UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FÁBIO NOGUEIRA DE SOUZA

"AVALIAÇÃO DE DESEMPENHO DE SERVIDORES DE APLICAÇÃO UTILIZANDO REDES DE PETRI"

RECIFE, JULHO/2006

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FÁBIO NOGUEIRA DE SOUZA

"AVALIAÇÃO DE DESEMPENHO DE SERVIDORES DE APLICAÇÃO UTILIZANDO REDES DE PETRI"

ESTE TRABALHO FOI APRESENTADO À PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO REQUISITO PARCIAL PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIA DA COMPUTAÇÃO.

ORIENTADOR: NELSON SOUTO ROSA
CO-ORIENTADOR: PAULO ROMERO MARTINS MACIEL

RECIFE, JULHO/2006

Souza, Fábio Nogueira de
Avaliação de desempenho de servidores de aplicação utilizando redes de Petri. / Fábio Nogueira de Souza. – Recife: O autor, 2006.
124 p.: il., fig.,tab., quadros.

Dissertação (mestrado) – Universidade Federal de Pernambuco. CIN. Ciência da Computação, 2006.

Inclui bibliografia, apêndice e glossário.

1.Sistemas distribuídos. 2. Avaliação de desempenho. 3. Redes de Petri. 4. Servidores de aplicação.

004.36 CDD (22.ed.) MEI2006-009

Agradecimentos

Aos meus pais, Denize e Francisco José, pelo imenso amor, carinho e dedicação demonstrados ao longo de toda a minha existência. Agradeço também pelo exemplo de vida e por me ensinarem que a educação é a riqueza mais importante que um ser humano pode ter.

À minha esposa Andrea, cuja simples existência ilumina os meus caminhos, pelo amor, apoio e compreensão presentes em todos os momentos no decorrer desta importante jornada. Ao meu filho Thiago, cujo sorriso afasta de mim qualquer tristeza, por me fazer renascer e por me ensinar que nossa vida não se esvai a cada dia, mas sim se renova!

Aos meus irmãos Bruno, Gabriel, Vinicius, Pedro e Gabriela pelo carinho, amizade, respeito e apoio demonstrados. À minha avó Nanci, *in memorian*, pelo amor e carinho, e por representar para mim um exemplo de garra e sabedoria. À minha tia Íris e aos meus primos Pedro e Juliana por me acolherem em sua casa e me fazerem sentir que ela sempre foi um pouco minha também.

Aos demais familiares por me mostrarem que a família é muito mais do que pais e irmãos fazendo-se sempre presentes em minha vida. À meu sogro José Vasconcelos, *in memorian*, e à minha sogra Luisa Carmen por me receberem como um filho.

Aos meus amigos Roberto Arteiro e Francisco Madeiro por demonstrarem o real sentido da palavra amizade e por todo o suporte sem o qual esta jornada seria, sem dúvidas, muito mais difícil.

Ao meu orientador, professor Nelson Rosa, pelo incentivo e pelos conhecimentos transmitidos. Aos professores Edna Barros, Paulo Borba, Fernando Buarque e Aparecido Jesuíno pelo apoio nos meus passos iniciais ao longo deste caminho.

Ao Banco Central do Brasil pelo suporte sem o qual a realização deste sonho não seria possível.

Resumo

A plataforma J2EE (Java 2 Platform Enterprise Edition) especifica um ambiente de suporte ao desenvolvimento e execução de aplicações corporativas distribuídas. Diversos fornecedores oferecem implementações desta especificação, as quais são referenciadas como servidores de aplicação J2EE. Este mercado concorrido gera um novo desafio para departamentos de TI que compreende a avaliação e seleção de um produto J2EE adequado. A escolha de uma implementação é uma tarefa árdua uma vez que envolve uma avaliação conjunta de muitas características, tais como custo, desempenho, escalabilidade, flexibilidade e adaptabilidade, facilidades de uso e de gerenciamento, e conformidade com os padrões. Contudo, essa escolha é usualmente centrada em desempenho, uma vez que avaliar servidores de aplicação com base em todas as suas características é difícil.

As abordagens atuais para avaliação de desempenho de servidores de aplicação têm se concentrado principalmente no uso da técnica de medição. Este trabalho, por sua vez, propõe a utilização de modelagem analítica e simulação, introduzindo uma abordagem para a avaliação de desempenho baseada em redes de Petri estocásticas. Para ilustrar a proposta, modelos para o servidor de aplicação JBoss são projetados e utilizados na derivação de métricas de desempenho. Os resultados obtidos são validados através da realização de medições em ambientes controlados.

Palavras-chave: Avaliação de desempenho, Servidores de aplicação, J2EE e Redes de Petri

Abstract

The J2EE (Java 2 Platform Enterprise Edition) specification defines an environment for supporting the development and deployment of distributed enterprise applications. Numerous product vendors offer implementations of this specification which are referred to as J2EE application servers. This crowded marketplace creates a new challenge for IT departments which involves the evaluation and selection of an adequate J2EE product. Choosing an implementation is a hard task since it involves evaluating many features, such as cost, performance, scalability, flexibility and adaptability, ease of use and management, and standards conformance. However, that choice is usually focused on performance, since evaluating application servers based on all their qualities is too hard.

Current approaches to assess the performance of application servers have mainly focused on the use of the measurement technique. The present work, by its turn, proposes the use of analytical modeling and simulation introducing an approach for applying stochastic Petri nets for performance evaluation of application servers. To ilustrate how the proposed approach may be applied, Petri nets models for the JBoss application server are designed and used for deriving performance metrics. Results obtained by the designed models are corroborated by results obtained from measurements conducted in controlled environments.

Keywords: Performance evaluation, Application Servers, J2EE, Petri nets.

Índice

Lista de Figuras

Lista de Tabelas

Glossário

CORBA	*Commom Object Request Broker Architecture*
CSIRO	*Commonwealth Scientific and Industrial Research Organisation*
DSPN	*Deterministic and Stochastic Petri Nets*
EJB	*Enterprise JavaBeans*
GSPN	*Generalized and Stochastic Petri Nets*
J2EE	*Java 2 Platform Enterprise Edition*
JSP	*Java Server Pages*
LQN	*Layered Queueing Networks*
NPFQN	*Non-Product-Form Queueing Networks*
PFQN	*Product-Form Queueing Networks*
QN	*Queueing Networks*
RMI	*Remote Method Invocation*
SPN	*Stochastic Petri Nets*

CAPÍTULO

1

Introdução

"I keep six honest serving men. They taught me all I knew. Their names are What and Why and When and How and Where and Who".

Rudyard Kipling.

O objetivo deste capítulo é introduzir o presente trabalho no contexto da área de avaliação de desempenho de *middleware*. Inicialmente são apresentados o contexto e a motivação para este trabalho, incluindo a função do *middleware* no suporte ao desenvolvimento de aplicações corporativas e o uso da plataforma J2EE [Sun 2003b] e suas implementações (servidores de aplicação) como uma das principais plataformas de *middleware* da atualidade. Em seguida, são relacionados diversos aspectos relevantes considerados na avaliação de servidores de aplicação, ressaltando-se, neste contexto, o papel do desempenho. Na seqüência, os principais desafios e técnicas envolvidos nos projetos de avaliação de desempenho de servidores de aplicação são apresentados. Por fim, são apresentadas a proposta deste trabalho e a estrutura da dissertação

1.1 Contexto e Motivação

A evolução da tecnologia a taxas aceleradas tem levado a uma demanda por aplicações corporativas cada vez mais complexas. Tais aplicações são, em geral, inerentemente distribuídas e seus componentes executam em ambientes heterogêneos. Um requisito freqüente destas aplicações é a necessidade de comunicação e integração com outras aplicações desenvolvidas, muitas vezes, em tecnologias diversas. Além disto, as aplicações corporativas normalmente

possuem requisitos não-funcionais de segurança, suporte a transações distribuídas, desempenho, disponibilidade e confiabilidade, dentre outros.

Diante desse cenário complexo, surge a necessidade de uma nova categoria de software que forneça uma infra-estrutura de suporte às aplicações corporativas. Os sistemas de software pertencentes a esta categoria, referenciados como plataformas de *middleware*, fornecem às aplicações uma abstração dos mecanismos de comunicação e possibilitam que estas tenham acesso aos serviços disponibilizados pelo seu ambiente de execução, sem que, contudo, seja necessário um conhecimento dos detalhes de baixo-nível envolvidos. Desta forma, as plataformas de *middleware* fornecem suporte à implementação dos requisitos não-funcionais, permitindo que as equipes de desenvolvimento se concentrem, efetivamente, nos problemas relacionados ao negócio.

Um conceito relacionado à concepção de *middleware* apresentada acima é o de servidores de aplicação. Segundo [Gartner 2005], os usuários qualificam como servidores de aplicação os sistemas de software que suportam a execução de suas aplicações de negócio e que fornecem ferramentas e bibliotecas que enriquecem a sua programação. A definição apresentada para servidores de aplicação corporativos compreende os ambientes de tempo de execução associados a várias plataformas de *middleware* disponíveis.

[Gartner 2005] apresenta, ainda, um levantamento dos principais servidores de aplicação disponíveis. Dentre os vinte e oito produtos pesquisados, aqueles oferecidos pela IBM, BEA, Oracle, Microsoft e JBoss são considerados os líderes e principais influenciadores do mercado. Dentre os cinco líderes apontados, apenas o produto fornecido pela Microsoft não corresponde à uma implementação da especificação *Java 2 Platform, Enterprise Edition* (J2EE), ressaltando a importância desta plataforma para o mercado corporativo.

Por outro lado, a melhor escolha de um servidor de aplicação, nem sempre corresponde à simples seleção do produto que detém a maior fração do mercado. A diversidade de servidores de aplicação J2EE disponíveis torna a escolha técnica de uma implementação uma tarefa complexa, visto que diversos parâmetros devem ser considerados e avaliados. Dentre os parâmetros, destacam-se: custo, desempenho, escalabilidade, flexibilidade, facilidades de uso e de gerenciamento e conformidade com os padrões estabelecidos. Uma avaliação criteriosa e conjunta destes parâmetros é uma tarefa complexa. Primeiro, o número de parâmetros é grande e alguns deles podem variar amplamente. Segundo, há parâmetros conflitantes, como flexibilidade e desempenho. Por fim, alguns parâmetros são objetos de avaliação subjetiva. Estas dificuldades têm restringido a avaliação a apenas dois parâmetros: custo e desempenho [Vinoski 2003]. Este último constitui o objeto de interesse deste trabalho.

1.2 Caracterização do Problema

Considerando o papel primordial do aspecto de desempenho na análise de servidores de aplicação, torna-se necessário definir uma abordagem que permita

sistematizar o seu processo de avaliação. Fundamentalmente, esta abordagem deve fornecer uma resposta para a seguinte questão:

- Como avaliar, de forma precisa, o desempenho de servidores de aplicação?

Embora este pareça um questionamento simples, existem alguns aspectos importantes a serem considerados, tais como:

- Que serviços possuem maior impacto no desempenho global de um servidor de aplicação?
- Quais os critérios de desempenho devem ser considerados na realização de uma avaliação?
- Quais parâmetros relacionados ao ambiente possuem impacto no desempenho e podem ser ajustados com vistas à sua otimização?
- Como avaliar o servidor de forma independente das aplicações nele instaladas? Ou ainda, como minimizar o impacto de uma aplicação na avaliação de desempenho do servidor?
- Qual (ou quais) a técnica mais adequada para avaliar o desempenho de servidores de aplicação?

Abordagens sistemáticas para avaliação de desempenho devem levar em consideração cada um dos aspectos mencionados acima e estabelecer seqüências de atividades a serem realizadas para a obtenção dos resultados de desempenho. Uma vez estabelecida a abordagem, esta pode ser utilizada em projetos de avaliação de desempenho realizados com as mais distintas finalidades, tais como seleção e aquisição de produtos, identificação de gargalos, realização de *tunning*, planejamento de capacidade e predição de desempenho.

1.3 Estado da Arte

Três diferentes técnicas podem ser usadas em avaliações de desempenho: medição, modelagem analítica e simulação. A técnica de medição consiste, basicamente, no estabelecimento de critérios de desempenho e na sua aferição diretamente no sistema em execução. Esta técnica tem sido largamente utilizada na avaliação de desempenho de servidores de aplicação. [Cecchet et al. 2002] apresenta uma análise do impacto das arquiteturas das aplicações e dos *containers* no desempenho global de um sistema Enterprise JavaBean (EJB) [Sun 2003a]. [CSIRO 2002] avalia e compara diferentes servidores de aplicação, apresentando uma análise qualitativa de suas características e uma análise quantitativa baseada em testes de desempenho realizados usando uma aplicação de *benchmark* especialmente desenvolvida para este fim. [Kounev et al. 2004] usa a aplicação de *benchmark* SPECjAppServer2004 para explorar o efeito de diferentes parâmetros de configuração no desempenho do servidor de aplicação JBoss. [TPC 2005] especifica um *benchmark* padrão para servidores de aplicação, que é referenciado como TPC *Benchmark*™ App (TPC-App).

A despeito de sua larga utilização, a técnica de medição tem importantes limitações. Primeiramente, sua alta sensibilidade a variações em parâmetros relacionados ao ambiente pode comprometer a precisão dos resultados. Além disto, experimentos de medição requerem a construção e configuração de ambientes separados. Os custos associados aos equipamentos, ferramentas e tempo necessários à configuração desses ambientes podem ser muito altos.

A modelagem analítica pode derivar métricas de desempenho rapidamente, fornecendo informações valiosas relativas ao desempenho de um sistema sem os custos extras decorrentes da replicação do ambiente real. [Lladó e Harrison 2000] apresenta um modelo analítico, baseado na teoria de filas, para um servidor de aplicação idealizado. Utilizando este modelo diferentes tendências podem ser inferidas. Contudo, os resultados de desempenho obtidos não foram validados a partir de servidores de aplicação reais. [Kounev e Buchmann 2003] utiliza *Non-Product-Form Queueing Networks* para propor um modelo analítico para um *cluster* de servidores de aplicação executando a aplicação de *benchmark* SPECjAppServer2002. A despeito dos resultados impressionantes obtidos para CPU e *throughput*, o modelo proposto não é capaz de realizar predições precisas para o tempo de resposta uma vez que a competição por recursos de software não é representada. [Liu et al. 2004] propõe uma abordagem para predição de desempenho de aplicações de aplicações ainda a nível de projeto. Esta predição é realizada com base em um modelo em redes de fila que é configurado com parâmetros relacionados à carga e ao próprio servidor de aplicação utilizado. Este trabalho é particularmente promissor, uma vez que ele fornece uma previsão do desempenho de uma aplicação antes desta estar completamente desenvolvida. Um ponto importante a ser considerado, contudo, é que algumas métricas importantes, tais como as relacionadas à utilização de recursos não podem ser derivadas.

Contudo, modelos analíticos, em geral, requerem muitas simplificações e suposições de forma que é difícil a obtenção de resultados precisos [Jain 1991]. Outro problema concernente a algumas ferramentas utilizadas na modelagem analítica é conhecido como explosão de estados. Este problema ocorre devido ao crescimento exponencial no número de estados que está relacionado ao tamanho do modelo. O crescimento exponencial no número de estados implica em um aumento exponencial no número de equações matemáticas que representam o sistema e, conseqüentemente, dificulta ou inviabiliza o cálculo das métricas de desempenho, uma vez que os recursos computacionais são rapidamente esgotados. Em particular, o problema da explosão de estados pode representar um importante limitador para o uso de modelos analíticos na representação de servidores de aplicação, uma vez que estes são sistemas de software grandes e complexos. A viabilidade da utilização destes modelos depende fundamentalmente da capacidade de abstração do projetista.

Uma vez que os modelos de simulação são construídos para serem executados, e não para serem solucionados, eles podem incorporar mais detalhes e, como conseqüência, menos suposições do que os modelos analíticos. Desta forma, estes modelos podem, teoricamente, ser mais flexíveis e se aproximar mais do comportamento real dos sistemas. [McGuinness e Murphy 2005] apresenta um modelo de simulação de um *cluster* de servidores de aplicação que pode ser parametrizado com dados descrevendo as interações com usuários e com características dos servidores que compõem o cluster. Simulações deste modelo produzem informações acerca de diversas métricas de desempenho, incluindo tempo médio de execução, *throughput* e utilização de CPU e I/O. Embora o modelo proposto seja bastante flexível, o trabalho não apresenta uma comparação dos resultados obtidos através dos experimentos de simulação com resultados obtidos a partir de medições em algum servidor de aplicação.

A despeito das pontencialidades advindas do uso da técnica de simulação, é importante destacar que modelos mais elaborados normamente requerem muito tempo para a realização de execuções bem sucedidas. Diante deste contexto, avaliações envolvendo diversos cenários diferentes podem requerer um tempo muito grande, dificultando a utilização desta técnica de avaliação de desempenho.

Em suma, cada técnica possui características positivas e negativas tornando difícil a seleção de uma técnica única a ser adotada em todos os cenários. Desta forma, uma recomendação usual consiste em avaliar o desempenho de sistemas combinando mais de uma técnica de forma que os resultados assim obtidos possam ser validados.

1.4 Objetivos e Contribuições

O principal objetivo deste trabalho é demonstrar a viabilidade de utilização das redes de Petri Estocásticas na avaliação de desempenho de servidores de aplicação. Em particular, os modelos desenvolvidos neste trabalho são baseados em *Generilized and Stochastic Petri Nets* (GSPN) [Marsan et al. 1984] e em *Deterministic and Stochastic Petri Nets* (DSPN) [Marsan e Chiola 1987].

Na escolha das redes de Petri como ferramenta de modelagem os seguintes aspectos foram considerados:

- Natureza estocástica dos modelos desenvolvidos: as redes de Petri estocásticas possuem uma equivalência com as cadeias de Markov, mas são mais compactas na representação de sistemas, o que possibilita a avaliação de desempenho através de modelos de alto-nível;
- Possibilidade de projeto de modelos visuais;
- Suporte inerente às técnicas analíticas e de simulação; e
- Amplo suporte de ferramentas.

Para validar a abordagem proposta, são projetados modelos de desempenho para um servidor de aplicação J2EE específico: o *JBoss Application Server* [JBoss 2004]. Inicialmente são definidos modelos do subsistema responsável pelo gerenciamento de componentes *Enterprise JavaBeans* (EJB) [Sun 2003a], destacando, em particular, o serviço de *pooling* de instâncias, que representa um importante fator do ponto de vista de desempenho de servidores de aplicação. Em seguida, os modelos projetados são validados através da realização de experimentos de simulação e/ou de análise e da comparação dos resultados obtidos com resultados de medições realizadas diretamente no servidor de aplicação.

1.5 Estrutura da Dissertação

O restante deste trabalho está estruturado em cinco capítulos.

O Capítulo 2 introduz os conceitos básicos necessários ao entendimento deste trabalho. Com este intuito, são apresentados o conceito de *middleware*, a plataforma J2EE e seus componentes, o servidor de aplicação JBoss, e as

principais técnicas e métricas disponíveis para os projetos de avaliação de desempenho.

No Capítulo 3 é feita uma análise detalhada do estado da arte. Os principais trabalhos na área de avaliação de desempenho de servidores de aplicação são apresentados e avaliados com relação ao que está sendo proposto.

O Capítulo 4 apresenta a abordagem proposta para modelagem de desempenho de servidores de aplicação, detalhando as atividades realizadas e os resultados esperados de cada uma delas. O Capítulo 5 utiliza esta abordagem construção de modelos de desempenho para o servidor de aplicação JBoss. Nesta seção, o processo de construção e validação dos modelos é detalhado, culminando com a proposição de diferentes modelos representando diferentes níveis de abstração. Os modelos validados são utilizados na realização de experimentos com vistas à predição de desempenho do servidor de aplicação dada uma carga variável submetida.

Por fim, o Capítulo 6 apresenta as conclusões da dissertação, ressaltando as contribuições à área de avaliação de desempenho de servidores de aplicação, e os possíveis trabalhos futuros.

CAPÍTULO

2

Conceitos Básicos

"Measurements are not to provide numbers but insight".

Ingrid Bucher.

Neste capítulo são descritos os componentes básicos definidos pela plataforma J2EE e o ambiente de suporte a estes componentes referenciados como servidores de aplicação. Na seqüência, o servidor de aplicação JBoss e os seus principais elementos são discutidos, com ênfase nos elementos relacionados ao gerenciamento de componentes EJB. Em seguida, são apresentados os conceitos fundamentais associados à área de avaliação de desempenho, destacando-se a utilização de modelos de simulação e análise. Por fim, são apresentadas as Redes de Petri Estocásticas que constituem a base dos modelos de desempenho desenvolvidos ao longo deste trabalho

2.1 Middleware e J2EE

A natureza inerentemente distribuída dos sistemas corporativos (*enterprise systems*) apresenta uma série de desafios para as equipes de desenvolvimento que não estão presentes na construção de sistemas monolíticos. Dentre estes desafios, o principal diz respeito à heterogeneidade de hardware, sistemas operacionais, redes, e linguagens de programação. Visando simplificar o desenvolvimento desta classe de sistemas surgiu uma nova categoria de software chamada *middleware*. O middleware, como o próprio sugere, é uma camada intermediária entre a aplicação e o sistema operacional que fornece serviços de forma independente do ambiente operacional [Bernstein 1996] [Campbell et al. 1999]. Dentre os serviços normalmente fornecidos por um *middleware* pode-se citar os serviços de comunicação, nomes e diretórios,

transação, persistência, concorrência e segurança. O grau de suporte a estes serviços varia de um *middleware* para outro de acordo com a sua complexidade e abrangência.

O CORBA (*Common Object Request Broker Architecture*) [OMG 2002] foi uma das primeiras plataformas de *middleware* propostas, e certamente a mais completa. Este *middleware* estabelece uma série de serviços padrões e uma arquitetura independente de plataforma e de linguagem, de forma a simplificar o desenvolvimento de sistemas distribuídos construídos em ambientes heterogêneos.

Em Java, as tecnologias de *middleware* surgiram através do suporte direto à plataforma CORBA e das especificações *Remote Method Invocation* (RMI) [Sun 2004], *Java 2 Platform Enterprise Edition* (J2EE) [Sun 2003b] e *Enterprise JavaBeans* (EJB) [Sun 2003a].

A especificação de RMI introduziu na plataforma Java um *middleware* simplificado cujo escopo está basicamente restrito a fornecer uma abstração dos mecanismos de comunicação, viabilizando a construção de aplicações distribuídas. Posteriormente, uma nova versão de RMI foi especificada permitindo o desenvolvimento de objetos Java que podem interagir diretamente com componentes CORBA, eventualmente desenvolvidos em outras plataformas.

A especificação J2EE provê uma abordagem baseada em componentes para o projeto, desenvolvimento, montagem e instalação de aplicações corporativas. De acordo com [Sun 2005], um componente J2EE é uma unidade funcional e autocontida de software utilizada na montagem de aplicações J2EE e é capaz de se comunicar com outros componentes. Os componentes definidos na especificação podem ser de quatro tipos diferentes:

- *Aplicações cliente*: programas que normalmente possuem uma interface gráfica e que executam em máquinas clientes;
- *Applets*: componentes instalados e gerenciados no servidor, mas cuja execução se dá na máquina cliente, tipicamente em máquinas virtuais executando em navegadores web;
- *Servlets* e *Java Server Pages* (JSP): componentes web instalados, gerenciados e executados no servidor, os quais são tipicamente responsáveis pela lógica de controle e de apresentação das aplicações. O acesso a estes componentes ocorre através do envio de requisições HTTP (Hyper Text Transfer Protocol).
- *Enterprise JavaBeans* (EJB): componentes responsáveis pela implementação da lógica de negócio. Assim como os componentes web, os componentes EJB também são instalados, gerenciados e executados no servidor.

Para cada tipo de componente previsto, a plataforma J2EE especifica um *container* correspondente. Os *containers* são ambientes de execução controlados, que funcionam como interfaces padrão entre os componentes e o mundo exterior. Cada *container*, além de ser responsável pelo gerenciamento dos componentes nele instalados, fornece a estes um conjunto de serviços configuráveis previstos na especificação.

Com base na especificação J2EE, várias implementações foram desenvolvidas e são referenciadas como servidores de aplicação. É importante observar que, como a especificação só determina o comportamento aparente desta tecnologia, servidores de aplicação distintos possuem diferentes características de desempenho, escalabilidade, interoperabilidade, entre outras. A Figura 2.1 ilustra os conceitos de componentes J2EE, *containers* e servidores de aplicação.

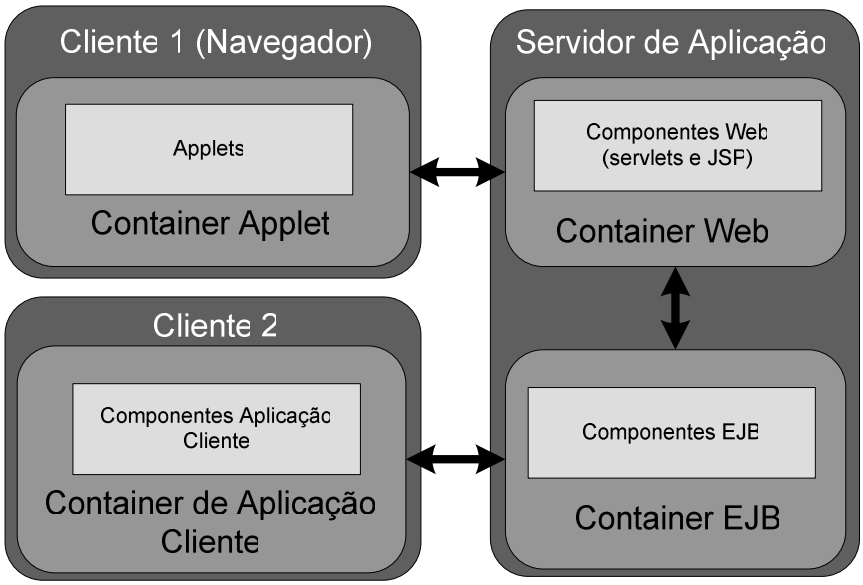


Figura 2.1. Componentes, *Containers* e Servidores de Aplicação.

Para que os componentes possam ser executados em *containers*, é necessário que eles sejam empacotados e instalados. O processo de empacotamento compreende a montagem de módulos J2EE. Cada um destes módulos consiste de um ou mais componentes destinados a um mesmo tipo de *container*, e de descritores de instalação (*deployment descriptors*) [Sun 2005]. Um descritor de instalação é um arquivo XML que descreve as configurações necessárias à correta instalação dos componentes no *container* correspondente.

Normalmente, cada módulo J2EE possui dois descritores associados. Um deles é padrão para o tipo de *container* em questão e é detalhado na própria especificação da plataforma J2EE. Este descritor é suportado por qualquer servidor de aplicação compatível com esta plataforma e permite a configuração de opções padrões relativas à instalação dos componentes. Por outro lado, o descritor não-padrão permite a configuração de recursos específicos para um dado servidor de aplicação.

2.1.1 *Enterprise JavaBeans*

O ponto central da especificação J2EE é a definição de um *framework* para o desenvolvimento de componentes de negócio conhecidos como *Enterprise JavaBeans* (EJB). Conforme mencionado, instâncias de componentes EJB, executam dentro de um *container* EJB que é responsável pelo gerenciamento do seu ciclo de vida e por fornecer serviços configurados de forma declarativa. De acordo com a especificação EJB 2.1 [Sun 2003a], existem três tipos de componentes EJB: *session beans*, *entity beans* e *message-driven beans*.

Session beans são componentes não persistentes normalmente utilizados para representar processos de negócio. Estes componentes não são compartilháveis, significando que, em qualquer instante, cada *session bean* representa um único cliente. Os *session beans* podem ser de dois tipos: *statefull* ou *stateless*. Os *session beans statefull* mantém seu estado entre requisições consecutivas de um mesmo cliente, enquanto que os *session beans stateless* não mantém o estado entre requisições.

Entity beans são componentes compartilháveis e persistentes utilizados para representar os dados de negócio. Estes *beans* podem ser de dois tipos: *Container Managed Persistence* (CMP), cuja responsabilidade pela persistência é delegada ao *container*; e *Bean Managed Persistence* (BMP), responsáveis por implementar sua própria persistência.

É importante mencionar que *session beans* e *entity beans* respondem de forma síncrona às requisições que recebem. O suporte a componentes capazes de processar requisições de forma assíncrona foi embutido na especificação EJB com a inclusão dos *message-driven beans*. Estes *beans* são acionados através de envio de mensagens para filas com as quais estes estão associados.

2.1.2 EJB in JBoss

Na arquitetura do JBoss [Fleury e Reverbel 2003] [JBoss 2004], uma aplicação cliente (local ou remota) tem acesso a um componente EJB através de um *proxy*, o qual é obtido através do serviço de nomes e diretórios e expõe os mesmos métodos que são expostos pelo próprio componente. Cada *proxy* possui uma cadeia de interceptadores, que é responsável por coletar informações necessárias à realização dos serviços.

Quando uma aplicação cliente invoca um dos métodos do *proxy*, este coleta informações sobre o método invocado, armazena-as em um objeto chamado *invocation* e encaminha este objeto para o primeiro interceptador de sua cadeia. Este interceptador obtém informações sobre o contexto no qual o método foi acionado (por exemplo, informações sobre o usuário realizando a requisição), adiciona esta informação ao objeto *invocation* e encaminha este objeto para o próximo interceptador da cadeia. Desta forma, cada interceptador é acionado pelo interceptador anterior, processa a invocação e a encaminha para o interceptador seguinte.

O último interceptador, conhecido como *invoker interceptor*, é responsável por encaminhar o objeto *invocation* representando a requisição para o componente no servidor chamado *invoker*. Se o cliente e o servidor estiverem executando na mesma máquina virtual, o objeto *invocation* é simplesmente passado por referência, caso contrário, ele é serializado e transferido através da rede para o lado servidor.

Do lado servidor, quando um componente EJB é instalado no JBoss, um *container* correspondente é criado. Este *container* é, então, associado a um *invoker* que é responsável por receber requisições de *invoker proxies*, "desserializar" os objetos *invocation* (se necessário) e encaminhá-los para o *container* correspondente.

Como qualquer *container* compatível com a especificação, o *container* implementado pelo JBoss é responsável pelo gerenciamento das instâncias do

componente e por fornecer um conjunto de serviços estabelecido na instalação do componente EJB. Contudo, ao invés de implementar estes serviços, um *container* JBoss possui uma cadeia de interceptadores, onde cada um é responsável pela implementação de um serviço específico.

Quando um *container* EJB recebe um objeto *invocation*, ele encaminha este objeto para o primeiro interceptador da cadeia. Este interceptador usa as informações contidas no objeto *invocation* para realizar o serviço correspondente e, então, encaminha o objeto *invocation* para o próximo interceptador na cadeia. Alguns interceptadores importantes estão usualmente presentes na cadeia de interceptadores do servidor, tais como o interceptador de segurança, de transação e de instância. O interceptador de segurança obtém as informações relativas ao usuário que está acessando o componente e verifica a sua autorização. O interceptador de transação é o responsável pela garantia dos aspectos transacionais. O interceptador de instância é o responsável por obter a instância que será utilizada no processamento da requisição.

Finalmente, o último interceptador da cadeia é o responsável pela identificação do método requerido e pela invocação deste método na instância obtida. O retorno do método percorre a cadeia no sentido inverso até ser recebido pelo *container*, o qual entrega este retorno para o *invoker*. O *invoker*, então, serializa o retorno e o envia para o *invoker proxy*. O retorno recebido pelo *invoker proxy* é desserializado e encaminhado através da cadeia de interceptadores do cliente, também na ordem inversa. Por fim, o valor retornado é encaminhado para a aplicação cliente.

A configuração dos interceptadores utilizados no cliente e no servidor ocorre no momento da instalação do componente e é feita com base nas informações contidas no descritor de instalação do JBoss. A Figura 2 apresenta um *proxy* cliente e um EJB *container* interagindo de forma a atender a uma requisição de usuário.

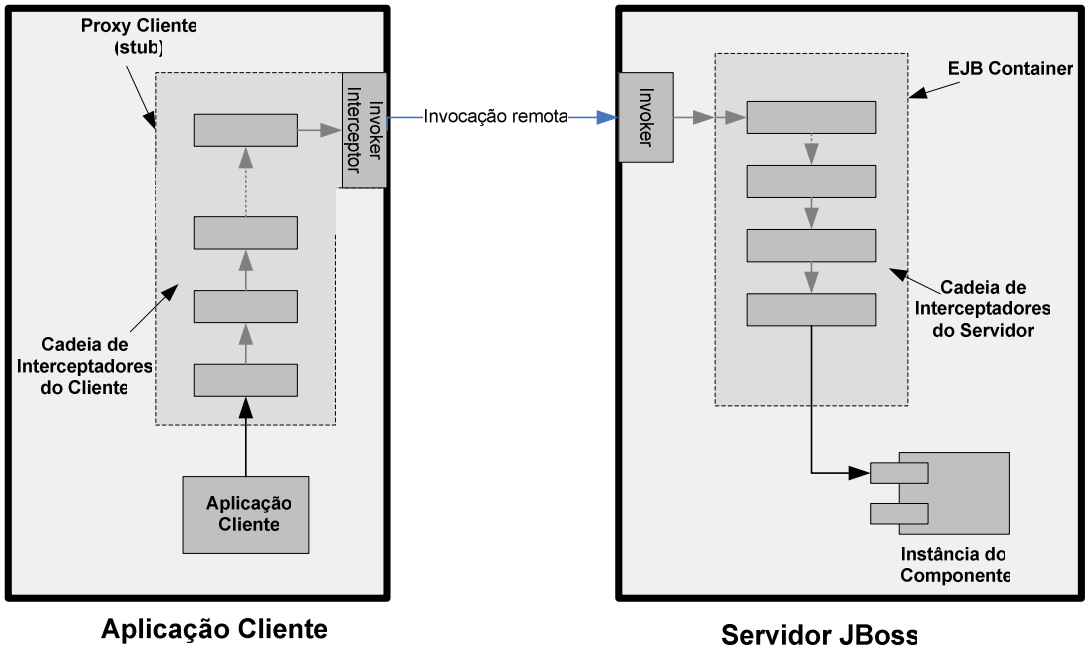


Figura 2.2. Arquitetura do JBoss.

2.1.3 *Pooling* de instâncias no JBoss

Um importante serviço oferecido por um *container* EJB é o *pooling* de instâncias. Quando um *container* EJB intercepta uma requisição de um cliente ele pode criar um novo componente para processar esta requisição. Por outro lado, se existir uma instância criada anteriormente em memória, ela pode ser reutilizá-la, reduzindo o tempo e a memória necessários ao processamento da requisição.

Algumas vezes, um *container* EJB pode reduzir os recursos alocados destruindo as instâncias que não tenham sido usados por um algum tempo. Todo este gerenciamento de instâncias é chamado de *pooling* de instâncias. É importante observar que o mecanismo exato usado para implementar o *pooling* de instâncias não é parte da especificação EJB, de forma que a sua implementação varia de *container* para *container*. Nesta seção, é apresentada a implementação *default* para o mecanismo de *pooling* de instâncias utilizado pelo servidor de aplicação JBoss.

O mecanismo de *pooling* de instâncias do JBoss pode ser configurado utilizando o descritor de instalação específico do JBoss. Os parâmetros configuráveis são: tamanho mínimo e máximo do *pool*, *timeout* e modo de operação.Tamanho máximo/mínimo define o número máximo/mínimo de instâncias que o *pool* pode armazenar (os valores *default* são 100 e 0, respectivamente). Quando a primeira requisição para um componente chega, o *pool* cria e inicializa o número mínimo de instâncias configurado. O *timeout* configurado representa o tempo máximo de espera que uma requisição pode aguardar até que uma instância seja alocada à ela. Caso este *timeout* expire para uma requisição em espera, um erro é gerado para o cliente. Por *default*, o *timeout* configurado para as requisições é extremamente grande, indicando que uma requisição virtualmente não expira.

O *pool* do JBoss pode operar em dois modos diferentes: estrito e não-estrito. No modo não estrito, o *pool* pode criar um número qualquer de instâncias para processar requisições simultâneas, mas pode manter e reusar somente o número máximo configurado. No modo estrito, o *pool* pode criar somente o número máximo de instâncias configurado, não importando o número de requisições simultâneas recebidas.

Quando um interceptador de instância recebe uma requisição, ele deve requisitar uma instância do *pool* para que esta requisição possa ser processada. Se o *pool* tiver instâncias disponíveis, ele retorna uma delas, caso contrário, se o número de instâncias já criadas for menor do que o tamanho máximo configurado, o *pool* cria e retorna uma nova instância. Senão, o comportamento do *pool* depende de seu modo de operação. Se o modo de operação for não estrito, ele cria e retorna uma nova instância ao solicitante, caso contrário, o solicitante terá que aguardar até que uma instância se torne disponível no *pool*. Conforme mencionado, o parâmetro *timeout* define o tempo máximo de espera permitido.

Depois de obter uma instância, o interceptador de instâncias invoca o próximo interceptador na cadeia de interceptadores do *container*. Uma vez que este interceptador conclui sua execução, o interceptador de instâncias encaminha a instância utilizada de volta para o *pool*, o qual apaga todas as

informações relacionadas à requisição recém concluída. Se o *pool* não estiver cheio, ele retém a instância liberada, caso contrário, a instância se tornará elegível para ser removida pelo mecanismo de coleta de lixo.

2.2 Avaliação de Desempenho

O desempenho é um importante critério de qualidade a ser observado no projeto, na aquisição e no uso dos sistemas computacionais. De forma mais sucinta, o desempenho de um sistema pode ser visto como uma medida da sua capacidade de resposta.

Atualmente, os sistemas computacionais vêm assumindo um papel cada vez mais importante no suporte aos aspectos relacionados ao cotidiano das pessoas e empresas. Diante deste cenário, o desempenho dos sistemas tem se tornado um fator de qualidade fundamental, ressaltando a necessidade de se estabelecer métricas (critérios), técnicas e ferramentas de avaliação que permitam a sua análise de forma objetiva e quantitativa.

Infelizmente, diante da diversidade dos tipos de sistemas computacionais existentes e cenários de uso, não é possível o estabelecimento de um conjunto de métricas universais, nem tão pouco, de uma técnica única de avaliação. Desta forma, a seleção das métricas e técnicas utilizadas em um projeto de avaliação de desempenho deve levar em consideração as suas particularidades e objetivos.

As combinações de métricas, técnicas e ferramentas com conjuntos de práticas recomendadas compõem abordagens sistemáticas que podem ser utilizadas em projetos de avaliação de desempenho. Estes projetos objetivam a identificação de gargalos, avaliação de alternativas de projeto e/ou implementação, determinação da configuração ideal para os parâmetros de um sistema (*performance tunning*), previsões de desempenho em função de projeções de carga (*forecasting*) e planejamento de capacidade.

2.2.1 Métricas de Desempenho

A avaliação de desempenho de um sistema deve ser realizada com base em um conjunto de critérios ou métricas. Dentre as métricas de desempenho relevantes, as seguintes são comumente utilizadas [Jain 1991] [Menascé et al. 2004] e são consideradas ao longo deste trabalho:

- **Tempo de Resposta**

Corresponde ao intervalo de tempo entre o envio de uma requisição por um usuário e a chegada da resposta do sistema, sendo usualmente medido em segundos. Para um sistema Web, o tempo de resposta possui dois componentes principais [Menascé et al. 2004]: tempo da rede e tempo de residência no servidor (ver Figura 2.3).

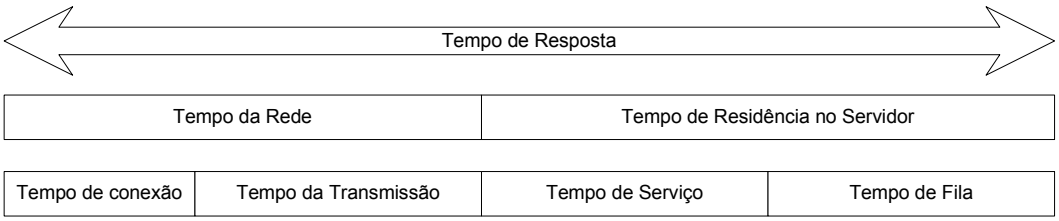


Figura 2.3. Tempo de Resposta de um Sistema Web.

O tempo da rede possui dois componentes, tempo de conexão e tempo de transmissão. O tempo de transmissão compreende o período de tempo necessário à transmissão de todos os pacotes contendo a requisição do usuário e a resposta do sistema. O tempo de conexão compreende os tempos de abertura e fechamento das conexões necessárias.

Por sua vez, o tempo de residência no servidor pode ser decomposto em: tempo de serviço e tempo de fila (ou de espera). O tempo de serviço corresponde ao período de tempo no qual a requisição está recebendo o serviço de algum recurso do servidor, como por exemplo, a CPU. Desta forma, este tempo corresponde ao tempo de processamento efetivo da requisição. Por outro lado, o tempo de fila compreende o período de tempo no qual a requisição está aguardando acesso a algum recurso para que seu processamento possa prosseguir. Os tempos de fila tendem a aumentar com o aumento do número de requisições sendo processadas simultaneamente pelo servidor, provocando um aumento no tempo de resposta.

- **Taxa de Processamento (*throughput*)**

O *throughput* é a taxa na qual as requisições dos usuários são atendidas (processadas) por um sistema computacional, sendo medido em requisições por unidade de tempo. De forma geral, o *throughput* de um sistema inicialmente cresce com a carga a que este está [Menascé et al. 2004]. À medida que a carga aumenta, este crescimento começa a diminuir, deixando de ser linear e indicando que o sistema está ficando saturado. Caso a carga continue crescendo, o *throughput* atingirá seu valor máximo quando um dos recursos do sistema atingir 100% de utilização. Neste ponto, o sistema está completamente saturado de forma que aumentos na carga não provocarão um aumento no *throughput*. De fato, nestes cenários, o *throughput* do sistema pode até mesmo cair como resultado da grande concorrência gerada internamente, que pode, por exemplo, provocar um excesso de operações de *swap* no servidor, deteriorando o desempenho global. Este fenômeno é conhecido como *thrashing*. A Figura 2.4 ([Jain 1991]) apresenta curvas típicas de *throughput* e tempo de resposta em função da carga recebida por um sistema.

O valor teórico máximo (condições ideais) para o *throughput* de um sistema é definido como sendo a sua capacidade nominal, enquanto que a sua capacidade útil corresponde ao *throughput* máximo alcançável.

É importante observar a relação existente entre o tempo de resposta de um sistema e o seu *throughput*. Conforme mencionado, um aumento de carga provoca um aumento linear no *throughput* até que o sistema começa a ficar saturado. Esta saturação também se reflete no tempo de resposta. Um aumento na carga de um sistema provoca um crescimento no tempo de fila das requisições e, conseqüentemente, no seu tempo de resposta. Este crescimento

varia inicialmente de forma linear com a carga e é bem pequeno. Contudo, à medida que a carga continua aumentando, o sistema vai ficando saturado e o crescimento do tempo de resposta se acentua bastante, sem que haja, contudo, um aumento significativo do *throughput*. O ponto ideal de operação do sistema deve ser aquele no qual o *throughput* assume o maior valor dentro dos limites considerados satisfatórios para o tempo de resposta. Usualmente este ponto ótimo é observado no "joelho" de ambas as curvas (ver Figura 2.4) e o *throughput* correspondente é denominado capacidade do joelho.

