José Geraldo Ribeiro Júnior

Smart Proxies para Invocação de Serviços Web Replicados

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para a obtenção do grau de Mestre em Informática.

Belo Horizonte

Fevereiro de 2007

PUC Minas
Programa de Pós-graduação em Informática

FOLHA DE APROVAÇÃO

Smart Proxies para Invocação de Serviços Web Replicados

JOSÉ GERALDO RIBEIRO JÚNIOR

Dissertação defendida e aprovada pela banca examinadora constituida pelos senhores:

Prof. Marco Túlio de Oliveira Valente (PUC Minas) – Orientador
Doutor em Ciência da Computação - UFMG

Profa. Mariza Andrade da Silva Bigonha (UFMG)
Doutora em Informática PUC-Rio

Profa. Lucila Ishitani (PUC Minas)
Doutora em Ciência da Computação - UFMG

Belo Horizonte, 07 de fevereiro de 2007.

Resumo

Smart proxies são objetos usados para adaptar e customizar de forma não-invasiva aplicações distribuídas, notadamente para adicionar nas mesmas parâmetros de qualidade de serviço. Visando especialmente atributos como disponibilidade e desempenho, investiga-se, nesta dissertação, o uso de *smart proxies* para acrescentar transparência de replicação em sistemas de *middleware* destinados ao desenvolvimento de clientes de serviços Web. O sistema proposto, chamado SmartWS, inclui suporte às principais políticas propostas na literatura para seleção de serviços replicados. São elas: Aleatória, Estática, Paralela e via Melhor Mediana. Para a escolha da réplica, SmartWS considera fatores como características do cliente, latência de rede e carga de processamento nos servidores. Adicionalmente, propõe-se uma nova política de seleção de réplicas, denominada PBM (*Parallel Best Median*), que congrega pontos positivos das políticas existentes.

Outras funcionalidades foram incorporadas a SmartWS. Dentre estas funcionalidades incluem-se:

- O uso de adaptadores de interface, permitindo compatibilizar as interfaces remotas das réplicas com a interface abstrata utilizada por clientes de serviços Web.
- O estabelecimento de sessões de uso, que auxiliam na manutenção de consistência de dados da réplica acessada.
- Protocolo de invocação de métodos, que possibilita a economia de recursos de comunicação.

Ao longo da dissertação, são fornecidas diretrizes que auxiliam um usuário de serviços Web a optar pela política mais adequada a sua aplicação. Com o intuito de fornecer resultados quantitativos sobre a implementação de SmartWS, são apresentados experimentos realizados com clientes reais de serviços Web, geograficamente espalhados pelo Brasil.

Abstract

Smart proxies are objects often used to adapt and customize distributed object-oriented systems in a non-invasive way. For example, smart proxies are common used to provide support to quality of service attributes. In this work, we investigate the use of smart proxies in order to provide replication transparency in middleware systems that support the implementation of web services. The system presented, called SmartWS, supports the prime replicated server selection policies proposed in the literature. These policies are: Random, Static, Parallel and Best Median. In order to select the replica, SmartWS considers factors such as local connection capacity, external network conditions and servers workload. In addition a new server selection policy is proposed, called PBM (Parallel Best Median), that combines several advantages of the already proposed policies.

SmartWS also provides support to the following tasks:

- definition of object adapters that translate the interfaces supported by remote servers into the interfaces required by clients;
- definition of operations that start and finish a session of calls that should be dispatched to a single web server;
- definition of an invocation protocol, i.e., a protocol that specifies the legal sequences of messages that can be exchanged between a web service and its clients.

This work also provides guidelines that help web service users to choose the more suitable policy to a given application. Finally, this work presents results obtained from experiments performed with a Java prototype implementation of SmartWS. Such results reinforce the policy selection guidelines presented.

Dedico esta dissertação a Deus,
à minha esposa Cecília, ao meu filho Rafael,
ao meu pai, minha mãe, família e amigos.

Agradecimentos

Embora uma dissertação seja, por sua finalidade acadêmica, um trabalho individual, há algumas contribuições que não podem, nem devem, deixar de ser destacadas. Por essa razão, desejo expressar os meus sinceros agradecimentos:

Ao Prof. Doutor Marco Túlio de Oliveira Valente, professor e orientador, pela disponibilidade revelada ao longo destes dois anos. E pelas críticas e sugestões relevantes feitas durante a orientação.

Aos demais professores que tiveram a disponibilidade de me receber e ouvir, tirar dúvidas e ajudar a superar os inúmeros obstáculos.

À sempre prestativa Giovana pela inquestionável competência.

Aos meus amigos do Mestrado pela excelente relação pessoal que tivemos e que espero não se perca. Em especial a Anne, Cristiano, Léo, Michelle, Pedro, Sandro e Wanderley, que estavam sempre ao lado na busca por soluções aos tantos desafios e problemas colocados em questão. Estes são verdadeiros exemplos de pessoas que possuem espírito de equipe e que sabem que crescerão também sempre que algum de nós crescer.

Aos diretores e amigos do Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG). Escola em que me orgulho de ser servidor.

Ao Prof. Doutor Nabor Chagas, da UNIFOR, pela disponibilidade demonstrada desde o primeiro contato quando apresentei em um seminário uma dissertação por ele orientada.

Ao Glauber, que com dedicação e conhecimento tornou-se parte integrante deste projeto. Ao Breno, que disponibilizou alguns dos serviços Web utilizados nos experimentos.

Aos meus sogros que, durante todo esse período, ajudaram a cuidar do meu filhote, Rafael. Sem a ajuda de vocês e claro, do Angelo, nada disso seria possível.

Por último um agradecimento todo especial a minha esposa Cecília, ao meu filho Rafael e ao meu pai José Geraldo pelo inestimável apoio familiar e pela paciência e compreensão reveladas ao longo destes meses. Estas, sem dúvida, são as pessoas mais importantes da minha vida e somente graças a todo apoio e incentivo demonstrado este sonho se torna agora realidade.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

Computação orientada por serviços (*Service-Oriented Computing*) é um novo paradigma de computação distribuída que se acredita possuir potencial para alterar de forma radical o modo como sistemas Web são atualmente projetados, implementados, publicados e localizados [HS05, PG03]. Computação orientada por serviços possibilita a integração de diferentes aplicações, independentemente dos seus ambientes de desenvolvimento e de suas plataformas de execução. O objetivo final deste paradigma é, portanto, viabilizar o desenvolvimento de aplicações distribuídas fracamente acopladas que se beneficiem da infra-estrutura global de comunicação disponibilizada pela Web para integrar e automatizar processos de negócios envolvendo diferentes organizações.

Atualmente, serviços Web constituem o conjunto de tecnologias com maior potencial para transformar em realidade as idéias e princípios propostos pelo paradigma de Computação orientada por serviços [ACKM04, FF03, CDK+02]. Essencialmente, serviços Web são aplicações autônomas e distribuídas, identificadas por uma URI (*Universal Resource Identifier*), que se valem das seguintes linguagens e protocolos baseados em XML (*eXtensible Markup Language*) [ACKM04]:

- WSDL (*Web Services Description Language*) - para definição de interfaces remotas;
- UDDI (*Universal Description Discovery and Integration*) - para publicação e localização de serviços;
- SOAP (*Simple Object Access Protocol*) - para troca de mensagens com processos remotos;

- HTTP (*Hypertext Transfer Protocol*) - para transporte de mensagens entre aplicações clientes e servidoras.

Serviços Web representam uma evolução da Web tradicional para integrar processos de negócio além dos limites de uma empresa. No entanto, diversos desafios precisam ser ainda vencidos a fim de que o paradigma de computação orientada por serviços e a tecnologia subjacente de serviços Web sejam largamente utilizados no desenvolvimento e integração de sistemas geograficamente distribuídos. Dentre estes desafios, merece especial destaque a incorporação em serviços Web de mecanismos que garantam o atendimento a atributos de qualidade de serviço (QoS) [Men02, ZBN+04]. Particularmente, dois destes atributos – desempenho e disponibilidade – são críticos em aplicações destinadas a uma rede, como a Internet, sujeita a flutuações de largura de banda, alto número de falhas e elevadas latências.

Por outro lado, três décadas de pesquisa e prática em sistemas distribuídos demonstram que replicação é um conceito chave quando se objetiva incrementar o desempenho e a disponibilidade de aplicações distribuídas [CDK05]. Oferecer um mesmo serviço em um conjunto de réplicas aumenta a disponibilidade pois, se um serviço falha, outras réplicas podem continuar provendo o mesmo. Da mesma forma, definir corretamente qual réplica apresenta o menor tempo de resposta, por exemplo, faz com que o desempenho seja superior.

Assim, no caso prático de emprego de serviços Web, vislumbra-se a existência de diversos servidores, funcionalmente semelhantes e distribuídos ao longo da Internet. No entanto, não é razoável assumir qualquer nível de comunicação e coordenação entre estes servidores, visto que os mesmos são – e continuarão a ser – desenvolvidos de forma independente e autônoma por programadores localizados em qualquer ponto do planeta. Além disso, não é razoável assumir que todas as réplicas de um determinado serviço utilizam a mesma interface. Em geral, estas interfaces podem diferir em diversos detalhes, incluindo nome dos métodos, número, tipo e ordem dos parâmetros de entrada, dentre outros.

Clientes de serviços replicados utilizam sempre interfaces abstratas [AMS+06], as quais têm a função de padronizar as diversas interfaces remotas providas pelas réplicas utilizadas. Porém, isso não garante a padronização de interface entre as réplicas. Para ilustrar esta situação, a seguir são mostrados três códigos. O primeiro apresenta uma interface abstrata de uma aplicação cliente para acesso a servidores que disponibilizam um serviço de conversão de temperatura (de graus *Celsius* para *Fahrenheit* e vice-versa).

```
interface TemperatureConverter {
  int toFahrenheit(int tc);
```

```
  int toCelsius(int tf);
}
```

Os códigos a seguir apresentam interfaces remotas de dois servidores que disponibilizam o serviço de conversão de temperatura. O primeiro servidor, hospedado em um repositório de serviços Web conhecido como WebserviceX (*www.Webservicex.net*) e o segundo, hospedado na empresa de treinamento DeveloperDays (*www.developerdays.com*). É possível perceber as diferenças entre as interfaces concretas, necessitando de algum mecanismo de adaptação entre as mesmas.

```
interface ConvertTemperatureSoap {
  int convertTemp(temp, int fromUnit, int toUnit);
}

interface ITempConverter {
  int ctoF(int t);
  int ftoC(int t);
}
```

É importante dizer que nos cenários onde a solução proposta nesta dissertação se aplica, todas as réplicas do serviço são previamente conhecidas e não é possível qualquer tipo de interferência ou configuração extra dos servidores.

Clientes de serviços replicados devem suportar políticas e algoritmos para realização de tarefas como: escolha do servidor que melhor atenda aos seus requisitos de qualidade de serviço e compatibilização da interface provida por este servidor com a interface requerida pelo cliente.

A teoria e a prática em sistemas distribuídos recomendam que estas tarefas sejam encapsuladas por um sistema de *middleware* de apoio ao desenvolvimento de clientes de serviços Web, de forma a evitar um entrelaçamento entre o código funcional destes clientes e o código responsável pela implementação das referidas tarefas [VVJ06]. Em resumo, o sistema de *middleware* utilizado deve prover *transparência de replicação*. No entanto, sendo serviços Web uma tecnologia recente, as plataformas de *middleware* mais utilizadas atualmente para desenvolvimento deste tipo de sistema ainda não disponibilizam suporte a replicação. Dentre elas podemos citar: Apache SOAP [Fou], Apache Axis [Axi], .NET [Mica] e JAX-RPC [Micb].

Smart proxies são um dos principais mecanismos de meta-programação usados para extensão e customização de sistemas de *middleware* [KK00b, WPS01]. Por exemplo, suporte a *smart proxies* já foi proposto para outras plataformas de *middleware*, como

TAO [WPS01] e Java RMI [VSC03, SMS02], sempre com o objetivo de adicionar parâmetros de qualidade de serviço em uma aplicação de forma modular, transparente e não-invasiva.

Propor rotinas para seleção de réplicas em uma camada subjacente à aplicação é mais interessante, pois retira do desenvolvedor a responsabilidade de desenvolver rotinas não-funcionais. No entanto, no caso de servidores replicados, freqüentemente é necessário desabilitar o uso de uma política de seleção de réplicas após a execução de determinadas operações. Em geral, sessões de uso de réplicas devem ser estabelecidas quando uma operação altera o estado de uma réplica e exige que operações seguintes sejam processadas sobre o estado alterado. Por exemplo, em um sistema de comércio eletrônico, operações como *pesquisaProduto* podem ser processadas por qualquer servidor. Por outro lado, as operações *login* e *logout* delimitam uma sessão, fixando o uso de uma réplica.

Interfaces são normalmente usadas para definir as assinaturas dos métodos constituintes de um serviço e com isso viabilizar verificação estática de tipos em clientes de tais métodos. No entanto, quase sempre existe um conjunto de regras que restringe a ordem em que as operações descritas em uma interface podem ser invocadas. Por exemplo, em um sistema de comércio eletrônico, a operação `logout` somente pode ser executada se antes tiver sido invocada a operação `login`. Em sistemas distribuídos, protocolos de invocação são usados para definir as seqüências válidas de mensagens que podem ser enviadas para servidores remotos [BL89, YS97, CFP+03].

Em sistemas distribuídos tradicionais, para redes locais ou corporativas, o uso explícito de um protocolo de invocação pode não representar uma vantagem, já que nestes casos clientes são desenvolvidos em geral pela mesma equipe de programadores encarregada do desenvolvimento de provedores de serviços. Como esta equipe conhece em detalhes o protocolo de invocação de cada interface remota, as chances de ocorrência de seqüências inválidas de chamadas são menores. Por outro lado, esta característica não se repete no caso de aplicações baseadas em serviços Web. Nestes casos, um determinado serviço pode ter milhares ou milhões de clientes, desenvolvidos por programadores diferentes. Com isso, as chances de ocorrência de seqüências inválidas de chamadas são maiores e, portanto, a incorporação explícita de código nos clientes para detectar tais chamadas evita que elas sejam transmitidas pela Web e processadas em servidores remotos. Existe ainda uma tendência que interfaces de serviços Web tenham "maior granularidade" – isto é, tenham mais métodos complexos – que interfaces de aplicações tradicionais baseadas em objetos distribuídos [BD05].

1.2 Objetivos

Este trabalho de dissertação de mestrado teve como objetivo propor e avaliar o uso de *smart proxies* para prover transparência de replicação em aplicações baseadas na tecnologia de serviços Web. O sistema proposto, chamado SmartWS [JCV06], disponibiliza recursos para:

- definição das interfaces e da localização dos servidores que disponibilizam um determinado serviço Web;
- definição das seguintes políticas já propostas na literatura para seleção de serviços Web replicados [DRJ00, MS05]: Estática, Aleatória, Paralela e Melhor Mediana. Adicionalmente, propõe-se nesta dissertação uma nova política de seleção de réplicas, chamada PBM (*Parallel Best Median*), que congrega pontos positivos das políticas anteriores;
- geração automática de *smart proxies*, a partir de uma especificação realizada pelo usuário do sistema. SmartWS utiliza *smart proxies* como extensão de um sistema de *middleware* para prover suporte a transparência de replicação;
- definição de objetos adaptadores, responsáveis pela conversão e compatibilização das interfaces remotas providas pelo conjunto de servidores replicados para a interface abstrata requerida pelos clientes. Basicamente, um objeto adaptador deve implementar a interface abstrata e possuir uma referência para o serviço que está sendo adaptado. A implementação dos métodos da interface abstrata em um objeto adaptador deve se responsabilizar por adaptar e converter tais métodos à sintaxe esperada pelo método da interface remota da réplica. Em SmartWS, adaptadores devem ser definidos manualmente pelos usuários do sistema, já que estes possuem conhecimento tanto da interface abstrata de um serviço como da interface remota das réplicas utilizadas no sistema;
- definição de sessões de uso, com o intuito de manter a consistência de dados da réplica acessada. Basicamente, em algumas situações, se uma determinada requisição op_{start} é destinada a uma réplica r de um serviço, deve-se destinar todas as outras requisições subseqüentes deste cliente para a mesma réplica r, até que uma outra operação op_{end} seja executada. Nestes casos, diz-se que as operações op_{start} e op_{end} delimitam uma sessão de uso de uma réplica, isto é, as mesmas respectivamente desabilitam e voltam a habilitar o emprego de uma política de seleção de réplica;

- definição de um protocolo que especifica as seqüências legais de invocação dos métodos de uma interface abstrata. Isto é, define-se um conjunto de regras que determinam as seqüências válidas de chamadas de métodos definidos na interface de um serviço Web. Este protocolo é descrito por meio de um autômato finito determinístico associado a cada cliente dos serviços providos por uma interface remota. As transições deste autômato representam chamadas aos métodos da interface abstrata à qual o protocolo está associado. Ao se invocar um método remoto, caso o estado atual do autômato não possua uma transição correspondente ao método, então a referida chamada é inválida, isto é, a mesma não deveria ter sido solicitada no estado atual do sistema.

1.3 Contribuições

As principais contribuições deste trabalho são listadas abaixo:

- proposta de uma nova política de seleção de réplicas chamada PBM (*Parallel Best Median*), que combina aspectos positivos de políticas que apresentaram os melhores desempenhos em trabalhos anteriores, tanto no que se refere a tempo de resposta quanto a tolerância a falhas;

- implementação de um protótipo em Java do sistema SmartWS. Neste protótipo, todo o código dos *smart proxies* é gerado de forma automática a partir de uma especificação realizada pelo usuário de SmartWS. A especificação é feita em um arquivo de configuração, utilizando uma linguagem própria e simples;

- validação do protótipo proposto, por meio de experimentos baseados em serviços Web reais. Foram realizados experimentos de longa duração com o objetivo de comparar o desempenho das políticas apresentadas nesta dissertação, bem como apresentar os principais benefícios proporcionados pelas mesmas;

- pela avaliação de cenários distintos, foi possível definir diretrizes que permitem a um programador de clientes de serviços Web optar pela política mais adequada à sua aplicação.

1.4 Estrutura da Dissertação

O restante desta dissertação está organizado em cinco capítulos, descritos a seguir:

- No Capítulo 2 são apresentadas e discutidas as tecnologias envolvidas no projeto e implementação de SmartWS e trabalhos relacionados com o mesmo;
- No Capítulo 3 são apresentadas as principais funcionalidades providas pelo sistema SmartWS, incluindo as políticas de seleção de réplicas, o mecanismo de adaptação de interface e as funcionalidades de sessão de uso e protocolo de invocação de métodos;
- No Capítulo 4 descreve-se a interface de programação do sistema. Neste capítulo, além da arquitetura do sistema SmartWS, são apresentados detalhes da linguagem de especificação com a qual se define as configurações de replicação do sistema;
- No Capítulo 5 são descritos experimentos realizados com SmartWS. Estes experimentos foram conduzidos com o intuito de validar o projeto de SmartWS e propor diretrizes para escolha da política de seleção de réplicas mais adequada a uma determinada aplicação cliente de serviços Web;
- No Capítulo 6 são apresentadas as conclusões desta dissertação e indicadas possíveis linhas para trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

2.1 Serviços Web

Limitações existentes na Web tradicional não permitem que outras aplicações, além de *browsers*, acessem e manipulem informações disponibilizadas na Internet. A utilização de linguagens e protocolos originais da Web inviabiliza a integração de aplicações geograficamente distribuídas, uma vez que estas apresentam como saída documentos textuais, manipulados facilmente por homens, mas não por outras aplicações. Porém, a necessidade de comunicação e a busca pela interoperabilidade entre aplicações tem servido de motivação para uma mudança de paradigma que aponta para a migração rumo a uma arquitetura orientada por serviços (*Service-oriented Architecture*), usando serviços Web.

O W3C (*World Wide Web Consortium*) [Con], define serviços Web como aplicações identificadas por uma URI, cujas interfaces e invocações são descritas, especificadas e descobertas utilizando XML. Serviços Web são constituídos por uma pilha de padrões caracterizados por um alto nível de flexibilidade, conectividade, acessibilidade e interoperabilidade [KR04]. Os principais padrões existentes em serviços Web são [ACKM04]:

- SOAP (*Simple Object Access Protocol*) - protocolo W3C para troca de informações na Internet. Utiliza XML para codificação de mensagens e, na maioria das vezes, HTTP para transporte de mensagens de um nodo a outro. SOAP especifica como representar em XML todas informações necessárias para se executar uma chamada remota;

- WSDL (*Web Service Description Language*) - linguagem para descrever a localização (nome e URL do serviço) e a interface de um serviço Web (assinatura dos métodos disponíveis para chamada remota);

- UDDI (*Universal Description, Discovery and Integration*) - especificação em WSDL que permite a um cliente obter todas as informações para interagir com um serviço Web. UDDI é um conjunto de especificações para registrar e pesquisar por um serviço Web, via interface Web ou SOAP.

O objetivo principal de serviços Web é proporcionar interoperabilidade e comunicação transparente entre sistemas geograficamente distribuídos, independentemente da plataforma, tecnologia de rede, linguagem de programação ou sistema operacional utilizado. Porém, a interoperabilidade entre processos inter-organizacionais apresenta um custo com a latência de comunicação entre as aplicações [Lit02, EPL02]. No entanto, se comparadas a tecnologias tradicionais de componentes como CORBA [Gro00], DCOM [Mod96] e RMI [WRW96], que apresentam grande dificuldade de integração, em uma relação custo/benefício, a latência é um pequeno custo perto do ganho com a comunicação entre plataformas distintas e heterogêneas.

2.1.1 Modelo Conceitual

A Figura 2.1 apresenta o modelo conceitual da arquitetura de serviços Web. É possível identificar três tipos de entidades em um ambiente típico de serviços Web: provedor de serviços, consumidor de serviços e registro de serviços. Pode-se identificar ainda as principais operações realizadas em arquiteturas baseadas em serviços Web: publicação, localização e chamada.

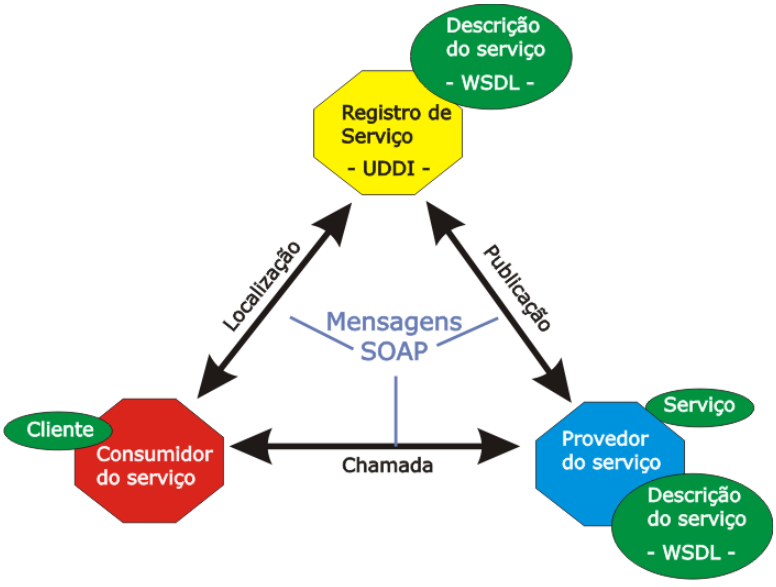

Figura 2.1: Modelo conceitual da arquitetura de serviços Web. Adaptado de [FF03]

O provedor de serviço define a descrição WSDL do serviço e a publica para o consumidor de serviços no registro. O consumidor de serviços pode, via URI e a descrição WSDL do serviço publicado, usando UDDI, se ligar ao provedor de serviços e invocar ou interagir com a implementação do serviço Web. No arquivo que contém a descrição WSDL, o cliente encontrará os métodos disponíveis, além de detalhes sobre o tipo de resposta que obterá [Kre01].
Uma invocação é feita por meio da chamada de um método remoto, cujas informações e parâmetros são empacotados em uma mensagem SOAP, codificada em XML. Normalmente, a chamada termina com a recepção de uma segunda mensagem, também codificada em XML, com o resultado do processamento do serviço Web.

2.1.2 Protocolos e Tecnologias de Serviços Web

SOAP - Simple Object Access Protocol

O protocolo SOAP é responsável pela troca de informações entre aplicações clientes e servidoras de um serviço Web, não definindo um modelo de programação, nem uma interface para programadores de aplicações. SOAP define um mecanismo de troca de mensagens, via o empacotamento e codificação de dados baseados em esquemas XML.

Apesar do protocolo HTTP ser o mais utilizado em transações que envolvem serviços Web, SOAP não define o protocolo de transporte. SOAP permite adicionar serviços a um servidor Web e então acessar estes serviços remotamente a partir de programas comuns, pela especificação em XML de todas as informações necessárias para executar chamadas remotas.

A Figura 2.2 descreve a arquitetura de uma transação via protocolo SOAP. A aplicação faz uma chamada (passo 1). Em seguida, o *stub* interpreta a chamada, constrói a mensagem de requisição XML e a transmite (passo 2). O *skeleton* SOAP recebe, analisa gramaticalmente e valida a requisição (passo 3). O *skeleton* envia a mensagem para o componente servidor (passo 4). O *skeleton* recebe o resultado da chamada, constrói uma mensagem XML de resposta e a transmite (passo 5). Finalmente, o *stub* recebe, analisa gramaticalmente a resposta e retorna o resultado ao cliente (passo 6). Todo o processo descrito é transparente para o cliente e o componente servidor.

