível de ser utilizado por qualquer requisitante do mesmo. Ele evita a criação e destruição desnecessária desses recursos, diminuindo o tempo de resposta da aplicação que faz uso desse mecanismo.

Podemos ver a aplicação desse mecanismo em vários pontos de um servidor. Um deles está na implementação do protocolo que rege o mecanismo de invocação remota de objetos usada pelo servidor. O servidor utiliza um *pool* de *threads* para obter *threads* que serão usadas para processar as chamadas remotas.

Para cada requisição remota o servidor obtém uma *thread* do *pool*, e a *thread* retornada irá executar o código necessário para efetivar a requisição. Ao final da requisição remota a *thread* é hibernada e então retornada ao *pool*, aguardando para ser reutilizada em outra requisição.

O uso do *pool* de *threads* ajuda a otimizar o uso dos recursos do sistema operacional, bem como aumenta o tempo de resposta, pois a criação de *threads* é uma operação custosa.

O *pooling* também é usado para controlar o acesso às conexões com o banco de dados. Toda vez que a aplicação necessita de uma conexão *JDBC* com o banco de dados a mesma será obtida de um *pool*. Ao final da utilização da conexão ela é devolvida ao *pool*. Isso acarreta a diminuição do tempo de resposta das requisições, pois não existe a necessidade de criação constante de conexões com o banco.

A especificação do Enterprise JavaBeans define um modelo de ciclo de vida para as instâncias de um *entity bean* que presume o uso do mecanismo de *pooling*. Ela define 3 possíveis estados para um *entity bean*: ou o *bean* não existe, ou ele existe e se encontra no *pool* ou ele existe e está pronto para uso.

Quando o *bean* se encontra no *pool*, ele está disponível para ser usado para a execução dos

métodos de localização de instâncias da *home interface*, que não estão associados a nenhuma instância específica de um enterprise bean. Quando um enterprise bean é criado ou reativado, uma instância do *pool* é solicitada, fazendo com que ela se torne pronta para ser usada nas invocações do enterprise bean.

Toda vez que o bean não for mais necessário – fato que ocorre quando o mesmo é removido ou desativado – ele é retornado ao *pool*, ficando disponível para o uso por outros enterprise beans. Esse modelo permite uma reutilização de memória mais eficiente por parte do servidor, a partir do momento em que o mesmo não precisa ficar alocando instâncias dos *beans* a todo o momento.

Clustering de servidores

Os servidores de aplicação precisam ser capazes de suportar um grande número de clientes simultâneos, sem que a performance geral do sistema seja afetada. Mas existe uma limitação intrínseca que faz com que um servidor não possa exceder um determinado número de clientes, em função das limitações de hardware (memória, cpu e rede) bem como das limitações de software (sistema operacional e o próprio servidor).

Para contornar este problema é comum vermos implementações de servidores que permitem o uso de duas ou mais máquinas, cada qual com uma cópia do servidor sendo executada, que trabalham em conjunto para servir a aplicação desenvolvida. Esse tipo de configuração é conhecido como *clustering*.

No mecanismo de *clustering* o servidor procura distribuir de maneira uniforme o processamento das requisições feitas pelos clientes da maneira mais transparente possível, de forma que o cliente não precise se preocupar em quem está processando a sua requisição. Se eventualmente os servidores não estiverem conseguindo atender uma determinada carga de clientes, basta adicionar mais máquinas ao *cluster*, e os servidores irão redistribuir o processamento entre as novas máquinas e as que já estavam no *cluster*.

Embora existam diversas implementações para o mecanismo de *clustering*, todas elas são desenvolvidas em torno do serviço *JNDI* fornecido pelo servidor. Podem existir três abordagens para a implementação do *cluster*: independente, centralizado e compartilhado global.