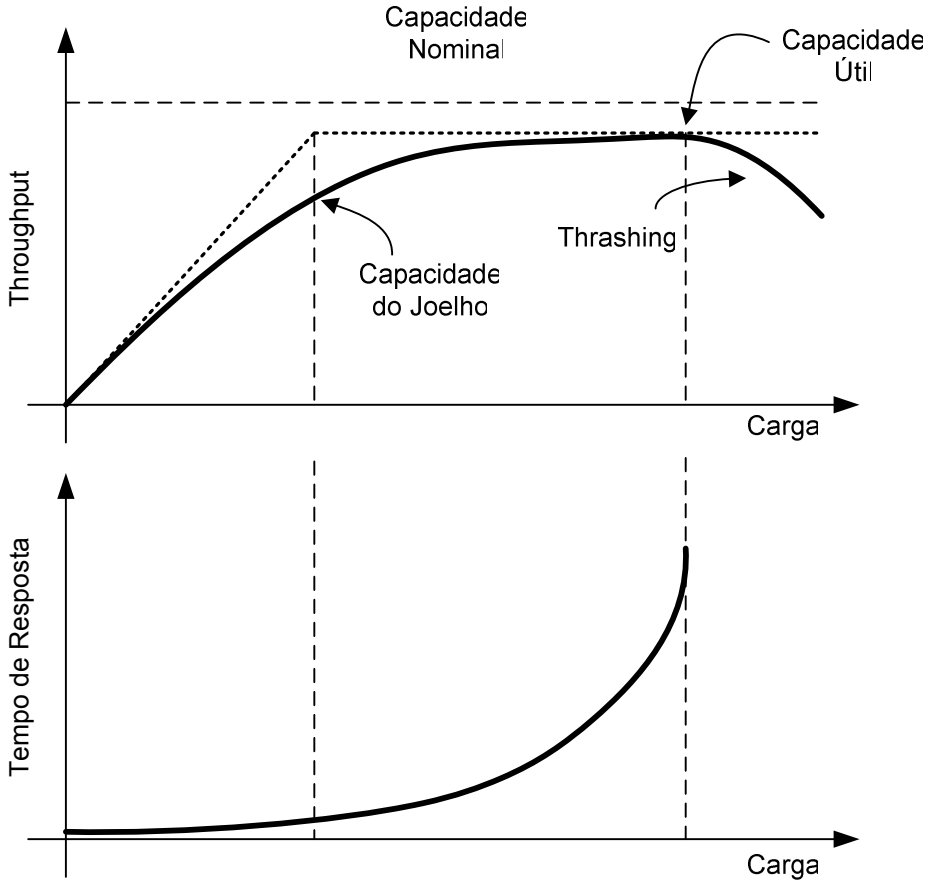


Figura 2.4. Comportamento Típico do *Throughput* e Tempo de Resposta.

Cenários nos quais uma carga elevada produz um aumento descontrolado no tempo de resposta caracterizam a existência de gargalos. Para garantir um tempo de fila máximo (e, conseqüentemente, de resposta mínimo) em um sistema é interessante utilizar os chamados *mecanismos de controle de admissão* (ou de congestionamento) [Menascé et al. 2004]. Estes mecanismos limitam a quantidade máxima de requisições que podem ser concorrentemente processadas pelo sistema a um valor configurável *k*. Ao receber uma nova requisição, o servidor checa o número corrente de requisições em processamento. Se este número for inferior a *k*, a requisição é aceita e o seu processamento é iniciado. Caso contrário, a requisição pode ser rejeitada ou colocada em uma fila de espera, sendo neste último caso, importante a configuração de um tempo máximo de espera na fila (mecanismo *de timeout*).

- **Utilização de um Recurso**

A utilização de um recurso corresponde à fração do tempo na qual este recurso está sendo utilizado para o processamento de requisições. Esta utilização pode ser calculada através da razão entre o período de utilização efetiva do recurso e o intervalo de tempo de observação considerado. Conforme mencionado anteriormente, a saturação de um sistema ocorre à medida que a utilização de um dado recurso se aproxima de 100%. Nesses casos, estes recursos são identificados como *gargalos* do sistema e a sua identificação é uma importante função dos projetos de avaliação de desempenho.

- **Disponibilidade**

A disponibilidade de um sistema é dada pela fração de tempo em que ele está operacional e disponível para seus clientes [Menascé et al. 2004]. Por exemplo, dado um sistema que tem uma disponibilidade de 99%. Durante um ano este sistema não estaria disponível a seus clientes por aproximadamente:

Período de indisp. = (1-0.99) * 1 ano * 365 dias/ano * 24 horas/dia

= 87,6 horas

Um sistema computacional pode se tornar indisponível devido à ocorrência de falhas ou de sobrecargas. Neste contexto, falhas compreendem aspectos de hardware ou de software. A ocorrência de falha em qualquer componente do sistema e/ou de interconexão pode torná-lo inacessível a seus usuários.

A disponibilidade de um sistema também é afetada pela utilização de mecanismos de controle de admissão que podem ocasionar a rejeição de requisições. Conforme mencionado, estes mecanismos podem ser usados para garantir a qualidade de serviço para as requisições em processamento.

2.2.2 Técnicas de Avaliação de Desempenho

Definidos os critérios utilizados na avaliação de desempenho, torna-se necessário a aplicação de técnicas que permitam a avaliação destes critérios para um sistema considerado. Estas técnicas podem ser classificadas como técnicas de medição ou de modelagem.

- **Medição**

Uma abordagem possível para a realização de avaliações de desempenho corresponde ao planejamento e execução de experimentos de medição diretamente no sistema em estudo e sob as condições de carga atuais. Embora esta seja uma técnica muito utilizada, as informações assim obtidas refletem a condição de uso atual do sistema e não podem ser facilmente generalizadas. Uma solução para este problema envolve o uso de geradores de carga para simular cargas artificiais permitindo a observação do comportamento das métricas de desempenho sob diferentes demandas de serviço. O uso desta técnica no contexto de avaliação de desempenho de sistemas de *middleware* requer a instalação de aplicações de *benchmark*, uma vez que o próprio *middleware* em si não responde diretamente às requisições de usuários.

Em muitos cenários reais, um sistema pode não estar disponível para a realização de medições diretas (por exemplo, por ainda estar em desenvolvimento), sendo inviável o uso da abordagem proposta. Diante destes

cenários, uma possível solução envolve o desenvolvimento e utilização de *protótipos*. Para que os resultados obtidos a partir de experimentos realizados em protótipos sejam significativos, é necessário que o próprio protótipo apresente um alto nível de detalhes, o que, muitas vezes torna o seu desenvolvimento caro ou inviável.

- **Modelagem**

Uma outra categoria de técnicas usada para a avaliação de desempenho não requer a realização de experimentos de medição em sistemas ou em protótipos, mas a construção e utilização de representações abstratas destes sistemas. Tais técnicas, referidas de forma genérica como técnicas de modelagem, propõem a construção de modelos a partir de informações extraídas diretamente do próprio sistema e/ou de sua documentação. Tais modelos podem ser configurados a partir do ajuste de parâmetros definidos para representar diferentes configurações do sistema e a carga que pode ser submetida. Dois tipos diferentes de modelos podem ser desenvolvidos para um sistema: modelos de simulação e modelos analíticos.

Nos modelos de simulação utilizados para fins de avaliação de desempenho, a descrição de um sistema é embutida em um programa de computador que ao executar simula a sua operação. Nestes modelos, o estado de um sistema é normalmente representado através de variáveis que assumem valores discretos. Ao longo de um experimento de simulação, as operações do sistema são simuladas repetidas vezes e as métricas configuradas são computadas. Um experimento de simulação deve continuar em execução até que o intervalo de confiança para a média de cada métrica configurada no modelo atinja uma dimensão estabelecida. É importante observar que as dimensões de um modelo de simulação, a dimensão do intervalo de confiança desejado e a quantidade de métricas a serem coletadas possuem um grande impacto em termos do tempo necessário à correta execução de um experimento, podendo este durar diversas horas ou até mesmo dias.

Nos modelos analíticos, a descrição de um sistema é realizada através de um conjunto de formulações matemáticas que podem ser solucionadas para um conjunto de parâmetros de entrada, permitindo a derivação das métricas de desempenho necessárias. Para que um modelo analítico possa ser tratável é necessário que suas dimensões sejam pequenas, ou seja, que ele apresente uma visão do sistema em um alto nível de abstração, incorporando assim, apenas as características indispensáveis do sistema. Desta forma, os modelos analíticos geralmente embutem uma menor quantidade de detalhes do que os modelos de simulação. Em contrapartida, os modelos analíticos executam mais rapidamente e, uma vez solucionados fornecem resultados exatos, sem a necessidade de especificação de intervalos de confiança. É importante que se esclareça que o termo exatidão neste contexto se refere ao fato de que os resultados obtidos são soluções de sistemas de equações, não indicando que o modelo reflete o sistema representado. A proximidade dos resultados obtidos através de modelos com os resultados medidos diretamente nos sistemas reais depende fundamentalmente da adequação da representação destes sistemas através dos modelos correspondentes.

De forma geral, os modelos analíticos podem ser classificados em modelos baseados em espaço de estados e modelos não baseados em espaço de estados. Dentro da categoria dos modelos baseados em espaço de estados, as cadeias de Markov têm um papel fundamental. Estas cadeias são compostas por um conjunto de estados interligados através de um conjunto de transições. Os estados descrevem as diferentes condições observadas em um sistema, enquanto as transições descrevem as possíveis mudanças entre estes estados. Depois de um período de permanência em um dado estado a cadeia de Markov realiza uma transição para um outro estado. A seleção da transição a ser realizada, e conseqüentemente, do próximo estado a ser assumido é efetuada de forma estocástica. Isto significa que a seleção é feita com base apenas dos rótulos das transições possíveis a partir do estado atual. Tais rótulos contêm dados referentes à freqüência com que as transições devem ocorrer e são representados através de probabilidades (no caso das cadeias de Markov de tempo discreto) ou de taxas (no caso das cadeias de Markov de tempo contínuo).

As soluções para as cadeias de Makov correspondem às probabilidades do sistema se encontrar em cada um dos estados após um determinado intervalo de tempo (solução transiente) ou após um longo período de execução (solução estacionária). A solução estacionária para uma cadeia de Markov pode ser obtida através da resolução de um sistema de equações lineares, possuindo uma equação para cada estado definido na cadeia, enquanto que soluções transientes para cadeias de Markov de tempo contínuo podem ser obtidas através da resolução de sistemas de equações diferenciais ordinárias. A partir das probabilidades calculadas, diversas métricas de desempenho necessárias podem ser derivadas.

Modelos analíticos mais concisos podem ser construídos utilizando outros formalismos como, por exemplo, Redes de Petri Estocásticas (Stochastic Petri Nets, SPN) e Redes de Filas (Queueing Networks, QN). Estes formalismos possuem uma correspondência com as cadeias de Markov, permitindo o uso de ferramentas automatizadas para a geração e solução das cadeias subjacentes.

Um problema fundamental com os modelos analíticos baseados em espaço de estados é que o tamanho deste espaço cresce exponencialmente com as dimensões do sistema representado, inviabilizando, muitas vezes, sua construção e/ou solução. Visando aliviar estes problemas, foram desenvolvidas diversas técnicas baseadas em idéias como:

- Representação condensada do espaço de estados;
- Redução no espaço de estados através de transformações nos modelos de alto nível (por exemplo, redes de Petri); e
- Construção do espaço de estados sob demanda.

Modelos não baseados em espaço de estado também começaram a emergir no mundo de avaliação de desempenho. Dentre eles, destaca-se uma classe de redes de filas chamada *product-form queueing network* (PFQN). Estas redes podem ser utilizadas no cálculo de métricas de desempenho em regime estacionário sem que seja necessária a geração do espaço de estados subjacentes. Outros exemplos de modelos analíticos não baseados em espaço de

estados compreendem diagramas de blocos, grafos de confiabilidade e árvore de falhas.

Em suma, a representação de sistemas através de modelos analíticos é uma tarefa árdua e depende consideravelmente da habilidade/experiência do modelador de visualizar e representar estes sistemas em um alto nível de abstração.

2.3 Redes de Petri

As redes de Petri são uma família de ferramentas gráficas utilizadas para descrição formal de sistemas que possuem características como concorrência, conflito, sincronização e exclusão mútua. Esta família de formalismos baseia-se na idéia de que os sistemas assumem, ao longo de sua execução, diferentes estados, os quais são modificados a partir da ocorrência de determinados eventos. Os modelos em redes de Petri representam estes estados e eventos, capturando características estáticas e dinâmicas dos sistemas.

2.3.1 Definições

Uma rede de Petri é multi-grafo bipartite e direcionado. Um grafo bipartite possui dois tipos diferentes de nós. No caso das redes de Petri, estes nós são chamados lugares e transições e são representados através de círculos e retângulos, respectivamente. Sendo um grafo direcionado, os lados presentes em uma rede de Petri, chamados arcos, possuem uma direção associada. Grafos bipartite possuem uma importante característica, seus arcos só podem conectar dois nós que pertençam a tipos diferentes. Desta forma, um arco em uma rede de Petri interliga um lugar à uma transição (arco de entrada) ou uma transição a um lugar (arco de saída). Os multi-grafos podem possuir mais de um arco interligando dois nós em uma mesma direção. Em redes de Petri, arcos paralelos e com uma mesma direção são normalmente representados através de um único arco rotulado com um número natural referenciado como multiplicidade. Desta forma, um arco com multiplicidade *k* corresponde a um conjunto de *k* arcos dispostos paralelamente.

Utilizando uma notação mais precisa, uma rede de Petri pode ser representada através de uma tupla $PN = (P, T, I, O)$, onde:

$P = \{p_1, p_2, ..., p_n\}$ é um conjunto de lugares,

$T = \{t_1, t_2, ..., t_m\}$ é um conjunto de transições, $P \cap T = \phi$,

$I(p,t)$: $P \times T \rightarrow \aleph$ é uma função que mapeia um par (p,t) em um número natural correspondente à multiplicidade do arco direcionado do lugar *p* para a transição *t*. Neste caso, o lugar *p* é dito um lugar de entrada da transição *t*. Caso não exista tal arco, o valor assumido pela função é *0*,

$O(t,p)$: $T \times P \rightarrow \aleph$ é uma função que mapeia um par (t,p) em um número natural correspondente à multiplicidade do arco direcionado da transição *t* ao lugar *p*. Neste caso, o lugar *p* é dito um lugar de saída da transição *t*. Caso não exista este arco, o valor assumido pela função é *0*.

Um modelo em redes de Petri possui, além dos elementos descritos acima, os chamados *tokens* que são marcas indistinguíveis entre si. Os *tokens*

são denotados por pontos sólidos que residem nos lugares e trafegam pela rede através dos arcos. O fluxo de *tokens* é controlado pelas transições. A marcação de um modelo em um dado instante, denotada por *M(p)*, é uma função que associa a cada lugar um número natural, representando a quantidade de *tokens* residindo naquele lugar. Outra representação comum para a marcação é a de um vetor unidimensional, no qual o *i-ésimo* elemento representa a quantidade de *tokens* contida no lugar p_i. Cada marcação possível corresponde a um estado global no qual o modelo pode se encontrar.

Utilizando novamente a notação de tupla, um modelo em redes de Petri pode ser representada por $M = (P, T, I, O, M_0)$, onde:

P, T, I, O são descritos na definição de rede de Petri apresentada acima, e

$M_0(p) : P \rightarrow \aleph$ é a marcação inicial da rede.

A execução de um modelo provoca alterações em sua marcação e é ocasionada pelo disparo das transições que são os componentes ativos do modelo. Para que uma transição possa disparar, é necessário que cada lugar de entrada desta transição possua pelo menos um número de *tokens* igual à multiplicidade do arco que o conecta à transição. Neste cenário diz-se que a transição está habilitada.

O disparo de uma transição promove uma alteração na marcação corrente de um modelo removendo de cada lugar de entrada, o número de *tokens* indicado pela multiplicidade do arco que o conecta a transição e criando, em cada lugar de saída, o número de *tokens* indicado pela multiplicidade do arco de saída que o liga à transição disparada. O disparo é uma operação atômica, de forma que os *tokens* são removidos dos lugares de entrada e criados nos lugares de saída em uma operação indivisível.

O disparo de uma seqüência de transições promove sucessivas transformações na marcação de um modelo. Em um modelo em redes de Petri, uma marcação M' é dita alcançável a partir de uma marcação *M*, se e somente se, existir uma seqüência de transições $\sigma = t_1, t_2, ..., t_n$ que possa ser disparada a partir de *M*, de tal forma que, ao final de sua execução, o modelo esteja na marcação M'.

A Figura 2.5 apresenta o disparo de uma seqüência de transições em um modelo em redes de Petri. Na marcação inicial, representada na Figura 2.5(a), apenas a transição t_1 está habilitada. O disparo desta transição leva o modelo a uma nova marcação, representada na Figura 2.5(b). Nesta marcação, o modelo possui um número menor de *tokens* do que na marcação inicial, demonstrando que a quantidade total de *tokens* pode ser alterada ao longo da execução. Na nova marcação, apenas a transição t_2 está habilitada e o seu eventual disparo leva o modelo a uma marcação na qual nenhuma das transições está habilitada, caracterizando a ocorrência de um *deadlock*. Esta marcação é representada na Figura 2.5(c).

Uma propriedade interessante do modelo representado na Figura 2.5 é que a quantidade máxima de *tokens* armazenados em cada lugar é limitada a um. Um modelo no qual, em qualquer marcação alcançável, cada lugar pode conter no máximo *k tokens* é dito *k*-limitado. Uma importante conseqüência da

limitação é que ela implica em um número finito de marcações alcancáveis, e, conseqüentemente, em um número finito de estados. De fato, considerando um modelo com n lugares, no qual cada lugar está limitado a k tokens, o número máximo teórico de possíveis estados é dado por $(k+1)^n$. Embora normalmente o número real de estados atingíveis seja bem menor, devido às restrições impostas pelas condições de habilitação das transições, pode-se constatar um crescimento exponencial do número de estados com relação ao número de lugares do modelo.

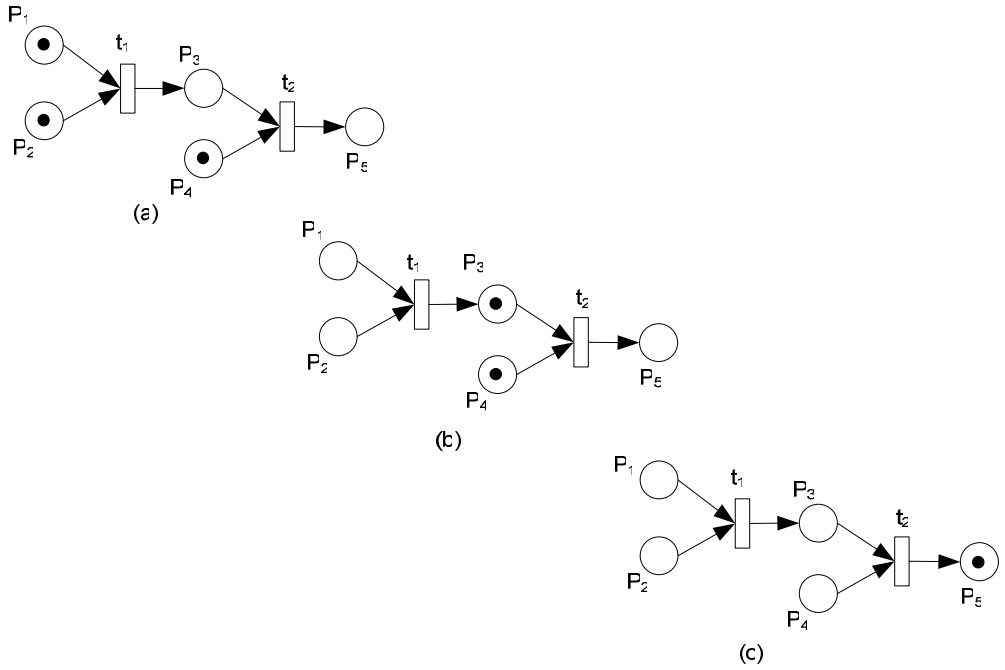


Figura 2.5. Seqüência de Disparos em um Modelo em Redes de Petri.

Um exemplo de modelo não limitado é apresentado na Figura 2.6. A única diferença entre este modelo e aquele apresentado na Figura 2.5 é a inclusão de dois arcos de saída interligando a transição t_2 aos lugares P_3 e P_4. Estes arcos fazem com que a transição t_2 continue habilitada após o seu disparo, podendo gerar um número ilimitado de *tokens* acumulados no lugar P_5. De fato, o modelo apresenta a propriedade de ausência de *deadlocks*, uma vez que em qualquer marcação alcançável há sempre uma transição habilitada.

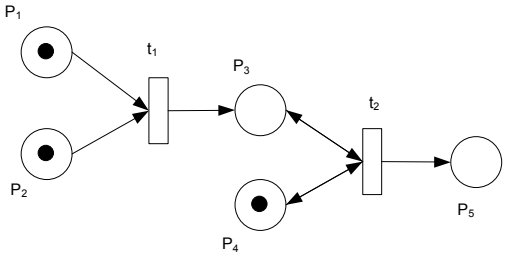


Figura 2.6. Modelo Não Limitado.

A análise de propriedades qualitativas, tais como limitação e ausência de *deadlocks*, fornece informações relevantes acerca do comportamento dos sistemas modelados em redes de Petri. A semântica destas propriedades está intimamente relacionada à natureza dos sistemas em si. O Apêndice A deste

trabalho apresenta as principais propriedades e técnicas de avaliação disponíveis no domínio das redes de Petri.

2.3.2 Semântica

Para que um modelo em redes de Petri possa ser interpretado é necessária uma atribuição de significado aos seus elementos. Os lugares são componentes passivos tipicamente utilizados para descrever os possíveis estados locais de um sistema, uma condição ou a disponibilidade de um tipo de recurso. Por outro lado, as transições são componentes ativos que descrevem as atividades que podem ser realizadas pelo sistema, modificando o seu estado atual. Cada atividade começa e termina com dois eventos consecutivos. O primeiro evento determina a habilitação da transição, marcando o início da atividade correspondente. O segundo evento corresponde ao seu disparo representando a conclusão da atividade modelada. Os arcos representam a relação estrutural entre transições e lugares, estabelecendo pré-condições e/ou pós-condições relativas à realização das atividades modeladas pelas referidas transições. A semântica dos *tokens* depende da semântica do lugar no qual ele está residindo. Por exemplo, se um lugar representa uma dada condição, a presença de um *token* neste lugar representa que a condição é verdadeira, e a sua ausência que a condição é falsa. Caso o lugar represente um tipo de recurso, a quantidade de *tokens* no lugar representa o número de recursos daquele tipo que estão disponíveis.

A Figura 2.7 apresenta um modelo em redes de Petri representando uma chave que permite ligar e desligar um dispositivo. O modelo é composto por dois lugares: P_{on} representando que o dispositivo está ligado; e P_{off} representando que o mesmo está desligado. Neste sistema, duas atividades diferentes são possíveis: desligar o dispositivo, modelada pela transição Toff; e ligar o dispositivo, modelada pela transição T_{on}. O estado deste modelo no instante representado na figura é dado por P_{on}=1 e P_{off}=0. O *token* contido em P_{on} indica que o dispositivo está ligado.

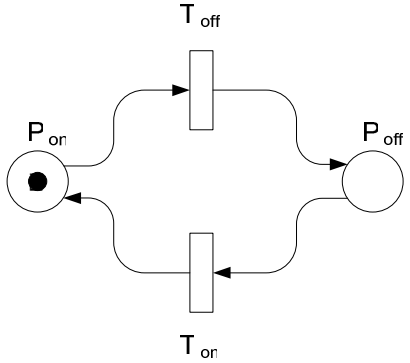


Figura 2.7. Rede de Petri Representando uma Chave para Ligar e Desligar Dispositivo.

No exemplo apresentado na Figura 2.7, a única transição correntemente habilitada é a transição T_{off}. O disparo desta transição remove o *token* do lugar P_{on} e cria um novo *token* no lugar P_{off}, representando o desligamento do dispositivo modelado. A Figura 2.8 representa o novo estado atingido pelo modelo.

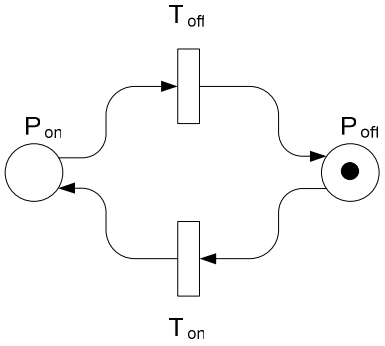


Figura 2.8. Dispositivo Desligado.

Uma importante extensão que amplia o poder de modelagem das redes de Petri são os arcos inibidores. Um arco inibidor é representado por um arco com uma extremidade circular e foi introduzido no formalismo para possibilitar o teste de um lugar para zero. A inclusão dos arcos inibidores modifica a condição de habilitação das transições de forma que, uma transição somente é considerada habilitada a disparar se: (1) cada lugar de entrada contiver um número de *tokens* maior ou igual a um limite estabelecido pela multiplicidade do arco de entrada correspondente; (2) cada lugar inibidor (lugar conectado à transição através de um arco inibidor) contiver um número de *tokens* menor do que o limite estabelecido pela multiplicidade do arco inibidor correspondente.

A Figura 2.9 demonstra o conceito de arco inibidor em um modelo representando uma chave que pode falhar. Neste cenário, uma falha da chave significa que ela permanece no seu estado local (ligada ou desligada), não podendo ser operada até que seja recuperada. Dois arcos inibidores são representados no modelo, um ligando $P_{NaoFuncional}$ a T_{off}, e outro ligando $P_{NaoFuncional}$ a T_{on}. Estes arcos inibidores garantem que, quando a chave estiver em falha (condição representada pelo lugar $P_{NaoFuncional}$), ela não poderá ser ligada ou desligada.

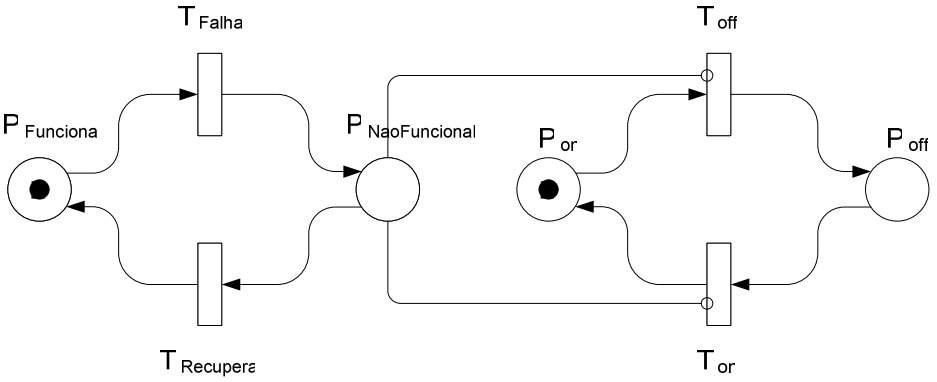


Figura 2.9. Chave com Estado de Falha.

A Figura 2.9 ilustra uma situação comum na modelagem de sistemas em redes de Petri que é a ocorrência de conflitos. Um conflito ocorre em um modelo quando mais de uma transição está simultaneamente habilitada e o disparo de uma delas interfere com a possibilidade de disparos futuros das demais. Uma definição mais precisa de conflito envolve o conceito de grau de habilitação de uma transição que corresponde ao número de vezes que uma transição pode ser disparada sucessivamente antes de se tornar desabilitada. Em

função deste conceito, pode-se definir um conflito como sendo qualquer situação na qual o disparo de uma transição reduz o grau de habilitação de outra transição.

Na marcação representada na figura, as transições T_{falha} e T_{off} estão habilitadas. Embora um disparo da transição T_{off} não tenha impacto sobre a possibilidade de disparos da transição T_{falha}, o inverso não é verdadeiro. De fato, um disparo da transição T_{falha} gera um *token* no lugar $P_{NaoFuncional}$, o qual inibe a transição T_{off}, impedindo que esta possa ser disparada. Esta situação caracteriza a existência de um conflito entre as duas transições mencionadas.

2.3.3 Redes de Petri Temporizadas

Nos modelos apresentados até agora não foi feita nenhuma consideração acerca da quantidade de tempo necessária à realização das atividades modeladas. O conceito de tempo pode ser incorporado aos modelos através da associação de cada transição com um atraso determinístico que é contabilizado a partir do instante de sua habilitação. Esta abordagem viabiliza o projeto de modelos determinísticos que podem, em princípio, ser utilizados na derivação de resultados de desempenho [Desrochers e Al-Jaar 1995].

Um problema fundamental com esta abordagem é que os sistemas reais normalmente são muito grandes e complexos, dificultando a construção de modelos determinísticos, uma vez que para que estes possam fornecer bons resultados é necessária a representação de todos os detalhes envolvidos. Diante deste cenário é comum a utilização de modelos estocásticos, os quais permitem a representação de incertezas de forma natural.

No contexto de redes de Petri, as primeiras idéias à cerca das redes estocásticas surgiram nos trabalhos de [Symons 1980], [Natkin 1980] e [Molloy 1982]. O ponto central destes trabalhos é a proposição de uma extensão ao modelo original que associa às transições, uma variável aleatória representando o conceito de tempo.

No modelo proposto por Molloy, as variáveis aleatórias utilizadas na representação do conceito temporal são exponenciais. Isto possibilita uma isomorfia entre estas redes e as cadeias de Markov de tempo contínuo, dada a natureza sem memória desta distribuição de probabilidade. A relação entre estes modelos teve fundamental importância para o sucesso do modelo de Molloy, o qual viria a ser chamado simplesmente de SPN (*Stochastic Petri Nets*).

Nos modelos SPN, o disparo de uma transição ocorre de forma atômica. Sendo assim, os *tokens* permanecem nos lugares de entrada de uma transição durante o tempo necessário à execução da atividade correspondente. Conforme mencionado, este tempo não é determinístico, mas estocástico, sendo configurável através do valor médio de uma variável aleatória exponencial. Quando a atividade é concluída, ocorre o disparo da transição, provocando a remoção de *tokens* de seus lugares de entrada e a criação de *tokens* em seus lugares de saída, em uma operação indivisível. Esta regra de disparo preserva a semântica definida no modelo não temporizado de redes de Petri, permitindo uma análise de propriedades qualitativas, tais como alcançabilidade, reversibilidade e ausência de *deadlock* (ver Apêndice A.5).

As situações de conflito envolvendo duas ou mais transições temporizadas simultaneamente habilitadas são solucionadas com base em condições de corrida (*race conditions*), indicando que a seleção da transição a ser disparada decorre dos tempos efetivamente associados à cada uma delas.

Uma importante característica das transições temporizadas é a semântica de disparo correspondente. Existem três semânticas de disparo possíveis [Marsan et al. 1994]:

- ***Single-server*:** simula a execução da transição em um único servidor disponível, de forma que os disparos da transição com um grau de habilitação maior do que um ocorrem de forma seqüencial. Ou seja, se o grau de habilitação da transição for 4, a transição é disparada 4 vezes, uma após a outra;
- ***Infinite-server*:** simula a execução da transição quando se tem um número infinito de servidores. Desta forma, a execução de uma transição com grau de habilitação *n* ocorre sob a forma de *n* execuções em paralelo; e
- ***Multiple-server*:** simula a execução da transição quando se tem um número finito de servidores disponíveis para ela. A execução de uma transição com grau de habilitação *n* utilizando-se *k* servidores ocorre através de seqüências de *k* execuções simultâneas.

2.3.4 Redes GSPN e DSPN

Uma situação comum durante a modelagem de sistemas, é a necessidade de representação de atividades internas que consomem um tempo ordens de grandeza menor do que os tempos consumidos pelas demais atividades representadas na rede. Para representar adequadamente estas situações, é conveniente a inclusão de um novo tipo de transição, referenciado como transição imediata.

Neste contexto, surge o modelo GSPN (*Generalized and Stochastic Petri Nets*) [Marsan et al. 1984] que estende a proposta de Molloy, incorporando o conceito de transições imediatas, as quais, como o próprio nome indica, possuem tempo de disparo zero. Na representação gráfica de uma rede GSPN, as transições temporizadas são representadas por caixas brancas, enquanto que as transições imediatas são representadas por barras.

No modelo GSPN, as regras que regem os disparos das transições são modificadas de forma a comportar as diferentes classes de transição. De fato, o modelo atribui às transições imediatas uma prioridade maior do que aquela atribuida às transições temporizadas. Desta forma, se em uma dada marcação, uma transição temporizada e uma transição imediata estiverem simultaneamente habilitadas, a transição imediata será disparada.

Conflitos entre transições imediatas podem ser resolvidos deterministicamente através da modificação do nível de prioridade associado às transições conflitantes. As prioridades correspondem a números naturais que podem ser modificados permitindo o particionamento das transições em classes. Todas as transições de uma classe possuem um mesmo nível de prioridade, e somente podem estar habilitadas, se nenhuma transição pertencente a outra

classe de maior prioridade estiver habilitada. Mais precisamente, em uma rede de Petri com prioridades uma transição está habilitada quando: (1) seus lugares de entrada tiverem um número de *tokens* maior ou igual a um limite estabelecido pela multiplicidade do arco correspondente; (2) seus lugares inibidores tiverem um número de *tokens* menor do que o limite estabelecido pela multiplicidade dos arcos inibidores correspondentes; e (3) não houver nenhuma transição com maior prioridade que atenda às duas condições anteriores.

Esses conflitos podem, ainda, ser resolvidos de forma estocástica com base no conceito de peso. Os pesos são valores númericos utilizados como fatores de ponderação no cálculo da probabilidade de disparo das transições conflitantes. Desta forma, suponha que duas transições imediatas t_1 e t_2 tenham uma mesma prioridade e estejam simultaneamente habilitadas. Nesta situação, se um peso de 0,7 é atribuído a t_1 e 0,3 é atribuído a t_2, t_1 terá probabilidade de disparar de 70%.

Por fim, outra maneira de resolver conflitos entre transições imediatas é utilizar funções de habilitação (também referenciadas como condições de guarda) que são expressões booleanas associadas às transições. Uma transição que possui condição de habilitação só pode estar habilitada se, na marcação corrente, esta condição for avaliada como verdadeira.

Um exemplo de modelo GSPN é apresentado na Figura 2.10. Este modelo representa clientes gerando requisições que são acumuladas em um *buffer*. Cada *token* contido no lugar *ClientesProntos* corresponde a um cliente preparado para gerar uma nova requisição. A marcação inicial indica que 5 clientes estão inicialmente nesta situação. Os *tokens* em *ClientesProntos* habilitam a transição *gerarRequisicao* que é temporizada e possui um tempo médio de disparo representando o intervalo entre requições geradas por um mesmo cliente. Como este tempo possui uma distribuição exponencial, a taxa de requisições geradas segue uma distribuição de *Poisson*. Cada requisição realizada é representada através de um *token* criado no lugar *Buffer*. A transição *gerarRequisicao* também gera *tokens* no lugar *RequisicaoGerada*, os quais habilitam a transição imediata *gerarProxima*, indicando que os clientes estarão imediatamente aptos a gerar novas requisições.

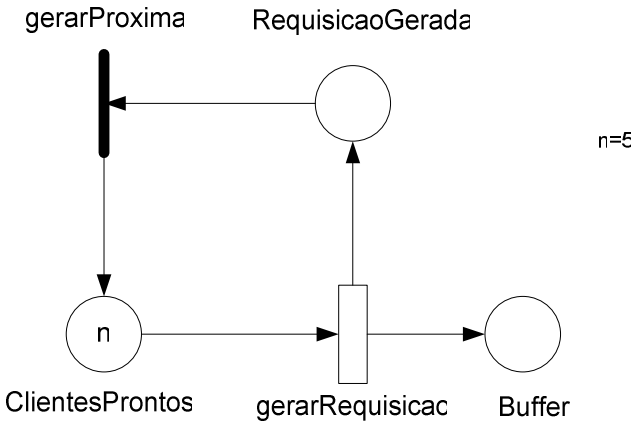


Figura 2.10. Exemplo GSPN: Buffer Ilimitado

Neste modelo, a semântica de disparo da transição temporizada *gerarRequisicao* tem um importante papel. Caso esta semântica seja *single*

server, apenas um cliente estará efetivamente gerando requisições a cada instante. Por outro lado, se a semântica associada for *infinite server*, todos os clientes estão gerando requisições de forma simultânea.

A introdução de transições imediatas permite uma classificação das marcações (estados) alcançáveis por uma rede. Uma vez que as transições imediatas são disparadas assim que se tornam habilitadas, não há consumo de tempo nas marcações que habilitam este tipo de transição. Desta forma, estas marcações são referenciadas como *vanishing*. Por outro lado, as marcações que habilitam somente transições temporizadas, e que, portanto, possuem um tempo de permanência maior do que zero, são chamadas de tangíveis. Uma vez que uma marcação *vanishing* não representa um estado no qual o sistema permanece por um intervalo de tempo, estas marcações podem ser desprezadas para efeito de avaliações de desempenho. Desta forma, estas marcações não são consideradas durante a geração da cadeia de Markov correspondente a um modelo GSPN.

Embora seja mais flexível do que o modelo de SPN original por permitir a modelagem de atividades cujo tempo associado seja tão pequeno que possa ser desprezado, o modelo GSPN possui uma limitação para a modelagem de processos que ocorrem no mundo real. Tal limitação decorre do fato de que um único tipo de transição temporizada é possível, o qual possui um modelo de disparo baseado em variáveis aleatórias exponenciais. Neste modelo, que é, por natureza, sem memória, o momento preciso do disparo de uma transição não pode ser determinado e independe de disparos ocorridos anteriormente.

Esta limitação motivou a proposição de um novo modelo para redes de Petri estocásticas, o qual é conhecido como *Deterministic and Stochastic Petri Nets* (DSPN) [Marsan e Chiola 1987]. Este modelo estende o modelo GSPN acrescentando uma nova categoria de transições temporizadas referenciadas como transições determinísticas. Como o próprio nome indica, tais transições possuem um tempo fixo associado ao seu disparo que ocorre de forma atômica. Desta forma, no modelo DSPN as transições temporizadas podem ser classificadas como exponenciais ou determinísticas. Em termos de representação gráfica, o modelo DSPN mantém as representações do modelo GSPN para transições exponenciais e para imediatas, e acrescenta uma notação para as transições determinísticas, as quais são representadas por caixas pretas.

2.3.5 *Moment Matching*

Freqüentemente, a duração de uma atividade observada só pode ser adequadamente modelada por uma variável aleatória com uma distribuição de probabilidades empírica que não possui natureza exponencial ou determinística. Em geral, uma boa maneira de modelar este tipo de atividade é utilizar combinações de transições exponenciais que são conhecidas como Phase Type distributions (PH distributions) [Neuts 1975]. Algoritmos que fazem mapeamentos de distribuições empíricas em combinações de exponenciais, chamados de algoritmos de casamento de momentos (*moment matching algorithms*), buscam fazer com que um certo número de momentos de ambas as distribuições tenham o mesmo valor.

Durante o projeto de modelos DSPN ou GSPN, uma primeira abordagem para representar uma atividade com distribuição empírica consiste em

aproximá-la por uma transição exponencial com tempo igual à média das durações medidas para a atividade em questão. O algoritmo utilizado neste cenário, casa somente o primeiro momento (média) das distribuições. Contudo, resultados melhores podem ser obtidos utilizando outros algoritmos que casem mais momentos além do primeiro.

Um algoritmo de casamento de momentos interessante, utilizado ao longo deste trabalho, foi proposto por [Watson e Desrochers 1991]. Este algoritmo tira proveito do fato de que distribuições de Erlang usualmente têm média maior que o desvio padrão, enquanto que distribuições hiperexponenciais geralmente possuem média menor que o desvio. Desta forma, o algoritmo propõe que uma atividade com duração empiricamente distribuída seja modelada através de subredes que representem distribuições de Erlang ou hiperexponenciais, chamadas *s-transitions*. De acordo com o coeficiente de variação (razão entre desvio e média amostrais) associado à duração de uma atividade, uma implementação de *s-transition* (Erlangiana ou hiperexponencial) pode ser selecionada. Para cada implementação de *s-transition* há parâmetros que podem ser configurados de forma que o primeiro e o segundo momentos associados à duração de uma atividade empírica casem com o primeiro e segundo momentos da *s-transition* como um todo.

O mapeamento de uma transição empírica para uma *s-transition* hiperexponencial proposto por Watson III é apresentado na Figura 2.11. A implementação da *s-transition* possui três parâmetros: r1 e r2 que representam os pesos das transições imediatas utilizadas e λ que representa a taxa da componente exponencial da implementação. Para aproximar uma distribuição empírica com duração média μ e desvio padrão σ, onde μ<σ, estes parâmetros devem ser configurados para r1 = 2μ2/(μ2+σ2), r2 = 1 – r1, e λ = 2μ/(μ2+σ2), fazendo com que o primeiro e segundo momentos da *s-transition* casem com os momentos correspondentes medidos.

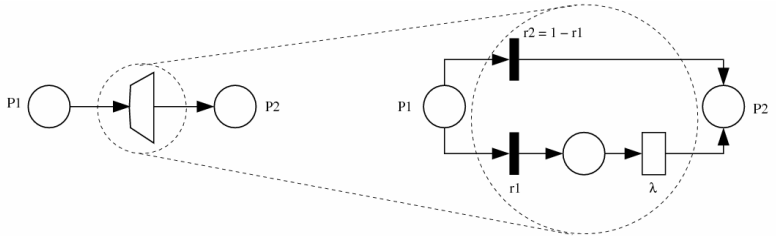


Figura 2.11. Implementação de uma s-transition hiperexponencial.

A Figura 2.12 apresenta o mapeamento de uma transição empírica para uma *s-transition* Erlangiana. A implementação da *s-transition* possui três parâmetros: λ1 e λ2 que representam as taxas das transições exponenciais utilizadas na implementação da *s-transition*; e γ que é um inteiro representando o número de fases da distribuição de Erlang. Para aproximar uma distribuição empírica com duração média μ e desvio padrão σ, onde μ>σ. Estes parâmetros devem ser configurados de forma que: (μ/σ)2–1≤ γ ≤ (μ/σ)2, λ1=1/μ1 e λ2=1/μ2, onde:

$$\mu_1 = \frac{\mu \mp \sqrt{\gamma(\gamma+1)\sigma^2 - \gamma\mu^2}}{\gamma+1} \quad \text{e} \quad \mu_2 = \frac{\gamma\mu \pm \sqrt{\gamma(\gamma+1)\sigma^2 - \gamma\mu^2}}{\gamma+1}.$$

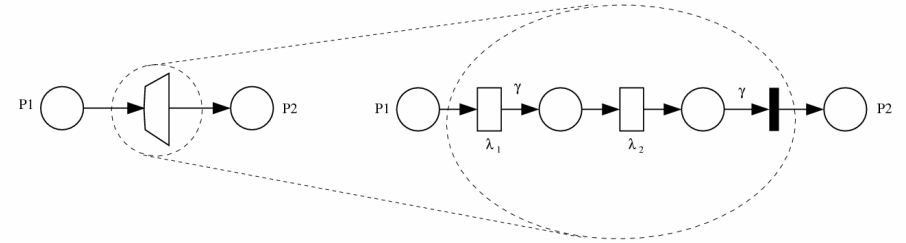


Figura 2.12. Implementação de uma s-transition Erlangiana.

Maiores detalhes concernentes às redes de Petri e suas propriedades são apresentados no Apêndice A.

2.4 Considerações Finais

Este capítulo sumariza os conceitos que fundamentam o presente trabalho. Neste sentido, apresenta-se inicialmente o conceito de *middleware*, destacando-se a plataforma J2EE e suas diferentes implementações (referenciadas como servidores de aplicação J2EE) como relevantes exemplos. Em seguida, os diferentes tipos de componentes introduzidos pela plataforma J2EE são apresentados, enfatisando-se os componentes EJB. Na seqüência, discorre-se, em detalhes, sobre o suporte fornecido a estes componentes pelo servidor de aplicação JBoss, destacando-se a noção de serviço e o papel dos interceptadores. Especial atenção é dada ao serviço de *pooling* de instâncias devido ao seu impacto no desempenho global dos sistemas baseados na tecnologia EJB. Para concluir este capítulo, são apresentados os principais conceitos relacionados à área de avaliação de desempenho, destacando-se as métricas consideradas ao longo do trabalho, as técnicas de avaliação de desempenho disponíveis (medição, simulação e modelagem analítica) e as Redes de Petri Estocásticas, enquanto importantes ferramentas de modelagem de desempenho.

CAPÍTULO

3

Trabalhos Relacionados

"Wise men learn by other men's mistakes,
fools by their own"

H.G.Wells.

Este capítulo apresenta os principais trabalhos relacionados à área de avaliação de desempenho de servidores de aplicação. Ao longo do texto estes trabalhos são descritos, classificados de acordo com a técnica de avaliação de desempenho utilizada (medição, simulação e modelagem analítica) e comparados com a proposta desta dissertação.