WSDL - Web Services Description Language

Um documento WSDL define as operações, interfaces, tipos de dados, localização e protocolo de transporte de um serviço Web [Cer02]. Em um documento WSDL é possível encontrar informações como nome e URL do serviço, além da assinatura dos métodos

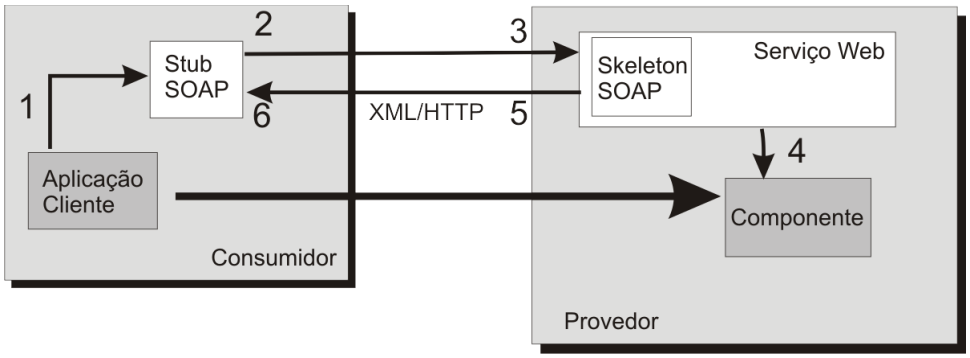

Figura 2.2: Arquitetura SOAP. Adaptado de [PM05]

disponíveis para a chamada remota. O conteúdo de um documento WSDL se assemelha a um documento IDL CORBA porém, é baseado em XML.

WSDL possibilita descrever os serviços e suas operações sem levar em conta o formato da mensagem ou o protocolo de rede utilizado. Um documento WSDL é formado por metadados sobre as operações implementadas no serviço Web.

Existem ferramentas que geram especificações WSDL a partir do código de implementação do serviço. Da mesma forma, o código para consumir o serviço pode ser gerado pela interpretação do conteúdo de um documento WSDL. O *framework* AXIS [Axi] é uma das ferramentas que fazem este trabalho automaticamente, seja a partir do código fonte da aplicação ou do serviço publicado. A ferramenta *WSDL2Java* é responsável por desempenhar este papel. A partir de uma interface WSDL, WSDL2Java gera todas as interfaces e classes (*stubs*) necessárias para que uma aplicação Java possa consumir o serviço Web.

O código abaixo apresenta parte de uma interface WSDL de um serviço *sayHello*.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="urn:hello"
   ......
   <wsdl:message name="sayHelloResponse">
      <wsdl:part name="sayHelloReturn" type="xsd:string"/>
   </wsdl:message>
   <wsdl:message name="sayHelloRequest">
      <wsdl:part name="in0" type="xsd:string"/>
   </wsdl:message>
   <wsdl:portType name="HelloServer">
      <wsdl:operation name="sayHello" parameterOrder="in0">
         <wsdl:input message="intf:sayHelloRequest"
```

```
         name="sayHelloRequest"/>
         <wsdl:output message="intf:sayHelloResponse"
         name="sayHelloResponse"/>
      </wsdl:operation>
   </wsdl:portType>
   ..........
   </wsdl:binding>
   <wsdl:service name="HelloServerService">
      <wsdl:port binding="intf:rpcrouterSoapBinding"
      name="rpcrouter">
         <wsdlsoap:address location=
         "http://localhost8080/soap/servlet/rpcrouter"/>
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

UDDI - Universal Description, Discovery and Integration

O protocolo UDDI, via um mecanismo padrão para publicação e descoberta de informações, permite ao cliente obter informações para interagir com um serviço Web. Criar um padrão "universal" UDDI tem por objetivo obter uma plataforma independente e aberta para descrever, descobrir e integrar serviços Web.

Em um ambiente global e distribuído, deixa de ser interessante ter um mecanismo manual para descoberta de serviços uma vez que vem crescendo o uso de serviços Web, principalmente em sistemas de comércio eletrônico. Em registros UDDI são fornecidas informações como nome do serviço, uma breve descrição do que ele faz, um endereço onde ele pode ser acessado e a descrição da interface para acessar o serviço.

Basicamente as pesquisas por um serviço podem ser realizadas em [OT02]:

- páginas brancas: contêm informações básicas como nome e informações de contato. Essas informações permitem descobrir um serviço Web pela identificação de seu provedor;
- páginas amarelas: serviços organizados em categorias ou classificações.

2.1.3 Coordenação e Composição de Serviços Web

Tecnologias de serviços Web permitem ainda que ambientes mais complexos sejam criados, por meio de uma composição de serviços primários. O objetivo é a utilização de todos os serviços Web envolvidos, segundo uma lógica de negócio. Em uma composição de serviços Web, define-se quais serviços serão invocados, em que ordem e como serão tratadas as exceções.

Uma composição de serviços Web pode ser utilizada para [Arc]:

- coordenação - controlar a execução dos serviços componentes;
- monitoramento - obter informações dos serviços componentes via informações fornecidas pelos mesmos;
- validação - garantir a integridade da composição verificando os parâmetros, impondo restrições e reforçando regras de negócio;
- prover QoS - por meio de uma configuração apropriada, é possível definir responsabilidades e prioridades para os serviços envolvidos dentro de uma lógica de negócio.

Várias especificações têm sido propostas como mecanismos para composição de serviços Web. Curbera [CKM+03] define BPEL (*Business Process Execution Language*) como uma linguagem para definição de composição de serviços na forma de processos de negócios. BPEL está sendo padronizada pela OASIS (*Organization for the Advancement of Structured Information Standard*). Verbeek [ADH+05] considera que, apesar de existir uma expectativa de que BPEL será utilizada para desenvolvimento de sistemas de informação intra-organizacionais, o uso da linguagem é complexo. Para Verbeek, a representação XML de BPEL é muito verbosa e capaz de ser lida somente por olhos treinados. Isto acontece pelo fato de BPEL oferecer diversos construtores, podendo-se implementar rotinas de muitas maneiras. Mesmo com a existência de ferramentas gráficas para gerar o código BPEL, estas representam uma imagem direta do código e não são intuitivas para o usuário final.

Curbera apresenta ainda o WS-Coordination [CFL+a] e o WS-Transaction [CFL+b] para coordenação e transação entre serviços Web. No que se refere à segurança de mensagens SOAP provendo integridade, confidencialidade e autenticação, existem os seguintes esforços: WS-Security [Sec] e WS-ReliableMessaging [WR].

A especificação WS-Policy [WP] provê uma gramática flexível que estende WSDL para permitir a codificação e acréscimo de informações de QoS, na forma de políticas de serviços reutilizáveis, isto é, define os requisitos que devem ser cumpridos pelo cliente e pelo provedor do serviço.

2.2 Políticas de Seleção

Políticas para seleção transparente de serviços Web replicados, implementadas por clientes, já foram objeto de outros estudos. Porém, a maioria destas pesquisas se limitou a avaliar políticas de acesso a recursos na forma de documentos HTML e imagens. Dykes [DRJ00] apresenta mecanismos e algoritmos para seleção de servidores replicados com a intenção de avaliar políticas de escolha de servidores que ofereçam melhor desempenho.

Mendonça e Silva apresentam o *framework* RWS [SM04], o qual serviu de inspiração para o projeto e implementação do sistema descrito nesta dissertação. O *framework* RWS é uma extensão do *framework* AXIS para implementação de serviços Web. Ele explora alternativas de invocação para melhorar e tornar transparente o acesso a serviços Web replicados, deixando a cargo de clientes a decisão sobre qual réplica do serviço invocar. Segundo Mendonça e Silva, para tornar a escolha da réplica transparente para a aplicação, o ideal é que as camadas auxiliares, na forma de *stubs*, incorporem mecanismos que realizem a invocação do modo mais eficiente.

Algumas contribuições do *framework* RWS são:

- criação de uma estrutura para melhorar o desempenho de clientes no acesso a serviços Web geograficamente replicados;
- tratamento da localização das réplicas dos serviços - explorando o modelo de descrição de serviços oferecido pela linguagem WSDL;
- avaliação empírica de políticas que possam tornar mais eficiente o processo de invocação das réplicas no lado do cliente.

A estrutura geral do *framework* RWS é bem semelhante à do AXIS, porém, para substituir a classe *locator*, responsável por obter a localização do serviço, foram adicionados dois novos componentes: Gerente de seleção (*Selection manager*) e Gerente de invocação (*Invocation manager*).

Diferentemente do *framework* AXIS, RWS trata a identificação dos endereços alternativos, onde estão disponibilizadas as réplicas para um mesmo serviço, obtidos a partir de seu documento WSDL, de forma transparente para a aplicação cliente. Para que isso fosse possível, foi necessário modificar a classe *WSDL2Java*, integrante do AXIS. A nova classe, *WSDL2JavaR*, trata a existência de múltiplos endereços para um mesmo serviço. Outra diferença aparece na geração do *stub*, onde este passa a possuir uma operação *setPolicy*, que implementará a política selecionada pela aplicação.

Foram utilizadas cinco políticas para seleção de servidores: Seleção Aleatória (chamada pelos autores de "Randômica"), Paralela, HTTPing, Melhor Última e Melhor Mediana.

Seleção Aleatória: a aplicação cliente informa ao objeto *stub* que a seleção será feita de forma aleatória. A escolha do servidor é efetuada pela classe *SelectionManager*, a partir da geração de um número aleatório dentro do intervalo de endereços disponíveis para o serviço. Em seguida, a requisição é enviada ao servidor escolhido, através de um objeto da classe *Call*, que representa uma requisição na infra-estrutura originalmente provida pelo AXIS. Nos experimentos realizados, esta política apresentou o pior desempenho, sendo inclusive considerada inviável no processo de invocação de serviços Web. O resultado não chega a surpreender, pois se nenhum critério como carga, tráfego ou tempo de acesso é levado em consideração é possível que um servidor com o pior desempenho seja selecionado várias vezes.

Seleção Paralela: envia uma requisição de um serviço Web, de forma concorrente, a todos os servidores que disponibilizem uma réplica do serviço. Na execução desta política, a aplicação cliente informa ao objeto *stub* que a requisição será enviada a todos os servidores de forma paralela. A chamada é encaminhada por meio de múltiplas *threads*. A quantidade de *threads* instanciadas e executadas equivale ao número de endereços onde o serviço está disponível. Para cada *thread* é instanciado um novo objeto da classe *Call*. A *thread* que recebe a resposta do servidor em primeiro lugar a repassa à instância da classe *InvocationManager*, que por sua vez a repassa ao objeto *stub*, daí chegando à aplicação cliente. As demais *threads* são interrompidas e as respectivas respostas, quando recebidas, são ignoradas. Esta política se mostrou a mais interessante em redes locais e *links* de conexão à Internet com maior largura de banda disponível.

Seleção HTTPing: envia uma requisição do tipo HTTP HEAD, antes do envio da requisição ao servidor. A requisição HTTP é enviada, de forma concorrente, a todos os servidores que hospedam o serviço Web replicado. Aquele que primeiro responder à requisição é selecionado para a invocação do serviço. Duas classes adicionais foram implementadas para essa política. A classe *HTTPingHead* tem o papel de gerenciar a instanciação e controlar a execução de objetos da classe *HTTPingThread*. Esta segunda classe tem como papel o envio de uma requisição HTTP HEAD para cada um dos servidores com réplicas disponíveis. Durante a execução, o objeto da classe *stub*, devidamente parametrizado pela aplicação cliente, interage com uma instância da classe *SelectionManager*, que por sua vez aciona uma instância da classe *HTTPingHead*, a qual, de posse

da lista de servidores que oferecem uma réplica do serviço, instancia objetos da classe *HTTPingThread* de acordo com a quantidade de servidores existentes para o serviço em questão. Esses objetos enviam uma requisição HTTP HEAD a cada um dos servidores. O primeiro servidor a responder é selecionado. Essa informação é passada pelo objeto da classe *HTTPingHead*, por meio da instância de *SelectionManager*, até o objeto *stub*, que em seguida envia a requisição ao servidor selecionado. As principais motivações para a implementação da política HTTPing estão relacionadas à necessidade de avaliar não somente os níveis de serviço de rede, mas também o nível de carga no serviço HTTP de cada servidor e contornar algumas limitações de natureza técnica, como por exemplo, a ausência de resposta, pela maioria dos servidores na Internet, aos pacotes ICMP enviados na forma de *pings*. Além disso, é comum a utilização de servidores *proxy*, do lado do cliente, com a função de controle de acesso à Internet, o que normalmente implica no bloqueio do envio/recebimento de pacotes utilizados pela técnica TCPing.

Os experimentos mostraram que a política HTTPing não apresentou o resultado esperado uma vez que, segundo conclusão dos próprios autores, nem sempre o servidor que respondia à requisição HTTPing era o servidor com o melhor desempenho.

Seleção Melhor Última: é uma das políticas do *framework* RWS que, juntamente com a política Melhor Mediana, realiza a seleção de servidores considerando dados de invocações anteriores. Para sua implementação, foram criadas classes adicionais que têm como papel o armazenamento e tratamento dos dados históricos utilizados na escolha dos servidores para o envio de chamadas futuras.

Tanto na política Melhor Última quanto na Melhor Mediana são armazenadas as informações referentes aos resultados de invocações anteriores feitas a cada um dos servidores que contêm réplicas do serviço. O método *getBestLastEndpoint()*¸ definido na classe *HistoryLog*, retorna o endereço do servidor que teve o melhor desempenho dentre as últimas invocações enviadas a cada um dos servidores participantes do conjunto de réplicas.

Nas políticas Melhor Última e Melhor Mediana, a aplicação cliente, ao receber a resposta de uma chamada ao serviço realizada com sucesso, faz o registro dos dados necessários para uso estatístico posterior. Ao solicitar uma nova invocação ao serviço, a aplicação cliente aciona o objeto da classe *stub*, o qual aciona um objeto da classe *SelectionManager*, que por sua vez aciona uma instância da classe *HistoryLog*, para finalmente selecionar um servidor de acordo com o critério descrito.

A política Melhor Última apresenta uma deficiência que é deixar de invocar um servidor, ou pelo menos reduzir sua prioridade, após uma única resposta ruim. Esta deficiência, em casos extremos, pode levar esta política a um desempenho pior que a política aleatória.

Seleção Melhor Mediana: consiste na seleção do servidor com o melhor desempenho mediano, calculado a partir dos dados históricos das k últimas invocações. O valor de k é um parâmetro fornecido pela aplicação cliente na utilização desta política. No caso de $k = 1$, a política Melhor Mediana se reduz à política Melhor Última. A diferença, em relação à política Melhor Última, está na interação com a classe *HistoryLog*, que agora é realizada pelo método *getBestMedianEndpoint()*. Este método retorna o endereço do servidor que ofereceu o melhor desempenho mediano dentre as k últimas invocações bem sucedidas registradas no histórico.

Experimentos: a metodologia de avaliação contou com duas fases: (1) invocação periódica de uma operação de um serviço Web replicado, fazendo uso alternado das cinco políticas apresentadas e (2) armazenamento e análise dos dados referentes ao resultado de cada invocação permitindo uma avaliação de desempenho sob diferentes cenários.

Foi estipulado um tempo máximo para que as chamadas ao serviço replicado fossem executadas. Para isso, a aplicação cliente teve que ser executada sob controle externo de outra aplicação, a qual ficaria encarregada de monitorar o tempo de execução das chamadas e, caso o tempo máximo fosse ultrapassado, registrar uma ocorrência no arquivo de *log* referente à falha por tempo expirado (*timeout*).

Para os testes, o serviço Web utilizado foi o *Web UDDI Business Registry*, disponível em quatro servidores (Microsoft, IBM, NTT e SAP) localizados em três continentes (América do Norte, Europa e Ásia). Cada réplica do serviço foi invocada por meio da operação *GetServiceDetail* que retorna os detalhes armazenados pelo serviço UDDI sobre um ou mais serviços. A estrutura física para testes contou com duas estações clientes dedicadas, localizadas em Fortaleza e conectadas à Internet visa diferentes *backbones* de comunicação (UNIFOR - Universidade de Fortaleza e BNB - Banco do Nordeste). A rede da UNIFOR possuía um *link* de 2 Mbps e a do BNB um *link* de 4 Mbps.

Para medir o desempenho foi utilizado como métrica o tempo de resposta pela aplicação cliente. O processamento de estatísticas requerido pelas políticas baseadas em *ping* e dados estatísticos foi considerado dentro do tempo total observado em cada invocação. Esta métrica teve como objetivo refletir todo o tempo experimentado pelo cliente, em qualquer uma das políticas de invocação. Com a utilização desse modelo foi possível observar que a média da amostra não refletia o comportamento típico da maioria dos tempos de resposta obtidos. A mediana dos tempos de resposta evidenciou-se como a medida mais adequada para avaliação, uma vez que foi pouco afetada pelos altos tempos de resposta considerados atípicos.

Em uma análise quantitativa, foram identificados os períodos de maior e menor tráfego nos dois clientes e avaliado o comportamento das políticas separadamente em cada período. Foram considerados tanto os valores absolutos do desempenho obtido em cada política, quanto o ganho relativo das políticas de melhor desempenho em relação à política Aleatória, que apresentou o pior resultado.

A avaliação realizada permitiu a obtenção das seguintes conclusões:

- a melhor alternativa, quando se leva em consideração características específicas do cliente, como limite de rede, é a política Melhor Mediana;
- os desempenhos obtidos pelas políticas Melhor Última e Melhor Mediana ficaram bem próximos em ambos os clientes;
- o melhor desempenho quando se tem largura de banda suficiente para absorver a concorrência no recebimento das respostas dos quatro servidores foi obtido com a política Paralela;
- o pior desempenho foi produzido pela política Aleatória, mostrando não ser uma política a ser utilizada em invocações de serviços Web replicados.

Avaliação crítica: O trabalho apresentado por Mendonça e Silva apresenta uma boa avaliação empírica na qual foram realizados experimentos variados e de longa duração. Os experimentos contaram com serviços Web reais, espalhados por três continentes. Porém, em RWS cabe à aplicação cliente comandar a definição da política de seleção de réplicas. O trabalho não apresenta diretrizes claras e objetivas sobre os ambientes de rede adequados a cada uma das políticas de seleção de réplicas propostas. A análise sobre os resultados apresentados não considera o fato dos servidores Web, justamente por estarem em continentes diferentes, possuírem picos distintos de processamento e tráfego de rede de acordo com os fuso horários. Pelo fato de utilizar servidores dedicados, de grande porte, que sofreram poucas variações de desempenho, seja por falha ou aumento no tempo de resposta, ignorou-se o tratamento de falhas. Uma conseqüência direta foi não detectar um problema de adaptação da política via Melhor Mediana.

Erradi [EMT06] propõe o *framework* denominado MASC (*Manageable and Adaptive Service Composition*) para o tratamento e gerenciamento de falhas em composição de serviços Web. A proposta conta com um sistema independente de monitoramento dos servidores, classificando seu "estado" no momento anterior a requisição. Três políticas de seleção são suportadas na proposta: Aleatória, Paralela e Melhor Desempenho (baseado

nas n últimas invocações). O artigo apresenta um estudo de caso, executado em um protótipo desenvolvido em .NET, em que simulações são feitas com um cliente e um servidor, dentro de uma LAN. Além dos experimentos serem curtos, com apenas 500 invocações, não foram utilizados serviços Web reais. Ainda sobre os experimentos, não fica claro se as diferenças apresentadas utilizando MASC são válidas, isto é, se essas diferenças são estáveis ou variam muito de experimento para experimento. Como os valores obtidos são muito pequenos, da ordem de 915-1045 ms, dependendo da influência da rede nesses tempos, as diferenças observadas podem não refletir o que aconteceria com o *overhead* apresentado pelo uso de MASC, caso fossem utilizadas invocações envolvendo mensagens significativamente maiores.

2.3 Smart Proxies

Smart proxies [WPS01, KK00a] são objetos usados para adaptar e customizar de forma não-invasiva aplicações distribuídas, notadamente para adicionar nas mesmas mecanismos como *log*, *cache*, QoS e tolerância a falhas. Estes mecanismos são disponibilizados de forma transparente, uma vez que o desenvolvedor da aplicação se preocupa exclusivamente com o código da mesma, deixando para o *smart proxy* a implementação do requisito não-funcional demandado pelo cliente.

Ledru [Led02] propõe a incorporação de *smart proxies* em sistemas baseados em Jini [Wal99], com o objetivo de redirecionar, de forma transparente, chamadas remotas em caso de falhas no servidor primário encarregado do processamento das mesmas. A solução proposta por Ledru possibilita a um cliente de uma federação Jini reconectar-se automaticamente a outra instância do serviço, em caso de falha.

Outras soluções já foram propostas fazendo o uso de *smart proxies* como mecanismo de meta-programação. Valente [VSC03] apresenta uma extensão de RMI, chamada RMI+, com suporte a interceptadores de chamadas remotas de métodos. O sistema proposto permite ao desenvolvedor da aplicação definir meta-objetos que são executados automaticamente quando uma chamada remota é disparada, de forma não-invasiva. Na proposta apresentada, tais interceptadores são introduzidos tanto do lado do cliente quanto do servidor de uma aplicação distribuída. O sistema de interceptação proposto por Valente é dinâmico, já que interceptadores são criados, inseridos e removidos em tempo de execução.

Santos [SMS02] propõe a criação de uma camada adicional, denominada SmartRMI, onde objetos são incluídos no percurso de uma invocação de um método remoto entre o cliente e o servidor. A proposta consiste da substituição do *stub* por um *proxy* dinâmico. Desta forma, quando um cliente tenta acessar um objeto remoto, em vez do *stub* ele re-

cebe, além de um *proxy* dinâmico, um manipulador de invocação. Os manipuladores de invocação, além de encaminharem a invocação do método para o servidor, provêem mecanismos que podem ser utilizados para implementar funcionalidades específicas do serviço, como por exemplo, implementar segurança do lado do servidor. O encaminhamento da invocação é possível graças à utilização de um objeto remoto padrão RMI, que permite a inclusão de interceptadores, tanto do lado do cliente quanto do servidor. Os interceptadores adicionados possuem uma referência para o objeto real do servidor e usam mecanismos de reflexão para invocar este método.

LuaProxy [MR05] é uma implementação de *smart proxies* para o *middleware* LuaOrb [CCI99]. Tais *smart proxies* são capazes de reagir, em tempo de execução, a condições inesperadas, como por exemplo, uma sobrecarga de requisições locais e remotas devido a picos momentâneos na rede. Em LuaProxy, os *stubs* são substituídos por *smart proxies* em tempo de execução, os quais são criados dinamicamente a partir de uma descrição de requisitos de QoS. O uso da linguagem Lua permitiu ainda a utilização de *downloadable stubs* (*dstubs*) e *uploadable stubs* (*ustubs*) que, combinados com LuaProxy, apresentam-se como uma alternativa para adaptação dinâmica.

Avaliação crítica: Os trabalhos mencionados nesta Seção apresentam soluções de *smart proxies* para Jini, RMI, CORBA e LuaOrb. Porém, nenhum dos trabalhos apresenta uma proposta de utilização de *smart proxies* para serviços Web.

2.4 Adaptadores de Interface

O problema de adaptação de interfaces em sistemas de objetos distribuídos já foi investigado em outros trabalhos. Vayssiere [Vay01] sugere o uso de um repositório de adaptadores para sistemas baseados em Jini. Na referida proposta, um cliente Jini que esteja interessado em iniciar uma conversação com um serviço do tipo A deve consultar o serviço de nomes de Jini. Caso este possua um *proxy* registrado do tipo A, o mesmo é retornado para o cliente. Caso contrário, o próprio serviço de nomes consulta o repositório de adaptadores, procurando por um adaptador de A para um outro tipo A'. Caso exista tal adaptador, o mesmo é retornado para o cliente.

Antonellis e colegas [AMS+06] apresentam a definição, projeto e implementação de uma arquitetura chamada VISPO (*Virtual-district Internet-based Service Plataform*), que permite a orquestração dinâmica e flexível de processos de negócios. O artigo descreve uma metodologia para determinação do grau de similaridade de duas interfaces remotas. O modelo introduz o conceito de classes de compatibilidade, que é um conjunto de serviços

semanticamente equivalentes. O objetivo é viabilizar a substituição de um servidor por outro que implemente uma interface similar. Basicamente, o projeto se divide em quatro etapas: definição dos serviços abstratos, definição dos serviços concretos, definição de compatibilidade de serviços e orquestração. No entanto, a metodologia proposta não chegou a ser validada com servidores Web reais, implementados e disponibilizados na Internet.

Ponnekanti [PF04] descreve uma taxonomia dos principais tipos de incompatibilidades que podem ocorrer em uma interface remota quando a mesma evolui de forma independente de seus clientes.

2.5 Protocolos de Invocação

Van den Bos e Laffra [BL89] definem um sistema denominado PROCOL que usa expressões regulares para definir a seqüência de métodos que um objeto pode acessar. PROCOL permite, por meio de uma análise dos requisitos semânticos, restringir o acesso a alguns métodos dependendo do estado do objeto. Esta restrição é definida pelo uso de *guardas*, que aumentam o poder das expressões regulares. No entanto, acredita-se não ser razoável incorporar ao protocolo de invocação a capacidade de detectar tal categoria de chamadas inválidas, pois para isso, há a necessidade de especificar no cliente parte da lógica da aplicação servidora. Por exemplo, para impedir que um cliente invoque um método do tipo `removeItemCarrinho`, é preciso que o cliente tenha a informação de que pelo menos um item foi inserido no carrinho. Porém, manter este controle em todas as aplicações cliente tem um alto custo.

Yellin e Strom [YS97] utilizam um autômato finito determinístico, especificando um conjunto de estados e transições, para expressar as condições de invocação e bloqueio do protocolo. Canal e colegas [CFP+03] propõem uma extensão de CORBA IDL que usa π-calculus para descrever protocolos de serviços, baseado no conceito de papéis. Papéis permitem a especificação modular do comportamento observável de objetos CORBA. Porém, nenhum dos trabalhos citados apresenta suporte a serviços Web.