Na abordagem independente cada servidor possui o seu serviço *JNDI*. O balanceamento de carga é feito pela implementação do serviço de localização *JNDI* no cliente. Quando um cliente executa uma busca por um bean a parte cliente do serviço escolhe um dos servidores disponíveis e o usa para o processamento das requisições. Essa escolha pode ser aleatória, ou função da carga corrente nos servidores.

Na abordagem centralizada existe um serviço central que conhece todos os servidores e objetos existentes. Esse serviço é geralmente implementado como um serviço de nomes *CORBA* [8](*CosNaming*). O próprio serviço *JNDI* é responsável por definir qual servidor será escolhido.

Na abordagem compartilhada global cada servidor possui um servidor *JNDI*, mas notifica os demais servidores da sua existência, de forma que se um servidor se julgar sobrecarregado, ele mesmo pode repassar as requisições a outro serviço *JNDI*.

Possíveis gargalos

Existem situações em que os componentes mencionados não provam a sua eficiência, e podem ser sintomas de que podem existir erros na modelagem e na implementação da aplicação.

Uma dessas situações é quando o tempo médio entre duas requisições a um recurso de um *pool* é menor que o tempo médio que um recurso leva para ser devolvido ao *pool*. Com o passar do tempo o limite máximo de instâncias do *pool* é alcançado, o que irá acarretar no aumento gradativo do tempo de espera por um recurso, aumentando o tempo de resposta da aplicação e prejudicando a performance do sistema.

Outro possível gargalo se concentra nos recursos de rede. Deve-se ter cuidado especial na modelagem de uma aplicação a fim de se evitar o uso desnecessário de recursos da rede. Por exemplo, deve-se procurar diminuir ao máximo a necessidade de chamadas a objetos remotos, afim de que seja possível executar o maior número de requisições possíveis com a capacidade de tráfego existente.

Também é possível a ocorrência de gargalos de recursos de rede quando se usa uma configuração de *cluster*, devido à necessidade de comunicação entre os nós do *cluster* para fins de sincronização. Quanto maior o número de nós do *cluster*, maior a quantidade de pacotes de sincronização circulando na rede.

Baterias de Teste e Sistemas de Saturação de Carga

Para avaliarmos como um servidor reage a situações de saturação foi desenvolvido um conjunto de enterprise beans e uma aplicação cliente que juntos irão colher estatísticas sobre os tempos de resposta do servidor. Essas estatísticas serão usadas como forma de comparação entre os servidores.

Foram desenvolvidos três enterprise beans para este propósito: *StressEntity*, *EntityInitializer*, *StressEntityFinder*. O enterprise bean *StressEntity* é um *entity bean* cuja persistência é gerenciada pelo servidor EJB. Ele possui dois campos persistentes: um identificador do tipo *String*, que é gerado automaticamente, e um valor inteiro qualquer. Ele será usado para avaliarmos a performance dos mecanismos de acesso ao banco de dados.

O enterprise bean *EntityInitializer* é um *session bean stateless* cujo objetivo é criar uma certa quantidade de instâncias de *StressEntity* beans. Ele possui apenas um método *createBeans*, que irá criar instâncias de *StressEntity* seguindo a faixa de valores passada como parâmetro.

Por fim, o enterprise bean *StressEntityFinder* é um *session bean stateless* cujo objetivo é executar o método *findByValueRange* da *home interface* de *StressEntity*. Ele possui apenas um método *findBeans*, que irá processar a localização das instâncias, seguindo a faixa de valores passada como parâmetro.

A aplicação cliente desenvolvida irá disparar um número predefinido de requisições ao servidor. Ao final de cada requisição os dados coletados são escritos em um arquivo texto. Esses arquivos texto são então importados para uma planilha eletrônica, aonde os dados podem ser melhor avaliados.

Esse teste analisa apenas como o servidor se comporta em situações onde ocorrem uma chegada rápida de pedidos de requisição. Podemos simular situações onde o servidor pode facilmente entrar em um estado indefinido em função da quantidade de conexões. Podemos analisar se as estratégias de *pooling* estão realmente sendo utilizadas pelo servidor, bem como testar como o mesmo reage quando utilizarmos um *cluster* de servidores.