3.1 Avaliação de Desempenho de Servidores de Aplicação

Os servidores de aplicação normalmente implementam especificações padrões, sejam estas abertas ou proprietárias. Contudo, embora baseados em especificações comuns, os servidores de aplicação apresentam diferentes características em termos de suporte de ferramentas, aderência ao padrão, simplicidade, flexibilidade, escalabilidade, custo e desempenho, entre outras.

A escolha de um servidor de aplicação com base em todas as combinações dessas características é, certamente, uma tarefa árdua, exigindo uma vasta experiência com produtos distintos. Tal dificuldade vem restringindo a avaliação destes servidores aos aspectos relacionados a custo e desempenho. Diante desse contexto, a avaliação de servidores de aplicação pode ser realizada através das técnicas de medição, de simulação e/ou de modelagem analítica.

3.2 Medição

A maioria dos projetos de avaliação de desempenho de servidores de aplicação descritos na literatura utiliza a técnica de medição. Normalmente, estes projetos objetivam avaliar implementações distintas de uma mesma especificação, otimizar parâmetros de configuração do servidor e avaliar o impacto da utilização de diferentes perfis de hardware, ou mesmo de diferentes arquiteturas de aplicação.

3.2.1 Projeto MTE

O projeto MTE [Gorton et al. 2003] [Ran et al. 2001] [Liu et al. 2002a] [Liu et al. 2002b] [Liu e Gorton 2004] [Liu et al. 2004] foi responsável pela produção de uma série de *papers* e relatórios técnicos contendo uma análise detalhada de diversos servidores de aplicação J2EE, dentre os quais destacam-se: IBM WebSphere, BEA WebLogic, Borland *Enterprise Server* e JBoss. A análise destes produtos levou em consideração aspectos qualitativos relativos à escalabilidade, facilidade de gerenciamento, disponibilidade, suporte de ferramentas, dentre outros; e aspectos quantitativos relativos ao desempenho do *container* EJB embutido em cada um dos produtos.

Os experimentos de desempenho realizados foram baseados em uma aplicação desenvolvida ao longo do próprio projeto, referenciada como *Stock Online*, a qual simula uma aplicação simples de compra e venda. Para garantir a igualdade de condições nos diversos experimentos realizados, estabeleceu-se um ambiente padrão e isolado, no qual cada servidor é instalado e testado separadamente. A especificação deste ambiente determina a utilização de máquinas separadas na execução dos clientes, do servidor e do banco de dados, assim como a rede que interliga tais máquinas e as plataformas de hardware e sistema operacional de cada uma. Esta homogeneidade de ambiente permite uma comparação direta dos resultados obtidos com cada servidor de aplicação individualmente. Este procedimento é contrário ao adotado em outros *benchmarks*, como o ECPerf, segundo os quais os produtos podem ser avaliados em ambientes distintos.

Para garantir o melhor desempenho possível em cada experimento realizado, o servidor de aplicação e a máquina virtual responsável pela sua execução são configurados de forma a otimizar o desempenho do servidor sob avaliação. Esta configuração varia de produto para produto, mas em geral envolve o tamanho dos *pools* de threads, de conexão com a base de dados e de instância, o tamanho da *heap* e a versão da máquina virtual, entre outros.

3.2.2 Cecchet

Em um trabalho bastante relevante, Cecchet [Cecchet et al. 2002] avalia, de forma detalhada, o impacto da arquitetura de uma aplicação J2EE e a relevância do projeto do próprio servidor de aplicação no desempenho de um sistema. Neste trabalho, seis versões de uma aplicação de leilão eletrônico são implementadas e instaladas em dois importantes servidores de aplicação de código aberto disponíveis no mercado, JBoss 2.4.4 e JOnAs 2.4.4.

A primeira versão utiliza apenas servlets, as quais implementam as lógicas de negócio e de apresentação. Esta versão não utiliza EJB e é utilizada

apenas como referência de desempenho. A segunda versão utiliza apenas *Stateless Session Beans*, os quais são acessados pelas servlets e realizam acesso à base de dados. A terceira versão encapsula o acesso à base de dados em *entity beans* CMP, que são diretamente acessados pelas servlets. Na quarta implementação, os *entity beans* CMPs são substituídos por BMPs, retirando do *container* EJB a responsabilidade direta pela persistência. Na quinta versão, o acesso aos *entity beans* é realizado através de um *session bean* implementando o padrão de projeto *Session Façade*. Por fim, a sexta versão da aplicação é implementada fazendo um uso explícito das interfaces locais disponibilizadas a partir da versão 2.0 da especificação de EJB. Tais interfaces permitem que componentes EJB sejam acessados diretamente através de referências locais sem a necessidade do uso da tecnologia *Remote Method Invocation* RMI.

As versões desenvolvidas são instaladas nos dois servidores avaliados, os quais encontram-se em execução em ambientes idênticos. Para avaliar essas versões, dois tipos de clientes foram concebidos. O primeiro deles somente realiza operações de consulta de itens vendidos (leitura), enquanto que no segundo 15% das transações realizadas envolvem operações de escrita. Por fim, cenários envolvendo estes tipos de cliente são montados e submetidos para as diferentes versões instaladas em ambos servidores.

Como resultado, o trabalho demonstrou que o fator mais importante na determinação do desempenho de um sistema é a arquitetura utilizada no desenvolvimento das aplicações. Demonstrou, ainda, que o impacto no desempenho relativo ao projeto do servidor de aplicação em si é pouco relevante nas aplicações que só envolvem *Session Beans*, mas pode tornar-se significativo quando a arquitetura da aplicação incorpora *entity beans*.

3.2.3 Avaliação dos Trabalhos de Medição

De forma geral, os trabalhos apresentados nesta seção tiveram uma forte influência na fundamentação da dissertação. De fato, a seleção de métricas, parâmetros e fatores (ver Capítulo 4) utilizados nos experimentos de avaliação de desempenho desenvolvidos baseou-se nas pesquisas apresentadas.

Outro aspecto comum importante é a idéia de separar o impacto sobre o desempenho devido à arquitetura da aplicação daquele devido à infra-estrutura fornecida pelos servidores de aplicação através da utilização de aplicações de *benchmark* simplificadas. Por fim, limitações similares são aplicadas a todos os trabalhos apresentados, inclusive ao presente. Tais limitações estabelecem que os resultados obtidos são válidos somente para as plataformas de hardware, sistema operacional, máquina virtual e servidores de aplicação avaliados, não podendo, assim, serem interpretados em um âmbito mais amplo. Conforme observado nos projetos mencionados, o desempenho global é fortemente afetado pelas características particulares de cada produto, não sendo possível uma generalização de resultados.

Apesar das semelhanças apresentadas, o presente trabalho possui características peculiares que o diferenciam significativamente dos demais. Primeiramente, este trabalho propõe modelos formais para avaliação de desempenho do JBoss desenvolvidos em redes de Petri estocásticas. Tais modelos possibilitam a utilização das técnicas de simulação e/ou análise na derivação dos resultados de desempenho e a verificação de propriedades

qualitativas, tais como, limitação e ausência de *deadlocks*, as quais podem ser mapeadas em características desejáveis, de acordo com a semântica do modelo proposto. A ausência de *deadlocks*, por exemplo, demonstra que o servidor de aplicação não "trava" durante o processamento das requisições.

Por outro lado, a possibilidade de utilização das técnicas de simulação e/ou análise, além de promover uma validação de resultados obtidos em experimentos de medição, simplifica a montagem de cenários diversos, torna os experimentos de avaliação mais imunes a variações no ambiente externo e fornece resultados, em geral, mais rapidamente. Por fim, deve-se destacar que o uso da análise, quando viável, garante a precisão matemática dos resultados obtidos.

Outra característica que diferencia esse trabalho é a utilização de medições "caixa-branca" através da instrumentação do código fonte do servidor JBoss. Esta característica permitiu uma investigação detalhada dos aspectos internos deste servidor de aplicação e a representação de tais aspectos diretamente nos modelos. Este nível de detalhe permite uma análise minuciosa do funcionamento do servidor e facilita a identificação de eventuais gargalos.

3.3 Modelagem

Os modelos de desempenho de um sistema capturam os aspectos mais relevantes do comportamento deste sistema em termos de como a carga de trabalho a que este está submetido consomem os recursos disponíveis. Conforme mencionado anteriormente, há dois tipos de modelos de desempenho, modelos de simulação e modelos analíticos. Os modelos analíticos representam o comportamento do sistema através de um conjunto de formulações matemáticas, cujas soluções oferecem informações sobre as métricas de desempenho estabelecidas. Por outro lado, os modelos de simulação são programas que imitam o comportamento de um sistema simulando a geração e o processamento de eventos. Ambos os tipos de modelo devem considerar a competição por recursos e as filas correspondentes, sejam estes recursos de hardware ou software. De forma geral, a utilização das técnicas de modelagem envolvem um compromisso entre a expressividade de um lado e o tamanho e a complexidade de outro [Menascé e Almeida 2003].

3.3.1 Modelagem Analítica

A técnica de modelagem analítica é freqüentemente considerada a melhor solução, do ponto de vista da relação custo/benefício, quando comparada com a de simulação ou medição. Este fato se deve, fundamentalmente, a precisão matemática associada aos resultados obtidos a partir dos modelos analíticos, a velocidade de solução de tais modelos e ao custo associado ao seu desenvolvimento.

Os modelos analíticos podem ser classificados como baseados ou não em espaço de estados. Os modelos baseados em espaço de estados mais comuns correspondem às cadeias de Markov. Tais cadeias são compostas por conjuntos de estados e transições rotuladas entre eles. Diversos modelos formais são baseados nestas cadeias, destacando-se dentre eles as redes de Petri estocásticas, as Layered Queueing Networks (LQNs), as Extended Queueing Networks e as Queueing Petri Nets. O maior problema com as cadeias de

Markov (e com os modelos baseados em espaço de estados, de forma geral) é conhecido como explosão de estados, onde o tamanho do modelo cresce exponencialmente e os recursos de memória se esgotam rapidamente [Menascé et al. 2004].

Para um modelo em rede de Petri, o número de estados da cadeia de Markov correspondente varia em função do número de *tokens* e de lugares presentes na rede (ver Seção 2.3). De fato, observa-se uma dependência exponencial no número de lugares, indicando que a dimensão de um modelo pode rapidamente inviabilizar a solução da cadeia de Markov subjacente.

Uma vez que os modelos analíticos não baseados em espaço de estados não sofrem as limitações impostas pela explosão de estados, torna-se possível a construção de modelos maiores e mais detalhados. Nesta categoria destacam-se as Product-Form Queueing Networks (PFQNs). Esta classe de redes de fila não requer a geração do espaço de estados na derivação das métricas de desempenho em regime estacionário.

Um dos trabalhos pioneiros sobre avaliação de desempenho de sistemas com suporte à tecnologia EJB utilizando modelos analíticos foi desenvolvido por Lladó [Lladó e Harrison 2000] [Lladó et al. 2002]. Este trabalho apresenta um modelo de desempenho para um sistema teórico de suporte à tecnologia EJB (ou seja, não representa qualquer servidor de aplicação disponível) utilizando redes de fila. O modelo projetado é reduzido utilizando-se técnicas de agregação de forma a se tornar matematicamente tratável. As soluções analíticas obtidas do modelo são validadas através de comparações com simulações realizadas. Neste processo de validação é considerada, basicamente, a métrica de *throughput*.

Na construção do modelo de desempenho, Lladó considera apenas um tipo de componente EJB, os *entity beans* e desenvolve um estudo à cerca do mecanismo de invocação de métodos em um componente deste tipo. Lladó observa que o acesso à instâncias destes componentes é compartilhado pelos clientes. Cada instância de um *entity bean* representa um objeto persistente, que pode ser entendido como um "registro" em uma base de dados. Tal instância pode ser "concorrentemente" acessada por clientes visando a realização de consultas ou atualizações. Este acesso "concorrente" deve ser controlado pelo ambiente de execução do componente, referenciado como *container*, de forma evitar a ocorrência de possíveis inconsistências. O modelo desenvolvido considera que o container provê acesso sincronizado aos métodos do componente evitando, assim, tal problema. Esta sincronização é realizada pelo *container* durante o processo de interceptação de requisições. Conforme mencionado no Capítulo 2, um *container* intercepta requisições destinadas aos componentes por ele gerenciados provendo serviços de forma transparente. Tais serviços são selecionados e configurados de forma declarativa através de um descritor XML referenciado como descritor de instalação (*deployment descriptor*). O modelo projetado por Lladó assume que um *container* EJB gerencia apenas um componente (o qual pode ter várias instâncias).

Outro elemento importante considerado no modelo é o gerenciador de *threads*. Ao receber uma requisição destinada a um componente EJB, um servidor de aplicação aloca uma *thread* para processá-la. O número máximo de

threads simultaneamente em execução é controlado pelo servidor, buscando garantir uma utilização controlada de recursos. No modelo projetado, as *threads* utilizadas são mantidas em um *pool* e controladas através de um gerente (*thread manager*).

Diante do cenário descrito, cada requisição recebida passa por um conjunto de filas de espera para que possa ser efetivamente processada. Tais filas compreendem a fila de espera por uma *thread* disponível, a fila de espera para o acesso a um *container*, e, por fim, a fila de espera pela obtenção de um acesso individual a uma instância (sincronização). Um modelo natural para representar tal cenário é, de fato, uma rede de filas composta por $1 + C + C * M$ estações, sendo que 1 corresponde ao *thread manager*, C é o número de *containers* de *entity beans* que o servidor possui (um para cada tipo de *entity bean* instalado) e M é o número máximo de instâncias de um mesmo tipo de *bean* que podem estar simultaneamente ativas.

Durante seu projeto, Lladó assume algumas simplificações de forma a tornar o modelo matematicamente tratável. Em primeiro lugar, assume que os tempos de serviço associados às estações representadas podem ser adequadamente representados através de variáveis aleatórias exponenciais. Assume, ainda, que as disciplinas de acesso associadas às filas são *First-Come-First-Served* (FCFS) e que as probabilidades de chegada de requisições para os C *containers* são constantes e iguais.

Finalmente, para simplificar o modelo originalmente proposto é utilizado um método conhecido como *Flow Equivalent Server* (FES), o qual permite uma redução no número de nós através da agregação de sub-redes em nós mais complexos.

Para validar o modelo proposto, Lladó desenvolveu um programa utilizando QNAP2, uma linguagem de modelagem desenvolvida pelo INRIA (*Institut Nacional de Reserche en Informatique et Automatique*, França), a qual provê um simulador de eventos discretos. Os resultados obtidos através do simulador apresentaram uma diferença de menos de 5% com relação à solução analítica do modelo a um nível de confiança de 95%.

A despeito da indiscutível relevância do trabalho realizado, há uma importante consideração a ser feita. Os resultados obtidos não foram comparados com medições realizadas em servidores de aplicação reais. Todo o trabalho desenvolvido foi baseado em um modelo teórico de um ambiente de suporte à componentes EJB, não incorporando particularidades de nenhum produto específico. Trabalhos de diversos pesquisadores da área [Gorton e Liu 2003] [Ran et al. 2001] [Liu et al. 2002a], demonstram que o desempenho de um ambiente de suporte à tecnologia EJB é fortemente influenciado pelos detalhes incorporados à implementação de cada fornecedor. A ausência de representação de tais detalhes dificulta a utilização do modelo proposto em previsões de desempenho realizadas para servidores de aplicação específicos. Por outro lado, a incorporação de muitos detalhes em modelos analíticos pode inviabilizar a sua solução.

Há alguns aspectos importantes comuns entre trabalhos desenvolvidos por Lladó e por Souza [Souza et al. 2006a] [Souza et al. 2006b] (detalhados na presente dissertação). Dentre estes aspectos destacam-se a confecção de

modelos formais, a utilização das técnicas de análise e simulação e o foco na tecnologia EJB. Contudo, há diferença significativas entre os modelos propostos. Dentre as principais, destacam-se:

- Formalismo: Lladó utiliza redes de fila, e Souza redes de Petri estocásticas;
- O tipo de componente EJB: Lladó considera *entity beans*, enquanto Souza desenvolveu seus modelos com base no suporte do servidor JBoss a *session beans*;
- Mecanismo avaliado: Lladó avalia o mecanismo de invocação de métodos em *entity beans*; Souza investiga o mecanismo de *pooling* de instâncias;
- Métricas utilizadas: os trabalhos desenvolvidos por Lladó consideram basicamente a métrica de *throughput*. Ao longo de seus trabalhos Souza considera, além da métrica citada, a utilização de CPU, o número de instâncias criadas e o tempo de resposta;
- Implementação base: Lladó baseia-se em um modelo de servidor de aplicação teórico, enquanto que os modelos desenvolvidos por Souza são baseados no servidor de aplicação JBoss;
- Validação dos resultados de desempenho: Lladó valida seus modelos através da comparação com resultados obtidos em experimentos de simulação. Por outro lado, Souza utiliza os resultados de medição para este mesmo fim.

As semelhanças e diferenças apresentadas demonstram a relevância e contribuição de cada pesquisa, assim como a independência existente entre elas.

Outro trabalho relevante na área de modelagem analítica de servidores de aplicação foi desenvolvido por Kounev e Buchmann [Kounev e Buchmann 2003]. Este trabalho tem como objetivo comprovar a viabilidade de utilização desta técnica de avaliação de desempenho no planejamento de capacidade de ambientes corporativos. Para tanto, Kounev projeta um modelo baseado em Non-Product-Form Queueing Networks representando a execução de uma aplicação J2EE realística (SPECjAppServer2002[1]) em um ambiente composto por um *cluster* de servidores de aplicação acessando uma base de dados compartilhada.

No desenvolvimento do modelo de desempenho, os tipos de transação suportados pela aplicação são identificados e agrupados em classes de requição, de acordo com suas as características. Para cada classe de requisição, os recursos de hardware utilizados são identificados (basicamente, CPU e disco) e medidos em um ambiente real composto por um cluster de servidores de aplicação WebLogic e uma base de dados Oracle. Esses recursos são representados no modelo sob a forma de nós e os dados referentes à sua utilização são usados na parametrização.

[1] SPECjAppServer2002 é uma aplicação padrão de *benchmark* utilizada para avaliar o desempenho e a escalabilidade de servidores de aplicação J2EE que foi originalmente desenvolvida pela Sun [SUN] em conjunto com fornecedores de servidores de aplicação e sob o nome de ECPerf [ECPERF].

Para validar o modelo proposto, Kounev realiza predições de desempenho considerando diferentes cenários de utilização (baixa, média e alta cargas) e diferentes tamanhos de *cluster*. Tais predições envolvem o cálculo do *throughput* do sistema, da utilização da CPU dos servidores de aplicação e da base de dados, e do tempo de resposta. Os resultados obtidos nesses experimentos são comparados com medições reais.

De forma geral, os resultados referentes ao *throughput* e à utilização de CPUs foram bastante satisfatórios em todos os cenários considerados. De fato, o erro médio da modelagem com relação à medição foi de cerca de 2% para o *throughput* e de 6% para a utilização de CPU, demonstrando, definitivamente, a validade do modelo e a viabilidade de sua utilização para o cálculo destas métricas.

Por outro lado, o erro médio para tempo de resposta foi de 18%, chegando, para um cenário e transação específicos, a atingir mais de 35%. Estas diferenças demonstram uma limitação no modelo, a qual, conforme o próprio autor identifica, está relacionada à ausência de representação da concorrência por recursos de software, tais como, conexões com bases de dados, instâncias de componentes e *threads*.

Em suma, o trabalho desenvolvido por Kounev representa uma importante contribuição para a área de avaliação de desempenho de servidores de aplicação, uma vez que comprova a viabilidade da utilização da técnica de modelagem analítica no planejamento de capacidade de ambientes J2EE envolvendo aplicações corporativas reais e complexas. Dentre as semelhanças entre este trabalho e aquele a ser apresentado na presente dissertação destacam-se a utilização de modelos analíticos baseados em espaço de estados, a validação dos modelos através de dados de medições reais e a semelhança entre as métricas consideradas. Contudo, tais trabalhos possuem diferenças fundamentais, as quais merecem destaque.

A primeira diferença entre esses trabalhos diz respeito ao tipo de aplicação considerada. Conforme mencionado anteriormente, os modelos apresentados ao longo desta dissertação consideram uma aplicação simples, composta apenas por *stateless session beans*. Esta aplicação possui uma única transação e não envolve acessos a bases de dados. Por outro lado, o modelo desenvolvido por Kounev considera uma aplicação mais completa que é composta por *session* e *entity beans* e disponibiliza um conjunto de transações para seus usuários, as quais envolvem acessos de leitura e escrita a uma base de dados.

Outra diferença relevante, diz respeito à configuração dos ambientes utilizados na parametrização e validação dos modelos. O presente trabalho considera um ambiente composto por um único servidor de aplicação rodando JBoss[2], estando este ambiente devidamente refletido nos modelos concebidos. Embora os modelos projetados não possibilitem variações no número de servidores utilizados, eles são parametrizáveis, permitindo variações na configuração do servidor representado. O conjunto de parâmetros configuráveis

[2] Opções de cluster não foram consideradas dadas as limitações dos ambientes operacionais utilizados.

varia de modelo para modelo, podendo compreender o tamanho do *pool* de instâncias, o número de CPUs do servidor e o tamanho do *pool* de *threads*.

O ambiente utilizado por Kounev é composto por um *cluster* de servidores WebLogic de tamanho variável acessando uma base Oracle. O modelo projetado reflete bem este ambiente. Contudo, para fins de simplicidade, este modelo possui um único parâmetro que pode ser configurado, o tamanho do *cluster*, ou seja, o número de servidores que o compõe. A configuração de cada um destes servidores no modelo, contudo, é estática, sendo incapaz de refletir diretamente alterações nas configurações dos servidores reais. Desta forma, caso haja necessidade, por exemplo, de uma avaliação do efeito no desempenho provocado por um aumento no tamanho do *pool* de *threads*, será necessário medir novamente os tempos de serviço refletidos no modelo.

Uma abordagem interessante proposta por [Liu et al. 2004] defende, também, a utilização de modelos analíticos em redes de fila na predição de desempenho de aplicações que utilizam *middleware* ainda em tempo de projeto. Esta abordagem compreende os seguintes passos:

1. Modelagem: durante esta etapa é construído um modelo geral representando os principais componentes do *middleware* escolhido e relação entre eles. Os componentes devem ser lógicos (ao contrário do modelo proposto por Kounev, no qual os componentes representados eram físicos, tais como CPUs e discos) e devem ser representados em um alto nível de abstração.
2. Calibragem: este passo compreende o mapeamento das atividades realizadas pelos componentes arquiteturais utilizados no projeto de alto nível da aplicação em atividades de baixo nível realizadas pelos componentes do *middleware* representados no modelo.
3. Caracterização: determina a freqüência de ocorrência de cada uma das atividades consideradas, as quais foram mapeadas em termos de utilização de infra-estrutura no passo anterior.
4. *Benchmarking*: determina o custo, em termos de tempo de serviço, de cada uma das operações de baixo nível representadas. Para determinar este custo, uma aplicação trivial que envolva tais atividades é projetada e construída, e os tempos correspondentes são medidos.
5. População: compreende a substituição no modelo dos dados obtidos nos passos anteriores.

Para validar a abordagem proposta, Liu projeta um modelo de desempenho para representar um ambiente de suporte a aplicações J2EE. Este modelo genérico possui três componentes de alto nível: um fila de requisições, uma fila representando um *container* EJB e uma fila representando uma fonte de dados. Este modelo é utilizado na realização de predições de desempenho para uma aplicação simples referenciada como *Stock-Online* (mencionada anteriormente neste trabalho).

Com o modelo estabelecido, são realizadas as etapas de calibragem e caracterização, nas quais as características referentes ao uso da aplicação em questão são determinadas. Na etapa de *benchmarking*, os dados necessários ao modelo são obtidos considerando-se um ambiente composto por um único

servidor de aplicação WebLogic acessando uma base Oracle. Tais dados são alimentados no modelo durante a etapa de população, deixando o modelo completamente preparado para a realização de experimentos de predição.

A etapa de validação é concluída comparando-se os dados referentes a tempos de resposta obtidos a partir do modelo com dados equivalentes obtidos através de medições realizadas em cenários distintos, envolvendo entre 50 e 500 usuários. Os resultados de tais comparações demonstram diferenças entre medição e análise variando entre 5% e 9%, comprovando assim, a acurácia do modelo e validando a abordagem proposta.

Um aspecto particular e interessante do trabalho desenvolvido por Liu é a possibilidade de predição de desempenho de aplicações que ainda não foram implementadas, bastando, para tanto, realizar um mapeamento das interações entre os componentes arquiteturais da aplicação para o *middleware* no qual esta está baseada. Esta característica, indubitavelmente, diferencia este trabalho dos demais trabalhos publicados na área.

A despeito da relevância deste trabalho, uma deficiência a ser mencionada é a desconsideração de métricas de desempenho importantes, tais como *throughput* e taxas de utilização de recursos, tais como CPU. Por não representar explicitamente recursos físicos, a identificação de gargalos fica comprometida. Esta é uma diferença importante desse trabalho com relação ao apresentado na presente dissertação. De fato, ao longo dos diversos modelos projetados ao longo do presente trabalho de mestrado, métricas distintas são consideradas e avaliadas. Desta forma, enquanto no modelo mais simples apenas uma métrica de desempenho, o número de instâncias criadas, é considerada, nos modelos mais completos métricas fundamentais, tais como *throughput*, tempo de resposta e disponibilidade, também são incluídas e avaliadas.

Outra diferença importante entre o trabalho de Lui e o presente diz respeito à possibilidade de alteração nas configurações do servidor. Assim como no modelo de Kounev, a configuração do servidor de aplicação no modelo de Liu é estática, sendo incapaz de refletir diretamente alterações nas configurações, como por exemplo um aumento no tamanho do *pool* de instâncias. Para que o efeito desta variação seja percebido é necessária a realização de um novo *benchmarking*. Por outro lado, no presente trabalho o servidor modelado é parametrizável, e o efeito de uma alteração dessa natureza pode ser avaliado através do ajuste no parâmetro correspondente ao tamanho do *pool*.

Por fim, em um trabalho diferente, também baseado em modelagem analítica, [Chen 2002] propõe um modelo para predição de desempenho de servidores de aplicação composto, basicamente, por uma equação matemática obtida empiricamente. Esta equação relaciona o tempo de resposta com o número de clientes e com o tamanho do *pool* de *threads*, e envolve três coeficientes que são específicos para um dado servidor de aplicação e podem ser obtidos através da realização de experimentos de medição.

Para obter o valor destes coeficientes, Chen sugere a utilização de aplicações que exercitem a infra-estrutura fornecida pelo servidor, sem, contudo, impor um grande *overhead* adicional. O intuito é que a aplicação

utilizada represente um mínimo de impacto de forma que se possa atribuir os resultados de desempenho obtidos diretamente ao servidor em avaliação. Na validação de seu modelo matemático, Chen utiliza a aplicação identidade discutida anteriormente, e submete a esta uma carga variável, anotando os tempos de resposta obtidos. Em seguida, Chen utiliza um software estatístico para realizar o ajuste da equação e adequá-la aos valores medidos.

A partir da equação proposta, já ajustada com os coeficientes obtidos, Chen calcula o tamanho ideal do *pool* de *threads* para diferentes cenários envolvendo números de usuários distintos. Os resultados obtidos através da solução da equação são, de fato, bastante próximos daqueles verificados nos experimentos de medição realizados. Chen conclui a validação de seu modelo matemático apresentando curvas relacionando tempo de resposta e tamanho do *pool* de *threads* para diferentes quantidades de usuário.

A partir dos resultados apresentados constata-se que o modelo matemático concebido apresenta uma razoável aproximação na maioria dos casos. Contudo, observa-se, em alguns cenários, uma divergência clara em termos dos tempos de resposta calculados e medidos, chegando-se, em casos isolados, a erros da ordem de mais de 50%. Um problema ainda em aberto no modelo em questão é uma forma de incorporar aspectos relativos às características arquiteturais das aplicações. Outro problema importante é que o único parâmetro de configuração do servidor de aplicação diretamente representado no modelo matemático é o tamanho do *pool* de *threads*. Quaisquer outras alterações nas configurações do servidor de aplicação, tais como tamanho dos *pools* de instância ou utilização de *cluster*, implicariam em uma necessidade de recalculo dos coeficientes. Por fim, um último problema que merece destaque é que o modelo proposto só contempla uma métrica, que é o tempo de resposta, não sendo possível a avaliação de nenhum outro critério de desempenho. Desta forma, informações relevantes referentes à utilização de recursos e *throughput* não são consideradas, restringindo, bastante, a utilidade do modelo.

O trabalho desenvolvido por Chen é claramente diferente do presente trabalho de mestrado, sendo o fato de ambos utilizarem modelos analíticos na avaliação de desempenho de servidores de aplicação, a única semelhança entre eles.

Na seqüência são apresentados alguns trabalhos de avaliação de desempenho de servidores de aplicação utilizando modelos de simulação.

3.3.2 Simulação

Segundo [Bagrodia e Shen 1991], as interações complexas entre software, hardware e componentes humanos de muitos sistemas distribuídos tipicamente impedem a construção ou solução de modelos analíticos detalhados. Mesmo quando modelos analíticos podem ser construídos, a ausência de dados suficientes impede uma análise compreensiva das questões de desempenho. A alternativa usual para modelos analíticos é o uso de simulação.

Conforme mencionado na Seção 2.2.2, nos modelos de simulação, a descrição de um sistema é embutida em um programa de computador, que ao executar simula a sua operação. Desta forma, ao contrário de modelos

analíticos, os modelos de simulação são executados e não solucionados. A execução de um modelo de simulação envolve o disparo repetido das operações do modelo, criando uma "história" artificial. Durante a execução da simulação, dados referentes às métricas configuradas são coletados e utilizados na estimativa de seus valores reais. Um experimento de simulação deve continuar em execução até que o intervalo de confiança para a média de cada métrica configurada no modelo atinja uma dimensão estabelecida. À despeito da utilidade da técnica de simulação, algumas observações são pertinentes. Primeiramente, deve-se ressaltar que a quantidade de detalhes embutidos no modelo de simulação pode inviabilizar a obtenção de resultados em um tempo razoável. Outro fator importante a ser considerado é o dimensionamento do intervalo de confiança. Quanto menor este intervalo, maior o tempo necessário à conclusão dos experimentos de simulação.

Embora, sem dúvida, a simulação seja uma técnica de avaliação de desempenho bastante relevante, existem poucos trabalhos publicados na área de simulação de servidores de aplicação J2EE.

Dentre esses trabalhos, destacam-se os de [McGuinness et al. 2004] [McGuinness e Murphy 2005]. Em [McGuinness et al. 2004], McGuinness descreve um simulador para um servidor de aplicação com suporte à EJB que roda no Ptolemy II, um simulador genérico criado e mantido na Universidade de Berkley.

O modelo representa um único servidor executando *stateless session beans*. Este modelo pode ser configurado através de um conjunto de parâmetros de entrada informados pelo usuário, os quais incluem parâmentros de configuração de hardware e software. Os parâmetros de hardware incluem tamanho de memória, número de CPUs e fatores escalares representando velocidade de CPU e de I/O. Os parâmetros de software incluem o tamanho do *pool* de *threads* e o número de componentes instalados. Outros parâmetros controlam a execução dos experimentos de simulação. Dentre eles estão o tempo de *warm-up* do sistema, a duração do experimento (tempo simulado), o número de transações realizadas, o número de usuários por transação e parâmetros de configuração das transações, tais como consumo de CPU, memória e I/O.

Dada a ampla gama de parâmetros configuráveis (fatores) diversos experimentos podem ser projetados. Os experimentos descritos neste trabalho investigam os efeitos provocados por mudanças nas configurações de hardware (número e velocidade de CPUs) e de software (tamanho do *pool* de *threads* e número de componentes instalados), assim como nos fatores de carga. Os resultados destes experimentos foram analisados em termos de tempo de resposta, *throughput* e utilização de recursos. Embora tais resultados apresentassem a mesma tendência observada em servidores de aplicação reais, o modelo não foi validado. É importante observar, ainda, que o servidor de aplicação modelado não reflete nenhuma implementação específica.

Em outro trabalho [McGuinness e Murphy 2005], McGuinness descreve uma versão do modelo de simulação do trabalho anterior [McGuinness et al. 2004] incluindo múltiplos servidores. Esta nova versão foi criada utilizando Hyperformix Workbench. Os experimentos executados com esta nova versão

são similares aos experimentos anteriores, à exceção de um novo experimento que examina o efeito de mudanças no número de servidores que compõem o *cluster*. Os resultados deste último experimento mostram o esperado: quanto mais servidores são adicionados menor o tempo de resposta e maior o *throughput*. Contudo, a partir de um certo número de servidores os ganhos em termos destas duas métricas começam a diminuir possibilitando assim a escolha de uma relação custo-benefício adequada. Novamente, o modelo apresentado é genérico e os resultados obtidos não foram validados.

Em suma, embora os modelos publicados sejam extremamente flexíveis e os resultados obtidos sejam relevantes, a ausência de validação compromete, sensivelmente, os trabalhos mencionados.

3.4 Considerações Finais

Este capítulo apresentou os principais trabalhos relacionados à área de avaliação de desempenho de servidores de aplicação J2EE. Diversos trabalhos baseados nas técnicas de medição, modelagem analítica e simulação são descritos e os principais resultados obtidos são analisados. Por fim, estes trabalhos são comparados com o presente trabalho de mestrado ressaltando-se as principais diferenças entre eles.

CAPÍTULO

4

Modelagem de Desempenho de Servidores de Aplicação

"It is not what you say, by how you say it"

A. Putt.

Este capítulo apresenta uma abordagem para modelagem de desempenho de servidores de aplicação utilizando como formalismo as redes de Petri estocásticas. O capítulo é estruturado de forma que cada atividade da abordagem é apresentada em uma seção distinta.

4.1 Abordagem para Modelagem de Desempenho

Projetos de avaliação de desempenho devem ser fundamentados em abordagens sistemáticas que podem ser estruturadas através de fluxos de atividades bem definidos. Um fluxo especifica uma seqüência lógica de atividades, as quais estão associadas a conjuntos de artefatos utilizados como entrada e produzidos como saída. Estes fluxos são, em geral, representados através de diagramas de atividades.

O fluxo apresentado na Figura 4.1 representa a abordagem proposta. Cada atividade definida consiste em uma unidade de trabalho que produz um resultado significativo dentro do contexto do projeto de avaliação de desempenho.

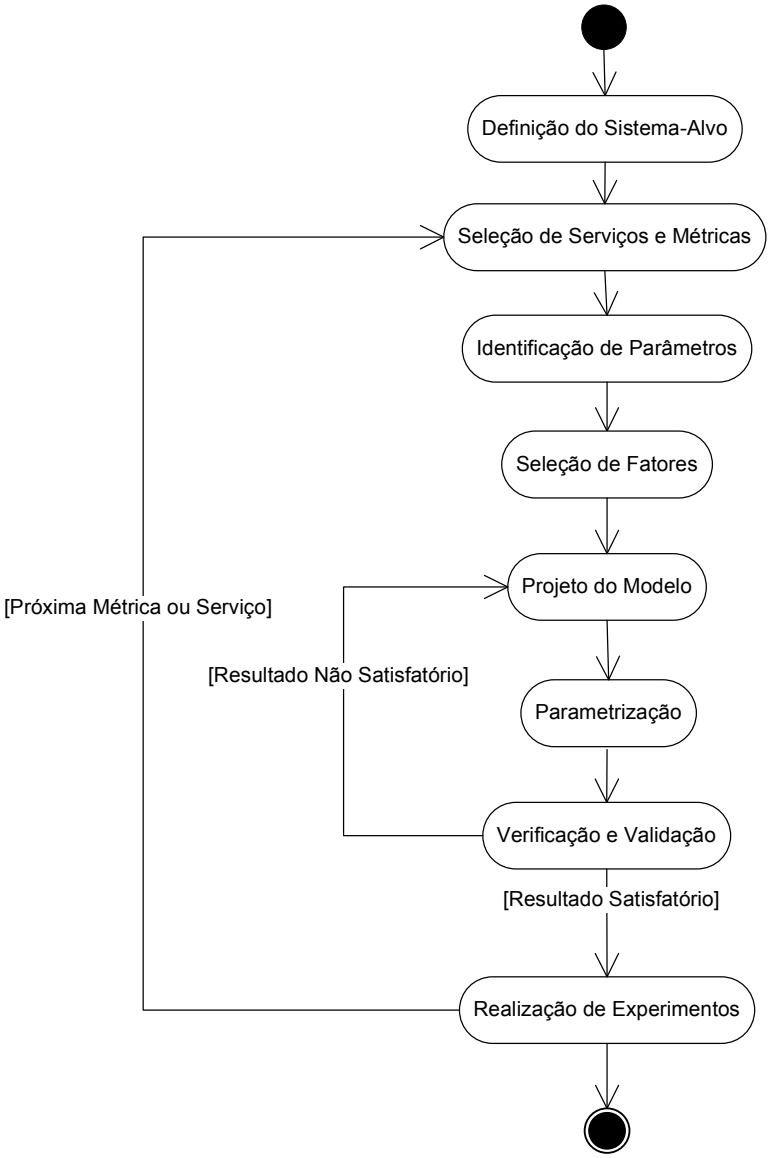


Figura 4.1. Abordagem Utilizada.

É importante constatar que, segundo a abordagem proposta, projetos de avaliação de desempenho de servidores de aplicação podem ser particionados em diferentes iterações, cada uma avaliando um serviço fornecido. As seções seguintes detalham as atividades que compõem a abordagem.

4.2 Definição do Sistema-Alvo

A primeira atividade realizada em qualquer projeto de avaliação de desempenho consiste no estabelecimento de um escopo adequado, ou seja, na delimitação das fronteiras do sistema-alvo do projeto. Esta delimitação deve ser realizada com base nos objetivos estabelecidos, levando-se em consideração fatores de natureza técnica, orçamentária e restrições de cronograma. A delimitação do sistema avaliado contextualiza todas as demais atividades desenvolvidas ao longo do projeto.

4.3 Seleção de Serviços e Métricas

A finalidade desta atividade é selecionar, no âmbito do sistema delimitado na seção anterior, os serviços que serão considerados no projeto de avaliação de desempenho. Esta seleção deve ser realizada com base no impacto esperado destes serviços sobre desempenho global do sistema.

Projetos que envolvem a avaliação de mais de um serviço devem ser particionados em iterações, de forma que, a cada iteração um único serviço seja selecionado, modelado e avaliado. Um modelo completo de desempenho pode ser obtido integrando-se os modelos que representam os serviços individuais. Esta integração, contudo, não é sempre recomendada (nem, muitas vezes, factível), uma vez que o modelo completo pode se tornar muito grande, inviabilizando a obtenção de soluções analíticas e tornando lentas as simulações.

A avaliação de um serviço deve ser realizada com base em um conjunto de critérios de desempenho, chamados métricas. A seleção adequada do conjunto de métricas é um importante fator de sucesso para um projeto. Para se selecionar as métricas a serem utilizadas, os possíveis resultados da execução de cada serviço devem ser levantados. Para cada resultado possível, uma ou mais métricas devem ser definidas. De forma geral, esses resultados correspondem a uma execução correta, incorreta, ou mesmo, a uma recusa de execução do serviço considerado.

As métricas associadas à execuções corretas podem estar relacionadas ao tempo necessário ao atendimento de uma requisição (tempo de resposta), à taxa de requisições efetivamente atendidas (*throughput*) ou à quantidade de recursos consumidos durante a execução do serviço (utilização de recursos) (veja Seção 2.2.1). Por outro lado, métricas relacionadas à execuções incorretas (erros), chamadas de *métricas de confiabilidade*, estão normalmente associadas à probabilidade de ocorrência de uma dada classe (categoria) de erro. Por fim, as métricas ligadas a recusas de execução (falhas), referenciadas como *métricas de disponibilidade*, geralmente envolvem a probabilidade de ocorrência de falhas e o cálculo do tempo médio entre elas.

Dois importantes princípios devem ser observados na seleção do conjunto das métricas a serem avaliadas: completude e não-replicação. Por completude entende-se que as métricas selecionadas devem refletir todos os aspectos de interesse sobre o serviço em estudo, ou seja, devem ser suficientes para uma ampla compreensão das condições operacionais do serviço. Contudo, a inclusão de uma métrica em um projeto de avaliação de desempenho normalmente implica em um custo adicional de recursos. Desta forma, é importante também que o conjunto de métricas selecionadas não possua métricas distintas que forneçam a mesma informação. Este princípio é conhecido como não-replicação.

O conjunto de métricas estabelecido também pode ser utilizado no particionamento do projeto em iterações. Versões mais simplificadas dos modelos de desempenho podem ser projetadas de forma a contemplar um subconjunto dessas métricas. Nas iterações seguintes, os modelos desenvolvidos são reavaliados para incluir as demais métricas consideradas.

4.4 Identificação de Parâmetros

Após a seleção do serviço e das métricas correspondentes, as variáveis que têm impacto no desempenho, chamadas de *parâmetros*, devem ser identificadas e relacionadas. A lista produzida deve ser ampla e deve incluir parâmetros de sistema e parâmetros de carga. Os parâmetros de sistema estão relacionados à configuração do sistema em si, abrangendo tanto configurações de hardware, quanto de software. Por sua vez, os parâmetros de carga estão relacionados à caracterização da carga de trabalho submetida. Como exemplos de parâmetros de carga podem ser citados o número de usuários simultâneos e a taxa média de requisições por unidade de tempo.

A completude da lista de parâmetros é importante para o sucesso do projeto como um todo, uma vez que parâmetros não identificados representam variáveis externas desconsideradas e que podem interferir na precisão dos resultados. Além disto, a existências de parâmetros não identificados pode impedir a replicação dos experimentos realizados, dificultando a aceitação dos resultados obtidos. É importante ressaltar que, embora esta lista deva ser completa, parâmetros redundantes podem ser seguramente eliminados.

4.5 Seleção de Fatores

Dentre os parâmetros identificados, deve-se selecionar aqueles que serão variados durante os experimentos, os quais são chamados de fatores. A seleção do conjunto de fatores deve ocorrer de acordo com os objetivos estabelecidos para o projeto.

Projetos centrados na realização de testes de desempenho de um sistema têm como finalidade principal entender o comportamento deste sistema sob diferentes condições de carga de trabalho. Para estes projetos, os parâmetros de sistema devem ser fixos, refletindo a configuração do ambiente no qual o sistema está inserido, ou seja, os parâmetros de sistema não devem ser considerados como fatores. Por outro lado, os parâmetros de carga devem variar de forma a representar diferentes demandas.

Por outro lado, projetos que visam avaliar eventuais migrações de plataforma normalmente utilizam como fatores os parâmetros de sistema. Nestes projetos, os fatores são variados de forma a representar cada uma das plataformas consideradas na avaliação. Um exemplo possível é um projeto que vise dimensionar o número de processadores ideal para atender a uma determinada demanda. Em projetos desta natureza, os parâmetros de carga devem ser configurados de forma a representar essa demanda, não sofrendo alterações ao longo dos experimentos. Por outro lado, o número de processadores, que é um exemplo típico de parâmetro de sistema, pode ser utilizado como fator, variando ao longo dos experimentos. Os diferentes valores assumidos representam as configurações de máquina avaliadas, permitindo a determinação de uma relação custo/benefício ideal.

Projetos de avaliação de desempenho devem iniciar com um conjunto pequeno de fatores, simplificando a identificação do impacto de cada um no desempenho do sistema. Diversas iterações podem ser realizadas caso a avaliação de um conjunto maior de fatores seja conveniente.

4.6 Projeto do Modelo

Nesta atividade são projetados modelos de desempenho para os servidores de aplicação, utilizando redes de Petri estocásticas. A elaboração destes modelos é uma atividade complexa e envolve, além do conhecimento das técnicas de modelagem, um amplo entendimento dos mecanismos internos envolvidos. Os modelos projetados devem representar os fatores selecionados e permitir a derivação das métricas estabelecidas. O principal artefato utilizado como entrada para esta atividade é a documentação do sistema modelado (incluindo o seu código fonte), enquanto que o principal artefato produzido como saída consiste nos próprios modelos de desempenho.