2.6 Replicação de Serviços

O crescimento acelerado de usuários da Web apresenta como conseqüência direta o tráfego acentuado na rede e o aumento na carga de trabalho dos servidores. Além disso, com a queda de desempenho, causada pela freqüência de acessos a um determinado serviço,

outra conseqüência é o aumento do tempo de resposta de um servidor para uma aplicação cliente.

Replicação é a principal técnica para prover disponibilidade e melhorar o desempenho. Pela disponibilização de um mesmo serviço em um conjunto de réplicas é possível, por exemplo, fazer uso de mecanismos que aumentem a tolerância a falhas onde, se um serviço falha, outro serviço pode ser invocado. Outro ganho está relacionado ao desempenho, uma vez que é possível inserir mecanismos de balanceamento de carga, evitando que algum servidor fique sobrecarregado.

Mecanismos de replicação podem ser utilizados do lado do servidor, na forma de *proxy* ou incorporados ao cliente. Do lado do servidor algumas técnicas conhecidas são: DNS (*Domain Name Service*), CISCO *Distributed Director* e CISCO *Local Director*, além de *clusters* e descentralização de servidores [SM04]. Estas técnicas agem na resolução de problemas onde servidores individuais não suportam altas taxas de requisições, resultando em uma acentuada queda de desempenho e consequentemente elevados tempos de resposta. A utilização de *clusters*, por exemplo, possibilita suportar uma maior carga de processamento, mas não considera o problema de gargalos criados com a sobrecarga na rede. A técnica de DNS, com servidores distribuídos geograficamente, possibilita o balanceamento de carga entre servidores, porém, sem considerar a carga da rede e/ou carga individual de cada servidor.

O uso de *proxies* como ferramenta para replicação consiste em replicar informações que são acessadas constantemente, em pontos mais próximos dos clientes. Porém, no caso de serviços Web, esta técnica somente é interessante para armazenar resultados de chamadas idempotentes.

A literatura apresenta três diferentes técnicas de replicação: replicação ativa [Sch90], semi-ativa [VBB$^+$91] e passiva [BST92]. Na replicação ativa, todos as requisições são processadas por todas as réplicas. As requisições replicadas devem ser processadas na mesma ordem para garantir que todas as réplicas tenham o mesmo estado. Nesta abordagem, se uma réplica falha, as demais réplicas podem continuar provendo o serviço de forma transparente para o cliente, sem perder nenhum estado. Dependendo do nível de confiabilidade, o cliente pode retomar o processamento após a primeira resposta de qualquer uma das réplicas, após a resposta da maioria das réplicas ou somente após a resposta de todas as réplicas. Na replicação semi-ativa as requisições são difundidas entre todas as réplicas pela réplica mestre (primária), mas somente o mestre envia os resultados ao cliente. Quando o mestre falha, uma das réplicas restantes assume o seu papel. A abordagem que trata de replicação passiva aponta um dos servidores replicados como primário e outros como *backups*. Clientes submetem suas requisições para o servidor primário, que processa e

envia o estado resultante para as demais réplicas (*backups*). Nesta abordagem, somente o servidor primário executa a requisição cliente. No caso de falha no servidor primário, um dos *backups* assume como novo servidor primário. Em casos onde o primário falha antes mesmo de enviar a requisição para as réplicas, a falha não é totalmente transparente do ponto de vista do cliente, pois pode ser necessário reenviar a última requisição.

WS-Replication [SPSPMJP06] é um sistema que utiliza comunicação em grupo para suportar replicação ativa em serviços Web. WS-Replication requer que as réplicas acessadas sejam configuradas com componentes providos pelo sistema para suportar uma semântica de comunicação em grupo baseada em SOAP. Desta maneira, WS-Replication não disponibiliza transparência de replicação quando acessa provedores de serviços Web padrão, como aqueles usados nos experimentos a serem descritos no Capítulo 5.

2.7 Conclusões

Neste capítulo foram apresentados os principais padrões existentes em serviços Web. Dentre eles, destacam-se SOAP, WSDL e UDDI. Estas são as tecnologias que o sistema SmartWS, descrito nos próximos capítulos, utiliza para invocação de serviços Web. Foram apresentados também alguns trabalhos relacionados a políticas de seleção de réplicas de Serviços Web. Destaca-se o trabalho que teve como resultado o *framework* RWS [SM04] e que serviu como base para esta dissertação. Foram destacados pontos positivos e algumas deficiências encontradas no referido trabalho, para as quais serão apontadas novas propostas a partir do Capítulo 3.

Foram descritos ainda alguns trabalhos sobre *smart proxies*, adaptadores de interface, protocolos de invocação e replicação de serviços. Nenhum dos trabalhos citados apresentava uma proposta satisfatória para acesso transparente por parte de clientes a serviços Web replicados, decorrendo deste fato a proposta de desenvolvimento do sistema SmartWS.

Capítulo 3

SmartWS: Funções Básicas

Neste capítulo, são descritas as cinco políticas para seleção de réplicas implementadas por SmartWS. Como afirmado, quatro delas já existem na literatura: Estática, Aleatória, Paralela e via Melhor Mediana. Uma nova política é proposta nesta dissertação, denominada PBM (*Parallel Best Median*). O capítulo apresenta ainda outras funções suportadas por SmartWS como adaptadores de interface, protocolo de invocação e sessões de uso.

3.1 Políticas para Seleção de Servidores

3.1.1 Seleção Estática

Nesta política, cabe ao programador definir estaticamente, em tempo de implantação da aplicação, a ordem segundo a qual as réplicas de um determinado serviço serão invocadas. O primeiro serviço nesta ordem é invocado; se ele falhar, invoca-se o próximo e assim sucessivamente. Este processo se repete até que uma réplica responda com sucesso ou então até que todas as réplicas falhem, quando propaga-se uma exceção para o cliente.

Esta política é adequada para cenários onde se conhece *a priori* as características e o tempo médio de resposta de cada réplica de um serviço. Além disso, estas características não devem sofrer variações significativas ao longo do tempo. Com isso, é plenamente possível determinar e fixar em tempo de implantação do sistema a ordem com que as réplicas de um serviço devem ser invocadas. Evidentemente, deve-se invocar primeiro o servidor que reconhecidamente possui o menor tempo de resposta e assim sucessivamente. Com isso, otimiza-se o desempenho do cliente e simultaneamente acrescenta-se ao mesmo um mecanismo de tolerância a falhas (isto é, no caso do servidor mais rápido falhar, transparentemente invoca-se o próximo e assim sucessivamente).

A política de seleção estática permite alocar estaticamente clientes a suas réplicas. Por exemplo, pode-se definir que clientes de um determinado país acessarão preferencialmente os servidores deste país; em caso de eventual indisponibilidade deste servidor, pode-se definir que deve ser acessado, por exemplo, o servidor de um país vizinho.

Sintaxe: a seguir, mostra-se como se define estaticamente em SmartWS a ordem de invocação de N serviços:

```
policy
  service1 > service2 > ... > serviceN
```

Desvantagem: a principal desvantagem desta política é o fato de que cenários estáveis e bem conhecidos são mais comuns em redes corporativas do que em redes abertas e dinâmicas, como a Internet. Com isso, qualquer variação no desempenho dos servidores definidos como prioritários afeta sensivelmente o desempenho desta política.

3.1.2 Seleção Aleatória

Esta política escolhe aleatoriamente, dentre um conjunto de servidores Web, aquele que será invocado pelo cliente. Caso esta invocação seja bem sucedida, o resultado da mesma é retornado à aplicação cliente. Caso esta invocação falhe, faz-se uma nova escolha dentre as réplicas restantes. Este processo se repete até que uma réplica responda com sucesso ou então até que todas as réplicas falhem, quando se propaga uma exceção para a aplicação cliente.

Esta política é recomendada quando se sabe *a priori* que *todas* as réplicas de um determinado serviço possuem características e tempos de resposta semelhantes. Ou seja, nestes casos, não faz sentido definir qualquer ordem no acesso às mesmas. Além disso, como esta política distribui uniformemente as requisições geradas por um cliente entre as réplicas disponíveis, ela contribui para balancear a carga de processamento de tais réplicas.

Sintaxe: a seguir, mostra-se como se define aleatoriamente em SmartWS a ordem de invocação de N serviços:

```
policy
  service1 ? service2 ? ... ? serviceN
```

Desvantagem: assim como no caso da seleção estática, a principal desvantagem desta política é que cenários constituídos por servidores com capacidades e cargas de processamento semelhantes nem sempre são comuns no contexto de serviços Web. Com isso, servidores com baixo desempenho são acessados na mesma proporção que servidores com melhor desempenho.

3.1.3 Seleção Paralela

Esta política invoca concorrentemente todas as réplicas conhecidas de um determinado serviço Web. A primeira resposta obtida é retornada à aplicação cliente e as demais são descartadas. A política falha quando todas as invocações disparadas concorrentemente falharem. A política de seleção Paralela é recomendada quando a redução no tempo de resposta de invocações é um dos principais requisitos de um sistema.

Sintaxe: a seguir, mostra-se como se define em SmartWS a invocação de N serviços de forma paralela:

```
policy
  service1 | service2 | ... | serviceN
```

Desvantagens: esta política pode introduzir uma carga de processamento considerável tanto no cliente (que deve disparar múltiplas *threads* para realizar as invocações) como nos servidores disponíveis (pois cabe a todos eles processar a requisição do cliente). Além disso, esta política gera um maior tráfego na rede.

3.1.4 Seleção via Melhor Mediana

Para cada réplica de um serviço Web, esta política calcula a mediana dos tempos de resposta dentre as k-últimas invocações deste serviço. O serviço efetivamente invocado é aquele que possui a menor mediana. Se este serviço falhar, prossegue-se invocando o serviço com a segunda menor mediana e assim sucessivamente. A política falha quando falharem todas as invocações disparadas segundo a ordem das medianas.

Em geral, em amostras sujeitas a diferenças consideráveis entre o menor e o maior elemento, como é freqüente no caso do tempo de resposta de servidores Web [DRJ00], a mediana representa de forma mais adequada uma amostra do que a média. Ao contrário das políticas Estática e Aleatória, a seleção via Melhor Mediana propicia ainda uma adaptação a cenários dinâmicos, onde o tempo de resposta dos servidores consultados apresenta pequenas variações ao longo do tempo.

Servidor A			
Posição no Buffer	**Situação 1 (*ms*)**	**9 invocações depois (*ms*)**	**Mediana**
1	400	400	400
2	300	300	350
3	350	350	350
4	340	340	345
5	390	390	350
6	400	400	370
7	380	380	380
8	340	340	365
9	370	370	370
10	-	10000	375
11	-	10000	380
12	-	10000	385
13	-	10000	390
14	-	10000	395
15	-	10000	400
16	-	10000	400
17	-	10000	400
18	-	10000	5200
19	-	-	-
20	-	-	-

Tabela 3.1: Exemplo do uso de um *buffer* com 20 posições na política via Melhor Mediana

Em SmartWS, nas políticas baseadas em estatísticas, um *buffer* é armazenado em memória primária e é pelos tempos contidos neste *buffer* que é calculada a mediana dos tempos. O tamanho do *buffer* está diretamente relacionado ao tempo necessário para adaptação após variações nos tempos de resposta dos servidores. Quanto maior o tamanho do *buffer*, maior o tempo gasto para adaptação, pois maior o número de invocações necessárias para detectar alterações de desempenho nos servidores.

Para ilustrar esta situação suponha que, em um cenário com 4 servidores Web replicados, os servidores, denominados A, B, C e D em determinado momento possuem as seguintes medianas: $A = 400\ ms$, $B = 620\ ms$, $C = 500\ ms$ e $D = 800\ ms$. Neste exemplo utiliza-se um *buffer* de tamanho 20. A Tabela 3.1 ilustra as 20 posições reservadas para armazenamento dos tempos de resposta do servidor A. A coluna "Situação 1" apresenta os tempos de resposta das 9 primeiras invocações. Durante todas estas invocações, em nenhum momento o servidor A deixou de ter a menor mediana, mantendo-se como servidor preferencial. A partir da invocação 10, o servidor A apresentou uma queda de desempenho, como mostra a terceira coluna. Como o *buffer* é grande e nenhum tempo foi

descartado ainda, somente na invocação 18 o servidor A deixa de ter a menor mediana, passando assim a não mais ser invocado.

Sintaxe: a seguir, mostra-se como se define em SmartWS a ordem de invocação de N serviços de acordo com as medianas:

```
policy
  bestmedian(service1, service2, ..., serviceN)
```

Desvantagens: no caso de falhas, a capacidade de adaptação da política baseada na melhor mediana é prejudicada (mesmo com *buffer* pequeno). O motivo é que se o melhor servidor falhar, nas próximas invocações, sua mediana ainda poderá ser a menor e, portanto, ele voltará a ser selecionado. O resultado será uma degradação do tempo de resposta do serviço (já que se deve esperar a falha do primeiro servidor para só então redirecionar a chamada para o segundo).

Uma alternativa, adotada em SmartWS, consiste em aumentar de forma arbitrária o tempo de resposta de um servidor que falhou, de forma a forçar sua retirada da primeira posição da fila de servidores a serem invocados. Esta alternativa, no entanto, tem a desvantagem de prejudicar este servidor caso o mesmo se recupere e volte a apresentar tempos de resposta que o qualifiquem como possuindo a menor mediana.

Outra desvantagem é que, caso não haja histórico de invocações anteriores, a política via Melhor Mediana invocará, pelo menos uma vez, todas as réplicas. Isso acontece porque, com os tempos zerados, a melhor mediana será sempre 0 *ms*, até que todos os serviços tenham um tempo de resposta registrado no *buffer*. Uma consequência disto é a perda de desempenho caso existam servidores com desempenho bastante inferior.

3.1.5 Seleção PBM (Parallel Best Median)

A política PBM (*Parallel Best Median*), proposta nesta dissertação, combina aspectos positivos das políticas de seleção Paralela e via Melhor Mediana. As invocações são divididas em ciclos, de tamanho n, definido pelo usuário de SmartWS. Em cada ciclo, nas t primeiras invocações, a política PBM se comporta de forma idêntica à seleção Paralela. O objetivo é solucionar uma falha da política Melhor Mediana atualizando, a cada início de ciclo, o tempo de resposta de todos os servidores, inclusive servidores que, por alguma razão, caíram de rendimento ou falharam em alguma invocação e tiveram seu tempo de resposta arbitrariamente incrementado. A idéia é, ao contrário do que ocorre na seleção via Melhor Mediana, permitir que tais servidores voltem a ser selecionados caso, após uma

seqüência de falhas, os mesmos se recuperem e forneçam tempos de respostas atrativos. É importante ressaltar que para que este recurso tenha resultado, o valor de t deve ser igual a $\lceil b/2 \rceil$, onde b é o tamanho do *buffer*.

Nas demais invocações que compõem o ciclo, a política PBM invoca de forma concorrente o serviço com a menor mediana m, juntamente com os serviços cujas medianas são menores ou iguais a $m * k$, limitados a um máximo de p servidores ($k > 1$, $p \leq r$), onde r é o número de réplicas de um serviço Web. Assim, k define o grau de paralelismo da solução e p define um limite superior para este paralelismo. Se todas as invocações disparadas em paralelo falharem, dispara-se de forma seqüencial invocações aos demais servidores, segundo a ordem de suas medianas, de forma a aumentar o grau de tolerância a falhas da política. Assim, uma invocação via política PBM falha quando todas as réplicas falharem.

No caso de falha de um servidor, como na Melhor Mediana, registra-se que seu tempo de resposta é igual ao maior tempo de resposta dos demais servidores mais uma unidade de tempo. O objetivo é incrementar sua mediana e, com isso, reduzir a chance deste servidor ser novamente selecionado nas próximas invocações dentro do ciclo.

Cabe ao usuário da política PBM definir o tamanho do *buffer* e os valores dos parâmetros k, p, n, e t mencionados acima, em função dos requisitos de sua aplicação, das características do ambiente de rede e dos servidores consultados. Particularmente, se $p = 1$, a política PBM se comporta de forma similar à política da Melhor Mediana (se diferenciando pelo fato de a cada n invocações, disparar as próximas t invocações em paralelo). Por outro lado, se $p = r$ e $k = \infty$, então a política PBM tem comportamento idêntico à política paralela.

A política PBM não deve iniciar com os tempos dos servidores (no *buffer*) zerados. Em um pior caso, se na primeira invocação após as t invocações, algum servidor ainda não tivesse respondido sequer uma vez, este ainda teria como mediana o tempo de 0 *ms*. Desta forma, este servidor, que estaria com um tempo de resposta alto, seria invocado uma vez que sua mediana é a menor. A solução encontrada foi fazer com que a política PBM inicie com os tempos iguais a infinito (usou-se a constante $Integer.MAX\_VALUE$ para representar um valor infinito).

Como afirmado, a seleção PBM conjuga em uma só política as vantagens das políticas de seleção Paralela e via Melhor Mediana. Em linhas gerais, esta política deve ser usada quando se deseja conciliar tempo de resposta, balanceamento de carga e adaptação a cenários dinâmicos e sujeitos a falhas.

Sintaxe: a seguir mostra-se a sintaxe básica da política PBM. Os parâmetros definidos são, respectivamente, k, p, n e t.

```
policy
  pbm(1.2, 2, 10, 3, service1, service2, ..., serviceN)
```

3.2 Composição de Políticas

SmartWS permite ainda que as políticas sejam combinadas em tempo de compilação. É possível, por exemplo, invocar três serviços em paralelo e somente se os três servidores falharem, invoca-se um quarto servidor. O código a seguir mostra esta forma de composição:

```
policy
  (service1 | service2 | service3) >  service4
```

Outro exemplo possível é combinar a política PBM com outras políticas. No código a seguir, a política PBM, que utiliza três servidores, funcionará com o recurso de, caso os três servidores falharem, invocar em paralelo outros dois serviços. Uma situação prática para este exemplo é imaginar que os servidores 4 e 5 disponibilizam serviços de boa qualidade mas que apresentam um custo para acessá-los, enquanto os outros são gratuitos e com desempenho razoável. Portanto, a aplicação cliente só acessará os serviços 4 e 5 caso os outros três falhem.

```
policy
  pbm(1.2, 2, 10, 3, service1, service2, service3) > (service4 | service5)
```

3.3 Adaptadores de Interfaces

Adaptadores devem ser usados quando as interfaces remotas providas pelas réplicas de um serviço são funcionalmente semelhantes, mas seus tipos não são compatíveis, devido a diferenças sintáticas na assinatura dos métodos.

A fim de compatibilizar as interfaces remotas de um conjunto de servidores replicados, SmartWS se vale dos conceitos de interfaces abstratas e adaptadores de interfaces. Clientes de serviços replicados utilizam sempre interfaces abstratas, as quais têm a função de padronizar as diversas interfaces remotas providas pelas réplicas utilizadas. Um objeto adaptador deve implementar a interface abstrata e possuir uma referência para o serviço que está sendo adaptado. A implementação dos métodos da interface abstrata em um

objeto adaptador deve se responsabilizar por adaptar e converter tais métodos à sintaxe esperada pelo método da interface remota da réplica. Em SmartWS, adaptadores devem ser definidos manualmente pelos usuários do sistema, já que estes possuem conhecimento tanto da interface abstrata de um serviço como da interface remota das réplicas utilizadas no sistema.

A fim de apresentar o uso de adaptadores de interfaces, serão utilizados dois servidores Web citados no Capítulo 1, originalmente desenvolvidos com o objetivo de ilustrar o uso da tecnologia de serviços Web. Ambos servidores disponibilizam um serviço de conversão de temperatura, de graus *Celsius* para *Fahrenheit* e vice-versa.

O primeiro servidor, hospedado em um repositório de serviços Web conhecido como WebserviceX[1], disponibiliza a seguinte interface remota:

```
interface ConvertTemperatureSoap {
  int convertTemp(temp, int fromUnit, int toUnit);
}
```

O segundo servidor, hospedado na empresa de treinamento DeveloperDays[2], disponibiliza a seguinte interface remota:

```
interface ITempConverter {
   int ctoF(int t);
   int ftoC(int t);
}
```

Suponha uma aplicação cliente que utilize a seguinte interface abstrata para acessar estes dois servidores:

```
interface TemperatureConverter {
   int toFahrenheit(int tc);
   int toCelsius(int tf);
}
```

A fim de assegurar transparência de replicação, deve-se definir duas classes adaptadoras para compatibilizar a interface abstrata do cliente com as interfaces remotas dos dois servidores. O primeiro adaptador converte a interface abstrata para a interface

[1] www.webservicex.net
[2] www.developerdays.com

`ConvertTemperatureSoap` requerida pelo servidor `WebserviceX`, conforme mostrado a seguir:

```
//Adaptador de Interface
class WebserviceXAdapter implements TemperatureConverter {
   //Associa o adaptador ao servidor
   private ConvertTemperatureSoap server;
   WebserviceXAdapter (ConvertTemperatureSoap s){
     this.server= s;
   }
   //implementa os métodos da interface abstrata
   public int toFahrenheit(int tc) {
     return server.convertTemp (tc, TemperatureUnit.degreeCelsius,
      TemperatureUnit.degreeFahrenheit);
   }
   public int toCelsius(int tf) {
     return server.convertTemp (tf, TemperatureUnit.degreeFahrenheit,
      TemperatureUnit.degreeCelsius);
   }
}
```

O segundo adaptador converte a interface abstrata `TemperatureConverter` para a interface remota `ITempConverter`, conforme mostrado a seguir:

```
//Adaptador de Interface
class DeveloperDaysAdapter implements TemperatureConverter {
   //Associa o adaptador ao servidor
   private ITempConverter server;
   DeveloperDaysAdapter (ITempConverter server) {
      this.server= server;
   }
   //implementa os métodos da interface abstrata
   public int toFahrenheit(int tc) {
      return server.ctoF(tc);
   }
   public int toCelsius(int tf) {
      return server.ftoC(tf);
   }
}
```

Método Invocado	*Servidor*	*Estado*
pesqProduto	1	q0
pesqProduto	3	q0
addItemCarrinho	-	Msg Erro
login	2	q1
pesqProduto	2	q1
addItemCarrinho	2	q1
listaCarrinho	2	q1
efetuaCompra	2	q1
logout	2	q0
pesqProd	3	q0
pesqProd	4	q0
pesqProd	2	q0

Tabela 3.2: **Estudo de caso - Sessão de uso e Protocolo de invocação**

3.4 Sessões de Uso

O uso de réplicas de um serviço cria possibilidades para o desenvolvimento de mecanismos para melhorar o desempenho e aumentar a tolerância a falhas. Criar estes mecanismos é fundamental em cenários onde se aplica o paradigma de arquitetura orientada por serviços, como a Internet. Estes mecanismos atuam diretamente em características como alta latência, alta taxa de falhas e grande variação de largura de banda.

Porém, em algumas situações é necessário desabilitar o uso de políticas de seleção dinâmica de réplicas após a execução de determinadas operações. Por exemplo, requisições a métodos como `login` e `logout` são utilizados para delimitar o início e fim de uso de um sistema, destinando a uma mesma réplica todas as requisições subseqüentes ao método `login` até a execução do método `logout`.

Sessões de uso, em geral, são estabelecidas quando uma determinada operação altera o estado de uma réplica e exige que operações seguintes sejam processadas sobre o estado alterado.

Um exemplo do funcionamento de sessões de uso em SmartWS é ilustrado na Tabela 3.2. Esta tabela, além dos doze métodos acessados em um sistema de comércio eletrônico, mostra em sua 2ª coluna o servidor que respondeu a cada invocação. Como pode ser identificado, foi criada uma sessão entre a invocação do método `login` até a invocação do método `logout`, onde o servidor 2 respondeu a todas as invocações. A sintaxe para definição de sessões de uso em SmartWS é mostrada no Capítulo 4.

3.5 Protocolo de Invocação

A criação de um conjunto de regras para restringir a ordem em que as operações descritas em uma interface podem ser invocadas, permite definir seqüências legais de invocação de seus métodos. Em SmartWS, estas regras são descritas por meio de um autômato finito determinístico. As transições deste autômato representam chamadas aos métodos da interface abstrata à qual o protocolo está associado. Quando um método remoto é invocado, SmartWS verifica no autômato se existe alguma transição correspondente ao método no estado atual. Caso não haja nenhuma transição correspondente, então a chamada é inválida. Neste caso, SmartWS propaga uma exceção para a aplicação cliente informando que a chamada não deveria ter sido solicitada no estado atual.

A Figura 3.1 descreve um autômato que modela o protocolo de invocação de uma aplicação de comércio eletrônico. Este autômato expressa uma regra de uso do sistema segundo a qual determinadas operações, tais como adicionar e remover produtos do carrinho e efetuar compras, somente podem ser solicitadas caso o usuário esteja "logado" no sistema. A 3ª coluna da Tabela 3.2 ilustra o comportamento de SmartWS em uma sequência de invocações. No exemplo são utilizados quatro servidores replicados e o protocolo de invocação é aquele definido na Figura 3.1.

É importante ressaltar que o objetivo de um protocolo de invocação é tão somente capturar seqüências de chamadas que são inválidas por construção, isto é, sem que seja necessário analisar requisitos semânticos do sistema alvo da invocação. Em outras palavras, existem seqüências de invocações que violam requisitos funcionais de uma aplicação e que não são detectadas por meio de um protocolo de invocação. Por exemplo, se um usuário tentar efetuar uma compra sem ter adicionado nenhum produto em seu carrinho de compras. Certamente, não é razoável dotar um protocolo de invocação de poder de expressão suficiente para detectar tal categoria de chamadas inválidas, pois isso demandaria especificar no cliente parte significativa da lógica da aplicação servidora.