Avaliação de Resultados Sobre o Pano de Fundo Arquitetural

Nesta seção vamos apresentar os resultados dos testes que foram realizados em alguns servidores de aplicação, e tentaremos analisar os resultados e obter conclusões sobre a viabilidade da performance de ambos os servidores. Os testes foram realizados com os seguintes servidores: Inprise Application Server 4.1.1 e Jboss 2.1.

O Inprise Application Server 4.1.1 [11] é um servidor de aplicação comercial da Borland, que implementa todas os serviços requeridos na especificação *J2EE* [12]. O servidor é totalmente baseado em *CORBA*, desde os serviços *JNDI* e de transação até o protocolo de comunicação remota de objetos. Dispõe de mecanismos de *clustering* de servidores e uma interface de administração bastante amigável.

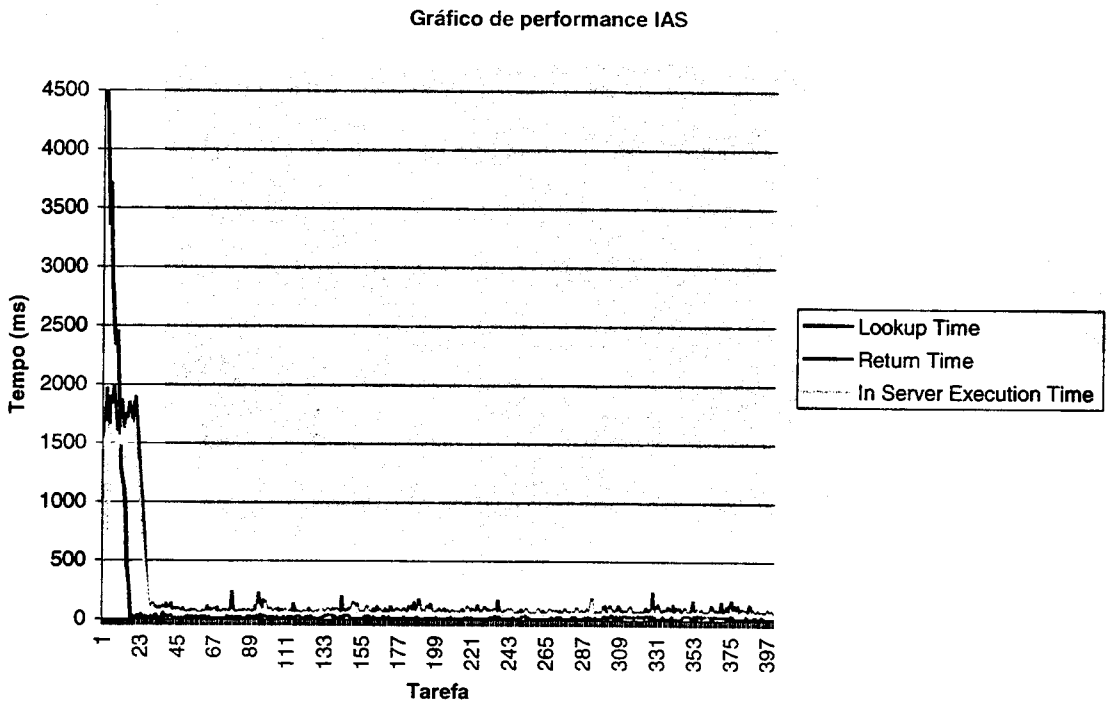
O Jboss 2.1 [10] é um servidor de aplicação *freeware*, e seu desenvolvimento é realizado por uma comunidade de programadores que se encontra espalhada pelo mundo. Embora sendo um software *opensource*, ele possui características robustas que lhe proporcionam uma boa performance em relação a outros servidores. Ele implementa todos os requisitos exigidos pela especificação *J2EE*, mas ainda não possui a funcionalidade de clustering, e sua interface de administração não é muito amigável.