Uma ferramenta auxiliar importante durante a construção dos modelos são os *profilers*. Estas ferramentas são utilizadas para permitir a monitoração e o rastreamento de eventos que ocorrem em tempo de execução, podendo fornecer informações sobre trechos de código que consomem mais tempo de CPU ou que alocam mais memória. Contudo, os tempos medidos com a utilização direta dos *profilers* não devem considerados como tempos absolutos (e conseqüentemente, não devem ser utilizados nos modelos desenvolvidos), mas como valores relativos, uma vez que o próprio *profiler* em si representa um custo adicional de desempenho.

Na abordagem proposta, os *profilers* são utilizados para rastrear o fluxo de requisições dentro do servidor de aplicação facilitando a identificação das atividades relevantes realizadas pelos componentes que implementam o serviço avaliado. Contudo, a efetiva medição dos tempos associados à estas atividades é realizada através da instrumentação manual do servidor visando, assim, minimizar qualquer interferência externa.

O processo de modelagem em si inicia-se com a construção de um modelo de alto nível não orientado a desempenho, chamado modelo funcional ou abstrato. Este modelo visa promover um aprofundamento do entendimento do servidor de aplicação e contextualizar o projeto. O modelo funcional deve representar a relação entre o serviço sendo modelado e os demais componentes do servidor.

Nesse modelo, as atividades representadas possuem uma granulosidade grossa e são modeladas através de transições genéricas. Os estados observados neste nível de abstração são representados através de lugares. Nenhuma informação à cerca das métricas selecionadas é representada, deixando claro que este modelo não deve ser utilizado para a realização de experimentos de avaliação de desempenho. Embora o modelo abstrato não represente informações temporais, sua construção também é auxiliada pelas informações relativas ao rastreamento das requisições, as quais são fornecidas pelas ferramentas de *profiling*.

O primeiro modelo de desempenho é projetado a partir do modelo abstrato. Este modelo representa, além do próprio serviço em si, os componentes do modelo abstrato que interagem diretamente com ele. No modelo de desempenho, o serviço selecionado é decomposto em um conjunto de componentes internos. Cada componente é representado através de uma subrede, onde as principais atividades realizadas são representadas através de transições e os estados assumidos são representados através de lugares.

Cada transição representada pode ser (1) imediata, caso o tempo consumido pela atividade correspondente não seja relevante, (2) determinística, caso o tempo da atividade seja constante (por exemplo, na modelagem de *timeouts*), (3) exponencial, nos demais casos. Desta forma, nos modelos de desempenho inicialmente projetados, é assumido que quaisquer atividades associadas com tempo variável serão modeladas através de transições exponenciais.

Assim como as atividades e estados, os recursos utilizados pelos componentes devem ser representados em suas subredes. Os tipos de recursos são representados como lugares, enquanto que a quantidade de recursos disponíveis é representada através de *tokens*. Cada atividade que necessite de um dado recurso deve representar esta dependência através de um arco de entrada ligando o lugar que representa o tipo do recurso à transição correspondente. A quantidade de recursos necessária é representada através da multiplicidade deste arco. A liberação dos recursos utilizados deve ser representada através de arcos de saída ligando a transição responsável pela liberação aos lugares que representam os tipos dos recursos liberados. A quantidade de recursos liberada é indicada pela multiplicidade destes arcos.

Além das transições, lugares e *tokens* utilizados na representação de atividades, estados e recursos, as subredes devem incorporar outros lugares e transições para permitir a avaliação das métricas selecionadas na análise. Estas métricas são derivadas através do cálculo de probabilidades e valores médios associados à existência de *tokens* em determinados conjuntos de lugares.

Por fim, a rede que representa o serviço como um todo é uma composição das subredes que representam os componentes, na qual a interação entre estes é representada através de arcos.

Novos modelos podem ser derivados desse primeiro modelo de desempenho através da inclusão de novas métricas, do fracionamento ou da composição das atividades representadas e da possibilidade de contemplação de novos cenários.

Outra situação na qual um novo modelo pode ser necessário é quando a atividade de validação indica que os resultados obtidos a partir do modelo não estão suficientemente próximos dos resultados medidos. Tais cenários são possíveis devido a erros na modelagem ou à representação inadequada das atividades modeladas através das transições exponenciais.

Como mencionado anteriormente, uma importante suposição implícita nos procedimentos de modelagem e de parametrização propostos é a de que as atividades temporizadas são exponencialmente distribuídas. Contudo, muitas vezes a distribuição empírica dos valores medidos para a duração de uma atividade não é exponencial. Diante deste cenário, uma possível solução é a utilização de uma técnica de refinamento que compreende a realização de testes para averiguação da qualidade desta suposição para cada atividade considerada e o ajuste na representação utilizada no modelo de atividades cuja duração não tenha natureza exponencial. Tais atividades devem ser representadas através de combinações de exponenciais referenciadas como *Phase-Type distributions*, conforme mencionado na Seção 2.3.5.

4.7 Parametrização

Para que os modelos projetados possam ser utilizados em experimentos de avaliação de desempenho, eles devem ser *parametrizados*. A parametrização compreende a configuração dos parâmetros associados às transições exponenciais com os valores dos tempos médios medidos para atividades correspondentes. Em suma, a abordagem propõe um casamento entre o primeiro momento associado à variável aleatória que representa a duração medida para uma atividade e o primeiro momento associado à variável aleatória que representa a sua duração no modelo.

A parametrização envolve a realização de experimentos de medição projetados para determinar as durações das atividades representadas no modelo por transições exponenciais. Os resultados destes experimentos são utilizados na composição de amostras. Cada amostra contém, para cada atividade medida, n observações. Para uma atividade, a média das n observações que compõem a amostra é referenciada como média amostral e fornece uma estimativa para a duração média real da atividade correspondente (média populacional). Dadas k amostras, k estimativas diferentes podem ser obtidas, restando então uma questão, como se pode determinar uma única estimativa representativa a partir das k estimativas obtidas?

De fato, uma estimativa perfeita para o tempo médio de uma atividade não pode ser obtida a partir de um número finito de amostras de tamanho finito. Diante deste cenário, deve-se recorrer a limites probabilísticos e determinar um intervalo (intervalo de confiança) que possua uma alta probabilidade (nível de confiança) de conter o tempo médio real (média populacional) para a atividade medida. O teorema central do limite pode ser utilizado para a determinação do número de amostras e de observações necessários à determinação deste intervalo. Este teorema estabelece que, se as observações de uma amostra são independentes e vêm de uma mesma população que tem média μ e desvio padrão σ, então, a média amostral, para amostras grandes, é uma variável aleatória com distribuição aproximadamente normal, com média μ e desvio padrão $\sigma/\sqrt{n}$. Desta forma, para amostras grande o desvio entre as médias amostrais tende a zero, sendo suficente utilizar uma única amostra para o cálculo da média. É importante destacar que por amostra grande entende-se amostras com pelo menos 25 a 30 observações [Jain 1991].

Utilizando o teorema central do limite, a um nível de 100(1-α)%, o intervalo de confiança para o tempo médio real de uma atividade é dado por [Jain 1991]:

$$\left(\bar{x} - z_{(1-\alpha)/2} \times s/\sqrt{n}, \bar{x} + z_{(1-\alpha)/2} \times s/\sqrt{n} \right) \qquad (4.1)$$

onde,

$\bar{x}$: média amostral;

s: desvio amostral;

n: tamanho da amostra;

α: nível de significância; e

$z_{(1-\alpha)/2}$: $(1-\alpha/2)$-quantil de uma distribuição normal padrão.

A partir da formulação matemática apresentada na Equação (4.1) percebe-se que o tamanho do intervalo de confiança depende do tamanho da amostra considerada. Quanto maior a amostra, menor o intervalo de confiança e mais preciso o resultado. Contudo, amostras muito grandes requerem maior esforço e quantidade de recursos. Desta forma, é importante que se possa determinar qual o menor tamanho de amostra capaz de provê o nível de confiança desejável.

Para que se possa estimar a média populacional com precisão (erro máximo relativo) de $\pm r\%$ e nível de confiança de 100(1-α)%, o número de observações requeridas pode ser calculado através da seguinte equação [Jain 1991]:

$$n = \left(\frac{100 \times z_{(1-\alpha/2)} \times s}{r \times \bar{x}} \right)^2 \qquad (4.2)$$

Para fins deste trabalho o dimensionamento da amostra será realizado com base em um intervalo de confiança com precisão de $\pm 10\%$ e nível de confiança de 95%, uma vez que estes são os parâmetros default utilizados nos experimentos de simulação realizados na ferramenta TimeNet [Zimmermann 2001].

Por fim, são necessárias algumas considerações à cerca das condições nas quais são realizados os experimentos necessários à atividade de parametrização. Uma primeira consideração diz respeito às condições de carga nas quais os tempos associados às transições são medidos. Uma abordagem possível é medir o tempo considerando-se um único cliente solicitando, repetidamente, a execução do serviço. É importante observar que, nesta situação, cada requisição realizada pelo cliente (considerando requisições síncronas) não concorre com nenhuma outra. Desta forma, o tempo medido não incorpora nenhuma fração relativa à concorrência por recursos, ou seja, o tempo de fila para o atendimento das requisições pode ser desconsiderado.

Outra abordagem é realizar a medição do tempo das atividades dentro de cenários reais de uso, ou seja, medir as atividades com o serviço sendo prestado simultaneamente a vários clientes. Nesta abordagem, os tempos medidos possuem dois componentes representativos, uma fração correspondendo ao tempo de espera pela obtenção de recursos (*e.g.* processador), e outra ao tempo de execução da atividade em si. Conforme mencionado na Seção 2.2.1, estes tempos correspondem, respectivamente, aos tempos de fila e de serviço.

A distinção entre as abordagens é significativa, pois tem um impacto direto no modelo projetado. Caso os tempos das atividades tenham sido medidos com base na primeira abordagem (um único usuário gerando requisições), é importante que o modelo represente explicitamente os recursos que são disputados durante o processamento das requisições, representando de forma explícita a concorrência por estes recursos. Esta representação promoverá o surgimento dos tempos de fila, uma vez que as requisições têm que aguardar a disponibilidade dos recursos para que possam ser processadas. Por outro lado, no caso da adoção da segunda abordagem, os tempos de fila já estão embutidos

nos tempos medidos e associados às transições. Diante deste cenário, não é necessária a representação explícita dos recursos.

Dois aspectos devem ser considerados na hora de escolher a melhor alternativa dentre as duas abordagens discutidas acima. Por um lado há o aspecto relativo à precisão dos resultados. Na primeira abordagem o tempo alimentado no modelo é medido em uma condição diferente dos cenários representados no próprio modelo (que normalmente envolvem o processamento simultâneo de várias requisições). Neste caso, espera-se que os resultados obtidos possam ser menos precisos do que os resultados correspondentes obtidos de modelos alimentados com tempos medidos nas mesmas condições representadas nos experimentos. Por outro lado, a parametrização de modelos com tempos obtidos em cenários semelhantes aos representados pelo modelo limitam a sua utilidade e levam a uma constante necessidade de realização de experimentos de medição. Caso esta segunda abordagem seja escolhida, ferramentas de geração de carga podem ser utilizadas para criar e configurar usuários virtuais responsáveis pela geração de requisições para serviço permitindo uma simulação das condições de carga reais.

Outra consideração importante a ser feita, independentemente da abordagem escolhida para a realização das medições, diz respeito às necessidades de *warmup* das máquinas virtuais Java (JVMs). As JVMs atuais são, geralmente, baseadas na tecnologia *HotSpot* que é caracterizada pelo emprego de técnicas de compilação adaptativa, as quais permitem que partes de uma aplicação sejam compiladas sob demanda. Durante a execução de uma aplicação, o seu código é analisado e informações referentes a chamadas de métodos são coletadas. Quando um método sofre um determinado número de chamadas ele é compilado, de forma que as chamadas posteriores resultam na execução de código nativo. Desta forma, antes de se medir os tempos de uma atividade, um número grande de requisições devem ser realizadas (o valor exato depende da JVM utilizada) para forçar o processo de compilação. Após este processo, um outro conjunto de requisições é submetido para realizar efetivamente as medições com as atividades já em código nativo.

Ao final dos experimentos realizados para fins de parametrização, ferramentas automatizadas podem ser utilizadas para auxiliar o cálculo de estatísticas (e.g. valores médios, desvios, coeficientes de variação, quartis e percentis) a partir dos tempos medidos. Em particular, durante este trabalho foi desenvolvida uma ferramenta que suportam a leitura e processamento dos arquivos de *log* gerados.

4.8 Verificação e Validação

Conforme mencionado anteriormente, a representação de um serviço através de um modelo em Redes de Petri é uma tarefa complexa. Em função desta complexidade, é possível que os modelos projetados apresentem propriedades diferentes daquelas apresentadas pelo serviço modelado. Neste contexto, um dos objetivos desta atividade é definir propriedades desejáveis em um modelo e aplicar métodos que permitam a sua verificação.

Em geral, a escolha das propriedades a serem verificadas varia de acordo com as características e com a semântica associadas a cada modelo. Em outras

palavras, propriedades que são desejáveis para um modelo podem não ser desejáveis para outro. Um exemplo disto pode ser visto com a propriedade de *reversibilidade*, a qual estabelece que um modelo é sempre capaz de voltar ao seu estado inicial (ver Apêndice A.5). Em modelos que representam um serviço de transação, por exemplo, esta propriedade pode significar que após a realização de um *commit* ou *rollback*, este serviço retorna ao seu estado inicial, não retendo nenhuma informação de configuração. Para modelos representando este tipo de serviço, a reversibilidade pode ser considerada uma propriedade desejável. Por outro lado, o serviço de *pooling* de instâncias do JBoss não retorna ao seu estado original (*pool* vazio) depois do processamento da primeira requisição, uma vez que as instâncias criadas são retidas para o processamento das requisições subseqüentes. Desta forma, a propriedade de *reversibilidade* não é esperada em modelos que representem este serviço.

É importante observar que o número de propriedades selecionadas tem forte impacto nas técnicas utilizadas no processo de verificação. Embora, por um lado, a escolha de um número grande de propriedades aumente a confiança no modelo, por outro, dificulta a utilização de algoritmos eficientes de verificação. Um conjunto muito restrito também pode não ser adequado, uma vez que propriedades fundamentais podem deixar de ser verificadas.

De uma forma geral, há um conjunto mínimo de propriedades que são interessantes para modelos que representam sistemas concorrentes, dentre as quais destacam-se a limitação e ausência de *deadlocks* (ver Apêndice A.5). A verificação de outras propriedades deve ser considerada caso a caso.

A limitação de um modelo em Redes de Petri diz respeito à quantidade máxima de *tokens* que pode ser acumulada nos lugares representados. Conforme mencionado na Seção A.5, um modelo é dito *k-limitado* quando nenhum dos seus lugares pode acumular mais do que um número *k* finito de *tokens*. Uma conseqüência desta limitação é a garantia de que o espaço de estados associado ao modelo é finito. Esta limitação no número de estados garante a viabilidade "teórica" da construção do "grafo de alcançabilidade", a partir do qual diversas propriedades do modelo podem ser verificadas. Contudo, uma vez que o tamanho do grafo de alcançabilidade cresce exponencialmente com o número de lugares da rede, limitações em termos dos recursos computacionais disponíveis podem inviabilizar a sua construção.

Do ponto de vista semântico, a importância da propriedade de limitação em modelos representando serviço está relacionada à garantia de utilização de uma quantidade finita de recursos. Considerando que o número de recursos utilizados por um serviço é representado através de *tokens*, uma limitação no número máximo destes indica que o serviço modelado não consome recursos infinitos. Além disto, é importante ressaltar que a limitação é uma condição necessária para garantir a viabilidade da utilização de técnicas analíticas na avaliação de desempenho do modelo projetado.

Conforme mencionado na Seção A.5, a constatação da limitação de um modelo pode ser realizada através da utilização de técnicas estruturais, em particular, do cálculo dos invariantes de lugar. Se todos os lugares representados em um modelo forem cobertos por estes invariantes, a rede é dita

estruturalmente limitada. Neste caso, o modelo é limitado independentemente da marcação inicial representada.

A ausência de *deadlocks* em modelos em Redes de Petri garante a inexistência de marcações "mortas", ou seja, marcações nas quais nenhuma transição está habilitada. A constatação desta propriedade em redes que modelam serviços indica que os serviços modelados continuam processando requisições independentemente de estados que estes possam ter assumido durante o processamento de requisições anteriores.

Para verificar a ausência de *deadlocks* em um modelo pode-se construir seu grafo de alcançabilidade. Se este grafo apresentar algum estado no qual não existam transições habilitadas, o modelo apresenta *deadlock*. Caso contrário, o modelo é dito livre de *deadlocks* e a propriedade é verificada. Assim como a limitação, a ausência de *deadlocks* é condição necessária para a geração da cadeia de Markov isomórfica ao modelo, e conseqüentemente, para a utilização da técnica análise na obtenção dos resultados de desempenho.

É fundamental destacar que, mesmo os modelos que não apresentam as propriedades descritas acima podem ser utilizados em experimentos de avaliação de desempenho. Nestes modelos, embora técnicas analíticas não possam ser aplicadas, bons resultados podem ser obtidos através da realização de experimentos de simulação.

A validação de um modelo envolve (1) a realização de experimentos de simulação e/ou análise representando diferentes cenários, e (2) a comparação dos valores das métricas obtidas nestes experimentos com os valores correspondentes extraídos de experimentos de medição realizados a partir do sistema real.

Neste ponto, uma consideração importante diz respeito ao número de experimentos necessários à validação do modelo. Novamente, o teorema central do limite sugere a necessidade de uma única amostra de tamanho grande (pelo menos, 25 a 30 observações). Desta forma, considere um projeto que envolve uma métrica de *throughput* médio do servidor. Um experimento possível para a estimativa desta métrica consiste na submissão de um certo número de requisições a partir de um conjunto de clientes. Ao final do experimento, a métrica de *throughput* pode ser calculada a partir da razão entre o número total de requisições processadas pelo servidor e o tempo correspondente à duração do experimento. Neste caso, cada experimento fornece uma única observação para a métrica, sendo necessária a realização de um conjunto de 25 a 30 experimentos para que a média amostral possa ser considerada representativa.

Caso a métrica avaliada seja a utilização de CPU ao longo do tempo, um cenário um pouco diferente deve ser considerado. Este cenário envolve diversas variáveis aleatórias, cada uma correspondendo à utilização de CPU em um dado instante de tempo. Desta forma, a variável aleatória *CPU(1)* corresponderia ao valor da métrica no instante $t=1$, *CPU(2)* ao valor no instante $t=2$, e assim sucessivamente. Em cada experimento de medição realizado, é obtida uma observação de cada uma destas variáveis. Assim sendo, são necessários os mesmos 25 a 30 experimentos para que a utilização de CPU em cada instante possa ser determinada de forma consistente.

Por fim, a validação propriamente dita envolve uma comparação dos resultados obtidos via medição e via simulação do modelo. Caso estes resultados sejam considerados próximos, o modelo é considerado válido e pode ser utilizado em predições de desempenho. Caso contrário, o modelo deve passar por um processo de refinamento e ser novamente validado. O conceito de próximo varia de acordo com a finalidade do experimento, mas [Menascé et al. 2004] sugere que métricas associadas à utilização de recursos podem apresentar diferenças de 10%, *throughput* do sistema de 10% e tempo de resposta de 20%.

4.9 Realização de Experimentos

Nesta atividade, os modelos formais projetados são utilizados em experimentos de avaliação de desempenho. Considerando os objetivos do projeto e os fatores estabelecidos, diversos cenários são montados, os quais podem ser avaliados utilizando técnicas analíticas ou de simulação.

Em princípio, recomenda-se fortemente a utilização de técnicas analíticas para a solução do modelo, uma vez que os resultados assim obtidos são exatos. Contudo, o tamanho do modelo limita fortemente a capacidade de realização dos cálculos matemáticos necessários à obtenção das métricas estabelecidas. Conforme mencionado, a cadeia de Markov equivalente à rede de Petri estocástica cresce exponencialmente com o número de lugares representados, podendo inviabilizar a solução do modelo ainda que este seja limitado.

Nos cenários nos quais as técnicas analíticas se mostrem inviáveis, uma solução possível é a utilização de simulações. Neste contexto, um problema possível é o tempo necessário para que os valores das métricas se tornem estáveis. Este problema é ainda mais grave em cenários envolvendo eventos raros. Quanto maior a quantidade de métricas e a precisão desejada, maior é o tempo necessário para a correta finalização dos experimentos.

4.10 Considerações Finais

Este capítulo apresenta uma abordagem sistemática para a realização de avaliações de desempenho de servidores de aplicação desenvolvidos na tecnologia Java. A abordagem proposta é orientada a serviços, utilizando este conceito como unidade de particionamento de projetos. Em particular, a abordagem propõe a construção de modelos formais em redes de Petri estocásticas e a utilização das técnicas de modelagem analítica e/ou simulação para a derivação das métricas de desempenho.

CAPÍTULO

5

Avaliação de Desempenho do JBoss

"Eu ouço e eu esqueço.Eu vejo e eu lembro.
Eu faço e eu entendo."

Provérbio Chinês.

Neste capítulo são apresentados os modelos de desempenho para o servidor de aplicação JBoss projetados com base na abordagem proposta. Inicialmente, o capítulo apresenta a definição do sistema a ser avaliado e dos serviços e métricas considerados nesta avaliação. Em seguida, os parâmetros que possuem efeito significativo sobre o desempenho são identificados e aqueles que são variados ao longo dos experimentos (fatores) são definidos. Por fim, diferentes modelos de desempenho são propostos, parametrizados e validados, sendo as suas principais características e limitações apresentadas em detalhes.

5.1 Definição do Sistema-Alvo

O servidor JBoss foi escolhido como alvo primário da dissertação considerando-se o fato de ser de código aberto e de possuir uma fatia representativa do mercado (segundo dados do Gartner Group [Gartner 2005]). Em particular, o presente trabalho avalia o suporte fornecido por este servidor aos componentes EJB através da implementação de um *container* compatível com a correspondente especificação. Dados o tamanho e a complexidade característicos dos servidores de aplicação, uma análise completa deste produto está além do escopo do presente trabalho.

5.2 Seleção de Serviços e Métricas

Os serviços fornecidos pelos servidores de aplicação, como segurança, transação e persistência, têm características bastante diferentes, e, conseqüentemente, diferentes impactos em termos de desempenho. Em geral, a utilização destes serviços é, pelo menos em parte, opcional, cabendo ao arquiteto de software decidir, para cada aplicação desenvolvida, quais serviços do servidor de aplicação deverão ser utilizados e quais deverão ser implementados diretamente na aplicação. Contudo, o gerenciamento do ciclo de vida de componentes EJB é de responsabilidade exclusiva dos servidores de aplicação, de forma que os serviços relativos à esta atividade são utilizados por todas as aplicações baseadas nesta tecnologia. O serviço de *pooling* de instâncias é um destes serviços.

Além de ser um serviço amplamente utilizado, o serviço de *pooling* representa um grande impacto sob o ponto de vista do desempenho, uma vez que proporciona uma redução nos requisitos de memória e de tempo necessários ao processamento de requisições individuais. Estas características tornam o serviço de *pooling* um candidato natural para projetos de avaliação de desempenho focados na tecnologia EJB, sendo este serviço, portanto, selecionado como alvo nesse projeto.

Para estudar de forma ampla o serviço de *pooling* é interessante que o conjunto de métricas selecionadas possibilite a análise tanto de aspectos internos do servidor, de interesse dos administradores do sistema, quanto de aspectos externos, relacionados com o ponto de vista dos clientes.

Do ponto de vista dos administradores de sistemas, uma variável importante a ser gerenciada é o tamanho do *pool* de instâncias para os diversos componentes instalados. Um bom dimensionamento desta variável minimiza a quantidade de instâncias criadas, reduzindo as operações de coleta de lixo e racionalizando a utilização dos recursos do sistema. Para uma configuração adequada dessa variável é necessário que se realizem experimentos estabelecendo um tamanho esperado para o *pool* relativo a um dado componente (que varia de acordo com a previsão de carga para o componente em questão) e se acompanhe o número de instâncias efetivamente criadas. A necessidade desses experimentos típicos de planejamento de capacidade torna o número de instâncias criadas uma métrica de utilização bastante relevante para os administradores de servidores de aplicação.

Outras métricas de utilização importantes dizem respeito ao percentual de utilização dos recursos da máquina. Um recurso de importância vital para o desempenho de qualquer sistema computacional é a(s) CPU(s). Desta forma, uma métrica considerada relevante é o percentual de utilização da CPU, a qual permite avaliar se este recurso está representando um gargalo para o desempenho global.

Com relação ao ponto de vista dos clientes, uma métrica de produtividade fundamental é o *throughput*, que funciona como um indicador da capacidade de resposta do servidor, facilitando o entendimento do efeito do serviço de *pooling* sobre a capacidade de processamento do sistema.

5.3 Identificação de Parâmetros

Diversos parâmetros têm efeito sob o desempenho do sistema. Dentre eles destacam-se aspectos relacionados a:

- Hardware e sistema operacional;
- Máquina virtual;
- Configuração do servidor de aplicação;
- Características da aplicação de testes utilizada;
- Parâmetros de carga.

5.3.1 Hardware e Sistema Operacional (SO)

Os aspectos de hardware têm impacto significante sobre o desempenho dos sistemas. Dentre eles destacam-se CPU, memória RAM e disco rígido. O sistema operacional também tem grande relevância para o desempenho global pois, além de gerenciar o hardware, ele é responsável pelo gerenciamento de processos e threads, de memória virtual e de importantes protocolos de rede.

Do ponto de vista do projeto de avaliação de desempenho considerado nesta dissertação, os parâmetros relativos ao hardware e aos sistemas operacionais permaneceram constantes. Todos os cenários montados nos experimentos foram projetados com base na utilização de duas estações, uma utilizada como cliente e outra como servidora, interligadas através de uma rede Ethernet 10/100 completamente isolada. As principais características de hardware e SO de ambas as estações são resumidas na Tabela 5.1.

Tabela 5.1. Parâmetros das Máquinas Cliente e Servidor.

Estação	CPU	Mem. RAM	Mem. Virtual	Sistema Operacional
Cliente	Athlon 2000+	768 MB		Windows 2000 Professional Edition
Servidor	Pentium M 1.60 GHz	750 MB	756MB	Windows XP Home Edition

É importante mencionar que durante a realização dos experimentos de medição as aplicações e serviços desnecessários rodando nas estações cliente e servidora foram parados de forma a mimizar a interferência externa.

5.3.2 Máquina Virtual Java (JVM)

No projeto de avaliação de desempenho relatado nesta dissertação, os experimentos de medição realizados utilizam duas máquinas virtuais Java, uma na estação cliente e outra na servidora. Em ambos os casos adotou-se a implementação da JVM fornecida pela Sun em sua versão 5.0 (versão de jdk correspondente: 1.5.0).

A JVM executada na estação cliente é utilizada somente para executar uma ferramenta de geração de carga (JMeter 2.0.3). Utilizando-se esta ferramenta, os parâmetros relativos à carga podem ser configurados (ver Seção

5.3.5). Uma vez que a única função desta ferramenta é gerar requisições para o sistema numa determinada taxa, não são necessárias otimizações nas configurações padrões da JVM.

Por outro lado, como a JVM da estação servidora é utilizada na execução do JBoss, que é o alvo primário da avaliação de desempenho, é importante que a sua configuração seja otimizada, de forma a minimizar a interferência de fenômenos relacionados à máquina virtual (como, por exemplo, o mecanismo de coleta de lixo) nos experimentos de medição.

Dentre os parâmetros de configuração oferecidos pela implementação da JVM, o tamanho da *heap* e o compilador de tempo de execução têm um impacto significativo no desempenho.O tamanho da *heap* afeta diretamente a frequência das coletas de lixo e o tempo de cada uma. Uma *heap* pequena tende a ser preenchida rapidamente, aumentando a frequência de execução das coletas de lixo. Por outro lado, uma *heap* grande tende a demorar mais para encher, mas sua coleta pode ser mais lenta, levando a pausas maiores na execução das aplicações. A máquina virtual varia, ao longo de sua execução, o tamanho da *heap* buscando maximizar o desempenho do sistema. Os tamanhos máximo e mínimo podem ser manualmente configurados durante a inicialização da JVM.

A implementação da JVM é baseada na tecnologia *HotSpot*, cuja principal característica é a compilação adaptativa. O compilador utilizado pela máquina virtual em tempo de execução (*runtime compiler*) pode ser selecionado durante o processo de inicialização. As duas opções disponíveis na versão 5 de Java são conhecidas como compilador cliente e servidor. O compilador cliente é o *default* para máquinas que possuem um único processador. Ele é otimizado para trabalhar com um baixo requisito inicial de memória (*footprint*) e para minimizar o tempo de inicialização da máquina virtual. O compilador servidor procura minimizar o tempo gasto na compilação de código através da criação de mais *threads*, contudo implica em um maior requisito em termos de processador e memória. Este compilador é considerado ideal para máquinas que possuam dois ou mais processadores e pelo menos 2GB de memória. Os dois compiladores também diferem em termos da escolha do número de requisições que um método deve receber para ser considerado para compilação. Para o compilador cliente, a partir de 1500 chamadas o método deve ser compilado. Para o compilador servidor este número de chamadas é de 10000. Estes valores devem ser considerados no planejamento dos experimentos uma vez que indicam necessidades diferentes de *warmup*.

Na otimização da JVM servidora, as quatro diferentes configurações baseadas em combinações de tamanhos de *heap* e de compilador de tempo de execução foram consideradas. Para testar estas configurações projetou-se um experimento que consistia, basicamente, em inicializar repetidas vezes o JBoss e anotar o tempo necessário à conclusão desta inicialização. A Tabela 5.2 apresenta os resultados obtidos para as quatro configurações de máquina virtual testadas.

Tabela 5.2. Configurações da JVM do JBoss

Runtime Compiler	Tamanho Mínimo da *Heap* (MB)	Tamanho Máximo da *Heap* (MB)	Média (s)	Desvio Padrão
cliente	2	64	11,2231	0,149701
cliente	256	512	9,9663	0,102620
servidor	2	64	22,8948	1,080010
servidor	256	512	22,8499	1,819467

Diante dos resultados apresentados, foi escolhida a configuração baseada na utilização do compilador cliente com a *heap* variando entre 256MB e 512MB.

5.3.3 Configuração do Servidor de Aplicação

Todos os experimentos realizados foram baseados na configuração *default* do JBoss v3.2.7 [JBoss 2004], a qual inclui todos os serviços previstos pela plataforma J2EE.

5.3.4 Características da Aplicação de Testes Utilizada

A aplicação teste utilizada na realização dos experimentos de medição é composta por um *stateless session bean* e um conjunto de componentes *web*. Clientes acessam esta aplicação enviando requisições HTTP, que são processadas pelos componentes *web* e encaminhas por estes a uma instância do *bean*. Esta instância retorna uma mensagem constante que é, então, embutida em uma página HTML retornada para os clientes.

Durante a realização dos experimentos, a aplicação foi configurada para utilizar os serviços previstos pelo JBoss em sua configuração *default*, não sendo realizada nenhuma customização. No caso do serviço de *pooling* de instâncias, esta configuração determina o modo de operação não-estrito, com o *pool* possuindo um tamanho mínimo de 0 e um tamanho máximo de 100 instâncias.

5.3.5 Parâmetros de Carga

Do ponto de vista de carga, os dois parâmetros que possuem um impacto direto no desempenho do sistema são o número de clientes simultâneos e a taxa média de requisições realizadas por estes.

5.4 Seleção de Fatores

Considerando-se a diversidade de parâmetros identificados, várias combinações possíveis de fatores poderiam ser selecionadas. Contudo, a escolha de diversos fatores para os experimentos pode dificultar a percepção clara do efeito de cada um sobre o desempenho global do sistema. Visando tornar os resultados mais expressivos, optou-se pela utilização de configurações padrões em termos da máquina virtual (exceto o tamanho da *heap*), do servidor JBoss e dos serviços utilizados pela aplicação. Os parâmetros referentes a *hardware* e sistema operacional também permaneceram constantes.

Os principais fatores selecionados correspondem aos parâmetros relativos à caracterização da carga. Esta escolha visa permitir a realização de

experimentos que simulem diferentes demandas fornecendo informações sobre o comportamento do sistema diante desses cenários e possibilitando um dimensionamento de sua real capacidade. Em particular, a caracterização da carga nos cenários considerados ocorre através da configuração do número de clientes simultâneos e da taxa média de requisições por cliente.

Do ponto de vista do serviço de *pooling* de instâncias, um fator de interesse é o tamanho do *pool*. Experimentos de planejamento de capacidade envolvem uma variação deste fator em busca de uma configuração ótima para o sistema.

5.5 Projeto dos Modelos

Esta seção apresenta modelos DSPN representando diferentes visões do serviço de *pooling* de instâncias implementado no JBoss.

O primeiro modelo apresentado é um modelo abstrato, que tem como objetivos: mostrar uma visão global da arquitetura embutida no JBoss para suporte a componentes EJB; e contextualizar os demais modelos desenvolvidos. Neste modelo estão representados tanto os componentes que atuam na estação cliente quanto aqueles que atuam dentro do servidor. É importante ressaltar que este modelo tem um caráter puramente funcional, não sendo, assim, válido para a realização de experimentos de avaliação de desempenho.

O segundo modelo desenvolvido, chamado modelo servidor, representa uma visão detalhada do serviço de *pooling* de instâncias, apresentando explicitamente apenas os componentes relacionados à sua implementação. Neste modelo, as atividades temporizadas são representadas através de transições exponenciais cujo parâmetro é igual ao tempo médio obtido através de experimentos de medição realizados em cenários reais.

O terceiro modelo, chamado modelo servidor refinado, é derivado do segundo a partir do processo de refinamento, que tem como objetivo tornar mais precisos os resultados obtidos através de simulações. Para tanto, são inseridas no modelo informações sobre a variabilidade do tempo associado às atividades temporizadas.

No quarto modelo apresentado, chamado modelo cliente/servidor, estão presentes tanto os componentes da porção cliente quanto os da porção servidora. Neste modelo, a informação de tempo representada foi obtida através de experimentos de medição realizados em um ambiente controlado no qual um único cliente submetia sucessivas requisições.

Por fim, o último modelo representa uma visão de alto nível do serviço de *pooling*. Neste modelo, cada atividade representada é uma composição de atividades representadas isoladamente nos modelos anteriores. Sendo um modelo de alto nível, é possível a utilização de técnicas analíticas no cálculo das métricas de desempenho.

5.5.1 Modelo Cliente/Servidor Funcional

Conforme mencionado na Seção 2.1.2, uma aplicação cliente interage com um componente EJB através de um *proxy* que o representa. Este *proxy* possui os mesmos métodos que o próprio componente e implementa a mesma

interface, de forma que a sua existência é transparente para as aplicações. A cada *proxy* está associado um *handler*, para o qual todas as requisições que esse recebe são encaminhadas. Esse *handler* encapsula as informações da requisição (tais como, os identificadores do componente e do método acionado e os parâmetros correspondentes) em um objeto chamado *invocation* e o encaminha através de uma cadeia de interceptadores. Cada interceptador processa o objeto *invocation* e adiciona a ele informações que julgue relevantes. Ao final da cadeia de interceptadores, a requisição, representada pelo objeto *invocation*, é encaminhada para um objeto conhecido como *invoker proxy* cuja função é transmitir essa requisição através da rede.

Do lado servidor, um objeto chamado *invoker* é encarregado de receber as requisições e transmití-las ao EJB *container* responsável pelo gerenciamento do componente correspondente. O *container* possui uma cadeia de interceptadores que é acionada por ele para realizar os serviços configurados para o componente em questão. Um desses interceptadores é o interceptador de instância, o qual é responsável pela obtenção da instância do componente usada no processamento da requisição. A Figura 5.1 ilustra esta arquitetura utilizando um modelo em redes de Petri.

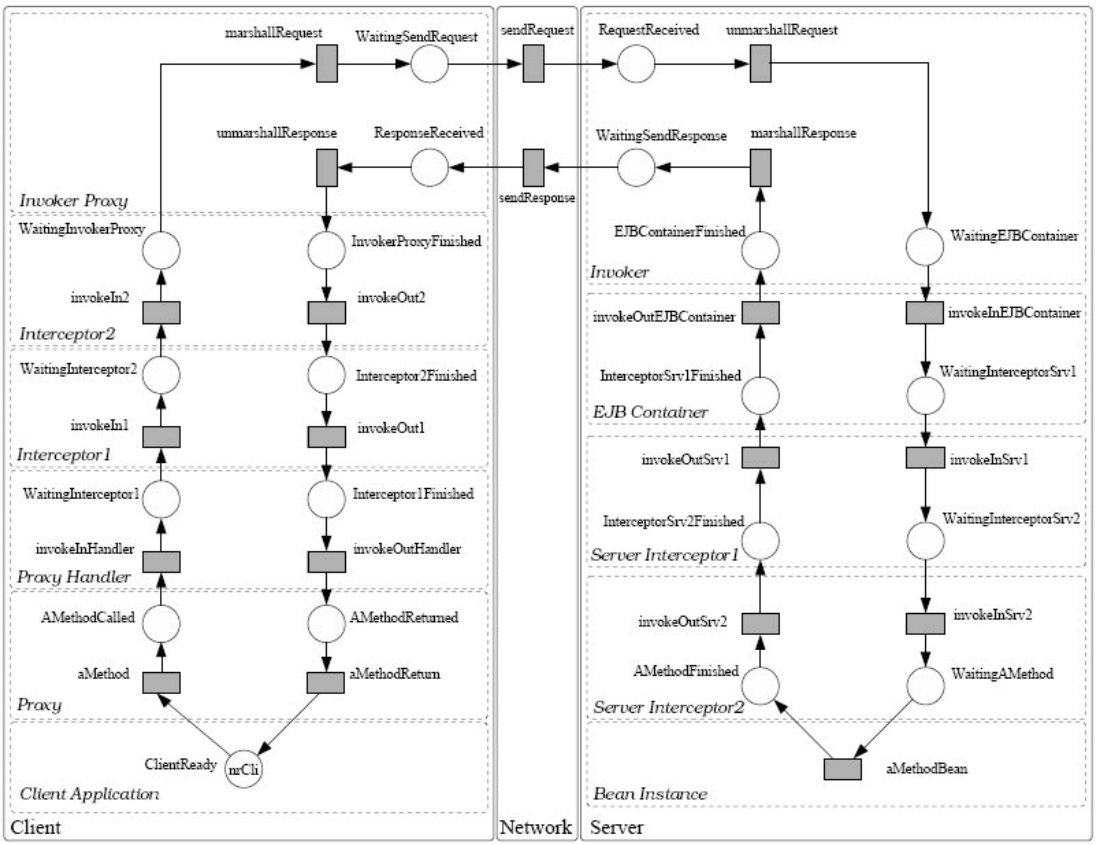


Figura 5.1. Visão Geral Modelo Cliente/Servidor Funcional

No modelo cada componente da arquitetura é identificado e representado por uma subrede. O componente *Client Application* representa os clientes que interagem com o *bean*. A subrede correspondente possui um único lugar, no qual cada *token* depositado representa um cliente pronto para realizar uma requisição.

As requisições destinadas a um componente EJB são recebidas pelo *proxy* que o representa através do acionamento de um de seus métodos. No modelo, um *proxy* é representado através do componente *Proxy* com sua respectiva subrede. Este componente representa todos os métodos de um *proxy* através de uma única transição chamada genericamente de *aMethod*. O acionamento de um método qualquer do *bean* corresponde, no modelo, ao disparo desta transição. Ao ser disparada, a transição gera um *token* no lugar *AMethodCalled* sinalizando para o *Proxy* a requisição recebida. Este *token* habilita a transição *invokeInHandler* representando o acinamento do *handler* (modelado pelo componente *Proxy Handler*). Devido à natureza síncrona da requisição, após este acionamento o *Proxy* entra em um estado de espera, passando a aguardar um retorno do *Proxy Handler*. Este retorno é sinalizado pela inserção de um *token* no lugar *AMethodReturned*. Após recebê-lo, o *Proxy* encaminha o retorno para a aplicação cliente (*Client Application*) através do disparo da transição *aMethodReturn*.

Ao receber uma requisição o *Proxy Handler* monta um objeto *invocation* para representá-la, sendo este processamento modelado através da transição *invokeInHandler*. Quando esta transição dispara, sinalizando a conclusão da montagem, o *Proxy Handler* aciona o primeiro interceptador de sua cadeia. Este acionamento é representado no modelo pela habilitação da transição *invokeIn1*, decorrente da geração de um *token* no lugar *WaitingInterceptor1*. Dado o sincronismo da requição, o *Proxy Handler* passa a esperar pelo retorno do primeiro interceptador, o qual é sinalizado pela inclusão de *token* no lugar *Interceptor1Finished*. Este retorno é processado e encaminhado ao *Proxy* através do disparo da transição *invokeOutHandler*.

O modelo apresentado possui apenas 2 interceptadores no cliente. Cada um deles está representado de maneira genérica por um componente *InterceptorN* (onde *N* corresponde ao número do interceptador). Ao ser acionado, um interceptador processa a requisição e a encaminha para o próximo na cadeia. É importante ressaltar que neste momento a sua função ainda não está concluída, uma vez que a resposta do *bean* também flui através da cadeia de interceptadores. A natureza do processamento realizado no envio e no retorno depende do interceptador acionado. Desta forma, o processamento realizado por cada interceptador está representado separadamente por duas transições. A primeira, chamada *invokeInN*, representa o processamento realizado no envio da requisição. A segunda, chamada *invokeOutN*, representa o processamento realizado no recebimento da resposta.

Após receber e processar uma requisição, o último interceptador da cadeia encaminha-a para o *Invoker Proxy*, gerando um *token* no lugar *WaitingInvokerProxy*, e passa a esperar pelo seu retorno, o qual é sinalizado pela chegada de *token* no lugar *InvokerProxyFinished*. O retorno recebido é, então, enviado no sentido oposto pela cadeia até chegar ao primeiro interceptador.

Ao receber uma requisição, o *Invoker Proxy* realiza a operação de *marshalling* (representada no modelo pela transição *marshallingRequest*) preparando as informações para transmissão através da rede. A solicitação de transmissão é representada pela geração de um *token* no lugar *WaitingSendRequest*. Após o envio da requisição, o *Invoker Proxy* fica em

estado de espera aguardando o recebimento da resposta, que é sinalizado pela geração de um *token* no lugar *ResponseReceived*. Ao receber a resposta, o *Invoker Proxy* realiza o *unmarshalling* do retorno, sendo esta operação representada no modelo pela transição *unmarshallingResponse*. Após esta operação, o retorno é repassado ao último interceptador da cadeia.

A rede que interliga clientes e servidores é representada pelo componente *Network*, que compreende apenas duas transições, *sendRequest* e *sendResponse*, representando, respectivamente, a transmissão da requisição e da resposta.

Do lado servidor, o componente encarregado de receber as requisições é o *Invoker*. Cada requisição recebida é representada por um *token* gerado no lugar *RequestReceived*. Ao receber uma requisição, o *Invoker* realiza o processo de *unmarshalling* representado pela transição *unmarshallRequest*. Em seguida, o EJB *container* que gerencia o componente associado à requisição recebida é localizado e acionado. Este acionamento é representado pela geração de um *token* no lugar *WaitingEJBContainer*. O *Invoker* passa, então, a esperar pelo retorno, que é sinalizado pela inserção de *token* no lugar *EJBContainerFinished*. O retorno recebido passa por um processo de *marshalling*, representado pela transição *marshallResponse*, ficando pronto para ser encaminhado para o cliente. O encaminhamento é requisitado à rede através da geração de um *token* no lugar *WaitingSendResponse*.

Da mesma maneira que um *Proxy*, um *EJB Container* também está associado à uma cadeia de interceptadores que são responsáveis pela implementação dos serviços fornecidos aos componentes EJB. Ao receber uma requisição, que ocorre através do disparo da transição *invokeInEJBContainer*, este componente envia-a para o primeiro interceptador da cadeia, o qual é acionado quando um *token* é depositado no lugar *WaitingInterceptorSrv1*. Então, o *container* passa a esperar o retorno desse interceptador que é sinalizado através da inserção de *token* no lugar *InterceptorSrv1Finished*. Quando este retorno é recebido, o *container* processa-o e encaminha para o cliente através do *Invoker*.