3.6 Timeouts

Um dos principais problemas encontrados quando se opta por utilizar soluções distribuídas, baseadas nas tecnologias de serviços Web, é desempenho. Nem sempre o usuário de uma aplicação cliente pode esperar, por tempo indeterminado, pela resposta de uma das réplicas. Baseado nisso, SmartWS possui suporte a definição de um tempo máximo para a resposta (*timeout*). O tempo máximo no qual o cliente está disposto a esperar é considerado a partir da invocação até o momento em que a aplicação cliente recebe a

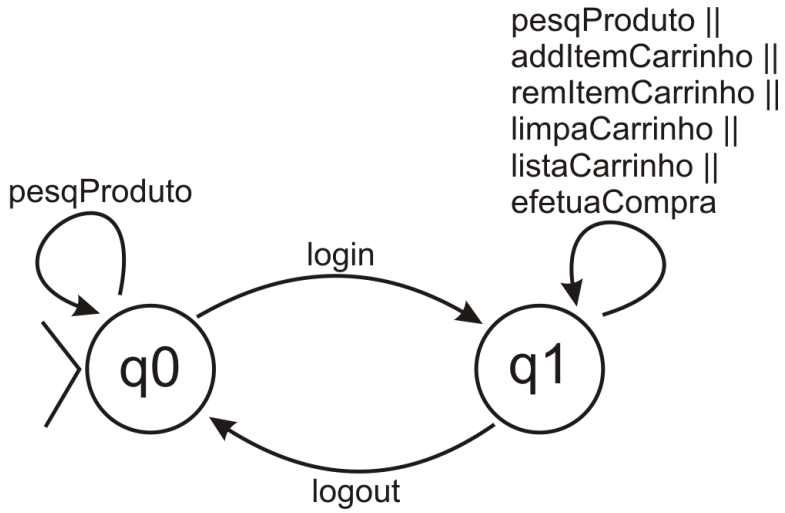

Figura 3.1: Protocolo de invocação de um sistema de comércio eletrônico

resposta.

A seguir segue um exemplo da sintaxe para definição de *timeouts*. O tempo é definido em *ms*.

```
timeout = 2000
```

***Timeouts* em sessões de uso:** O mecanismo que suporta sessões de uso também implementa *timeouts*. Se durante uma sessão de uso, o cliente perde a conexão, SmartWS irá tentar, antes do *timeout*, restabelecer a conexão por x vezes, com um intervalo de t *ms* entre as tentativas. Os valos de x e t são definidos pelo usuário de SmartWS por meio dos parâmetros *retry* e *interval*, respectivamente, como mostra o código a seguir.

```
session
    begin = login
    finish = logout
    retry = 5
    interval = 300
```

3.7 Conclusões

Neste Capítulo foram descritos o funcionamento e a sintaxe de definição de cada uma das políticas suportadas por SmartWS. Estas políticas são: Aleatória, Estática, Paralela e via Melhor Mediana, já existentes na literatura e a política proposta nesta

dissertação, denominada PBM (*Parallel Best Median*). A política PBM (*Parallel Best Median*) congrega características das políticas Paralela e via Melhor Mediana, além de mecanismos de tolerância a falhas.

Foram apresentados ainda alguns problemas existentes na política Mediana, como o fato de não conseguir se adaptar a um cenário onde ocorrem falhas ou mudanças bruscas nos tempos de resposta dos servidores envolvidos. Outra característica demonstrada neste Capítulo foi a capacidade de SmartWS combinar as políticas suportadas.

SmartWS faz uso de adaptadores de interface com o objetivo de compatibilizar interfaces remotas de um conjunto de servidores replicados com a interface abstrata utilizada pelo cliente de um serviço Web. No sistema proposto estes adaptadores devem ser implementados manualmente, uma vez que é preciso ter conhecimento tanto dos métodos da interface abstrata quanto da interface concreta.

Recursos como estabelecimento de sessões e definição de um protocolo de invocações de métodos auxiliam na manutenção de consistência de dados da réplica acessada e na economia de recursos de comunicação. Foi demonstrado neste Capítulo, por meio de um exemplo, o funcionamento destas funções em SmartWS.

No próximo capítulo, descreve-se a implementação de SmartWS, incluindo as principais classes do sistema.

Capítulo 4

Interface de Programação

4.1 Arquitetura do Sistema

Em SmartWS, *proxies* especiais, chamados *smart proxies*, são encarregados de implementar as funcionalidades básicas descritas no Capítulo 3. Basicamente, *smart proxies* são objetos que situam-se entre o cliente de um serviço Web e o *stub* disponibilizado pelo sistema de *middleware* para prover acesso remoto a este serviço. Assim, toda chamada remota de método realizada via SmartWS é capturada pelo *smart proxy* gerado pelo sistema como mostra a Figura 4.1. Cabe a este *proxy* selecionar uma réplica do serviço remoto que está sendo requisitado pelo cliente, possivelmente invocar um adaptador para compatibilizar a interface abstrata deste cliente com a interface remota do servidor selecionado e então destinar a chamada ao *stub* do sistema de *middleware* subjacente.

Etapas de Geração e Execução: Em SmartWS, *smart proxies* são gerados automaticamente a partir de uma linguagem de domínio específico, por meio da qual o programador da aplicação cliente define a interface abstrata, os serviços replicados, a política de invocação, os adaptadores (se necessários), entre outros. O protótipo desenvolvido como prova de conceito foi implementado em Java.

O uso de SmartWS começa com a definição do arquivo de configuração, utilizado para gerar o *smart proxy*. Este arquivo deve ser definido pelo programador SmartWS, uma vez que este precisa conhecer as características dos serviços Web replicados a serem acessados. Além deste arquivo, o programador SmartWS deve criar uma interface abstrata, que será implementada pelo *smart proxy*, definindo assim a comunicação com o cliente. Quando necessário, deve-se também implementar manualmente os adaptadores de interfaces. Em seguida, SmartWS gera um conjunto de arquivos Java que compõem o *smart proxy*. Uma vez gerado, o *smart proxy* deve ser instanciado pela aplicação cliente que realizará todas

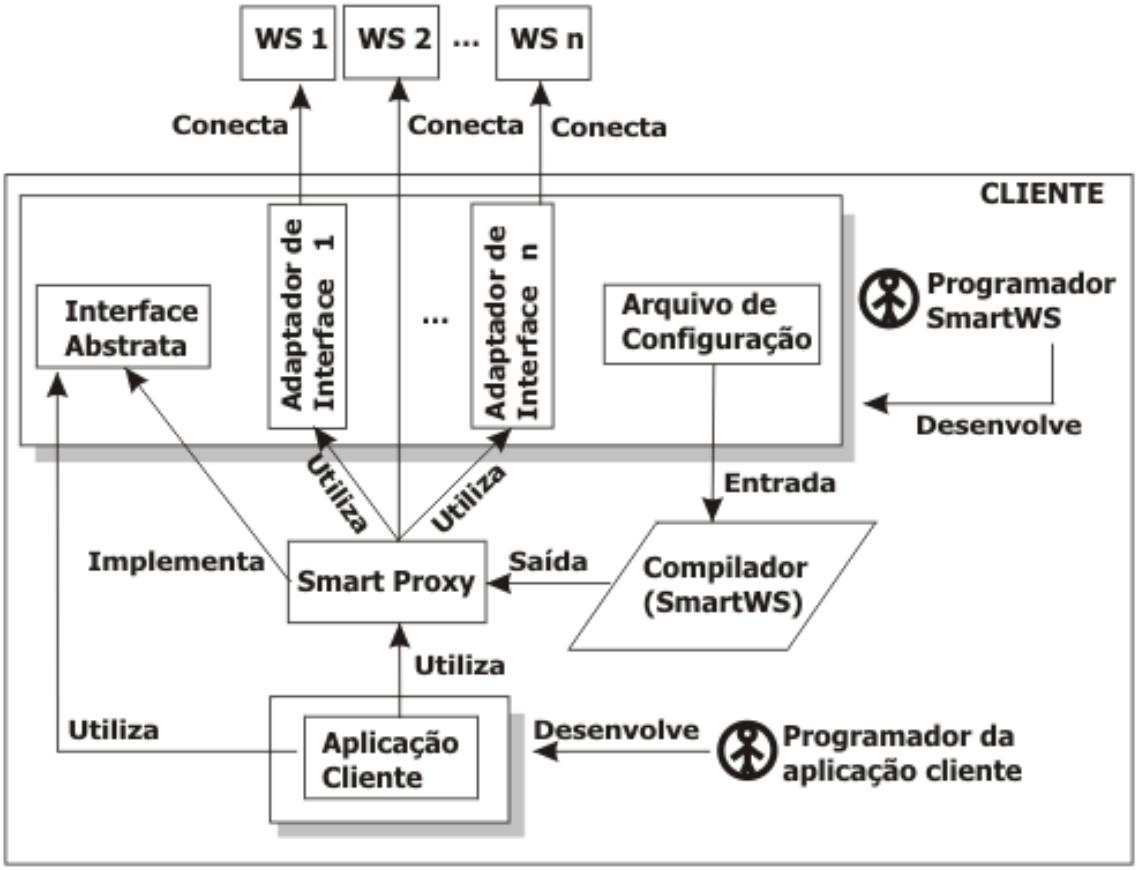

Figura 4.1: Arquitetura do Sistema SmartWS

as invocações aos serviços Web através dele.

Conhecida a interface abstrata que será implementada pelo *smart proxy*, a aplicação cliente poderá ser desenvolvida por outro programador que cuidará exclusivamente da lógica do negócio. A aplicação deverá conter um objeto do tipo da interface abstrata, referenciando o *smart proxy*. Todas as invocações aos serviços Web deverão ser direcionadas a este objeto, que implementa os métodos da interface abstrata. Ao ser chamado, o *smart proxy* escolhe, segundo a política de seleção definida, a réplica do serviço. São avaliados também, quando pertinentes, fatores relacionados ao protocolo de invocação de métodos e às sessões de uso. Por fim, são realizadas as chamadas aos servidores utilizando, se necessário, adaptadores de interface.

Para demonstrar as fases de geração e execução do *smart proxy* será utilizada como exemplo a política PBM (*Parallel Best Median*). A política PBM contém a estrutura mais complexa dentre as políticas suportadas por SmartWS, utilizando as principais classes envolvidas no processo.

A Figura 4.2 apresenta as etapas que compõem a geração dos *smart proxies*. O pro-

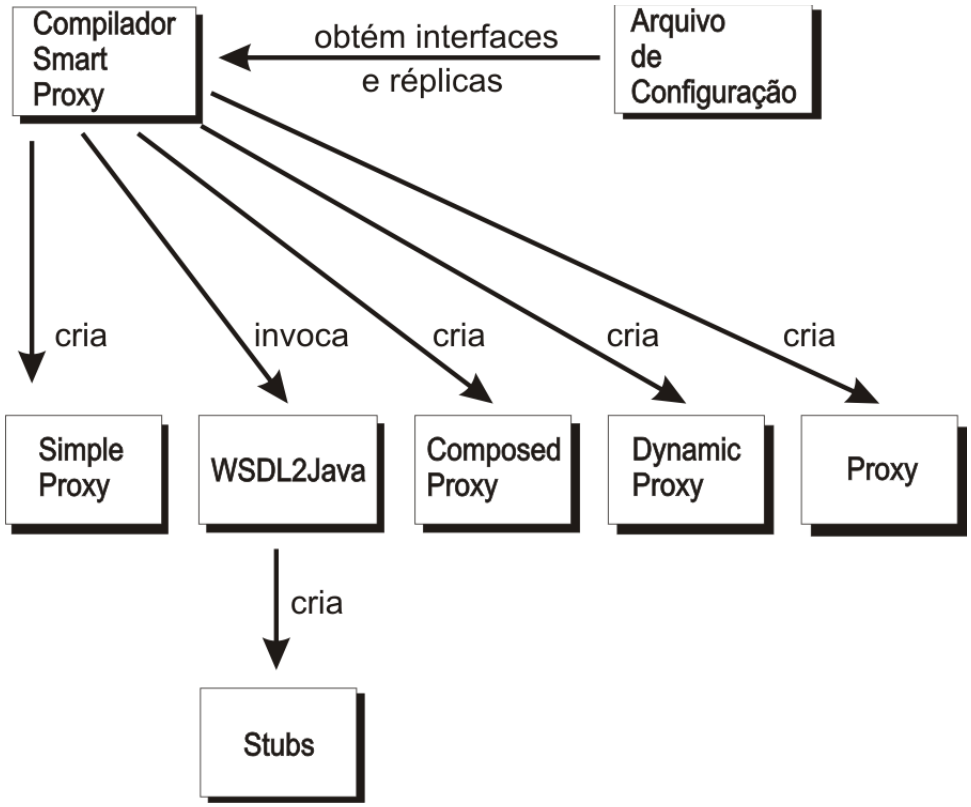

Figura 4.2: Etapas de geração dos *smart proxies*

cesso se inicia com o Compilador *Smart Proxy*, denominado *spc* (*smart proxy compiler*), acessando o arquivo de configuração para obter o nome da interface e as informações necessárias para gerar a estrutura de acesso. Os endereços dos arquivos WSDL dos servidores replicados também são obtidos no arquivo de configuração para que, utilizando uma ferramenta do Apache Axis, chamada WSDL2Java, sejam gerados os *stubs*. As primeiras classes geradas são as *SimpleProxy*, que contêm as estruturas de acesso aos serviços. Cada uma destas classes é responsável por acessar um serviço Web específico. Em seguida, são geradas as duas classes responsáveis pela implementação das políticas de acesso. A classe *ComposedProxy* é utilizada na árvore que descreve a política selecionada pelo usuário. Esta árvore é gerada utilizando o padrão de projeto *Interpretador* [GHJV95]. A classe *DynamicProxy* conterá as regras de acesso da política PBM e o controle do uso de sessões (descrito no Capítulo 3). A criação da classe principal (cujo nome é passado por parâmetro pelo usuário e será chamada simplesmente de *proxy*) acontece durante todo o processo, uma vez que depende de informações distribuídas por todas as fases.

O padrão *Interpretador* descreve uma forma de representar sentenças de uma linguagem e interpretá-las. Um grande benefício deste padrão é a possibilidade de adicionar novas estruturas à gramática de forma simples. Um exemplo de uma árvore gerada pelo

padrão *Interpretador* pode ser visto na Figura 4.3. A árvore mostrada representa as regras de invocação onde dois serviços (S1 e S2) são invocados em paralelo e, no caso de ambos falharem, outro serviço (S3) é invocado seqüencialmente. Uma representação sintática seria $(S1|S2) > S3$.

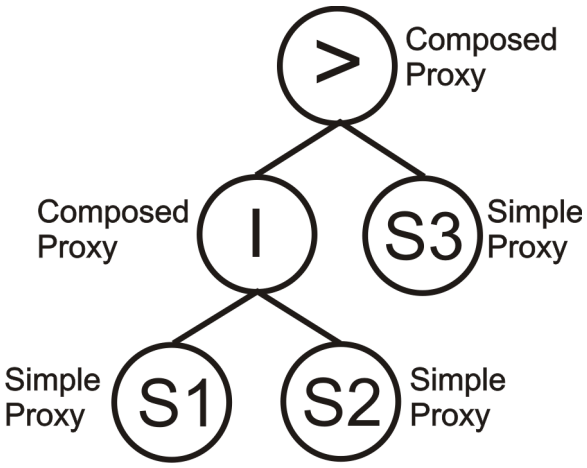

Figura 4.3: Árvore gerada para a invocação $(S1|S2) > S3$

SmartWS gera também uma classe que especifica todas as configurações do *smart proxy* (incluindo a instanciação da árvore que define a política). Um método de nome `getInstance` é implementado, sendo este invocado pela aplicação cliente que deseja acessar os serviços Web. Ao realizar esta chamada, a aplicação cliente recebe uma instância da árvore. Na Seção 4.3, mostra-se um exemplo de implementação de uma aplicação cliente.

A árvore possui implementados todos os métodos da interface abstrata utilizada pela aplicação cliente, podendo portanto, ser acessada pela aplicação. A estrutura interna desta árvore está diretamente ligada à política de seleção definida pelo programador de SmartWS, uma vez que objetos do tipo *DynamicProxy* são responsáveis por políticas dinâmicas e recursos como sessões de uso, enquanto o *ComposedProxy* implementa políticas estáticas e estatísticas. Como mostra a Figura 4.3, as folhas são objetos do tipo *SimpleProxy*, responsáveis por recursos como protocolo de invocação, geração de histórico e acesso ao servidor.

A Figura 4.4 ilustra os passos necessários para efetuar uma chamada remota, usando um diagrama de seqüência. Neste exemplo, utiliza-se a política PBM e um adaptador de interface. No exemplo, o serviço acessado faz a conversão de temperatura e também é definido um *timeout*. O registro do tempo de resposta no *buffer* é realizado pela classe *SimpleProxy*.

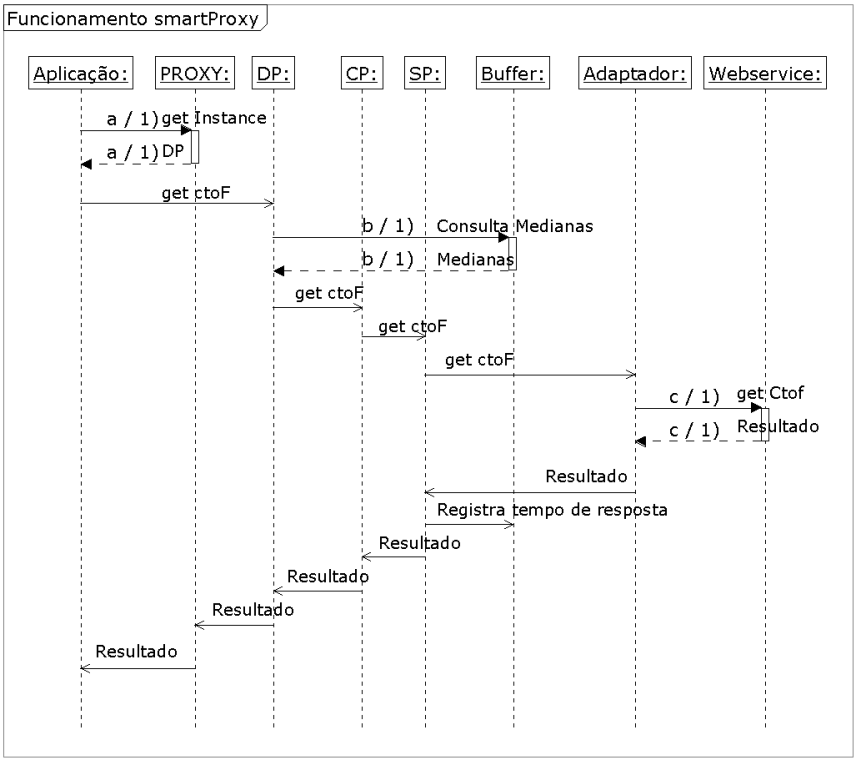

Figura 4.4: Diagrama de seqüência das etapas de funcionamento do SmartWS

4.2 Linguagem de Especificação

Em sistemas de *middleware* para desenvolvimento de aplicações baseadas na tecnologia de serviços Web, *stubs* são gerados automaticamente, a partir de uma especificação em WSDL. Da mesma forma, em SmartWS, o código de *smart proxies* também é criado de forma automática, a partir de uma especificação realizada pelo programador SmartWS.

O código a seguir apresenta um exemplo da sintaxe da linguagem de especificação utilizada nos experimentos a serem descritos no Capítulo 5. No código apresentado, a política utilizada é a PBM. A especificação consiste nas seguintes cláusulas: `interface`, `webservice`, `buffer`, `timeout` e `policy`. Uma descrição completa da gramática da linguagem é apresentada no Apêndice A.

```
interface
   name= EchoInt
webservice
     alias= service1
     uri= http://200.154.100.23/echo.asmx?WSDL
```

```
     adapter = terra1
webservice
     alias= service2
     uri= http://200.154.100.232/echo.asmx?WSDL
     adapter= terra2
webservice
     alias= service3
     uri= http://201.87.225.6/echo.asmx?WSDL
     adapter= empresabh1
webservice
     alias= service4
     uri= http://200.214.173.151/MPVRisk/echo.asmx?WSDL
     adapter= empresabh2
webservice
     alias= service5
     uri= http://200.131.41.132:8180/axis/Echo.jws?wsdl
webservice
     alias= service6
     uri= http://143.106.7.180/axis/Echo.jws?wsdl
buffer
     size= 5
     refresh= 20
timeout= 20000
policy
     pbm(1.2, 2, 10, 3, service1, service2, service3, service4,
     service5, service6)
end
```

Por meio da cláusula `interface` define-se o nome da interface abstrata que padroniza a comunicação entre a aplicação cliente e o *smart proxy*. O item `name` denota o nome da interface Java contendo a definição de uma interface abstrata.

A cláusula `webservice` é usada para descrever os servidores que são acessados pela aplicação cliente. Para cada servidor, deve-se informar os seguintes valores: um *alias* para o servidor, a URI do arquivo contendo a especificação WSDL deste servidor, e opcionalmente, o nome da classe adaptadora da interface abstrata para a interface implementada pelo servidor. Normalmente, um arquivo de especificação de *smart proxies* possui mais de um elemento `webservice` (como pode ser visto no exemplo mostrado).

A cláusula `buffer` define o tamanho do *buffer* utilizado pelas políticas baseadas em estatísticas. Além disso, nesta cláusula o usuário de SmartWS também define a freqüência com que os tempos das invocações realizadas serão persistidos em disco.

A cláusula `timeout` define o tempo máximo que a aplicação cliente irá esperar até obter uma resposta para uma requisição. Este tempo máximo engloba todas as tentativas de restabelecer a conexão caso o sistema utilize sessões de uso. O tempo é definido em *ms*.

A cláusula `policy` especifica a política de seleção de réplicas que deve ser implementada pelo *smart proxy* a ser gerado, usando-se para isso as seguintes palavras-chave: `static`, `random`, `parallel`, `bestmedian` ou `pbm`. A política escolhida é aplicada sobre todos os servidores especificados nas cláusulas `webservice` definidas no arquivo de especificação. O exemplo seguinte ilustra a sintaxe utilizada para selecionar a política PBM:

```
policy
    pbm(1.2, 2, 10, 3, service1, service2, service3, service4, service5,
    service6)
```

Neste exemplo os parâmetros são correspondentes, respectivamente, a k, p, n e t, onde:

- k: grau de paralelismo da solução, isto é, percentual máximo entre a melhor e a pior mediana a serem invocadas, no ciclo, após as t primeiras invocações.
- p: limite superior para o paralelismo, isto é, número máximo de servidores que serão invocados em paralelo após as t primeiras invocações de cada ciclo;
- n: tamanho do ciclo;
- t: número de invocações, em paralelo, para todas as réplicas no início de cada ciclo.

O código a seguir mostra um outro exemplo onde são definidas sessões de uso e um protocolo de invocação de métodos.

```
interface
     ...
webservice1
     ...
webserviceN
     ...
buffer
     ...
```

```
session
    begin= login
    finish= logout
    retry= 5
    interval= 300
protocol
    methods= pesqProduto, login, logout, addItemCarrinho,
    remItemCarrinho, efetuaCompra, limpaCarrinho, listaCarrinho
    states= q0, q1
    transitions=
     q0, pesqProduto, q0;
     q0, login, q1;
     q1, addItemCarrinho || remItemCarrinho || efetuaCompra ||
     limpaCarrinho || listaCarrinho, q1;
     q1, logout, q0;
timeout = 20000
policy
    ...
end
```

A cláusula `session` especifica os métodos da interface abstrata que delimitam uma sessão de uso. Neste exemplo, os métodos `login` e `logout`, respectivamente, iniciam e finalizam uma sessão de uso de uma réplica.

Na cláusula `protocol` especifica-se o protocolo de invocação que determina a ordem em que devem ser chamados os métodos da interface abstrata.

4.3 Implementação

Em SmartWS, *stubs* são gerados a partir da especificação WSDL de cada serviço e para isso usa-se uma ferramenta do sistema Apache Axis, chamada WSDL2Java. Para construir os *stubs* é necessário criar um serviço específico e obter uma porta (ou um *proxy*) para o serviço criado. Por meio de uma análise sintática no documento WSDL de cada serviço são encontradas as informações necessárias para a construção dos *stubs*. O código a seguir apresenta a interface WSDL de um dos serviços Web utilizados nos experimentos.

```
<wsdl:definitions targetNamespace="http://200.131.41.132:8180/axis/Echo.jws">
```

```
...
  <wsdl:types>
...
  </wsdl:types>
<wsdl:message name="echoResponse">
  ...
  </wsdl:message>
<wsdl:message name="echoRequest">
  ...
  </wsdl:message>
  <wsdl:portType name="Echo">
  ...
  </wsdl:portType>
  <wsdl:binding name="EchoSoapBinding" type="impl:Echo">
  ...
  </wsdl:binding>
  <wsdl:service name="EchoService">
   <wsdl:port binding="impl:EchoSoapBinding" name="Echo">
   <wsdlsoap:address location="http://200.131.41.132:8180/axis/Echo.jws"/>
   </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

A utilização da ferramenta WSDL2Java para criação dos *stubs* a partir de uma interface WSDL contribuiu para simplificar a implementação de SmartWS. Porém, algumas referências devem ser adicionadas nas classes que farão as invocações ao serviço. As informações necessárias para criar um serviço são encontradas na *tag* <`service name=`“ ”>. Já as informações necessárias para obter uma porta para o serviço criado são encontradas nas *tags* <`portType name=`“ ” > e <`port name=`“ ” >. Como não existe uma padronização na descrição WSDL, outra forma comum é encontrar as *tags* precedidas por “wsdl:”, como por exemplo <`wsdl:service name =`“ ” >.

Para criar uma instância do *smart proxy* gerado, deve-se declarar um objeto do tipo da interface abstrata indicada no arquivo de configuração, atribuindo a ele o objeto retornado pelo método `getInstance`. Este objeto implementa todos os métodos da interface abstrata. O código a seguir apresenta um exemplo de implementação de um cliente que fará 100 invocações a um serviço de conversão de temperatura. É possível observar no código que a chamada do método remoto ocorre da mesma forma que se invoca um objeto

local, simplificando a implementação da aplicação cliente.