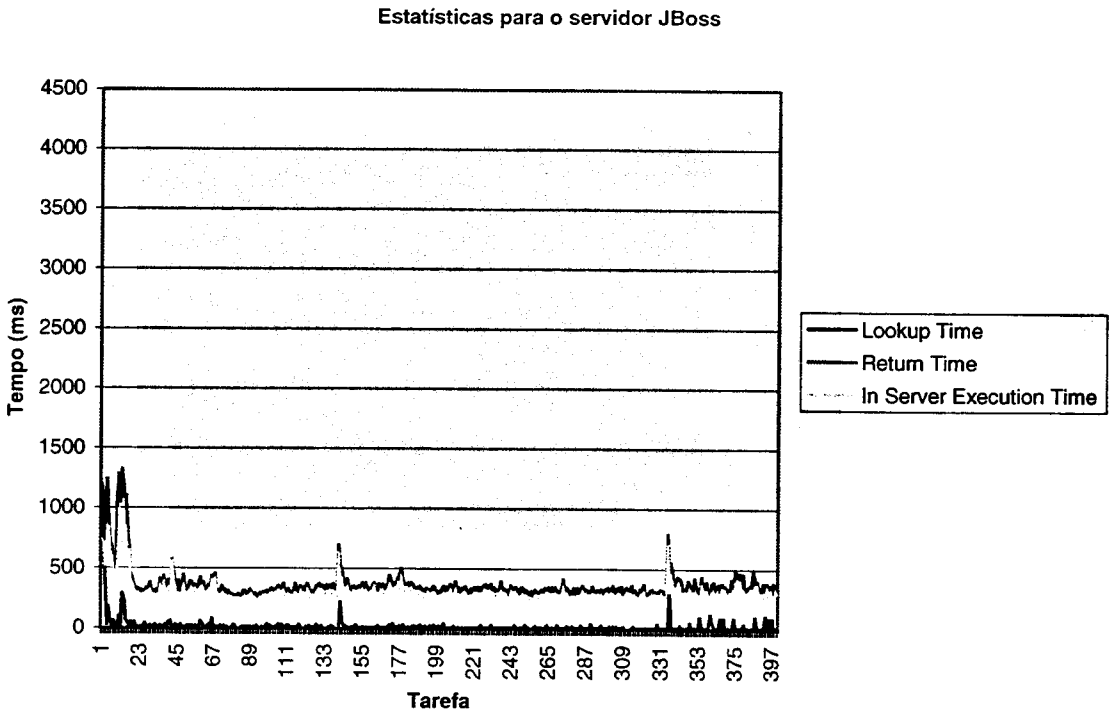
Os testes foram feitos em uma máquina com a seguinte configuração:

- AMD K6 2 450
- 128 MB de memória RAM
- Banco de dados: SQLServer 7.0
- Rede de 10 Mbits

Foram analisados os seguintes itens de performance:

- *Lookup time*: tempo gasto para a localização de uma instância da home interface do session bean a ser invocado através do serviço JNDI do servidor;
- *Return time*: tempo gasto desde o início da invocação da tarefa até que a mesma retorne os resultados para o cliente;
- *In Server Execution Time*: tempo gasto pelo método implementado pelo bean para ser executado. A diferença entre o *return time* e o *in server execution time* corresponde ao tempo gasto com a sobrecarga que o servidor adiciona ao código do *bean* (controle de transações, persistência, segurança, etc.) e ao tempo gasto com o mecanismo de *marshaling* de parâmetros pelas chamadas remotas.