No modelo desenvolvido, apenas dois interceptadores estão representados no servidor, através dos componentes *Server InterceptorN*, onde *N* representa o número do interceptador na cadeia. Cada interceptador, quando acionado, processa a requisição e encaminha-a ao interceptador seguinte. Da mesma forma que acontece com os interceptadores clientes, os interceptadores do servidor também realizam processamento antes de transmitir a requisição para o *bean* e após receberem o retorno deste, representados, respectivamente, pelas transições *invokeInSrvN* e *invokeOutSrvN*.

A instância do componente EJB, representada no modelo como *Bean Instance*, é acionada pelo último interceptador da cadeia. Todos os métodos do componente são representados esquematicamente pela mesma transição, *aMethodBean*. O retorno do componente é devolvido ao último interceptador quando um *token* é gerado no lugar *AMethodFinished* e é encaminhado de volta ao cliente.

Este primeiro modelo, embora não possa ser utilizado para fins de avaliação de desempenho, teve um papel fundamental na consolidação do entendimento detalhado da arquitetura do JBoss. Uma vez que este modelo não

é orientado a desempenho, os procedimentos de obtenção de dados estatísticos, validação, calibragem e refinamento não se aplicam.

5.5.2 Modelo Servidor

O primeiro modelo de desempenho desenvolvido para o serviço de *pooling* de instâncias representa apenas os componentes do lado servidor que estão diretamente relacionados à este serviço. A identificação destes componentes, de suas fronteiras, das atividades relevantes realizadas por cada um e da interação entre eles, foi realizada com base na documentação do servidor JBoss, no estudo do código-fonte, e, principalmente, na utilização das ferramentas de *profiling* desenvolvidas pelo projeto Eclipse TPTP (*Test & Performance Tools Platform*). Neste primeiro modelo de desempenho, a única métrica avaliada é o número de instâncias que fornece subsídio para um dimensionamento adequado do *pool*.

O modelo proposto, apresentado na Figura 5.2, compreende quatro componentes: *Instance Interceptor*, *Instance Pool*, *Previous Interceptor* e *Next Interceptor*. Este modelo assume que as durações das atividades temporizadas têm distribuição exponencial. Logo, cada uma é representada por uma transição exponencial com parâmetro igual à média dos tempos medidos para a atividade correspondente. Os procedimentos adotados para a medição dos tempos das atividades são detalhados na Seção 5.5.2.1, enquanto que os tempos médios calculados são apresentadas na Tabela 5.3.

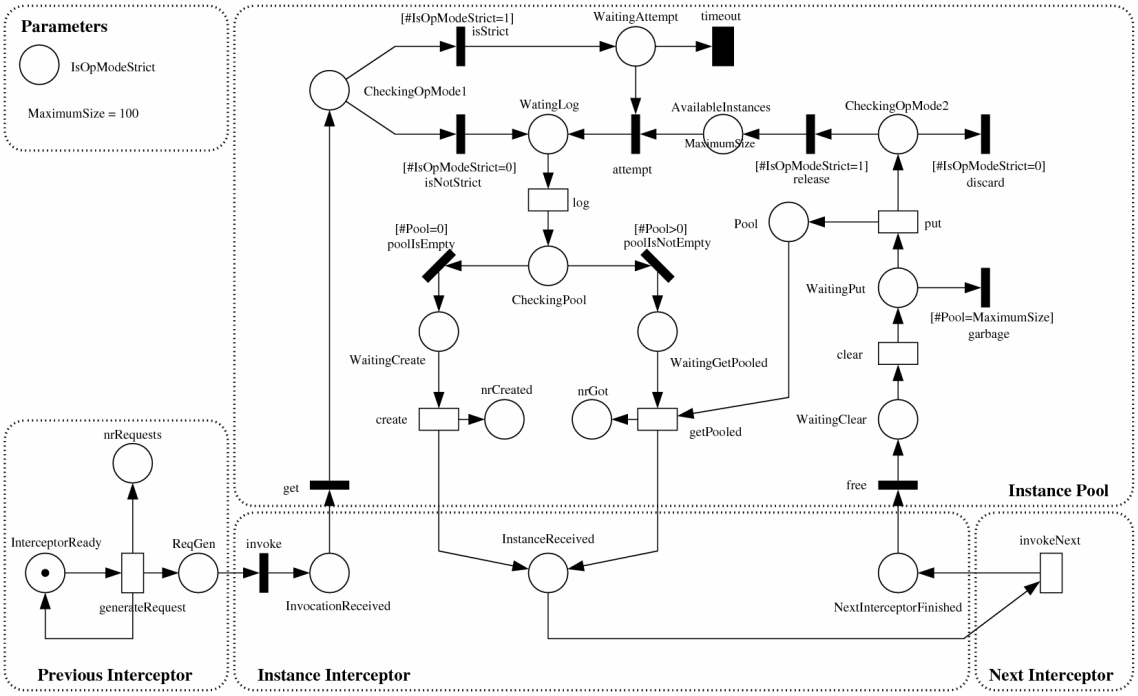


Figura 5.2. Visão Geral do Modelo Servidor.

Além das transições exponenciais, o modelo contém transições imediatas e transições determinísticas. As imediatas representam atividades que não consomem tempos representativos, mas são incluídas para que se possa obter um modelo semanticamente completo. A Tabela 5.4 apresenta as informações sobre peso, prioridade e função de habilitação relativos à estas transições. Por fim, a única transição deterministica presente no modelo é utilizada para

representar um *timeout* no processamento de uma requisição. No ambiente real este *timeout* pode ser configurado quando uma aplicação é instalada e seu valor *default* é bastante elevado (equivalente aproximadamente a 292.471.208 de anos), visando garantir que todas as requisições sejam eventualmente atendidas. No restante desta seção, cada componente é apresentado detalhadamente.

Tabela 5.3. Transições Exponenciais do Modelo Servidor: Tempos Médios Medidos

Transição	Tempo Médio de Execução (μs)
clear	45,1564
create	187,1151
generateRequest	1.021,7390
getPooled	2,0182
invokeNext	16,2403
log	41,3749
put	3,1402

O componente *Previous Interceptor* representa o interceptador imediatamente anterior ao *Instance Interceptor* na cadeia de interceptadores do *container* EJB. Este componente atua como um gerador de carga, uma vez que transmite as requisições geradas pelos clientes, os quais não são explicitamente modelados. Um *token* no lugar *InterceptorReady* significa que este interceptador está pronto para encaminhar uma nova requisição cujo recebimento é modelado através do disparo da transição *generateRequest*. Como o tempo médio entre disparos desta transição é exponencial, o modelo representa um processo de chegada de requisições seguindo a distribuição de *Poisson*. A cada disparo da transição *generateRequest*, um *token* é gerado no lugar *nrRequests* e outro no lugar *ReqGen*, os quais modelam, respectivamente, um contador usado para computar o número de requisições recebidas e a indicação de que o *InstanceInterceptor* possui uma nova requisição para processar. Um detalhe importante é que todas as transições exponencias representadas no modelo servidor possuem a semântica *single-server*, de forma que os clientes simultâneos são representados diretamente através da taxa associada à transição *generateRequest* e não através de *tokens* representados diretamente no modelo.

O componente *Instance Interceptor* modela o interceptador responsável pelo acionamento do serviço de *pooling* de instâncias, o qual é responsável por obter a instância do componente EJB utilizada no processamento da requisição. Este componente recebe uma requisição através do disparo de sua transição *invoke*. Ao ser disparada, esta transição gera um *token* no lugar *InvocationReceived* sinalizando que uma instância deve ser requisitada ao *Instance Pool*.

O componente *Instance Pool* representa o *pool* propriamente dito e atua como um repositório de instâncias de um componente EJB. Estas instâncias podem ser requisitadas através do disparo da transição *get*. Quando esta transição é disparada, um *token* é gerado no lugar *CheckingOpMode1*. Neste momento, uma escolha deve ser feita com base no modo de operação configurado para o *pool* (representado através do lugar *IsOpModeStrict*). Esta

escolha é modelada pelas transições *isNotStrict* e *isStrict*, as quais são mutuamente exclusivas, conforme pode ser verificado pelas suas funções de habilitação: *#IsOpModeStrict=0* e *#IsOpModeStrict=1*, respectivamente.

Tabela 5.4. Peso, Prioridade e Função de Habilitação.

Transição	Peso	Prioridade	Função de Habilitação
attempt	1	3	-
discard	1	1	*#IsOpModeStrict=0*
free	1	4	-
garbage	1	5	*#Pool=MaximumSize*
get	1	3	-
invoke	1	2	-
isNotStrict1	1	1	*#IsOpModeStrict=0*
isStrict	1	1	*#IsOpModeStrict=1*
poolIsEmpty	1	8	*#Pool=0*
poolIsNotEmpty	1	8	*#Pool>0*
release	1	4	*#IsOpModeStrict=1*

Caso o *pool* esteja configurado no "modo não-estrito", a função de habilitação *#IsOpModeStrict=0* é verdadeira e o número de instâncias do componente não está sendo controlado (este é o modo de operação *default*). Neste modo de operação, a transição *isNotStrict* dispara, sendo gerado um *token* no lugar *WaitingLog*. Este *token* habilita imediatamente a transição *log* que representa a gravação de um registro da requisição em um arquivo de *log* do sistema.

Por outro lado, com o *pool* operando no "modo estrito", a condição *#IsOpModeStrict=1* é verdadeira e o número de requisições simultaneamente processadas, e, conseqüentemente, de instâncias criadas, é limitado pela transição *attempt*. Neste cenário, um *token* depositado no lugar *CheckingOpMode1* habilita a transição *isStrict* que dispara gerando um *token* no lugar *WaitingAttempt*. Quando isto acontece, a transição *attempt* testa se número de requisições simultâneas em processamento atingiu seu limite máximo, verificando o lugar *AvailableInstances*. Caso haja algum *token* neste lugar, o limite de requisições simultaneamente processadas ainda não foi alcançado e a requisição corrente poderá ser imediatamente atendida. Desta forma, a transição *attempt* dispara e gera um *token* no lugar *WaitingLog*, o qual habilita a transição *log*. Caso contrário, a requisição corrente, representada pelo *token* depositado no lugar *WaitingAttempt*, tem que esperar. Se o processamento de uma requisição em atendimento for concluído antes do tempo configurado para *timeout*, a transição *attempt* dispara permitindo que a requisição em espera possa seguir o seu curso. Caso contrário, a transição *timeout* dispara e o processamento da requisição em espera é considerado encerrado.

Após logar as requisições, o *Instance Pool* checa o lugar *Pool*, que armazena as instâncias disponíveis representadas através de *tokens*. Esta

checagem é indicada pelo lugar *CheckingPool* e pelas transições *poolIsEmpty* e *poolIsNotEmpty* com suas respectivas funções de habilitação.

Se o *Pool* contém alguma instância disponível (i.e. *#Pool>0*), a transição *poolIsNotEmpty* dispara e gera um *token* no lugar *WaitingGetPooled*. Este *token* habilita a transição *getPooled*, a qual representa a atividade de obtenção de uma instância já disponível. Cada instância obtida é registrada no lugar *nrGot*. Por outro lado, se *Pool* não possuir nenhuma instância correntemente disponível (i.e. *#Pool=0*), a transição *poolIsEmpty* dispara e um *token* é movido para o lugar *WaitingCreate*, significando que uma nova instância deve ser criada. Neste modelo, a criação de instâncias é realizada pela transição *create*. As instâncias criadas são registradas no lugar *nrCreated*.Por fim, sendo criada ou reutilizada, uma instância é retornada para o *Instance Interceptor* na forma de um *token* depositado no lugar *InstanceReceived*.

Após receber uma instância, o *Instance Interceptor* a associa à requisição corrente e invoca o próximo interceptador na cadeia do *container*, representado no modelo pelo componente *Next Interceptor*. Este componente possui uma única transição, *invokeNext*, que representa, conjuntamente, as atividades realizadas pelos interceptadores seguintes na cadeia e pelo próprio componente em si. Quando esta transição dispara, o componente *Next Interceptor* completa sua execução e retorna. Este retorno é sinalizado através da criação de um *token* no lugar *NextInterceptorFinished*. Em seguida, o componente *Instance Interceptor* aciona a transição *free*, solicitando ao componente *Instance Pool* a liberação da instância utilizada.

Ao receber uma requisição para liberar uma instância, o *Instance Pool* remove da instância as informações relacionadas à requisição anterior, sendo esta atividade representada no modelo através da transição *clear*. Com a instância "limpa", um *token* é armazenado no lugar *WaitingPut* e a quantidade de instâncias já armazenadas no *Pool* é checada. Se o *Pool* estiver cheio (*#Pool=MaximumSize*), a transição *garbage* dispara indicando que a instância utilizada está agora disponível para o mecanismo de coleta automática de lixo. Caso contrário, a transição *garbage* é desabilitada e a transição *put* irá disparar. Quando isto ocorrer, um *token* será gerado no lugar *Pool*, indicando que a instância já pode ser reutilizada. A transição *put* também deposita um *token* no lugar *CheckingOpMode2*, fazendo com que o modo de operação seja novamente checado. Se o *Instance Pool* está operando no "modo estrito", a transição *release* dispara, gerando um *token* no lugar *AvailableInstances* que indica que uma requisição foi concluída. Caso contrário, a transição *discard* dispara indicando que o número de requisições simultâneas não está sendo controlado.

5.5.2.1 Parametrização

Para realizar a parametrização, o código do servidor JBoss foi manualmente instrumentado, de forma a permitir a gravação dos tempos das atividades representadas no modelo por transições exponenciais. O código inserido no processo de instrumentação pode ser configurado declarativamente, habilitando ou desabilitando a gravação do tempo de cada atividade individual. Para obter o tempo de uma atividade, o código anota os instantes nos quais a atividade se inicia e se encerra, gravando a diferença entre eles em um arquivo de *log* correspondente.

Além dos tempos associados às atividades, o código inserido na instrumentação pode ser configurado para registrar o momento em que cada instância de um componente é criada, permitindo assim, o acompanhamento da métrica relativa à criação destas instâncias.

Para realizar a parametrização do modelo, seis experimentos foram executados em uma rede isolada, composta unicamente por uma máquina cliente e uma servidora, eliminando qualquer interferência causada por tráfego externo. Visando observar as necessidades de *warmup* da máquina virtual utilizada para a execução do JBoss, cada experimento de medição é precedido pela execução de um experimento de *warmup*, consistindo em único cliente realizando uma sequência de 10.000 requisições. É importante ressaltar que, embora para a JVM utilizada (*hotspot client*) só fossem necessárias 1.500 requisições para que o processo de compilação das porções críticas do servidor ocorresse, os *scripts* de teste foram configurados para 10.000 de forma a contemplar uma eventual substituição da JVM. Ao final de cada experimento, o servidor de aplicação é reiniciado, evitando que a realização de um experimento influencie os resultados obtidos à partir dos seguintes.

A carga utilizada em cada um destes experimentos foi a mesma utilizada nos experimentos de simulação realizados com o modelo, correspondendo a 10 clientes, cada um tentando realizar 100 requisições por segundo durante um intervalo de 100 segundos. Desta forma, cada experimento corresponde a uma carga teórica de 100.000 requisições submetidas ao servidor, totalizando, ao final dos seis experimentos, 600.000 requisições.

Contudo, é importante considerar que as requisições são síncronas, de forma que cada cliente só pode gerar uma nova requisição após obter uma resposta para a anterior. Desta forma, a taxa real de requisições geradas pode ser diferente daquela que foi configurada para os clientes. De fato, devido à concorrência por recursos e às operações de I/O necessárias ao registro dos tempos das atividades, apenas 401.740 requisições foram geradas. À exceção das atividades de criação e obtenção de instâncias, modeladas pelas transições *create* e *getPooled*, todas as demais atividades foram executadas exatamente este número de vezes. Ao longo dos experimentos observou-se a criação de 60 instâncias. A diferença entre o total de requisições e o número de instâncias criadas corresponde ao número de execuções da operação *getPooled*.

Os resultados para média, desvio padrão e coeficente de variação de cada atividade medida são apresentados na Tabela 5.5.

Tabela 5.5. Média (μ), Desvio Padrão (σ) e Coeficiente de Variação para os Tempos das Atividades Representadas no Modelo

Transição	Média (μs)	Desvio Padrão (μs)	Coeficiente de Variação
clear	45,156	364,558	8,073
create	187,115	477,775	2,553
generateRequest	1.021,739	4.572,393	4,475
getPooled	2,018	6,450	3,196
invokeNext	16,207	299,122	18,456

log	41,375	323,135	7,810
put	3,140	59,657	19,011

A partir dos resultados obtidos, é necessário avaliar se a quantidade de observações medidas é suficiente. A Tabela 5.6 apresenta o número teórico de observações recomendado para cada atividade. Este número foi calculado com base na Equação (4.2), considerando-se os dados referentes à média e desvio amostrais apresentados na Tabela 5.5, um nível de confiança de 95% e uma precisão de 10% (erro máximo relativo). Além deste número, a Tabela 5.6 apresenta o número real de observações realizadas e o intervalo de confiança ao nível de 95% para as médias (representado através das colunas Mínimo e Máximo).

Tabela 5.6. Número de Observações e Intervalo de Confiança por Atividade.

Transição	Número Teórico	Número Real	Mínimo(μs)	Máximo(μs)
clear	25.039	401.740	44,029	46,283
create	2.505	60	66,221	308,009
generateRequest	7.694	401.740	1007,600	1035,878
getPooled	3.925	401.680	1,998	2,038
invokeNext	130.859	401.740	15,282	17,132
log	23.432	401.740	40,376	42,374
put	138.668	401.740	2,956	3,324

Conforme pode ser verificado, para quase todas as atividades o número de observações é bem maior do que o número teórico necessário, de forma que, para estas, pode-se assumir que a média populacional é bem representada pela média da amostra (ver Tabela 5.5). A exceção à esta regra é a atividade de criação de instâncias. Considerando-se a raridade desta atividade e as dificuldades de obtenção de amostras grandes, adotou-se a média calculada como média populacional.

Para concluir a etapa de parametrização do modelo, o tempo de cada transição temporizada apresentada foi configurado para o valor médio obtido para a atividade correspondente.

5.5.2.2 Verificação e Validação

Seguindo a abordagem apresentada no Capítulo 4, a etapa de verificação corresponde à análise de propriedades consideradas desejáveis para o modelo em questão. Para os modelos de desempenho projetados ao longo deste trabalho, a primeira propriedade a ser verificada é a limitação (ver Apêndice A.5). Esta propriedade permite que seja constatada a viabilidade de utilização de técnicas analíticas na derivação dos resultados de desempenho a partir dos modelos.

Para verificar a limitação do modelo servidor, seus invariantes de lugar devem ser calculados. Em geral, este cálculo deve ser realizado com o auxílio de ferramenta, uma vez que a solução do sistema de equações correspondentes

não é trivial. Em particular, a própria ferramenta utilizada para o projeto do modelo em si (TimeNET) [Zimmermann 2001] oferece esse recurso permitindo uma análise direta da limitação da rede. No que concerne ao modelo servidor, apenas os lugares *IsOpModeStrict* e *InterceptorReady* são cobertos pelos invariantes, e, portanto, limitados estruturalmente. Os demais lugares não possuem um limite teórico máximo para o número de *tokens* armazenados.

A ausência da propriedade de limitação impossibilita a obtenção de uma solução matemática para o modelo projetado. Logo, a validação deve ser realizada através simulação. Para o modelo servidor, o procedimento de validação compreende: 1) realização de novos experimentos de medição; 2) realização de simulação; 3) comparação dos resultados obtidos a partir dessas duas técnicas.

Uma decisão importante diz respeito ao regime considerado na realização dos experimentos de validação: transiente ou estacionário. Para tomar esta decisão, deve-se inicialmente estudar o tipo de informação a ser obtida a partir das métricas selecionadas, nesse caso, o número de instâncias. O objetivo associado à escolha de tal métrica é determinar um tamanho adequado para o *pool* de instâncias associado ao componente avaliado. Para este fim, uma escolha que pode parecer óbvia, em princípio, é a de um regime estacionário. Neste contexto, não se estaria interessado em descobrir o número de instâncias criadas durante um intervalo de tempo em particular, mas sim, o número total de instâncias necessárias para processar as requisições na taxa prevista. Contudo, a realização de um único experimento de medição em regime estacionário pode requerer horas. Além disso, uma vez que o modelo é teoricamente ilimitado, não há garantia de que essa métrica estaciona para a taxa prevista.

Por outro lado, a ausência de limitação inviabiliza a utilização de técnicas analíticas na obtenção de soluções para o modelo. A utilização de técnicas de simulação também não oferece uma solução, uma vez que experimentos de simulação estacionária podem ser extremamente lentos. Um aspecto que pode ter bastante relevância na duração de um experimento de simulação é a freqüência de ocorrência dos eventos. Simulações envolvendo eventos raros podem ser particularmente lentas, uma vez que os valores obtidos podem não convergir para o nível de confiança e a precisão estabelecidos.

Dado que o objetivo de um *pool* é minimizar o número de instâncias criadas, a criação de uma nova instância pode ser um evento raro. Neste cenário uma simulação estacionária do número de instâncias criadas pode ser inviável. De fato, experimentos de simulação estacionária para os modelos servidor e servidor refinado foram realizados. Contudo, esses experimentos acabaram por ser interrompidos após dias de execução sem a produção de resultados. Desta forma, o regime transiente foi selecionado para todos os experimentos realizados. Para que os resultados obtidos fossem minimamente afetados por essa escolha, estabeleceu-se uma duração 100 segundos por experimento, que é considerada elevada quando comparada às durações das atividades representadas no modelo, as quais são da ordem de microssegundos.

Nos novos experimentos de medição realizados para fins de validação, a gravação de informações referentes aos tempos das operações foi desabilitada e

a única porção do código de instrumentação habilitada foi a responsável pela gravação dos dados referentes à métrica selecionada (número de instâncias criadas). Ao todo, vinte e cinco novos experimentos de medição foram realizados. Novamente é importante mencionar que foram observadas as necessidades de *warmup* e de renício do servidor antes da realização de cada um.

A determinação do número experimentos necessários foi realizada com base na métrica considerada, neste caso, número de instâncias criadas ao longo do tempo. Pode-se pensar nesta métrica como um conjunto de variáveis aleatórias, cada uma correspondendo ao número de instâncias criadas até um determinado momento. Desta forma, cada experimento fornece uma única observação para cada instante de tempo particular. Conforme mencionado anteriormente, para que a média populacional possa ser adequadamente representada pela média amostral é necessário que a amostra possua pelo menos 25 a 30 observações. O resultado dos experimentos de medição é, então, dado através de uma curva, na qual o número de instâncias em cada instante é obtido através da média das 25 observações.

Os experimentos de simulação, por sua vez, foram realizados com o auxílio da ferramenta TimeNET. Estes experimentos finalizam sua execução quando os resultados atingem uma precisão configurada. A precisão das simulações de transiente realizadas são estabelecidas através de dois parâmetros. O primeiro é o nível de confiança que define a probabilidade de que o valor real para a medida pertença ao intervalo de confiança estabelecido. O segundo é o erro máximo relativo que define a largura do intervalo de confiança em função de um percentual da medida avaliada. Em particular, todos os experimentos de simulação realizados utilizaram as configurações padrões da ferramenta, as quais estabelecem um intervalo de confiança de $\pm 10\%$ e um nível de confiança de 95%. É importante observar que alterações nestas configurações para aumentar a precisão dos experimentos têm um forte impacto em termos dos tempos necessários à finalização destes.

A Figura 5.3 apresenta o número de instâncias do componente que foram criadas durante a realização dos experimentos de simulação e de medição. A partir da curva que representa os resultados da medição, pode-se observar que o crescimento no número de instâncias diminui a partir dos 30 segundos e que a curva começa a apresentar uma tendência de estabilização entre 7 e 8 instâncias.

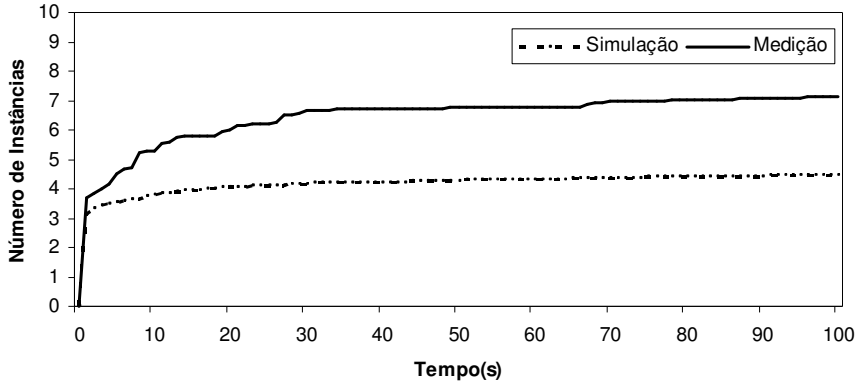


Figura 5.3. Modelo Servidor: Número de Instâncias Criadas

A Figura 5.4 mostra a diferença percentual entre os resultados obtidos nos experimentos de simulação e medição. Conforme pode ser observado, a diferença entre estes resultados não está dentro da faixa de 10 a 30 % de erro, que seria considerada razoável para experimentos de planejamento de capacidade [Menascé et al. 2004]. Esta constatação levou à necessidade de utilização das técnicas de *moment matching* descritas na Seção 2.3.5 e à definição do modelo servidor refinado apresentado na próxima seção.

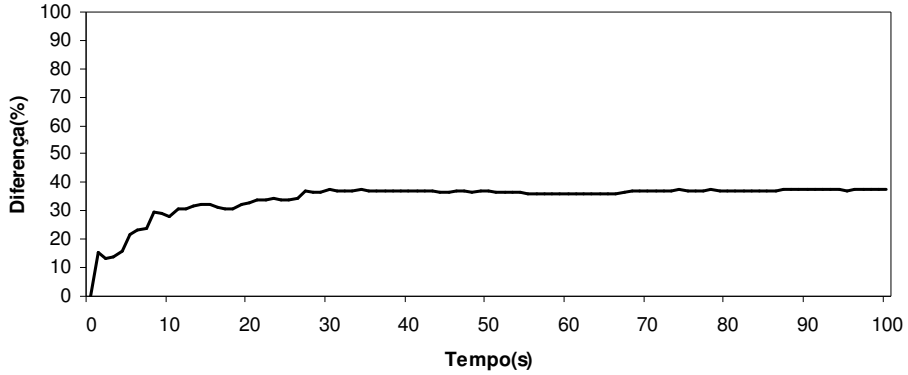


Figura 5.4. Modelo Servidor: Diferença entre Simulação e Medição

5.5.3 Modelo Servidor Refinado

Uma importante suposição implícita no modelo servidor é a de que os tempos associados às atividades podem ser representados através de variáveis aleatórias exponenciais com parâmetros iguais aos valores médios medidos. Contudo, os resultados dos experimentos de medição realizados indicam uma natureza exponencial para os tempos associados às atividades representadas. Em uma distribuição exponencial, a média e o desvio padrão têm um mesmo valor, de forma que o coeficiente de variação é unitário. Este não é o caso para as atividades temporizadas representadas no modelo (ver dados apresentados na Tabela 5.5). De fato, os dados obtidos nos experimentos de medição foram analisados utilizando-se a ferramenta Arena para executar os testes Chi-quadrado e Kolmogorov, levando à conclusão de que a aproximação por uma exponencial não é adequada.

Visando melhorar os resultados obtidos, o modelo proposto foi refinado utilizando o algoritmo de *moment matching* apresentado na Seção 2.3.5. Segundo este algoritmo, todas as transições temporizadas apresentadas no modelo servidor devem ser aproximadas por *s-transitions* hiperexponenciais, uma vez que os coeficientes de variação correspondentes são maiores do que um (ver Tabela 5.5). Uma *s-transition* hiperexponencial é apresentada na Figura 2.11. Os valores dos parâmetros r_1, r_2 e λ, para cada transição temporizada representada no modelo são apresentados na Tabela 5.7.

O modelo obtido a partir da substituição das transições temporizadas por *s-transitions* é chamado "modelo servidor refinado". A parte central deste modelo é apresentada na Figura 5.5 e é referenciada como "subrede de controle", uma vez que possui controle sobre a dinâmica do sistema modelado.

É importante observar que todas as transições temporizadas aproximadas por *s-trasitions* foram removidas da subrede de controle e representadas em subredes separadas (ver Figura 5.6). A subrede de controle e as subredes que

representam *s-transitons* são conectadas usando lugares e transições cujos nomes começam com *Starting*, *Ready*, *startUp*, *tearDown*, *nextStep*, *enable*, *run* e *remove*. A utilização destes lugares e transições, chamados "conectores", associados ao conjunto de funções de habilitação e de prioridades facilitam a substituição das transições temporizadas pelas *s-transitons* correspondentes.

Tabela 5.7. Modelo Servidor Refinado: Parâmetros das *s-transitons*.

Transição	r1	r2	λ
clear	0,0302	0,9698	0,0006692729
create	0,2660	0,7340	0,0014214124
getPooled	0,1783	0,8217	0,0883707783
generateRequest	0,0951	0,9049	0,0000930938
log	0,0323	0,9677	0,0007797157
invokeNext	0,0059	0,9941	0,0003612167
put	0,0055	0,9945	0,0017597787

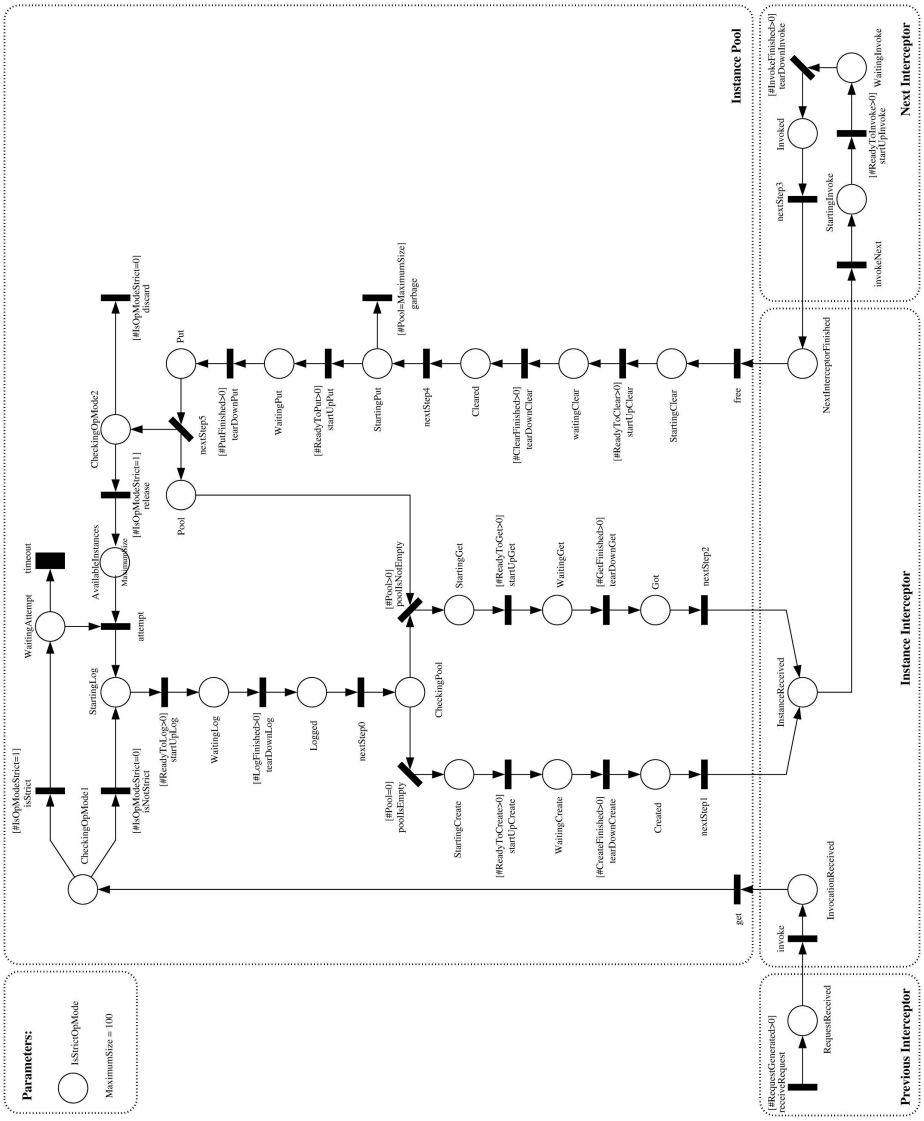


Figura 5.5. Modelo Servidor Refinado: Subrede de Controle.

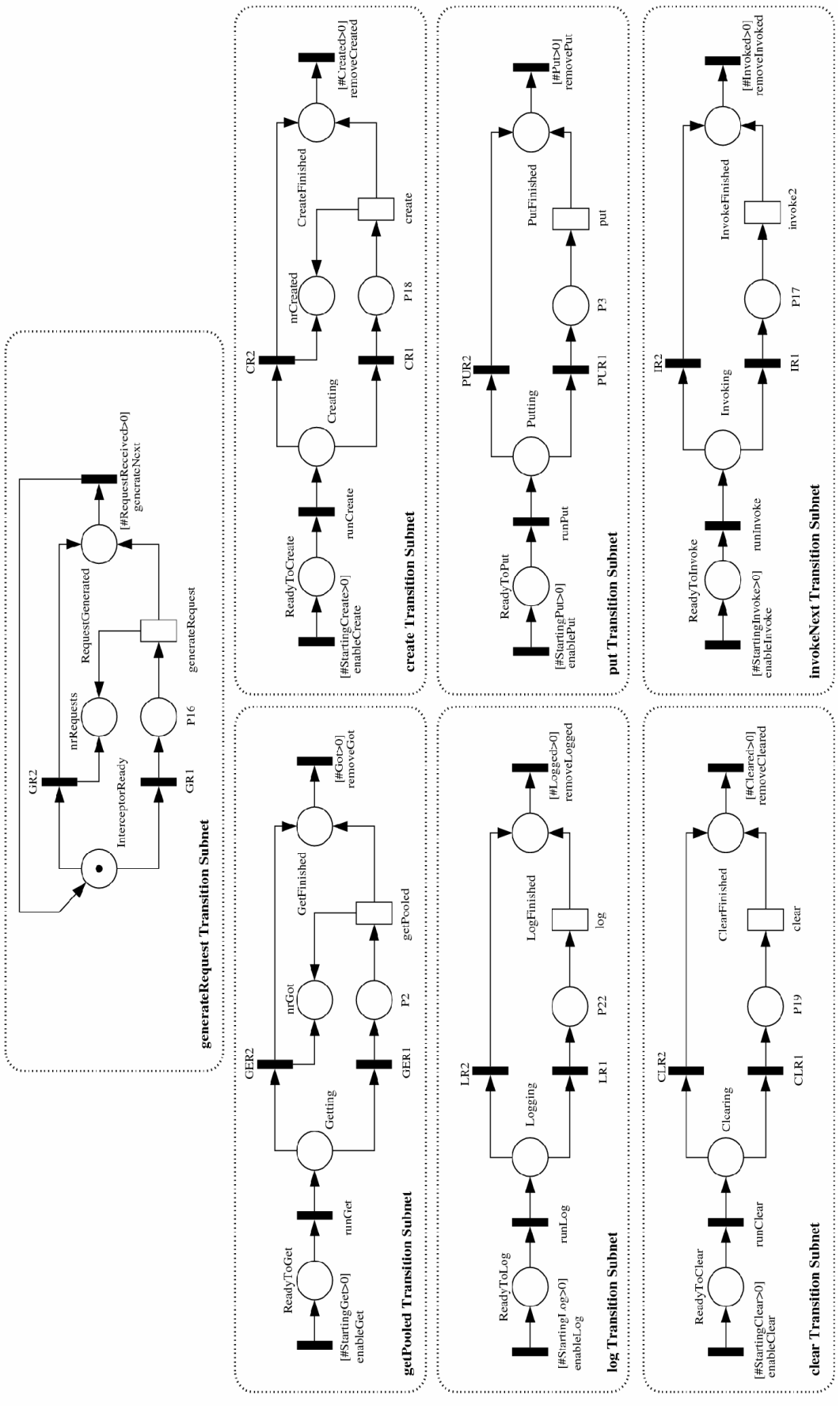


Figura 5.6. Visão Geral Modelo Servidor Refinado: *s-transitons*.

A Tabela 5.8 apresenta os pesos, prioridades e condições de guarda para todas as transições imediatas presentes no modelo, completando a sua descrição.

Tabela 5.8. Modelo Servidor Refinado: Peso, Prioridade e Função de Habilitação para as transicões imediatas

Transição	Peso	Prioridade	Função de Habilitação
receiveRequest	1	1	*#RequestGenerated>0*
invoke	1	2	-
get	1	1	-
isStrict	1	2	*#IsOpModeStrict=1*
isNotStrict	1	2	*#IsOpModeStrict=0*
attempt	1	1	-
startUpLog	1	15	*#ReadyToLog>0*
tearDownLog	1	13	*#LogFinished>0*
nextStep0	1	5	-
poolIsEmpty	1	8	*#Pool=0*
poolIsNotEmpty	1	8	
startUpCreate	1	15	*#ReadyToCreate>0*
tearDownCreate	1	13	*#CreateFinished>0*
nextStep1	1	5	-
startUpGet	1	15	*#ReadyToGet>0*
tearDownGet	1	13	*#GetFinished>0*
nextStep2	1	5	-
free	1	5	-
startUpClear	1	15	*#ReadyToClear>0*
tearDownClear	1	13	*#ClearFinished>0*
nextStep4	1	5	-
garbage	1	20	*#Pool=MaximumSize*
discard	1	1	*#IsOpModeStrict=0*
startUpPut	1	15	*#ReadyToPut>0*
tearDownPut	1	13	*#PutFinished>0*
nextStep5	1	5	-
release	1	1	*#IsOpModeStrict=1*
enableInvoke	1	4	*#StartingInvoke>0*
runInvoke	1	5	-
IR1	0,0059	2	-
IR2	0,9941	2	-
removeInvoke	1	10	*#Invoked>0*
GR1	0,0951	2	-
GR2	0,9049	2	-
generateNext	1	9	*#RequestReceived>0*
enableCreate	1	4	*#StartingCreate>0*
runCreate	1	5	-
CR1	0,2660	2	-
CR2	0,7340	2	-
removeCreated	1	10	*#Created>0*
enableGet	1	4	*#StartingGet>0*

Transição	Peso	Prioridade	Função de Habilitação
runGet	1	5	-
GER1	0,1783	2	-
GER2	0,8217	2	-
removeGot	1	10	*#Got>0*
enableLog	1	4	*#StartingLog>0*
runLog	1	5	-
LR1	0,0323	2	-
LR2	0,9677	2	-
removeLogged	1	10	*#Logged>0*
enableClear	1	4	*#StartingClear>0*
runClear	1	5	-
CLR1	0,0302	2	-
CLR2	0,9698	2	-
removeCleared	1	10	*#Cleared>0*
enablePut	1	4	*#StartingPut>0*
runPut	1	5	-
PUR1	0,0055	2	-
PUR2	0,9945	2	-
removePut	1	10	*#Put>0*
invokeNext	1	5	-
startUpInvoke	1	15	*#ReadyToInvoke*
tearDownInvoke	1	13	*#InvokeFinished*
nextStep3	1	5	-

5.5.3.1 Verificação e Validação

Assim como o modelo anterior, o modelo servidor refinado também é teoricamente ilimitado, uma vez que diversos lugares representados não são corbertos por invariantes. A ausência da propriedade de limitação inviabiliza a obtenção de soluções analíticas, motivando o uso da técnica de simulação na obtenção dos resultados de desempenho.

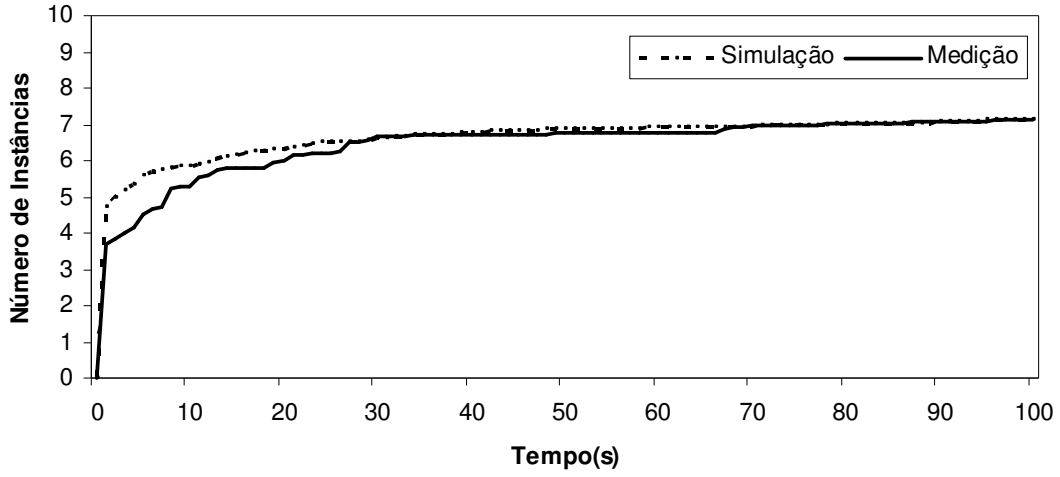


Figura 5.7. Modelo Servidor Refinado: Número de Instâncias Criadas.

Desta forma, para validar o modelo refinado novos experimentos de simulação foram realizados. Novamente foram utilizados os valores *default* dos

parâmetros de simulação configuráveis através da ferramenta (nível de confiança igual a 95% e erro máximo relativo de 10%). A Figura 5.7 apresenta o número de instâncias criadas nesses experimentos. A diferença percentual entre os resultados obtidos na simulação do modelo e nos experimentos de medição é apresentada na Figura 5.8.

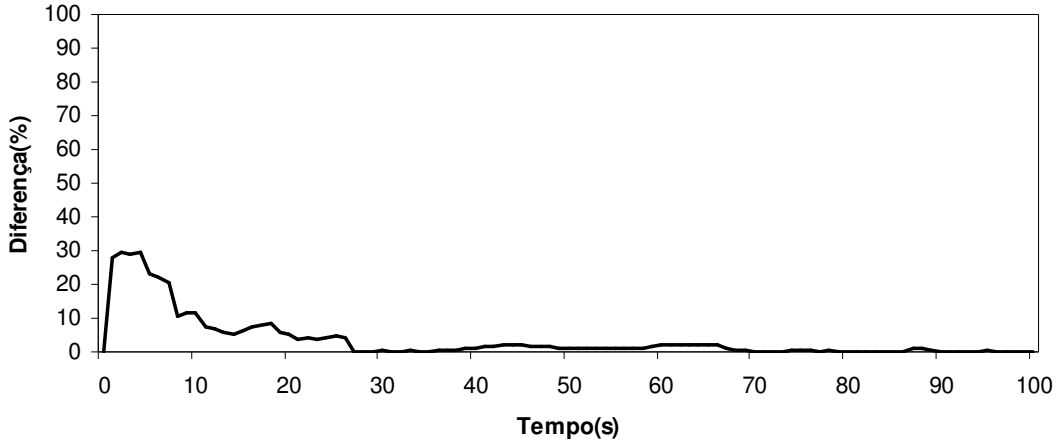


Figura 5.8. Modelo Servidor Refinado: Diferença entre Simulação e Medição.

Pode-se observar que durante o intervalo de observação a diferença entre os resultados de simulação e medição não ultrapassaram 30%. Além disto, a diferença percentual entre estes resultados diminui ao longo do tempo, ficando na maior parte dentro de uma margem de 5% de erro. Desta forma, pode-se afirmar que os resultados são bastante próximos, valindando o refinamento realizado e o próprio modelo desenvolvido em si.

Algumas importantes considerações podem ser feitas à cerca dos modelos apresentados até aqui. Primeiramente, é importante relembrar que os modelos foram parametrizados com base em medições realizadas com o sistema atendendo à mesma carga utilizada na realização dos experimentos de simulação e medição, correspondendo à execução simultânea de 10 clientes tentando realizar, cada um, 100 requisições por segundo. Sendo assim, durante o experimento realizado para fins de parametrização, o servidor está efetivamente processando requisições concorrentes. Para processar cada requisição, o servidor deve realizar diversas atividades, dentre elas, aquelas representadas no modelo. A execução simultânea dessas atividades envolve uma disputa entre elas por recursos de hardware (*e.g.*, CPU) e de software fornecidos pelo servidor. Desta forma, o tempo medido para cada atividade no experimento de parametrização envolve, além do tempo de serviço em si, um tempo de fila correspondendo à espera pela alocação dos recursos necessários (ver Seção 2.2.1). Além disto, dependendo do recurso, o próprio tempo do serviço em si pode variar de acordo com a carga aplicada. A implicação imediata destes fatos é que os tempos com os quais os modelos foram parametrizados estão totalmente relacionados à própria carga em si. Desta forma, a utilização destes modelos fica limitada aos cenários envolvendo a aplicação da mesma carga utilizada na sua parametrização.

Outra consideração importante diz respeito às métricas que podem ser extraídas a partir dos modelos propostos. Conforme mencionado anteriormente, os modelos apresentados representam uma visão do processamento de

requisições tomada do lado servidor, de forma que os clientes não são explicitamente representados. Nestes modelos, o componente *Previous Interceptor* representa a carga resultante das requisições simultâneas enviadas pelos clientes. Sendo assim, métricas relativas aos clientes não podem ser representadas. Da mesma forma, como no modelo não há a representação explicita dos recursos, métricas de utilização não podem ser estabelecidas.

Uma consideração final a ser feita diz respeito à ausência da propriedade de limitação da rede. O modelo apresentado, tanto em sua versão original, quanto na refinada, não é coberto por invariantes de lugar, de forma que o número de *tokens* não possui um limite teórico. Esta ausência de limitação inviabiliza a solução do modelo através da utilização de técnicas de análise, sendo viável apenas a utilização da técnica de simulação para a obtenção das métricas de desempenho.

Para minimizar os problemas mencionados, o próximo modelo representa explicitamente a concorrência pela CPU entre atividades realizadas no servidor, e é parametrizado a partir de dados estatísticos medidos com um único cliente em execução. Além disto, esse modelo é limitado, permitindo a obtenção de resultados através de técnicas de análise ou de simulação. Por fim, o novo modelo viabiliza o cálculo de outras métricas uma vez que representa tanto a porção servidora quanto a porção cliente da aplicação de teste, sendo, desta forma, referenciado como “modelo cliente/servidor".

5.5.4 Modelo Cliente/Servidor

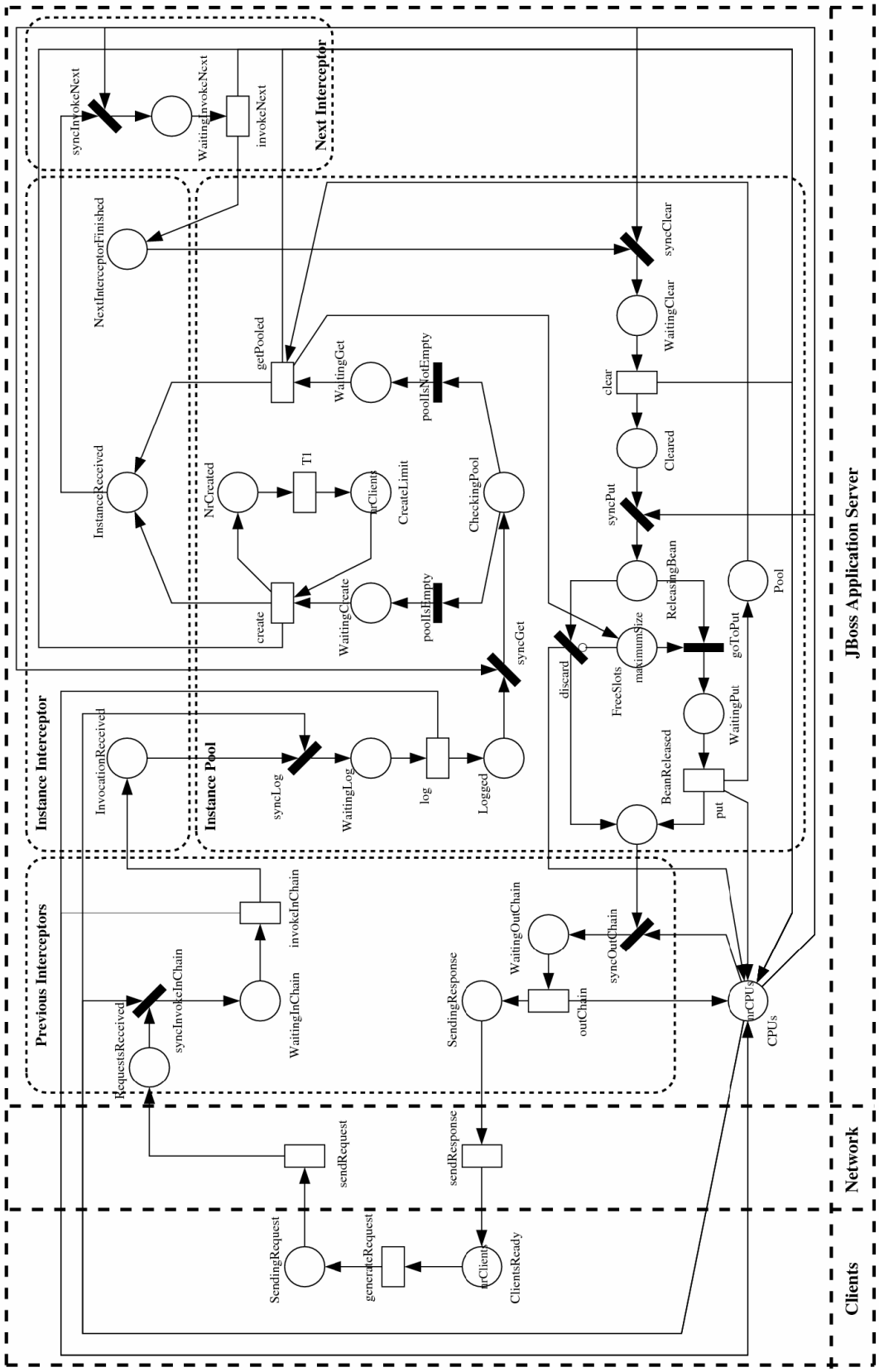
A Figura 5.9 apresenta o modelo cliente/servidor que possui três componentes: *Clients*, *Network* e *JBoss Application Server*. Neste modelo, optou-se por representar apenas o modo de operação não estrito. Desta forma, o número de instâncias criadas não é limitado.

O componente *Clients* representa a interação dos usuários com a aplicação de teste gerando requisições. Conforme mencionado anteriormente, a aplicação de teste desenvolvida é uma aplicação Web, de forma que toda esta interação ocorre através de um navegador (*browser*). Cada solicitação do usuário corresponde à uma requisição HTTP.

A representação do componente *Clients* no modelo é realizada através da subrede composta pelos lugares *ClientsReady* e *SendingRequest*, pela transição *generateRequest* e pelos seus arcos de entrada e saída. Na sua marcação inicial, *ClientsReady* possui *nrClients tokens*, cada um representando um cliente ativo. Para modelar clientes simultâneos gerando requisições com taxas seguindo uma distribuição de *Poisson*, a transição *generateRequest* é exponencial e tem semântica *infinite-server*. A geração de requisições é representada através do disparo da transição *generateRequest*, levando à criação de *tokens* no lugar *SendingRequest*. Estes *tokens* habilitam a transição *sendRequest* representando a solicitação de transmissão das requisições através da rede. O tempo da transição *generateRequest*, modelado pela variável *dRequest*, é utilizado como fator no modelo, podendo ser variado de forma a representar diferentes taxas de requisição.

A rede que interliga clientes e servidor é representada no modelo através do componente *Network*, o qual compreende duas transições exponenciais,

sendRequest e *sendResponse*, representando, respectivamente, a transmissão de requisições e de respostas.



Por sua vez, o componente *JBoss Application Server* possui quatro componentes internos: *Previous Interceptors*, *Instance Interceptor*, *Instance Pool* e *Next Interceptors*. Estes componentes acessam a CPU de forma

compartilhada, a qual é representada explicitamente através de um *token* no lugar *CPU*. Antes de realizar qualquer atividade representada por uma transição temporizada, um componente deve alocar uma CPU disponível. Esta alocação é efetuada no modelo proposto através das transições imediatas com nome iniciando com *sync*. Após obter a CPU, um componente entra em um de seus estados locais representados pelos lugares cujos nomes começam com *Waiting* e permanece neste estado até que a atividade correspondente conclua sua execução e libere a CPU.

O componente *PreviousInterceptors* representa os elementos do JBoss responsáveis pela interação deste servidor com o mundo exterior. Ao receber uma requisição, modelada através de um *token* no lugar *RequestsReceived*, este componente a encaminha para o componente correspondente através das cadeias de interceptadores do *proxy* e do *container*. Todo processamento realizado sobre esta requisição até que ela seja recebida pelo *Instance Interceptor* é representado pela transição *inChain*.

Da mesma forma que a requisição, a resposta também é processada, passando pelas mesmas cadeias de interceptadores, mas agora no sentido contrário. Todo processamento realizado após a liberação da instância e seu eventual armazenamento no *pool* é representado pela transição *outChain*. Quando esta transição dispara, indicando a conclusão do processamento da resposta, um *token* é criado no lugar *SendingResponse*, indicando que a resposta está pronta para ser encaminhada para o cliente.

O componente *Instance Interceptor* é o responsável pela obtenção de uma instância do componente a ser utilizada no processamento da requisição. As requisições recebidas por este componente são representadas através de *tokens* gerados no lugar *InvocationReceived*. Para processar uma requisição, o *Instance Interceptor* requisita uma instância do *bean* ao componente *Instance Pool*, o qual atua como um repositório de instâncias disponíveis.

De forma geral, a obtenção de instâncias segue os mesmos passos representados no modelo servidor. Cada requisição de instância recebida pelo *Instance Pool* é registrada em um arquivo de *log* do JBoss, sendo este registro representado no modelo através da transição *log*. Em seguida, o *Instance Pool* verifica a existência de instâncias disponíveis, as quais são modeladas através de *tokens* no lugar *Pool*. Esta verificação é representada pelo lugar *CheckingPool* e pelas transições *poolIsEmpty* e *poolIsNotEmpty*, as quais são mutuamente exclusivas, conforme indicado por suas funções de habilitação, *#Pool=0* e *#Pool>0*, respectivamente. Se o *Pool* contém instâncias (*tokens*), a transição *poolIsNotEmpty* dispara e gera um *token* no lugar *WaitingGet*. Este *token* habilita a transição *getPooled* que representa a atividade de obtenção de uma instância que já foi anteriormente criada. Caso contrário, se não há instâncias disponíveis, a transição *poolIsEmpty* dispara, movendo um *token* para o lugar *WaitingCreate*, indicando que uma nova instância deve ser criada.

No modelo, a criação de instâncias é representada pela transição *create*. Cada instância criada é contabilizada através da geração de um *token* no lugar *NrCreated*. Para que o modelo cliente/servidor seja estruturalmente limitado, os *tokens* não devem ser acumulados indefinidamente neste lugar. Desta forma, para viabilizar a contagem das instâncias criadas e, ao mesmo tempo, manter a

limitação do modelo, foram inseridos o lugar *CreateLimit* e a transição *T1*. Para que a representação destes dois elementos não interfira no comportamento geral do modelo é necessário que o lugar *CreateLimit* seja inicializado com *nrClients tokens*, garantindo que o número máximo teórico de instâncias pode ser criado (este número máximo corresponde à criação de uma instância para cada cliente). Além disto, é importante que a transição *T1* tenha um *delay* elevado, minimizando a ocorrência de disparos e a influência destes na contabilização das instâncias criadas. Em experimentos de transiente, o ideal é que o tempo médio associado à *T1* seja maior que o tempo considerado para o experimento.

A instância selecionada pelo *Instance Pool*, recém-criada ou não, é retornada para o *Instance Interceptor* na forma de um *token* depositado no lugar *InstanceReceived*. Ao recebê-la, o *Instance Interceptor* aciona o componente *Next Interceptor*, dando sequência ao processamento da requisição. Este componente possui uma única transição, *invokeNext*, que representa conjuntamente toda a atividade realizada pelo demais interceptadores da cadeia do *container* e pelo próprio componente. Quando esta transição dispara, a instância do *bean* utilizada no processamento da requisição é retornada ao *Instance Interceptor* sob a forma de um *token* criado no lugar *NextInterceptorFinished*. Finalmente, o *Instance Interceptor* pode retornar a instância utilizada para o *Instance Pool*.

Ao receber de volta uma instância, o *Instance Pool* remove todas as informações associadas ao processamento da requisição anterior. Esta atividade é representada no modelo através da transição *clear*. Em seguida, o *Instance Pool* verifica o número de instâncias armazenadas por ele (ou seja, o número de *tokens* no lugar *Pool*) e a capacidade máxima configurada, representada pelo parâmetro *maximumSize*. Com base nestas informações, o *InstancePool* decide se a instância deve ser descartada (tornar-se elegível pelo mecanismo de coleta de lixo) ou se deve ser armazenada para reuso futuro. Esta decisão é modelada através dos lugares *ReleasingBean* e *FreeSlots* e das transições *discard* e *goToPut*. Por fim, um *token* é gerado no lugar *BeanReleased* indicando que o *bean* foi liberado e que a resposta pode seguir através dos demais interceptadores.

As atividades realizadas pelos demais interceptadores no processamento da resposta são representadas conjuntamente através da transição *outChain*. Quando esta transição dispara, um *token* é gerado no lugar *SendingResponse*, indicando que uma resposta está sendo enviada para o cliente. A trasmissão desta resposta é modelada através da transição *sendResponse*.

Completando a descrição do modelo, Tabela 5.9 e Tabela 5.10 apresentam os detalhes das transições imediatas e das transições temporizadas, respectivamente. A obtenção dos tempos apresentados é realizada na atividade de parametrização que é descrita na seção seguinte.

Tabela 5.9. Modelo Cliente/Servidor: Transições Imediatas.

Transição	Peso	Prioridade	Função de Habilitação
discard	1	1	-
poolIsEmpty	1	1	#Pool=0
poolIsNotEmpty	1	1	#Pool>0
syncClear	1	1	-
syncGet	1	1	-
syncInvokeInChain	1	1	-
syncInvokeNext	1	1	-
syncLog	1	1	-
syncOutChain	1	1	-
syncPut	1	1	-

Tabela 5.10. Modelo Cliente/Servidor: Transições Temporizadas.

Transição	Tempo Médio (μs)	Semântica de Disparo
clear	2,147	*Infinite-Server*
create	187,115	*Infinite-Server*
getPooled	1,967	*Infinite-Server*
inChain	212,198	*Infinite-Server*
invokeNext	5,607	*Infinite-Server*
log	2,274	*Infinite-Server*
outChain	24,739	*Infinite-Server*
put	1,985	*Infinite-Server*
sendRequest	24,508	*Single-Server*
sendResponse	59,668	*Single-Server*

5.5.4.1 Parametrização

A parametrização do modelo cliente/servidor envolve a parametrização de cada um de seus três componentes: *Clients*, *Network* e *JBoss Application Server*. Contudo, uma vez que a única atividade realizada pelo componente *Clients* é a geração de requisições e que a taxa de geração é utilizada como fator variável nos cenários montados a partir do modelo, este componente não necessita ser parametrizado.

Para parametrizar o componente *JBoss Application Server* foram novamente realizados seis experimentos. Em cada um deles, um único cliente realiza 50.000 requisições a um componente EJB utilizando a mesma aplicação Web de testes desenvolvida para calibrar o modelo servidor. Destas 50.000 requisições, 10.000 são descartadas para atender às necessidades de *warm-up* da JVM executando JBoss. Desta forma, o total de requisições consideradas para efeito de cálculo dos tempos médios das atividades é de 240.000. Durante estes experimentos, o código de instrumentação foi configurado para registrar a duração de cada atividade representada no modelo. Entre um experimento e outro, o servidor JBoss foi reiniciado, minimizando a interferência entre dois experimentos consecutivos.

A Tabela 5.11 apresenta média, desvio-padrão e coeficiente de variação referentes aos tempos medidos para cada atividade. Nessa tabela, um caso particular é o da transição *create*. Uma vez que um único cliente é utilizado em cada experimento de parametrização do modelo cliente/servidor, uma única instância é criada e utilizada para atender todas as 50.000 requisições. Conforme pode ser visto na Tabela 5.5, que apresenta os resultados obtidos na parametrização do modelo servidor apresentado anteriormente, o tempo de criação de uma instância varia muito. Dada esta variação, optou-se por, ao invés de parametrizar o modelo cliente/servidor com o tempo referente à criação da única instância utilizada nos experimentos de parametrização desse modelo, utilizar o tempo médio de criação de instâncias obtidas nos experimentos de parametrização do modelo servidor.

Tabela 5.11. Média, Desvio Padrão e Coef. De Variação Medidos

Transição	Média (μs)	Desvio Padrão (μs)	Coeficiente de Variação
clear	2,147	0,568	0,265
create	187,115	477,775	2,553
getPooled	1,967	0,521	0,265
inChain	212,198	805,235	3,795
invokeNext	5,607	88,007	15,695
log	2,274	2,413	1,061
outChain	24,739	98,401	3,978
put	1,985	0,552	0,278

Assim como foi feito na parametrização do modelo servidor, deve-se avaliar se a quantidade de observações realizadas para cada atividade é suficiente. Neste caso, todas as 240.000 requisições geradas pelo único cliente utilizado foram processadas pelo servidor, de forma que todas as atividades, exceto as modeladas por *create* e *getPooled*, foram executadas exatamente este número de vezes. A Tabela 5.12 apresenta o número teórico de observações recomendado para cada atividade e o número real de observações realizadas. O cálculo do número recomendado teve como base a Equação (4.2) e considerou um nível de confiança de 95% e uma precisão de 10% (erro máximo relativo). A Tabela 5.12 apresenta, também, o intervalo de confiança ao nível de 95% para o tempo médio das atividades considerando o número real de observações realizadas (representado através das colunas Mínimo e Máximo).

Tabela 5.12. Número de Observações e Intervalo de Confiança por Atividade.

Transição	Número Teórico	Número Real	Mínimo(μs)	Máximo(μs)
clear	27	240.000	2,145	2,149
getPooled	27	239.994	1,965	1,969
inChain	5532	240.000	208,976	215,420
invokeNext	94643	240.000	5,255	5,959
log	433	240.000	2,264	2,284
outChain	6078	240.000	24,345	25,133

put	30	240.000	1,983	1,987

Conforme pode ser observado, para todas as atividades, à exceção da atividade de criação de instâncias, o número real de observações realizadas foi bem maior do que o número recomendado, garantindo a validade da utilização das médias amostrais na parametrização do componente *JBoss Application Server*.

Para obter os dados necessários à parametrização do componente *Network*, os experimentos realizados foram monitorados através da ferramenta Ethereal.

Cada solicitação realizada por um cliente através da aplicação de teste corresponde à realização de uma requisição HTTP. Como resposta à esta requisição, a aplicação monta uma página HTML simples (sem nenhum objeto como figura ou *applet* embutido) e a encaminha para o cliente. Os tamanhos dos frames Ethernet necessários à trasmissão de uma requisição e de uma resposta (medidos com o auxílio da ferramenta) são 276 e 639 bytes, respectivamente (já incluindo os cabeçalhos dos protocolos da pilha TCP/IP). Considerando-se que a rede utilizada tem uma largura de banda de 100 Mbps, pode-se calcular o tempo médio de serviço da rede para ambos os casos.

Sejam:

T_{req}: Tempo de envio de requisição,

T_{resp}: Tempo de envio de resposta,

B: Largura de banda da rede em Mbps,

S_{req}: Tamanho da mensagem de requisição em bytes,

S_{resp}: Tamanho da mensagem de resposta em bytes.

Considerando-se os valores medidos pela ferramenta:

S_{req} = 276 bytes

S_{resp}=639 bytes

Tem-se:

$$T_{req} = \frac{S_{req} \times 8}{B} = \frac{276 \times 8}{100 \times 10^6} = 22{,}08 \times 10^{-6} s = 22{,}08 \mu s \quad \text{e}$$

$$T_{resp} = \frac{S_{resp} \times 8}{B} = \frac{639 \times 8}{100 \times 10^6} = 51{,}12 \times 10^{-6} s = 51{,}12 \mu s$$

Nos cálculos apresentados acima não foram considerados os tempos de abertura e fechamento de conexões. Para que estes tempos possam ser levados em consideração é necessário que seja feita uma proporção considerando-se o número de conexões abertas e o número total de requisições. Considerando-se que em 100.000 requisições observadas nos experimentos somente 1.709 provocaram a abertura de conexões TCP, o tempo corrigido referente ao envio de requisições e de respostas pode ser calculado conforme apresentado abaixo.

Sejam:

N_{req}: Número de requisições,

N_{con}: Número de conexões abertas,

$T_{abertura}$: Tempo médio de abertura de conexão observado na ferramenta a partir dos tempos das mensagens trocadas durante o *three-way handshake*,

$T_{fechamento}$: Tempo médio de fechamento de conexão observado na ferramenta a partir dos tempos associados as quatro mensagens de encerramento de conexão,

T_{reqCor}: Tempo de envio de requisição corrigido,

$T_{respCor}$: Tempo de envio de resposta corrigido.

Através da ferramenta Ethereal, tem-se:

N_{req}=100.000

N_{con}=1.709

$T_{abertura}$= 142,08μs

$T_{fechamento}$= 500,21μs

A partir destes dados, pode-se calcular o tempo médio para requisição e resposta conforme indicado abaixo:

$$T_{reqCor} = \frac{N_{con}}{N_{req}} \times \left(T_{abertura} + T_{req}\right) + \left(1 - \frac{N_{con}}{N_{req}}\right) \times T_{req}$$

$$= \frac{1709}{100000} \times (142{,}08 + 22{,}08) + \left(1 - \frac{1709}{100000}\right) \times 22{,}08 = 24{,}5081\mu s$$

$$T_{respCor} = \frac{N_{con}}{N_{req}} \times \left(T_{fechamento} + T_{resp}\right) + \left(1 - \frac{N_{con}}{N_{req}}\right) \times T_{resp}$$

$$= \frac{1709}{100000} \times (500{,}21 + 51{,}12) + \left(1 - \frac{1709}{100000}\right) \times 51{,}12 = 59{,}6685\mu s$$

5.5.4.2 Verificação e Validação

O modelo cliente/servidor, ao contrário dos modelos anteriormente apresentados, é estruturalmente limitado. De fato, este modelo apresenta quatro invariantes cobrindo todos os lugares da rede, conforme pode ser constatado através de uma análise estrutural do modelo realizada com o auxílio da ferramenta TimeNET (ver Quadro 5.1). Outra propriedade interessante apresentada pelo modelo cliente/servidor é a ausência de *deadlocks*. Tais propriedades garantem a possibilidade teórica de geração do grafo de alcançabilidade, e, conseqüentemente, de utilização de técnicas analíticas na obtenção dos resultados de desempenho.

A viabilidade da construção e solução da cadeia de Markov a partir do modelo depende, contudo, do número máximo de *tokens* acumulados nos lugares representados e da capacidade de processamento da máquina que executa o algoritmo de análise embutido na ferramenta TimeNET. Resultados analíticos foram obtidos a partir do modelo cliente/servidor para um número

máximo de cinco clientes. Resultados com um número maior de clientes podem ser obtidos utilizando-se simulação, embora experimentos desta natureza possam demorar bastante. Por exemplo, alguns experimentos de simulação executados com o modelo cliente/servidor chegaram a demorar dez dias antes de fornecer resultados.

Quadro 5.1. Modelo Cliente/Servidor: Invariantes de Lugar.

```
The net contains 4 P-invariants.

CPUs + CheckingPool + WaitingCreate + ReleasingBean +
WaitingGet + WaitingInChain + WaitingOutChain + WaitingLog +
WaitingInvokeNext + WaitingClear + WaitingPut = nrCPUs

ClientsReady + CheckingPool + WaitingCreate +
InstanceReceived + ReleasingBean + BeanReleased +
WaitingGet + RequestsReceived + InvocationReceived +
SendingResponse + SendingRequest + WaitingInChain +
WaitingOutChain + WaitingLog + Logged + WaitingInvokeNext +
NextInterceptorFinished + WaitingClear + Cleared +
WaitingPut = nrClients

Pool + FreeSlots + WaitingPut = maximumSize

NrCreated + CreateLimit = 100

All places are covered by p-invariants.
```

O processo de validação do modelo cliente/servidor envolveu a realização de experimentos baseados nas técnicas de medição e simulação. Nestes experimentos foram considerados três cenários distintos. Os dois primeiros compreendem a utilização de 5 e 10 clientes intensivos em termos de solicitações, cada um tentando realizar 100 requisições por segundo. O terceiro cenário representa uma carga mais real, composta por 100 clientes, cada um tentando realizar 1 requisição por segundo.

Três métricas independentes fornecem uma ampla visão da situação operacional do servidor de aplicação, compreedendo tanto aspectos visíveis do ponto de vista dos clientes quando do servidor. A primeira delas é a mesma métrica considerada no modelo servidor apresentado anteriormente e corresponde ao número de instâncias criadas para atender as requisições recebidas. A segunda métrica avaliada corresponde a taxa na qual o servidor de aplicação consegue processar requisições (*throughput*). Por fim, a terceira métrica considerada corresponde ao percentual de utilização da CPU. Os valores assumidos por estas métricas ao longo da realização dos experimentos são utilizados para efeito de validação do modelo proposto. O Quadro 5.2 apresenta as fórmulas utilizadas no cálculo destas métricas em função dos lugares representados no modelo. Nestas fórmulas $\#P1$ representa o número de *tokens* contidos no lugar *P1*, $E\{\#P1\}$ significa a esperança matemática do número de *tokens* contidos em *P1* e *dRequest* é o intervalo de tempo entre requisições sucessivas de um mesmo cliente em microssegundos.

Quadro 5.2. Modelo Cliente/Servidor: Métricas

```
CPUUtilization=(1-E{#CPUs})*100

Instances=E#{NrCreated}
```

```
Throughput=E{#ClientsReady}*(1/dRequest)*1000000
```

Para cada um dos três cenários utilizados na validação desse modelo foram realizados trinta e cinco experimentos de medição, visando, assim, obter um número mínimo de observações de forma a garantir a representatividade dos resultados conforme discutido na Seção 4.8. Novamente, todas as necessidades de *warmup* e de reinicialização do servidor foram observadas, e a gravação de informações referentes aos tempos das operações foi desabilitada. Por sua vez, os experimentos de simulação realizados utilizaram os valores *default* para erro máximo relativo (±10%) e nível de confiança (95%). Na seqüência são apresentados os resultados obtidos em cada um dos cenários considerados.

Cenário 1: 5 Clientes a 100 req/s

Neste primeiro cenário, os clientes realizam requisições a uma taxa bastante elevada fazendo com que seja necessário praticamente uma instância do componente para cada cliente. A Figura 5.10 apresenta o número médio de instâncias criadas nos experimentos de medição e simulação.

Conforme pode ser observado, há uma pequena diferença entre as curvas de medição e simulação. De fato, após os 100 segundos considerados nos experimentos de medição foi constatado que, em média, haviam sido criados 4,97 instâncias, enquanto que o resultado dos experimentos de simulação apontam um total de 4,32 instâncias, compreendendo uma diferença percentual de apenas 13,09%.

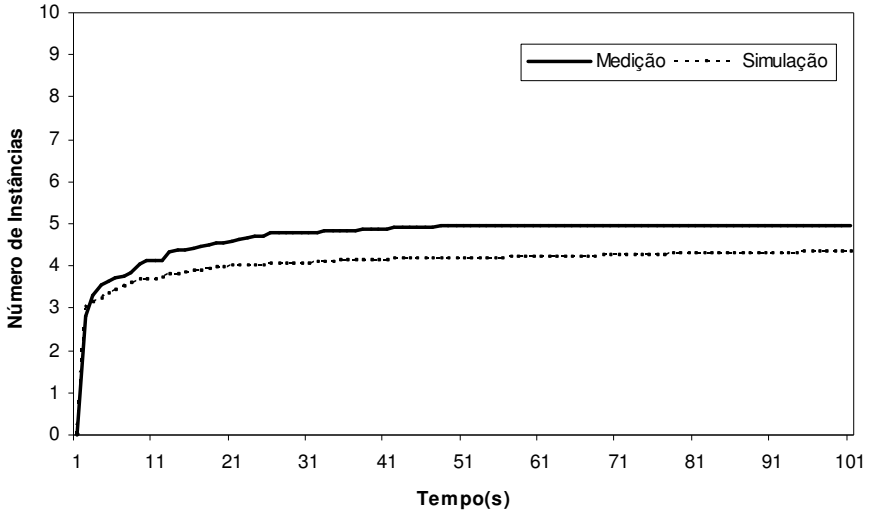


Figura 5.10. Cenário1 (5 Clientes a 100 Req/s): Número de Instâncias Criadas.

A diferença obtida entre os resultados de medição e simulação (apresentada graficamente na Figura 5.11), contudo, é considerada bastante razoável (dentro da faixa de 10% a 30% mencionada anteriormente) para uma métrica de utilização, ainda mais considerando que o número de instâncias é por natureza, um valor inteiro.

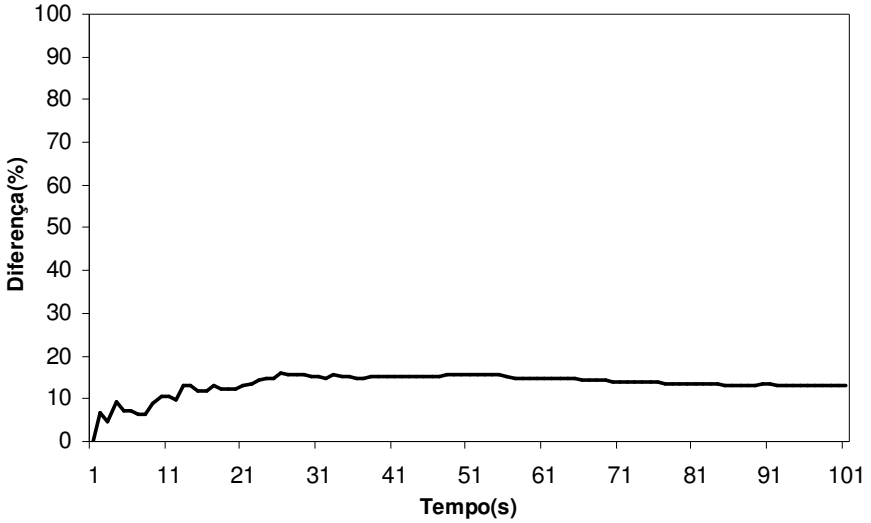


Figura 5.11. Cenário1 (5 Clientes a 100 Req/s): Diferença entre Simulação e Medição.

Do ponto de vista de *throughput*, uma vez que o componente não executa nenhum processamento ou entrada/saída elaborados, o servidor consegue processar requisições praticamente na taxa nominal associada aos clientes, ou seja, 100 req/s. Efetivamente observou-se nos experimentos de medição que, em média, o servidor processa 499,77 req/s, enquanto que nos experimentos de simulação, uma média de 481,28 req/s, correspondendo a uma diferença percentual de 3,70%. Os resultados obtidos mostram que o servidor não está saturado, uma vez que é capaz de atender as requisições praticamente na mesma taxa na qual ele é demandado. A Figura 5.12 apresenta graficamente estes resultados.

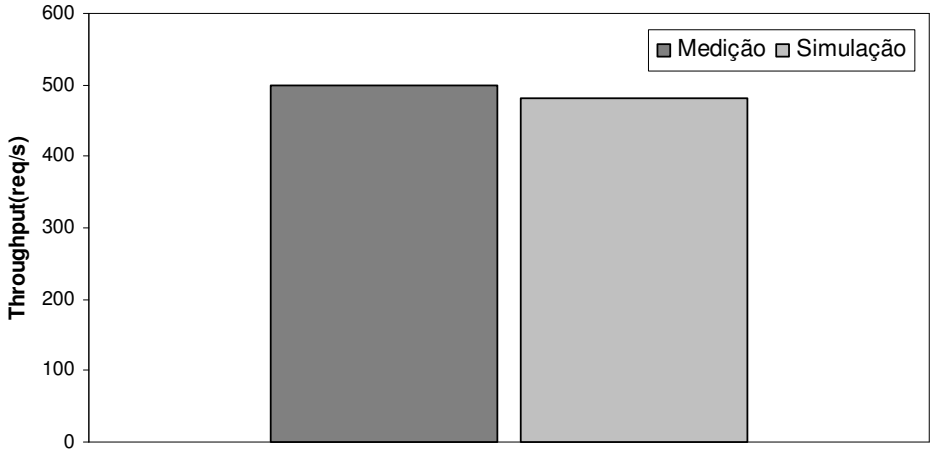


Figura 5.12. Cenário 1 (5 Clientes a 100 Req/s): *Throughput*.

A última métrica considerada é a utilização de CPU. Para monitorar esta utilização no servidor de aplicação foi usada a ferramenta *Performance Monitor*, que permite um acompanhamento dos recursos utilizados por cada processo de maneira individual. Os resultados medidos e simulados para a utilização de CPU são apresentados no gráfico da Figura 5.13.

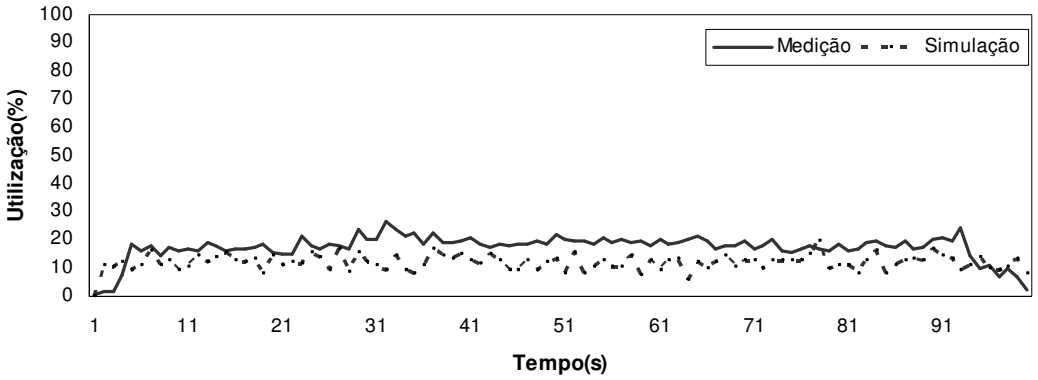


Figura 5.13. Cenário 1 (5 Clientes a 100 Req/s): Utilização de CPU.

Os experimentos de medição mostram que a utilização média da CPU é 17,89%, enquanto que nos experimentos de simulação esta média é de 11,85%. Esses resultados comprovam que a CPU da estação servidora não representa, neste cenário, um gargalo. A partir do gráfico apresentado é possível a identificação clara de uma tendência. Em geral, a utilização da CPU durante os experimentos de medição é ligueiramente maior do que durante os experimentos de simulação. Esta diferença pode ser facilmente entendida quando se observa que o modelo não representa outras atividades constantemente realizadas pelo servidor de aplicação, tais como a coleta automática de lixo. Contudo, a proximidade desses resultados é bastante satisfatória, ratificando, mais uma vez, a validade do modelo.

É importante lembrar que dado o pequeno número de clientes envolvidos nesse cenário, é possível a geração da cadeia de Markov correspondente, a partir da qual as métricas podem ser calculadas, tanto em regime estacionário quanto em regime transiente. Para verificar o comportamento estacionário do sistema nesse cenário, a ferramenta TimeNET foi utilizada na geração e solução da cadeia de Markov. Os resultados calculados indicam um total de 4,73 instâncias criadas, um *throughput* de 481,59 requisições por segundo e uma utilização de CPU de 12,08%. Estes resultados ratificam as simulações e medições realizadas.

Cenário 2: 10 Clientes a 100 req/s

No segundo cenário avaliado, o número de clientes dobra com relação ao cenário anterior, enquanto a taxa de requisições é mantida, de forma que o servidor recebe, potencialmente, 1.000 req/s. O gráfico apresentado na Figura 5.14 representa o número de instâncias criadas durante o processamento destas requisições nos experimentos de medição e simulação.

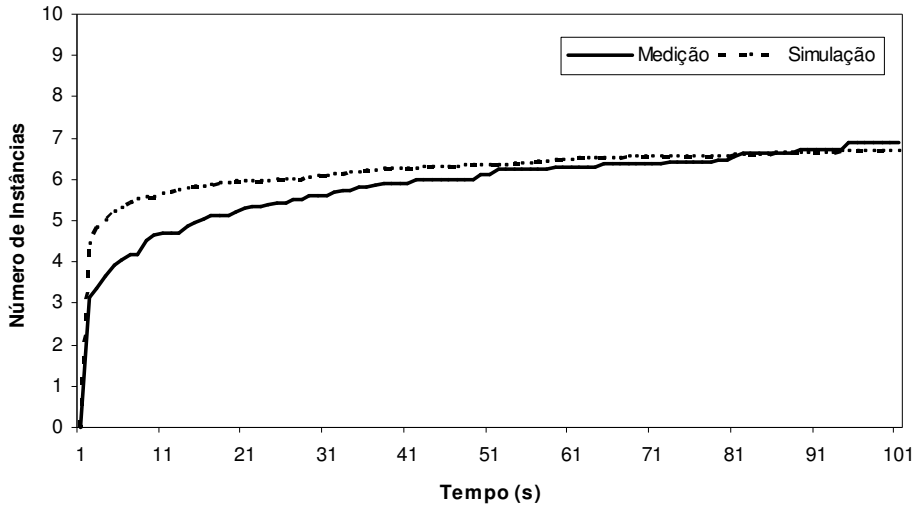


Figura 5.14. Cenário 2 (10 Clientes a 100 Req/s): Número de Instâncias Criadas.

Após os 100 segundos considerados nos experimentos de medição, observou-se que uma média de 6,89 instâncias do componente foram necessárias para processar as requisições recebidas, enquanto que nos experimentos de simulação esta média foi de 6,68 correspondendo a uma diferença de pouco mais de 3%. Analisando-se os experimentos ao longo de toda a sua duração, pode-se observar que após alguns segundos iniciais as curvas de simulação e medição convergem e a diferença percentual entre elas diminui sensivelmente, permanecendo dentro da faixa desejável (de 10% a 30%).

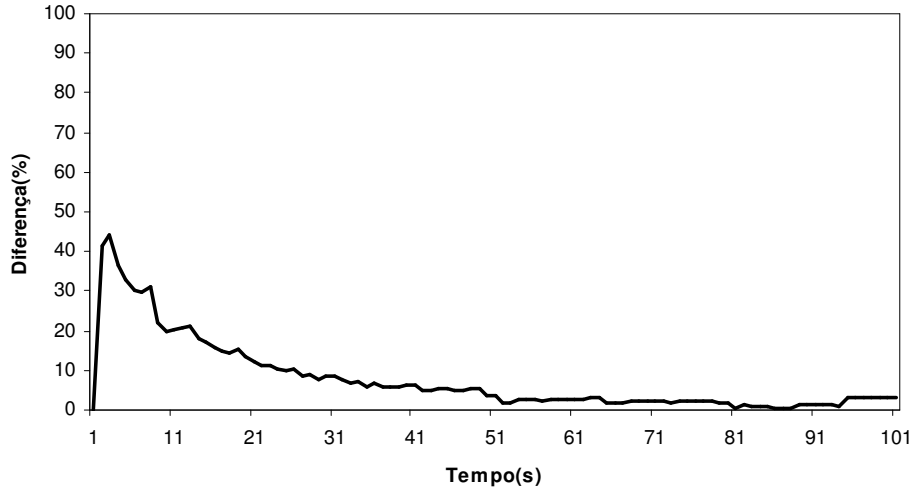


Figura 5.15. Cenário 2 (10 Clientes a 100 Req/s): Diferença entre Simulação e Medição.

Em relação à métrica de *throughput*, os resultados obtidos através dos experimentos de medição apontam para uma taxa de processamento do servidor de 999,85 req/s, enquanto que as simulações indicam uma taxa média de 956,47. A diferença percentual entre estas taxas, as quais são representadas na Figura 5.16, corresponde a 4,34%.

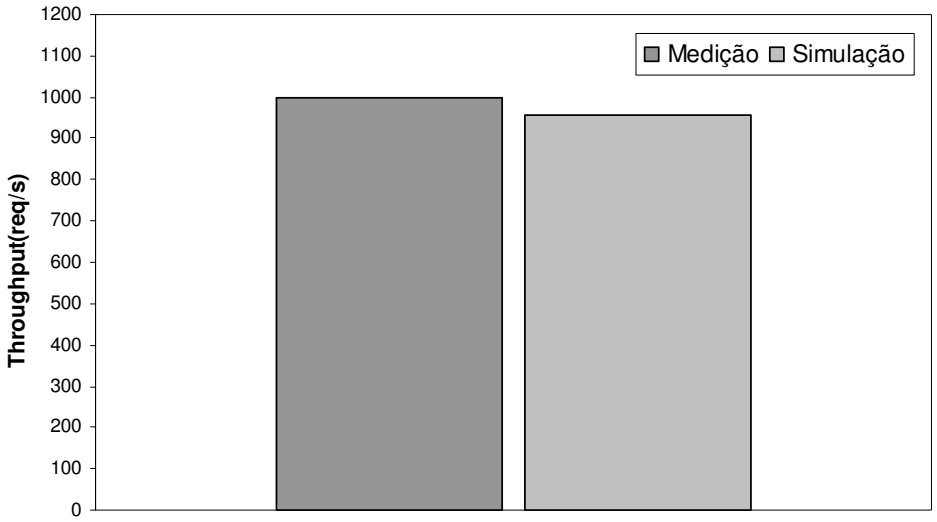


Figura 5.16. Cenário 2 (10 Clientes a 100 Req/s): *Throughput*.

Por fim, a Figura 5.17 apresenta a utilização média de CPU nos experimentos de medição e simulação. Conforme pode ser observado, mesmo com o número de usuários duplicados em relação ao cenário anterior, a CPU ainda não representa um gargalo para o sistema. Novamente constata-se uma utilização real (medição) de CPU ligeiramente superior às projeções realizadas a partir do modelo. Esta diferença, da mesma forma que para o modelo anterior, pode ser explicada pelo fato de o servidor de aplicação realizar atividades não modeladas, uma vez que não dizem respeito ao serviço de *pooling* de instâncias. Em termos médios, as medições indicam uma utilização de 31,97% da CPU, e as simulações indicam 23,67%.

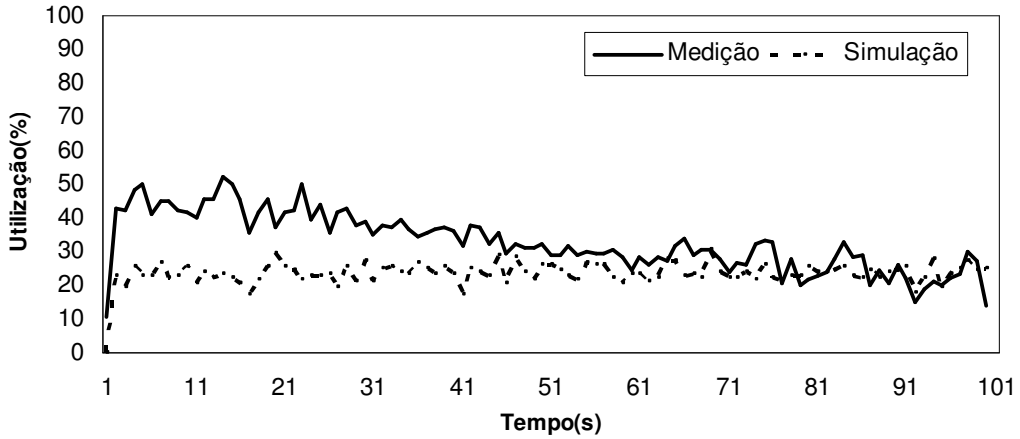


Figura 5.17. Cenário 2 (10 Clientes a 100 Req/s): Utilização de CPU.

Conforme explicado anteriormente, não foi possível a obtenção de resultados analíticos para este número de clientes. Além disso, a ocorrência rara do evento de criação de instâncias dificulta a realização de simulações em regime estacionário. Este problema, contudo, pode ser resolvido removendo-se esta métrica do modelo. Simulações estacionárias foram realizadas para a derivação das demais métricas indicando um *throughput* médio de 956,35 req/s e uma utilização de CPU média de aproximadamente 24%. Estes resultados validam os experimentos anteriores realizados.

Cenário 3: 100 Clientes a 1 req/s

Este cenário representa uma situação mais próxima da realidade de uso de um sistema, representando um maior número de usuários e uma taxa de requisição menor. Neste contexto, uma decisão típica tomada por um administrador responsável pelo gerenciamento do servidor de aplicação no qual o componente está instalado é configurar um *pool* capaz de armazenar 100 instâncias desse componente. Desta forma, busca-se garantir que todos os clientes possam ser simultaneamente atendidos. Contudo, conforme pode ser observado pelos resultados dos experimentos de medição e simulação apresentados na Figura 5.18, um número bem menor de instâncias é suficiente para processar estas requisições de forma simultânea. De fato, os experimentos de simulação apontam para um número médio de 3,14 instâncias, enquanto que os experimentos de medição apontam para 2,54. Além disto, em todos os 35 experimentos de medição realizados o número máximo de instâncias criadas foi de 4, indicando que, para o componente e as características de carga considerados, um *pool* com tamanho de 10 instâncias seria mais do que suficiente. A diferença percentual entre os resultados das medições e simulações realizadas é apresentado na Figura 5.19.

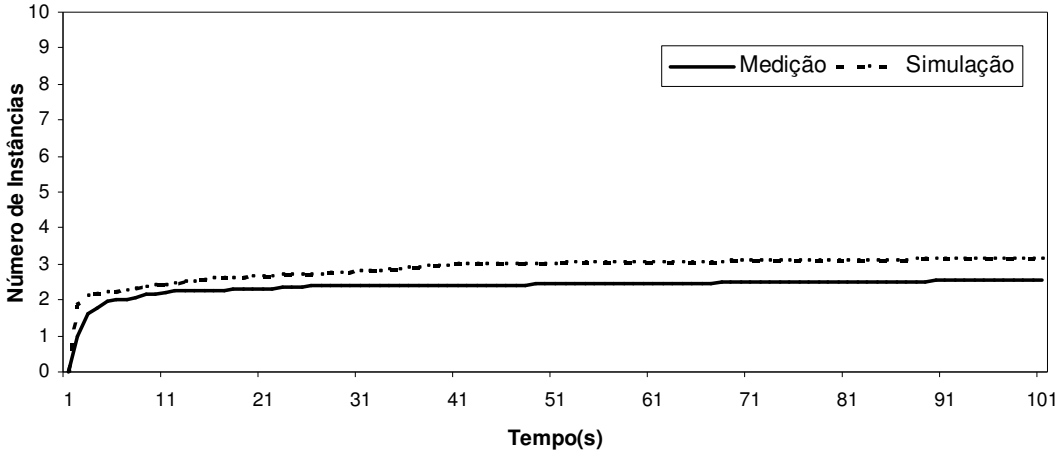


Figura 5.18. Cenário 3 (100 Clientes a 1 Req/s): Número de Instâncias Criadas.

Para entender os resultados obtidos é importante perceber que os clientes, quando avaliados de forma conjunta, representam para o servidor uma taxa total de 100 req/s. Esta taxa acumulada corresponde a 1 (uma) requisição a cada 10 milissegundos e é inclusive menor do que a taxa de requisições recebidas pelo servidor nos dois cenários anteriores. Observando-se cuidadosamente o modelo apresentado e somando-se os tempos das atividades envolvidas no processamento de uma requisição, verifica-se que (desconsiderando-se temporariamente a concorrência pela CPU) uma requisição é processada em menos de 1 milissegundo, mesmo que uma instância do componente tenha que ser criada para processá-la. Desta forma, não é surpresa o baixo número de instâncias criadas pelo servidor de aplicação.

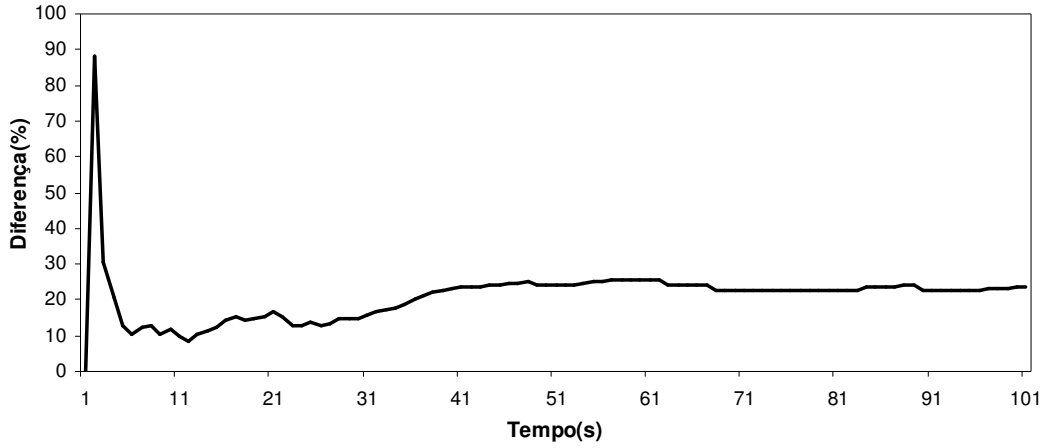


Figura 5.19. Cenário 3 (100 Clientes a 1 Req/s): Diferença entre Simulação e Medição.

Do ponto de vista de *throughput*, os experimentos de medição e simulação apresentam uma diferença bem pequena. Enquanto nos expererimentos de medição foi constatado um *throughput* médio de 99,99 req/s, observou-se nas simulações uma média de 99,97 req/s, correspondendo a uma diferença percentual de 0,02%. Estes resultados são representados no gráfico da Figure 5.20.

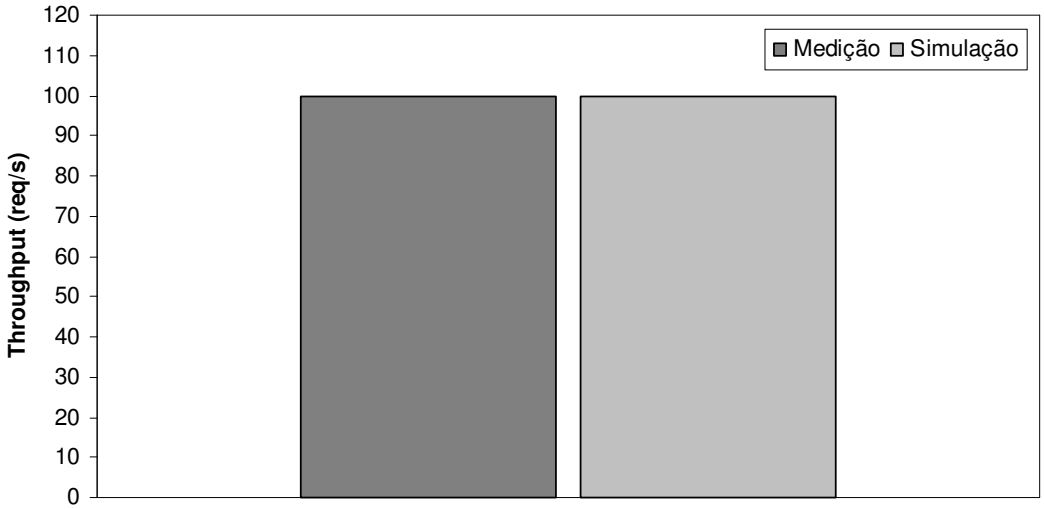


Figure 5.20. Cenário 3 (100 Clientes a 1 Req/s): *Throughput*.

Por fim, devido à reduzida taxa de requisições submetidas ao servidor, a CPU é muito pouco requisitada, conforme pode ser constatado pelo gráfico de utilização de CPU apresentado na Figura 5.21. Neste gráfico verifica-se que as curvas referentes aos experimentos de medição e simulação estão bastante próximas e que indicam uma utilização bastante reduzida do recurso.

Uma análise mais detalhada dos dados obtidos nos experimentos, tanto de simulação quanto de medição, indica que durante praticamente todo o intervalo de tempo considerado o servidor de aplicação utilizou menos de 5% da CPU. Em termos de valores médios, os experimentos de medição apresentaram uma utilização de 3,87% da CPU, enquanto que as simulações resultaram em uma utilização de 2,20%. Verificou-se ainda que durante os experimentos de medição a utilização máxima da CPU foi de 8,90% e que nas

simulações a utilização máxima foi de 7%. Esses resultados demonstram, claramente, que este recurso não está representando um gargalo para o desempenho do sistema.

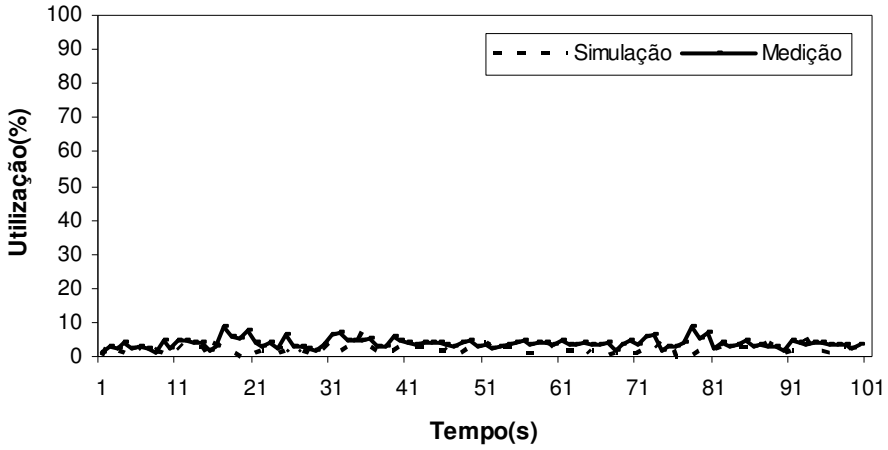


Figura 5.21. Cenário 3 (100 Clientes a 1 Req/s): Utilização de CPU.

Novamente, desconsiderando-se a métrica de número de instâncias foram realizadas simulações estacionárias, as quais indicam um *throuput* médio de 99,97 req/s e uma utilização de 2,50% da CPU.

Uma consideração importante a ser feita em relação ao modelo cliente/servidor proposto diz respeito ao nível de abstração considerado no projeto. Um modelo mais detalhado fornece, potencialmente, mais informações. Contudo, devido ao número de lugares e *tokens* representados há um potencial problema de explosão de estados (uma vez que o número de estados possíveis varia exponencialmente com o número de lugares, ver seção 2.3), o qual pode inviabilizar a utilização de técnicas analíticas na derivação das métricas selecionadas, deixando apenas a alternativa de uso da técnica de simulação.

Para o modelo cliente/servidor, dado o número elevado de lugares, os cenários passíveis de solução por técnicas analíticas estão fortemente limitados em termos do número máximo de clientes, uma vez que cada cliente é modelado através de um *token*. Uma possível solução para a limitação apresentada envolve um aumento no nível de abstração utilizado no projeto, restringindo, assim, o número de lugares, e, por conseguinte, o número de estados da cadeia de Markov associada. Neste sentido, um novo modelo, chamado de cliente/servidor sintético foi projetado. É importante destacar que, para viabilizar este novo modelo, duas simplificações foram feitas. Primeiro, apenas o modo de operação não estrito foi considerado. Desta forma o número de instâncias criadas para processar as requisições não é limitado, tendo, o servidor, a flexibilidade de ajustá-lo de acordo com a taxa de requisições recebidas. Segundo, apenas soluções estacionárias podem ser corretamente calculadas através do modelo.

5.5.5 Modelo Cliente/Servidor Sintético

O modelo cliente/servidor sintético, apresentado na Figura 5.22, foi projetado para viabilizar soluções analíticas para um número elevado de clientes. Para tanto, é essencial que ele seja limitado e que o número de lugares representados seja pequeno, uma vez que o número de estados da cadeia de

Markov associada cresce exponencialmente com esta variável. É importante lembrar, ainda, que o número de *tokens* representados no modelo também possui influência no número de estados gerados, devendo, portanto, ser minimizado (ver Seção 2.3).

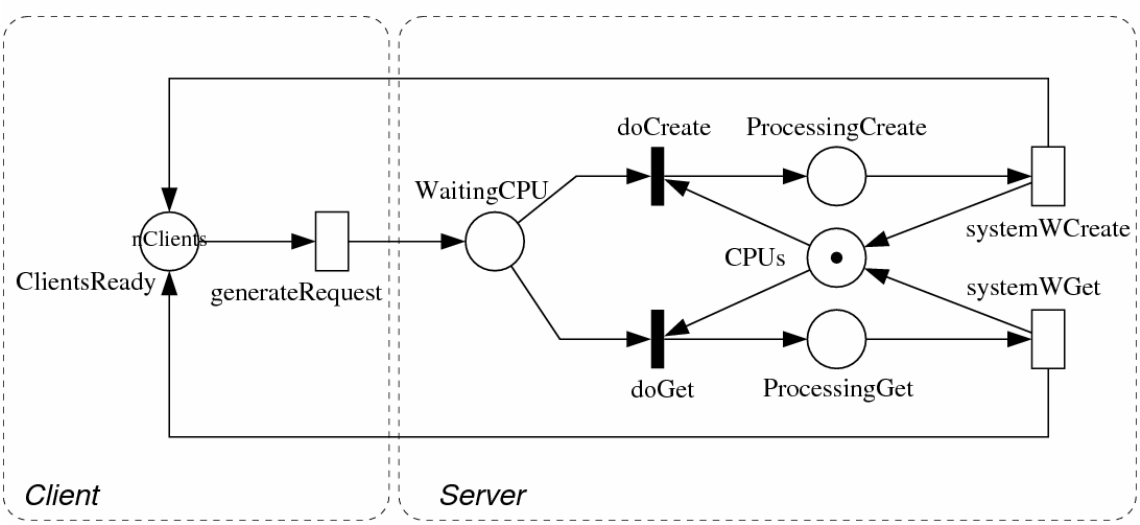


Figura 5.22. Modelo Cliente/Servidor Sintético

O modelo apresentado possui apenas 5 lugares e é limitado, pois todos eles são cobertos por invariantes (conforme pode ser visto no Quadro 5.3). A partir desses invariantes, pode-se constatar que o modelo possui, em qualquer instante, um número máximo de *nClients* + 1 *tokens*, onde *nClients* representa o número de clientes. Este número reduzido de lugares e *tokens* viabiliza, efetivamente, soluções analíticas, mesmo para um número bastante elevado de clientes. De fato, cenários envolvendo 10.000 clientes foram representados e solucionados em segundos.

Quadro 5.3. Modelo Cliente/Servidor Sintético: Invariantes de Lugar.

```
The net contains 2 P-invariants.
ClientsReady + WaitingCPU + ProcessingGet + ProcessingCreate =
nClients

ProcessingGet + ProcessingCreate + CPU = 1

All places are covered by p-invariants.
```

O modelo cliente/servidor sintético possui dois componentes: *Client* e *Server*.

O componente *Client* é composto por um único lugar, *ClientsReady*, representando os clientes prontos para realizarem requisições, e por uma única transição, *generateRequest*, cuja taxa representa o número de requisições geradas por cada cliente na unidade de tempo. É importante destacar que essa transição é exponencial, de forma que as requisições são geradas segundo uma distribuição de *Poisson*, e que a sua semântica de temporização é *infinite-server*, permitindo a representação de clientes independentes e concorrentes.

O componente *Server* representa uma junção dos componentes *Network* e *JBoss Application Server* presentes no modelo cliente/servidor e é composto pelas transições temporizadas *systemWCreate* e *systemWGet*, pelas transições

imediatas *doCreate* e *doGet* e pelos lugares *WaitingCPU*, *CPUs*, *ProcessingCreate* e *ProcessingGet*, além dos arcos de ligação.

A transição *systemWGet* representa, de forma conjunta, todas as atividades realizadas pela rede e pelo servidor de aplicação no processamento de uma requisição, considerando que uma instância possa ser obtida do *pool* e devolvida a este. Desta forma, esta transição representa todo um fluxo de atividades presente no modelo cliente/servidor exibido na Figura 5.9. Este fluxo compreende as transições *sendRequest*, *inChain*, *log*, *getPooled*, *invokeNext*, *clear*, *put*, *outChain* e *sendResponse*. Desta forma, o tempo médio associado à transição *systemWGet* é dado pela soma dos tempos das transições que compõe o fluxo mencionado, correspondendo a 335,093 microssegundos.

A transição *systemWCreate* representa o conjunto de atividades envolvidas no processamento de uma requisição quando é necessária a criação de uma nova instância do componente. Este conjunto compreende as transições *sendRequest*, *inChain*, *log*, *create*, *invokeNext*, *clear*, *outChain* e *sendResponse*. O tempo associado a esta transição é dado pela soma dos tempos das atividades que o compõem, totalizando 518,257 microssegundos. A Tabela 5.13 sintetiza as características das transições temporizadas representadas no modelo.

Tabela 5.13. Modelo Cliente/Servidor Sintético: Transições Temporizadas

Transição	Tempo Médio (μs)	Semântica de Disparo
systemWCreate	335,093	Infinite-Server
systemWGet	518,257	Infinite-Server

As requisições geradas pelos clientes são representadas por *tokens* armazenados inicialmente no lugar *WaitingCPU*. Para que estas requisições possam ser processadas é necessária a alocação de uma CPU, representada por um *token* contido no lugar *CPUs*. Quando uma CPU estiver disponível, as duas transições imediadas *doGet* e *doCreate* estarão habilitadas e o conflito resultante é resolvido através do peso associado a cada uma.

Para atribuir estes pesos de forma correta deve-se avaliar em que situações as transições *systemWGet* e *systemWCreate* devem ser disparadas. Em regime estacionário, a transição *systemWCreate* deve ser disparada somente quando a taxa de geração de requisições é superior à taxa de processamento e o número de clientes submetendo requisições é maior do que a capacidade do *pool*. Em qualquer outra situação, as requisições geradas serão processadas utilizando-se uma instância obtida do *pool*, em termos de modelo isto significa que a transição *systemWGet* deve ser disparada. O mapeamento destas condições no peso das transições é realizado através do uso de uma sentença condicional.

A Tabela 5.14 resume as características das transições imediatas. Nesta tabela *nClients* representa o número de clientes, *dRequest* representa o intervalo entre requisições dado em microssegundos, *dSystemWGet* representa o *delay* da transição *systemWGet*, ou seja, o tempo necessário ao processamento de uma requisição considerando que uma instância já está disponível e *poolSize* representa o tamanho configurado para o *pool* de instâncias do componente.

A partir destas definições, pode-se perceber que a razão $nClients/d\,\mathrm{Re}quest$ representa a taxa de geração de requisições, enquanto que $1/dSystemWGet$ representa a taxa de processamento. Logo, a sentença condicional representa que, se a taxa de requisição for maior do que a de processamento e o número de clientes for maior que o tamanho do *pool*, a taxa de disparos da transição *doGet* será proporcional *poolSize*, enquanto que a da transição *doCreate* será proporcional a $nClients - poolSize$. Em qualquer outra situação, o peso de *doGet* será bastante elevado, enquanto que o de *doCreate* será praticamente zero, indicando que as requisições serão processadas por instâncias já disponíveis.

Tabela 5.14. Modelo Cliente/Servidor Sintético: Transições Imediatas

Transição	Peso	Prioridade	Função de Habilitação
doGet	IF(nClients/dRequest>=(1/dSystemWGet))AND (nClients>poolSize):(poolSize) ELSE 100000000000000;	1	-
doCreate	IF(nClients/dRequest>=(1/dSystemWGet))AND (nClients>poolSize):nClients-poolSize ELSE 0.00000000000001;	1	-

O processamento de uma requisição tem início após a alocação de uma CPU. Neste momento, um *token* representando a requisição é gerado em um dos lugares, *ProcessingCreate* ou *ProcessingGet* (de acordo com a transição imediata disparada), onde permanece até que o processamento da requisição seja finalizado, o que é representado pelo disparo da transição exponencial correspondente. Quando esse processamento é finalizado, um retorno é devolvido ao cliente sob forma de um novo *token* depositado no lugar *ClientsReady*, indicando que ele está apto a gerar uma nova requisição.

Por fim, é importante ressaltar que, embora a semântica de disparo das transições *systemWGet* e *systemWCreate* seja *infinite-server*, como uma única CPU é considerada nos experimentos realizados, esta semântica poderia ser mudada para *single-server* sem nenhum impacto nos resultados.

5.5.5.1 Verificação e Validação

A cobertura de todos os lugares do modelo cliente/servidor sintético por invariantes garante que ele é estruturalmente limitado. Associando-se esta característica à ausência de *deadlocks* e ao reduzido número de lugares e *tokens* representados, tem-se um modelo que pode ser completamente avaliado através de técnicas puramente analíticas, permitindo a obtenção de soluções matemáticas precisas.

Desta forma, o processo de validação do modelo cliente/servidor sintético compreende a obtenção de soluções analíticas (através da solução da cadeia de Markov) para as métricas e cenários propostos na validação do modelo cliente/servidor e a comparação dos resultados assim obtidos com os resultados das medições e simulações apresentados anteriormente. É importante destacar que, visando reduzir as dimensões do modelo, nem o *pool*, nem as instâncias são representados explicitamente no modelo. De fato, a representação do *pool* foi resumida a um parâmetro, chamado *poolSize*, o qual representa o

tamanho configurado para o *pool*. Desta forma, o modelo sintético assume que o *pool* já está devidamente preenchido, representando, portanto, apenas a operação do sistema em regime estacionário. Neste cenário, o acompanhamento da métrica relativa ao número de instâncias criadas não faz sentido. O Quadro 5.4 apresenta as fórmulas utilizadas no cálculo das métricas a partir do modelo.

Quadro 5.4. Modelo Cliente/Servidor Sintético: Métricas

```
CPUUtilization=(1-E{#CPU})*100
ClientsThroughput=E{#Ready}*(1/dRequest)*1000000
ServerThroughput=(E{#ProcessingCreate}*(1/dSystemWCreate)+
                  E{#ProcessingGet}*(1/dSystemWGet))* 1000000
```

Dentre as métricas apresentadas, duas estão relacionadas ao *throughput*. *ClientsThroghput* representando a taxa de requisições geradas em conjunto pelos clientes e *ServerThroughput* representando a taxa de requisições processadas pelo servidor. Como o modelo é síncrono, uma vez que representa o acionamento de um *Stateless SessionBean*, estas taxas são iguais. Desta forma, utilizar-se-á insdistintamente o termo *throughput* para referenciar ambos os pontos de vista (do cliente ou do servidor).

A Tabela 5.15 apresenta uma comparação dos resultados obtidos a partir das medições, da análise estacionária do modelo cliente/servidor sintético, e das simulações estacionárias realizadas no modelo cliente/servidor completo.

Tabela 5.15. Resultados: Medição X Simulação X Análise

Clientes	Taxa Nominal (req/s)	Medição/Modelo	*Throughput* (req/s)	Utilização de CPU (%)
5	100	Medição	499,77	17,89
5	100	Completo	481,59	12,08
5	100	Sintético	481,56	16,14
10	100	Medição	999,85	31,97
10	100	Completo	956,35	23,98
10	100	Sintético	955,69	32,02
100	1	Medição	99,99	3,87
100	1	Completo	99,97	2,50
100	1	Sintético	99,97	3,35

Conforme pode ser observado, há uma pequena diferença entre os resultados obtidos para utilização de CPU com os modelos completo e sintético. Esta diferença é resultante da granulosidade das atividades representadas em cada um. Enquanto no modelo completo a CPU é alocada separadamente em cada epata do processamento de uma requisição, no modelo sintético esta alocação acontece para o processamento como um todo. Apesar das pequenas diferenças mencionadas, os resultados apresentados validam o modelo cliente/servidor sintético.

5.6 Realização de Experimentos

A concepção do modelo cliente/servidor sintético possibilita a representação de diversos cenários e a sua avaliação através de técnicas analíticas. De fato, as dimensões deste modelo viabilizam a geração e solução da cadeia de Markov correspondente utilizando os recursos computacionais atualmente disponíveis.

Dentre os diversos cenários de interesse, são de particular relevância aqueles relativos às predições de desempenho dadas diferentes condições de carga. Para investigar o comportamento do sistema diante deste tipo de cenário planejou-se uma série de experimentos envolvendo uma variação no número de clientes entre 100 e 5.000. Para cada cliente, o intervalo de tempo decorrido entre a recepção de uma resposta e o envio de uma nova requisição (referenciado como *think time*) é representado através de uma distribuição exponencial com um tempo médio de 1 segundo. A utilização de um intervalo entre requisições de natureza exponencial gera uma taxa de requisições seguindo uma distribuição de *Poisson*, a qual é particularmente apropriada em cenários envolvendo um grande número de fontes independentes [Jain 1991]. Este mesmo tipo de distribuição é utilizado em *benchmarks* como TPC-W.

Os experimentos mencionados foram repetidos considerando-se diferentes tamanhos de pool de instâncias, permitindo, assim, uma investigação do efeito de variações desta variável nas diferentes métricas de desempenho consideradas. Os resultados obtidos são apresentados na Figura 5.24 e na Figura 5.25.

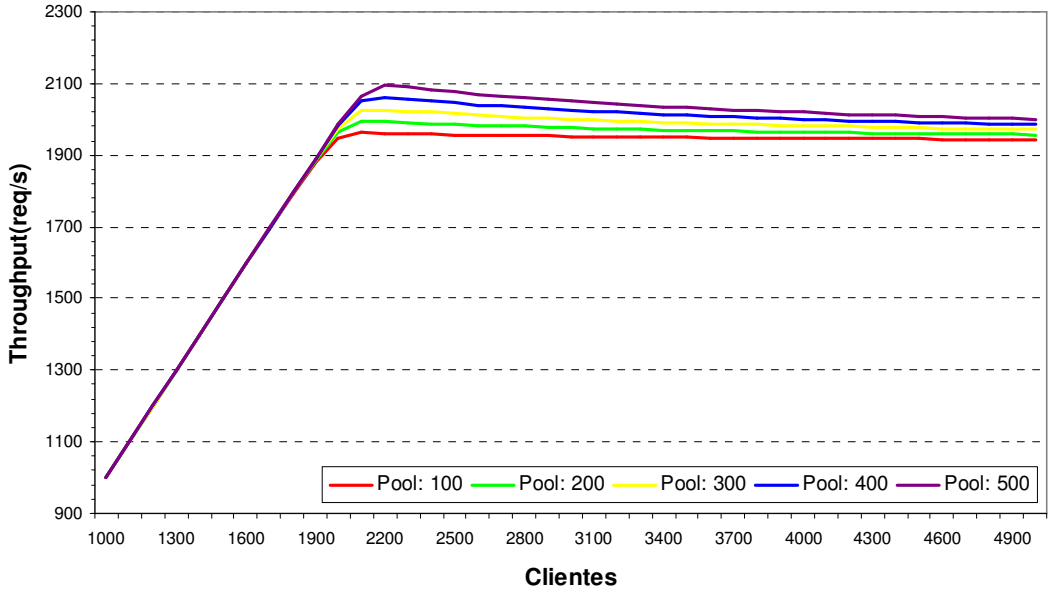


Figura 5.23. Modelo Cliente/Servidor Sintético: Throughput Versus Número de Clientes para Diferentes Tamanhos de Pool.

Conforme pode ser observado na Figura 5.23, a taxa de requisições processadas pelo sistema (*throughput*) aumenta, inicialmente, de forma linear com o número de clientes independentemente do tamanho do *pool* configurado. Um aumento contínuo da carga provoca uma saturação do servidor, levando à uma diminuição no aumento do *throughput* que deixa de crescer de forma linear. O ponto de saturação varia levemente com o tamanho do *pool* mas, em

geral, acontece próximo a 2.000 clientes. Com o sistema já saturado, um aumento no número de clientes faz com que o sistema atinja a sua capacidade útil, determinada pelo ponto de máximo da curva de *throughput*. Para um *pool* com tamanho de 100 instâncias a capacidade útil é de aproximadamente 1.962 requisições por segundo, enquanto que para um *pool* de 500 esta capacidade é de 2.096. A diferença entre estes valores é de 134 requisições por segundo, indicando que um aumento de 400% no tamanho do *pool* representa um incremento de *throughput* de 6,83%.

É importante observar o impacto de um aumento no tamanho do *pool* no consumo de memória por parte do sistema. Um *pool* mantém referências para um conjunto de instâncias de um componente que estão "ativas" na memória. Uma vez que estas instâncias estão referenciadas, elas não são consideradas pelos mecanismos de coleta automática de lixo, representando, portanto, um incremento na memória necessária à execução do sistema. Diante do exposto, uma configuração de *pools* de grande tamanho pode não ser adequada.

O gráfico da Figura 5.23 mostra, ainda, que, após atingir a sua capacidade útil, o sistema apresenta um decréscimo no *throughput*, indicando a ocorrência do fenômeno conhecido como *thrashing* (ver Seção 2.2.1).

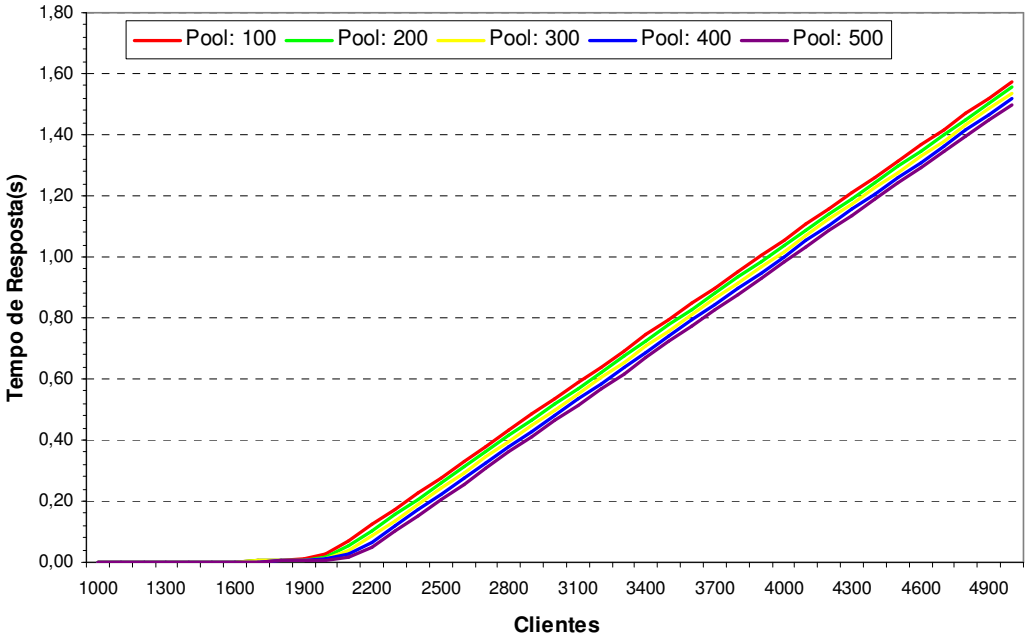


Figura 5.24. Modelo Cliente/Servidor: Tempo de Resposta Versus Número de Clientes para Diferentes Tamanhos de Pool.

Do ponto de vista de tempo de resposta, o impacto provocado pelo aumento no número de clientes pode ser observado na Figura 5.24. Percebe-se que os tempos obtidos para cenários nos quais o sistema não está saturado são bastante próximos de zero. Com a saturação, os tempos crescem rapidamente e o impacto representado por alterações nas configurações de tamanho de *pool* não é significativo.

É importante observar a coerência entre os gráficos de *throughput* e de tempo de resposta, ambos indicando uma saturação do sistema por volta de 2.000 usuários simultâneos. Esta saturação é resultante de uma escassez de recursos no sistema, indicando claramente a presença de gargalos.

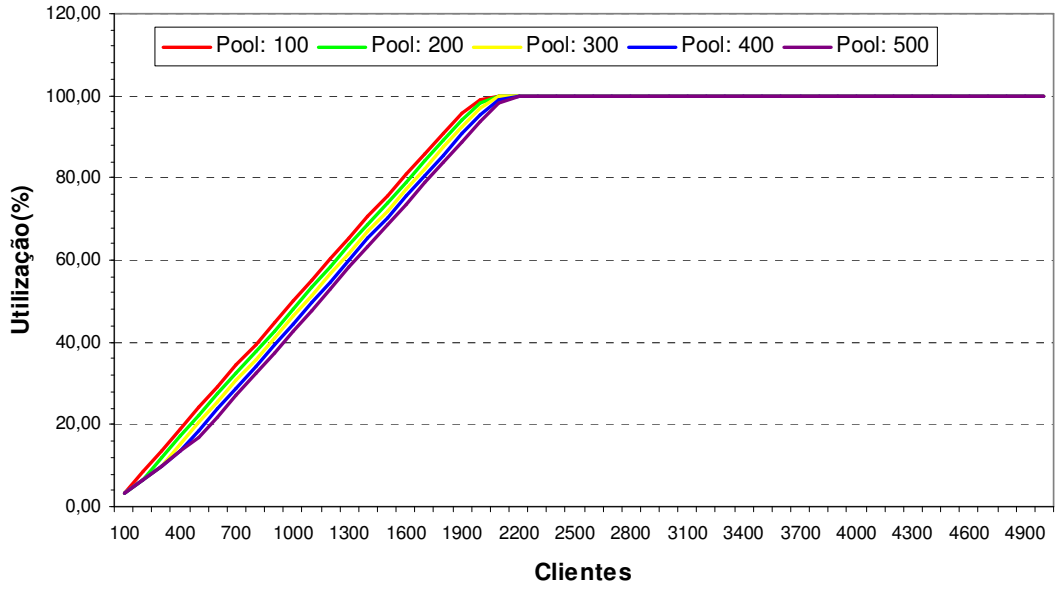


Figura 5.25. Modelo Cliente/Servidor: Utilização de CPU Versus Número de Clientes para Diferentes Tamanhos de Pool.

Uma análise da utilização da CPU representada no modelo explica a saturação detectada. Conforme pode ser observado na Figura 5.25, os aumentos sucessivos na carga provocam uma utilização de 100% da CPU, tornando este recurso um gargalo para o sistema.

Uma análise dos resultados obtidos indica que o crescimento acentuado do tempo de resposta após a saturação do sistema pode comprometer a qualidade esperada do serviço. Conforme mencionado na Seção 2.2.1, nestas situações é comum a configuração dos mecanismos de controle de admissão. Tais mecanismos controlam a concorrência interna por recursos do servidor, permitindo uma limitação no tempo de resposta. O modelo apresentado na Figura 5.26 inclui um destes mecanismos, o qual é responsável por rejeitar novas requisições quando o sistema já está processando um número configurável delas.

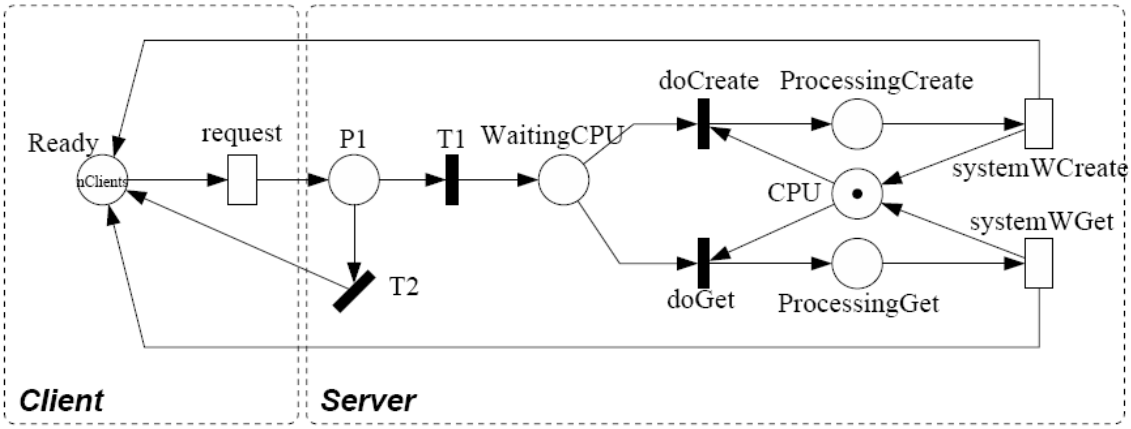


Figura 5.26. Modelo Cliente/Servidor Sintético com Mecanismo de Controle de Admissão.

Neste modelo, as transições imeditas *T1* e *T2* são responsáveis pelo aceite e pela rejeição de requisições, respectivamente. A concorrência entre

estas transições é resolvida através de condições de habilitação, as quais checam o número de requisições correntemente no sistema (calculado através do número de *tokens* nos lugares *WaitingCPU*, *ProcessingCreate* e *ProcessingGet*) e habilitam ou desabilitam as mesmas de acordo com um limite estabelecido.

A inclusão do mecanismo de controle de admissão potencialmente reduz a disponibilidade do sistema, garantindo, contudo, um tempo de resposta máximo e controlado. Figura 5.27, Figura 5.28, Figura 5.29 e Figura 5.30 apresentam os resultados obtidos a partir da solução deste modelo.

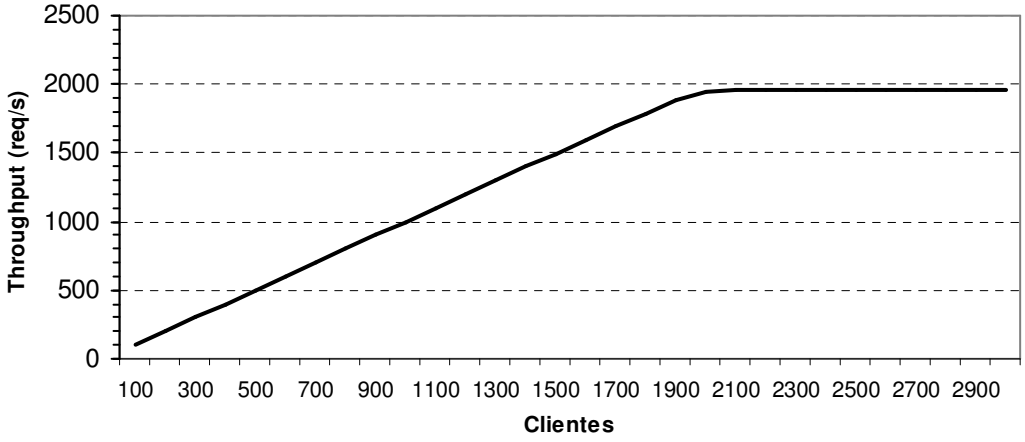


Figura 5.27. Modelo Cliente/Servidor Sintético com Controle de Admissão: Throughput.

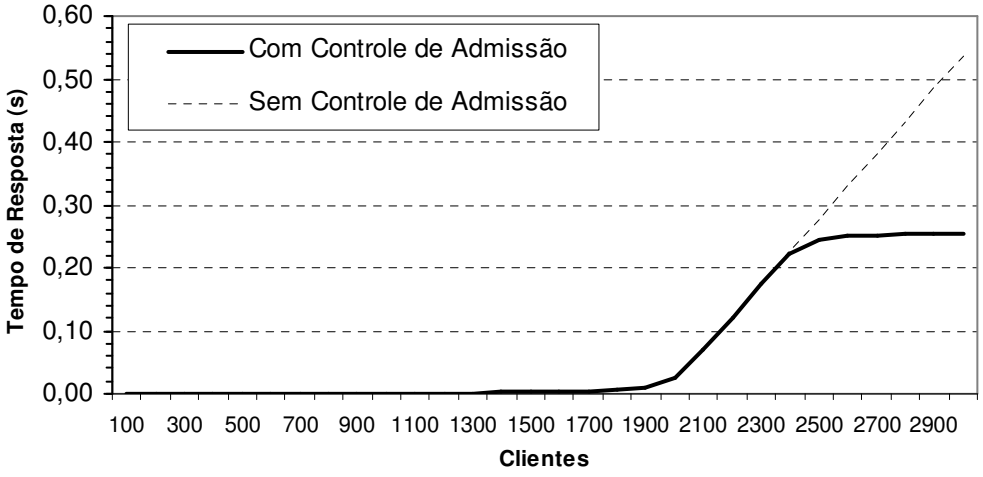


Figura 5.28. Modelo Cliente/Servidor Sintético com Controle de Admissão: Tempo de Resposta.

Pode-se constatar que o mecanismo de controle de admissão não afeta, de forma significativa, as métricas de *throughput* e utilização de CPU. Tal mecanismo, contudo, reduz a concorrência por recursos internos ao servidor e, como conseqüência, o tempo de fila associado a estes. Conforme apresentado na Seção 2.2.1, o tempo de fila corresponde a uma porção significativa do tempo de resposta, de forma que sua redução implica em uma redução correspondente no tempo de resposta externamente observável. A Figura 5.28 mostra esta redução que pode ser observada a partir da saturação do sistema e é crescente com o aumento no número de clientes a partir deste ponto.

A contrapartida à limitação estabelecida no tempo de resposta é a redução observada na disponibilidade do sistema. A ausência de um mecanismo de controle de admissão faz com que novas requisições sejam sempre aceitas implicando em uma disponibilidade de 100%. No cenário considerado, o mecanismo de controle de admissão reduz a concorrência interna no sistema através de rejeição de requisições quando o sistema está saturado. Desta forma, embora a utilização deste tipo de mecanismo provoque uma redução no tempo de resposta do sistema, há uma redução na disponibilidade deste, a qual pode ser observada na Figura 5.30.

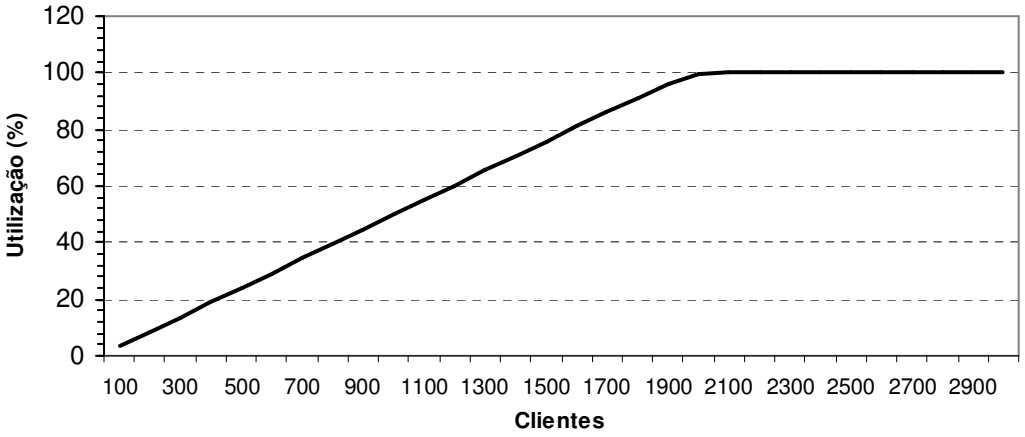


Figura 5.29. Modelo Cliente/Servidor Sintético com Controle de Admissão: Utilização de CPU.

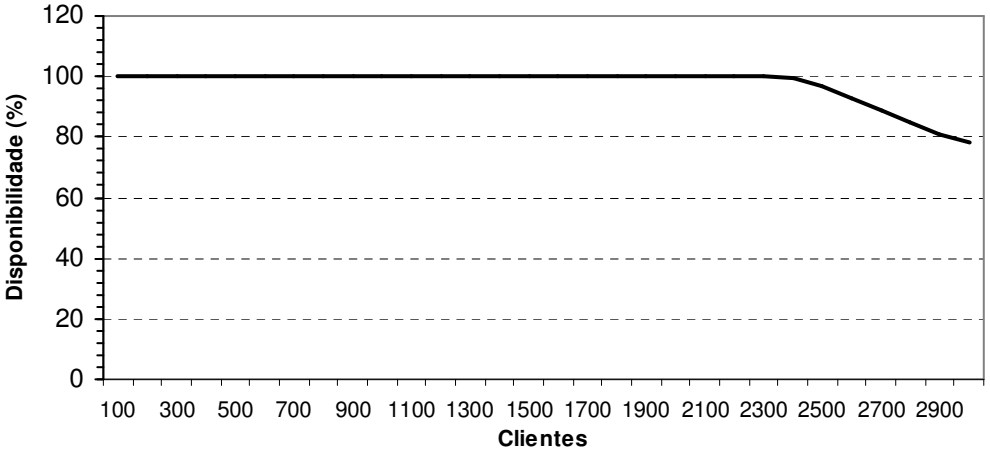


Figura 5.30. Modelo Cliente/Servidor Sintético com Controle de Admissão: Disponibilidade.

Por fim, é importante fazer algumas considerações concernentes aos resultados obtidos ao longo dos cenários apresentados. Dado os pequenos ganhos em termos de *throughput* e tempo de resposta provocados por aumentos no tamanho do *pool*, pode parecer que o impacto do mecanismo de *pooling* é pouco relevante. Contudo, é importante mencionar que a dimensão do impacto observado depende do "custo" de criação e inicialização das instâncias do componente. É, também, fundamental lembrar que, conforme mencionado anteriormente, o modelo cliente/servidor representa apenas o modo de operação não estrito do *pool* (ver Seção 2.1.3). Neste modo de operação, não há restrição ao número de instâncias criadas, de forma que o *pool* não representa efetivamente um gargalo.

5.7 Considerações Finais

Neste capítulo a abordagem para avaliação de desempenho proposta no capítulo anterior foi utilizada no desenvolvimento e validação de modelos para o servidor de aplicação JBoss e, em particular, para o suporte fornecido por este servidor à componentes desenvolvidos utilizando-se a tecnologia de *Stateless SessionBeans*.

Os modelos projetados foram centrados no serviço de *pooling* de instâncias, o qual foi selecionado para avaliação devido ao seu impacto direto no desempenho do sistema. Para validar os modelos foram selecionadas as métricas de número de instâncias, *throughput* e utilização de CPU, as quais, quando consideradas em conjunto, fornecem uma ampla visão da situação operacional do servidor.

Ao longo do capítulo, foram apresentados diferentes modelos, os quais diferem entre si em termos de nível de abstração e de capacidade de representação.

CAPÍTULO

6

Conclusão e Trabalhos Futuros

"A problem well stated is a problem half solved"

Charles F. Kettering.

Este capítulo apresenta as principais contribuições do presente trabalho e seus aspectos mais relevantes. As limitações dos modelos desenvolvidos e os trabalhos futuros a serem realizados dentro desta linha de pesquisa também são destacados.

6.1 Conclusão

O presente trabalho de mestrado propõe uma abordagem para avaliação de desempenho de servidores de aplicação baseada na utilização de modelos formais em redes de Petri estocásticas, considerando, em particular, os modelos GSPN e DSPN. A utilização de tais formalismos visa viabilizar a obtenção de resultados de desempenho através das técnicas de análise e de simulação, permitindo, ainda, avaliações de propriedades qualitativas à cerda dos modelos projetados.

Utilizando a abordagem, o presente trabalho propõe um conjunto de modelos de desempenho para o servidor de aplicação JBoss, o qual foi escolhido considerando-se a sua projeção no mercado e a disponibilidade de seu código fonte.

O primeiro modelo proposto [Souza et al. 2006a], denominado modelo servidor, apresenta uma visão interna do servidor de aplicação, representando diretamente apenas os componentes relacionados ao suporte do serviço de *pooling* de instâncias. Este modelo foi apresentado em sua versão original e

posteriormente em sua versão refinada, a qual incorpora elementos responsáveis pela representação da variabilidade dos dados medidos. O modelo servidor permite o acompanhamento do processo de criação de instâncias de um componente EJB e auxilia o dimensionamento e o gerenciamente do *pool* de instâncias mantido pelo servidor de aplicação.

A despeito de sua relevância, o modelo servidor possui algumas deficiências que limitam a sua pontencial utilização. Dentre elas, destaca-se a ausência de limitação estrutural que inviabiliza a utilização de técnicas analíticas na avaliação das métricas de desempenho, ficando esta avaliação restrita ao uso da técnica de simulação. Outra limitação importante diz respeito à ausência de representação dos aspectos relativos à concorrência por recursos de *hardware* e *software*, de forma que os tempos relacionados às disputas por tais recursos não aparecem explicitamente. Desta forma, a obtenção de resultados precisos através do modelo servidor está vinculada à realização de uma parametrização deste modelo em condições reais de carga. Por fim, o modelo servidor, como o próprio nome indica, limita-se à representação dos elementos internos ao servidor de aplicação, não sendo capaz de fornecer métricas relativas ponto de vista dos clientes.

As limitações do modelo servidor levaram à proposição de um novo modelo, referenciado como modelo cliente/servidor [Souza et al. 2006b]. Este modelo é estruturalmente limitado e, como o próprio nome indica, representa cliente e servidor. Outra característica importante é a modelagem explícita da CPU, permitindo que o modelo represente a disputa por este recurso e originando os tempos de fila. Esta representação permite que o modelo seja parametrizado utilizando dados obtidos em condições diferentes das condições normais de operação. A proposição do modelo cliente/servidor amplia o conjunto de métricas que podem ser avaliadas, incluindo *throughput*, número de instâncias criadas e percentual de utilização de CPU no servidor.

Apesar de representar uma nítida evolução em relação ao modelo servidor, o modelo cliente/servidor possui uma importante limitação. De fato, embora o modelo cliente/servidor seja estruturalmente limitado, ele apresenta um nível de detalhe grande, o que restringe a capacidade de utilização das técnicas analíticas. Desta forma, apenas cenários com um número reduzido de clientes podem ser avaliados utilizando tais técnicas.

As limitações provocadas pelo baixo nível de abstração do modelo cliente/servidor levaram ao projeto de um modelo alto nível, referenciado como modelo cliente/servidor sintético, cujas dimensões reduzidas viabilizam a utilização das técnicas de análise e simulação na solução de um amplo universo de cenários, envolvendo, possivelmente, um número elevado de clientes. Este modelo viabiliza uma avaliação mais ampla dos aspectos relacionados ao desempenho, uma vez que permite a incorporação outras métricas além daquelas representadas no modelo cliente/servidor, destacando-se o tempo de resposta. De fato, considerando-se variedade de cenários possíveis e as métricas disponíveis, o modelo cliente/servidor sintético pode ser utilizado em uma diversidade de projetos relativos à avaliação de desempenho compreendendo, entre outros, planejamento de capacidade, testes de estresse e identificação de gargalos.

Ainda no contexto deste trabalho, foi desenvolvida uma extensão para o modelo cliente/servidor sintético que incorpora um mecanismo de controle de admissão. Este tipo de mecanismo é comumente configurado em servidores de aplicação para limitar a quantidade de requisições em atendimento, visando, assim, garantir a qualidade no processamento das requisições que estão sendo correntemente atendidas. As requisições excedentes são rejeitadas pelo servidor e devem ser submetida novamente a posteriori. A inclusão deste mecanismo afeta a métrica referente à disponibilidade do servidor, a qual pode ser avaliada diretamente do modelo.

6.2 Contribuições

A principal contribuição deste trabalho é propor uma abordagem para a avaliação de desempenho de servidores de aplicação baseada na utilização de redes de Petri estocásticas. Como estudo de caso da abordagem proposta, foram desenvolvidos e validados modelos de desempenho para o servidor de aplicação JBoss. Estes modelos também são considerados uma importante contribuição, uma vez que, além de fornecerem resultados relativos ao desempenho, permitem a verificação de propriedades qualitativas do sistema modelado, tais como a ausência de *deadlocks*. É importante ressaltar que, até onde temos conhecimento, não existem outros modelos em redes de Petri para esse servidor.

Atualmente, a medição é a técnica mais utilizada em projetos de avaliação de desempenho de servidores de aplicação. Contudo, esta técnica possui importantes limitações, dentre as quais destacam-se o custo de manutenção e gerenciamento de um ambiente próprio e semelhante ao de produção, e a alta sensibilidade à variações neste ambiente.

Diante deste cenário, possuir modelos precisos e capazes de fornecer, através da realização de experimentos de simulação ou da utilização de técnicas analíticas, um conjunto de informações que permitam uma ampla visão do estado atual de operação do servidor, assim como a realização de predições de desempenho, é, claramente, uma importante vantagem.

6.3 Limitações

A despeito da validade e relevância da abordagem e dos modelos propostos, é importante destacar algumas limitações do presente trabalho. Primeiramente, deve-se destacar que embora a abordagem proposta seja suficientemente genérica, os modelos projetados estão intimamente vinculados ao servidor de aplicação JBoss e não podem ser facilmente adaptados para refletir outros servidores de aplicação.

Os modelos estão centrados no suporte fornecido pelo servidor JBoss a *Stateless Session Beans* e não devem ser utilizados em predições de desempenho em cenários envolvendo outros tipos de componentes. Aspectos relacionados à arquitetura das aplicações também não estão diretamente representados.

Por fim, importantes serviços fornecidos por servidores de aplicação, tais como transação e segurança não foram explicitamente modelados, não sendo possível a sua avaliação.

6.4 Trabalhos Futuros

Trabalhos futuros devem considerar o desenvolvimento de modelos que contemplem outros tipos de componente tais como *entity beans* e *message-driven beans*. Importantes serviços, tais como concorrência, transação e segurança também devem ser estudados e avaliados. Outro aspectos importante que pode ser considerado diz respeito aos diferentes mecanismos de comunicação e de *clustering* usualmente fornecidos pelos servidores de aplicação.

Outra questão a ser insvestigada corresponde à validação de utilização da abordagem proposta na avaliação de servidores de aplicação que não são de código aberto. Esta validação envolve novos desafios uma vez que a ausência do código fonte dificulta a obtenção dos tempos referentes à realização de atividades que não sejam visíveis externamente, exigindo, assim, a construção de modelos mais abstratos.

Por fim, uma relevante linha de pesquisa a ser explorada corresponde à incorporação de fatores relacionados à adaptação nos modelos em redes Petri que representam os servidores de aplicação. As pesquisas sobre *middleware* adaptativo correspondem ao estado da arte na área de sistemas distribuídos.

Referências

[Bagrodia e Shen 1991] Bagrodia, R. e Shen, C. (1991) "MIDAS: Integrated Design and Performance Evaluation of Distributed Systems", IEEE Transactions on Software Engineering, Vol. 7(10), pp. 1042-1058.

[Bernstein 1996] Bernstein, P. (1996) "Middleware: A Model for Distributed System Services", Communications of the ACM, Vol.39(2), pp. 87-98.

[Campbell et al. 1999] Campbell, A., Couson, G., Kounavis, M. (1999) "Managing Complexity: Middleware Explained", IT Professional, IEEE Computer Society, Vol.1(5), pp. 22-28.

[Cecchet et al. 2002] Cecchet, E., Marguerite, J., Zwaenepoel, W. (2002) "Performance and Scalability of EJB Applications", Proc. 17th Conference on Object-Oriented Programming, Systems, Languages and Applications, Seattle, WA.

[Chen 2002] Chen, S., Gorton, I., Liu, A., Liu, Y. (2002) "Performance Prediction of COTS Component-based Enterprise Applications", Proc. 5th ICSE Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly, Orlando, Flórida, USA.

[CSIRO 2002] CSIRO (2002) "Middleware Technology Evaluation Series: Evaluating J2EE Application Servers v.2.1".

[Desrochers e Al-Jaar 1995] Desrochers, A. e Al-Jaar, R. (1995) "Applications of Petri Nets in Manufacturing Systems: Modeling, Control, and Performance Analysis", IEEE Press, ISBN: 0-87942-295-5.

[Fleury e Reverbel 2003] Fleury, M. e Reverbel, F. (2003) "The JBoss Extensible Server", Proc. International Middleware Conference.

[Gartner 2005] Gartner Group (2005) "Magic Quadrant for Enterprise Application Servers, 2Q05".

[Gorton e Liu 2003] Gorton, I. e Liu, A. (2003) "Evaluating the Performance of EJB Components", IEEE Internet Computing Vol.7 (3), pp18-23.

[Gorton et al. 2003] Gorton, I., Liu, A., Brebner, P., (2003) "Rigorous Evaluation of COTS Middleware Technology", IEEE Computer, Vol.36(3), pp. 50-55.

[Jain 1991] Jain, R., (1991) "The Art of Computer Systems Performance Evaluation", Wiley Computer Publishing, ISBN: 0-471-50336-3.

[JBoss 2004] JBoss Inc. (2004) "JBoss Admin Development Guide, JBoss v3.2.7".

[Kounev e Buchmann 2003] Kounev, S. e Buchmann, A. (2003) "Performance Modeling and Evaluation of Large-Scale J2EE Applications". Proc. 29th International Conference of the Computer Measurement Group (CMG) on Resource Management and Performance Evaluation of Enterprise Computing Systems (CMG-2003), pp.7-12.

[Kounev et al. 2004] Kounev, S., Weis, B. e Buchmann, A. (2004) "Performance Tuning and Optimization of J2EE Applications on the JBoss Platform", Journal of Computer Resource Management, Issue 113.

[Liu e Gorton 2004] Liu, Y. e Gorton, I. (2004) "Accuracy of Performance Prediction for EJB Applications: A Statistical analysis, in Software Engineering for Middleware (SEM'04), Linz, Austria, Springer-Verlag.

[Liu et al. 2002a] Liu, Y., Gorton, I., Liu, A., Jiang, N., Chen, S. (2002) "Designing a Test Suite for Empirically based Middleware Performance Predication", Proc. of The 4th Int. Conf. on Technology of Object Oriented Languages and Systems Pacific (TOOLS2002), Sydney, Australia.

[Liu et al. 2002b] Liu, Y., Gorton, I., Liu, A., Chen, S., (2002) "Evaluating the Scalability of Enterprise JavaBeans Technology", APSEC 2002, pp. 74-83.

[Liu et al. 2004] Liu, Y., Fekete, A., Gorton, I. (2004) "Predicting the Performance of Middleware-based Applications at the Design Level", Proc. 4th Int. Workshop on Software and Performance.

[Liu et al. 2004] Liu, Y., Fekete, A., Gorton, I. (2004) "Predicting the Performance of Middleware-based Applications at the Design Level", Proc. of the 4th International Workshop on Software and Performance, Redwood Shores, California.

[Lladó 2001] Lladó, C. (2001) PhD Thesis: "Performance Evaluation of Enterprise JavaBeans Architectures",

Department of Computing, Imperial College of Science, Technology and Medicine, University of London.

[Lladó e Harrison 2000] Lladó, C. e Harrison, P. (2000) "Performance Evaluation of an Enterprise JavaBeans Server Implementation", Proc. 2nd International Workshop on Software and Performance, Ottawa, Canada, pp.17-20.

[Lladó et al. 2002] Lladó, C., Lüthi, J., Harrison, P. (2002) "Studying Sensitivities of an EJB Performance Model", Proc. of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'02).

[Marsan e Chiola 1987] Marsan, M. e Chiola, G., (1987) "On Petri Nets with Deterministic and Exponentially Distributed Firing Times", LNCS 266, Springer-Verlag, pp. 132-145.

[Marsan et al. 1984] Marsan, M., Balbo, G., Conte, G. (1984) "A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems", ACM Transactions on Computer Systems, Vol.2(2), pp.93-122.

[Marsan et al. 1994] Marsan, M., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G. (1995) "Modelling with Generalised Stochastic Petri Nets", John Wiley & Sons, ISBN: 0-471-93059-8.

[McGuinness e Murphy 2005] McGuinness, D., e Murphy, L. (2005) "A Simulation Model of a Multi-server EJB system, A-MOST'05, St. Louis, Missouri, USA.

[McGuinness et al. 2004] McGuinness, D., Murphy, L., Lee, A. (2004) "Issues in Developing a Simulation Model of an EJB System, Computer Measurement Group 2004 International Conference (CMG 2004), Las Vegas, Nevada.

[Menascé e Almeida 2003] Menascé, D. e Almeida, V. (2003) "Planejamento de Capacidade para Serviços na Web", Editora Campus Ltda, ISBN: 85-352-1102-0.

[Menascé et al. 2004] Menascé, D., Almeida, V., Dowdy, L. (2003) "Performance by Design: Computer Capacity Planning by Example", Prentice Hall, ISBN: 0-13-090673-5.

[Molloy 1982] Molloy, M. (1982) "Performance Analysis Using Stochastic Petri Nets", Transactions on Computers, Vol.31(9), pp. 913-917.

[Natkin 1980] Natkin, S. (1980) "Reseaux de Petri Stochastiques". Ph.D. dissertation, CNAM-PARIS.

[Neuts 1975] Neuts, M. (1975) “Probability distributions of phase type.” In: Liber Amicorum Professor Emeritus H. Florin, University of Louvain, Belgium, pp. 173-206.

[OMG 2002] OMG (2002) “Common Object Request Broker Architecture: Core Specification v3.0”.

[Ran et al. 2001] Ran, S., Brebner,P., Gorton, I., (2001) “The Rigorous Evaluation of Enterprise Java Bean Technology”, In 15th International Conference on Information Networking (ICOIN-15), Beppu City, Japan.

[Souza et al. 2006a] Souza, F., Arteiro, R., Rosa, N., Maciel, P. (2006) “Using Stochastic Petri Nets for Performance Modelling of Application Servers”, Proc. 5th International Workshop on Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEO-PDS) on 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS), Rhodes Island, Greece.

[Souza et al. 2006b] Souza, F., Arteiro, R., Rosa, N., Maciel, P. (2006) “Using Stochastic Petri Nets for Performance Modelling of JBoss Application Server”, V Workshop de Desempenho de Sistemas Computacionais e de Comunicação (WPerformance), Campo Grande, Mato Grosso do Sul.

[Sun 2003a] Sun Microsystems Inc. (2003a) “Enterprise JavaBeans™ Specification v2.1”.

[Sun 2003b] Sun Microsystems Inc. (2003b) “Java™ 2 Platform, Enterprise Edition Specification v1.4”.

[Sun 2004] Sun Microsystems Inc. (2004) “Java™ Remote Method Invocation Specification v1.5.0”.

[Sun 2005] Sun Microsystems Inc. (2005) “The J2EE™ 1.4 Tutorial”.

[Symons 1980] Symons, F. (1980) “Introduction to Numerical Petri Nets, a General Graphical Model of Concurrent Processing Systems”, Australian Telecommunications Research, Vol.14, pp. 28-33.

[TPC 2005] Transaction Processing Performance Council (2005) “TPC Benchmark App (Application Server) Specification v1.1.1”

[Vinoski 2003] Vinoski, S. (2003) "The Performance Presumption", IEEE Internet Computing, Vol.07(2), pp. 88-90.

[Watson e Desrochers 1991] Watson III, J. e Desrochers, A. (1991) “Applying Generalized Stochastic Petri Nets to Manufacturing Systems Containing Nonexponential Transition

Functions", IEEE Transactions on Systems, MAN, and Cybernetics, Vol.21(5), pp. 1008-1017.

[Zimmermann 2001] Zimmermann, A. (2001) "TimeNET: A Software Tool for the Performability Evaluation with Stochastic Petri Nets", Performance Evaluation Group, TU Berlin.

APÊNDICE

A

Redes de Petri

Este apêndice apresenta uma representação mais formal para os conceitos relacionados às redes de Petri, os quais foram brevemente introduzidos na Seção 2.3. As redes de Petri correspodem a uma família de formalismos compreendendo desde modelos não temporizados até os modelos estocásticos. Desta forma, as primeiras definições apresentadas neste apêndice são fundamentadas nos modelos originais de redes de Petri, desconsiderando-se os aspectos temporais. Ao longo do texto são apresentadas as principais extensões propostas com o intuito de ampliar o poder de representação deste formalismo, culminando com a introdução do conceito de tempo através da definição dos modelos estocásticos GSPN e DSPN.

A.1 Definição

Um modelo em redes de Petri pode ser formalmente descrito da seguinte forma:

Definição 1: Um modelo em rede de Petri é uma tupla:

$$\mu = (P, T, I, O, H, M_0)$$

onde:

P é um conjunto de lugares;

T é um conjunto de transições, $P \cap T = \phi$;

$I(p,t)$: $P \times T \rightarrow \aleph$ é uma função que mapeia um par (p,t) em um número natural correspondente à multiplicidade do arco direcionado do lugar p para a transição t. Caso não exista tal arco, o valor assumido pela função é *0*. Para a transição t, um lugar p tal que $I(p,t)>0$ é referenciado como lugar de entrada e o conjunto destes lugares é dado por ${}^{\bullet}t = \{p \in P : I(p,t) > 0\}$;

$O(t,p)$: $T \times P \rightarrow \aleph$ é uma função que mapeia um par (t,p) em um número natural correspondente à multiplicidade do arco direcionado da transição t ao lugar p. Caso não exista este arco, o valor assumido pela função é *0*.

Para a transição t, um lugar p tal que $O(t,p)>0$ é referenciado como lugar de saída e o conjunto destes lugares é dado por $t^{\bullet} = \{p \in P : O(t,p) > 0\}$;

$H(p,t)$: $P \times T \rightarrow \aleph$ é uma função que mapeia um par (p,t) em um número natural correspondente à multiplicidade do arco inibidor que liga o lugar p à transição t. Caso não exista este arco, o valor assumido pela função é *0*. Para a transição t, um lugar p tal que $H(p,t)>0$ é referenciado como lugar inibidor e o conjunto destes lugares é dado por ${}^{\circ}t = \{p \in P : H(p,t) > 0\}$; e

M_0: $P \rightarrow \aleph$ é uma função que define a marcação inicial associando a cada lugar um número natural.

As funções I, O e H podem, ainda, ser vistas como funções que associam transições a multi-conjuntos de lugares, onde o número de repetições de um lugar em um multi-conjuntos indica a multiplicidade do arco de entrada, de saída ou inibidor correspondentes. Formalmente diz-se que:

$$I(t) : T \rightarrow Bag(P);$$

$$O(t) : T \rightarrow Bag(P); \text{ e}$$

$$H(t) : T \rightarrow Bag(P).$$

A.2 Regras de Habilitação e Disparo

Em uma rede de Petri, uma transição somente é considerada habilitada a disparar se: (1) cada lugar de entrada contiver um número de *tokens* maior ou igual a um limite estabelecido pela multiplicidade do arco correspondente, (2) cada lugar inibidor (lugar conectado à transição através de um arco inibidor) contiver um número de *tokens* menor do que o limite estabelecido multiplicidade do arco inibidor correspondente. Uma descrição formal para a condição de habilitação de uma transição é apresentada abaixo:

Definição 2: Uma transição t é dita habilitada em uma marcação M se e somente se:

- $\forall p \in {}^{\bullet}t, M(p) \geq I(p,t)$ e
- $\forall p \in {}^{\circ}t, M(p) < H(p,t)$.

O disparo de uma transição remove *tokens* de cada lugar $p \in {}^{\bullet}t$ em número igual à multiplicidade do arco de entrada correspondente, e cria *tokens* em cada lugar de saída $p \in t^{\bullet}$ em número igual à multiplicidade do arco de saída ligando p a t. Em termos formais:

Definição 3: O disparo de uma transição t habilitada em uma marcação M produz uma marcação M' de tal forma que:

$$M' = M + O(t) - I(t)$$

Diz-se então que M' é diretamente alcançável a partir de M. Este fato é normalmente denotado por $M|t\rangle M'$.

A.3 Conjunto e Grafo de Alcançabilidade

Partindo da marcação inicial, é possível determinar todas as marcações alcançáveis pela rede através dos disparos de transições habilitadas. O conjunto destas marcações é chamado de conjunto de alcançabilidade.

Definição 4: O Conjunto de Alcançabilidade (RS) de uma rede de Petri com marcação inicial M_0 é descrito por $RS(M_0)$ e é definido como o menor conjunto de marcações tal que:

- $M_0 \in RS(M_0)$, e
- $M_1 \in RS(M_0) \wedge \exists t \in T : M_1 | t \rangle M_2 \Rightarrow M_2 \in RS(M_0)$.

Definição 5: O Grafo de Alcançabilidade de uma rede de Petri com marcação inicial M_0 e conjunto de alcançabilidade $RS(M_0)$ é um multi-grafo rotulado e dirigido onde o conjunto de nós é RS e o conjunto de arcos A é dado por:

- $A \subseteq RS \times RS \times T$,
- $\langle M_i, M_j, t \rangle \in A \Leftrightarrow M_i | t \rangle M_j$.

Para o modelo em redes de Petri representado na Figura 2.7, o Conjunto de Alcançabilidade RS é dado por $RS = \{(1,0),(0,1)\}$ e o Grafo de Alcançabilidade é representado na Figura A.6.1.

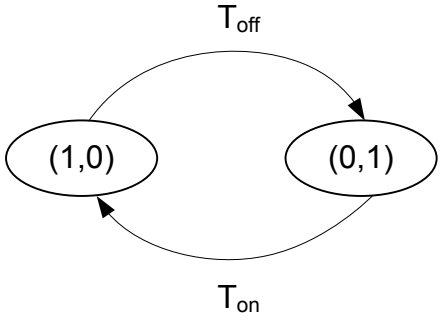


Figura A.6.1. Grafo de Alcançabilidade para Chave *On-Off*.

O Grafo de Alcançabilidade é a chave para um estudo completo do comportamento de um sistema modelado em redes de Petri. Este grafo, além de representar todas as marcações atingíveis, ou seja, todos os estados pelos quais o sistema pode passar, representa ainda todas as possíveis transições entre eles e os eventos relacionados. Contudo, a despeito da importância, sua construção pode não ser possível, uma vez que Conjunto de Alcançabilidade de uma rede de Petri pode ser infinito.

A.4 Conflito, Concorrência e Confusão

Uma situação comum na modelagem de sistemas em redes de Petri é a ocorrência de conflitos. Um conflito corresponde a uma escolha e ocorre em um modelo quando mais de uma transição está simultaneamente habilitada e o disparo de uma delas interfere com a possibilidade de disparos futuros das demais. Uma definição mais formal para esta situação depende da definição de grau de habilitação, a qual é apresentada a seguir.

Definição 6: Para uma rede de Petri, o grau de habilitação é uma função $ED: T \times [P \to \aleph] \to \aleph$, de forma que $\forall t \in T, \forall M: P \to \aleph, ED(t,M) = k$, se e somente se:

- $\forall p \in {}^{\bullet}t, M(p) \geq k \times I(p,t)$,
- $\forall p \in t^{\circ}, M(p) < H(p,t)$,
- $\exists p \in {}^{\bullet}t: M(p) < (k+1) \times I(p,t)$.

Desta forma, um conflito é uma situação na qual o disparo de uma transição diminue o grau de habilitação de outra, podendo, inclusive, desabilitá-la. Uma transição t_1 está em conflito efetivo com uma transição t_2 em uma marcação M, denotado por $t_1 EC(M) t_2$, quando ambas as transições estão habilitadas em M, mas o grau de habilitação da transição t_2 na marcação M', produzida pelo disparo de t_1, é menor do que seu grau de habilitação em M.

Definição 7: Para um modelo em rede de Petri, $\forall t_1, t_2 \in T | t_1 \neq t_2, \forall M: P \to \aleph$, a transição t_1 está em conflito efetivo com a transição t_2 (ou seja, $t_1 EC(M) t_2$), se e somente se:

- $M | t_1 \rangle M'$, e
- $ED(t_2, M) < ED(t_2, M')$.

É importante ressaltar que um conflito efetivo entre a transição t_1 e a transição t_2 pode ser assimétrico, se t_2 não estiver em conflito com t_1, ou simétrico, caso contrário.

Por outro lado, a concorrência entre duas ou mais transições é caracterizada pelo paralelismo das atividades representadas. Transições t_1 e t_2 são concorrentes em uma marcação M se ambas estão habilitadas em M, mas não estão em conflito.

Definição 8: Para um modelo em redes de Petri, as transições t_1 e t_2 são concorrentes em uma marcação M, se e somente se:

$$t_1, t_2 \in E(M) \Rightarrow not(t_1 EC(M) t_2) \wedge not(t_2 EC(M) t_1) .$$

A combinação de concorrência e conflito gera uma situação conhecida como confusão, a qual é ilustrada através de um exemplo na Figura A.6.2. No modelo apresentado na figura, embora as transições t_1 e t_3 sejam concorrentes, de forma que o disparo de uma não diminui o grau de habilitação da outra, a ordem dos disparos é relevante. Se t_1 disparar primeiro, as transições t_2 e t_3 estarão em conflito. Por outro lado, se t_3 disparar primeiro, o conflito não ocorrerá. Em geral, este tipo de situação deve ser evitado, pois pode provocar um comportamento indesejado do modelo.

Uma solução possível é determinar, em tempo de projeto, qual das duas transições deve disparar primeiro. Surge, então, a necessidade de uma nova extensão das redes de Petri, a inclusão do conceito de prioridade entre as transições.

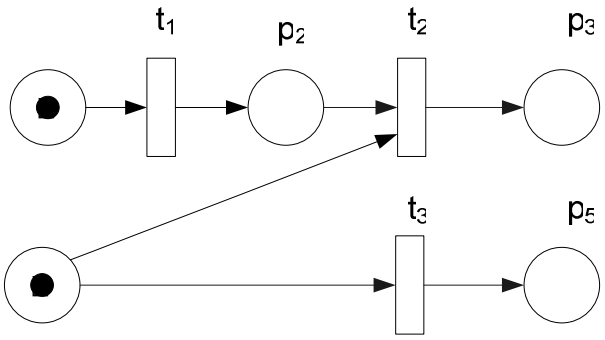


Figura A.6.2. Exemplo de Confusão.

As prioridades correspondem a números naturais associados à transições, permitindo o particionamento destas em classes, de forma que, a cada momento, somente a classe com a maior prioridade associada pode conter transições habilitadas. Mais precisamente, em uma rede de Petri com prioridades uma transição está habilitada quando: (1) seus lugares de entrada tiverem um número de *tokens* maior ou igual a um limite estabelecido pela multiplicidade do arco correspondente; (2) seus lugares inibidores tiverem um número de *tokens* menor do que o limite estabelecido pela multiplicidade dos arcos inibidores correspondentes; e (3) não houver nenhuma transição com maior prioridade que atenda às duas condições anteriores.

Definição 9: De maneira formal, uma rede de Petri prioridades pode ser representada por:

$$\mu = (P, T, I, O, H, \pi, M_0)$$

onde,

P, T, I, O,H e M_0 possuem as mesmas definições anteriores,

$\pi : T \rightarrow \aleph$ é uma função associando a cada transição um número natural representando a sua prioridade.

Na seqüência, são apresentadas as principais propriedades relativas às redes de Petri e os métodos disponíveis para a sua verificação.

A.5 Propriedades

O comportamento de sistemas modelados em redes de Petri pode ser analisado com base em um conjunto de propriedades qualitativas, cuja semântica está vinculada à natureza dos sistemas em si.

Algumas propriedades podem ser verificadas diretamente a partir da estrutura da rede, sem a necessidade de nenhuma informação a cerca do estado inicial do sistema. Estas propriedades recebem o nome de propriedades estruturais e são válidas para todas as marcações. Por outro lado, muitas propriedades somente podem ser analisadas com base em um estado inicial. Estas propriedades são referenciadas como propriedades comportamentais e, em geral, fornecem mais informações sobre o sistema do que as estruturais, uma vez que, além da estrutura, consideram a marcação inicial. O problema fundamental com relação às propriedades comportamentais é a dificuldade de seu esbelecimento. Normalmente, a análise destas propriedades requer a construção do grafo de alcançabilidade, o que pode ser inviável considerando

que o número de estados teóricos possíveis pode ser infinito ou mesmo muito grande, não havendo recursos computacionais suficientes para a sua construção (explosão de estados). Outro fator limitante para o uso das propriedades comportamentais de um sistema é que elas têm que ser reavaliadas para cada marcação inicial.

Na seqüência, são descritas as principais propriedades qualitativas consideradas e as principais técnicas disponíveis para a sua avaliação.

Alcançabilidade e Reversibilidade: A propriedade de alcançabilidade está relacionada à possibilidade do sistema atingir um determinado estado (especificado através de uma marcação). De forma mais precisa, uma marcação M' é dita alcançável se existir uma seqüência de transições $\sigma_n = t_1,t_2,...,t_n$, tal que $M|\sigma_n\rangle M'$. Uma propriedade importante relacionada à alcançabilidade é a reversibilidade. Um modelo em rede de Petri é dito reversível quando a sua marcação inicial M_0 puder ser alcançada a partir de qualquer outra marcação alcançável, ou seja, $\forall M \in RS(M_0), M_0 \in RS(M)$. Em outras palavras, um modelo reversível pode sempre retornar ao seu estado inicial.

Em muitas situações não se está interessado no retorno do sistema ao seu estado inicial, mas sim na possibilidade permanente de retorno a um outro estado qualquer representado através de uma marcação *M*. Uma marcação que pode ser sempre atingida a partir de qualquer outra é chamada de *home state*.

Ausência de *deadlock*: Um modelo em redes de Petri possui *deadlock* quando ele pode atingir uma marcação na qual nenhuma transição está habilitada. Desta forma, um modelo livre de *deadlock* possui sempre uma ou mais transições habilitadas.

***Liveness*:** Uma transição *t* é dita viva (*live*) se, para toda marcação *M* atingível a partir de M_0, existe uma marcação M', atingível a partir de *M*, tal que $t \in E(M')$, onde $E(M')$ é o conjunto de todas as transições habilitadas em M'. Transições que não apresentam esta característica são ditas mortas. Um modelo em rede de Petri é dito vivo se todas as suas transições forem vivas.

Por definição, qualquer modelo vivo não possui *deadlock*. De fato, se um modelo possui pelo menos uma transição viva a ausência de *deadlock* já é verificada. É importante observar, contudo, que a ausência de *deadlock* é condição necessária, mas não suficiente para a verificação da propriedade *liveness* em um modelo. Modelos sem *deadlock*, mas com transições mortas são ditos parcialmente vivos.

Limitação: Um lugar *p* de um modelo em rede de Petri é dito *k*-limitado quando, em qualquer marcação alcançável, o número máximo de *tokens* acumulados em *p* não ultrapassa *k*. Por outro lado, o modelo como um todo é dito *k*-limitado quando todos os seus lugares forem *k*-limitados.

É importante observar que, para um modelo com *N* lugares e *k*-limitado, o número máximo de estados possíveis é dado por $(k + 1)^N$. Desta forma, a limitação de um modelo é condição suficiente para a possibilidade teórica de construção do seu grafo de alcançabilidade. Contudo, dependendo dos valores de *N* e *k*, esta construção pode não ser possível devido à limitação dos recursos computacionais.

Exclusão Mútua: Dois lugares *p* e *q* são mutuamente exclusivos em um modelo em rede de Petri, se e somente se, nenhuma marcação atingível pelo modelo pode conter *tokens* em *p* e *q*, simultaneamente. Por sua vez, duas transições são ditas mutuamente exclusivas se elas nunca estão habilitadas simultaneamente em uma mesma marcação.

As propriedades apresentadas são genéricas, de forma que sua interpretação depende fortemente da natureza do sistema modelado. Dois tipos de técnicas estão disponíveis para a verificação das propriedades de um modelo em redes de Petri: técnicas estruturais e técnicas baseadas no espaço de estados.

A.6 Técnicas Estruturais

As técnicas estruturais consideram somente os componentes estáticos das redes de Petri: lugares, transições e arcos. Neste trabalho, as técnicas estruturais apresentadas são baseadas em álgebra linear. Em tais técnicas, os modelos são representados através de matrizes e as propriedades são verificadas através de equações matriciais.

Definição 10: Dada uma rede de Petri com *m* lugares e *n* transições, a matriz de incidência associada é uma matriz $m \times n$, na qual cada entrada $C(p,t)$ é dada por:

$$C(p,t) = O(t,p) - I(t,p)$$

Semanticamente, cada entrada $C(p,t)$ representa a variação da quantidade de *tokens* no lugar *p* resultante do disparo da transição *t*. Desta forma, cada coluna da matriz *C*, denotada por $C(.,t)$, representa a modificação induzida pelo disparo de *t* no número de *tokens* em todos os lugares da rede. Por outro lado, cada linha desta matriz, denotada $C(p,.)$, representa a variação no número de *tokens* no lugar *p* resultante do disparo de cada uma das transições.

Considerando *I* e *O* como matrizes de inteiros, a matriz de incidência pode ser representada por:

$$C = O^T - I^T$$

onde, I^T e O^T representam as matrizes transpostas de *I* e *O*, respectivamente.

A partir da matriz de incidência, a dinâmica do sistema provocada pelo disparo de uma transição *t* pode ser representada através da equação matricial:

$$M' = M + C(\cdot,t)^T$$

De fato, esta equação pode ser generalizada para uma seqüência de disparos $\sigma_n = t_1, t_2, ..., t_n$, através da equação:

$$M'' = M + [CV_\sigma]^T$$

onde, V_σ é um vetor coluna possuindo $|T|$ componentes, cada um indicando o número de vezes que a transição correspondente foi disparada nesta seqüência. Deve-se ressaltar que as marcações obtidas a partir desta equação não são necessariamente alcançáveis. Isto ocorre porque a habilitação da seqüência de transições não é verificada. Em outras palavras, a equação

apresentada acima, referida como equação de estados, não pode ser utilizada para garantir a propriedade de alcançabilidade de uma marcação, mas somente, para garantir a não-alcançabilidade, uma vez que qualquer marcação alcançável deve ser solução desta equação. De fato, uma marcação obtida a partir da equação de estados é uma condição necessária, mas não suficiente, para garantir a sua alcançabilidade.

Um conceito importante relacionado à representação matricial de um modelo é o conceito de invariante. De fato, dois tipos de invariantes podem ser analisados: de lugar e de transição.

A.6.1 Invariantes de Lugar

Seja Y um vetor coluna de números naturais com dimensão igual ao número de lugares da rede, de forma que o *i*-ésimo componente deste vetor corresponda ao *i*-ésimo lugar da rede. Considere uma marcação M' obtida a partir da marcação M pelo disparo da transição t. Tem-se, então:

$$M' = M + C(\cdot, t)^T$$

Tomando-se o produto escalar em ambos os lado por Y, tem-se:

$$M' \cdot Y = M \cdot Y + C(\cdot, t)^T \cdot Y$$

Se $C(\cdot, t)^T \cdot Y = 0$, o número de *tokens* na rede ponderados por Y permanece constante, quando do disparo da transição t. Esta relação é chamada de invariante de lugar.

Para generalizar este conceito, de forma a obter invariantes para toda a rede independentemente da transição disparada, deve-se calcular o conjunto de vetores Y que são soluções para a equação matricial $C^T \cdot Y = 0$. Os vetores que correspondem à soluções não negativas são chamados de *P-semiflows*. A cada um destes *semiflows* corresponde um invariante de lugar, estabelecendo que o número de *tokens* nos lugares da rede poderados por este *semiflow* deve permanecer constante.

Uma propriedade que pode ser verificada a partir dos invariantes de lugar é a limitação de um modelo. Se a cada lugar da rede corresponder um componente não nulo de algum *P-semiflow*, a rede é estruturalmente limitada, significando que o número de *tokens* acumulados em qualquer lugar da rede é finito e garantindo a limitação do espaço de estados.

A.6.2 Invariantes de Transição

Os invariantes de transição correspondem a seqüências de transições, as quais, se puderem ser disparadas, são capazes de trazer o sistema de volta à uma dada marcação. Estes invariantes podem ser obtidos através da equação $M'' = M + [CV_\sigma]^T$. De fato, se σ é uma seqüência de disparos, de tal forma que $[CV_\sigma]^T = 0$, o disparo de σ a partir de uma marcação M faz com que o sistema retorne à esta marcação. Desta forma, as soluções inteiras para a equação matricial $C \cdot X = 0$ são chamadas de *T-semiflows*, onde X é um vetor cuja dimensão é igual ao número de transições presentes no modelo. Cada uma destas soluções estabelece uma relação invariante, chamada invariante de

transição, que determina uma seqüência de disparos capaz de trazer o sistema de volta para o estado em que este se encontrava antes do início da seqüência.

Deve-se observar que os invariantes de transição são obtidos diretamente de considerações feitas na equação de estados, a qual não considera marcação inicial. Desta forma, não há garantias de que as seqüências de transições calculadas a partir dos invariantes podem de fato ser disparadas, ou seja, a existência dos invariantes de transição é apenas uma condição necessária para que um modelo possa retornar ao seu estado inicial, ou seja, para verificar a reversibilidade de um modelo.

A.7 Técnicas Baseadas no Espaço de Estados

Conforme mencionado no início desta seção, existem propriedades das redes de Petri que dependem da marcação inicial. Tais propriedades, referidas como propriedades comportamentais, são normalmente verificadas a partir da geração do grafo de alcançabilidade correspondente ao modelo. Uma vez que o grafo de alcançabilidade contém todos os estados nos quais o modelo pode se encontrar, as técnicas de verficação baseadas neste grafo são referenciadas como técnicas baseadas em enumeração do espaço de estados. Em princípio, estas técnicas devem ser aplicadas para modelos limitados, uma vez que não é possível a construção do grafo de alcançabilidade para modelos ilimitados.

O principal problema da utilização de métodos baseados na enumeração do espaço de estados é que a quantidade de estados em sistemas não triviais cresce demasiadamente. De fato, o número de estados de um modelo cresce exponencialmente com o número de lugares que este possui. Este crescimento pode, inclusive, inviabilizar a construção do grafo de alcançabilidade, dado que os recursos computacionais disponíveis podem não ser suficientes.

Nos modelos em que a geração do grafo de alcançabilidade é possível, as propriedades podem ser verificadas utilizando algoritmos clássicos para a análise de grafos. Tais algoritmos permitem, por exemplo:

- Verificar se uma marcação é alcançável;
- Verificar se uma marcação é um *home state*;
- Verificar se um modelo é reversível;
- Constatar a ausência de *deadlock*; e
- Detectar transições mortas.

Por fim, é fundamental ressaltar que o auxílio de ferramentas é indispensável para a verificação de propriedades em modelos representando sistemas reais.

A.8 Modelos GSPN e DSPN

Os principais conceitos relacionados às redes GSPN e DSPN foram introduzidos na Seção 2.3.4. Esta Seção limita-se a apresentar uma definição formal para modelos desenvolvidos com base nestes formalismos. Com este intuito, define-se:

Definição 11: Um modelo GSPN é uma tupla:

$$\mu = (P, T, I, O, H, G, M_0, W, \pi)$$

onde:

P, I, O, *H* e M_0 possuem as mesmas definições apresentadas anterioremente;

T é o conjunto das transições que podem ser classificadas como exponenciais ou imediatas;

G é uma função de habilitação que, dada uma transição imediata e uma marcação, determina se a transição está habilitada ou não;

W é uma função que associa para cada transição um número real não-negativo representando o tempo médio de disparo, caso a transição seja exponencial ou o peso, caso a transição seja imediata; e

π associa a cada transição imediata um número natural que representa o seu nível de prioriade.

O modelo DSPN é uma extensão do GSPN que inclui a possibilidade de representar transições com duração determinística. Formalmente, tem-se:

Definição 12: Um modelo DSPN é uma tupla:

$$\mu = (P, T, I, O, H, G, \tau, M_0, W, \pi),$$

onde,

P, T, I, O, H, G, M_0 e π são definidos como em GSPN;

τ é uma função que associa a cada transição temporizada (determinística ou exponencial) um número real não-negativo representando o seu tempo médio de disparo; e

W é uma função que associa um número real não-negativo a cada transição imediata representando o seu peso.

Dissertação de Mestrado apresentada por **Fabio Nogueira de Souza** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título , **"Avaliação de Desempenho de Servidores de Aplicações Utilizando Redes de Petri – O Caso do JBoss"**, orientada pelo **Prof. Nelson Souto Rosa** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Paulo Roberto Freire Cunha
Centro de Informática / UFPE

Prof. Gledson Elias da Silveira
Departamento de Informática / UFPB

Prof. Nelson Souto Rosa
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 10 de julho de 2006.

Prof. FRANCISCO DE ASSIS TENÓRIO DE CARVALHO
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.