```
public class cliente {
  public static void main(String[] args){
    //Instancia o objeto do tipo PROXY
    convTemp teste = PROXY.getInstance();
    for(int x=1; x<=100; x++) {
      try {
         teste.ctoF(100);
      }
      catch (WSFailureInvocation e) {
        ...
      }
      catch (ProtocolFailure e) {
        ...
      }
      catch (Exception e) {
        ...
      }
    }
  }
}
```

4.4 Conclusões

Este capítulo apresentou detalhes da implementação de SmartWS, incluindo a etapa de geração dos *smart proxies* e o funcionamento das principais classes do sistema. Após a definição das cláusulas do arquivo de configuração, em uma linguagem de alto nível, SmartWS gera os *smart proxies* de forma automática, ficando a cargo do desenvolvedor da aplicação somente detalhes da lógica de negócio. Ao programador de SmartWS cabe, além da definição do arquivo de configuração, a criação da interface abstrata e dos adaptadores de interface.

O protótipo desenvolvido em Java, que serviu como prova de conceito, foi utilizado nos experimentos que serão descritos no Capítulo a seguir.

Capítulo 5

Resultados Experimentais

Neste Capítulo são descritos dois cenários onde foram realizados experimentos com o intuito de validar o projeto de SmartWS e formular diretrizes para escolha da política de seleção de réplicas mais adequada a uma determinada aplicação cliente de serviços *Web*. Os resultados dos experimentos foram armazenados em arquivos e analisados posteriormente, possibilitando obter conclusões sobre as políticas de seleção de réplicas utilizadas.

5.1 Cenário 1

5.1.1 Estrutura

O primeiro conjunto de experimentos teve como objetivo principal fornecer números preliminares sobre o desempenho das políticas para que fosse possível validar a implementação de SmartWS. Neste cenário, utilizou-se como exemplo um serviço de conversão de temperatura (de graus *Celsius* para *Fahrenheit* e vice-versa). O primeiro serviço utilizado encontrava-se hospedado em um repositório de serviços Web conhecido como WebserviceX[1] e o segundo serviço na empresa de treinamento DeveloperDays[2]. Foi implementado um cliente que solicita a conversão de 100 temperaturas de grau *Celsius* para *Fahrenheit*. Este cliente foi executado em uma máquina Pentium IV 3 Ghz, com 1 GB RAM, Microsoft Windows 2000, JDK 5.0 e interface de rede FastEthernet 100 Mbps.

A seguir, descrevem-se os quatro experimentos realizados com este cliente, cada um deles correspondendo a uma configuração diferente de servidores replicados.

[1] `www.webservicex.net`
[2] `www.developerdays.com`

Experimento 1

Neste primeiro experimento foram utilizados três servidores: WebserviceX, DeveloperDays e um servidor local instalado no laboratório de Computação Distribuída da PUC Minas, possuindo a seguinte configuração: AMD Athlon 2600+, 320 MB RAM, Microsoft Windows 2000, JDK 5.0, Apache Axis 1.3 e Apache Tomcat 5.5.9. Conhecia-se previamente que o tempo de resposta do servidor local era sempre inferior aos tempos dos servidores remotos.

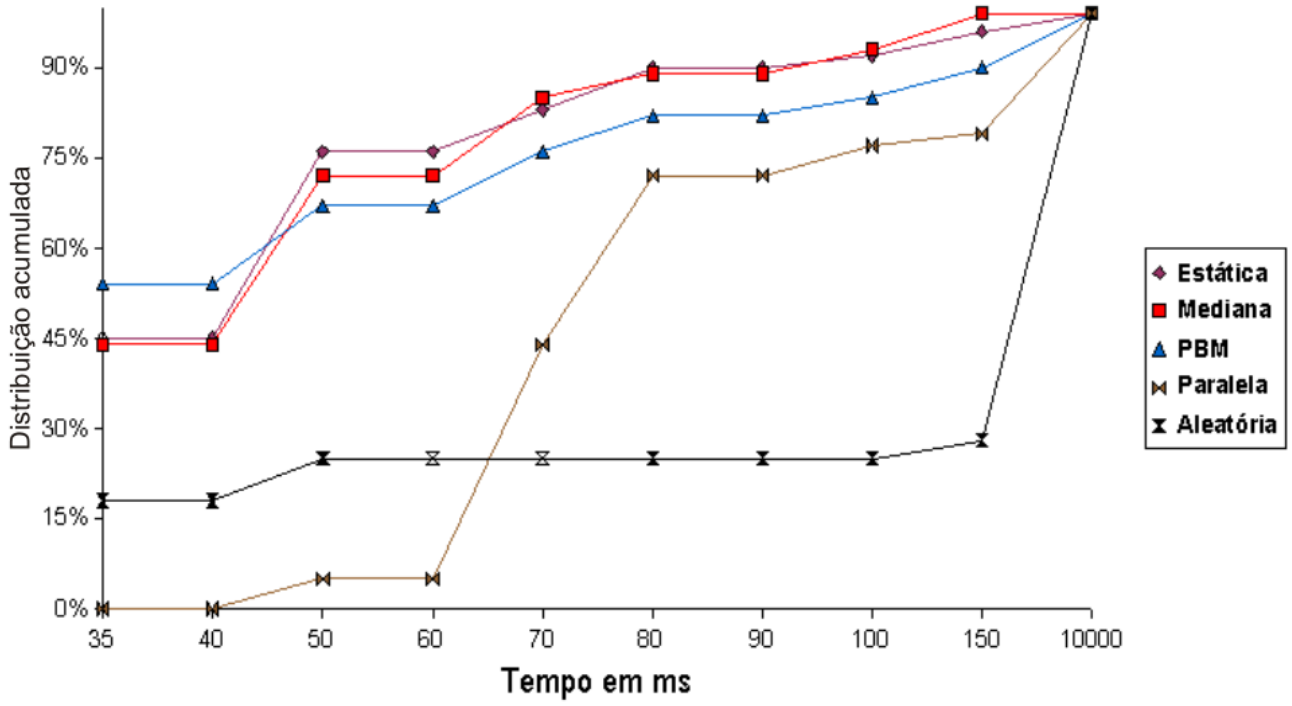

Figura 5.1: Distribuição acumulada do Experimento 1 (Cenário 1)

Para identificar as políticas com melhor desempenho durante cada experimento, com base nos tempos de resposta observados, foi utilizada uma função de distribuição acumulada [DC02]. Este tipo de função permite conduzir uma avaliação mais criteriosa de qual política apresenta os melhores tempos de respostas em dado nível, entre os vários intervalos de tempos observados. A função de distribuição acumulada é definida por meio de coordenadas (x,y), onde $y\%$ dos resultados apresentam tempos menores ou iguais a x *ms*.

O gráfico da Figura 5.1 apresenta a distribuição acumulada dos tempos de respostas obtidos para cada política. É possível perceber que as políticas Estática e Mediana apresentaram um desempenho semelhante, obtendo cerca de 75% dos tempos de resposta menores ou iguais a 50 *ms*. A política PBM, com desempenho pouco inferior só alcançou

os mesmos 75% com tempos menores ou iguais a 70 *ms*. Em todos os experimentos do Cenário 1 foram utilizados os seguintes parâmetros de configuração para a política de seleção PBM: $k = 1.2$, $p = 2$, $n = 10$ e $t = 3$.

A política Paralela, devido ao *overhead* gerado na aplicação cliente pela criação de múltiplas *threads*, apresentou um desempenho inferior, superando somente a política de seleção Aleatória. A política Aleatória, pelo fato de acessar várias vezes os servidores externos, apresentou o pior desempenho.

Experimento 2

No segundo experimento foram utilizados três servidores locais idênticos ao servidor local descrito no Experimento 1, instalados no laboratório de Computação Distribuída da PUC Minas. Esses servidores foram iniciados sem nenhuma carga extra de processamento, além daquela gerada pelo experimento.

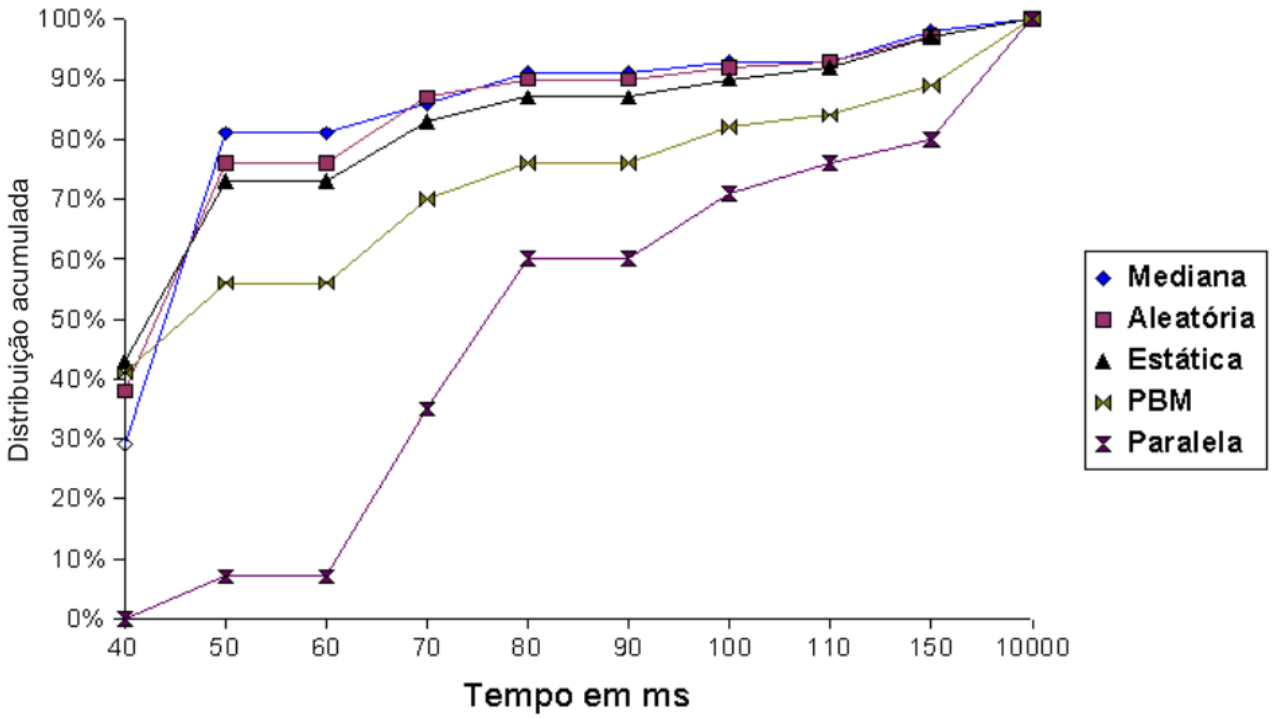

Figura 5.2: Distribuição acumulada do Experimento 2 (Cenário 1)

O gráfico da Figura 5.2 apresenta a distribuição acumulada dos tempos de respostas obtidos para cada política no cliente. Como pode ser observado, as políticas Melhor Mediana, Estática e Aleatória apresentaram desempenhos semelhantes, obtendo cerca de 90% dos tempos menores ou iguais a 80 *ms*. As políticas PBM e Paralela apresentaram

desempenho bastante inferior (certamente devido ao *overhead* gerado na aplicação cliente devido à criação de múltiplas *threads* e ao aumento do tráfego na rede).

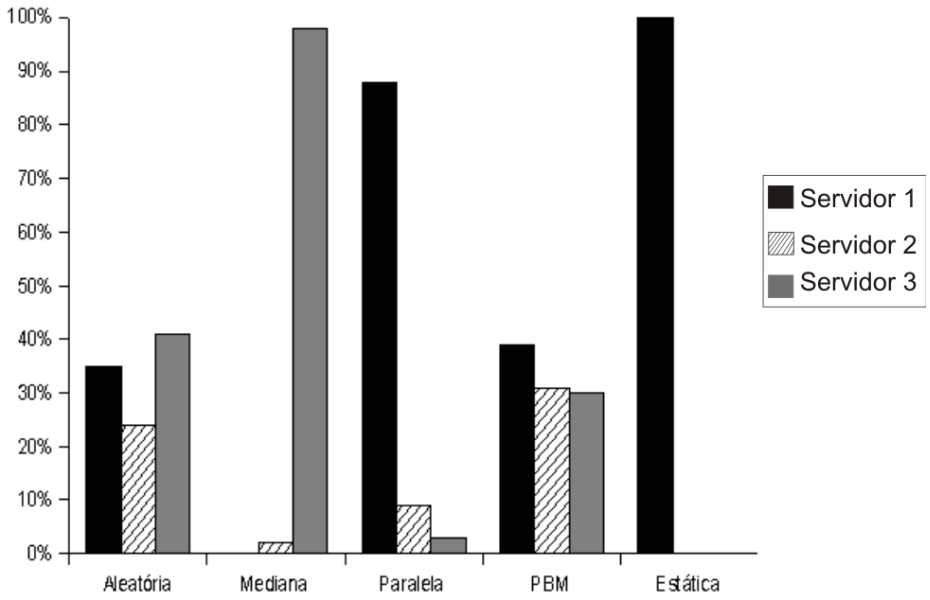

Figura 5.3: Distribuição de requisições do Experimento 2 (Cenário 1)

O gráfico da Figura 5.3 apresenta o número de chamadas que foram endereçadas a cada um dos três servidores do experimento. Como se pode observar, as políticas aleatória e PBM foram as que apresentaram a melhor distribuição de servidores. Assim, combinando os resultados apresentados nos gráficos das Figuras 5.2 e 5.3, pode-se afirmar que a política Aleatória foi a que apresentou o melhor comportamento neste segundo experimento.

Experimento 3

No terceiro experimento, foram utilizados três servidores remotos: WebserviceX, DeveloperDays e um servidor instalado no CEFET-MG. Sendo estes servidores remotos, durante o período de desenvolvimento e condução do experimento, o tempo de resposta dos mesmos variou de forma considerável, dependendo do dia da semana e do horário de acesso. Porém, a experiência prévia de uso dos três servidores permitia afirmar que os mesmos apresentavam sempre uma baixa taxa de falhas.

É possível observar no gráfico da Figura 5.4 que durante todo o experimento a política Mediana apresentou o melhor desempenho. Neste experimento, onde a maioria dos tempos de resposta ficaram entre 400 e 450 *ms*, as políticas PBM e Paralela apresentaram 90% dos seus tempos de resposta menores que 550 *ms*. A política Aleatória, certamente por acessar, na mesma proporção, os serviços com melhores e piores desempenhos, apresentou

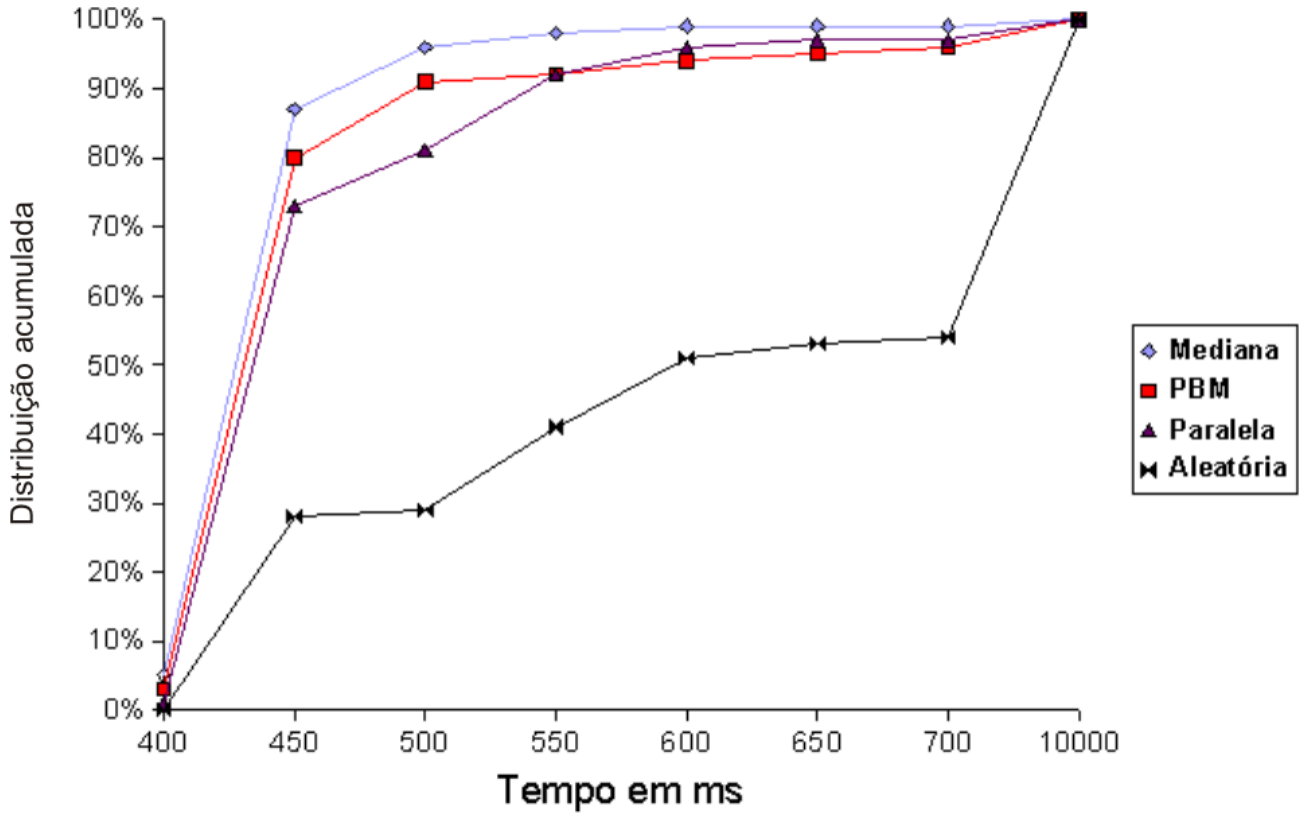

Figura 5.4: Distribuição acumulada do Experimento 3 (Cenário 1)

os maiores tempos de resposta.

Experimento 4

No quarto experimento, foram utilizados dois servidores que ficaram disponíveis durante todo o experimento (WebserviceX e DeveloperDays) e um servidor local que ficou indisponível na primeira metade do teste (isto é, este servidor local falhou nas primeiras cinqüenta invocações e respondeu com sucesso às cinqüenta invocações finais). O servidor local tinha a mesma configuração de *hardware* e *software* dos servidores descritos no Experimento 1.

O gráfico da Figura 5.5 apresenta a distribuição acumulada dos tempos de respostas obtidos para cada política no cliente. A política PBM apresentou mais de 30% dos tempos de resposta menores ou iguais a 60 *ms*. Por bastante tempo, as políticas PBM e Paralela apresentaram um desempenho bastante semelhante, obtendo cerca de 45% das respostas com tempos menores que 100 *ms*. Este bom desempenho se justifica pelo fato de que as políticas PBM e paralela detectaram imediatamente a volta do servidor local, passando a

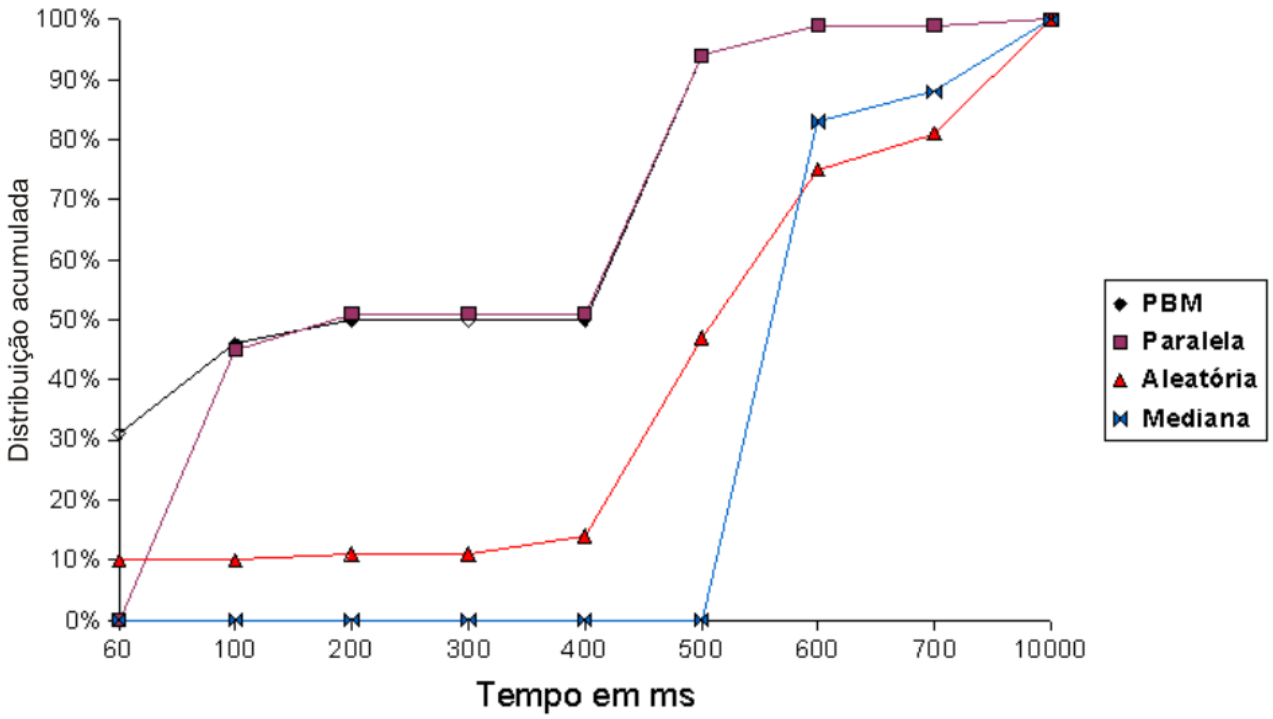

Figura 5.5: Distribuição acumulada do Experimento 4 (Cenário 1)

utilizá-lo sempre, como mostra o gráfico da Figura 5.6. Este gráfico apresenta o número de chamadas que foram endereçadas ao servidor local na segunda metade do experimento (quando este servidor passou a estar ativo).

Por outro lado, a política de seleção via Melhor Mediana não foi capaz de detectar o restabelecimento do servidor local, pelos motivos descritos na Seção 3.1.4. Com isso, a Mediana apresentou, neste experimento, o pior desempenho entre as políticas, perdendo até mesmo para a Aleatória.

Nos Experimentos 3 e 4, a política de seleção estática não foi avaliada, uma vez que o comportamento dos servidores (tempo de resposta) variava constantemente. Desta forma, não foi possível definir estaticamente uma seqüência de acesso aos mesmos.

5.2 Cenário 2

Neste segundo cenário, foram realizados três experimentos com quatro dias de duração cada (de segunda a quinta-feira). Nos experimentos, um cliente de um serviço Web replicado fez invocações periódicas durante 24 horas por dia, fazendo uso alternado de cinco políticas (Aleatória, Paralela, Mediana, além de 2 variações da política PBM). A política de seleção estática não foi avaliada nos experimentos uma vez que, como o tempo

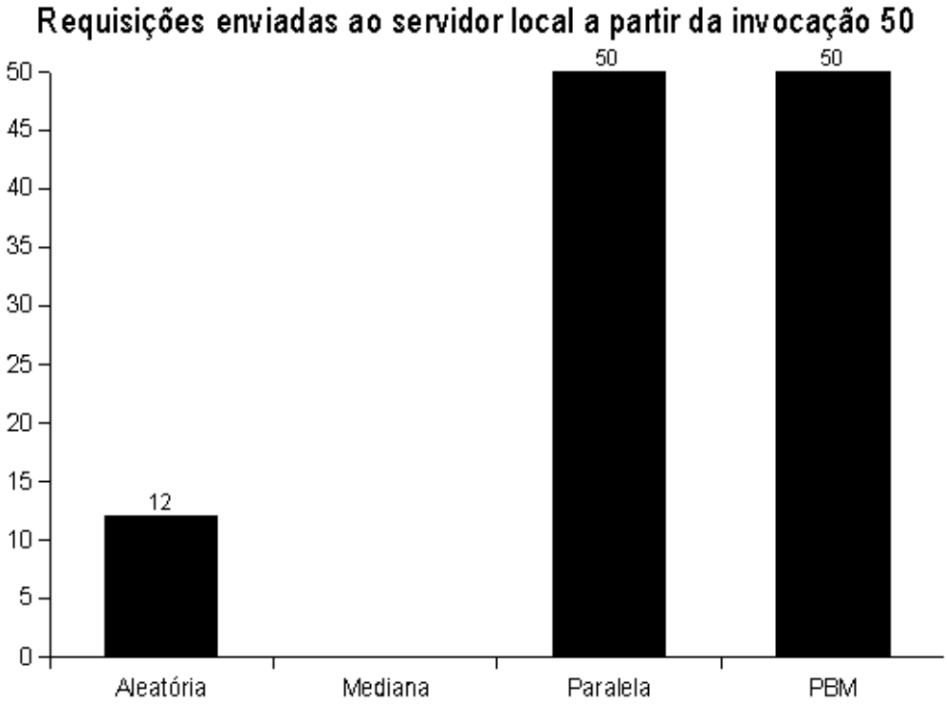

Figura 5.6: Distribuição de requisições do Experimento 4 (Cenário 1)

de resposta dos servidores variava constantemente, não foi possível definir estaticamente uma seqüência de acesso aos mesmos. Foram utilizados três tamanhos de mensagens 8 KB, 4 KB e 1 KB. Em cada semana utilizou-se um tamanho de mensagem.

5.2.1 Aplicação Cliente

A aplicação cliente, desenvolvida para os experimentos deste cenário, era responsável por fazer as requisições periódicas a uma operação disponibilizada pelo serviço Web replicado. A aplicação também era responsável por fazer o registro dos resultados de cada invocação (tamanho da mensagem, política utilizada, servidor escolhido, tempo de resposta, entre outros) em um arquivo.

As invocações foram divididas em ciclos compostos por cinco invocações (cada invocação utilizando uma das políticas citadas no início desta Seção). Entre cada ciclo foi estabelecido um intervalo de 30 (trinta) segundos. Cada invocação feita pela aplicação cliente deveria obter uma resposta dentro de um prazo máximo (*timeout*) de 20 segundos.

5.2.2 Serviço Web utilizado

Utilizou-se um serviço `echo`, com duas operações `Object echo(Object element)` e `Object[] echo_n(Object element, int n)`. A primeira operação (utilizada no Cenário 2) simplesmente retorna o valor do parâmetro passado na invocação. Desta forma, variando o tamanho do elemento passado como parâmetro, têm-se invocações envolvendo mensagens de diferentes tamanhos. A segunda operação retorna o parâmetro de invocação copiado n vezes em um vetor. Dessa forma, variando o valor de n, têm-se invocações com uma mensagem de entrada de tamanho fixo e mensagens de resposta de diferentes tamanhos.

O código a seguir apresenta a implementação do serviço Web:

```
public class Echo {
  public Object echo(Object obj) {
    return obj;
  }

  public Object[] echo(Object obj, int t) {
    Object[] r = new Object[t];
    for(int i=0; i<t; i++)
      r[i] = obj;
    return r;
  }
}
```

5.2.3 Cliente e Servidores

A aplicação cliente foi executada, de forma dedicada, em uma máquina Pentium IV 3 GHz, com 1 GB RAM, SOR Linux, JDK 5.0 e interface de rede FastEthernet 100 Mbps. A estação cliente foi instalada no laboratório de Computação Distribuída da PUC Minas que possui, para acesso externo, um link DSL de 2 Mbps.

Foram utilizados seis servidores, distribuídos geograficamente pelo Brasil, como mostra a Figura 5.7. Os servidores responsáveis pela hospedagem das seis réplicas do serviço `Echo` estão localizados nas seguintes URIs:

- WS 1 - http://200.176.0.129/echo.asmx?WSDL (Terra)
- WS 2 - http://200.154.100.232/echo.asmx?WSDL (Terra)

- WS 3 - http://201.87.225.6/echo.asmx?WSDL (Empresa de BH)
- WS 4 - http://200.214.173.151/MPVRisk/echo.asmx?WSDL (Empresa de BH)
- WS 5 - http://200.131.41.132:8180/axis/Echo.jws?wsdl (CEFET - MG)
- WS 6 - http://143.106.7.180/axis/Echo.jws?wsdl (Unicamp)

Os nomes das empresas localizadas em Belo Horizonte não foram mencionados, uma vez que não se obteve autorização para tal.

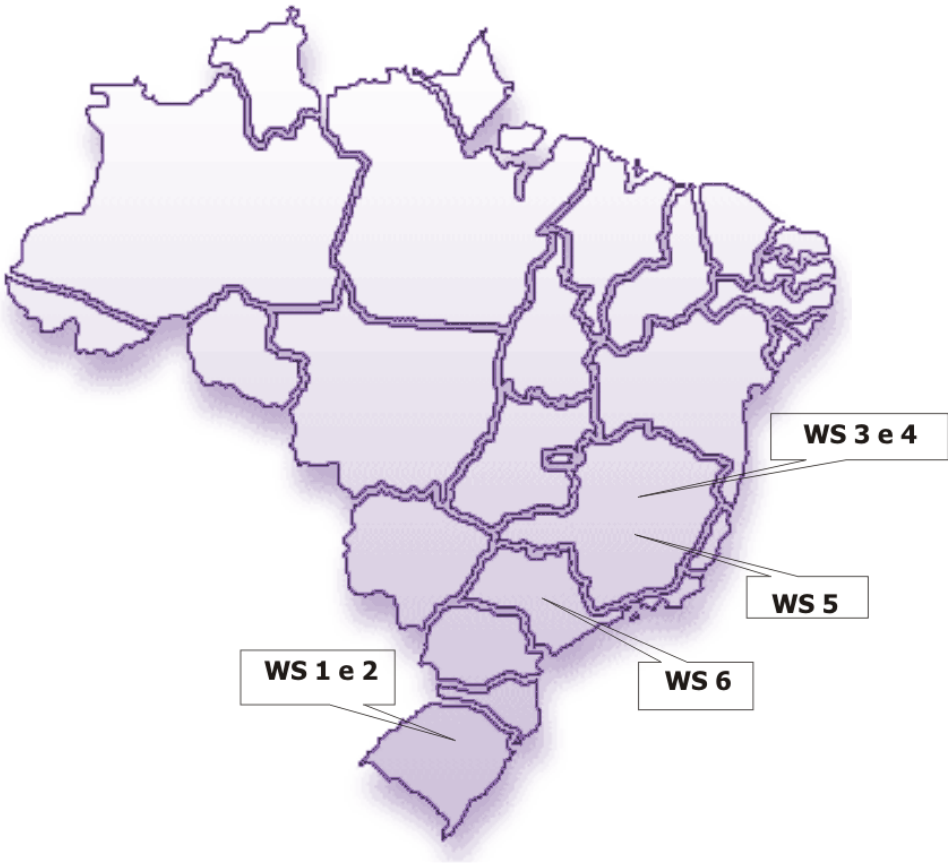

Figura 5.7: Localização geográfica dos servidores utilizados nos experimentos

5.2.4 Ciclos de Invocação

Os experimentos foram executados durante três semanas. Na primeira semana, usou-se uma mensagem de tamanho 8 KB, na segunda 4 KB e na terceira 1 KB. O objetivo foi analisar o comportamento do sistema em 3 cenários distintos. As invocações foram divididas em ciclos onde foram aplicadas, seqüencialmente, quatro das políticas suportadas

por SmartWS, incluindo duas variações da política PBM. Entre cada ciclo foi adicionado um intervalo de trinta segundos. Foi estipulado um *timeout* de 20 segundos onde, caso uma resposta não fosse recebida dentro deste tempo definido, a operação seria tratada como uma falha por tempo expirado. Foi necessário acrescentar este *timeout* devido ao fato de algumas chamadas demorarem muito mais que o tempo médio de resposta, inviabilizando a execução de um número mínimo de invocações ao longo do dia.

5.2.5 Experimentos

Nos três experimentos, que são descritos a seguir, foram utilizadas cinco políticas de invocação: Aleatória, Mediana e Paralela, além de 2 variações da política PBM. Nas variações da política PBM e na política Mediana, utilizou-se um *buffer* de tamanho 5. Outros parâmetros de configuração das políticas PBM e PBM2 são apresentadas na Tabela 5.1. Para cada invocação foi criado um *log* contendo as seguintes informações: número da invocação, tamanho da mensagem, política de seleção utilizada, servidor escolhido, tempo de resposta, data, hora e falha. Essas informações foram utilizadas para gerar os gráficos que são apresentados no restante da Seção.

Argumentos	**PBM**	**PBM2**
n	16	16
t	3	3
k	1.2	1.2
p	1	2

Tabela 5.1: Configurações da política PBM utilizadas no Cenário 2

Semana	**Mensagens**	**Experimento**	**Invocações**	***Timeouts***	**Outras falhas**
1ª	8 KB	5	30075	424	66
2ª	4 KB	6	26010	887	41
3ª	1 KB	7	38510	1382	29

Tabela 5.2: Registro de falhas durante os experimentos do Cenário 2

A Tabela 5.2 apresenta o número de invocações e o registro de falhas ocorridos no conjunto de experimentos do Cenário 2.

Experimento 5

Neste experimento utilizou-se de mensagens de tamanho 8 KB. Como pode ser visto na primeira linha da Tabela 5.2, cada política fez 6015 invocações, num total de 30075

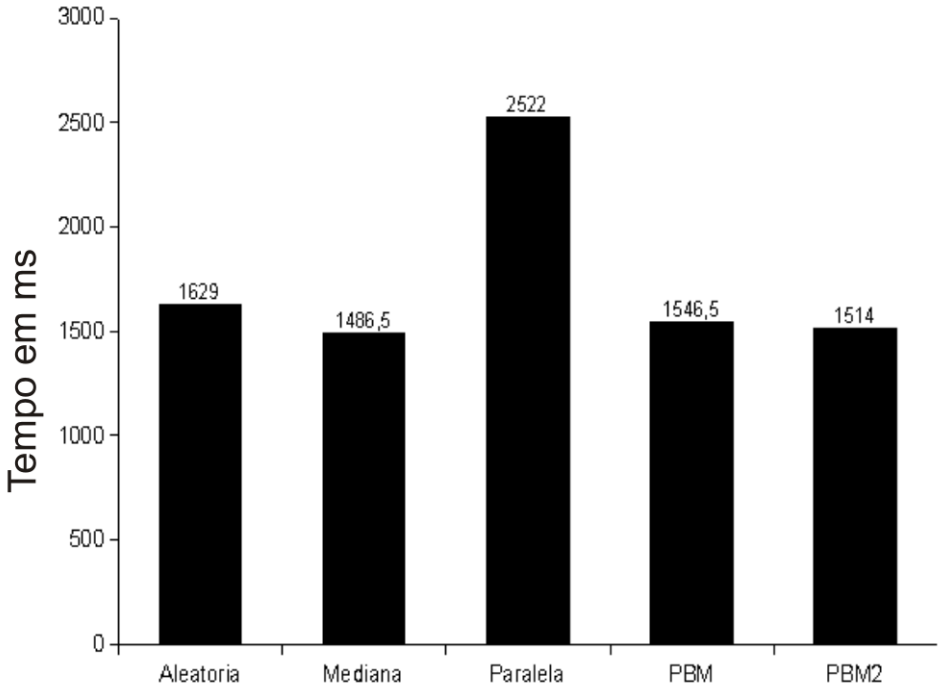

Figura 5.8: Mediana dos tempos de resposta do Experimento 5 (Cenário 2)

invocações. Com um *timeout* definido, 1,4% das requisições não obtiveram resposta dentro dos 20 segundos. Além disso, 0,2% das requisições falharam por outros motivos (falhas na rede local, no acesso à Internet ou nos servidores consultados).

A Figura 5.8 apresenta o gráfico com as medianas dos tempos de resposta obtidos pelas políticas de seleção de réplicas propostas no Experimento 5. Como foi observado que a média da amostra não refletia o comportamento típico da maioria dos tempos de resposta obtidos, usou-se a mediana como medida para avaliação, uma vez que esta foi pouco afetada pelos altos tempos de resposta considerados atípicos.

As políticas Melhor Mediana, PBM e PBM2 apresentaram as medianas gerais dos tempos de resposta muito próximas e isto se justifica por estas políticas apresentarem cerca de 50% dos tempos de resposta menores ou iguais a 1,5 segundo, como mostra o gráfico da Figura 5.9. A pequena vantagem obtida pela política Melhor Mediana foi construída devido a um maior número de respostas com tempo entre 1,5 e 2 segundos.

A política Aleatória apresentou um desempenho inferior às três políticas citadas acima durante todo o experimento. Já a política Paralela apresentou uma mediana geral cerca de 70% superior à mediana apresentada pela política Melhor Mediana. Esta diferença ocorreu devido ao *overhead* que esta gera na aplicação cliente devido à criação de múltiplas *threads*.

Apesar da política de seleção via Melhor Mediana apresentar a mais baixa mediana

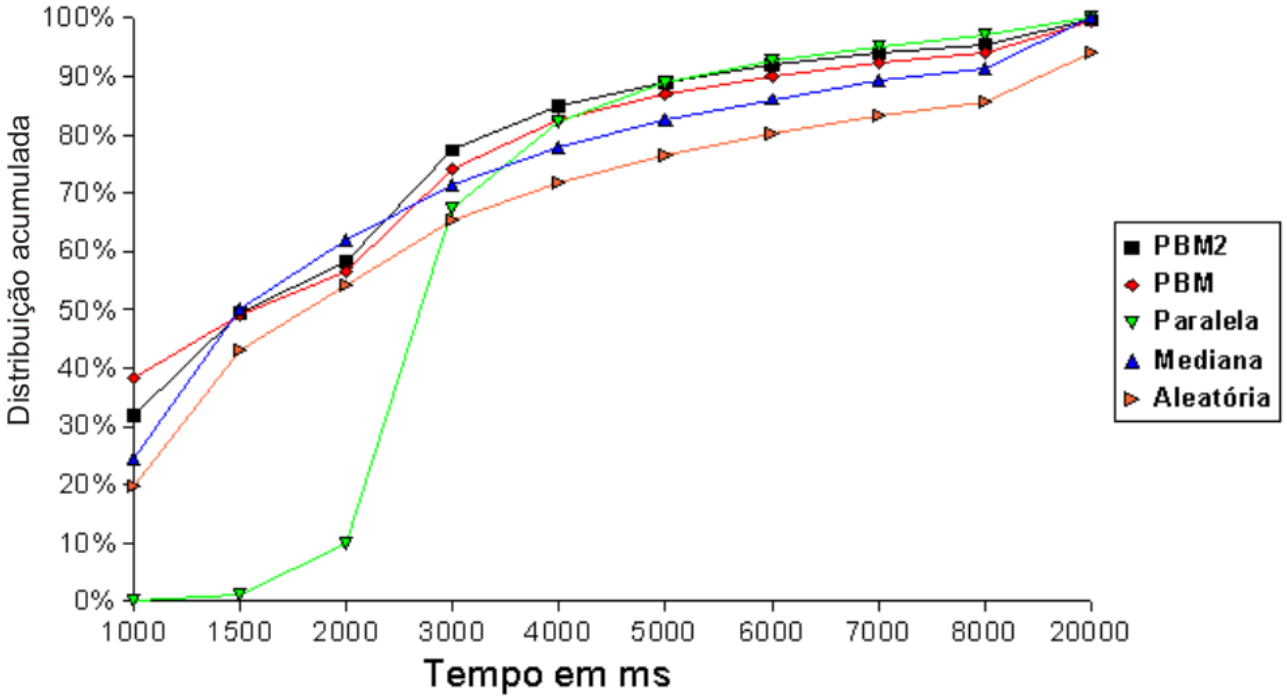

Figura 5.9: Distribuição acumulada - Experimento 5 (Cenário 2)

geral entre os tempos de resposta, esta política apresentou um problema no final da tarde do primeiro dia de experimento, mais precisamente nas invocações 1185, 1186 e 1187. Nestas invocações, a política Melhor Mediana registrou altos tempos de resposta do serviço 6 (Unicamp). Como o valor da mediana deste servidor aumentou muito, o servidor 6 deixou de ser invocado e provavelmente passou a ser a última opção na fila das medianas. Quando o servidor 6 voltou a apresentar o melhor tempo de resposta, a política Melhor Mediana não foi capaz de se adaptar ao novo cenário, invocando na maioria das vezes o servidor 1 (Terra), como mostra o gráfico da Figura 5.10. Esta situação resultou em uma queda no desempenho, como mostra o gráfico da Figura 5.9.

Apesar da política Aleatória apresentar o pior desempenho em todos os períodos do dia, como mostram os Gráficos 5.11 e 5.9, esta apresenta como aspecto positivo a característica de possibilitar um balanceamento de carga entre os servidores (Figura 5.10).

As políticas PBM e PBM2, que se diferenciam pelo valor de p (PBM, $p = 1$ e PBM2, $p = 2$), apresentaram os melhores desempenhos, inclusive nos horários de pico, como mostra o gráfico da Figura 5.11. Estas políticas apresentaram quase 80% dos tempos de resposta menores ou iguais a 3 segundos (Figura 5.9).

O fraco desempenho da política paralela utilizando mensagens de tamanho 8 KB,

principalmente nos períodos de 0 às 8h e 21 às 24h, como mostra a Figura 5.11, é justificado pelo *overhead* com a criação das *threads* no cliente, pelo aumento do tráfego na rede e pela sobrecarga nos servidores, que impediram que seu tempo de resposta, mesmo nos períodos com menor tráfego, fosse menor que 2 segundos. Assim, como pode ser visto no gráfico da Figura 5.9, cerca de 55% dos tempos de resposta da política Paralela ficaram entre 2 e 3 segundos.

Outro dado importante é a porcentagem de falhas de cada política de seleção utilizada no experimento. A Tabela 5.3 apresenta o registro de falhas de cada política durante o Experimento 5. Como se pode ver, a grande maioria das falhas em função de *timeout* ocorreu quando a política utilizada era a Aleatória.

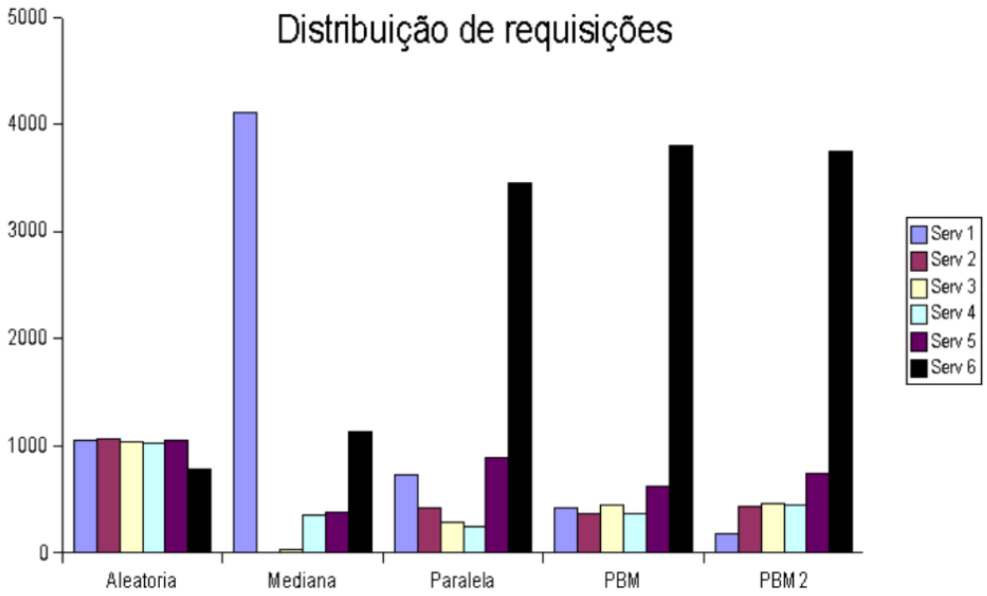

Figura 5.10: Distribuição de requisições do Experimento 5 (Cenário 2)

Política	***Timeouts***	**Outras falhas**
Aleatória	84%	0%
Mediana	0%	89%
Paralela	0%	6%
PBM	10%	2%
PBM2	6%	3%
Total	100%	100%

Tabela 5.3: Registro de falhas - Experimento 5 (Cenário 2)

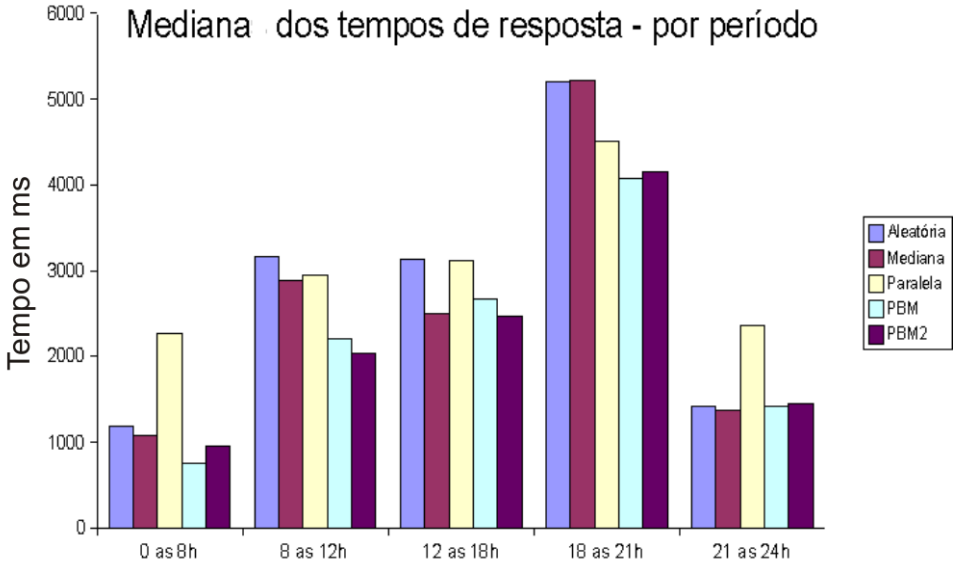

Figura 5.11: Mediana dos tempos de resposta por período - Experimento 5 (Cenário 2)

Experimento 6

Neste experimento utilizou-se mensagens de tamanho 4 KB. Como pode ser visto na segunda linha da Tabela 5.2, cada política fez 5202 invocações, em um total de 26010 invocações. Com um *timeout* definido, 3,4% das requisições não obtiveram resposta dentro dos 20 segundos. Além disso, 0,2% das requisições falharam por outros motivos.

Como no Experimento 5 o servidor 6 respondeu à grande maioria das invocações (mais de 70%) das políticas Paralela, PBM e PBM2, optou-se por deixá-lo de fora dos Experimentos 6 e 7. O objetivo foi avaliar o desempenho das políticas em um ambiente onde não existe um servidor que se destaca. A primeira conseqüência notada foi o aumento dos tempos de resposta, fazendo com que o experimento que utiliza mensagens de 4 KB apresentasse menos respostas que o experimento com mensagens de 8 KB.

É possível notar no gráfico da Figura 5.12 que as variações da política PBM alcançaram os melhores resultados no experimento, obtendo os menores tempos de mediana geral. As políticas Aleatória, Mediana e Paralela apresentaram uma mediana geral cerca de 14% mais alta que a política denominada PBM2.

Mais uma vez a política Aleatória apresentou o pior desempenho em todos os períodos, como mostram os gráficos das Figuras 5.13 e 5.14. Porém, manteve-se a característica de possibilitar o balanceamento de carga entre os servidores (gráfico da Figura 5.15).

Como no Experimento 5, a política de seleção via Melhor Mediana deixou de invocar o servidor 5 (CEFET - MG) após o registro de três tempos de resposta altos. Após esta

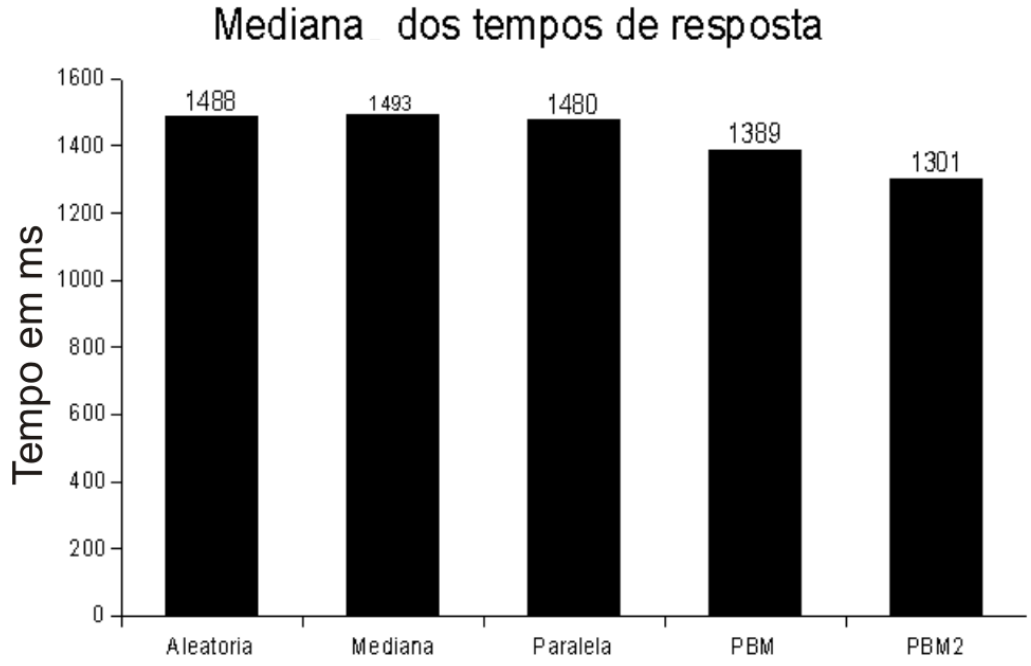

Figura 5.12: Mediana dos tempos de resposta - Experimento 6 (Cenário 2)

alteração na fila de medianas, a política Melhor Mediana passou a invocar, na maioria das vezes, o servidor 1 (Terra) e posteriormente o servidor 3 (Empresa de BH) (Figura 5.15). Esta alteração na fila de medianas não demonstrou ser a mais adequada, uma vez que resultou em um baixo desempenho em todos os períodos do dia, como mostra o gráfico da Figura 5.13.

O gráfico da Figura 5.14 apresenta a distribuição acumulada do Experimento 6. A política PBM2 se destacou no experimento, apresentando cerca de 58% dos tempos menores ou iguais a 1,5 segundo. Além disso, as variações da política PBM, apresentaram mais de 30% dos tempos de resposta menores ou iguais a 1 segundo.

A política de seleção Paralela apresentou cerca de 90% dos tempos menores ou iguais a 4 segundos. O fato de menos de 5% dos tempos serem menores ou iguais a 1 segundo se justifica pelo *overhead* apresentado pela política Paralela.

A Tabela 5.4 apresenta o registro de falhas de cada política durante o Experimento 6. Como se pode ver, 92% das falhas em função de *timeout* ocorreram quando a política utilizada era a Aleatória.

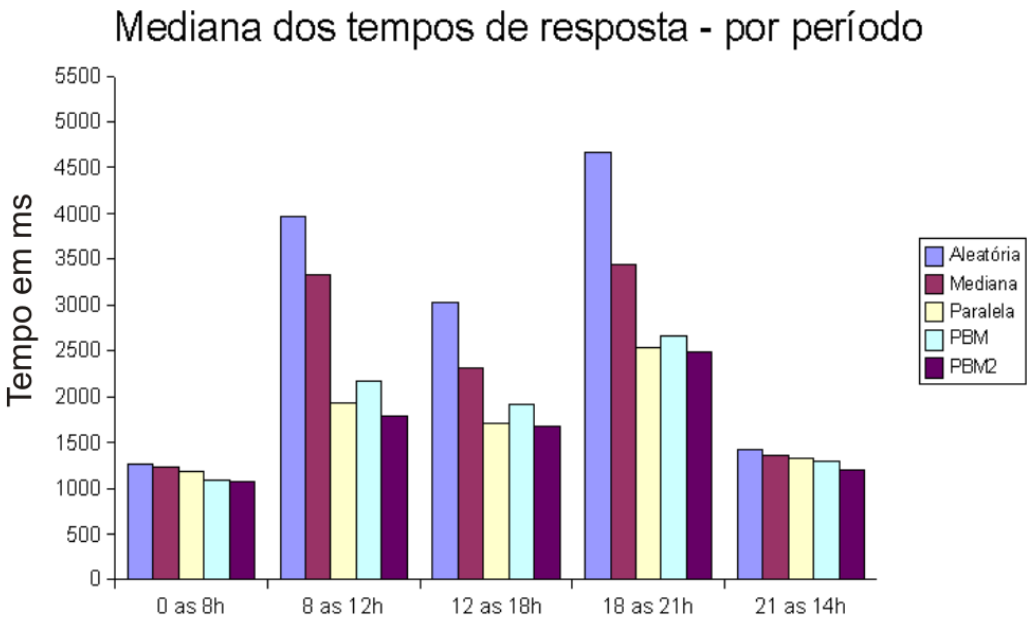

Figura 5.13: Mediana dos tempos de resposta - por período - Experimento 6 (Cenário 2)

Política	***Timeouts***	**Outras falhas**
Aleatória	92%	2%
Mediana	0%	88%
Paralela	0%	5%
PBM	5%	0%
PBM2	3%	5%
Total	100%	100%

Tabela 5.4: Registro de falhas - Experimento 6 (Cenário 2)

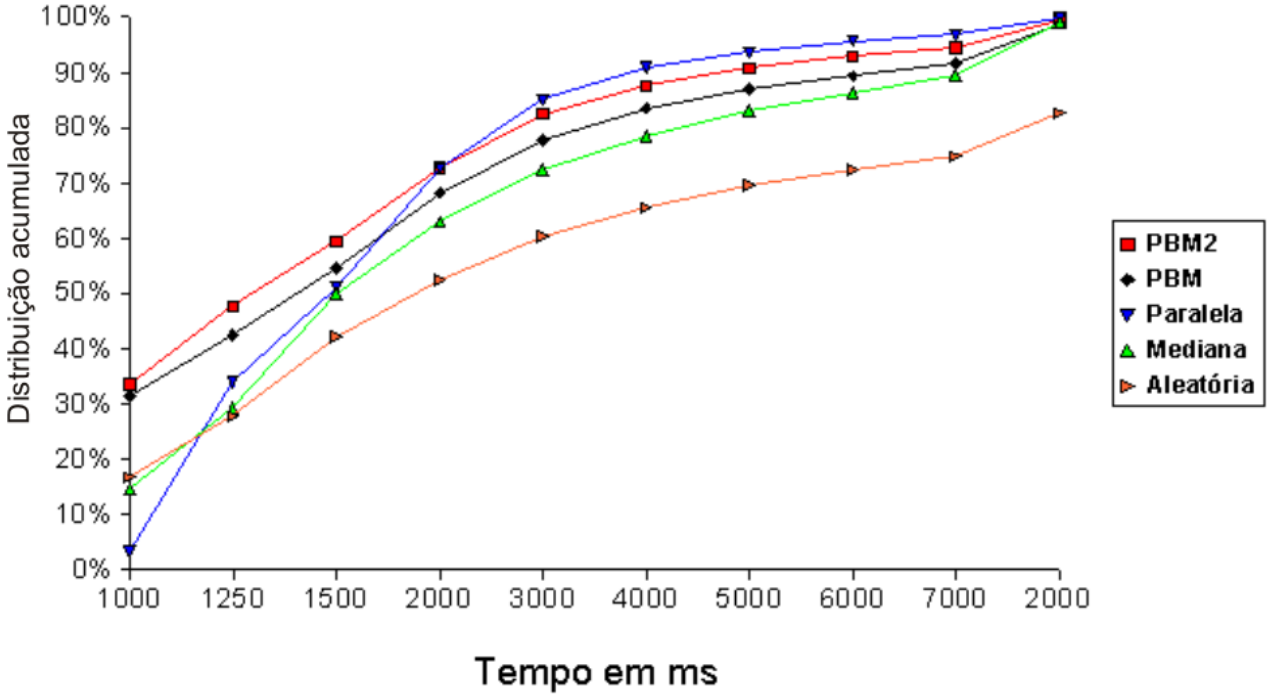

Figura 5.14: Distribuição acumulada - Experimento 6 (Cenário 2)

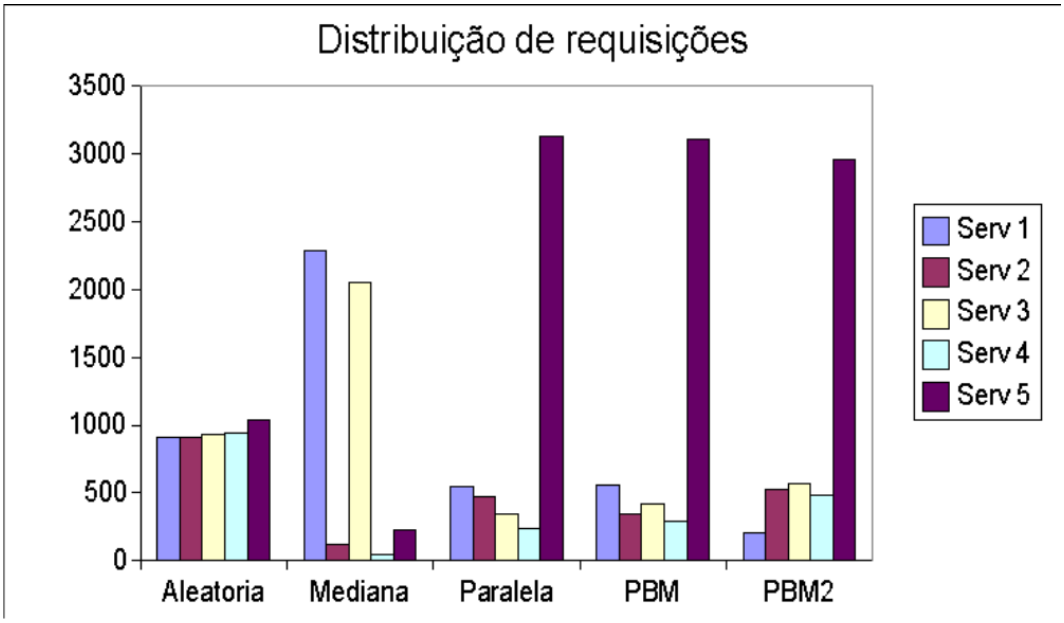

Figura 5.15: Distribuição de requisições - Experimento 6 (Cenário 2)

Experimento 7

Neste experimento, utilizou-se mensagens de tamanho 1 KB. Como pode ser visto na terceira linha da Tabela 5.2, cada política fez 7702 invocações, em um total de 38510 invocações. Com um *timeout* definido, 3,6% das requisições não obtiveram resposta dentro dos 20 segundos. Além disso, 0,08% das requisições falharam por outros motivos.

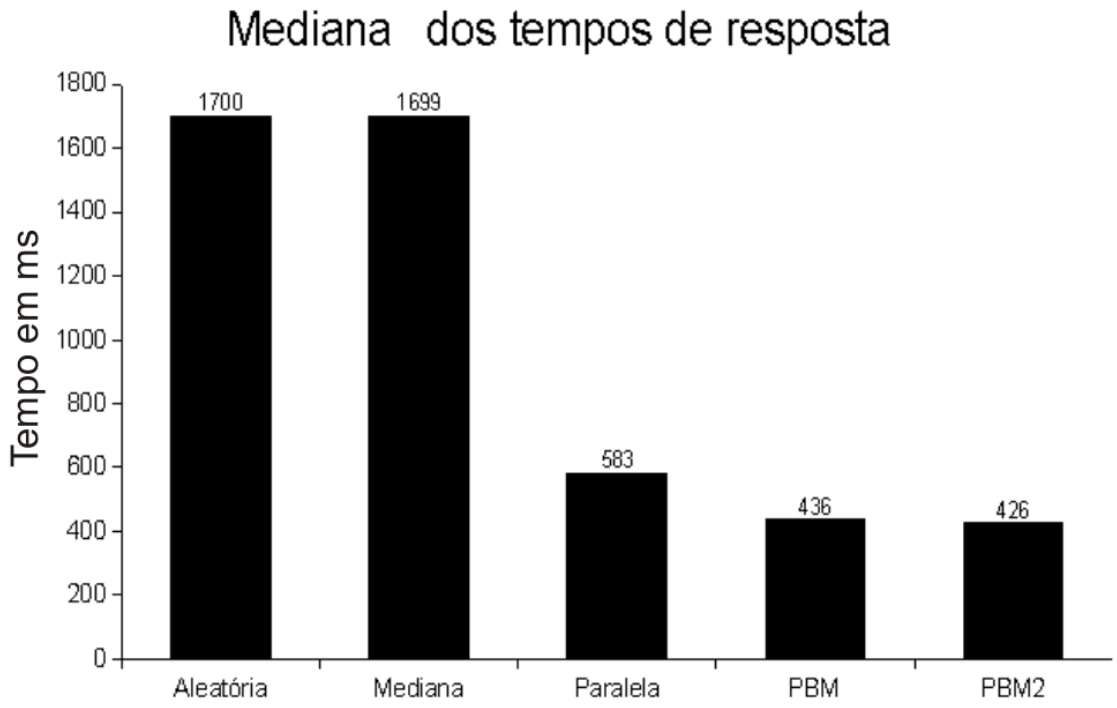

Figura 5.16: Mediana dos tempos de resposta - Experimento 7 (Cenário 2)

Como pode ser visto no gráfico da Figura 5.16, as variações da política PBM (PBM e PBM2) apresentaram as menores medianas gerais dos tempos de resposta. A política Paralela apresentou uma mediana geral cerca de 35% maior que a política PBM2. Estas três políticas invocaram, na grande maioria das vezes, o servidor 5 (CEFET), como mostra o gráfico da Figura 5.17.

As políticas Mediana e Aleatória apresentaram suas medianas cerca de 4 vezes maior que a mediana da política PBM2. Este tempo se justifica pelo fato de apresentarem um alto índice de balanceamento de carga (Figura 5.17), o que nesta situação, por serem servidores tão heterogêneos, resultou em uma queda de desempenho. Esta queda foi constante, durante todos os períodos do dia, como mostra o gráfico da Figura 5.18.

O gráfico da Figura 5.19 apresenta a distribuição acumulada dos tempos de resposta. Como pode ser observado, as políticas PBM e PBM2 apresentaram um desempenho praticamente idêntico, apresentando cerca de 55% dos tempos de resposta menores ou iguais

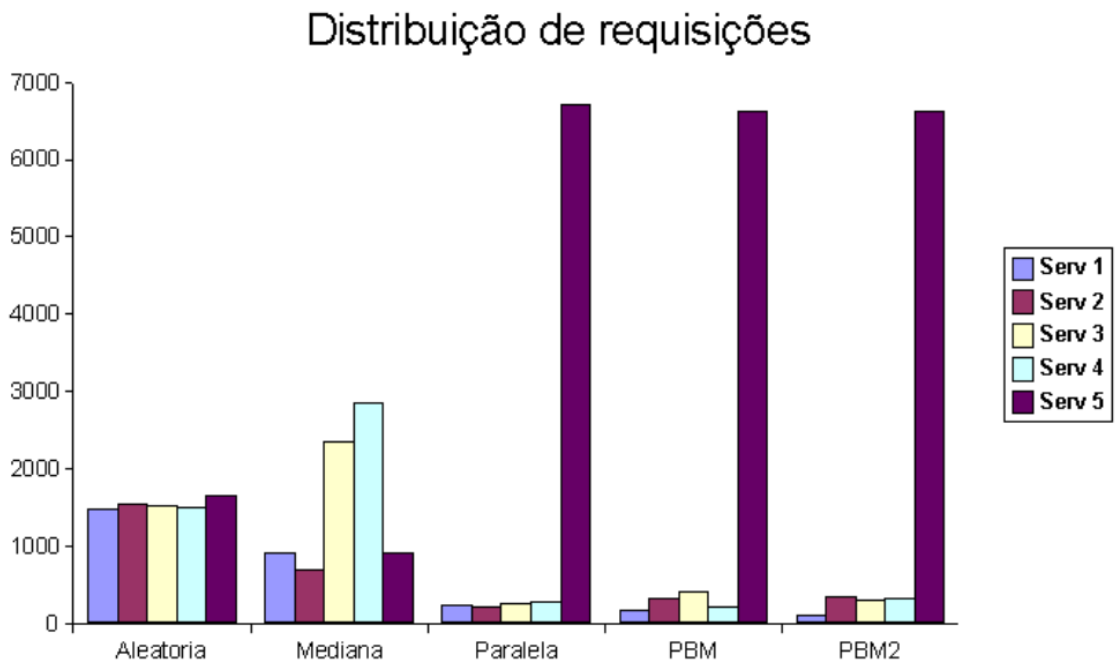

Figura 5.17: Distribuição de requisições - Experimento 7 (Cenário 2)

a 500 *ms*.

A política Paralela apresentou mais de 50% dos tempos de resposta menores ou iguais a 600 *ms* e, como as políticas PBM e PBM2, quase 70% dos tempos menores ou iguais a 700 *ms*.

O gráfico da Figura 5.19 também retrata o péssimo desempenho das políticas Mediana e Aleatória. Estas políticas apresentaram, respectivamente, 8% e 11% dos tempos menores ou iguais a 1,5 segundo.

Outro dado importante é a porcentagem de falhas de cada política de seleção utilizada no experimento. A Tabela 5.5 apresenta o registro de falhas de cada política durante o Experimento 7. Mais uma vez, a grande maioria das falhas em função de *timeout* ocorreu quando a política utilizada era a Aleatória.

Política	***Timeouts***	**Outras falhas**
Aleatória	92%	0%
Mediana	0%	100%
Paralela	0%	0%
PBM	4%	0%
PBM2	4%	0%
Total	100%	100%

Tabela 5.5: Registro de falhas - Experimento 7 (Cenário 2)

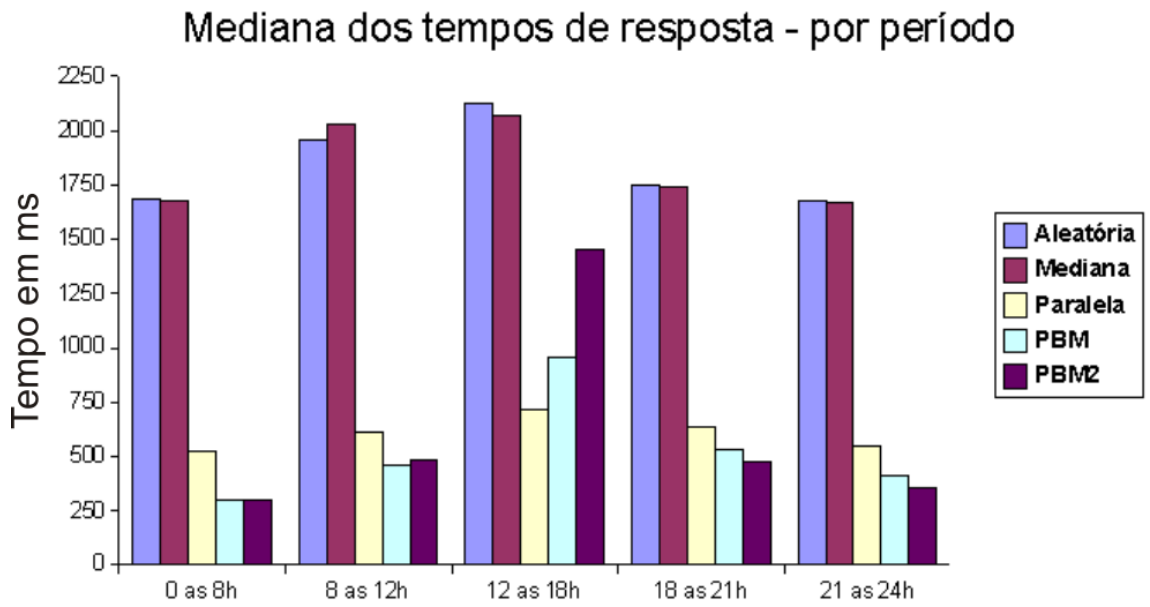