Ao analisarmos o gráfico para o servidor Inprise Application Server podemos notar que a demora inicial no serviço de busca corresponde ao tempo gasto pelo cliente para localizar o servidor de

aplicações. Isso ocorre devido ao fato de o servidor se utilizar de um serviço *CORBA* para a implementação do *JNDI*, e esse tempo inicial perdido corresponde à inicialização do *ORB* [8] pelo cliente. Com o passar do tempo, os tempos de busca diminuem consideravelmente, devido ao fato de a localização do servidor já ser conhecida. Para o caso do servidor Jboss, os tempos de busca são pequenos devido ao fato de que o cliente precisa ter conhecimento da localização do servidor, o que torna as buscas iniciais mais rápidas do que o do Inprise Application Server.

Os tempos de execução altos do início correspondem ao tempo em que os *pools* de conexões de banco de dados e de *threads* estão em seu estágio inicial, com poucos recursos disponíveis para atender a demanda inicial das tarefas. Conforme o tempo passa os *pools* se estabilizam e o tempo de resposta de ambos os servidores diminuem a um mesmo patamar, indicando a presença dessa implementação.

Outra conclusão que pode ser obtida nos gráficos se refere a diferença no tempo de resposta da execução do servidor Inprise Application Server em comparação ao Jboss. A diferença nos tempos de resposta nos leva a concluir que o servidor da Inprise dispõe de mecanismos mais robustos que o Jboss. As implementações da Inprise se mostraram mais eficientes, principalmente no que toca a parte de persistência (algoritmos de acesso ao banco) e na parte de chamadas remotas de métodos.

Fazendo uma análise geral podemos concluir que o servidor da Inprise é mais recomendado que o Jboss para situações em que é necessário um menor tempo de resposta para a execução das tarefas. O Jboss pode ser interessante para ser usado em projetos simples ou para a implementação de protótipos de projetos maiores.

Conclusões

O sucesso da tecnologia Java no mercado se deve principalmente ao fato de que todas as especificações e padrões são abertos a toda a comunidade, que passa a se tornar capaz de ajudar na proposição de melhorias e de novos padrões. O surgimento de um amplo mercado para os servidores de aplicação é a maior prova do sucesso dessa abordagem.

Todas as métricas levantadas nesse trabalho são capazes de fornecer a base necessária para a realização de testes comparativos entre servidores de aplicação, auxiliando no processo de tomada de decisão sobre qual servidor atende aos requisitos de um sistema a ser desenvolvido.

As métricas levantadas também podem ser usadas como base para a validação de eventuais implementações da especificação. Basta executar os testes e comparar os dados levantados com os dados de implementações já consolidadas no mercado. As baterias de teste criadas neste trabalho serão efetivamente usadas para a validação de uma implementação da especificação sendo desenvolvida pelo autor deste trabalho, a fim de se poder avaliar o quão próximo à mesma se encontra dos principais produtos do mercado.

Referências

[1] "Enterprise JavaBeans Specification Version 1.1", Sun Microsystems Inc., 1999

[2] Monson-Haefel, R., "Enterprise JavaBeans, 2nd Edition", O'Reilly & Associates, 2000

[3] Narayanan, S., Liu, J., "Enterprise Java Developer's Guide", McGraw Hill, 1999

[4] Roman, E., "Mastering EJB and the Java 2 Plataform, Enterprise Edition", John Wiley, 1999

[5] Ambler, S. W., "The Design of a Robust Persistence Layer For Relational Databases", 2000

[6] Rosemberg, D.; Scott, K.; "Use Case Driven Object Modeling With UML: A Pratical Approach"; Addison-Wesley; 1999

[7] Taylor, A., "JDBC Developer's Resource 2nd Edition", Prentice Hall PTR

[8] Vogel, A., Duddy, K., "Java Programming with CORBA: Advanced Techniques for Building Distributed Applications, Second Edition", John Wiley & Sons, Inc., 1998

[9] MacLaughlin, B., "Java and XML", O'Reilly, 2000

[10] Web – http://www.jboss.org

[11] Web – http://www.borland.com/appserver

[12] Web – http://www.javasoft.com

[13] Web – http://www.javaworld.com