Figura 5.18: Mediana dos tempos de resposta - por período - Experimento 7 (Cenário 2)

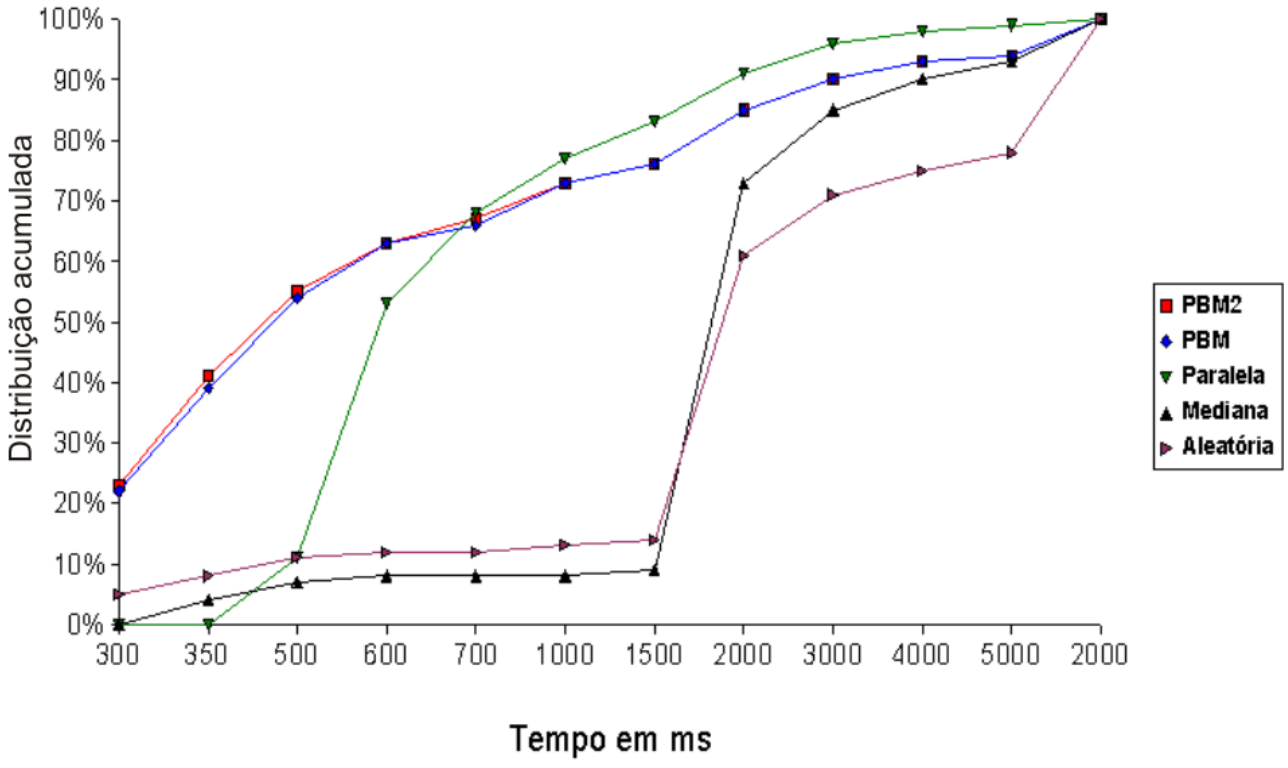

Figura 5.19: Distribuição acumulada - Experimento 7 (Cenário 2)

5.3 Conclusões

Neste capítulo foram apresentados sete experimentos realizados utilizando servidores Web reais, espalhados geograficamente pelo Brasil. Os quatro primeiros experimentos tiveram como objetivo fornecer informações preliminares sobre o desempenho das políticas para validar a implementação de SmartWS. Nesses experimentos, foi utilizado um serviço de Conversão de Temperatura, onde um cliente executou 100 invocações ao serviço utilizando cada política suportada.

Os Experimentos 5, 6 e 7 utilizaram seis servidores Web reais, espalhados geograficamente pelo Brasil. Esses experimentos foram executados por 12 dias, 24 horas por dia, divididos em ciclos de invocação.

Como resultado desses experimentos, a Tabela 5.6 resume um conjunto de diretrizes para escolha de políticas de seleção de réplicas, com o intuito de ajudar o programador de clientes de serviços Web a escolher a política mais adequada a um determinado ambiente computacional.

Os gráficos 5.20, 5.21, 5.22, apresentam os somatórios dos tempos de resposta obtidos nos Experimentos 5, 6 e 7, descritos neste Capítulo.

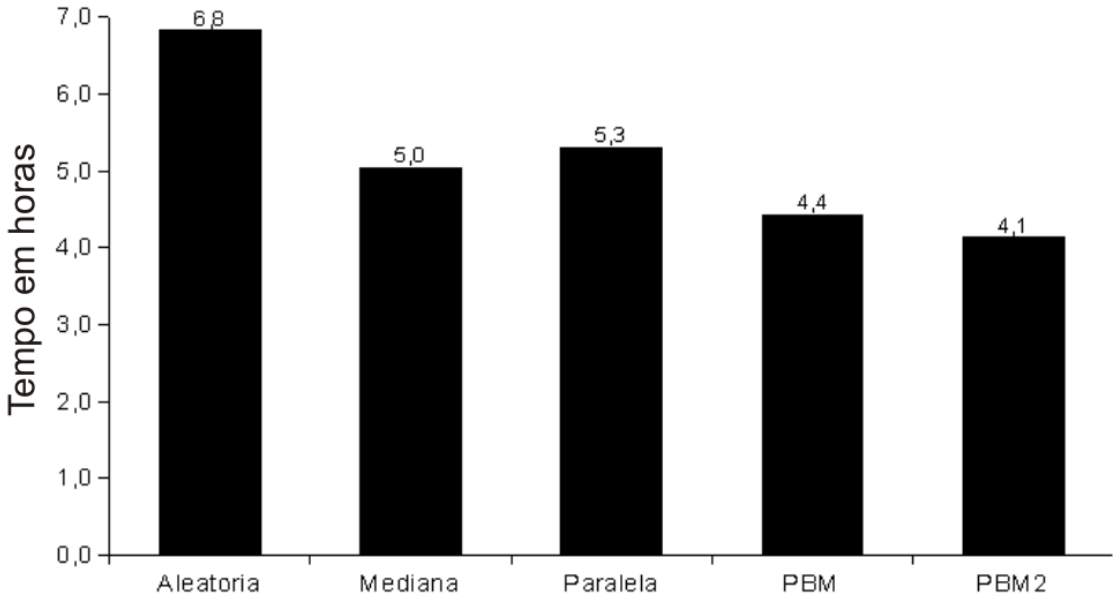

Figura 5.20: Somatório dos tempos de invocação - Mensagens de 8 KB

Política	**Ambiente Computacional**
Estática	Recomendada quando se tem servidores com tempos de resposta conhecidos *a priori*, estáveis e diferentes entre si, como demonstrado no Experimento 1.
Aleatória	Recomendada quando se tem servidores com tempos de resposta conhecidos *a priori*, estáveis e semelhantes entre si, como mostra o Experimento 2.
Mediana	Recomendada quando se tem servidores conhecidos *a priori*, sendo que estes apresentam pequenas variações nos tempos de resposta. Esta política não é recomendada para ambientes instáveis, como a Internet, pelos motivos apresentados na Seção 3.1.4 e demonstrados nos Experimentos 5, 6 e 7.
Paralela	Esta política é recomendada quando se tem servidores com tempos de resposta desconhecidos ou que variam ao longo do tempo, que falham com freqüência considerável e em situações onde é permitido invocar todas as réplicas simultaneamente. Além disso, o uso desta política é recomendado somente quando se utiliza um tamanho de mensagem que gere um *overhead* suportado pela rede, como mostram os gráficos das Figuras 5.21 e 5.22.
PBM	Esta política é recomendada quando se tem servidores com tempos de resposta desconhecidos ou que variam ao longo do tempo, que falham com freqüência considerável e onde o tamanho da mensagem não é previamente conhecido. Resumindo, como mostram os Experimentos 5, 6 e 7 e os gráficos das Figuras 5.20, 5.21 e 5.22, a política PBM é recomendada para ambientes como a Internet.

Tabela 5.6: Diretrizes para escolha de políticas de seleção de réplicas

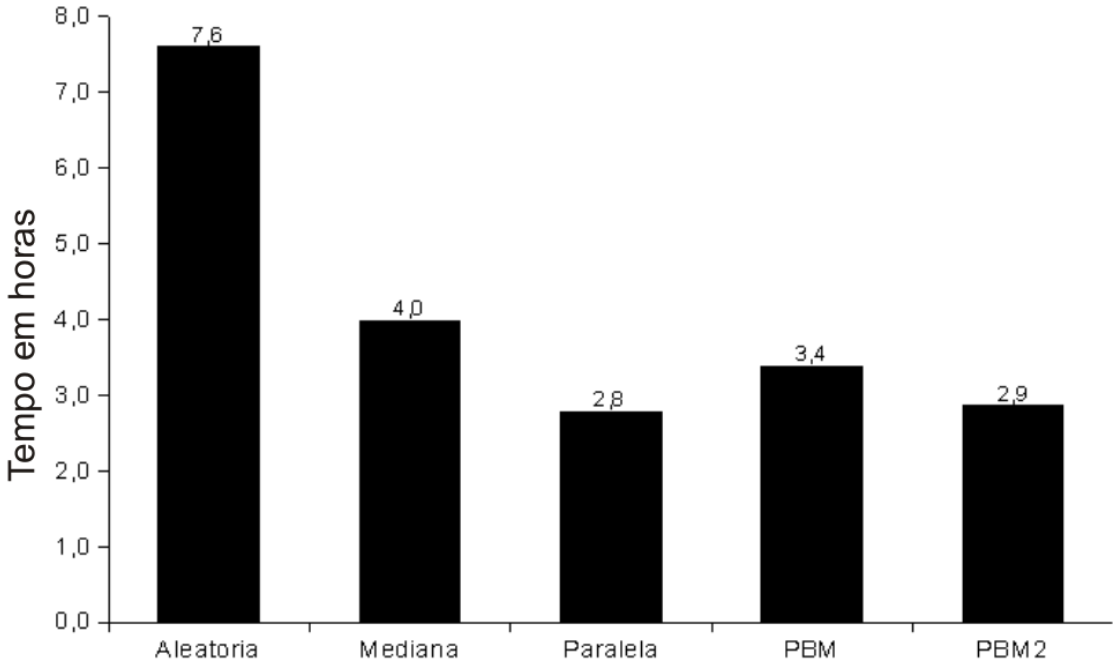

Figura 5.21: Somatório dos tempos de invocação - Mensagens de 4 KB

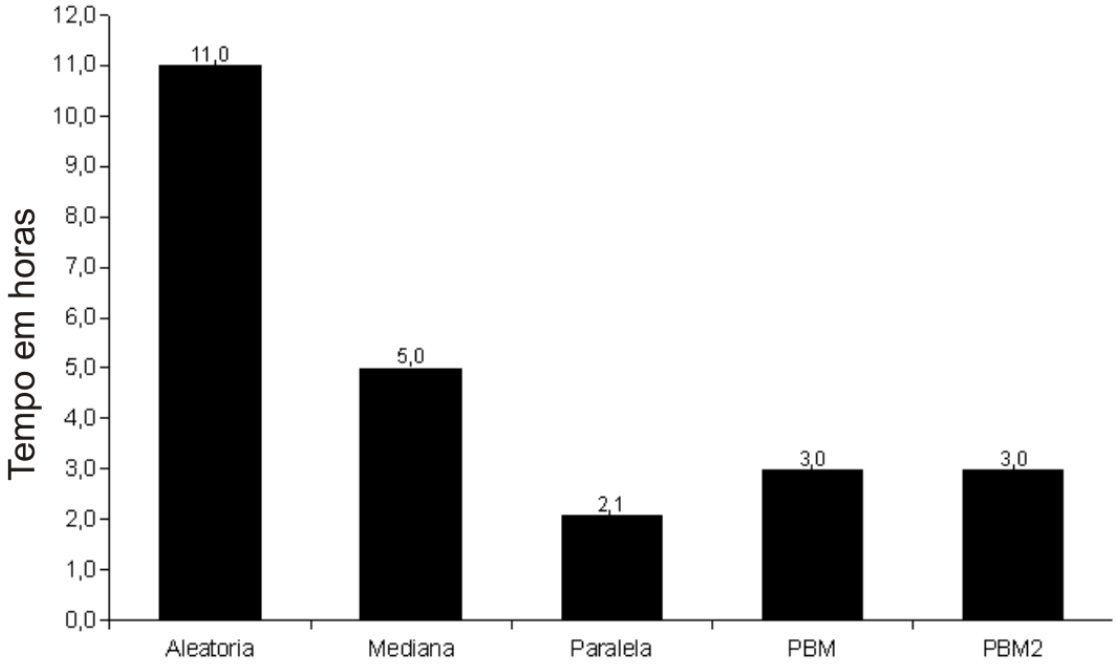

Figura 5.22: Somatório dos tempos de invocação - Mensagens de 1 KB

Capítulo 6

Conclusões

Nesta dissertação, descreveu-se o projeto e a implementação de um sistema com o objetivo principal de suportar políticas de invocação de serviços Web replicados, denominado SmartWS.

SmartWS tem por objetivo adicionar transparência de replicação a uma aplicação cliente de serviços Web replicados. Para evitar o entrelaçamento do código funcional do cliente com o código não-funcional responsável por prover esta transparência, encapsulou-se esta tarefa em um sistema de *middleware*. SmartWS se vale do conceito de *smart proxies*, que são os *proxies* especiais responsáveis por fornecer mecanismos para selecionar a réplica que melhor atenda às necessidades de uma aplicação cliente, de forma transparente.

No sistema proposto, *smart proxies* são gerados automaticamente, a partir de uma linguagem de especificação de alto nível, por meio da qual configura-se a política de acesso a servidores replicados de uma aplicação cliente. Um protótipo do sistema foi implementado em Java, tendo sido utilizado nos experimentos descritos no Capítulo 5.

A versão atual de SmartWS, apesar de ser independente de sistema operacional, suporta a geração de *smart proxies* compatíveis apenas com a plataforma Apache Axis, largamente utilizada no desenvolvimento de aplicações distribuídas baseadas na tecnologia de serviços Web. Pretende-se, futuramente, prover suporte a outras plataformas de *middleware*, como JAX-WS [JW] e Microsoft .NET [Mica].

Além de descrever o projeto e a implementação de SmartWS, foram apresentados, no Capítulo 5, sete experimentos divididos em dois cenários. Os Experimentos 1, 2, 3 e 4 (cenário 1) foram realizados com o objetivo de validar a implementação de SmartWS. Os Experimentos 5, 6 e 7 (cenário 2) foram executados durante 12 dias (24 horas por dia), por meio de uma avaliação empírica utilizando seis servidores Web reais distribuídos geograficamente pelo Brasil. O objetivo no cenário 2 era definir diretrizes para ajudar

o programador de clientes de serviços Web a escolher a política de seleção de réplicas mais adequada a um determinado ambiente computacional. As diretrizes sugeridas foram sumarizadas na Tabela 5.6.

A partir de uma análise das políticas suportadas por SmartWS é possível apresentar as seguintes conclusões:

- Política Estática: esta política apresenta como vantagem o fato de que o usuário define como prioritários os servidores com melhor desempenho. Como esta política invoca no máximo um servidor de cada vez, ela não gera nenhum tipo de *overhead*. Porém esta política não se aplica a ambientes dinâmicos como a Internet, uma vez que a ordem das réplicas é definida estaticamente em tempo de compilação.

- Política Aleatória: a principal vantagem desta política é o balanceamento de carga proporcionado pela escolha aleatória, em tempo de execução, dos servidores replicados disponíveis. Porém, o fato de acessar, na mesma proporção, servidores com melhor e pior desempenho, inviabiliza o uso desta política em ambientes heterogêneos como a Internet.

- Política Mediana: candidata a melhor política dentre as políticas avaliadas no início desta pesquisa, devido ao bom desempenho em outros estudos, principalmente por invocar somente um servidor por vez e se adaptar a pequenas alterações no cenário. No entanto, uma avaliação empírica, com servidores reais espalhados geograficamente pelo Brasil, revelou um grave problema desta política diante de alterações mais bruscas no cenário. Estas alterações incluem queda brusca de desempenho ou falha temporária de determinado servidor que apresentava o melhor desempenho e por um curto período esteve sobrecarregado ou indisponível. Como demonstrado, a política Mediana não é capaz de detectar o restabelecimento do servidor caso este volte a apresentar um bom desempenho. Para que a política Mediana volte a invocar este servidor, é preciso que todos os outros piorem ainda mais.

- Política Paralela: a política Paralela apresentou um bom desempenho em ambientes heterogêneos, onde ocorrem variações bruscas nos tempos de resposta e falhas constantes. Utilizando esta política é possível obter sempre a resposta do servidor que responde mais rápido. Porém, à medida que o tamanho da mensagem aumenta, aumenta também o *overhead* de transmitir em paralelo todas as requisições, o que afeta diretamente o desempenho desta política. Além disso, nem todo tipo de serviço permite que se acesse mais de uma réplica simultaneamente.

Como resultado das conclusões acima, foi incorporado em SmartWS a política PBM (*Parallel Best Median*), que possui características que combinam as qualidades das políticas Paralela e Melhor Mediana, somadas a mecanismos que buscam atenuar os problemas destas mesmas políticas, apresentados no Capítulo 3.

Para definir a política PBM partiu-se de três princípios básicos:

- Apesar da política Paralela obter sempre a resposta do servidor que responde primeiro, o *overhead* gerado pode impedir que se obtenha esta resposta no menor tempo possível. A solução proposta é que, baseado nas características da rede, o programador defina o grau máximo de paralelismo suportado.

- O critério de utilizar a melhor mediana dos tempos de resposta das k últimas invocações é interessante, mas é necessário que se utilize outros recursos para prover mecanismos de adaptação caso haja alterações bruscas nas condições da rede.

- É imprescindível, independentemente do critério para seleção da réplica, renovar os tempos de resposta de todos os servidores periodicamente. Somente assim é possível ter um novo panorama do cenário.

Além de políticas para seleção de réplicas, SmartWS apresenta uma série de outras funcionalidades. SmartWS possibilita que clientes utilizem serviços Web desenvolvidos por equipes distintas por meio de adaptadores de interface. Estes adaptadores possibilitam tornar compatíveis as interfaces concretas dos servidores (que podem possuir diferenças de implementação como nomes de métodos, tipos dos parâmetros, etc.) com a interface abstrata utilizada pelo cliente. Em SmartWS, estes adaptadores são construídos manualmente, uma vez que demandam conhecimento tanto dos métodos da interface abstrata quanto da interface concreta. A construção da interface abstrata é o único ponto onde deve haver uma interação entre o programador da aplicação (que usará a interface abstrata) e o programador SmartWS.

Dois outros recursos suportados por SmartWS têm como objetivo auxiliar na manutenção de consistência de dados da réplica acessada e economizar recursos de comunicação. Em determinadas situações, deixa de ser interessante o uso de políticas de seleção de réplica, quando a execução de um método altera o estado de uma réplica, exigindo que as operações seguintes sejam executadas sobre o estado alterado.

Em relação à economia de recursos de comunicação em rede, SmartWS permite que se defina uma seqüência legal de invocação de métodos. Este recurso se aplica em um ambiente como a Internet, onde na maioria das vezes, estas réplicas são desenvolvidas e disponibilizadas por equipes diferentes e geograficamente distribuídas, das quais não se

pode exigir qualquer nível de interação. O autômato finito determinístico, definido pelo programador SmartWS, evita o custo de se fazer uma chamada a um servidor sem que antes tenha sido executado outro método que serve como requisito para a execução.

A política de seleção PBM, proposta nesta dissertação, divide as invocações em ciclos. Porém as características destes ciclos como tamanho, número de invocações em paralelo, grau de paralelismo e limite superior para o paralelismo são definidos estaticamente em tempo de compilação. Uma proposta para trabalhos futuros é permitir que estas configurações sejam determinadas em tempo de execução. Com isso será possível, por exemplo, definir um tamanho de ciclo maior ou menor, com mais ou menos invocações em paralelo.

Para alterar estas informações dinamicamente é necessário que se desenvolva um mecanismo para monitoramento e análise dos dados históricos, com o objetivo de se detectar mudanças de comportamento no ambiente em que se encontram cliente e servidores. Esta análise dos dados do histórico poderia ser feita com técnicas de Inteligência Artificial como, por exemplo, usando-se Redes Neurais. Desta forma, seria possível que a política se ajustasse automaticamente ao período do dia, diminuindo o grau de paralelismo e/ou o limite superior para o paralelismo em horários de pico e, em horários menos congestionados, aumentasse o tamanho do ciclo evitando sucessivas atualizações nos tempos de todos os servidores.

Os gráficos 5.20, 5.21 e 5.22 comprovam que à medida que o tamanho da mensagem aumenta, aumenta o efeito do *overhead* gerado ao se invocar todas as réplicas disponíveis em paralelo. Porém, esta dissertação não apresenta dados suficientes para definir um limite de tamanho de mensagem para uma determinada estrutura de rede. Assim, uma sugestão para trabalho futuro é que se faça um estudo para definir o limite suportado com mensagens de tamanho x em uma rede com largura de banda y, acessando em paralelo z servidores Web.

Outra opção interessante seria permitir, em tempo de execução, uma combinação das políticas usadas em SmartWS, ou mesmo alteração entre elas. Por exemplo, iniciar utilizando a política Paralela e, caso somente um servidor responda à grande maioria das invocações, pode-se mudar para uma ordem definida estaticamente ou baseada na mediana. Caso este servidor piore o desempenho ou pare de responder, pode-se ativar novamente o uso da política Paralela.

Uma limitação de SmartWS é o fato de não armazenar os tempos de resposta divididos por método ou classes de métodos. Esta limitação pode influenciar diretamente em servidores que respondam a mais de um método. Caso os métodos tenham tempo de resposta muito distintos, na atual forma em que o histórico é tratado, o acesso a um método com alto tempo de resposta pode comprometer este servidor em uma próxima invocação a um

outro método. A utilização de *buffers* diferentes para cada método de um servidor é a solução mais recomendada para este problema.

Apêndice A

Linguagem para Seleção de Réplicas

Mostra-se nesse apêndice a gramática completa da linguagem de especificação, utilizada pelo usuário de SmartWS. Na descrição da gramática usa-se uma variante da BNF proposta em [Wir77]. Nessa variação, terminais são mostrados entre aspas duplas (ex: "**webservice**"), repetição é expressa por chaves (ex: $\{a\}$ equivale a $\epsilon|a|aa|aaa|\cdots$) e opcionalidade por colchetes (ex: $[a]$ equivale a $\epsilon|a$).

```
program= abstract_interface web_service { web_service } [ buffer ]
[ session ] [ protocol ] [ timeout ] policy "end"

abstract_interface= "interface" "name" "=" string

web_service= "webservice" alias uri [ adapter ]
alias= "alias" "=" string
uri= "uri" "=" string
adapter= "adapter" "=" string

buffer= "buffer" size refresh
size= "size" "=" integer
refresh= "refresh" "=" integer

session= "session" begin finish retry interval
begin= "begin" "=" string
finish= "finish" "=" string
retry= "retry" "=" integer
interval= "interval" "=" double
```

```
protocol= "protocol" methods states transitions
methods= "methods" "=" string { "," string }
states= "states" "=" string { "," string }
transitions= "transitions" "=" transition_def ";" { transition_def ";" }
transition_def= string "," method_def "," string
method_def= string { "||" string }

timeout= "timeout" "=" double

policy= "policy" "=" policy op policy ||
                     simple_policy ||
                     "(" policy ")"
simple_policy = pbm_def || bestmedian_def || string
op= "?" || ">" || "|"
pbm_def= "pbm" "(" double "," integer "," integer "," integer "," string
{ "," string } ")"
bestmedian_def= "bestmedian" "(" string { "," string } ")"
```

Bibliografia

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architecture and Applications*. Springer Verlag, 2004.

[ADH+05] Wil M. P. Aalst, Marlon Dumas, Arthur H. M. Hofstede, Nick Russell, H. M. W. (Eric) Verbeek, and Petia Wohed. Life after BPEL? In *European Performance Engineering Workshop*, pages 35–50, 2005.

[AMS+06] Valeria De Antonellis, Michele Melchiori, Luca De Santis, Massimo Mecella, Enrico Mussi, Barbara Pernici, and Pierluigi Plebani. A layered architecture for flexible web service invocation. *Software Practice and Experience*, 36(2):191–223, 2006.

[Arc] Service-Oriented Architecture. http://www-128.ibm.com/developerworks/webservices.

[Axi] Apache Axis. http://ws.apache.org/axis/.

[BD05] Seán Baker and Simon Dobson. Comparing service-oriented and distributed object architectures. In *On The Move - OTM Federated Conferences and Workshops*, volume 3760 of *Lecture Notes in Computer Science*, pages 631–645. Springer, 2005.

[BL89] J. Van Den Bos and C. Laffra. Procol: a parallel object language with protocols. In *Object-oriented Programming Systems, Languages and Applications*, pages 95–102. ACM Press, 1989.

[BST92] N. Budhiraja, F. B. Schneider, and S. Toueg. Primary-backup protocols: lower bounds and optimal implementations. In *Proceedings of the Third IFIP Conference on Dependable Computing for Critical Applications*, pages 321–343, September 1992.

[CCI99] Renato Cerqueira, Carlos Cassino, and Roberto Ierusalimschy. Dynamic component gluing across different componentware systems. In *DOA '99: Proceedings of the International Symposium on Distributed Objects and Applications*, pages 362–371. IEEE Computer Society, 1999.

[CDK+02] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the web services web: An introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93, 2002.

[CDK05] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems Concepts and Design*. Addison-Wesley, 2005.

[Cer02] Ethan Cerami. *Web Services Essentials*. O'Reilly & Associates, Inc., 2002.

[CFL+a] Francisco Curbera, Don Ferguson, Frank Leymann, Jagan Peri, Satish Thatte, and Sanjiva Weerawarana. http://www-128.ibm.com/developerworks/library/specification/ws-tx/. Web Services Coordination (WS-Coordination). August 2005.

[CFL+b] Francisco Curbera, Don Ferguson, Frank Leymann, Jagan Peri, Satish Thatte, and Sanjiva Weerawarana. http://dev2dev.bea.com/pub/a/2004/01/ws-transaction.html. Web Services Transaction (WS-Transaction). BEA, IBM, Microsoft, Janeiro 2004.

[CFP+03] Carlos Canal, Lidia Fuentes, Ernesto Pimentel, José M. Troya, and Antonio Vallecillo. Adding roles to CORBA objects. *IEEE Transactions Software Engineering*, 29(3):242–260, 2003.

[CKM+03] Francisco Curbera, Rania Khalaf, Nirmal Mukhi, Stefan Tai, and Sanjiva Weerawarana. The next step in web services. *Communications of the ACM*, 46(10):29–34, 2003.

[Con] W3C World Wide Web Consortium. http://www.w3.org/.

[DC02] Douglas Downing and Jeffrey Clark. *Estatística Aplicada*. Editora Saraiva, 2002.

[DRJ00] Sandra G. Dykes, Kay A. Robbins, and Clinton L. Jeffery. An empirical evaluation of client-side server selection algorithms. In *IEEE Conference on Computer Communications – IEEE INFOCOM*, pages 1361–1370, 2000.

[EMT06] Abdelkarim Erradi, Piyush Maheshwari, and Vladimir Tosic. Recovery policies for enhancing web services reliability. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services (ICWS'06)*, pages 189–196. IEEE Computer Society, 2006.

[EPL02] Robert Elfwing, Ulf Paulsson, and Lars Lundberg. Performance of soap in web service environment compared to CORBA. In *APSEC '02: Proceedings of the Ninth Asia-Pacific Software Engineering Conference*, page 84. IEEE Computer Society, 2002.

[FF03] Christopher Ferris and Joel Farrell. What are web services? *Communications of the ACM*, 46(6):31, 2003.

[Fou] Apache Software Foundation. Apache SOAP. http://ws.apache.org/soap/.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[Gro00] Object Management Group. The common object request broker: Architecture and specification (version 2.4), October 2000.

[HS05] Michael N. Huhns and Munindar P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, 2005.

[JCV06] José Geraldo Ribeiro Júnior, Glauber Tadeu S. Carmo, and Marco Tulio Oliveira Valente. Invocation of replicated web services using smart proxies. In *WebMedia '06: Proceedings of the 12th Brazilian symposium on Multimedia and the web*, pages 138–147. ACM Press, 2006.

[JW] JAX-WS. http://java.sun.com/webservices/.

[KK00a] Rainer Koster and Thorsten Kramp. Loadable smart proxies and native-code shipping for CORBA. In *USM - Universal Service Market '00: Proceedings of the Third International IFIP/GI Working Conference on Trends in Distributed Systems*, pages 202–213. Springer-Verlag, 2000.

[KK00b] Rainer Koster and Thorsten Kramp. Structuring QoS-supporting services with smart proxies. In *IFIP/ACM Middleware Conference*, volume 1795 of *Lecture Notes in Computer Science*, pages 273–288. Springer, 2000.

[KR04] Su Myeon Kim and Marcel Catalin Rosu. A survey of public web services. In *13th International World Wide Web Conference*, pages 312–313. ACM Press, 2004.

[Kre01] Heather Kreger. Web Services Conceptual Architecture. *IBM Software Group*, 2001. http://www-3.ibm.com/software/solutions/webservices/pdf/WSCA.pdf.

[Led02] Pascal Ledru. Smart proxies for Jini services. *SIGPLAN Notices*, 37(4):57–61, 2002.

[Lit02] Marin Litoiu. Migrating to Web Services Latency and Scalability. In *Fourth International Workshop on Web Site Evolution*, pages 13–20. IEEE Computer Society, 2002.

[Men02] Daniel A. Menasce. QoS issues in web services. *IEEE Internet Computing*, 6(6):72–75, 2002.

[Mica] Microsoft. Microsoft .NET Web Services Technology. http://www.microsoft.com/net/default.mspx.

[Micb] Sun Microsystems. *JAX-RPC Specification*, 1.0 edition. http://java.sun.com/xml/downloads/jaxrpc.html.

[Mod96] Distributed Component Object Model. http://www.microsoft.com/com/default.mspx, 1996. Version 1.3.

[MR05] Helcio Mello and Noemi Rodriguez. Smart proxies in LuaOrb automatic adaptation and monitoring. In *23o Simpósio Brasileiro de Redes de Computadores*, 2005.

[MS05] Nabor C. Mendonça and José Airton F. Silva. An empirical evaluation of client-side server selection policies for accessing replicated web services. In *ACM Symposium on Applied computing*, pages 1704–1708. ACM Press, 2005.

[OT02] Sven Overhage and Peter Thomas. Ws-specification: Specifying web services using UDDI improvements. In *Web, Web-Services, and Database Systems*, pages 100–119, 2002.

[PF04] Shankar Ponnekanti and Armando Fox. Interoperability among independently evolving web services. In *ACM/IFIP/USENIX International Middleware Conference*, volume 3231 of *Lecture Notes in Computer Science*, pages 331–351. Springer, 2004.

[PG03] Mike P. Papazoglou and Dimitrios Georgakopoulos. Service oriented computing. *Communications of the ACM*, 46(10):24–28, 2003.

[PM05] Paulo Pires and Marta Matoso. Tecnologia de serviços web, 2005.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[Sec] OASIS Web Services Security. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss.

[SM04] José Airton Fernandes Silva and Nabor Chagas Mendonça. Implementação e Avaliação Empírica de Políticas de Invocação para Serviços Web replicados. *Dissertação de Mestrado, UNIFOR, Fortaleza*, 2004.

[SMS02] Nuno Santos, Paulo Marques, and Luis Silva. A framework for smart proxies and interceptors in RMI. In *ISCA 15th International Conference on Parallel and Distributed Computing Systems*, September 2002.

[SPSPMJP06] Jorge Salas, Francisco Perez-Sorrosal, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. WS-Replication: a framework for highly available web services. In *15th World Wide Web Conference*, pages 357–366. ACM, 2006.

[Vay01] Julien Vayssiere. Transparent Dissemination of Adapters in Jini. In *Third International Symposium on Distributed Objects and Applications*, pages 95–104. IEEE Computer Society, 2001.

[VBB+91] Paulo Verissimo, Peter Barrett, Peter Bond, Andrew Hilborne, Luis Rodrigues, and Douglas Seaton. The extra performance architecture (xpa).

Delta-4-A Generic Architecture for Dependable Distributed Computing, pages 211–266, Springer Verlag, Nov. 1991.

[VSC03] Marco Túlio Oliveira Valente, João Paulo Santos, and César Francisco Moura Couto. Interceptação de Métodos Remotos em Java RMI. In *VII Simpósio Brasileiro de Linguagens de Programação*, pages 50–63, 2003.

[VVJ06] Bart Verheecke, Wim Vanderperren, and Viviane Jonckers. Unraveling crosscutting concerns in web services middleware. *IEEE Software*, 23(1):42–50, 2006.

[Wal99] Jim Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, July 1999.

[Wir77] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntatic definitions? *Communications of ACM*, 20(11):822–823, 1977.

[WP] Web Services Policy 1.2 Framework (WS-Policy). http://www.w3.org/Submission/WS-Policy/.

[WPS01] Nanbor Wang, Kirthika Parameswaran, and Douglas C. Schmidt. The Design and Performance of Meta-Programming Mechanisms for Object Request Broker Middleware. In *6th USENIX Conference on Object-Oriented Technologies & Systems (COOTS)*, pages 219–232, 2001.

[WR] Web Services Reliable Messaging Protocol (WS-ReliableMessaging). http://xml.coverpages.org/ni2005-04-19-a.html.

[WRW96] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java system. In *2nd Conference on Object-Oriented Technologies & Systems*, pages 219–232. USENIX, 1996.

[YS97] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Transactions Programing Languages Systems*, 19(2):292–333, 1997.

[ZBN+04] Liangzhao Zeng, Boualem Benatallah, Anne H.H. Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. QoS-Aware middleware for

web services composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo