Pós-Graduação em Ciência da Computação

“DIRETRIZES E UM UTILITÁRIO PARA AVALIAÇÃO DE DESEMPENHO DE TOOLKITS WEB SERVICES”

Por

ANA CAROLINA CHAVES MACHADO

Dissertação de Mestrado

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, AGOSTO/2006

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

ANA CAROLINA CHAVES MACHADO

"DIRETRIZES E UM UTILITÁRIO PARA AVALIAÇÃO DE DESEMPENHO DE TOOLKITS WEB SERVICES"

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

ORIENTADOR(A): Prof. Dr. Carlos A. G. Ferraz

RECIFE, AGOSTO/2006

Machado, Ana Carolina Chaves
Diretrizes e um utilitário para avaliação de desempenho de toolkits web services / Ana Carolina Chaves Machado. – Recife : O Autor, 2006.
iv, 135 folhas : il., fig.,tab.

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

Inclui bibliografia.

1.Sistemas distribuídos. I. Título.

004.36 CDD (22.ed.) MEI2008-055

DEDICATÓRIAS

para minha família.

AGRADECIMENTOS

Agradeço ao Prof. Dr. Carlos André Guimarães Ferraz pela orientação no mestrado. Pelo seu comprometimento e presença nas dúvidas técnicas, informações sobre a vida acadêmica, ensino de disciplinas, enfim todo o incentivo e apoio durante o desenvolvimento deste trabalho.

Agradeço aos professores que forneceram conhecimentos nas disciplinas cursadas durante o mestrado. Agradeço a Profª Marcilia Andrade pela sua amizade e orientação.

Agradeço aos meus colegas de turma pelo compartilhamento de informações e pelos momentos de descontração durante essa jornada.

Agradeço ao C.E.S.A.R (Centro de Estudos e Sistemas Avançados do Recife) pela liberação de algumas horas do meu trabalho para me dedicar às pesquisas e ao Tribunal de Contas do Estado de Pernambuco (TCE-PE), princiaplmente a Saulo Malincônico, pela ajuda, compreensão e motivação.

Agradeço ao meu esposo Cláudio Morais e toda minha família pelo companheirismo e ajuda e por aceitar minha ausência em muitos momentos importantes da nossa família.

DIRETRIZES E UM UTILITÁRIO PARA AVALIAÇÃO DE DESEMPENHO DE TOOLKITS WEB SERVICES[1]

Autora: Ana Carolina Chaves Machado
Orientador: Prof. Dr. Carlos André Guimarães Ferraz

RESUMO

A tecnologia *Web Services* está se tornando a mais importante solução para prover a comunicação entre aplicações heterogêneas, contribuindo enormemente para o avanço na área de desenvolvimento de sistemas distribuídos. Uma conseqüência dessa popularidade é a existência de mais de setenta *Web Services toolkits* disponíveis para uma variedade de plataformas e linguagens de programação. Além disso, várias aplicações em áreas como, por exemplo, *e-commerce*, computação científica, saúde e finanças estão sendo expostas como *Web Services*. Dessa forma, o fato de estar sendo adotada por muitas empresas como a infra-estrutura para desenvolver seus sistemas, aumenta a demanda pela sua eficiência, uma vez que desempenho é um importante parâmetro da qualidade do serviço.

O problema é que o desempenho de *Web Services* é uma questão em aberto, uma vez que sua eficiência foi sacrificada para prover simplicidade, interoperabilidade e flexibilidade. Dessa forma, os desenvolvedores deveriam avaliar as condições de desempenho das aplicações *Web Services*, pois sua ineficiência pode limitar sua aplicabilidade em algumas situações.

O principal objetivo dessa dissertação é viabilizar a avaliação de desempenho de *Web Services toolkits*, propondo diretrizes que foram desenvolvidas baseando-se nos gargalos de desempenho de *Web Services*. A partir dessas diretrizes, foi elaborado um processo que tem como objetivo uniformizar a avaliação de desempenho de *toolkits* e facilitar a escolha do *toolkit* "ideal" para desenvolver uma aplicação. Também será apresentado o utilitário JWSPerf (*Java Web Service Performance*) que, juntamente com outras ferramentas, automatiza algumas tarefas desse processo, reduzindo o tempo e os custos necessários para sua execução.

Palavras-chave: Web Services, Toolkits e Avaliação de Desempenho.

[1] Dissertação de Mestrado em Ciência da Computação, Centro de Informática, Universidade Federal de Pernambuco, Recife, PE, 2006.

GUIDELINES AND UTILITY FOR PERFORMANCE EVALUATION OF WEB SERVICES TOOLKITS[2]

Author: Ana Carolina Chaves Machado
Adviser: Prof. Dr. Carlos André Guimarães Ferraz

ABSTRACT

The Web Services technology is becoming the most important solution to provide the communication between heterogeneous applications, contributing enormously for the advance in the area of distributed systems development. A consequence of this popularity is the existence of more than seventy Web Services toolkits available for a variety of platforms and programming languages. Moreover, various applications in areas such as e-commerce, scientific computation, health and finance have been exposed as Web Services. Therefore, the fact that it is being adopted for many companies as the infrastructure to develop its systems, increases the demand for its efficiency, because performance is an important parameter of quality of service.

The problem is that Web Services performance is an open question, because its efficiency was sacrificed to provide simplicity, interoperabilidade and flexibility. Therefore, the developers would have to evaluate the conditions of performance of the Web Services applications, because its inefficiency can limit its applicability in some situations.

The main objective of this dissertation is to make the performance evaluation of Web Services toolkits, publishing guidelines that had been developed based on the performance overheads of Web Services. Based on guidelines, a process was elaborated that has as objective to standardise the performance evaluation of toolkits and to facilitate the choice of the "ideal" toolkit to develop an application. Also it will present the utility JWSPerf (Java Web Service Performance) that, together with other tools, automatizes some tasks of this process, reducing the time and costs for its execution.

Keywords: Web Services, Toolkits and Performance Evaluation.

[2] Master of Science dissertation in Computer Science, Informatics Center, Federal University of Pernambuco, Recife, PE, 2006.

SUMÁRIO

LISTA DE TABELAS

LISTA DE FIGURAS

1 Introdução

1.1 Motivação

Web Services têm muitas qualidades como possível comunicação através de *firewalls* e promover a integração entre aplicações distribuídas na *Internet*. Devido a essas vantagens, a tecnologia *Web Services* está se tornando a mais importante solução para prover a comunicação entre aplicações heterogêneas, contribuindo enormemente para o avanço na área de desenvolvimento de sistemas distribuídos.

Uma conseqüência dessa popularidade é a existência de mais de setenta *Web Services toolkits* disponíveis para uma variedade de plataformas e linguagens de programação. Além disso, várias aplicações em áreas como, por exemplo, *e-commerce*, computação científica, saúde e finanças, estão sendo expostas como *Web Services*. Dessa forma, o fato de estar sendo adotada por muitas empresas como a infra-estrutura para desenvolver seus sistemas, aumenta a demanda pela sua eficiência, uma vez que desempenho é um importante parâmetro da qualidade do serviço.

O problema é que o desempenho de *Web Services* é uma questão em aberto, uma vez que sua eficiência foi sacrificada para prover interoperabilidade. Os gargalos de desempenho de *Web Services* se originam do projeto e implementação dos *toolkits*

utilizados, da escolha do protocolo de transporte da mensagem SOAP e dos gargalos inerentes ao próprio protocolo SOAP. Nesse contexto, algumas questões são pertinentes:

1) Qual é o desempenho apresentado pelas várias implementações *Web Services*?
2) Quais os gargalos introduzidos pelos protocolos SOAP e HTTP? Quais desses gargalos podem ser removidos através de melhores implementações?
3) Qual *Web Services toolkit* usar para desenvolver e expor um serviço?

Os desenvolvedores deveriam verificar as condições de desempenho das aplicações *Web Services*, pois sua ineficiência pode limitar sua aplicabilidade em algumas situações. Na maioria dos casos, uma implementação "ingênua" pode consumir muito tempo de processamento.

Dessa forma, é necessário avaliar o desempenho de *toolkits* antes de desenvolver os sistemas, a fim de identificar o mais apropriado para atender aos seus requisitos de eficiência. Porém, o processo de avaliação de desempenho pode consumir muito tempo analisando a documentação e escrevendo diferentes códigos para cada *toolkit* e demandar pessoas com experiência na tecnologia.

1.2 Objetivos e Metodologia

O principal objetivo dessa dissertação é viabilizar a avaliação de desempenho de *Web Services toolkits*, publicando diretrizes que foram desenvolvidas baseando-se nos gargalos de desempenho de *Web Services* (Figura 1.1). Observe-se que a Figura 1.1 descreve a metodologia de desenvolvimento adotada neste trabalho: 1) organizar os gargalos de desempenho; 2) publicar as diretrizes; 3) propor o processo de avaliação; e 4) desenvolver o utilitário JWSPerf.

Uma contribuição importante é a organização dos gargalos de desempenho de *Web Services toolkits*, permitindo que qualquer desenvolvedor entenda os fatores que

influenciam sua eficiência e identifique as possíveis otimizações para melhorar seu desempenho.

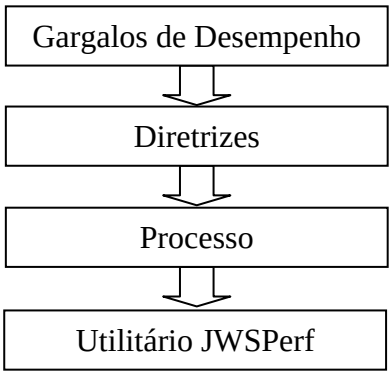

Figura 1.1 - Estratégia para avaliação de desempenho de *Web Services*

As diretrizes publicadas guiam a avaliação, focando nos principais aspectos de um *toolkit* que devem ser analisados. Além das diretrizes, são apresentadas recomendações para projetar a interface WSDL sem afetar a interoperabilidade da aplicação.

A partir das diretrizes, foi elaborado um processo que tem como objetivo uniformizar a avaliação de desempenho de *toolkits* e facilitar a escolha do *toolkit* "ideal" para desenvolver uma aplicação. De forma geral, o processo representa um guia prático composto por um conjunto de tarefas para executar a avaliação.

Uma vez que o processo de avaliação pode demandar muito tempo, verificou-se a necessidade de automatizar algumas de suas tarefas, principalmente os passos referentes à parte de implementação, compilação, execução da aplicação cliente e coleta de métricas. Nesse contexto, foi desenvolvido o utilitário JWSPerf (*Java Web Service Performance*) de código aberto e implementado em Java.

O utilitário é simples, fácil de usar e suporta três *Web Services toolkits* – Axis, *Systinet Server for Java* (SSJ) e *Java Web Services Developer Pack* (JWSDP). O utilitário JWSPerf, juntamente com outras ferramentas, automatiza parte desse processo, reduzindo o tempo e os custos necessários para sua execução.

1.3 Organização da Dissertação

A dissertação está organizada em 7 capítulos. Neste capítulo inicial foi apresentada a motivação para o trabalho, seguida da descrição dos objetivos e organização da dissertação.

O capítulo 2 apresenta uma introdução sucinta da tecnologia *Web Services*, focando principalmente nas principais decisões de projeto que desenvolvedores devem tomar durante o projeto da interface WSDL, pois afetam a estruturação e as regras de serialização e deserialização das mensagens.

O capítulo 3 procura dar uma visão estruturada dos trabalhos que investigaram a ineficiência de *Web Services*, detalhando seus gargalos e listando as possíveis otimizações para tornar os serviços mais eficientes.

O capítulo 4 apresenta as diretrizes para avaliação de desempenho de *Web Services toolkits*, propostas para que um arquiteto entenda o comportamento do *toolkit* sendo analisado e identifique seus gargalos de desempenho.

O capítulo 5 descreve o processo proposto e suas tarefas para avaliar o desempenho de *Web Services toolkits* e o utilitário JWSPerf (*Java Web Services Performance*), que visa automatizar uma parte desse processo.

O capítulo 6 apresenta experimentos, resultados e conclusões da avaliação de desempenho dos *toolkits* suportados pelo utilitário JWSPerf utilizando o processo proposto. A aplicação-teste utilizada como *benchmark* também é apresentada.

O capítulo 7 relata as conclusões finais dessa dissertação e apresenta propostas para trabalhos futuros que possam vir a contribuir para o crescimento da área.

2 A Tecnologia *Web Services*

2.1 Introdução

Os avanços recentes na padronização das tecnologias *Internet* têm impulsionado a publicação de protocolos baseados em XML que viabilizam a interoperabilidade entre aplicações em diferentes linguagens e plataformas, dessa forma encorajando a integração entre sistemas.

Nesse contexto, surgiu a tecnologia *Web Services* como uma solução baseada em tecnologias padrão para integrar sistemas distribuídos através da *Internet*. Atualmente, essa tecnologia é responsável pelo crescimento na área de desenvolvimento e integração de sistemas.

Existem várias definições para *Web Services*, porém segundo o W3C (*World Wide Web Consortium*), *Web Service* é uma aplicação identificada por uma URI (*Uniform Resource Identifier*), cuja interface pública e estilo de *binding* são definidos e descritos usando XML [Austin et al., 2004]. Sua definição pode ser descoberta por outras aplicações que devem interagir com esses *Web Services* de acordo com sua definição, usando mensagens XML transportadas por protocolos *Internet*.

Web Services baseiam-se em padrões, que são independentes de plataforma, e não é um *middleware* de objetos distribuídos como CORBA (*Common Object Request Broker Architecture*) e Java RMI (*Java Remote Method Invocation*), portanto não suportam algumas características como coletor de lixo de objetos remotos distribuídos e referências remotas. Dessa forma, a interface do serviço deve ser orientada a documentos, não expondo suas operações usando os conceitos de orientação a objetos como *overloading*, polimorfismo e herança.

De forma geral, para garantir a interoperabilidade entre as partes que necessitam se comunicar, as mesmas devem concordar com relação a alguns pontos:

1. Formatação dos dados;
2. Regras para serializar e deserializar o estado de uma entidade nesse formato;
3. Protocolo de comunicação;
4. Protocolo de transporte da mensagem.

Cada um desses pontos é garantido por uma ou mais camadas que compõem a pilha da tecnologia *Web Services* apresentada na próxima seção.

2.2 Arquitetura Orientada a Serviços

Web Services promovem um ambiente de integração que é interoperável e de baixo acoplamento, pois suas características originam da arquitetura conceitual chamada SOA (*Service-Oriented Architecture*), que é uma maneira de projetar *software* para fornecer serviços às aplicações de usuários finais ou para outros serviços através de interfaces publicáveis e descobertas [McGoven et al., 2003].

SOA emprega o paradigma *"find, bind and execute"*, que pode ser entendido como "localizar, conectar e executar". As entidades necessárias para implementar esse paradigma são (ver Figura 2.1):

1. **Consumidor do Serviço (*Service Consumer)***: é uma aplicação, serviço ou algum outro tipo de software que demanda por um serviço. É a entidade que inicia o processo de busca no registro por um serviço para, em seguida, acoplá-

lo e executá-lo. Para executar o serviço, o consumidor precisa enviar uma requisição ao serviço no formato estabelecido no contrato.

2. **Provedor do Serviço (*Service Provider)***: é o serviço em si. É a entidade que recebe as requisições dos consumidores e executa a tarefa solicitada, podendo ser um componente, um módulo, um sistema de um *mainframe* ou qualquer outro tipo de *software* que se registrou para fornecer um serviço aos consumidores mediante um contrato. Para que possa receber solicitações, todo serviço precisa ter um endereço na rede.
3. **Registro (*Service Registry)***: é o repositório onde os serviços são registrados e onde os consumidores vão procurar pelos serviços que atendam às suas necessidades. Quando o registro encontra um serviço compatível com a solicitação, o endereço do serviço é retornado para o consumidor, que pode, então, executá-lo.

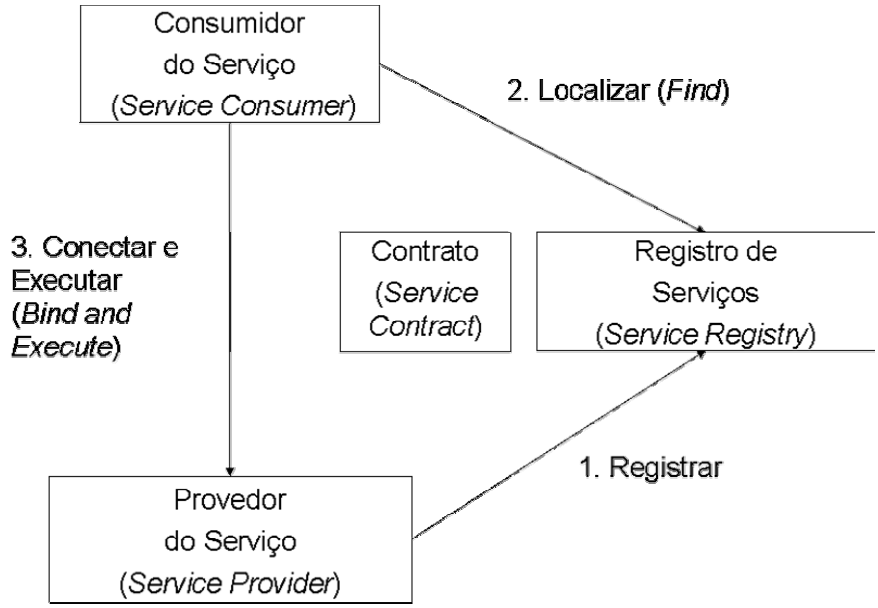

Figura 2.1 - Entidades do paradigma "find, bind and execute"

O contrato dita a forma como as duas partes devem se comunicar, além de estabelecer um conjunto de pré-condições e pós-condições necessárias à execução do serviço.

Para tornar um serviço disponível aos possíveis consumidores, um Provedor de Serviço precisa "publicá-lo" no Registro de Serviços, conforme mostra o passo 1. Para

utilizar um serviço, o Consumidor primeiro busca o serviço no Registro (passo 2), que por sua vez, retorna o endereço onde o serviço se encontra além do contrato. O contrato vai definir as regras para a utilização do serviço. De posse do endereço e do contrato do serviço desejado, o consumidor pode então "conectar-se" ao servidor e "executar" o serviço, conforme mostra o passo 3.

Do ponto de vista técnico, *Web Services* são simplesmente um conjunto de tecnologias que podem ser usadas para implementar o paradigma da arquitetura SOA [McGoven et al., 2003]. A Figura 2.2 ilustra a pilha conceitual de *Web Services*, categorizando suas tecnologias padrões em um modelo em camadas.

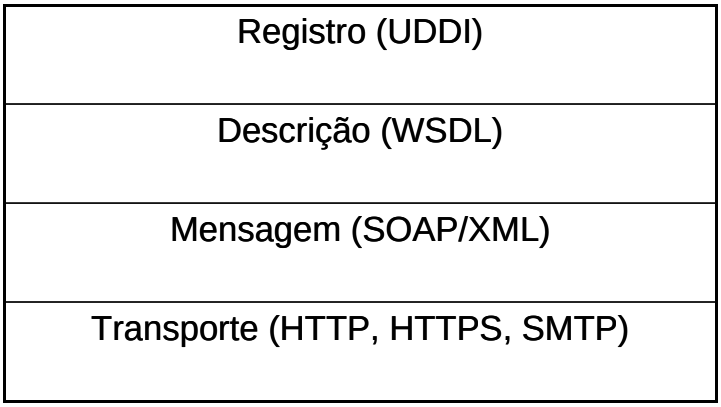

Figura 2.2 - Pilha das tecnologias de *Web Services*

Camada de Transporte

A principal função da camada de transporte é transferir dados de uma máquina para outra utilizando um protocolo para transportar a mensagem. *Web Services* podem usar múltiplos protocolos para transferir os dados como, por exemplo, HTTP, SMTP e FTP.

HTTP é o protocolo de transporte mais comumente adotado para transportar os dados *Web Services*, pois o mesmo freqüentemente não é bloqueado por *firewalls*, que tendem a ser estruturas de segurança de natureza bastante seletiva no que diz respeito ao tráfego de informações.

Camada de Empacotamento das Mensagens

No contexto de integração entre aplicações distribuídas, é necessário que o estado das entidades seja enviado sobre a rede seguindo um formato e usando um protocolo conhecidos. No caso de *Web Services*, isso significa que dentro da mensagem SOAP, o documento XML representando o estado deve estar no formato padrão, para que ambas as partes possam entender e interpretar corretamente as informações. A camada de empacotamento das mensagens é a responsável pela formatação dos dados transmitidos entre o cliente e o servidor sobre o protocolo de transporte.

A especificação padrão adotada por essa camada é o protocolo SOAP (*Simple Object Access Protocol*) – protocolo "leve" para a troca de informações em um ambiente descentralizado e distribuído [Box et al., 2000]. A especificação SOAP define três pontos importantes:

1. Formato em que as mensagens XML devem ser estruturadas, incluindo as mensagens de erro;
2. Os mecanismos de ligação ao protocolo de transporte da mensagem, ou seja, as regras que ditam como uma mensagem deve ser enviada sobre um protocolo em particular, chamados de SOAP *Binding*;
3. As regras de (de)serialização, chamadas de SOAP *Encoding*, para mapear as estruturas de dados da aplicação em XML e, vice-versa.

Uma mensagem SOAP é um documento XML que contém três principais elementos (ver Figura 2.3):

- **Envelope**: é o elemento raiz da mensagem XML e informa que a mensagem sendo processada se trata de uma mensagem SOAP;
- **Cabeçalho**: é um elemento opcional usado para carregar informações auxiliares para os serviços, por exemplo, de autenticação, segurança, transação e roteamento. Qualquer nó na cadeia de processamento da mensagem SOAP pode adicionar ou remover itens do cabeçalho, como também pode ignorá-los caso não sejam entendidos. Caso o cabeçalho esteja presente, o mesmo deve ser o primeiro elemento dentro do envelope;

- **Corpo**: é a parte principal da mensagem porque contém os dados que devem ser enviados. Esses dados podem representar uma chamada remota, descrevendo os parâmetros ou valor de retorno, ou um simples documento XML. Esses dois estilos de codificação do corpo das mensagens serão apresentados na Seção 2.3.1. Além desses, o corpo pode conter uma mensagem de erro indicando que houve algum problema no processamento da mensagem.

```
<soap:Envelop xmlns:soap="http://schemas.xmlsoap.org/soap/envelop/">
    <soap:Header>
        <!-- elemento(s) do cabeçalho -->
    </soap:Header>
    <soap:Body>
        <!-- chamada RPC ou um Documento XML -->
    <soap:Body>
</soap:Envelop>
```

Figura 2.3 - Estrutura das mensagens SOAP

Camada de Descrição do Serviço

Essa camada tem como objetivo responder as seguintes questões:

1. Quais operações um serviço oferece?
2. Quais dados devem ser enviados para invocar uma determinada operação?
3. Qual protocolo usar para invocar um serviço?

A descrição de um serviço consiste em especificar em detalhes suas operações, as mensagens que podem ser enviadas, os tipos de dados usados nessas mensagens, o estilo das mensagens, o protocolo que o consumidor deve usar para acessar o serviço e a sua localização.

A tecnologia padrão adotada para definir o contrato do serviço como um conjunto de endereços de rede (*endpoints*) que operam sobre as mensagens formatadas é a especificação WSDL (*Web Services Description Language*). O consumidor do serviço usa a descrição do serviço em uma das seguintes maneiras:

1. *Early Binding*: durante o desenvolvimento, o consumidor gera o *stub* do serviço a partir da interface WSDL, onde o consumidor do serviço faz referências estáticas ao mesmo em tempo de compilação;

2. *Late Binding*: utiliza o conceito de *proxy* gerado dinamicamente em tempo de execução a partir da interface WSDL.

```
<definitions>

 <types>
   <schema>
   </schema>
 </types>

 <message>
  <part>
  </part>
 </message>

 <portType>
  <operation>
    <input  message="">
    </input>
    <output message="">
    </output>
    <fault></fault>
  </operation>
 </portType>

 <binding>

  <soap:binding transport="" style="" />

  <operation>
    <input>
      <soap:body use="" />
    </input>
    <output>
     <soap:body use="" />
    </output>
  </operation>
 </binding>

 <service>
  <port>
    <soap:address location="" />
  </port>
 </service>

</definitions>
```

Figura 2.4 - Estrutura de um documento WSDL

Antes de entender como projetar uma interface WSDL (ver Seção 2.3.1), é importante apresentar a estrutura e a descrição dos principais elementos que compõem um documento WSDL (Figura 2.4):

- <***definitions***>: elemento raiz do documento WSDL;

- <***types***>: agrupa um ou mais elementos `<schema>` que contêm a declaração dos tipos de dados usados pelos elementos `<message>`, independentemente de linguagem e plataforma;
- <***message***>: define o formato das mensagens que devem ser trocadas dentro do corpo de uma mensagem SOAP, pois contém os parâmetros de entrada ou valor de retorno de um serviço. Cada elemento `<message>` pode ter zero ou mais elementos `<part>`, onde cada `<part>` tem um nome e um atributo *type* ou *element*;
- <***operation***>: esse elemento é uma definição abstrata de uma operação suportada pelo serviço em termos de suas mensagens de entrada e saída. A mensagem de entrada é definida pelo elemento `<input>` e a de saída é definida pelo elemento `<output>`;
- <***portType***>: define o conjunto de operações, ou seja, representa a interface do serviço;
- <***binding***>: representa a implementação concreta do elemento `<portType>` usando um determinado protocolo como, por exemplo, SOAP, que é o mais adotado. Se o serviço suportar mais de um protocolo, o arquivo WSDL deverá ter um elemento `<binding>` para cada protocolo;
- <***service***>: coleção de elementos `<port>`, onde cada `<port>` descreve a localização de rede para um elemento `<binding>`.

Camada de Registro

A camada de registro adota a especificação UDDI (*Universal Description, Discovery, and Integration*) para oferecer uma maneira padrão de publicação das informações de um *Web Services* e os mecanismos para descobrir quais serviços atendem às necessidades de um determinado consumidor. Um repositório UDDI é semelhante a um serviço de páginas amarelas, pois fornece operações de registro e descoberta de serviços a partir de determinadas características.

Web Services suportam o conceito de descoberta dinâmica de serviços. Um consumidor de um serviço pode usar um registro para encontrar os serviços de seu interesse. Os registros UDDI são *Web Services* que expõe sua API (*Application Program Interface*) como um conjunto de mensagens SOAP bem definidas. Para cada serviço registrado no repositório UDDI, são mantidos o contrato e informações sobre o seu negócio.

2.3 Desenvolvendo *Web Services*

Muitas aplicações *Web Services* estão sendo publicadas sem que os desenvolvedores tenham qualquer conhecimento sobre as tecnologias XML, SOAP e WSDL. Isso é perfeitamente viável, pois os *toolkits* atuais disponibilizam funcionalidades para gerar dinamicamente a interface WSDL do serviço e toda a camada de comunicação (*stubs* e *skeletons*) a partir do código da aplicação. Essa forma de desenvolvimento é chamada *Bottom-up*.

Apesar de ser uma maneira fácil e rápida de desenvolver, a mesma não é a mais apropriada quando o objetivo é alcançar a interoperabilidade entre aplicações heterogêneas, pois os *toolkits* podem não adotar as mesmas regras para gerar a interface WSDL e as mensagens SOAP. Além disso, o desenvolvedor “aceita” o funcionamento padrão desses *toolkits* que muitas vezes não está configurado para executar de forma eficiente e interoperável.

Uma segunda alternativa de desenvolvimento é a *Top-down* que consiste em iniciar pelo projeto da interface WSDL, pois a mesma representa o contrato que o cliente e o servidor devem aderir. Criar o contrato WSDL refere-se ao processo de projetar a interface baseando-se nas mensagens XML que devem ser trocadas, em vez de basear-se no código do serviço.

Uma vez projetada a interface WSDL, a mesma deveria ser usada para gerar os *skeletons* do lado do servidor que serão, por sua vez, usados como *templates* para a implementação do serviço. Essa é a maneira mais indicada de desenvolvimento quando

se deseja projetar uma aplicação *Web Services* corporativa, pois problemas de interoperabilidade podem ser evitados manipulando diretamente o arquivo WSDL.

A próxima seção tem como objetivo apresentar as regras e configurações que devem ser aplicadas durante o projeto da interface WSDL, uma vez que essa não é uma tarefa fácil.

2.3.1 Projetando a Interface WSDL

Essa seção explica as duas principais decisões que os desenvolvedores devem tomar durante o projeto da interface WSDL: configuração dos parâmetros *style* e *use*. Esses parâmetros são de especial importância porque afetam a formação do corpo das mensagens SOAP e as regras de codificação adotadas, porém os mesmos são freqüentemente desconhecidos pelos desenvolvedores. O objetivo é mostrar como configurar esses parâmetros e esclarecer a confusão sobre os diferentes formatos do corpo da mensagem SOAP.

O primeiro parâmetro é o atributo *style* (ver Tabela 2.1), que controla a estruturação do corpo da mensagem. *Web Services* podem expor suas operações seguindo os seguintes estilos de codificação:

1. ***RPC***: nesse estilo, o cliente invoca o método no servidor enviando no corpo da mensagem SOAP todas as informações necessárias para a sua execução e recebe a resposta da mesma maneira. Dessa forma, a estrutura do corpo da mensagem SOAP deve conter a chamada remota, indicando o nome do método e os parâmetros ou valor de retorno.
2. ***Document***: esse estilo reflete o uso mais natural de XML e é mais flexível, pois documentos XML são passados como entrada e saída dos serviços.

A segunda decisão, que especifica as regras de serialização e deserialização dos dados, é a configuração do atributo *use* (ver Tabela 2.2). As duas possíveis opções são:

1. ***Literal***: baseia-se num pré-acordo do esquema XML que define as regras para codificar e interpretar o corpo da mensagem SOAP. A tecnologia *XML Schema* é utilizada para definir os tipos dos dados.

2. ***Encoded***: baseia-se em um conjunto de regras definidas na especificação do protocolo SOAP. Essa codificação não é a obrigatória, e também não existe um padrão, pois depende de cada *toolkit*.

Tabela 2.1 - Regras para configurar o estilo de codificação do documento WSDL

Regra	RPC	*Document*
Atributo *style* do elemento `<soap:binding>`	style="rpc"	style="document"
Quantidade de elementos `<part>` dentro do elemento `<message>`	Pode conter zero ou mais elementos `<part>`, cada um contendo o atributo *type*.	Deve conter zero ou um único elemento `<part>` contendo um atributo *element*.

Além de configurar os valores dos atributos *style* e *use*, outras regras são necessárias para projetar a interface WSDL seguindo um desses estilos (ver Tabelas 2.1 e 2.2). Para as mensagens *Encoded*, deve-se configurar o atributo *encodingStyle* com uma URL que especifique as regras adotadas para codificar e interpretar o corpo da mensagem, enquanto que uma mensagem *Literal* adota um esquema XML como regra.

Tabela 2.2 - Regras para configurar o atributo *"use"* do documento WSDL

Regra	Encoded	Literal
Atributo *use* do elemento **`<soap:body>`**	use="encoded"	use="literal"
Outros atributos do elemento **`<soap:body>`**	encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"	

Independentemente da configuração adotada, as partes têm que concordar sobre o mesmo formato da mensagem SOAP e mecanismo de codificação usado para que a mensagem seja corretamente processada. A partir das possíveis configurações desses atributos, existem quatro combinações de estilo de codificação:

- *RPC/Encoded*;
- *RPC/Literal*;
- *Document*/*Encoded*;
- *Document*/*Literal*.

Adiciona-se a essas combinações, o estilo *Document/Literal Wrapped*, também chamado de *Literal/Wrapped*, definido pela Microsoft, porém o mesmo não é um estilo oficialmente documentado. *Document/Literal Wrapped* é uma convenção de programação que simula o estilo RPC, mas produz mensagens no estilo *Document/Literal*.

Embora *Document/Literal Wrapped* não seja um estilo oficial da especificação WSDL, o mesmo é composto por um conjunto de regras que devem ser seguidas durante o projeto da interface WSDL [Manes, 2004]:

1. A definição da mensagem de entrada e saída deve conter um único elemento **<**`part`**>** que deve obrigatoriamente conter um atributo *element* (não um *type*) com o mesmo nome da operação;
2. O nome do *element* referenciado no elemento `<part>` deve estar definido na seção `<types>` do documento WSDL como um tipo complexo que é uma seqüência de elementos, onde cada elemento dentro da seqüência representará um parâmetro do serviço;
3. Na seção **<**`binding`**>**, o elemento `<soap:binding>` deveria configurar o seu atributo *"style=document"* e o elemento `<soap:body>` deveria configurar o atributo "*use=literal"* e nada mais.

Dessa forma, existem cinco combinações para escolher durante o projeto da interface WSDL. A Diretriz 1 que será apresentada no Capítulo 4, aborda cada uma dessas combinações e apresenta as implicações no desempenho e na interoperabilidade decorrentes da seleção de uma combinação sobre a outra.

2.4 *Web Services Toolkits*

Várias implementações *Web Services* amadureceram rapidamente. As mesmas diferem no seu suporte aos tipos de dados definidos na aplicação, no modo de usar, na linguagem de implementação e, principalmente, no desempenho e suporte a otimizações [Govindaraju et al., 2004].

Atualmente mais de setenta *Web Services toolkits* estão disponíveis para uma variedade de plataformas e linguagens de programação como, por exemplo, Ada, C#, C++, Delphi, Java, Perl, Python e Visual Basic [SoapWare.Org, 2004].

Esse grande crescimento causou alguns problemas de interoperabilidade entre plataformas. Um caso comum era um envelope SOAP gerado por um *toolkit* não ser completamente entendido por um outro, pois os mesmos diferiam no tratamento do cabeçalho de uma mensagem SOAP ou na quantidade de dígitos para representar um tipo de dado decimal ou na geração do arquivo WSDL. Existem esforços dirigidos pelos participantes do fórum *Soap Builders* [SOAP Builders, 2004] para criar padrões de interoperabilidade entre *Web Services toolkits*, mesmo que desenvolvidos por fabricantes diferentes.

A seguir, serão apresentadas algumas implementações *Web Services* mais populares e que foram exploradas pelos trabalhos relacionados apresentados no próximo capítulo.

2.4.1 Apache Axis

Apache eXtensible Interaction System é uma implementação *Web Services* gratuita e de código aberto desenvolvida pela Apache Software Foundation [Apache Axis, 2004]. Surgiu como sucessora da implementação Apache SOAP [Apache SOAP, 2004], por isso também é conhecida como Apache SOAP 3.0.

O objetivo dessa substituição foi criar uma implementação SOAP mais modular, flexível e de alto desempenho. Entre as novas características incorporadas ao Axis, a

principal foi a utilização do *parser* XML SAX (*Simple API for XML*) [SAX, 2004] para melhorar o desempenho.

Axis é a implementação SOAP mais popular escrita na linguagem Java e boa parte das ferramentas de desenvolvimento do mercado incorporam na sua implementação esse *toolkit* como, por exemplo, JBoss, Borland JBuilder, Borland Enterprise Server e JOnAS (*Java™ Open Application Server*) [Apache Axis, 2004].

2.4.2 JWSDP (Java Web Services Developer Pack)

É um *toolkit* gratuito mantido pela Sun Microsystems para acelerar o desenvolvimento de aplicações *Web*, XML e *Web Services* [Sun, 2004]. O JWSDP contém ferramentas, APIs e tecnologias para simplificar a construção de *Web Services* na plataforma Java.

2.4.3 Glue

O *toolkit* Glue da webMethods é uma plataforma comercial para desenvolver aplicações Java com JSP, *Servlet* e *Web Services* [webMethods, 2004]. Com o uso de *plugins*, o Glue pode ser integrado a ambientes de desenvolvimento como JBuilder e Eclipse, permitindo a criação das aplicações *Web Services* através de assistentes de programa (*wizards*).

2.4.4 SSJ (Systinet Server for Java)

Systinet Server for Java é uma solução completa e gratuita desenvolvida pela Systinet para construir aplicações J2EE (*Java 2 Platform, Enterprise Edition*) e *Web Services* [Systinet, 2004]. Esse *toolkit* é fácil de usar, de alto desempenho e constitui um ambiente completo para criar, instalar e gerenciar suas aplicações, uma vez que embute seu próprio servidor de aplicação.

2.4.5 XSOAP

Anteriormente chamado de SoapRMI, foi desenvolvido pelo laboratório Extreme! da Universidade Indiana com o objetivo de estudar o protocolo SOAP aplicado em sistemas que demandam por alto desempenho [Extreme!, 2004]. É um sistema RMI baseado em SOAP, implementado em Java e C++, que permite criar e invocar *Web Services*.

O *parser* XML *Pull Parser* (XPP) [Extreme!, 2004] foi criado durante o desenvolvimento de SoapRMI, a fim de melhorar o seu desempenho ao trabalhar com grandes estruturas de dados. O *parser* XPP2, sucessor do XPP, faz parte agora do XSOAP.

2.4.6 Framework .NET

.NET é o atual *framework Web Services* da Microsoft, substituindo o *toolkit* Microsoft SOAP. Esse *framework* é um conjunto de ferramentas de desenvolvimento de *software* usadas para criar, publicar e consumir *Web Services* [Microsoft, 2004].

Apesar da capacidade de integração e comunicação com outras plataformas por meio de *Web Services*, .NET é centrado no ambiente Microsoft, ou seja, os serviços criados com .NET podem apenas ser instalados em sistemas operacionais da Microsoft. Esse *toolkit* suporta o estilo RPC, porém adota o estilo *Document* como o padrão.

2.4.7 gSOAP

O *toolkit* de desenvolvimento *Web Services* gSOAP – gratuito e de alto desempenho – permite a construção de aplicações *Web Services* em C/C++ [gSOAP, 2004]. A implementação oferece um compilador fácil de usar que gera *stub* e *skeleton* para integrar aplicações existentes em C/C++ com *Web Services* [Engelen and Gallivan, 2002].

2.4.8 bSOAP

bSOAP é uma implementação otimizada do protocolo SOAP para desenvolver aplicações *Web Services* em C++ [Abu-Ghazaler et al., 2004] [Abu-Ghazaler et al., 2004a] [Abu-Ghazaler et al., 2004b].

bSOAP foi desenvolvido com o objetivo de viabilizar a adoção de SOAP em aplicações científicas que demandam por alto desempenho e freqüentemente transmitem grandes *arrays* contendo números ponto flutuante e tipos de dados complexos.

2.5 Considerações Finais

O objetivo desse capítulo não foi simplesmente introduzir ao leitor a definição da tecnologia *Web Services*, uma vez que a mesma encontra-se bastante difundida no mercado e no meio acadêmico, mas dar uma visão geral dos conceitos necessários ao entendimento dos próximos capítulos dessa dissertação, principalmente no que se refere ao projeto de interfaces WSDL.

Com relação à tecnologia *Web Services*, foram apresentados os pontos em que as partes integrantes devem concordar para garantir a integração e cada uma das suas tecnologias – SOAP, WSDL e UDDI – adotadas para implementar o paradigma *"find, bind and execute"* da arquitetura SOA. O resultado da adoção dessas tecnologias ubíquas faz de *Web Services* uma solução independente de plataforma e linguagem.

Existem duas formas de desenvolvimento de aplicações *Web Services*, cada uma com suas vantagens e desvantagens. A primeira é chamada de *Bottom-up* e consiste em gerar a interface WSDL a partir do código da aplicação. Essa forma é a mais usada pelos desenvolvedores, porque além de ser mais rápida, os mesmos não têm qualquer contato com as tecnologias XML e WSDL. Entretanto, as aplicações geradas dessa forma podem estar suscetíveis a problemas de interoperabilidade e/ou eficiência, pois dependem das configurações do *toolkit* utilizado.

A segunda é chamada *Top-down* e consiste em projetar primeiramente a interface WSDL, e em seguida, usar a mesma para gerar os *skeletons* e *stubs*. Essa é a mais indicada quando se deseja projetar aplicações *Web Services* corporativas.

Durante o projeto do contrato WSDL, algumas decisões com relação aos parâmetros *style* e *use* devem ser tomadas a fim de definir a estruturação e regras de codificação das mensagens SOAP. Por isso a prática de desenvolvimento a partir da interface WSDL não é comumente adotada, uma vez que os projetistas não sabem como configurar esses parâmetros nem como aplicar algumas regras com relação à estrutura e quantidade de elementos que devem compor o documento WSDL.

Por fim, foram apresentados oito *Web Services toolkits* para desenvolver aplicações *Web Services* nas linguagens Java e C/C++ que são comumente adotados no mercado e foram estudados nos trabalhos relacionados apresentados no próximo capítulo.

3 Desempenho de *Web Services*

3.1 Introdução

A tecnologia *Web Services* está se tornando uma importante solução para prover a comunicação entre aplicações heterogêneas. Logo, o fato de está sendo adotada pelas empresas como a infra-estrutura para expor seus sistemas, aumenta a demanda pela sua eficiência.

Dessa forma, existem algumas discussões avaliando o desempenho de *Web Services*, porque o objetivo principal do seu projeto foi prover a interoperabilidade entre aplicações *Web Services* distribuídas. Durante a especificação do protocolo SOAP, o desempenho foi sacrificado a fim de obter simplicidade, interoperabilidade, universalidade e flexibilidade.

Os padrões adotados por *Web Services*, apresentados no capítulo anterior, incorporam gargalos adicionais comparados à interação *Web* tradicional, porque sua universalidade introduz um problema: as mensagens SOAP são textuais e seu tamanho é significantemente maior que as mensagens dos protocolos binários, aumentando, dessa forma, os custos de codificação e de comunicação.

De forma geral, as pesquisas na área de desempenho de *Web Services*, além de analisarem sua eficiência em domínios de aplicação, como computação científica e finanças, consistem em:

1) Estudar as vantagens e desvantagens do uso de XML, uma vez que WSDL, SOAP e UDDI baseiam-se nessa tecnologia;
2) Comparar a eficiência de *Web Services* com outros *middleware* como CORBA e Java RMI;
3) Avaliar o desempenho dos diferentes *Web Services toolkits*, implementados em diferentes linguagens;
4) Identificar os gargalos inerentes a *Web Services*, tanto na implementação quanto na camada de comunicação;
5) Desenvolver e aplicar possíveis otimizações para eliminar ou reduzir o impacto desses gargalos e projetar aplicações *Web Services* eficientes.

Para cada uma dessas abordagens, serão apresentados os principais trabalhos realizados, dando ênfase a seus resultados e conclusões. O objetivo é apresentar de forma clara o estado da arte do desempenho de *Web Services* e permitir que qualquer desenvolvedor ou arquiteto possa tomar decisões mais eficientes durante o projeto desses *Web Services toolkits*.

3.2 XML versus Representação Binária

O formato binário e XML são duas formas populares de representação das mensagens transportadas na rede, onde XML tem sido largamente adotado quando a interoperabilidade entre aplicações é necessária, enquanto que a representação binária é usada quando o desempenho é um fator crítico. Como a flexibilidade e o desempenho dos sistemas que se comunicam dependem da representação adotada, estudos têm sido realizados a fim de explorar as vantagens e desvantagens associadas a essas duas representações de dados.

Cai et al. (2002) compararam o tamanho em *bytes* das mensagens transmitidas entre o cliente e o servidor representadas nos formatos XML e binário e depois analisaram o papel da técnica de compressão na redução do tamanho das mensagens, adotando os algoritmos de compressão Zip e XMill [Suciu and Liefke, 2004], que é uma técnica exclusiva para comprimir dados XML. Os resultados obtidos foram que:

1. Sem aplicar qualquer técnica de compressão, as mensagens XML são, em média, cinco vezes maiores que sua representação binária;
2. A mensagem XML comprimida usando o algoritmo Zip é duas vezes maior que a mesma mensagem binária comprimida usando o mesmo algoritmo;
3. A mensagem XML comprimida utilizando o algoritmo XMill é menor que a mensagem binária comprimida com o algoritmo Zip. Porém, segundo os autores, esse resultado só é verdadeiro para mensagens com mais de um *megabyte*;
4. Embora as técnicas de compressão reduzam o tamanho das mensagens, elas aumentam o tempo de resposta devido ao tempo gasto na compressão dos dados. Então, os dados apenas deveriam ser comprimidos quando a largura de banda é limitada.

Hericko et al. (2003) analisaram o custo de espaço em memória e do tempo gasto do processo de (de)serialização binária e XML para as plataformas Java e .NET, com o objetivo de investigar as razões das diferenças de desempenho. Do ponto de vista da serialização binária, a plataforma Java apresentou um desempenho melhor por um fator de, aproximadamente, 1,3 a 2,3 e ocupa 25% menos espaço em memória. Enquanto que, o processo de serialização XML da plataforma .NET foi 65% a 85% mais rápido que Java usando a tecnologia JAXB (*Java Architecture for XML Binding*) e o tamanho das mensagens geradas foi 3% menor.

Kohlhoff e Steele (2003) obtiveram um importante resultado comparando o desempenho de duas representações textuais – XML e FIX (*Financial Information eXchange*) [FIX, 2005] – com a representação binária CDR (*Commom Data Representation*). Analisando o tamanho da mensagem, a latência e a vazão, verificou-se que o protocolo FIX foi mais eficiente e produziu mensagens mais compactas. Dessa forma, um formato de rede baseado em texto pode apresentar um desempenho melhor

que um binário e que o uso de padrões baseados em XML não é o único fator da ineficiência de *Web Services*.

3.3 Comparação entre *Web Services* e outros *Middleware*

Como os desenvolvedores de *software* podem optar entre várias tecnologias de *middleware*, desempenho tem sido um fator decisivo na escolha da solução apropriada para implementar uma aplicação distribuída. Dessa forma, vários estudos comparando o desempenho de *Web Services* com outros *middleware* foram realizados a fim de investigar o *trade-off* entre eficiência e interoperabilidade, pois os mesmos adotam diferentes protocolos e formatos de dados na troca das mensagens. Por exemplo, IIOP (*Internet Inter-Orb Protocol*) e JRMP (*Java Remote Messaging Protocol*) são protocolos binários que agem diretamente sobre TCP/IP, enquanto que *Web Services* transportam comumente as mensagens SOAP sobre o protocolo HTTP.

Uma análise detalhada do tempo de resposta e da vazão de diferentes protocolos RMI foi realizada por Govindaraju et al. (2000), comparando a eficiência de *Web Services* com a de Java RMI e Nexus RMI – implementação da API de Java RMI, porém adota Nexus como protocolo de comunicação. Outra diferença entre esses *middleware* é que Java RMI apenas suporta a interoperabilidade entre aplicações Java e Nexus RMI provê a comunicação entre aplicações Java e C++. Em todos os experimentos, o desempenho de *Web Services* foi dez vezes menor que os demais *middleware*, exceto quando pequenas mensagens eram trocadas, onde Nexus RMI foi o mais ineficiente. No contexto de computação científica de alto desempenho, os autores concluíram que as mensagens SOAP baseadas em XML não são apropriadas para transferir grandes volumes de dados numéricos, mas devido à sua flexibilidade e universalidade, podem ser utilizadas como parte de um sistema multi-protocolo usando SOAP como um protocolo de ‘língua franca’.

Elfwing et al. (2002) realizaram um estudo comparativo do desempenho de *Web Services* e CORBA focando nos aspectos da comunicação entre o cliente e o servidor.

Dois cenários de teste foram explorados: 1) Apenas um cliente enviando requisições ao servidor; 2) Além do cliente, um gerador de carga também enviava requisições simultâneas. A implementação *Web Services* apresentou um tempo de resposta 400 vezes mais lento, atingindo seu limite de saturação de processamento no primeiro cenário. Isso implica que existem gargalos de desempenho na implementação *Web Services* que é independente da carga submetida ao serviço.

Os resultados apresentandos em [Davis and Parashar, 2002] [Devaram and Andresen, 2003] e [Engelen, 2003] também confirmaram que *Web Services* no seu uso direto e "ingênuo" são mais ineficientes que Java RMI e CORBA, porém existem vários esforços que estão sendo avaliados para otimizar seu desempenho (ver Seção 3.6).

Juric et al. (2004) compararam o desempenho do *Web Services toolkit* JWSDP com as tecnologias de tunelamento de Java RMI (HTTP-to-port, HTTP-to-CGI e HTTP-to-Servlet) que podem ser utilizadas para desenvolver aplicações Java distribuídas que obrigatoriamente necessitam se comunicar através de *firewalls*. A vantagem de usar uma dessas tecnologias de tunelamento na comunicação entre aplicações Java RMI existentes, é que nenhuma alteração no seu código é necessária. Porém, nos testes realizados transportando apenas tipos de dados simples, *Web Services* foi três vezes mais eficiente. Diante desses resultados, cabe ao arquiteto decidir entre eficiência, segurança e o custo de implementação.

Gray (2005) avaliou o desempenho de diferentes *middleware* para desenvolver aplicações Java distribuídas – CORBA/IIOP, Java RMI/JRMP, Java RMI/HTTP e *Web Services*/JAX-RPC (*Java API for XML-Based RPC*) – focando na análise de métricas como o número total de pacotes e *bytes* transferidos e o tempo de resposta. Quando o cliente invocava uma operação simples que retorna uma *string* de tamanho fixo, os resultados encontrados foram consistentes com os dos trabalhos anteriores, onde Java RMI/JRMP apresentou o melhor desempenho e *Web Services* foi mais eficiente que Java RMI/HTTP. Porém, utilizando um *array* de tipos de dados simples e com poucos objetos, CORBA/IIOP apresentou o melhor desempenho e Java RMI/HTTP e *Web Services* apresentaram resultados semelhantes. Por fim, avaliando estruturas de dados grandes e complexas, *Web Services* foi o mais ineficiente, diferentemente dos resultados apresentados em [Juric et al., 2004]. A partir desses resultados, pode-se concluir que a

avaliação de desempenho de *Web Services* deve analisar a influência da natureza e do tamanho dos dados e da maneira como os mesmos são organizados em pacotes de rede.

3.4 Desempenho de *Web Services Toolkits*

O desempenho dos *Web Services toolkits* tem sido comparado de diferentes maneiras, principalmente utilizando os mais variados tipos de dados, com o objetivo de investigar a sua eficiência e permitir que os desenvolvedores possam escolher o *toolkit* mais apropriado para expor uma aplicação distribuída.

Davis e Parashar (2002) analisaram a eficiência das implementações *Web Services* – Apache SOAP, Apache Axis, Microsoft SOAP Toolkit, SOAP::Lite versão Perl e SoapRMI – operando sobre o protocolo HTTP e usando os servidores de aplicação Tomcat da Apache e o IIS (*Internet Information Services*) da Microsoft. As comparações foram feitas utilizando um *benchmark* simples com apenas três métodos: um método que nem recebia nem retornava valores, o segundo retornava uma *string* e o último retornava um *array* de inteiros. A partir dos experimentos realizados, os seguintes resultados foram obtidos:

1. Quando o cliente e o servidor rodavam em máquinas distintas, verificou-se um aumento de 200ms no tempo de resposta dos *toolkits* Apache SOAP, Microsoft SOAP Toolkit e SOAP::Lite. Esse aumento foi causado pelos gargalos de comunicação (ver Seção 3.5.5);
2. Microsoft SOAP Toolkit apresentou o melhor resultado quando um *array* de inteiros foi retornado;
3. Em relação ao Apache SOAP, o *toolkit* Apache Axis apresentou o melhor desempenho (ver Seção 2.4.1);
4. O *toolkit* SoapRMI teve um bom desempenho em todos os experimentos e o SOAP::Lite foi o mais ineficiente.

Ng et al. (2003) estudaram o desempenho de três implementações comerciais de *Web Services* rodando sobre o protocolo HTTP, porém seus nomes não foram

mencionados. O objetivo foi avaliar o impacto no desempenho dos diferentes estilos de codificação das mensagens SOAP suportados por cada um dos *toolkit*, como *RPC/Encoded*, *Document*/*Literal* e *Document*/*Encoded*. As métricas de desempenho utilizadas nesse processo de avaliação foram a latência, a vazão e os custos de serialização e deserialização de diferentes de tipos de dados. Além de requerer metade do número de *bytes* para representar as mensagens, o estilo de codificação *Document*/*Literal* apresentou o melhor desempenho, enquanto que o estilo *RPC/Encoded* foi o mais ineficiente. Outro resultado importante foi que os *toolkits* apresentaram diferentes resultados de desempenho para um mesmo estilo de codificação. A Figura 3.1 foi extraída de [Ng et al., 2003] e ilustra a latência dos *toolkits* analisados para enviar mensagens simples, média e complexas usando diferentes estilos de codificação.

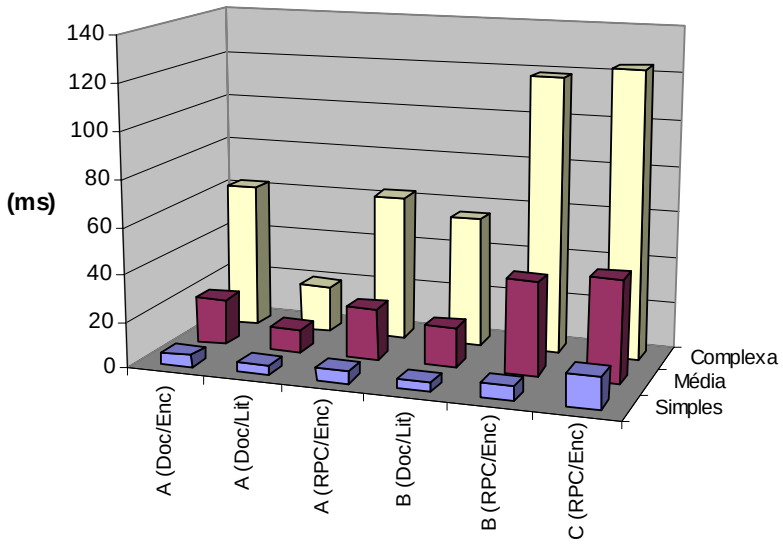

Figura 3.1 - Latência das mensagens SOAP usando diferentes estilos

Govindaraju et al. (2004) comparam o desempenho dos *toolkits* gSOAP, Axis C++, Axis Java, .NET e XSOAP4/XSUL utilizando os tipos de dados comuns em computação científica, como *array* de s*tring* e de números ponto flutuante. O objetivo foi identificar o *toolkit* mais apropriado para trabalhar com dados científicos. Entre os *toolkits* analisados, o gSOAP foi o mais eficiente e o Axis Java, o mais lento.

A Sun Microsystems desenvolveu um *benchmark* para comparar o desempenho do seu *toolkit* JWSDP (*Java Web Services Developer Pack*) com o *framework* .NET da Microsoft. Em todos os experimentos realizados, o *toolkit* JWSDP apresentou um desempenho e escalabilidade superiores [Sun, 2004b]. Em resposta a Sun, a Microsoft realizou os mesmos testes, porém usando uma versão mais atualizada do *toolkit* JWSDP, e obteve resultados totalmente de diferentes, onde o *framework* .NET foi duas a três vezes mais eficiente [Microsoft, 2004b].

Qworks [Qworks, 2004] realizou os mesmos testes de [Sun, 2004b] e [Microsoft, 2004b] e adicionou à avaliação o *toolkit* Axis Java. Usando J2SE (*Java 2 Standard Edition*) versão 1.5, os *toolkits* JWSDP e .NET apresentaram desempenhos semelhantes e foram mais eficientes que o Axis. Porém, usando J2SE versão 1.4 os resultados obtidos foram semelhantes aos publicados em [Microsoft, 2004b], onde .NET apresentou o melhor desempenho. Ambos JWSDP e Axis foram mais eficientes quando a versão do J2SE foi alterada de 1.4 para a versão 1.5, sendo a otimização do JWSDP maior. A partir dos resultados obtidos, detectou-se que a versão da plataforma Java adotada impacta no desempenho do *Web Services toolkit*.

Além de comparar o desempenho dos *toolkits* gSOAP, bSOAP, Axis versão Java e XSUL, Head et al. (2005) propuseram um *benchmark* para quantificar o desempenho desses *toolkits* usando *arrays* de diferentes tamanhos e tipos de dados (ponto flutuante, *string* e inteiros). O *benchmark* era composto por interfaces WSDL que definiam operações projetadas para testar a latência, o desempenho fim-a-fim e os custos de serialização e deserialização separadamente, pois o *toolkit* usado pelo cliente pode ser diferente do usado para implementar o serviço. As chamadas das operações definidas nessas interfaces WSDL foram implementadas para cada *toolkit* avaliado. Os autores apresentaram importantes resultados referentes ao desempenho dessas implementações *Web Services*: 1) o *toolkit* Axis apresentou a maior latência e o gSOAP, a menor; 2) Axis e XSUL apresentaram custos de serialização similares, porém, em termos de deserialização, o Axis é muito mais ineficiente quando *arrays* de tipos de dados simples são usados; e 3) o desempenho fim-a-fim do XSUL degrada consideravelmente usando tipos de dados complexos.

3.5 Gargalos de Desempenho

O uso do protocolo HTTP e de documentos XML formatados segundo o protocolo SOAP promovem a interoperabilidade entre as aplicações, porém incorporam um aumento significativo no tempo de processamento e nos custos de comunicação.

Nessa seção serão apresentados os trabalhos que focaram no levantamento de gargalos de desempenho associados a *Web Services* que são decorrentes de decisões de projeto tomadas durante o desenvolvimento dos *Web Services toolkits*, da escolha do protocolo de transporte da mensagem SOAP e os inerentes ao próprio protocolo SOAP.

Antes de detalhar os gargalos nas subseções seguintes, é importante entender os diferentes estágios [Chiu et al., 2002] que compõem o processo de envio e recebimento das mensagens SOAP (Figura 3.2).

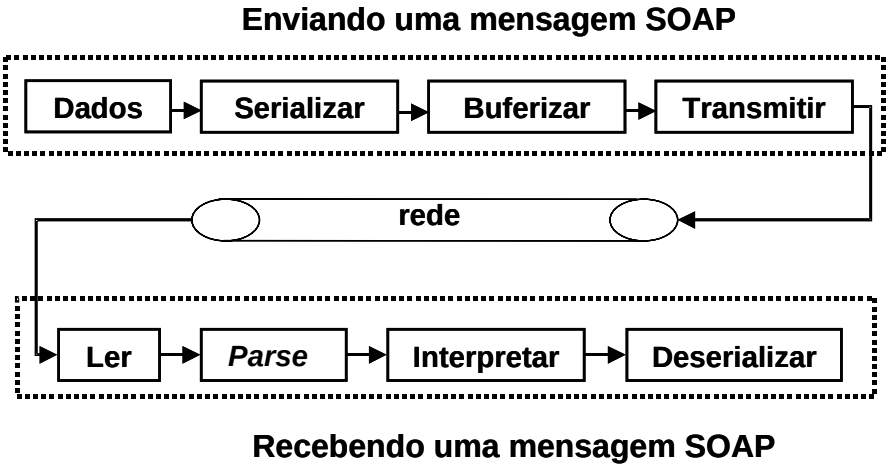

Figura 3.2 - Estágios para enviar e receber uma mensagem SOAP

Para enviar uma mensagem SOAP são necessárias, de forma geral, as fases para varrer as estruturas de dados, serializar os dados para XML, armazenar os dados XML no *buffer* e, por fim, transmitir o conteúdo do *buffer* na rede. O receptor deve ler a mensagem da rede, varrer (*parse*) o documento XML para validar sua sintaxe, interpretar o conteúdo de cada *tag* XML e deserializar o documento XML.

3.5.1 Tamanho da Mensagem

O tamanho da mensagem SOAP tem sido uma métrica de desempenho bastante analisada a fim de calcular o aumento no tamanho das mensagens devido ao uso da tecnologia XML.

A codificação das mensagens no formato de texto expande o tamanho da mensagem por um fator de 4 a 10 vezes em relação à sua representação binária [Govindaraju et al., 2000] [Kohlhoff and Steele, 2003] [Ng et al., 2003] [Ying et al., 2004]. Essa expansão pode ter um impacto significativo na comunicação e no tempo total de execução, pois requer um *buffer* de memória maior, mais largura de banda e mais processamento [Engelen, 2003].

3.5.2 Escolha do Parser XML

O processamento de documentos XML está assumindo uma grande importância nas infra-estruturas de tecnologia da informação nos tempos atuais e o cenário de uso mais conhecido são *Web Services*, pois SOAP é um protocolo baseado em XML e suas mensagens têm que ser varridas (*parsed)* e interpretadas antes de serem invocadas. Além das mensagens SOAP, as tecnologias WSDL, utilizada para definir as operações dos serviços, e XML *Schema*, utilizada para definir os tipos de dados das mensagens, também são baseadas em XML.

O *parsing* do documento XML em tempo de execução requer um tempo de processamento adicional que pode resultar em um longo tempo de resposta do servidor [Chiu et al., 2002] [Davis and Parashar, 2002] [Elfwing et al., 2002] [Kohlhoff and Steele, 2003] [Govindaraju et al., 2004].

Atualmente três modelos de *parsing* de documentos XML estão sendo usados pelos *toolkits*:

- *Document Object Model* (DOM): constrói uma representação orientada a objetos do documento em memória. Esse modelo de processamento é o mais indicado quando o documento necessita ser alterado;

- *Simple API for XML* (SAX): é um modelo orientado a eventos que notifica à aplicação a ocorrência de elementos no documento através de chamadas *callback*, e dessa forma, não necessita construir uma representação em memória do documento;
- XML *Pull Parsing* (XPP): oferece vantagens como alto desempenho, um uso otimizado de memória comparado ao modelo DOM e facilidade de uso. Permite que o *parsing* XML seja realizado de forma incremental, onde a aplicação controla e solicita ao *parser* o próximo evento XML apenas quando a mesma pode processá-lo, ou seja, o *parsing* pode ser interrompido a qualquer momento e retomado quando a aplicação estiver pronta para consumir mais dados.

No momento da escolha do modelo de processamento é importante entender as limitações de cada um e avaliar o *trade-off* entre facilidade de uso e eficiência. Uma desvantagem comum a todos esses modelos é que requerem que os dados sejam varridos duas vezes: a primeira para o *parser* fazer a análise sintática do documento, e a segunda, para a aplicação interpretar o conteúdo.

Elfwing et al. (2002) compararam os modelos de *parsing* DOM e SAX usando duas implementações – Xerces e Crimson. Conforme o esperado, o modelo SAX foi mais eficiente que o modelo DOM. Porém, para o mesmo modelo de processamento, existiu uma grande diferença de desempenho entre as implementações, pois o Xerces SAX foi seis vezes mais lento que Crimson SAX. Além disso, o Crimson DOM foi mais eficiente que o Xerces SAX. A partir desses resultados, conclui-se que é importante analisar a implementação do modelo de *parsing* utilizada pelo *toolkit*.

3.5.3 Custos de Serialização e Deserialização

O tempo gasto nas fases de serialização e deserialização das mensagens SOAP tem sido identificado como o de maior impacto no tempo total de execução, com o custo da deserialização maior que o da serialização [Govindaraju et al., 2000] [Chiu et al., 2002] [Davis and Parashar, 2002] [Devaram and Andresen, 2003] [Engelen, 2003] [Kohlhoff and Steele, 2003] [Govindaraju et al., 2004] [Ng et al., 2003].

Além de consumir dez vezes mais memória que o processo de (de)serialização binária, a conversão entre objetos Java e mensagens XML é consideravelmente maior que os custos associados à comunicação e à buferização dos dados [Govindaraju et al., 2000], devido ao uso da tecnologia *reflection* para instanciar os objetos.

Devaram e Andresen (2003) identificaram que 50% do tempo de execução são gastos na serialização da mensagem SOAP em XML antes que seja enviada para o servidor e na criação da conexão HTTP. Porém, quando a conversão envolve *arrays* do tipo `double`, as rotinas de (de)serialização podem gastar 90% do tempo total de uma chamada SOAP [Chiu et al., 2002].

3.5.4 Cálculo do Tamanho da Mensagem SOAP

De acordo com a especificação do protocolo HTTP 1.0 [Berners et al., 1996], é necessário especificar o tamanho exato do corpo da mensagem HTTP, que é a mensagem SOAP codificada em XML, no atributo “Content-Length” do cabeçalho, quando o protocolo HTTP 1.0 for adotado para transportar as requisições SOAP. Porém, para calcular o tamanho de uma mensagem SOAP que é dinâmica, o cliente deve primeiro serializá-la antes mesmo de finalizar a construção do cabeçalho [Chiu et al., 2002] [Kohlhoff and Steele, 2003].

A solução adotada por um *Web Services toolkit* para realizar esse cálculo tem impacto direto no seu desempenho. Uma solução simples e mais comumente utilizada é usar *buffers* separados para o cabeçalho e para o corpo HTTP. Apenas quando a mensagem é completamente serializada e armazenada no *buffer*, é que seu tamanho será calculado e o valor é colocado no atributo “Content-Length” [Govindaraju et al., 2004]. Esta solução apesar de ser fácil de implementar, pode apresentar os seguintes problemas [Chiu et al., 2002] [Shirasuma et al., 2002] [Kohlhoff and Steele, 2003]:

1. Consumir muita memória se a mensagem serializada for grande, uma vez que a mesma será armazenada no *buffer*;
2. Invocar várias chamadas de sistema para o sistema operacional transmitir os dados armazenados nos *buffers*;

3. Usando um *buffer* que excede o tamanho da *cache* de sistema, poderá aumentar a falta de *cache*;
4. As fases de serialização, de transmissão na rede e de deserialização são realizadas em seqüência (ver Figura 3.2). Tais fases não podem ser sobrepostas porque a mensagem não pode ser enviada até que o processo de serialização XML tenha terminado, para que se possa calcular o seu tamanho. E dependendo do modelo de *parsing* utilizado no receptor, a mensagem só poderá ser deserializada depois que todos os dados estejam armazenados no *buffer*.

3.5.5 Gargalos de Comunicação

Alguns estudos focaram na análise dos custos associados diretamente ao protocolo de transporte, principalmente o protocolo HTTP, e à implementação da camada de comunicação do *toolkit*.

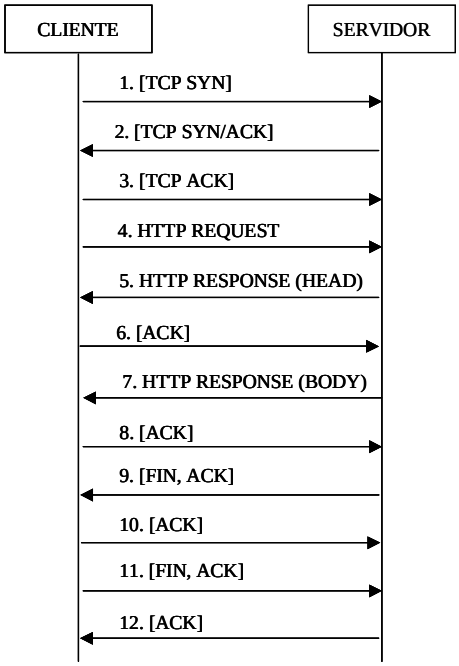

Figura 3.3 - Tráfego de pacotes para uma chamada SOAP/HTTP

O processo de identificação dos gargalos de comunicação baseou-se na análise detalhada do tráfego de pacotes da comunicação entre o cliente e o servidor. Os

seguintes gargalos foram identificados [Elfwing et al., 2002] [Davis and Parashar, 2002] [Gray, 2004]:

1. Quebra da requisição ou resposta do servidor em duas partes: a primeira contendo o cabeçalho HTTP e a segunda, o corpo da resposta que representa o envelope SOAP. A Figura 3.3 foi extraída de [Elfwing et al., 2002] e ilustra esse comportamento, onde a resposta do servidor foi quebrada. Outro ponto é que para cada uma dessas partes, um pacote de confirmação é transmitido na rede;
2. Número total de pacotes de dados necessários para transmitir a mensagem SOAP: além dos custos associados à transferência dos documentos, quanto mais dados são transmitidos na rede, mais controle dos pacotes é necessário. Os *Web Services* t*oolkits* geram, em média, de 3 a 5 vezes mais pacotes de dados.
3. Atraso para enviar os pacotes de confirmação (pacotes de número 6 e 8 na Figura 3.3): esse atraso acontece em dois momentos: o primeiro após o cliente receber o pacote de número 5 com cabeçalho, e o segundo após o cliente receber o pacote de número 7 com o corpo da mensagem. O tempo que o cliente espera para enviar o pacote de confirmação variou entre 100ms a 200ms e é causado pelo algoritmo *TCP delayed ACK*, que é configurado no sistema operacional;
4. Tempo esperando o pacote de confirmação: o servidor não envia o pacote de número 7 antes que o pacote de confirmação referente ao pacote de número 5 tenha chegado. Esse comportamento é causando pelo algoritmo *Nagle* habilitado no lado do servidor. O algoritmo *Nagle* é controlado pela propriedade `TCP_NODELAY` do *socket*. Para aplicações Java, a seguinte linha de código desabilita o algoritmo *Nagle*, onde `socket` é uma instância do tipo `java.net.Socket`:

```
socket.setTcpNoDelay(true);
```

5. Atraso associado ao fechamento da conexão. O fechamento da conexão inicia quando o servidor envia o pacote TCP/FIN para o cliente, que deve ser confirmado pelo cliente. Após a confirmação, o cliente também envia o pacote TCP/FIN para o servidor, que também deve ser confirmado. O gargalo não consistiu na quantidade de pacotes trocados, mas no atraso do servidor em enviar o primeiro pacote, pois o cliente fica lendo do *socket* até encontrar fim de

arquivo. Este gargalo foi responsável pela maior parte do tempo de execução, variando entre 7.9 a 439ms.

O gargalo introduzido pela combinação dos algoritmos *Nagle* e *TCP delayed ACK* foi, em média, de 350ms no tempo total para cada requisição, porém ambos foram projetados para reduzir o número de pequenos pacotes trafegando na rede.

3.5.6 Custo do Estabelecimento da Conexão

A especificação do protocolo HTTP 1.0 obriga que o cliente estabeleça uma nova conexão antes de cada requisição e que o servidor feche-a após finalizar o envio da resposta ao cliente [Berners et al., 1996].

Estabelecendo uma nova conexão para cada transação pode ter um impacto negativo no desempenho, pois o protocolo HTTP usa o protocolo TCP que estabelece conexões via *Three-Way Handshake* (ver Figura 3.3), onde o cliente envia a requisição para estabelecer uma conexão, o servidor confirma a solicitação e, por fim, o cliente também confirma. O estabelecimento de uma nova conexão para cada requisição também aumenta o número de pequenos pacotes trocados entre o cliente e o servidor [Davis and Parashar, 2002] [Elfwing et al., 2002] [Kohlhoff and Steele, 2003].

Em uma LAN onde o atraso é baixo e a perda de pacotes é rara, o custo do estabelecimento da conexão TCP é 1% menor que o custo de *parsing*. Porém, quando o atraso for alto, esse custo não é insignificante [Elfwing et al., 2002].

3.6 Técnicas de Otimização

Embora uma simples aplicação *Web Services* possa apresentar problemas de desempenho, algumas técnicas de otimização podem ser aplicadas a fim de amenizar sua ineficiência.

Além de identificarem os gargalos que afetam o desempenho de *Web Services*, alguns trabalhos também desenvolveram e avaliaram técnicas de otimização de

desempenho que serão detalhadas nas próximas subseções. De forma geral, tais técnicas visam reduzir o uso de memória, o tempo de processamento e o custo da comunicação.

3.6.1 Compressão dos Dados

Quando a largura de banda é baixa, o tamanho das mensagens é um gargalo limitante do desempenho [Kohlhoff and Steele, 2003]. Uma maneira de reduzir o número de *bytes* transferidos na rede é comprimir o tamanho das mensagens SOAP. Porém, como a compressão em tempo real requer um tempo de CPU extra, pode ocorrer um aumento no tempo de resposta e uma redução da vazão. Experimentos mostraram que a compressão em tempo real é cara e excede os custos da serialização e transmissão dos dados [Cai et al., 2002] [Engelen, 2003] [Kohlhoff and Steele, 2003].

Uma outra tentativa para reduzir o tamanho das mensagens XML foi usar *tags* XML compactas [Kohlhoff and Steele, 2003]. Essa otimização proporcionou uma melhora insignificante no tempo de serialização das mensagens, indicando que o maior custo da codificação e decodificação XML está na complexidade estrutural e sintaxe dos elementos e não apenas na natureza XML dos dados.

3.6.2 Parser Específico de Esquema XML

Parsers desenvolvidos para processar de um esquema XML específico apresentam um melhor desempenho em relação aos *parsers* de propósito geral – DOM, SAX e *Pull Parsing*, principalmente quando grandes estruturas de dados estão envolvidas [Chiu et al., 2002], porque varrem os dados XML apenas uma vez.

O *toolkit* gSOAP suporta essa otimização disponibilizando um compilador que gera o código para realizar o *parsing* e processamento de estruturas de dados a partir de um esquema XML.

3.6.3 Caching das Requisições SOAP

Após identificar que 50% do tempo de processamento do cliente são gastos na codificação da mensagem SOAP em XML e na criação da conexão HTTP, Devaram e

Andresen (2003) projetaram um eficiente mecanismo de *caching* para as requisições SOAP do cliente, com o objetivo de otimizar o seu desempenho diminuindo a necessidade de gerar uma nova mensagem XML para todas as requisições. Dois tipos de mecanismos de *caching* foram desenvolvidos – *caching* completa e *caching* parcial.

O mecanismo de *caching* completa aplica-se nos casos onde repetidas requisições SOAP são enviadas ao servidor. Antes de realizar qualquer requisição, o cliente deve verificar na *cache* se já existe uma mensagem associada à requisição desejada. Caso seja a primeira vez que uma determinada requisição é realizada, um arquivo contendo toda a mensagem SOAP é gerado e armazenado na *cache* e indexado por uma chave. Para que, nas requisições subseqüentes, essa mensagem é recuperada da *cache* e não seja novamente serializada, reduzindo, então, o tempo de execução.

O mecanismo de *caching* parcial aplica-se nos casos onde o cliente faz a mesma requisição, exceto pelos valores dos seus parâmetros. Nessa estratégia, quando a mensagem é encontrada na *cache*, o cliente deve preencher os valores das *tags* na mensagem com os novos valores dos parâmetros.

O pré-requisito para aplicar a estratégia de *caching c*ompleta é que o cliente tenha um número fixo de diferentes tipos de requisições, caso contrário, o tempo gasto com operações de entrada e saída (I/O) será alto devido ao aumento do tamanho da *cache*. E o pré-requisito para a aplicação do mecanismo de *caching* parcial, é que o número das *tags* cujos valores serão atualizados seja pequeno.

Nos experimentos realizados, o desempenho de uma aplicação Java utilizando a estratégia de *caching* foi melhor que o desempenho da mesma aplicação implementada em Java RMI. Usando mensagens grandes (20 KB) e complexas a estratégia de *caching* parcial foi mais eficiente que a estratégia de *caching* completa, devido ao crescimento do tamanho da *cache* [Devaram and Andresen, 2003].

3.6.4 Otimizando o Cálculo do Tamanho da Mensagem SOAP

Nessa seção serão apresentadas algumas técnicas que visam otimizar o gargalo do cálculo do tamanho da mensagem SOAP apresentado na seção 3.5.4.

Técnica 1: Preenchendo o Atributo "Content-Length" com Espaços

Essa técnica é chamada de *Back-Patching* e consiste em inserir espaços no atributo "Content-Length" durante a geração inicial do cabeçalho, que serão, posteriormente, substituídos pelo tamanho real quando a mensagem SOAP for processada [Chiu et al., 2002].

Utilizando essa técnica, não é necessário manter dois *buffers* separados, um para o cabeçalho e outro para o corpo da mensagem SOAP, e apenas uma chamada de sistema é realizada para enviar a mensagem.

Técnica 2: Envio Vetorizado das Mensagens

É uma solução alternativa à técnica anterior e permite o envio de vários *buffers* de memória com uma única chamada de sistema [Chiu et al., 2002]. Então, os *buffers* do cabeçalho e do corpo da mensagem podem ser enviados ao mesmo tempo. Essa técnica é chamada de *Vectored Send Call* e é utilizada pelo *toolkit* bSOAP [Govindaraju et al., 2004].

Técnica 3: Técnica de Serialização em Dois Estágios

Diferentemente das técnicas que armazenam a mensagem em *buffers* a fim de determinar o seu tamanho, a técnica de serialização em dois estágios (*two-stage serialization*) emprega um algoritmo rápido de serialização que é divido em duas fases: a primeira fase consiste em varrer os dados, contar o tamanho da mensagem sem armazenar em um *buffer* e verificar os ponteiros da estrutura de dados, caso seja cíclica ou um grafo. A segunda fase consiste em construir o cabeçalho e serializar a mensagem SOAP diretamente sobre TCP/IP. O *toolkit* gSOAP aplica essa técnica a fim de minimizar o uso de memória e preservar as estruturas de dados cíclicas e multi-referenciadas [Engelen and Gallivan, 2002].

Técnica 4: Transmissão dos Dados em Blocos (Chunked Transfer Coding)

HTTP 1.1 [Fielding et al., 1999] suporta uma forma simples de *streaming* chamada *Chunked Transfer Coding*, que permite a quebra da mensagem HTTP em vários blocos (*chunks)*, onde cada bloco é precedido pelo seu próprio tamanho. Quando essa técnica

de *streaming* é usada, não é necessário enviar o tamanho da mensagem no cabeçalho, mas indicar que a mensagem será enviada em blocos.

A sua vantagem é que pode ser aplicada tanto pelo cliente quanto pelo servidor, onde é mais freqüentemente adotada. Além disso, evita o armazenamento em memória dos dados antes da transmissão e permite a sobreposição entre as fases de serialização, de transmissão na rede e a de deserialização [Chiu et al., 2002] [Davis and Parashar, 2002] [Engelen, 2003] [Govindaraju et al., 2004]. O emissor da mensagem, após serializar um bloco, pode transmiti-lo ao mesmo tempo em que serializa o próximo bloco, enquanto que o receptor da mensagem pode iniciar as fases de *parsing* e de deserialização, assim que as mensagens cheguem no *buffer* (Figura 3.4).

O problema dessa técnica é determinar o tamanho ideal para os blocos, pois se for muito pequeno, muitas chamadas de sistemas são invocadas, e se for muito grande, aumenta a falta de *cache* de sistema.

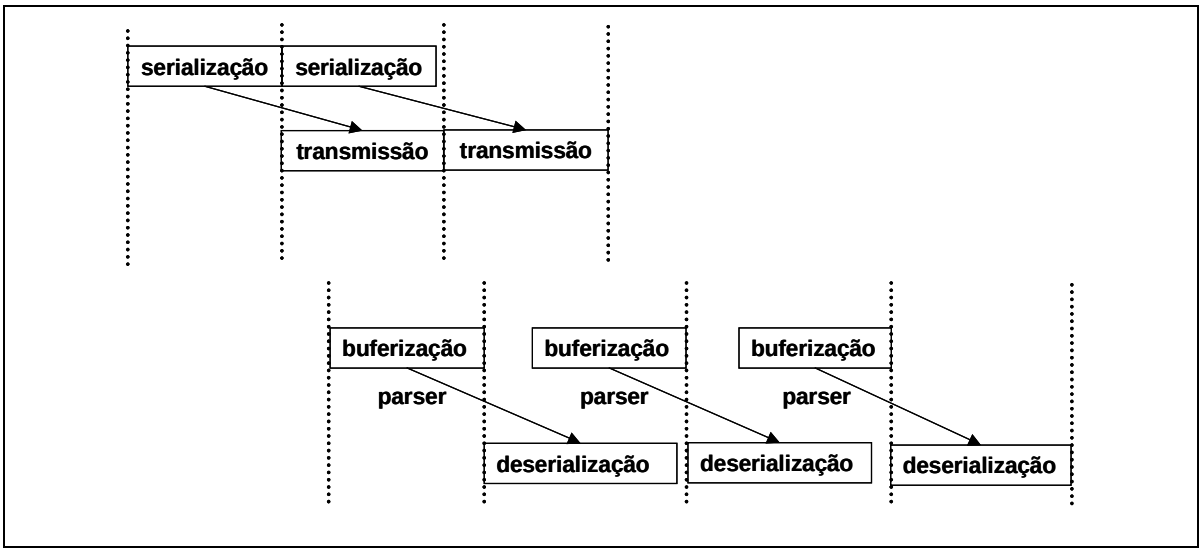

Figura 3.4 – Enviando uma mensagem SOAP com otimizações

Técnica 5: Eliminação do Atributo Content-Length

Essa técnica propõe que os servidores HTTP contenham pares de *tags* XML para determinar o final da mensagem SOAP. Dessa forma, eliminando o atributo “Content-Length” do cabeçalho e, conseqüentemente, eliminando o cálculo do tamanho da mensagem.

Com a omissão desse atributo, é possível sobrepor as fases de serialização, de transferência na rede e de deserialização (ver Figura 3.4), reduzindo o tempo total de processamento e solucionando o gargalo introduzido pelo cálculo do tamanho da mensagem (ver Seção 3.5.4).

A partir dos experimentos realizados numa LAN, Shirasuma et al. (2002) verificaram que essa técnica otimizou em aproximadamente 55% o tempo de resposta, porém sua aplicação viola a RFC 1945 [Berners et al., 1996], uma vez que a mesma especifica que é necessário enviar o tamanho da mensagem.

3.6.5 Otimizações na Comunicação

A partir dos gargalos de comunicação apresentados na Seção 3.5.5, Elfwing et al. (2002) propuseram algumas otimizações a fim de diminuir o número de pacotes e, principalmente, reduzir os atrasos inerentes aos pacotes de confirmação e de fechamento da conexão.

Técnica 1: Fechamento da Conexão Iniciada pelo Cliente

A Figura 3.5 ilustra o tráfego de pacotes onde o cliente é responsável pelo fechamento da conexão, iniciado pelo pacote de número sete. Para isso, o atributo “Content-Length” foi enviado no cabeçalho da resposta do servidor, permitindo que o cliente conte os *bytes* recebidos e feche a conexão, sem precisar esperar que o servidor envie o pacote TCP/FIN. Aplicando essa otimização nos seus experimentos, Elfwing et al. (2002) verificaram que o tempo de execução reduziu de 680ms para 200ms.

Embora essa solução tente eliminar o atraso inerente ao protocolo de fechamento da conexão, ela introduz o problema do cálculo do tamanho da mensagem no envio da resposta (ver Seção 3.5.4). Outra desvantagem é que a especificação do protocolo HTTP 1.0 não garante que esse atributo estará presente no cabeçalho da resposta do servidor. Antes de aplicar essa técnica, é necessário avaliar qual gargalo tem o maior impacto – o atraso para fechar a conexão ou o cálculo do tamanho da mensagem.

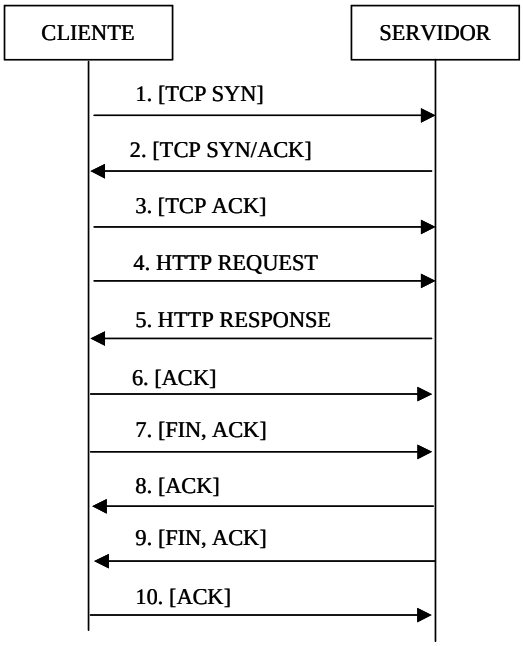

Figura 3.5 - Tráfego de pacotes para uma chamada SOAP com otimizações

Técnica 2: Enviando o Cabeçalho e a Mensagem SOAP em um Pacote HTTP

Além de diminuir o número de pacotes transmitidos na rede, essa otimização reduziu, em média, 150ms do tempo de resposta, originado pelo algoritmo *TCP delayed ACK*.

Aplicando essa otimização juntamente com a anterior, o tempo de execução reduziu para, aproximadamente, 42ms. A Figura 3.5 ilustra os pacotes trocados quando a resposta do servidor é enviada em um único pacote.

Técnica 3: Desabilitar o algoritmo Nagle e Configurar o Tempo do Algoritmo TCP delayed ACK para Zero

Como os algoritmos *Nagle* e *TCP delayed ACK* causam atrasos desnecessários no cenário da comunicação *Web Services*, desabilitando esses algoritmos ocasionaria uma diminuição, aproximadamente, de 350ms no tempo de execução. No entanto, a carga na rede poderá aumentar, uma vez que os mesmos foram projetados para reduzir o número de pacotes na rede.

3.6.6 Uso de Conexões Persistentes

O protocolo HTTP 1.1 suporta conexões persistentes (*HTTP keep-alive*) por *default*, a menos que o atributo "*Connection: Close*" seja especificado no cabeçalho.

Essa característica permite o reuso da mesma conexão TCP/IP para enviar múltiplas requisições, dessa forma, eliminando os gargalos de estabelecer (*Three-Way Handshake*) e fechar uma conexão para cada chamada e o atraso para iniciar o processo de fechamento da conexão [Elfwing et al., 2002] [Engelen, 2003] [Kohlhoff and Steele, 2003] [Govindaraju et al., 2004].

Uma vez que o custo de estabelecer uma conexão aumenta com o atraso da rede, o benefício dessa otimização será mais percebível em redes com alto atraso [Chiu et al., 2002].

3.6.7 Codificação Binária dos Dados XML

A codificação dos dados em Base64 é suportada pela tecnologia XML *Schema* e é uma representação atrativa para tornar mais eficiente a troca de *arrays* de números ponto flutuante, pois reduz a perda de precisão dos números, diminui o número de *bytes* e o gargalo da serialização [Engelen, 2003]. Os experimentos realizados para quantificar sua eficiência demonstram que essa otimização reduz o tempo total de execução em 75% [Shirasuma et al., 2002].

A desvantagem é que não suporta o envio de muitos dados binários de forma eficiente e, por ser uma técnica de codificação binária, perde-se a legibilidade dos dados. Nesses casos, técnicas como *SOAP with Attachments* e *WS-Attachment*, descritas na próxima subseção, deveriam ser utilizadas.

3.6.8 Enviando Mensagens SOAP com Anexos

Tanto SwA (*SOAP with Attachment*) quanto *WS-Attachment* são técnicas utilizadas para viabilizar a transferência de grandes dados binários – imagens e sons – em uma mensagem SOAP. A diferença está na estruturação e processamento da mensagem. A especificação de SwA encapsula uma mensagem SOAP e os demais anexos em uma

estrutura MIME (*Multipurpose Internet Mail Extension*) e a especificação *WS-Attachment*, utiliza uma estrutura DIME (*Direct Internet Message Encapsulation*).

Para determinar o número de anexos e os seus limites numa estrutura MIME é preciso varrer toda a mensagem. Com a estrutura DIME, o *parser* pode simplesmente usar os dados do cabeçalho dos registros para rapidamente indexá-los e calcular o número de anexos na mensagem [Govindaraju et al., 2004]. Outra diferença é que os anexos DIME podem ser transmitidos na forma de *streaming*.

Ying et al. (2004) compararam o padrão SOAP com as técnicas SwA e *WS-Attachment*. Em geral, as técnicas SwA e *WS-Attachment* têm melhor desempenho que o padrão SOAP, principalmente quando o tamanho dos dados aumenta. O benefício é que essas técnicas reduzem o tamanho da mensagem e o custo de serialização e deserialização, conseqüentemente melhorando o tempo de resposta. Porém, em termos de desempenho, a técnica *WS-Attachment* foi mais eficiente que a SwA.

3.6.9 Otimizando os Custos de Serialização

A técnica chamada *differential serialization* foi projetada e desenvolvida para reduzir os custos associados ao processo de serialização de uma mensagem SOAP e otimizar a comunicação do lado do emissor da mensagem [Abu-Ghazaler et al., 2004] [Abu-Ghazaler et al., 2004a] [Abu-Ghazaler et al., 2004b].

A técnica consiste em salvar uma cópia da mensagem serializada após o seu primeiro envio. Durante as requisições subseqüentes, para a mesma aplicação *Web Services*, apenas os elementos que mudaram serão serializados novamente.

A eficiência dessa otimização depende do tamanho da mensagem, do conteúdo e da similaridade entre as mensagens executadas. A partir dos estudos realizados em [Abu-Ghazaler et al., 2004] [Abu-Ghazaler et al., 2004a] [Abu-Ghazaler et al., 2004b], os seguintes resultados foram encontrados:

1. Quando a mensagem exata necessita ser enviada novamente, o processo de serialização é eliminado;
2. Quando todos os elementos necessitam ser serializados, onde apenas as *tags* e o envelope SOAP são reusados, o ganho de desempenho é de 17%;

3. Dependendo do percentual de elementos que necessitam ser serializados, o ganho de desempenho pode variar entre 22% e 68%.

O *toolkit* bSOAP aplica essa técnica, e atualmente, a técnica *differential deserialization* está sendo estudada para otimizar o processo de deserialização das mensagens SOAP.

3.7 Considerações Finais

Nesse capítulo foram apresentados vários trabalhos que investigaram a ineficiência de *Web Services*, detalhando seus gargalos e listando possíveis otimizações para tornar as aplicações *Web Services* mais eficientes. Além disso, foram apresentados alguns resultados da comparação de desempenho de diferentes *Web Services toolkits* entre si e também com outros *middleware*.

De forma geral, o desempenho de uma aplicação *Web Services* dependem do projeto e implementação do *toolkit* utilizado para implementá-la e dos gargalos introduzidos pelos protocolos SOAP e de transporte, onde o protocolo HTTP é o mais comumente adotado. Os gargalos detalhados nesse capítulo foram o tamanho e a complexidade das mensagens, a escolha do *parser*, os custos de serialização e deserialização, o tempo gasto para calcular o tamanho da mensagem, estilo de codificação, o custo de estabelecimento das conexões e os gargalos de comunicação como o atraso na troca de pacotes e o número de pacotes.

Desses gargalos, os custos associados ao processo de serialização e deserialização foram os de maior impacto no desempenho, logo as soluções que visam otimizar essas rotinas são as que apresentaram melhores resultados como as técnicas de *caching*, *differential serialization* e representação binária dos dados XML usando a codificação Base64, *SOAP with Attachment* ou *WS-Attachment*. Entretanto, a codificação binária dos dados ao mesmo tempo em que reduz o tamanho das mensagens, também reduz as características de universalidade e interoperabilidade, uma vez que a troca de mensagens XML é o coração de *Web Services*.

Atualmente, alguns *toolkits* como o gSOAP e bSOAP estão sendo projetados com foco em eficiência, aplicando técnicas de otimização para solucionar os gargalos de desempenho. O *toolkit* Axis da Apache, que é de código aberto e gratuito, está sendo estudado por vários autores e suas versões mais recentes são mais eficientes que as versões anteriores, embora o mesmo ainda apresente o pior desempenho.

Mesmo aplicando as otimizações propostas, a questão do desempenho de *Web Services* ainda está em aberto e muito se tem que estudar. Então, antes de ser utilizado para expor aplicações que demandam por alto desempenho, deve-se testar sua eficiência e comportamento no ambiente que simule as mesmas características do cenário real. Idealmente, o processo de escolha do *toolkit* deve incluir a avaliação de vários *Web Services toolkits*, a fim de selecionar o que mais atende aos requisitos da aplicação.

Diferentemente dos trabalhos apresentados nesse capítulo, essa dissertação foca na avaliação de desempenho de *Web Services toolkits* guiada por um conjunto de diretrizes desenvolvidas com o objetivo de identificar seus gargalos de desempenho e entender seu funcionamento. Dessa forma, contribui-se com a uniformidade do processo de avaliação de diferentes *Web Services toolkits*.

No próximo capítulo serão apresentadas as diretrizes de avaliação de desempenho de *Web Services* que foram desenvolvidas baseando-se nos gargalos, otimizações, métricas e resultados detalhados ao longo desse capítulo.

4 Diretrizes para Avaliação de Desempenho de *Web Services*

4.1 Introdução

O cenário atual da área de desenvolvimento de *software* é caracterizado por aplicações complexas, distribuídas e que demandam por alto desempenho e por várias tecnologias que os desenvolvedores podem escolher para implementar essas aplicações. No mundo de *Web Services* não é diferente, pois existem vários *Web Services toolkits* implementados em diferentes linguagens (ver Seção 2.4). Dessa forma, é necessário avaliar o desempenho desses *toolkits* antes de desenvolver as aplicações, a fim de identificar o mais apropriado para atender aos seus requisitos não funcionais como eficiência, latência baixa, alta vazão e uso eficiente de memória.

A avaliação de desempenho dos *Web Services toolkits* realizada nos trabalhos apresentados anteriormente foi feita, na maioria dos casos, de forma simples, usando apenas tipos de dados escalares e coletando algumas métricas de desempenho (ver Capítulo 3). Uma conseqüência é que os resultados de alguns desses estudos não foram totalmente abrangentes, pois o desempenho não foi avaliado usando tipos de dados complexos. Um exemplo dessa situação foi a contradição entre os resultados obtidos por

Juric et al. (2004) e Gray (2005). O primeiro concluiu que *Web Services* são mais eficientes que as tecnologias de tunelamento de Java RMI. Porém, explorando vários cenários de teste, Gray (2005) verificou que o desempenho de *Web Services* é melhor apenas quando tipos de dados simples são utilizados. Quando estruturas grandes e complexas são utilizadas, Java RMI sobre o protocolo HTTP é mais eficiente.

Dessa forma, existe a necessidade de um guia geral para realizar os testes de desempenho de *Web Services*, a fim de uniformizar o processo de avaliação. Nas seções seguintes serão apresentadas as diretrizes para avaliação de desempenho de *Web Services toolkits* que podem ser utilizadas, por desenvolvedores ou arquitetos com ou sem experiência na tecnologia *Web Services*, para avaliar a eficiência de qualquer *toolkit* [Machado and Ferraz, 2005].

4.2 Objetivo das Diretrizes

As diretrizes apresentadas a seguir foram desenvolvidas baseando-se nos resultados, métricas, gargalos e otimizações explicados no capítulo anterior, ou seja, resultam da análise detalhada e organização dos trabalhos relacionados ao estado da arte do desempenho de *Web Services*.

A proposta das diretrizes é descrever uma política de avaliação de desempenho, contribuindo, assim, com o avanço da área. A Figura 4.1 sintetiza tais gargalos que afetam o desempenho de *Web Services* e que foram utilizados como base para propor as diretrizes. De forma geral, as mesmas têm como objetivo permitir que um desenvolvedor ou arquiteto [Machado and Ferraz, 2005]:

1) Entenda o comportamento do *toolkit* sendo analisado;
2) Identifique os gargalos de desempenho;
3) Quantifique o tempo para transmitir as mensagens SOAP de diferentes tamanhos, complexidade e tipos de dados;
4) Monitore o tráfego de pacotes entre o cliente e o servidor;

5) Projete uma interface WSDL que atenda aos requisitos de eficiência e interoperabilidade;
6) Verifique a necessidade de aplicar otimizações na comunicação ou na implementação do *toolkit*.

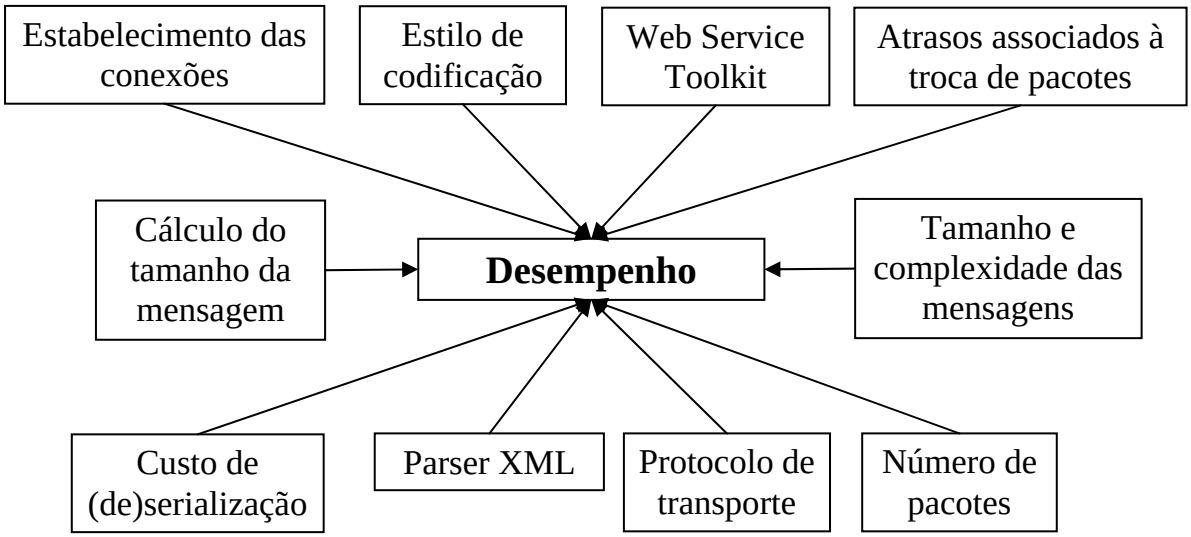

Figura 4.1 - Fatores que influenciam o desempenho de *Web Services*

Dessa forma, as diretrizes podem ser utilizadas na escolha do *toolkit* "ideal" para desenvolver um serviço que demanda por desempenho. Como o objetivo é melhorar o desempenho sem afetar a interoperabilidade, as diretrizes não exploram nenhuma técnica de codificação binária dos dados e nem a troca do protocolo HTTP por um outro, como SMTP.

4.3 Guia para Avaliação de Desempenho

As diretrizes listadas nessa seção foram projetadas para serem simples, práticas, eficientes e fáceis de usar, simplificando a escolha do melhor *toolkit* para desenvolver e expor um determinado serviço, além de propor uma padronização do processo de avaliação de desempenho dos diferentes *Web Services toolkits*.

As boas práticas para desenvolver uma aplicação distribuída são praticamente as mesmas para desenvolver aplicações *Web Services*:

1. Deve-se projetar uma interface de forma a otimizar o tráfego na rede, minimizando o número de chamadas remotas, para melhorar o desempenho;
2. A interface do serviço constitui um contrato entre o serviço e o cliente.

A granularidade da interface WSDL é uma importante decisão de projeto, pois a mesma se refere tanto ao escopo do domínio do serviço quanto ao escopo do domínio de cada método da interface. Em geral, o nível de granularidade apropriado para um serviço e seus métodos é *"coarse-grained"*, pois significa que o serviço disponibiliza várias funcionalidades que retornam muitos dados.

Determinar a granularidade de uma interface é uma decisão difícil, porque os projetistas não podem antecipar completamente a maneira como os serviços serão usados durante o seu projeto. Além disso, dentro da granularidade *"coarse-grained"* ainda existe uma escala com vários degraus de granularidade. De maneira geral, os serviços disponibilizados deveriam ser fáceis de usar e, ao mesmo tempo, deveriam satisfazer as necessidades dos seus consumidores.

Uma vez que os clientes invocam o serviço remotamente, é importante que as interfaces estejam bem projetadas, caso contrário, o consumidor do serviço poderá receber mais dados do que precisa ou poderá fazer muitas requisições para obter todas as informações de que necessita.

Os projetistas, tendo consciência desses impactos, deveriam desprender um tempo maior durante a definição do serviço, a fim de projetar uma interface que minimize o impacto no desempenho. Dessa forma, é importante estudar as possíveis soluções de projeto e implementar a mais apropriada.

Além do desempenho, o projeto da interface também afeta outros requisitos não funcionais da aplicação como modificabilidade, que representa o grau em que o sistema incorpora mudanças de forma "fácil", e a reusabilidade, habilidade de uma aplicação ser usada em diferentes contextos sem sofrer modificações.

De forma geral, como regra de desempenho, na dúvida sobre a granularidade de um serviço, deve-se publicar operações que façam muito trabalho, aceitem vários parâmetros e retorne uma porção de informações. O objetivo é minimizar o número de

requisições remotas. Além disso, os futuros clientes provavelmente poderão necessitar das informações extras.

As diretrizes apresentadas a seguir não exploram diretamente o impacto no desempenho causado pela granularidade de um serviço. O foco foi investigar o desempenho dos *Web Services toolkits* usando diferentes tipos de dados com tamanhos e complexidades variados, a fim de publicar regras para avaliar o desempenho desses *toolkits*, e não os gargalos introduzidos pela granularidade da interface da aplicação.

Diretriz 1: Adote o estilo Document/Literal Wrapped

Durante o projeto da interface WSDL, a configuração dos parâmetros *style* e *use* pode afetar não só o grau de interoperabilidade do serviço, como também o seu desempenho (ver Seção 2.3.1). Os possíveis estilos para codificar a interface WSDL são:

1. *RPC/Encoded*
2. *Document/Encoded*
3. *RPC/*Literal
4. *Document/Literal*
5. *Document/Literal Wrapped*

Uma vez que a escolha do estilo de codificação afeta o desempenho da aplicação [Cohen, 2003] [Ng et al., 2003], qual dessas cinco combinações deve ser usada para estruturar as mensagens SOAP?

O estilo *RPC/Encoded* tem sido o mais comumente adotado devido sua semelhança com os modelos de chamadas remotas tradicionais [Devaram and Andresen, 2003]. Esse estilo foi projetado para permitir que as mensagens SOAP simulem chamadas RPC. As principais vantagens desse estilo são a clareza do arquivo WSDL e o envio do nome da operação na mensagem SOAP, dessa forma o receptor da mensagem pode facilmente despachar a mensagem para a implementação do método solicitado. A desvantagem é que o conteúdo da mensagem SOAP não pode ser facilmente validado porque existem dados que não estão definidos no esquema XML dos tipos.

O estilo *Document*/*Encoded* tem sido visto como uma combinação inválida, não sendo suportado pelos *Web Services toolkits* atuais. Essa combinação deverá desaparecer nas futuras versões da especificação WSDL.

Como essas duas combinações são *Encoded*, ou seja, usam as regras de codificação detalhadas na especificação do protocolo SOAP, as mesmas não fazem parte do conjunto de recomendações propostas pelo WS-I (*Web Services Interoperability Organization*) para maximizar a interoperabilidade das aplicações *Web Services*. Além disso, o tipo de codificação *Encoded* é um ponto de degradação do desempenho.

O estilo *RPC/Literal* apesar de ser um estilo recomendado pelo WS-I, não é suportado por algumas plataformas *Web Services* como, por exemplo, o *toolkit* .NET. Então, por questões de interoperabilidade, esse estilo também não deveria ser adotado.

O estilo *Document/Literal* é a maneira mais recomendada para representar uma requisição *Web Services*. As mensagens SOAP codificadas nesse estilo podem ser facilmente analisadas por qualquer tecnologia de validação XML, uma vez que todo o conteúdo dentro da *tag* `<soap:body>` é definido por esquema XML.

Nos estudos realizados por Ng et al. (2003) e Cohen (2003), o estilo *Document/Literal* apresentou um melhor desempenho, porque produz mensagens menos complexas e requer, aproximadamente, metade do número de *bytes* para representar as mensagens, minimizando os custos de transmissão dos dados na rede e o tempo de resposta. Além disso, as regras de serialização e deserialização são mais eficientes que as do estilo *RPC*/*Encoded*.

Apesar do estilo *Document/Literal* ser o mais indicado para construir as mensagens SOAP, o mesmo tem uma desvantagem, pois o nome da operação sendo invocada não está presente na mensagem SOAP, dificultando o despacho da operação.

Para solucionar esse problema foi projetado o estilo *Document/Literal Wrapped*, também chamado de *Wrapped/Literal*, que além de possuir as mesmas vantagens do estilo *Document/Literal*, envia o nome da operação sendo invocada na mensagem SOAP. Do ponto de vista técnico, esse estilo é um caso especial do estilo *Document/Literal*.

Dessa forma, essa diretriz adota o uso do estilo *Document/Literal Wrapped*, porque além do desempenho, esse estilo apresenta bons resultados de interoperabilidade. As recomendações de interoperabilidade eliminam o uso do estilo *RPC/Encoded*, apesar de muitos *Java Web Services toolkits* adotarem esse estilo como o padrão.

Diretriz 2: Utilize mensagens de tamanhos e complexidades diferentes

A maioria dos estudos reportados anteriormente tem analisado *benchmarks* que envolvem pouca transferência de dados, usando operações simples que não tinham nem parâmetros nem valores de retorno, ou apenas tipos de dados simples como inteiros, ponto flutuante ou *string*.

Como os tipos de dados básicos consomem menos tempo de processamento e freqüentemente são empacotados em um único pacote, tais estudos não apresentam uma análise completa do desempenho cujos resultados possam ser totalmente usados na seleção do *toolkit* para expor as aplicações reais.

Um processo de avaliação deveria também usar tipos de dados mais representativos de aplicações *Web* ou de qualquer outro contexto que envolva não apenas grandes quantidades de dados, como também uma sintaxe complexa. Essa diretriz auxilia a responder questões como:

- Qual é a influência de diferentes tipos de dados usados como parâmetro e valores de retorno no desempenho?
- Qual é o tempo para serializar e deserializar diferentes tipos de dados?
- Qual é o impacto do tamanho dos dados no desempenho, devido a sua influência no empacotamento dos dados para a transmissão na rede?

A ineficiência de *Web Services* não é unicamente afetada pelo tamanho da mensagem, mas, principalmente, pelo tempo gasto na conversão das estruturas de dados em XML e vice-versa. Quanto mais complexa for uma mensagem, maior será o seu tempo de conversão.

Dessa forma, é importante avaliar o desempenho de um *toolkit* usando mensagens de vários tamanhos e complexidades, principalmente usando estruturas *arrays*, pois os *toolkits* podem representar essas estruturas de formas e tamanhos diferentes.

Diretriz 3: Analise as mensagens SOAP transportadas na rede

Monitorando as mensagens SOAP transportadas sobre o protocolo HTTP, é possível determinar o tamanho da requisição e da resposta para cada operação invocada, identificar o estilo de codificação das mensagens (RPC ou *Document*), a versão do protocolo HTTP e os atributos do seu cabeçalho. A versão do protocolo é informada na primeira linha do cabeçalho (Figura 4.2) e a requisição do cliente pode usar uma versão do protocolo diferente da versão usada nas respostas do servidor.

As Figuras 4.2, 4.3 e 4.4 representam exemplos de mensagens capturadas através de uma ferramenta gráfica de monitoramento das mensagens SOAP chamada TCP Monitor [Apache Axis, 2004]. Cada uma das figuras representa um exemplo de possíveis configurações do cabeçalho HTTP, usando diferentes atributos para transportar a mesma mensagem.

```
POST /service HTTP/1.1
Content-Type: text/xml; charset=utf-8
Content-Length: 454
SOAPAction: ""
User-Agent: Java/1.4.2_08
Host: 127.0.0.1
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive

<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
              xmlns:xsd="http://www.w3.org/2001/XMLSchema
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <env:Body>
   ...
  </env:Body>

</env:Envelope>
```

Figura 4.2 - Exemplo de uma requisição SOAP enviada via HTTP

A partir da análise do cabeçalho da mensagem ilustrada na Figura 4.2, observa-se o seguinte comportamento do *toolkit*:

1) A requisição é enviada ao servidor usando a versão 1.1 do protocolo HTTP;

2) O tamanho da mensagem é calculado e enviado através do atributo “Content-Length”. O processo de cálculo do tamanho da mensagem pode ser um gargalo de desempenho do *toolkit* (ver Seção 3.5.4);

3) A requisição solicita o estabelecimento de conexões persistentes devido a presença do atributo “Connection: keep-alive”.

```
HTTP/1.1 200 OK
SOAPAction: ""
Content-Type: text/xml;charset=utf-8
Transfer-Encoding: chunked
Date: Sun, 24 Jul 2005 13:52:55 GMT
Server: Sun-Java-System/Web-Services-Pack-1.4

<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
              xmlns:xsd="http://www.w3.org/2001/XMLSchema
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <env:Body>
     ...
  </env:Body>
</env:Envelope>
```

Figura 4.3 - Exemplo de uma resposta SOAP enviada via HTTP

A mensagem da Figura 4.3 indica que a resposta do servidor também é enviada usando a versão 1.1, o servidor não fecha a conexão devido à ausência do atributo “Connection: close”. O atributo “Transfer-Encoding: chunked” informa que a mensagem será transmitida em blocos usando a técnica de *streaming Chunked Transfer Coding*. A mensagem da Figura 4.4 indica que o servidor fechará a conexão após o envio da resposta ao cliente devido à presença do atributo “Connection: close”.

```
HTTP/1.1 200 OK
Content-Type: text/xml;charset=utf-8
Server: Apache-Coyote/1.1
Connection: close

<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
              xmlns:xsd="http://www.w3.org/2001/XMLSchema
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <env:Body>
    ...
  </env:Body>
</env:Envelope>
```

Figura 4.4 - Exemplo de uma resposta SOAP enviada via HTTP fechando a conexão

Os atributos “Content-Length”, “Connection” e “Transfer-Encoding” do cabeçalho HTTP são importantes porque eles impactam no desempenho. Essa diretriz é importante para verificar como a mensagem SOAP é transportada usando o protocolo HTTP, pois influencia como a mesma será processada. Além disso, a análise das mensagens também deve ser feita para validar se o projeto da interface WSDL está correto com relação à assinatura das operações, à formatação das mensagens e aos tipos de dados usados.

Diretriz 4: Verifique o parser suportado pelo toolkit

Baseado no fato que os documentos XML representando as mensagens SOAP são grandes e complexos, é necessário escolher um modelo de *parsing* eficiente e que apresente um bom gerenciamento de memória. A questão é que o número de diferentes modelos tem crescido, então não é trivial determinar qual modelo usar baseando-se em suas características de desempenho e de facilidade de uso.

Como diferentes implementações do mesmo modelo de *parsing* podem apresentar diferente desempenho, é importante identificar o *parser* XML adotado pelos *Web Services toolkits*, porém a implementação utilizada pelo *toolkit* quando o seu código não é aberto não é imediatamente determinada. Essa diretriz sugere que os projetistas da aplicação desprendam algum tempo analisando o *parser* do *toolkit*, baseando-se na sua documentação e em resultados publicados em outros trabalhos.

É recomendado também não habilitar a validação das mensagens pelo *parser* se o desempenho for um requisito prioritário, porém é uma boa prática de programação validar os valores dos parâmetros antes de invocar os métodos.

Para melhorar o desempenho do processo de recebimento de uma mensagem SOAP, uma implementação do modelo de processamento *Pull Parsing* poderia ser adotada pelos *toolkits* porque é mais eficiente e permite que o processo de *parsing* da mensagem inicie antes que todo o documento tenha sido recebido.

Alguns *toolkits* permitem que o desenvolvedor escolha qual a implementação do modelo de *parsing* usar para varrer e validar as mensagens SOAP. Nesses casos, é interessante fazer uma avaliação prévia das implementações disponíveis para determinar

qual delas apresenta o melhor desempenho para tratar os tipos de dados de uma determinada aplicação, pois nem sempre os gargalos associados ao *parser* são claros, e futuramente, podem ser necessárias alterações no código do *toolkit* para trocar de modelo.

Mesmo existindo progresso no desempenho do *parsing* dos documentos XML, é importante avaliar a implementação adotada por um *toolkit* antes de selecioná-lo. Os *toolkits* mais flexíveis permitem que o desenvolvedor selecione a implementação mais apropriada para sua aplicação, e atualize, de forma simplificada, seu código com implementações mais eficientes à medida que sejam desenvolvidas.

Diretriz 5: Monitore o tráfego de pacotes

Atualmente, os códigos das aplicações cliente e servidor são construídos baseados em *stubs* e *skeletons* que são automaticamente gerados a partir da interface do serviço e escondem do programador os detalhes da comunicação na rede. É importante avaliar os custos associados diretamente à camada de comunicação antes de adotar um *toolkit*, principalmente porque aplicações *Web Services* consomem muito tempo de processamento e transmitem muitos dados. Quanto mais dados são transmitidos na rede, mais controle dos pacotes é necessário.

Mesmo que a especificação SOAP não determine qual protocolo de transporte adotar, as mensagens SOAP são mais freqüentemente transportadas usando o protocolo HTTP. As vantagens de usar HTTP são claras, pois o mesmo é universalmente suportado e seu tráfego normalmente está configurado para passar por *firewalls*. Porém é necessário identificar os gargalos específicos do protocolo que afetam o desempenho de *Web Services*.

Analisando o tráfego de pacotes entre o cliente e o servidor é possível identificar alguns detalhes da transferência dos dados e gargalos específicos da comunicação que podem ter um significativo impacto no desempenho geral da aplicação *Web Services*. Também permite um entendimento detalhado da seqüência de pacotes gerada pelos diferentes *toolkits*, pois as implementações apresentam variações na seqüência dos pacotes e, conseqüentemente, na sua eficiência.

Essa diretriz auxilia a identificação dos gargalos relacionados à comunicação que foram apresentados na Seção 3.5.5 e no entendimento do funcionamento do *toolkit* para estabelecer a comunicação. Atualmente, existem várias ferramentas gráficas ou baseadas em linha de comando que automatizam o monitoramento do tráfego na rede [Kennington, 2005]. Monitorando o tráfego, é possível identificar os atrasos e comparar os pacotes trocados, além de responder as seguintes questões:

- Qual o número total de pacotes trocados entre o cliente e o servidor ?
- Em quantos pacotes de dados uma mensagem SOAP grande é quebrada, uma vez que as mensagens pequenas são preenchidas em um único pacote?
- Qual o número total de bytes transferidos?
- Quantas conexões foram abertas?
- Qual o custo para estabelecer e fechar uma conexão?
- Qual é o impacto dos algoritmos *Nagle* e *TCP delayed ACK*?

Dessa forma, essa diretriz avalia o desempenho do *toolkit* para uma aplicação particular baseando-se na sua implementação da camada de comunicação e nos custos adicionais inerentes ao protocolo de transporte.

Uma solução eficiente em termos de rede deveria suportar as técnicas de conexões persistentes e *Chunked Transfer Coding* suportadas pela especificação do protocolo HTTP 1.1. Tais técnicas são otimizações e melhoram o desempenho de *Web Services* (ver Seção 3.6).

Diretriz 6: Quantifique o desempenho do Web Services toolkit

Desempenho é uma medida da produtividade de uma aplicação e um importante critério para distinguir e selecionar o *Web Services toolkit* mais apropriado. As seguintes métricas têm sido comumente adotadas e podem ser usadas para quantificar a qualidade de um serviço em termos do seu desempenho:

- *Round Trip Time* (RTT): representa o tempo médio para enviar e receber uma mensagem, a partir do cliente ao servidor e de volta ao cliente. Essa métrica

inclui o tempo requerido para serializar e deserializar os argumentos e valores de retorno e o custo para transmitir os dados na rede. Quando o cliente e o servidor estão rodando na mesma máquina, o custo da rede é reduzido;

- Latência: representa o gargalo imposto pelo *toolkit* para enviar e receber uma mensagem sem parâmetros e sem valores de retorno;
- Vazão: representa o número de requisições dos clientes completadas dentro de uma certa unidade de tempo, tipicamente em segundos. Quando a taxa de requisição excede a capacidade do servidor, ou seja, quando a taxa de requisição é maior que a taxa de serviço, a vazão decresce e o tempo de resposta aumenta. A vazão deveria aumentar com o aumento do número de clientes simultâneos até saturar a capacidade máxima do servidor. A vazão e o RTT são inversamente proporcionais.
- Escalabilidade: métrica que avalia a degradação do desempenho quando vários clientes enviam requisições ao servidor simultaneamente.

Evitando qualquer processamento do lado do servidor quando calculando o RTT, como por exemplo, o acesso ao banco de dados, o resultado representará apenas o gargalo introduzido pelo uso da tecnologia *Web Services* na chamada remota, sem a interferência dos gargalos inerentes à execução da aplicação.

Para completar o processo de avaliação de desempenho dos *Web Services toolkits*, outras métricas podem ser calculadas: o tempo de instanciação do *stub*, o tamanho total das mensagens, o número total de pacotes transmitidos na rede e o custo para tratar uma exceção ou erro.

O tempo de instanciação do *stub* é o tempo para levantar e inicializar o *stub* que pode ser de forma estática ou dinâmica. Existe um *trade-off* entre desempenho e invocação dinâmica, pois o tempo de inicialização dos *toolkits* que utilizam *proxy* gerados em tempo de execução é maior que os *toolkits* que geram o código do *stub* estaticamente. Porém, o instanciamento dinâmico torna as aplicações clientes mais adaptáveis a possíveis alterações na interface do serviço.

O tamanho das mensagens representa o número total de *bytes* da mensagem XML trocada em cada transação, ou seja, no número de *bytes* da requisição do cliente e da resposta do servidor.

O número de pacotes associados a uma determinada chamada remota, incluindo os pacotes de confirmação e os referentes à abertura e fechamento da conexão, pode ser uma importante métrica quando grandes mensagens são avaliadas, pois o tamanho das mensagens influencia no número total de pacotes necessários para transmitir os dados.

O tempo para tratar uma exceção específica do usuário representa o custo para o cliente fazer a requisição, o servidor levantar a exceção e o cliente fazer o seu tratamento. Como o usuário é livre para projetar sua aplicação, uma exceção pode ser tão complexa quanto uma entidade de negócio.

4.4 Considerações Finais

Atualmente, os desenvolvedores podem escolher entre as várias soluções tecnológicas para construir uma aplicação *Web Services* e cada escolha feita pode afetar tanto o desempenho quanto a escalabilidade da aplicação desenvolvida. Outro fator que pode degradar o desempenho é a granularidade da interface do serviço. A interface deve ser projetada de forma a minimizar o tráfego na rede, evitando a troca de mensagens desnecessárias.

Para os projetistas terem uma posição correta durante a seleção do *toolkit* mais apropriado para desenvolver seu serviço que demanda por alto desempenho, seis diretrizes foram desenvolvidas e apresentadas nesse capítulo:

- **Diretriz 1.** Adote o estilo *Document/Literal Wrapped*;
- **Diretriz 2.** Utilize mensagens de tamanhos e complexidades diferentes;
- **Diretriz 3.** Analise as mensagens SOAP transportadas na rede;
- **Diretriz 4.** Verifique o *parser* suportado pelo *toolkit*;
- **Diretriz 5.** Monitore o tráfego de pacotes;

- **Diretriz 6.** Quantifique o desempenho do *Web Services toolkit*.

Em recentes investigações, foi descoberto que o estilo de codificação das mensagens SOAP, além de representar um acordo entre o cliente e o servidor sobre como interpretar as mensagens, também afeta o desempenho do *toolkit*.

Esse problema foi abordado pela Diretriz 1, onde foram descritos os diferentes estilos e mostrado o *trade-off* entre interoperabilidade e desempenho associado a cada estilo. Além disso, a Diretriz 1 funciona como uma otimização para os gargalos referentes ao tamanho da mensagem e aos custos de serialização e deserialização das mensagens.

De forma geral, as diretrizes 3, 4 e 5 foram desenvolvidas para facilitar a identificação dos gargalos inerentes aos *Web Services toolkits* e os diretamente associados ao protocolo HTTP.

As diretrizes 2 e 6 visam padronizar o processo de avaliação de desempenho de qualquer *toolkit*, permitindo que os resultados da avaliação sejam abrangentes e propondo métricas, respectivamente.

A partir das diretrizes, foi elaborado um processo focando no passo a passo que deve ser executado durante a avaliação de desempenho de *Web Services toolkits*. Para reduzir o tempo gasto na sua execução, algumas das tarefas desse processo serão automatizadas por um utilitário de código aberto e implementado em Java. Tanto o processo quanto o utilitário serão descritos no próximo capítulo.

5 Processo e um Utilitário para a Avaliação de Desempenho de *Web Services Toolkits*

5.1 Introdução

A avaliação de desempenho de *Web Services* pode não ser uma tarefa fácil, consumindo muito tempo e demandando desenvolvedores ou arquitetos com experiência na tecnologia. Devido a essas dificuldades, o processo de avaliação é muitas vezes realizado depois que a aplicação *Web Services* é completamente implementada ou, no pior caso, é simplesmente omitido. Porém o desempenho de *Web Services* não pode ser desconsiderado, uma vez que ainda é um problema em aberto, onde os *toolkits* apresentam eficiências diferentes.

Nesse capítulo serão apresentados um processo para avaliação de desempenho de *Web Services toolkits* e um utilitário, denominado JWSPerf (*Java Web Services Performance*), cujo objetivo é automatizar alguns passos desse processo. Dessa forma, é possível identificar os gargalos de desempenho e selecionar o *toolkit* "ideal" para expor o serviço.

5.2 Processo de Avaliação de Desempenho

O processo para a avaliação de desempenho de *Web Services toolkits* apresentado nessa seção é uma extensão das diretrizes apresentadas no capítulo anterior e seu objetivo é uniformizar a avaliação de desempenho de diferentes *toolkits*. De forma geral, o processo representa um guia prático, definindo um passo a passo para executar a avaliação, porém sempre embasado pelas diretrizes.

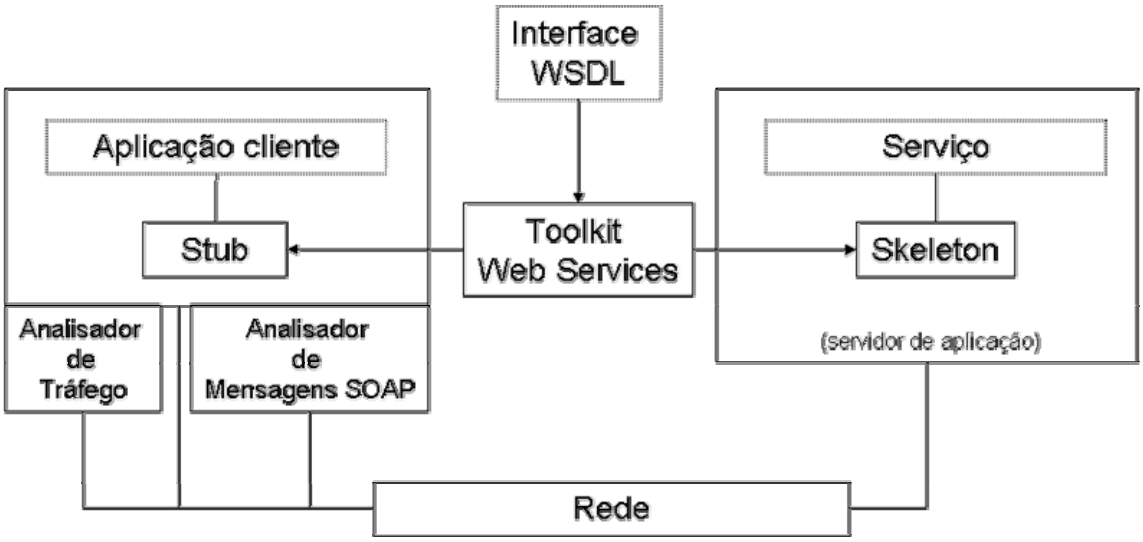

Figura 5.1 - Componentes do processo de avaliação de desempenho

O processo proposto é composto por um conjunto de tarefas que devem ser executadas do lado do cliente, pois o serviço deve estar implementado e rodando no servidor de aplicação. A Figura 5.1 ilustra a estrutura necessária para executar o processo seguindo as seguintes tarefas:

1. Recuperar a mesma interface WSDL utilizada para desenvolver o serviço que será avaliado. Essa interface deveria conter operações usando tipos de dados simples e complexos de vários tamanhos, assim respeitando a Diretriz 2 (Utilize mensagens de tamanhos e complexidades diferentes). Como a interface WSDL representa o contrato entre o cliente e o servidor, não é necessário que ambos adotem o mesmo *toolkit*;
2. De acordo com a Diretriz 1, selecionar um *Web Services toolkit* que suporte o estilo de codificação *Document/Literal Wrapped*;

3. Verificar, a partir da sua documentação, se o *parser* adotado pelo *toolkit* implementa o modelo SAX ou XPP, pois o modelo DOM é o mais ineficiente. Essa tarefa deve ser realizada para atender à Diretriz 4 (Verifique o *parser* suportado pelo *toolkit)*;
4. Gerar, a partir da interface WSDL, o *stub* cujo código varia de acordo com o *Web Services toolkit* selecionado. A maioria dos *toolkits* implementados em Java disponibiliza ferramentas para executar esse passo a partir de linha de comando passando apenas os parâmetros necessários como, por exemplo, o diretório onde será gerado o código e o caminho da interface WSDL. Dessa forma, para executar esse passo corretamente, é importante analisar a documentação da ferramenta disponibilizada, uma vez que cada *toolkit* adota uma maneira diferente;
5. Implementar a aplicação cliente, codificando todas as chamadas de operações definidas na interface WSDL. No mínimo, a implementação de cada chamada de operação deve acessar o *stub* gerado para invocar a operação do serviço remoto. O código para recuperar o *stub* varia de acordo com o *toolkit* selecionado. Além da chamada de operação em si, também é necessário escrever o código responsável pela coleta do tempo e cálculo das métricas de desempenho;
6. Invocar, passando os parâmetros desejados, as operações suportadas pelo serviço a fim de calcular o RTT (*Round Trip Time*) e a vazão. Dessa forma, a Diretriz 6 (Quantifique o desempenho do *Web Services toolkit*) é atendida;
7. Analisar as mensagens SOAP para determinar o seu tamanho, o estilo de codificação, a versão do protocolo HTTP e os atributos do seu cabeçalho a fim de atender à Diretriz 3 (Analise as mensagens SOAP transportadas na rede);
8. Adotar um analisador de tráfego de pacotes, a fim de identificar os gargalos de comunicação. Dessa forma, a Diretriz 5 (Monitore o tráfego de pacotes) é satisfeita.

Dependendo da quantidade de *toolkits* que serão analisados e da experiência do desenvolvedor ou arquiteto na tecnologia *Web Services*, esse processo de avaliação pode demandar muito tempo, pois é necessário estudar a documentação dos *toolkits* para executar os passos 2, 3 e 4. Além disso, para atender à Diretriz 2, a interface deve definir várias operações, implicando em um tempo maior para executar a tarefa 5. As

tarefas 6, 7 e 8 também consomem muito tempo, porque a mesma operação deve ser executada várias vezes, a fim de coletar resultados estáveis.

Para simplificar a execução dessas tarefas, o utilitário JWSPerf (*Java Web Services Performance*) foi desenvolvido para automatizar os passos 4, 5 e 6 desse processo. Os passos 1 e 2 são executados apenas configurando os parâmetros dos arquivos de propriedades do utilitário JWSPerf. Os passos 7 e 8 são auxiliados por ferramentas como, por exemplo, TCP Monitor, Ethereal, WinDump e TCP Sniffer. O passo 3 consiste pesadamente na análise da documentação dos *toolkits*. A Tabela 5.1 apresenta o mapamento das diretrizes com as tarefas do processo e o responsável pela execução.

Tabela 5.1 - Mapeamento entre as diretrizes, o processo e o responsável pela execução

Diretrizes	Processo	Execução
Diretriz 1	Tarefa 2	Configuração do parâmetro no arquivo de propriedade do utilitário JWSPerf.
Diretriz 2	Tarefa 1	Configurar parâmetro no arquivo de propriedade do utilitário JWSPerf.
	Tarefas 4 e 5	Módulo responsável pela geração de classes de teste a partir da interface WSDL do utilitário JWSPerf.
Diretriz 3	Tarefa 7	Analisador de mensagens SOAP.
Diretriz 4	Tarefa 3	Análise manual da documentação do *toolkit*.
Diretriz 5	Tarefa 8	Analisador de tráfego de rede.
Diretriz 6	Tarefa 6	Módulo responsável pela invocação e coleta das métricas de desempenho do utilitário JWSPerf.

5.3 Utilitário JWSPerf

JWSPerf é um utilitário de código aberto, fácil de usar e com suporte a múltiplas implementações *Web Services* [Machado and Ferraz, 2006]. Até o momento, três *Web Services toolkits* implementados em Java são suportados – Axis da Apache [Apache Axis, 2004], JWSDP (*Java Web Services Developer Pack*) da Sun [Sun, 2004] e SSJ

(*Systinet Server for Java*) [Systinet, 2004]. Cada um desses *toolkits* possui características diferentes para desenvolver a aplicação e instalar o serviço no servidor de aplicação. Todos esses *toolkits* atualmente incorporados ao JWSPerf atendem à Diretriz 1 e, conseqüentemente, à tarefa 2 do processo acima. Entretanto, *Document/Literal Wrapped* não é o estilo de codificação padrão desses *toolkits*, exceto para o *toolkit* SSJ, que permite que o usuário configure o estilo padrão durante a sua instalação.

Do ponto de vista de projeto, o JWSPerf é constituído por dois módulos (Figura 5.2): o de invocação e o de geração das classes de teste. Esses módulos foram projetados para permitir que qualquer desenvolvedor automaticamente gere a aplicação cliente, que consiste nas classes de teste e nos artefatos específicos do *toolkit*, a partir da interface WSDL, independentemente da sua complexidade. Além disso, também invoque as operações definidas na interface do serviço e apresente os resultados no final da execução. Devido a essa estruturação é que JWSPerf automatiza as tarefas 4, 5 e 6 do processo. As próximas seções detalham o funcionamento e a implementação de cada um desses módulos.

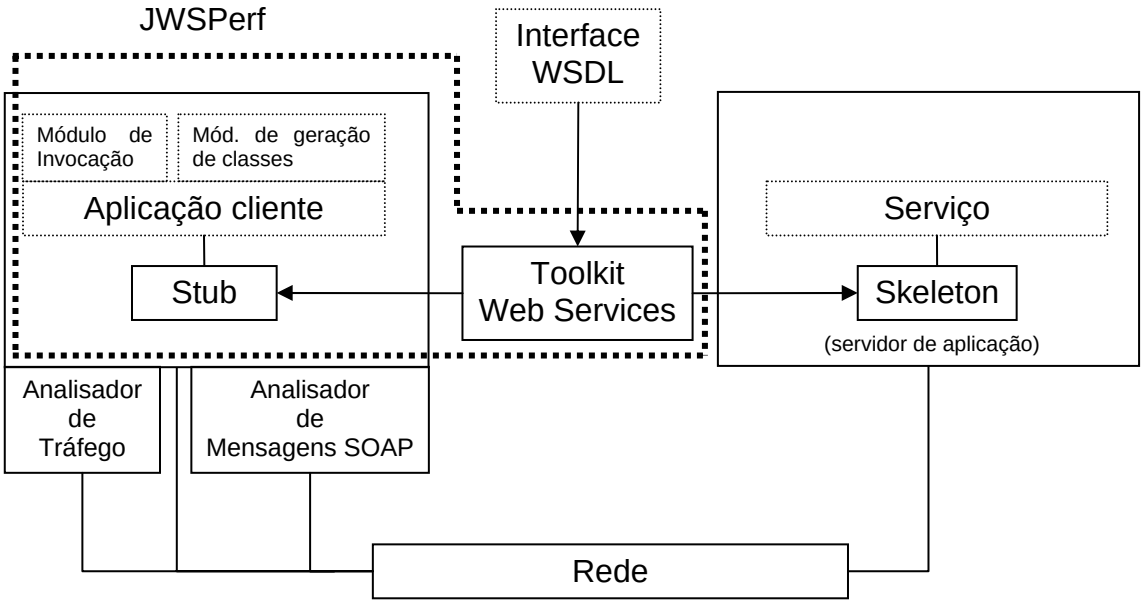

Figura 5.2 - Papel do utilitário JWSPerf

Módulo de Geração das Classes de Teste

O módulo de geração das classes de teste foi construído baseando-se na implementação do *parser* WSDL do *toolkit* Axis, e seu objetivo é gerar uma classe de teste para cada operação definida na interface WSDL. A Figura 5.3 ilustra o seu diagrama de classes, incluindo as classes do próprio *toolkit* Axis que foram reusadas – `Emitter`, `SymbolTable`, `JavaGeneratorFactory` e `JavaClassWriter`.

A classe `Emitter` funciona como um *parser* de documentos WSDL que gera, a partir dos parâmetros configurados, *stubs, skeletons* e classes representando os tipos de dados. A classe `SymbolTable` é um tipo definido para representar em memória o documento WSDL varrido.

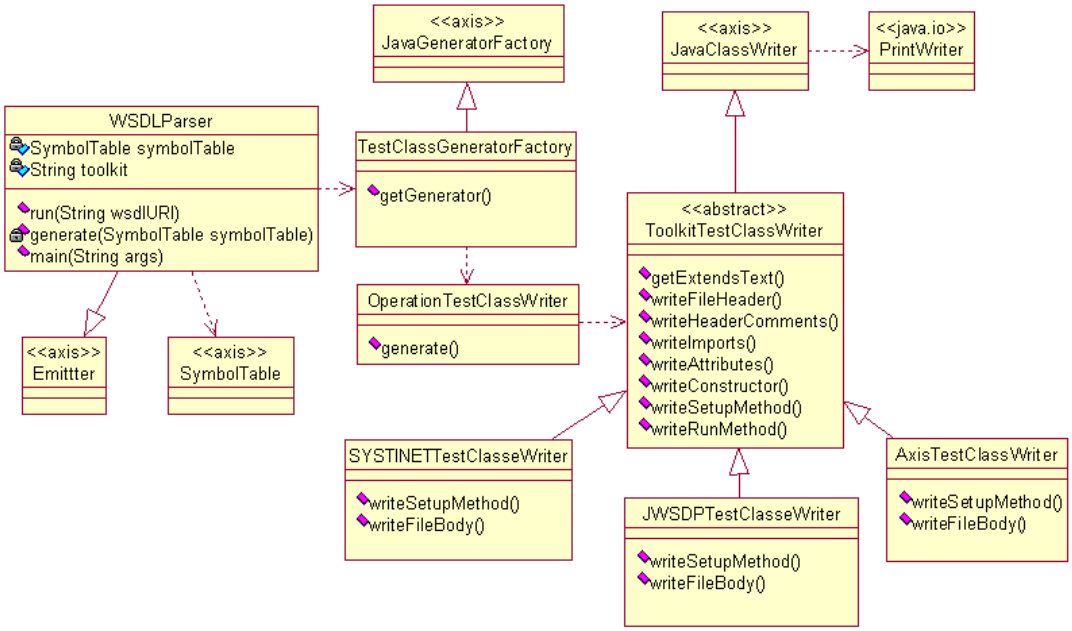

Figura 5.3 - Diagrama de classes do módulo de geração das classes de teste

A classe `WSDLParser` é uma especialização da classe `Emitter`, também funcionando como *parser*, porém foi adaptada para gerar apenas as classes de teste das operações definidas na interface WSDL. Os principais métodos definidos são `run` e `generate`. O primeiro cria uma instância da classe `SymbolTable` e invoca seu método `populate` para carregar a interface. Uma vez inicializada, classe `SymbolTable` é passada como parâmetro para o método `generate`.

As classes `TestClassGeneratorFactory`, `OperationTestClassWriter` e `ToolkitTestClassWriter` são responsáveis pela criação das classes de teste de acordo com o *toolkit* configurado. A classe `ToolkitTestClassWriter` herda da classe `JavaClassWriter`, que define os métodos responsáveis pela escrita do código Java das classes de teste geradas.

As classes `AxisTestClassWriter`, `JWSDPTestClassWriter` e `SYSTINETTestClassWriter` são especializações da classe `ToolkitTestClassWriter` e apenas contêm os métodos cuja implementação varia de acordo com o *toolkit*. A regra é criar uma classe <`TOOLKIT`>`TestClassWriter` para cada *toolkit* suportado, porque as implementações dos métodos `writeRunMethod` e `writeSetupMethod` dependem do *toolkit* e do modo como o *stub* é instanciado.

Uma instância da classe `PrintWriter` é criada para representar uma abstração alto nível do arquivo Java, que posteriormente será gerado fisicamente. Durante a execução, todo o código da nova classe é armazenado nessa instância.

5.3.1.1 Diagrama de Seqüência

Nessa seção será apresentado o diagrama de seqüência referente à execução do módulo de geração das classes de teste, mostrando a interação e a troca de mensagens entre as classes participantes (ver Figura 5.4). O diagrama apresentado é uma visão alto nível do funcionamento real, pois algumas classes foram omitidas.

A execução desse módulo é iniciada quando o método `main` da classe `WSDLParser` é invocado, que após validar o preenchimento dos parâmetros recebidos, cria uma instância da classe `WSDLParser`, que por sua vez, cria uma instância da classe `TestClassGeneratorFactory`.

Finalizada a criação da classe `WSDLParser`, o seu método `run` é acionado para iniciar o processo de geração das classes de teste. Esse método executa duas tarefas importantes. A primeira consiste em criar uma instância da classe `SymbolTable`, a fim de invocar seu método `populate`. Se a localização do arquivo WSDL for uma URL, o usuário deve garantir que o serviço esteja rodando; caso contrário, uma exceção será

levantada. A segunda tarefa é invocar o método `generate` da classe `WSDLParser`. A principal ação do método `generate` da classe `WSDLParser` é invocar o método `getGenerator` da classe `TestClassGeneratorFactory`, que simplesmente cria uma instância da classe `OperationTestClassWriter` – responsável pela geração das classes de teste, uma para cada operação definida na interface.

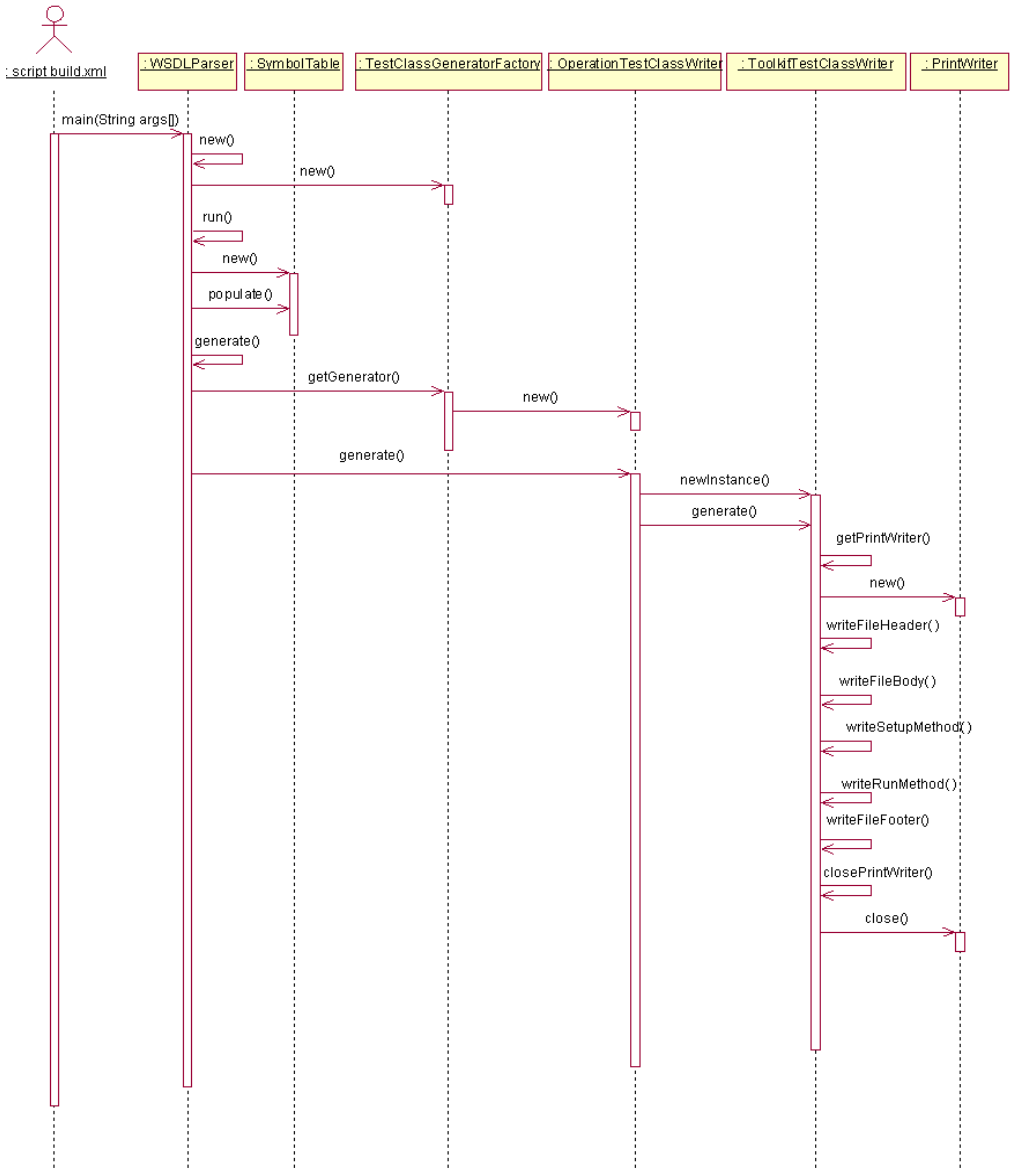

Figura 5.4 - Diagrama de seqüência do módulo de geração das classes de teste

Após a criação do objeto `OperationTestClassWriter`, seu método `generate` é invocado. A implementação desse método consiste em varrer todas as operações definidas na interface WSDL e para cada operação, o método `generate` da classe

`JavaClassWriter` é invocado. Esse último cria uma instância da classe `PrintWriter`, representando o arquivo Java a ser gerado, e invoca os métodos `writeFileHeader` para gerar o cabeçalho da classe, `writeFileBody` que escreve os atributos, o construtor e os demais métodos, e o método `writeFileFooter` para finalizar a declaração. Por fim, o método `closePrintWriter` é invocado, gerando fisicamente a classe com o código especificado.

Dependendo do *toolkit* informado como parâmetro pelo usuário, apenas uma das especializações da classe `ToolkitTestClassWriter` (`AXISTestClassWriter`, `JWSDPTestClassWriter` e `SYSTINETTestClassWriter`) é criada em tempo de execução. As implementações dos métodos `writeFileBody` e `writeSetupMethod` são diferentes para cada um dos *toolkits*.

Módulo de Invocação

O objetivo do módulo de invocação é rodar a operação solicitada pelo usuário e coletar as métricas de desempenho referentes à execução. Para isso, é preciso apenas configurar algumas propriedades antes de iniciar a execução. Para realizar sua tarefa, esse módulo é composto por classes e interfaces, com responsabilidades bem definidas (ver Figura 5.5).

A classe `JWSPerfDriver` é o ponto inicial da execução desse módulo. Suas principais tarefas são carregar os atributos da classe `Config`, gerenciar o ciclo de vida dos agentes rodando simultaneamente e solicitar à classe `MetricHandler` a geração do arquivo com os resultados.

A classe `Config` define os atributos que armazenam os parâmetros informados pelo usuário antes da execução e ficam acessíveis durante todo o processamento (ver Tabela 5.2, pp. 84). Apenas dois métodos estão definidos: `init`, que simplesmente inicializa seus atributos, e o método `getResultFileName`, que gera o caminho completo e o nome do arquivo de saída.

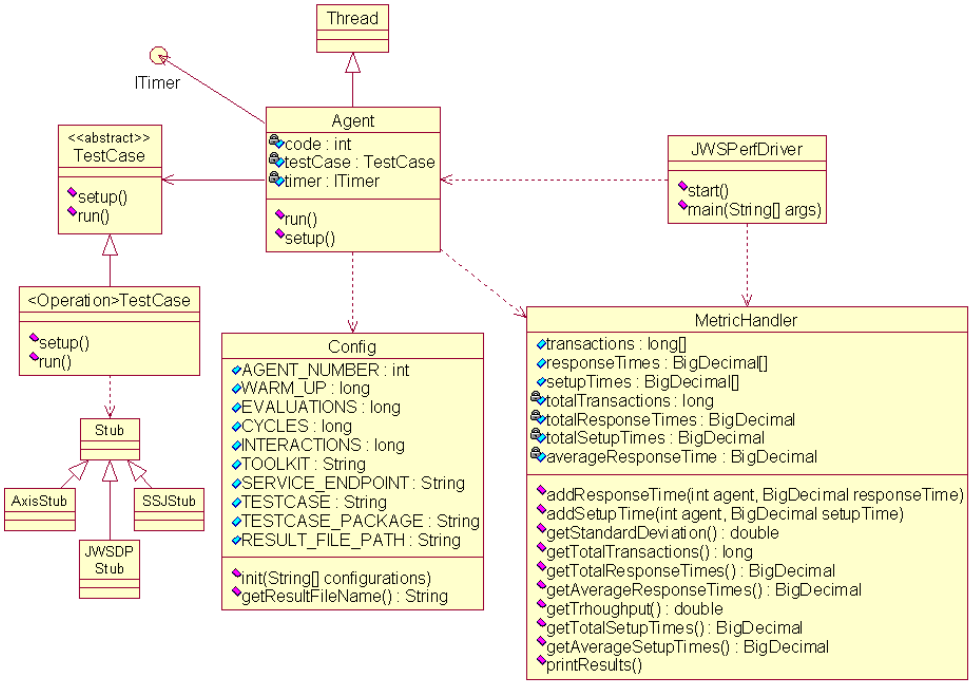

Figura 5.5 - Diagrama de classes do módulo de invocação

A classe `Agent` é uma especialização da classe `java.lang.Thread` e representa um cliente do serviço. É possível configurar múltiplos agentes rodando concorrentemente para simular um cenário real, onde uma *thread* é criada para cada novo agente a ser executado. Cada agente tem um identificador único, uma referência da biblioteca de medição de tempo e uma instância da classe de teste que será executada, anteriormente gerada pelo módulo de geração das classes de teste.

Os métodos definidos na classe `Agent` são `setup`, que se encarrega de instanciar a classe de teste em tempo de execução, e o método `run`, que invoca a classe de teste. De modo geral, a tarefa de um agente é executar a operação encapsulada na classe de teste. Os agentes também têm a responsabilidade de enviar à classe `MetricHandler` o tempo de resposta de cada requisição completada e o tempo para instanciar o *stub*.

A classe `TestCase` é uma abstração que declara os métodos que as classes de teste devem implementar, independentemente do tipo de tecnologia utilizada para fazer a invocação do serviço remoto. Dois métodos abstratos estão definidos: `setup`, que

contém o código necessário para instanciar o *stub*, e o método `run`, que invoca a operação desejada usando o *stub*.

Tabela 5.2 - Descrição dos atributos da classe Config

Atributo	Descrição
AGENT_NUMBER	Indica o número de agentes que devem rodar simultaneamente.
WARM_UP	Representa o número de invocações à operação para estabilizar o tempo.
EVALUATIONS	Indica o numero de avaliações que será realizada. Cada avaliação é composta por vários ciclos.
CYCLES	Indica a quantidade de ciclos de uma avaliação. Cada ciclo é composto por várias interações.
INTERACTIONS	Representa o número de invocações à operação dentro de um determinado ciclo.
TOOLKIT	Representa o *toolkit* selecionado pelo usuário.
SERVICE_ENDPOINT	Atributo utilizado pelo método `setup` de cada classe de teste para recuperar o *stub* do serviço solicitado.
TESTCASE	Nome da classe de teste que deverá ser instanciada durante a execução.
TESTCASE_PACKAGE	Representa o pacote das classes de teste e, juntamente com o atributo acima, é usada para instanciar a classe de teste usando a tecnologia *Reflection* de Java.
RESULT_FILE_PATH	Diretório onde o arquivo de saída deverá ser gerado.

A classe `<Operation>TestCase` é uma implementação concreta da classe `TestCase` e o código dos seus métodos varia de acordo com o *toolkit* usado para fazer a requisição e com a operação invocada. O módulo de geração das classes de teste é responsável por gerar uma classe `<Operation>TestCase` para cada operação da interface WSDL.

Apenas em tempo de execução o utilitário instancia a classe de teste configurada nos atributos `TESTCASE_PACKAGE` e `TESTCASE` da classe `Config`, utilizando a tecnologia *Reflection* de Java. Caso o usuário deseje testar outra operação, basta configurar a classe de teste responsável por invocar a operação desejada.

A classe `MetricHandler` é responsável por calcular as métricas utilizadas na avaliação de desempenho dos *Web Services toolkits* analisados. Essas métricas calculadas são: 1) o tempo de resposta médio (RTT); 2) a vazão e 3) o tempo médio para instanciar o *stub*. Todos os resultados calculados são armazenados em um arquivo gerado no diretório especificado pelo atributo `Config.RESULT_FILE_PATH`.

5.3.1.2 Diagrama de Seqüência

Nessa seção será apresentado um diagrama de seqüência da execução do módulo de invocação (Figura 5.6). Esse diagrama também é uma visão de alto nível do funcionamento real.

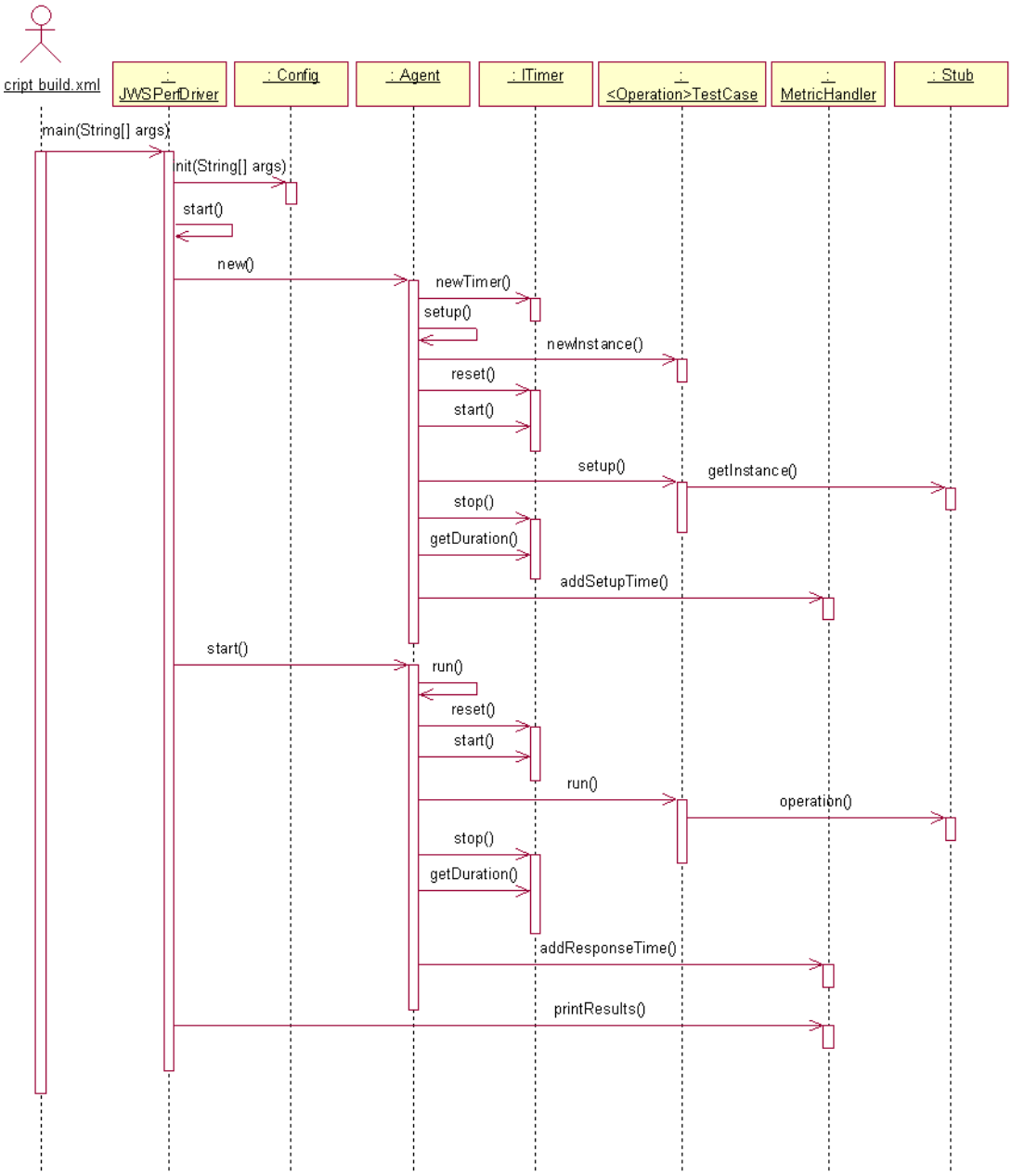

Figura 5.6 - Diagrama de seqüência do módulo de invocação

O usuário inicia a invocação do serviço executando o método `main` da classe `JWSPerfDriver`, passando os parâmetros necessários para inicializar a classe `Config`. Finalizada essa fase de configuração da execução, uma instância da classe `JWSPerfDriver` é criada e seu método `start` é invocado.

A primeira tarefa do método `start` é instanciar os objetos do tipo `Agent`, onde o número de agentes a ser criado é determinado pelo atributo `AGENT_NUMBER` da classe `Config`. Criar um agente significa recuperar uma instância da interface `ITimer` invocando seu método estático `newTimer` e, em seguida, executar seu método `setup`, que se encarrega de criar uma instância, em tempo de execução, da classe de teste que encasula a operação solicitada pelo usuário.

Uma vez instanciada a classe de teste, seu método `setup` é invocado, a fim de recuperar, de forma estática ou dinâmica, o *stub* que será utilizado para acessar o serviço e abstrair as questões referentes à comunicação. Como o tempo para instanciar o *stub* é uma métrica de desempenho proposta pela Diretriz 6, o mesmo é armazenado na classe `MetricHandler` através do método `addSetupTime`.

Após criar todos os agentes, a segunda tarefa do método `start` da classe `JWSPerfDriver` é invocar o método `start` da *thread* agente, que implicitamente executa o seu método `run`.

A implementação do método `run` da classe `Agent` simplesmente executa o método `run` da sua classe de teste que, por sua vez, invoca a operação na classe *stub* anteriormente recuperada. O tempo para executar a operação também é uma métrica de desempenho proposta pela Diretriz 6, portanto, o mesmo deve ser armazenado na classe `MetricHandler` invocando o seu método `addResponseTime`.

Por fim, a terceira tarefa do método `start` da classe `JWSPerDriver` é executar o método `printResults` da classe `MetricHandler`, depois que todos os agentes finalizaram sua execução e suas métricas foram coletadas. Seu objetivo é calcular as métricas de desempenho referentes à execução da operação e ao *toolkit* configurado e gerar um arquivo com os resultados no diretório de saída. Um arquivo diferente é gerado para cada execução.

5.4 Instalando o JWSPerf

Nessa seção será apresentado um guia rápido composto pelos passos básicos para um desenvolvedor instalar o utilitário de forma simples e rápida. Primeiramente, é necessário instalar separadamente cada *Web Services toolkit* suportado pelo utilitário, pois os mesmos não fazem parte do seu código. Atualmente, o desenvolvedor deve garantir a instalação dos *toolkits* Axis, JWSDP e SSJ. Após finalizar a instalação de cada um desses *toolkits,* os seguintes passos devem ser realizados para instalar o utilitário:

1. Fazer o *download* do arquivo "jwsperf 0.0.9 beta.zip" disponível no endereço http://code.google.com/p/jwsperf/;
2. Descompactar o arquivo baixado em qualquer diretório do sistema de arquivo. Dentro do diretório <JWSPERF_HOME>/jwsperf, onde <JWSPERF_HOME> representa o diretório de instalação do utilitário, devem estar presentes todos os diretórios (ver Figura 5.7) e arquivos de configuração (Tabela 5.3);
3. Alterar o arquivo env.bat, a fim de atualizar o *path* da máquina com as bibliotecas para rodar os *Web Services toolkits*;
4. Configurar as propriedades "axis.home", "jwsdp.home" e "ssj.home" do arquivo build.properties com os diretórios onde os *toolkits* Axis, JWSDP e SSJ foram instalados, respectivamente.

Tabela 5.3 - Principais arquivos de configuração

Arquivos	Descrição
build.xml	Descreve todos os comandos/tarefas para interagir com o utilitário.
build.properties	Define as propriedades referentes ao diretório de instalação dos *toolkits*.
env.bat	Arquivo de *batch* que configura as variáveis de ambiente para executar os *toolkits*.
parameters.properties	Define propriedades passadas como parâmetros ao módulo de invocação.
config.xml	Arquivo de configuração apenas utilizado pelo *toolkit* JWSDP.
jwsperf.xml	Define as propriedades referentes ao serviço sendo executado e configurações necessárias para executar o módulo de geração das classes de teste.
path.xml	Define o *classpath* necessário para executar cada *toolkit*.
properties.xml	Define as propriedades que representam a estrutura de diretórios.

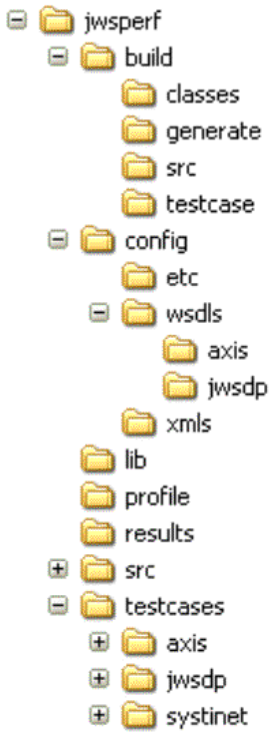

Figura 5.7 - Estrutura de diretórios do utilitário JWSPerf

A Figura 5.7 ilustra a estrutura de diretório do JWSPerf, onde cada um tem seu papel bem definido. Segue a descrição de suas responsabilidades:

- **jwsperf**: diretório raiz que contém os arquivos build.xml, build.properties e env.bat e os demais diretórios do utilitário;
- **src**: armazena o código fonte do utilitário JWSPerf, incluindo as classes dos módulos de invocação e de geração das classes de teste;
- **build**: usado como um repositório temporário dos arquivos gerados para construir todo o código que será executado. Contém os seguintes subdiretórios:
    - **classes**: armazena os arquivos resultantes da compilação das classes do utilitário e as geradas pelo mesmo;
    - **generate**: armazena as classes clientes geradas pelo utilitário cujo código é específico do *toolkit*;
    - **src**: contém o código fonte do utilitário, as classes de teste geradas e o código do diretório “generate”;
    - **testcase**: contém os arquivos resultantes da compilação do módulo de geração das classes de teste.

- **config**: contém os arquivos de configuração do utilitário (ver Tabela 5.3). Fazem parte desse diretório:
    - **etc**: armazena o arquivo de configuração parameters.properties;
    - **wsdls**: armazena os arquivos WSDL separados por *toolkit*;
    - **xmls**: contém os arquivos de configuração do tipo XML necessários para gerar as classes e rodar o utilitário – jwsperf.xml, path.xml, config.xml e properties.xml.
- **lib**: armazena as bibliotecas necessárias para construir e rodar o utilitário;
- **profile**: armazena os resultados da investigação (*profiling*) da execução do utilitário;
- **results**: contém os arquivos de saída com os resultados da execução do utilitário;
- **testcases**: armazena as classes de teste geradas pelo módulo de geração das classes de teste. As classes geradas são sobrescritas a cada execução, caso a propriedade "generate.testcase" do arquivo jwsperf.xml esteja habilitada. As classes geradas são separadas por *toolkit*.

5.5 Executando o Utilitário JWSPerf

Para facilitar a interação entre o desenvolvedor e o JWSPerf, a ferramenta Java de construção Ant foi adotada. Dessa forma, o utilitário pode ser executado a partir de linha de comando, automatizando a construção de todo o código do cliente, incluindo as classes de teste e os artefatos específicos de cada *toolkit*.

Um desenvolvedor que domine as tecnologias Ant e XML, pode facilmente obter todas as informações necessárias para desenvolver uma aplicação cliente com JWSPerf apenas analisando o arquivo build.xml (ver Tabela 5.3), pois o mesmo define todos os comandos suportados. Além disso, o utilitário foi estruturado de forma simples, seguindo uma nomenclatura familiar aos desenvolvedores de aplicações.

Entretanto, para os desenvolvedores que desejem usar o utilitário JWSPerf como uma caixa preta, onde apenas é necessário invocar os comandos e analisar os resultados gerados, será apresentado um guia com os passos que devem ser realizados e quais

comandos devem ser invocados para executar corretamente o utilitário. Os passos apresentados a seguir apenas podem ser efetuados depois que o guia descrito na seção anterior tenha sido corretamente executado.

Passo 1: Rodar o arquivo env.bat

Primeiramente, antes de executar esse passo, o desenvolvedor deve abrir uma janela DOS e mudar para o diretório <JWSPERF_HOME>/jwsperf. Em seguida, é necessário rodar o arquivo de *batch* env.bat, para garantir que as variáveis de ambiente da máquina estejam corretamente configuradas. Esse arquivo precisa ser executado todas as vezes que uma nova janela DOS for aberta.

Passo 2: Alterar os arquivos parameters.properties e jwsperf.xml

É necessário verificar e alterar as configurações armazenadas no arquivo parameters.properties localizado no diretório “config/etc”, pois as mesmas são passadas como parâmetro ao utilitário. Durante a execução, os parâmetros – o número de *threads* clientes, o número de invocações para estabilizar os resultados, os números de avaliações, de ciclos e interação, a classe de teste e o diretório onde serão gerados os resultados – são armazenados na classe `Config` (ver Seção 5.3.2).

O arquivo jwsperf.xml contém as propriedades que instruem como o utilitário deve gerar as classes de teste e informam sobre o serviço remoto. Então, é necessário configurar suas propriedades de acordo com o cenário que se deseja avaliar. As principais propriedades que devem ser configuradas são o *toolkit*, a URI do WSDL que pode ser local ou remota, o endereço do serviço e a propriedade que habilita ou não a geração das classes de teste.

Passo 3: Construir as classes cliente

A construção das classes cliente consiste em invocar o comando “build” do *script* build.xml. Seu objetivo é gerar as classes necessárias para executar o utilitário do lado do cliente.

O comando “build” aciona outros comandos básicos, que executados em conjunto, realizam as tarefas necessárias para construir toda a infra-estrutura para o cliente invocar o serviço.

```
<target name="build"
        depends="prepare,generate-testcases,
        generate-axis,generate-jwsdp,
        generate-systinet,
        copy-files,compile">
```

Figura 5.8 - Comando para construir as classes clientes

O comando “build” é responsável pela invocação, nessa ordem, dos comandos “prepare”, “generate-testcases”, “generate-axis”, “generate-jwsdp”, “generate-systinet”, “copy-files” e “compile” (Figura 5.8), que também podem ser executados separadamente. As próximas subseções apresentam as tarefas realizadas por cada um desses comandos.

Tarefa 1: Preparando o diretório build

Essa tarefa consiste em invocar o comando “prepare” (Figura 5.9), que é responsável por limpar todo o conteúdo dentro do diretório “build” e, novamente, criar todos os seus subdiretórios – classes, src, generate e testcase.

Embora seja um comando simples, o mesmo é importante para garantir ao desenvolvedor que nenhum outro arquivo, gerado em execuções anteriores, seja utilizado erradamente.

```
<target name="prepare">
      <delete dir="${build.dir}" />
      <mkdir  dir="${build.dir}" />
      <mkdir  dir="${build.src.dir}" />
      <mkdir  dir="${build.classes.dir}" />
      <mkdir  dir="${build.generate.dir}" />
      <mkdir  dir="${build.testcase.dir}" />
</target>
```

Figura 5.9 - Comando para preparar o diretório build

Tarefa 2: Gerando as classes de teste

Essa tarefa consiste em invocar o comando “generate-testcases” para gerar as classes que serão usadas para invocar as operações do serviço *Web Services*. Esse comando foi definido para automatizar a execução do módulo de geração das classes de teste e apenas é executado quando o desenvolvedor habilita a propriedade “generate-testcase” do arquivo jwsperf.xml, que indiretamente torna a propriedade “generate.testcase.present” verdadeira.

```
<target name="generate-testcases" if="generate.testcase.present">
      <copy todir="${build.testcase.dir}">
         <fileset dir="${src.dir}/br/ufpe/cin/jwsperf/wsdl"/>
      </copy>

      <javac srcdir="${build.testcase.dir}"
             destdir="${build.testcase.dir}">
          <classpath refid="axis.classpath"/>
      </javac>

      <delete dir="${testcases.dir}/${toolkit}" />

      <java classname="br.ufpe.cin.jwsperf.wsdl.WSDLParser"
            fork="true" classpathref="axis.classpath">
            <classpath path="${build.testcase.dir}"/>
            <arg value="${wsdl.uri}"/>
            <arg value="${toolkit}"/>
            <arg value="${stub.package}"/>
            <arg value="${testcases.dir}/${toolkit}"/>
      </java>
</target>
```

Figura 5.10 - Comando para gerar as classes de teste

A Figura 5.10 ilustra as tarefas realizadas por esse comando. As mesmas consistem em copiar para o diretório “build/testcase” apenas os arquivos fontes que compõem o módulo de geração das classes de teste, a fim de serem compilados. Antes de executar esse módulo invocando a classe `WSDLParser`, as classes de teste geradas em execuções anteriores são apagadas.

O código resultante da execução desse comando depende da configuração da propriedade “toolkit” do arquivo jwsperf.xml. Além do *toolkit*, também são informados a localização do arquivo WSDL, o pacote das classes geradas e o diretório para armazená-las.

Tarefa 3: Construindo com o toolkit Axis

Essa tarefa consiste em invocar o comando “generate-axis”, porém o mesmo apenas é executado se o desenvolvedor tiver configurado a propriedade “toolkit” para usar o Axis. Caso contrário, essa tarefa não será executada, porque a propriedade “axis.toolkit” será falsa.

```
<target name="generate-axis" if="axis.toolkit">
    <java classname="org.apache.axis.wsdl.WSDL2Java"
          classpathref="axis.classpath" fork="true">
        <arg value="${wsdl.uri}"/>
        <arg value="-o${build.generate.dir}"/>
        <arg value="-p${stub.package}" />
    </java>
</target>
```

Figura 5.11 - Comando para gerar as classes usando o *toolkit* Axis

O objetivo é invocar a classe `WSDL2Java` do *toolkit* Axis para gerar as classes específicas desse *toolkit*. A partir da Figura 5.11, verifica-se que os parâmetros para executar essa classe são a localização do arquivo WSDL (“${wsdl.uri}”), o diretório raiz para armazenar as classes geradas (“${build.generate.dir}”) e o pacote que todos os *namespaces* do arquivo WSDL deverão ser mapeados (“${stub.package}”). O valor padrão das propriedades “${build.generate.dir}” e “${stub.package}” são “build\generate” e “`br.ufpe.cin.jwsperf.communication.ws`”, respectivamente.

Tarefa 4: Construindo com o toolkit JWSDP

De forma semelhante à anterior, essa tarefa apenas é executada quando o *toolkit* JWSDP estiver configurado. O objetivo do comando “generate-jwsdp” é gerar os artefatos do cliente específicos desse *toolkit* usando a sua ferramenta WSCompile.

```
<target name="generate-jwsdp" if="jwsdp.toolkit">
   <wscompile keep="true" client="true"
              base="${build.generate.dir}"
              config="${jwsdp.config.file}">
              <classpath>
                 <path refid="jwsdp.classpath"/>
              </classpath>
   </wscompile>

</target>
```

Figura 5.12 - Comando para gerar as classes usando o *toolkit* JWSDP

WSCompile é uma ferramenta simples e foi projetada para obter a localização da interface WSDL a partir do arquivo de configuração “${jwsdp.config.file}” (Figura 5.12). O nome padrão desse arquivo é config.xml e está localizado no diretório “config\xmls”. Além do arquivo WSDL, a ferramenta também lê desse arquivo o atributo “packageName”, que especifica o pacote das classes geradas e seu valor está configurdo na configurado na propriedade “${stub.package}”.

Analisando a Figura 5.12, verifica-se que o comando “wscompile” apenas deve gerar os artefatos do lado do cliente (cliente=“true”), manter os arquivos gerados (keep=“true”) e armazená-los no diretório indicado pelo parâmetro base=“${build.generate.dir}”.

Tarefa 5: Construindo com o toolkit SSJ

Quando o *toolkit* SSJ é configurado, o comando “generate-systinet” é executado e as classes do cliente SSJ são geradas usando a ferramenta de linha de comando WSDL2Java disponibilizada pelo próprio *toolkit*. Semelhantemente aos comandos anteriores, as classes cliente também são geradas a partir da interface WSDL indicada pelo parâmetro wsdlURL=“${wsdl.uri}” (Figura 5.13).

```
<target name="generate-systinet" if="systinet.toolkit">
     <WSDL2Java outputDirectory="${build.generate.dir}/"
                interfacePackage="${stub.package}"
                generateJavaBeans="true"
                force="true"
                strictSchema="true"
                wsdlURL="${wsdl.uri}"/>

</target>
```

Figura 5.13 - Comando para gerar as classes usando o *toolkit* SSJ

Além da interface WSDL, também são passados como parâmetro o diretório onde as classes serão geradas (outputDirectory=“${build.generate.dir}"), seu pacote (interfacePackage="${stub.package}") e as propriedades indicando que as classes devem ser sobrescritas a cada execução (force="true") e devem ser geradas seguindo o padrão JavaBeans (generateJavaBeans=“true”) e o esquema definido no arquivo WSDL (strictSchema="true").

Tarefa 6: Copiando as classes geradas

Essa tarefa consiste em executar o comando “copy-files” (Figura 5.14) que copia para o diretório “build/src” todas as classes que compõem o módulo de invocação, as classes de teste geradas a partir do comando “generate-testcases” e as classes clientes específicas dos *toolkits* geradas pelos comandos “generate-axis”, “generate-jwsdp” ou “generate-systinet”.

```
<target name="copy-files">
      <copy todir="${build.src.dir}">
         <fileset dir="${src.dir}" excludes="**/wsdl/**"/>
      </copy>
      <copy todir="${build.src.dir}">
         <fileset dir="${build.generate.dir}"/>
         <fileset dir="${testcases.dir}/${toolkit}"/>
      </copy>
      <copy todir="${build.classes.dir}">
         <fileset dir="${build.src.dir}"
                  includes="**/native, **/**.h, **/**.c,
                            **/**.xmap" description="">
            <exclude name="**/*.java"/>
         </fileset>
      </copy>
</target>
```

Figura 5.14 - Comando para copiar as classes geradas

Tarefa 7: Compilando as classes

O comando “compile” (Figura 5.15) compila todos os arquivos Java gerados e os arquivos fontes do próprio utilitário, ambos armazenados no diretório “build/src”. Os arquivos resultantes da compilação são armazenados no diretório “build/classes”. Esse passo pode ser executado separadamente, caso as classes de teste geradas sejam alteradas para montar os objetos que serão passados como parâmetro,

```
<target name="compile" description="Compile all classes">
      <javac srcdir="${build.src.dir}"
             destdir="${build.classes.dir}">
          <classpath refid="${toolkit}.classpath"/>
          <classpath refid="axis.classpath"/>
          <classpath path=".:${hrtlib.jar}"/>
      </javac>
</target>
```

Figura 5.15 - Comando para compilar todas as classes

Passo 4: Executar o utilitário JWSPerf

Para executar o utilitário, dois comandos podem ser invocados. O primeiro é o comando "run" (Figura 5.16) que simplesmente roda o utilitário usando as classes localizadas no diretório "build/classes". O segundo é comando "profile" (Figura 5.17) que roda e investiga o fluxo de execução do *toolkit* selecionado.

```
<target name="run" description="Run the client">
      <java classname="br.ufpe.cin.jwsperf.JWSPerfDriver"
        classpathref="${toolkit}.classpath" fork="true">
            <classpath path="${build.classes.dir}:${hrtlib.jar}"/>
            <arg value="${parameters.properties}"/>
            <arg value="${toolkit}"/>
            <arg value="${service.endpoint}"/>
            <arg value="${testcase.package}"/>
      </java>
</target>
```

Figura 5.16 - Comando para rodar o utilitário

Ambos os comandos invocam a classe `JWSPerfDriver`, passando como parâmetro o arquivo parameters.properties, o *toolkit* selecionado, o endereço do serviço remoto e o pacote onde estão localizadas as classes de teste cujo valor padrão é "`br.ufpe.cin.jwsperf.communication.ws.testcase`".

```
<target name="profile">
  <java
     classname="br.ufpe.cin.jwsperf.JWSPerfDriver"
     classpathref="${toolkit}.classpath" fork="true">
      <classpath path="${build.classes.dir}:${hrtlib.jar}"/>
      <arg value="${parameters.properties}"/>
      <arg value="${toolkit}"/>
      <arg value="${service.endpoint}"/>
      <arg value="${testcase.package}"/>
  </java>
  <java jar="${PerfAnal.jar}" fork="true">
     <arg value="${profile.file.name}"/>
  </java>
</target>
```

Figura 5.17 - Comando para investigar a execução do *toolkit*

Uma diferença entre esses comandos é que o "run" apenas gera um arquivo com os resultados no diretório "results", enquanto que o comando "profile", além de gerar esse arquivo, também gera um outro arquivo com resultado da investigação dentro do diretório "profile" e, em seguida, invoca a ferramenta Java de análise de desempenho

PerfAnal [Meyers, 2005], que lê esse último arquivo e apresenta uma tela com todo o fluxo de execução juntamente com o percentual gasto em cada método (Figura 5.18) , dividida em quatro visões: percentuais por métodos invocados (quadrante superior esquerdo), percentuais a partir dos invocadores do método (quadrante inferior esquerdo) e percentuais pelo número da linha do método (quadrantes do lado direito).

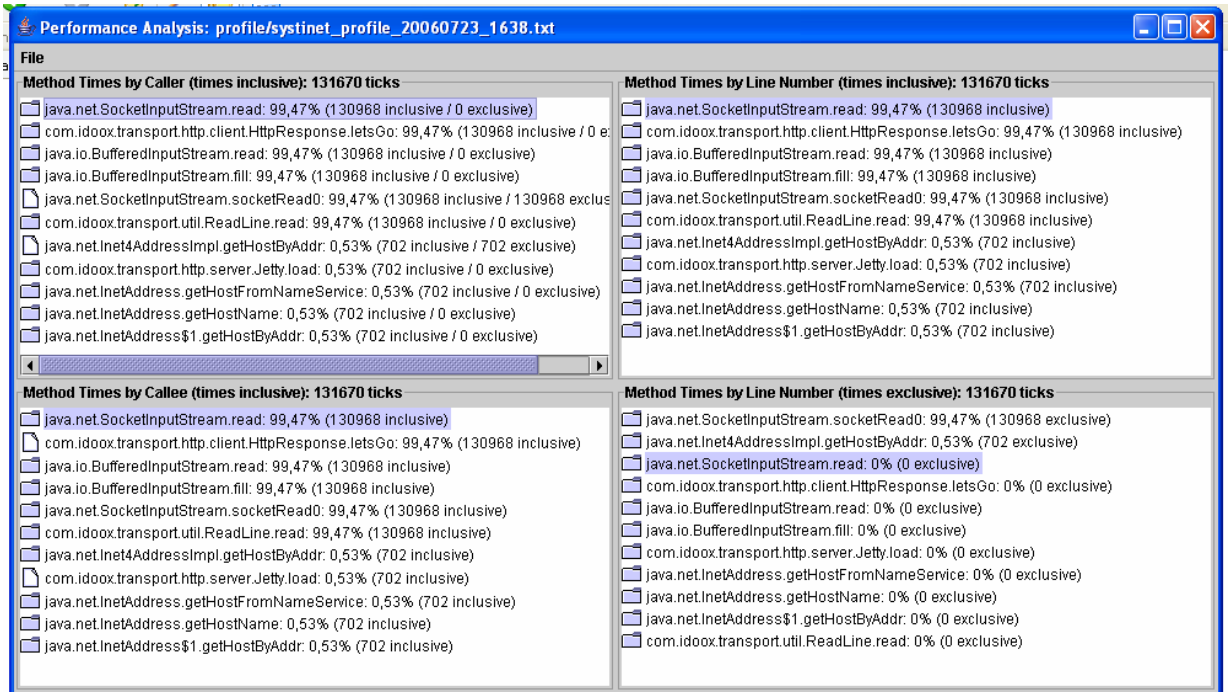

Figura 5.18 - Resultado da investigação do toolkit SSJ usando o PerfAnal

5.6 Guia para Incorporar Novos Toolkits

Nessa seção serão apresentados os passos que um desenvolvedor deveria fazer para incorporar novos *Web Services toolkits* ao utilitário JWSPerf. Os requisitos básicos que o novo *toolkit* deve atender são:

1. Suportar a geração das classes cliente a partir de um documento WSDL, localizado local ou remotamente;
2. Suportar o estilo de codificação *Document/Literal Wrapped*, atendendo à Diretriz 1.

Em relação ao código fonte existente do utilitário, nenhuma modificação é necessária. Apenas é preciso criar uma nova classe no pacote "`br.ufpe.cin.jwsperf.wsdl.toolkit`" e sua implementação deve conter todo o código específico do novo *toolkit*. Essa classe deve, obrigatoriamente, seguir o padrão de nome `<TOOLKIT>TestClassWriter`, onde `<TOOLKIT>` deve ser substituído pelo nome do *toolkit*, caso contrário, uma exceção será levantada. Com relação aos arquivos de configuração do utilitário, as seguintes alterações devem ser feitas para suportar o novo *toolkit*:

1. Após instalar o novo *toolkit* na máquina, o desenvolvedor deve alterar o arquivo env.bat, a fim de atualizar o *path* da máquina com as bibliotecas do *toolkit* instalado, caso seja necessário;
2. Incluir no arquivo build.properties a propriedade "`<toolkit>.home`", apontando para o diretório onde o *toolkit* foi instalado;
3. Definir no arquivo path.xml a propriedade "`<toolkit>.classpath`" que representará o *classpath* do novo *toolkit*. Essa propriedade deverá conter todas as bibliotecas necessárias para rodar o *toolkit*;
4. Criar a propriedade "`<nome>.toolkit`" no arquivo jwsperf.xml, que indicará se o novo *toolkit* foi selecionado pelo usuário;
5. Por fim, definir o comando "`generate-<toolkit>`" no arquivo build.xml que gerará as classes clientes específicas do *toolkit*, dentro do diretório representado pela propriedade "${build.generate}", cujo valor padrão é "build/genetarate".

5.7 Considerações Finais

Nesse capítulo foram apresentados um processo para avaliação de desempenho de *Web Services toolkits* e o utilitário JWSPerf (*Java Web Services Performance*) que visa automatizar os passos desse processo, referentes a parte de implementação, compilação e execução da aplicação cliente. Além disso, o utilitário também automatiza a coleta de métricas de desempenho.

O objetivo foi reduzir o tempo gasto no estudo da documentação dos *toolkits* e na implementação do código em si, uma vez que os desenvolvedores apenas necessitam configurar algumas propriedades nos arquivos de configuração e rodar os comandos para gerar e executar o utilitário. Do ponto de vista técnico, as principais características do utilitário são:

- Código aberto e implementado em Java;
- Utiliza a tecnologia Ant de Java para automatizar o processo de construção do código;
- Suporta três *Web Services toolkits* – Axis, *Java Web Services Developer Pack* (JWSDP) e *Systinet Server for Java* (SSJ);
- Fácil de usar, sendo apenas necessário invocar os comandos definidos e analisar os resultados gerados;
- Gera automaticamente todo o código cliente específico de um *Web Services toolkit* a partir de um arquivo WSDL, garantindo assim a interoperabilidade com o serviço;
- Coleta as métricas de desempenho referente à execução;
- Flexibilidade para incorporar novos *toolkits*;
- Composto por dois módulos com responsabilidades bem definidas: o módulo de geração das classes de teste e o módulo de invocação.

O módulo de geração das classes de teste facilita a aplicação da Diretriz 2, pois para cada operação definida na interface WSDL do serviço, uma classe de teste invocando essa operação é criada e sua implementação é específica do *toolkit* configurado.

O módulo de invocação recupera a instância do *stub* e executa a operação desejada pelo desenvolvedor, gerando as métricas de desempenho no final da execução. O módulo de invocação foi projetado para automatizar a Diretriz 6.

Todo o funcionamento desses módulos baseia-se nas propriedades dos seus arquivos de configuração, que precisam ser alteradas antes de invocar os comandos do utilitário. Com essa estruturação, quando um novo *toolkit* implementado em Java for incorporado, basta definir algumas propriedades e criar uma única classe.

Para realizar a avaliação de desempenho de um determinado serviço *Web Service* utilizando o JWSPerf, os seguintes passos do lado do servidor (Passo 1 e 2) e do cliente (Passos 3 a 10) devem ser realizados:

Passos	Execução
1. Construir a interface WSDL do serviço e prover sua implementação.	Manual
2. Gerar a camada de comunicação Web Services do serviço para os toolkits Axis, JWSDP e SSJ.	Usar os comandos que os *toolkits* disponibilizam para automatizar esse passo.
3. Instalar os toolkits Axis, JWSDP e SSJ e configurar o arquivo build.properties.	Manual
4. Instalar o utilitário JWSPerf.	Manual
5. Configurar e executar o arquivo env.bat	Manual
6. Configurar os arquivos parameters.properties e jswperf.xml.	Manual
7. Construir as classes de teste	Automatizado pelo JWSPerf
8. Alterar as classes testes para montar os objetos que serão passados como parâmetros. Esse passo é opcional.	Manual
9. Executar o utilitário JWSPerf	Automatizado pelo JWSPerf
10. Analisar os resultados	Manual

De forma geral, JWSPerf reduz o custo para desenvolver uma aplicação, uma vez que todo o código, incluindo as classes de teste, é gerado pelo mesmo, sem ônus para o desenvolvedor. Além disso, pode-se facilmente avaliar o desempenho da mesma aplicação utilizando três diferentes *Web Services toolkits*, permitindo a comparação e identificação dos seus gargalos de desempenho.

No próximo capítulo serão apresentados os experimentos realizados utilizando JWSPerf para analisar o desempenho dos *toolkits* suportados, aplicando as diretrizes e o processo propostos e utilizando uma aplicação-teste com operações bem definidas como *benchmark*.

6 Plataforma Experimental e Resultados

6.1 Introdução

Esse capítulo apresenta os resultados dos experimentos que foram executados seguindo o processo de avaliação e utilizando o utilitário JWSPerf. Uma aplicação-teste foi projetada e utilizada como *benchmark* para avaliar o desempenho dos três *Web Services toolkits* suportados pelo utilitário.

Segundo Buble (2003), para que um processo de avaliação obtenha resultados estáveis, é necessário executar algumas invocações à operação sendo avaliada antes de iniciar a coleta dos tempos. Esse passo é importante para minimizar ou eliminar a influência de fatores que podem tornar os resultados incorretos.

Além do tempo para estabilizar o resultado (*warm-up*), outra causa de erro na coleta de resultados é a medição imprecisa do tempo, pois a grande maioria dos *benchmarks* envolve a medição do tempo. Dessa forma é importante usar uma fonte precisa de medição, pois uma invocação remota pode gastar menos que 10ms.

No decorrer desse capítulo, será verificado que os experimentos realizados levaram em consideração esses dois pontos. Antes de iniciar a coleta, 3000 requisições

foram feitas a cada operação analisada e uma biblioteca de tempo precisa foi utilizada, conforme descrito a seguir.

Antes de apresentar os resultados, as próximas seções detalham a aplicação-teste, explicando seu projeto e os métodos da sua interface, e as configurações de *software* e *hardware* da plataforma experimental de avaliação (ver Seção 6.2.2).

6.2 Aplicação-teste

As operações suportadas pela aplicação-teste foram definidas baseando-se nos *benchmarks* definidos por Juric et al. (1999) e Slominski et al. (2005). A Figura 6.1 ilustra a interface `IWSBenchmark` definida com o objetivo de permitir que os resultados obtidos através do processo de avaliação sejam comparáveis entre si.

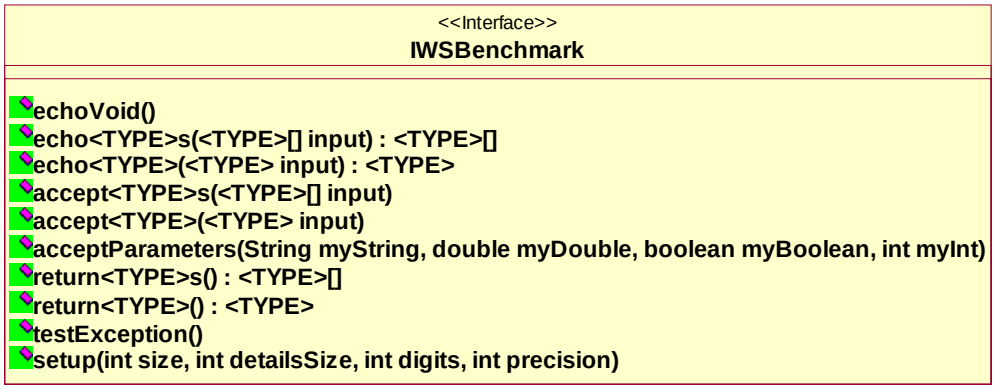

Figura 6.1 - Métodos da interface IWSBenchmark

De acordo com a Diretriz 2, a avaliação deveria analisar mensagens de diferentes tamanhos e complexidades, a fim de entender como diferentes tipos de dados utilizados como parâmetros e valores de retorno dos métodos influenciam no desempenho de uma aplicação *Web Services*. Dessa forma, a interface `IWSBenchmark` utiliza tanto tipos de dados simples – `double`, `int`, `boolean` e `string` – quanto tipos de dados complexos definidos pelo usuário.

O diagrama de classe da Figura 6.2 mostra os tipos de dados definidos pelo usuário, empregados pela interface `IWSBenchmark` – `MySimpleObject`,

`MyComplexObject` e `MyException`. A classe `MySimpleObject` representa um objeto que encapsula apenas os tipos de dados simples. A classe `MyComplexObject`, além de encapsular os tipos de dados simples, tem um *array* de objetos do tipo `MySimpleObject`, cuja quantidade de elementos é dinâmica.

A classe `MyExcpetion` representa uma exceção, uma vez que herda da classe `java.lang.Exception`. Essa classe encapsula apenas uma `string` representando a mensagem do erro ocorrido.

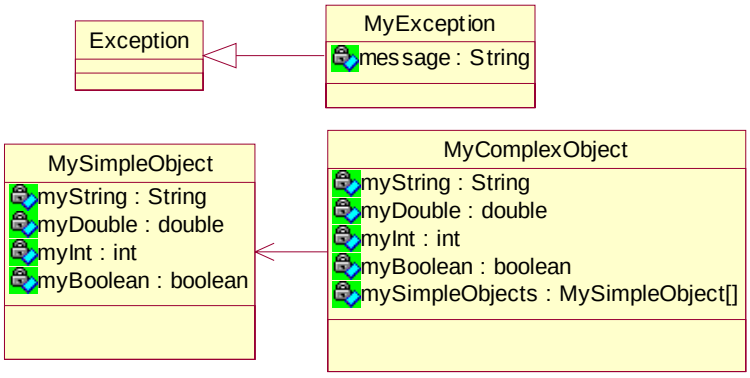

Figura 6.2 - Entidades de negócio definidas pelo usuário

Para cada tipo de dado simples ou complexo, a interface `IWSBenchmark` define um método que passa como parâmetro e/ou retorna o tipo de dado. Além do tipo de dado em si, também foram definidos métodos usando um *array* do tipo de dado, a fim de aumentar a complexidade e analisar os custos da serialização e deserialização desses dados. Dessa forma, a interface `IWSBenchmark` é constituída basicamente por quatro categorias de métodos:

- echo<**TYPE**>**:** essa categoria de método recebe um argumento de um determinado tipo e retorna um valor do mesmo tipo. Exemplos:
  - `public int echoInt(int input);`
  - `public MySimpleObject echoMySimpleObject(MySimpleObject input).`
- accept<**TYPE**>: essa categoria recebe um único argumento de um determinado tipo e não retorna nenhum valor. Exemplos:
  - `public void acceptInt(int input);`
  - `public void acceptMySimpleObject(MySimpleObject input).`

- return<**TYPE**>: essa categoria não recebe nenhum parâmetro e retorna apenas o tipo de dado. Exemplos:
  - `public int returnInt();`
  - `public MySimpleObject returnMySimpleObject().`
- echo<**TYPE**>s, accept<**TYPE**>s e return<**TYPE**>s: essa categoria é similar às descritas acima, exceto que utiliza um *array* do tipo de dado. Exemplos:
  - `public int[] echoInts(int[] input);`
  - `public void acceptInts(int[] input);`
  - `public int[] returnInts().`

Para complementar essas categorias, outros métodos auxiliares foram definidos, pois são importantes para a avaliação de desempenho:

- `echoVoid`: esse método é utilizado para determinar o gargalo associado com uma chamada SOAP imposta pelo *toolkit*;
- `acceptParameters`: esse método recebe quatro parâmetros do tipo `string`, `double`, `boolean` e `int`;
- `testException`: esse método simplesmente levanta uma exceção definida pelo usuário.

O método `setup` foi definido com o objetivo de configurar o tamanho em *bytes* da `string` ou a quantidade de dígitos de um número do tipo `double`. Ele também foi usado para configurar a quantidade de elementos do *array* retornado pelos métodos das categorias `echo` e `return`.

6.2.1 Projeto da Aplicação-teste

A aplicação-teste foi projetada em camadas (ver Figura 6.3) com o objetivo de atender aos requisitos de modularidade e reusabilidade. Dessa forma, foi possível reusar a mesma implementação da aplicação-teste, alterando apenas os componentes responsáveis pela comunicação que dependem do *Web Services toolkit* escolhido para expor o serviço.

A camada de comunicação trata as requisições dos clientes e as encaminha para a classe `Controller` da camada de negócio. O componente `ServiceAdapter` contém

uma instância da classe `Controller` e é responsável pelo processo de adaptação das requisições antes de encaminhá-las.

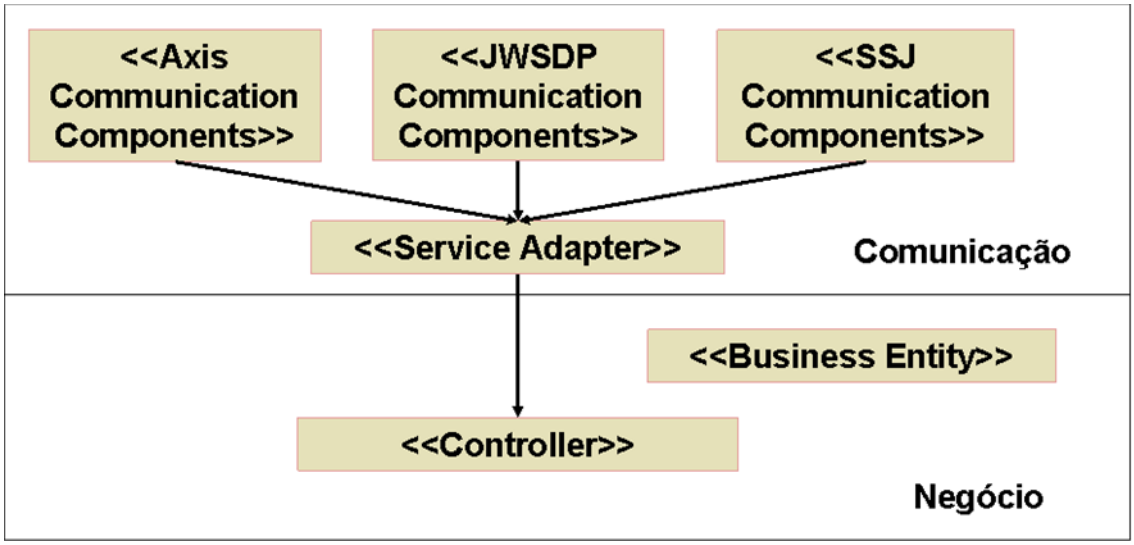

Figura 6.3 - Arquitetura da aplicação-teste

A camada de negócio é constituída pela classe `Controller`, que representa o ponto único de acesso da aplicação, e pelas entidades de negócio que são as entidades que representam objetos do mundo real e agrupam operações. A classe `Controller` foi implementada usando o padrão de projeto *Singleton*. As entidades de negócio fornecem as informações necessárias aos dados armazenados pela aplicação. No caso dessa aplicação-teste, as entidades de negócio estão definidas na Figura 6.2.

Para evitar que o tempo de acesso ao banco influenciasse o processo de avaliação de desempenho da camada de comunicação implementada em *Web Services*, a persistência dos dados foi simplesmente omitida.

A fim de garantir a interoperabilidade entre os *toolkits*, a interface WSDL foi projetada usando o estilo *Document/Literal Wrapped*. Uma vez definida a interface, a mesma foi utilizada para gerar o *skeleton* do lado do servidor, exceto para o *toolkit* SSJ que não suportou a geração do código a partir da interface WSDL, mas a partir do código Java. A ferramenta Ant foi utilizada para automatizar esse processo de construção da camada de comunicação *Web Services*. As seguintes regras de interoperabilidade também foram consideradas:

- Cada método deveria ter um nome de operação diferente. Dessa forma, não foi usado o *overloading* de métodos;
- Não usar o tipo de dado `char`, porque o mesmo não é suportado pela tecnologia XML *Schema*;
- Não expor as coleções de objetos usando tipos específicos da linguagem, assim como `Collection`, `Map`, `List` e `Hashtable`, porque não existe uma padronização entre os *toolkits* para enviar esses tipos de dados. Converter todos para *array*.

6.2.2 Configurações do Ambiente de Execução

Os códigos do cliente e do servidor foram implementados em Java e foram compilados e executados usando o *Java™ 2 SDK Standard Edition* versão 1.4.2_08 da Sun para o sistema operacional Windows da Microsoft.

Os *toolkits* Java avaliados nessa dissertação foram Axis versão 1.2 RC2 da Apache, *Web Services Developer Pack* (JWSDP) versão 1.5 da Sun e *Systinet Server for Java* (SSJ) versão 5.5. O Tomcat versão 5.0 foi utilizado como servidor de aplicação para os *toolkits* Axis e JWSDP, enquanto que o SSJ rodou no servidor de aplicação embutido na sua implementação.

A biblioteca de tempo HRTLib [Roubtsov, 2004] foi escolhida porque a resolução do método Java `System.currentTimeMillis()` não é ideal para investigar com precisão a execução do código Java. Essa biblioteca é simples e emprega a tecnologia JNI (*Java Native Interface*) para retornar o tempo transcorrido em milisegundos. Essa biblioteca foi escolhida para se ter uma fonte precisa de medição, embora esteja implementada apenas para rodar no sistema operacional Windows.

O tráfego TCP/IP entre o cliente e o servidor foi recuperado usando a ferramenta Ethereal versão 0.10.10 [Ethereal, 2004], que é um analisador do protocolo de transporte e executa em todas as plataformas populares como Unix, Linux e Windows.

As mensagens HTTP transportando chamadas SOAP foram recuperadas usando a ferramenta TCP Monitor [Apache Axis, 2004]. Dessa forma, pode-se monitorar as mensagens enquanto a aplicação é executada.

Os experimentos foram realizados usando dois computadores rodando o sistema operacional Windows XP da Microsoft Versão 2002 Service Pack 2 e conectados por uma rede *Ethernet* de 100Mbps exclusiva para os experimentos. A máquina servidora foi um Pentium 4 da Intel com o processador Pentium(R) 4 CPU de 2.80 GHz e 1 GB de memória RAM. A máquina cliente também foi um Pentium(R) 4 CPU da Intel, porém com um processador de 2.40 GHz e 2 GB de memória RAM.

6.3 Resultados da Avaliação de Desempenho

Essa seção apresenta os resultados da avaliação de desempenho seguindo o processo proposto e utilizando o utilitário JWSPerf. Os resultados apresentados a seguir representam uma média dos resultados obtidos na execução de 10 avaliações, onde cada avaliação executou 3000 invocações para estabilizar os resultados e 20 ciclos, com cada ciclo executando 500 invocações à operação sendo analisada. Dessa forma, foram executadas 13000 invocações para cada avaliação, sendo 3000 sem coleta de tempo e 10000 (20 x 500) com coleta.

Esse cenário de teste é facilmente configurado usando o utilitário JWSPerf, pois basta inicializar os parâmetros `WARM_UP`, `EVALUATIONS`, `CYCLES` e `INTERACTIONS` do arquivo de propriedades parameters.properties (ver Figura 6.4) com os valores 3000, 10, 20 e 500, respectivamente.

A seguir, serão detalhadas as tarefas do processo proposto, explicando o passo a passo que deve ser executado do lado do cliente.

```
################
# Parameters properties
#
# Developed by Ana Machado (accm2@cin.ufpe.br).
################

#Number of concurrent clients
AGENT_NUMBER=1

#Number of evaluation
EVALUATIONS=10

#Number of cycles
CYCLES=20

#Number of interactions
INTERACTIONS=500

#Number of invocations before the timing information is collected
WARM_UP=3000

#Testcase name
TESTCASE=EchoBooleanTestCase

#Output directory where the results are generated
RESULT_FILE_PATH=.\\results\\vazao
```

Figura 6.4 – Arquivo parameters.properties

Tarefa 1: Recuperar a interface WSDL

Essa tarefa deve ser o ponto de partida, pois a interface WSDL é o contrato entre as aplicações – essa é a primeira lei da interoperabilidade. Utilizando o utilitário JWSPerf, esse passo é executado configurando os parâmetros do arquivo de propriedade jwsperf.xml.

A interface WSDL utilizada nessa tarefa foi a mesma que foi utilizada para desenvolver a aplicação-teste, dessa forma todas as operações da interface `IWSBenchmark` são suportadas (ver Figura 6.1).

Tarefa 2: Escolher o Web Services toolkit

Os *toolkits* avaliados nos experimentos foram os suportados pelo utilitário JWSPerf – Axis, JWSDP e SSJ. Para checar se esses *toolkits* suportam o estilo *Document/Literal Wrapped*, foi necessário estudar suas documentações. Mesmo não sendo o estilo padrão, todos suportam o estilo *Document/Literal Wrapped*.

As versões anteriores do Axis não suportavam o estilo *Document/Literal Wrapped*, porém as versões mais recentes estão sendo implementadas suportando esse estilo para melhor atender aos requisitos de interoperabilidade [Apache Axis, 2004].

Usando o utilitário JWSPerf, essa tarefa é executada configurando o parâmetro "toolkit" do arquivo de propriedade jwsperf.xml. Uma vez configurado o *toolkit* e a interface WSDL, o utilitário é capaz de gerar o código da aplicação cliente implementando todas as operações do serviço.

Tarefa 3: Verificar o parser do toolkit

Analisando a documentação dos *toolkits* Axis e SSJ, verificou-se que ambos adotam o *parser* Xerces, que é uma implementação do modelo de *parsing* SAX. Embora a equipe de desenvolvimento do Axis recomende fortemente o uso desse *parser*, o Axis foi desenvolvido para suportar qualquer implementação compatível com a especificação JAXP 1.1 (*Java API for XML Processing*) como, por exemplo, o *parser* Crimson que é mais eficiente [Elfwing et al., 2002]. A versão adotada pelo Axis 1.2 RC2 foi Xerces 2.4.0, enquanto que o *toolkit* SSJ 5.5 adotou a versão Xerces 2.6.2.

O *toolkit* JWSDP 1.5 adota o *parser* SJSXP (*Sun Java Streaming XML Parser*) versão 1.0 EA [Sun, 2004], que é uma eficiente implementação da API Java padrão do modelo de *parsing Pull Parsing* chamada StAX (*Streaming API for XML Parser*). SJSXP é um *parser* simples de usar e permite a leitura e escrita de documentos XML de forma eficiente.

Tarefa 4: Gerar o Stub

O objetivo dessa tarefa é gerar o *stub*, que são classes específicas dos *toolkits* para abstrair detalhes da comunicação, usando as ferramentas disponibilizadas pelos mesmos. Para isso é necessário estudar sua documentação, pois cada ferramenta utiliza diferentes configurações para executar. Para eliminar esse ônus, o utilitário disponibiliza comandos para automatizar essa tarefa. A Tabela 6.1 apresenta para cada *toolkit*

analisado, a ferramenta e o comando do utilitário responsável pela execução dessa tarefa.

Tabela 6.1 - Ferramentas dos *toolkits*

Toolkit	**Ferramenta**	**Comando do JWSPerf**
Axis	WSDL2Java	generate-axis
JWSDP	WSCompile	generate-jwsdp
SSJ	WSDL2Java	generate-systinet

Embora possuam o mesmo nome, as ferramentas dos *toolkits* SSJ e Axis são implementações completamente distintas. Uma semelhança entre essas ferramentas, é que obrigatoriamente todas recebem como parâmetro o endereço local ou remoto da interface WSDL. Em cada avaliação realizada com o utilitário, apenas um desses comandos é executado, pois apenas um *toolkit* pode ser configurado.

Tarefa 5: Implementar a aplicação cliente

Nesse experimento, a aplicação cliente deve implementar as invocações às operações definidas na interface da aplicação-teste e o código de cada operação deve acessar o *stub* para invocar o serviço, porém a forma de instanciar o *stub* varia de acordo com o *toolkit*.

Segundo a Diretriz 2, a interface deveria conter operações que explorem tipos de dados de diferentes complexidades e tamanhos. Dessa forma, seria muito custoso desenvolver todas as operações definidas para os diferentes *toolkits*. Além disso, também deve ser escrito o código responsável pela coleta do tempo.

Com o objetivo de simplificar essa tarefa, foi projetado o módulo de geração de classes de teste (ver Seção 5.3.1) que é responsável pela geração de uma classe de teste para cada operação da interface WSDL. O código das classes geradas é específico do *toolkit* selecionado. Além desse módulo, o utilitário disponibiliza classes responsáveis pela coleta do tempo e geração de um arquivo com os resultados da execução. Dessa forma, não é necessário gastar tempo analisando documentação e implementando o código da aplicação cliente para todas as operações definidas.

O utilitário JWSPerf disponibiliza o comando “generate-testcases” para automatizar essa tarefa, considerando que o *toolkit* e a interface WSDL estão configurados. Na prática, o desenvolvedor invoca apenas o comando “buid” (ver Seção 5.5), pois o mesmo executa o comando “generate-testcases” e os comandos para executar a tarefa anterior, além de estruturar e compilar todo o código para a execução da próxima tarefa.

Tarefa 6: Invocar as operações do serviço

O objetivo dessa tarefa é invocar as operações do serviço e coletar suas métricas. Durante o processo de avaliação, apenas uma versão da implementação da aplicação-teste para um *toolkit* estava rodando no servidor de aplicação. Para medir apenas o tempo gasto na camada *Web Services*, qualquer processamento na camada de negócio foi evitado.

Para executar essa tarefa, foi projetado o módulo de invocação do utilitário (ver Seção 5.3.2) que é responsável pela invocação e coleta das métricas no final da execução. O desenvolvedor precisa apenas configurar a operação que deseja analisar e executar o comando “run” para o utilitário invocar a operação.

Primeiramente, foram avaliados, para cada *toolkit*, o tempo de instanciação do *stub*, a latência representada pelo método `echoVoid` e o RTT para enviar e receber mensagens compostas por tipos de dados simples e pelos definidos pelo usuário (ver Tabela 6.2).

Os *toolkits* Axis e JWSDP apresentaram um tempo de instanciação do *stub* menor que o SSJ porque o *stub* desses *toolkits* foi gerado estaticamente (*early binding)*, enquanto que o SSJ instancia o *stub* dinamicamente (*late binding)*.

Analisando o método `echoVoid`, a latência do Axis foi o dobro da latência dos outros *toolkits*, indicando que o mesmo tem problemas na sua implementação, independentemente da carga, uma vez que esse método não envia nenhum tipo de dado.

Os métodos `echoInt`, `echoDouble`, `echoBoolean`, e `echoString` foram analisados para medir o tempo para serializar, transmitir e deserializar tipos de dados simples (`double`, `int`, `boolean` e `string`), inicializados com seu valor *default*. Para um

determinado *toolkit*, o RTT calculado foi praticamente igual ao tempo do seu método `echoVoid` (Tabela 6.2). O *toolkit* Axis apresentou o maior tempo de processamento.

Tabela 6.2 - Tempos (ms) de instanciação do *stub* e dos métodos simples

Operações	JWSDP	SSJ	Axis
Instanciação do *stub*	3200	22100	4900
echoVoid	3	3	6
echoInt	5	4	7
echoDouble	5	4	7
echoBoolean	5	4	7
echoString	4	4	7
echoMySimpleObject	4	5	7
echoMyComplexObject	5	10	14
acceptParameters	5	4	7
acceptMySimpleObject	3	5	6

Depois de calcular o RTT para cada tipo básico separadamente, foi analisado o impacto para enviar todos esses tipos de dados como parâmetro para o servidor, invocando o método `acceptParameters`. Nenhum impacto foi detectando, pois todos os *toolkits* mantiveram os mesmos tempos.

A fim de comparar o tempo para enviar, simultaneamente, os dados como parâmetros (`acceptParameters`) com o tempo para enviar os mesmos dados encapsulados em um objeto, o método `acceptMySimpleObject` foi analisado (ver Tabela 6.2). Como os tempos desses métodos foram semelhantes, fica a critério do projetista da interface a escolha de encapsular ou não os dados transmitidos.

Com o objetivo de investigar o gargalo para enviar um dado complexo definido pelo usuário ao servidor, o tempo para invocar o método `echoMyComplexObject` foi calculado, onde o objeto `MyComplexObject` agregava vinte objetos do tipo `MySimpleObject`. O *toolkit* JWSDP não sofreu impacto e manteve praticamente o mesmo tempo. Porém, o aumento da complexidade dobrou o tempo de resposta dos *toolkits* Axis e SSJ, comparado ao método `echoMySimpleObject`.

Para entender as razões das diferenças de desempenho entre os *toolkits* analisados, a execução das operações `echoVoid` e `echoMyComplexObject` foi investigada para identificar os métodos que consomem a maior parte do tempo. Nesse experimento foi usado o comando “profile” do utilitário, que implicitamente utiliza a ferramenta PerfAnal (ver Seção 5.5).

Investigando a execução do método `echoVoid`, o Axis gasta 69,38% e 30,58% do seu tempo executando os métodos `connect` da classe `java.net.Socket` e `read` da classe `java.net.SocketInputStream`, respectivamente. Quando o método `echoMyComplexObject` foi investigado, esses percentuais foram alterados para 65,67% e 34,33%, respectivamente. Para o *toolkit* SSJ, 99,47% e 85,77% do tempo são gastos executando o método `read` da classe `java.net.SocketInputStream`, quando as operações `echoVoid` e `echoMyComplexObject` são executadas, respectivamente. A partir desses resultados, conclui-se que a maior parte do tempo é gasto na conexão entre o cliente e o servidor.

Analisando o *toolkit* JWSDP, 99,38% do seu tempo de execução foram gastos executando o método `read` da classe `java.net.SocketInputStream`, quando as operações `echoVoid` e `echoMyComplexObject` são invocadas. A partir desse resultado, pode-se concluir que esse *toolkit* possui rotinas eficientes de serialização e deserialização, mesmo aumentando a complexidade das mensagens transportadas.

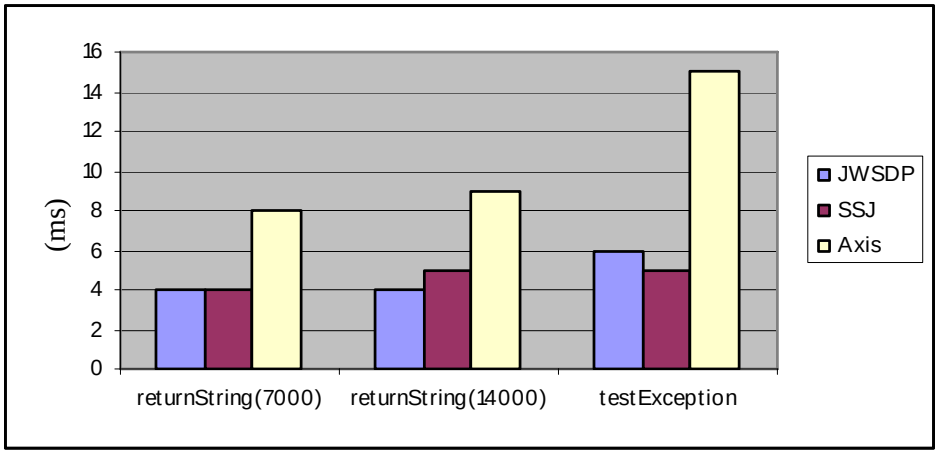

Figura 6.5 - RTT (ms) dos métodos testException e returnString

A classe `MyException`, herdando de `java.lang.Exception`, foi definida com o objetivo de calcular o tempo para tratar uma exceção definida pelo usuário. Analisando o método `testException` (Figura 6.5), verificou-se que o Axis gasta aproximadamente 15ms para tratar um elemento `<soap:fault>` da mensagem SOAP, enquanto que, os *toolkits* JWSDP e SSJ gastam aproximadamente 5ms.

A fim de investigar o impacto do tamanho da mensagem no tempo de resposta, o método `returnString` foi configurado, primeiramente, para retornar uma `string` de

7000 *bytes* e depois, uma `string` de 14000 *bytes*. Em ambos os experimentos a mensagem SOAP tinha os mesmos elementos, variando apenas o tamanho da mensagem transportada. A Figura 6.5 ilustra o tempo para transferir os dados na rede invocando esse método.

A partir da análise do tráfego de pacotes, verificou-se que o Axis utiliza 6 e 10 pacotes de rede para enviar uma `string` de 7000 e 14000 *bytes*, respectivamente, enquanto que os *toolkits* JWSDP e SSJ utilizam 7 e 12 pacotes. Mesmo utilizando menos pacotes de rede, em ambos os casos, o tempo de resposta do *toolkit* Axis foi maior, significando que o problema está nas suas rotinas de (de)serialização. O *toolkit* JWSDP manteve exatamente o mesmo tempo para executar as duas operações.

O impacto do processo de serialização e deserialização foi investigado invocando os métodos `returnDoubles` e `returnMyComplexObjects` (Figura 6.6). O primeiro método foi configurado para retornar um *array* com 500 números do tipo `double` com quatro dígitos de precisão. O método `returnMyComplexObjects` foi configurado para retornar um *array* com cinqüenta objetos do tipo `MyComplexObject`, cada um contendo vinte objetos do tipo `MySimpleObject`.

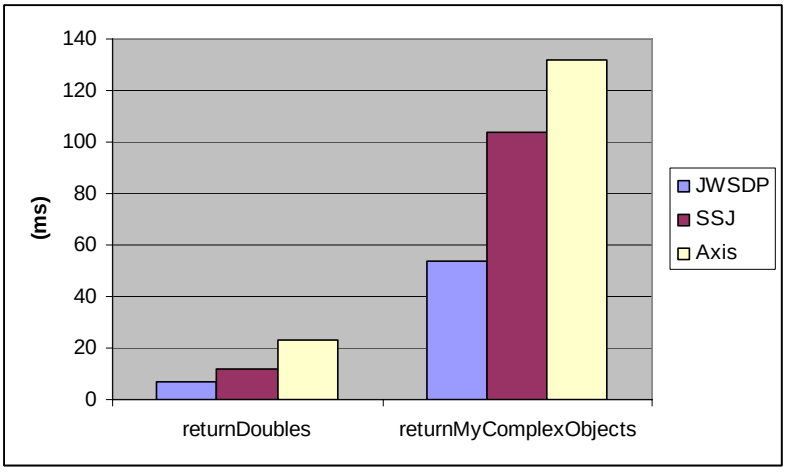

Figura 6.6 – RTT dos métodos returnDoubles e returnMyComplexObjects

A análise desses métodos foi importante para validar o impacto da complexidade dos tipos de dados no tempo de resposta, pois se verificou a ineficiência dos *toolkits* Axis e SSJ com relação ao *toolkit* JWSDP. Em relação aos resultados apresentados na Tabela 6.2, os *toolkits* SSJ e JWSDP apresentaram praticamente o mesmo desempenho,

porém comparando o resultado do método `returnMyComplexObjects`, os *toolkits* SSJ e Axis foram, respectivamente, 19,25% e 24,45% mais ineficientes que o JWSDP. Os custos de serialização e deserialização do *toolkit* Axis são maiores que os custos dos demais *toolkits* analisados.

Se apenas tipos de dados simples fossem utilizados no processo de avaliação, concluir-se-ia que os *toolkits* JWSDP e SSJ apresentam praticamente o mesmo tempo de resposta. Porém, segundo a Diretriz 2, é importante analisar métodos usando tipos de dados complexos para verificar o comportamento de um *toolkit* em situações críticas.

Um experimento aumentando o número de clientes concorrentes foi realizado para avaliar a degradação de desempenho dos *toolkits*. O JWSPerf foi projetado para criar uma nova *thread* para cada novo cliente, permitindo assim a execução concorrente pois as *threads* são independentes umas das outras. Entretanto, se a quantidade de clientes rodando numa mesma máquina for muito grande, a aplicação pode esgotar a utilização de recursos do sistema.

Nesse experimento, foi invocado o método `returnMyComplexObjects` que retorna, no caso do *toolkit* Axis, no mínimo 183 KB (ver Tabela 6.3, pp. 119). O resultado obtido foi preocupante porque os *toolkits* saturaram a utilização de seus recursos com apenas cinco clientes rodando simultaneamente. Um fator que contribuiu para esse resultado foi que todas as *threads* rodavam em uma única máquina, portanto as mesmas disputavam pelos mesmos recursos do computador como memória e processador. Para evitar problemas dessa natureza, o uso de recursos deve ser controlado e a carga deve ser distribuída em vários computadores, para realmente produzir carga intensa.

Tarefa 7: Monitorar as mensagens SOAP

Para monitorar as mensagens SOAP trocadas entre o servidor e o cliente, a ferramenta TCP Monitor foi utilizada. Os objetivos dessa tarefa foram calcular o tamanho da requisição e da resposta em *bytes* de uma determinada operação e analisar o cabeçalho HTTP, a fim de estudar as configurações adotadas por cada *toolkit*.

```
POST /axis/server/WSBenchmark?WSDL HTTP/1.0
Content-Type: text/xml; charset=utf-8
Accept: application/soap+xml, application/dime, multipart/related, text/*
User-Agent: Axis/1.2RC2
Host: 127.0.0.1
Content-Length: 325

<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <echoString xmlns="http://ws.communication.benchmark">
      <input xmlns="">WSBenchmark</input>
    </echoString>
  </soapenv:Body>
</soapenv:Envelope>
```

Figura 6.7 - Requisição SOAP/HTTP gerada pelo *toolkit* Axis

As Figuras 6.7, 6.9 e 6.11 representam a requisição do método `echoString` gerada pelos *toolkits* Axis, JWSDP e SSJ, respectivamente. As Figuras 6.8, 6.10 e 6.12 representam a resposta dessa operação para cada um dos *toolkits*. Comparando essas figuras, observa-se que para uma mesma operação e estilo de codificação (*Document/Literal Wrapped*), os *toolkits* geram mensagens diferentes, inclusive de tamanhos diferentes.

```
HTTP/1.1 200 OK
Content-Type: text/xml;charset=utf-8
Date: Sat, 22 Jul 2006 18:22:56 GMT
Server: Apache-Coyote/1.1
Connection: close

<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <echoStringResponse xmlns="http://ws.communication.benchmark">
      <result xmlns="">WSBenchmark</result>
    </echoStringResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Figura 6.8- Resposta SOAP/HTTP gerada pelo *toolkit* Axis

Analisando as Figuras 6.7 e 6.8, as seguintes características do *toolkit* Axis podem ser identificadas: a) o cliente usa o protocolo HTTP 1.0 para enviar a requisição,

enquanto que o serviço utiliza o protocolo HTTP 1.1 para enviar a resposta; b) o servidor sempre fecha a conexão, pois o atributo “Connection: close” é enviado; c) o atributo “Content-Length” da requisição informa o tamanho da mensagem SOAP serializada.

Diferentemente do *toolkit* Axis, as características do *toolkit* JWSDP são (ver Figuras 6.9 e 6.10): 1) adotar o protocolo HTTP 1.1 para transportar tanto a requisição e a resposta; 2) a fim de obter um melhor desempenho, o *toolkit* JWSDP configura o atributo “Connection: keep-alive” no cabeçalho da requisição, habilitando o uso de conexões persistentes; e 3) O atributo “Transfer-Encoding: chunked” no cabeçalho da resposta do servidor informa ao cliente que a resposta será dividida em vários blocos (*chunks*), onde cada bloco é precedido pelo seu tamanho (Figura 6.10).

```
POST /jwsdp/server/WSBenchmark HTTP/1.1
Content-Type: text/xml; charset=utf-8
Content-Length: 439
SOAPAction: ""
User-Agent: Java/1.4.2_08
Host: 192.168.1.1:8080
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive

<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
  xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:ns0="http://service.benchmark"
  xmlns:ns1="http://ws.communication.benchmark">
  <env:Body>
    <ns1:echoString>
      <input>WSBenchmark</input>
    </ns1:echoString>
  </env:Body>
</env:Envelope>
```

Figura 6.9 - Requisição SOAP/HTTP gerada pelo *toolkit* JWSDP

Aplicando a técnica de *streaming Chunked Transfer Coding*, o tamanho de cada bloco é informando na mensagem usando a notação hexadecimal. Por exemplo, a resposta da operação `echoString` retornou apenas 1 bloco de tamanho, em hexadecimal, 1c9 *bytes* (Figura 6.10).

```
HTTP/1.1 200 OK
SOAPAction: ""
Content-Type: text/xml;charset=utf-8
Transfer-Encoding: chunked
Server: Sun-Java-System/Web-Services-Pack-1.4

1c9
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
  xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:ns0="http://service.benchmark"
  xmlns:ns1="http://ws.communication.benchmark">
  <env:Body>
    <ns1:echoStringResponse>
      <result>WSBenchmark</result>
    </ns1:echoStringResponse>
  </env:Body>
</env:Envelope>
0
```

Figura 6.10 - Resposta SOAP/HTTP gerada pelo *toolkit* JWSDP

Analisando as Figuras 6.11 e 6.12, verificou-se que o *toolkit* SSJ também utiliza o protocolo HTTP 1.1 para enviar e receber as requisições e configura os atributos “Connection: keep-alive” na requisição da operação e “Transfer-Encoding: chunked” no cabeçalho da resposta do servidor.

```
POST /WSBenchmark/ HTTP/1.1
Host: 192.168.1.1:8080
Connection: keep-alive
Content-type: text/xml;charset=UTF-8
Content-length: 571

<?xml version="1.0" encoding="UTF-8"?>
<e:Envelope xmlns:e="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:d="http://www.w3.org/2001/XMLSchema"
  xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:wn2="http://systinet.com/wsdl/benchmark/service/"
  xmlns:wn3="http://systinet.com/wsdl/benchmark/communication/ws/">
  <e:Body>
    <wn3:echoString><wn3:p0 i:type="d:string">WSBenchmark</wn3:p0>
    </wn3:echoString>
  </e:Body>
</e:Envelope>
```

Figura 6.11 - Requisição SOAP/HTTP gerada pelo *toolkit* SSJ

Outra semelhança entre os *toolkits* JWSDP e SSJ é que os mesmos agrupam todos os *namespaces* XML dentro da *tag* <envelope> e informam a versão XML usada para formatar a mensagem, porém apenas o SSJ informa o tipo dos parâmetros da operação na mensagem SOAP.

```
HTTP/1.1 200 OK
Date: Sat, 22 Jul 2006 16:46:20 GMT
Content-type: text/xml;charset=UTF-8
Transfer-Encoding: chunked

27e
<?xml version="1.0" encoding="UTF-8"?>
<e:Envelope
  xmlns:e="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:d="http://www.w3.org/2001/XMLSchema"
  xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:wn1="http://systinet.com/xsd/SchemaTypes/"
  xmlns:wn3="http://systinet.com/wsdl/benchmark/service/"
  xmlns:wn4="http://systinet.com/wsdl/benchmark/communication/ws/">
  <e:Body>
    <wn4:echoStringResponse>
       <wn4:response i:type="d:string">WSBenchmark</wn4:response>
    </wn4:echoStringResponse>
  </e:Body>
</e:Envelope>
0
```

Figura 6.12 - Resposta SOAP/HTTP gerada pelo *toolkit* SSJ

Para comparar o tamanho das mensagens geradas pelos *toolkits*, operações transportando tipos de dados simples e complexos foram analisadas. A Tabela 6.3 lista algumas operações da interface `IWSBenchmark`, apresentando o tamanho das requisições e respostas, incluindo, no caso dos *toolkits* SSJ e JWSDP, a quantidade de blocos em que as mensagens foram quebradas quando mais de 1 bloco foi requerido.

Analisando os resultados obtidos, verifica-se que os *toolkits* geram mensagens de tamanhos diferentes para um mesmo estilo de codificação. Para todas as operações analisadas, o *toolkit* Axis gerou mensagens menores, enquanto que o *toolkit* SSJ gerou mensagens maiores que o JWSDP, com exceção das operações `echoVoid` e `acceptParameters`.

Analisando a operação `returnMyComplexObjects`, verificou-se que a diferença entre o tamanho da resposta gerada pelo SSJ em relação ao JWSDP foi de, aproximadamente, 8 *megabytes*, enquanto que a diferença entre o JWSDP e o Axis foi

de 723 *bytes*. A diferença no tamanho das mensagens entre o SSJ e JWSDP é maior, quando as operações utilizam *arrays* de algum tipo de dados como, por exemplo, `echoMyComplexObject`, `returnDoubles` e `returnMyComplexObjects`. Isso significa que o *toolkit* SSJ é ineficiente para representar esse tipo de estrutura de dados, pois o mesmo utiliza uma classe para encapsular essa estrutura de dados e os *toolkits* Axis e JWSDP enviam e retornam diretamente o *array*.

Tabela 6.3 - Tamanho das mensagens em *bytes*

Toolkit / Operação	AXIS		JWSDP		SSJ	
	Requisição	**Resposta**	**Requisição**	**Resposta**	**Requisição**	**Resposta**
echoVoid	276	284	395	403	306	314
echoInt	309	327	423	441	552	619
echoDouble	317	337	431	451	563	632
echoBoolean	320	338	434	452	564	631
echoString	325	343	439	457	571	638
echoMySimpleObject	418	439	532	553	773	843
echoMyComplexObject	2880	2961	2994	3075	5974	6125
acceptParameters	426	292	513	411	473	322
acceptMySimpleObject	422	296	536	415	619	326
returnSting (7000)	280	7336	399	7450	511	7630
returnString (14000)	280	14336	399	14450 (2 blocos)	511	14631 (2 blocos)
returnDoubles	281	16312	400	11935 (2 blocos)	512	25142 (4 blocos)
returnMyComplexObjects	290	183630	409	184353 (23 blocos)	521	8390097 (41 blocos)

Tarefa 8: Analisar o tráfego de pacotes

A última tarefa do processo de avaliação tem como objetivo entender os detalhes da comunicação dos *toolkits* Axis, JWSDP e SSJ analisando o tráfego de rede entre o cliente e o servidor usando a ferramenta Ethereal (ver Seção 6.2.2). As Figuras 6.13, 6.14 e 6.15 mostram os pacotes de rede trocados na comunicação SOAP desses *toolkits* respectivamente.

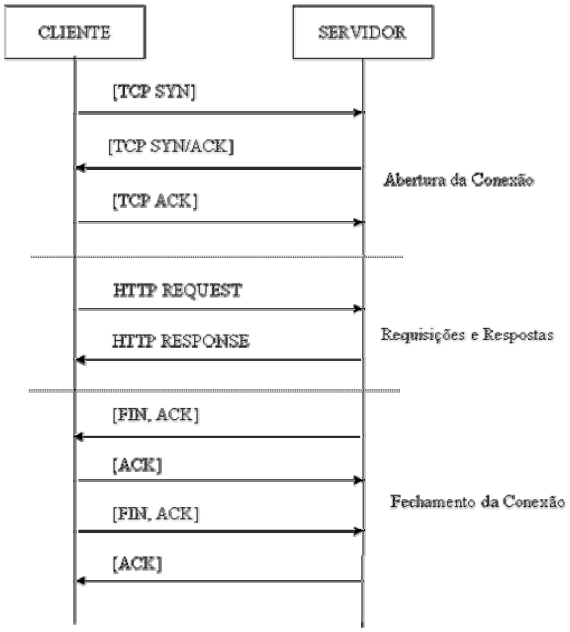

Figura 6.13 - Tráfego de pacotes do *toolkit* Axis

Analisando a Figura 6.13, verifica-se que o Axis não envia o cabeçalho HTTP e o corpo da mensagem em pacotes separados, conforme identificado nos trabalhos anteriores [Elfwing et al., 2002] [Davis and Parashar, 2002]. Isso significa que o Axis 1.2 RC2 resolveu esse problema, otimizando seu código. Outras características que podem ser identificadas são o fechamento da conexão iniciado pelo servidor e a abertura de uma nova conexão para cada nova requisição.

A documentação do *toolkit* Axis foi estudada para verificar a viabilidade de alterar, do lado do cliente, a versão do protocolo HTTP 1.0 para HTTP 1.1, a fim de habilitar o uso de conexões persistentes. A partir dessa análise, verificou-se que para usar o protocolo HTTP 1.1 no cliente Axis, é necessário configurar a classe `CommonsHTTPSender` no arquivo de instalação do serviço chamado de client-config.wsdd, que deve estar configurado no *classpath*. Mesmo fazendo essa alteração, uma nova conexão continua sendo aberta a cada requisição, porque a implementação do *toolkit* força o fechamento da mesma enviando o atributo “Connection: close”. Logo, essa alteração não reduz o tempo de execução.

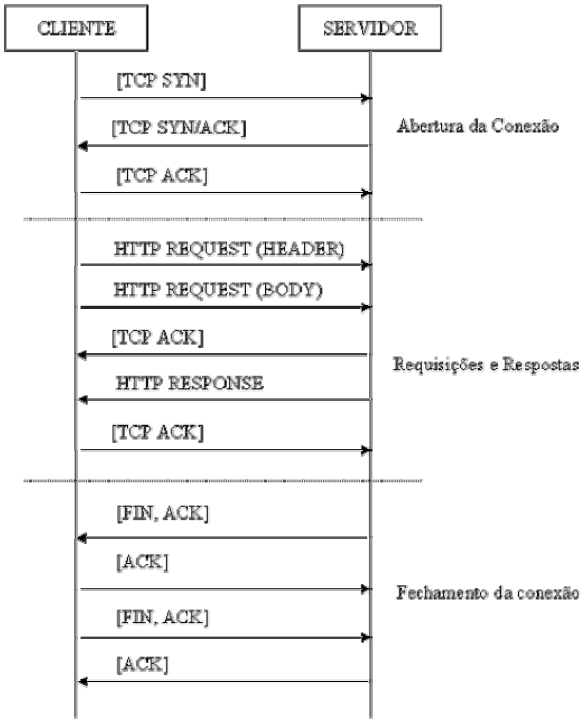

Figura 6.14 - Tráfego de pacotes do *toolkit* JWSDP

A Figura 6.14 ilustra os pacotes trocados na comunicação SOAP do *toolkit* JWSDP, e a partir da sua análise, verifica-se que o seu comportamento padrão é:

- Abrir uma conexão e apenas fechá-la depois que várias requisições são transmitidas;
- Separar o cabeçalho do corpo da mensagem em pacotes diferentes. Esse comportamento é um gargalo de desempenho, pois dois *buffers* e duas chamadas de sistemas são necessárias para enviar a requisição (ver Seção 3.5.4);
- Quebrar a resposta do servidor em vários blocos precedidos pelo seu tamanho. Porém, não é permitido ao cliente usar essa técnica, uma vez que o atributo “Content-Length” é sempre enviado no cabeçalho;
- Servidor inicia o fechamento da conexão.

Diferentemente dos *toolkits* Axis e JWSDP, na implementação do SSJ o responsável pelo fechamento da conexão é o cliente (ver Figura 6.15). Embora a Seção 3.6.5 considere esse comportamento uma otimização, analisando em detalhes o pacote que inicia o fechamento da conexão, verificou-se que o cliente espera, em média, 800ms

para enviar o mesmo ao servidor. Existem, portanto, problemas na implementação no cliente do *toolkit* SSJ com relação ao fechamento da conexão. Além desse comportamento, SSJ também usa conexões persistentes e adota a técnica de *streaming* dos dados.

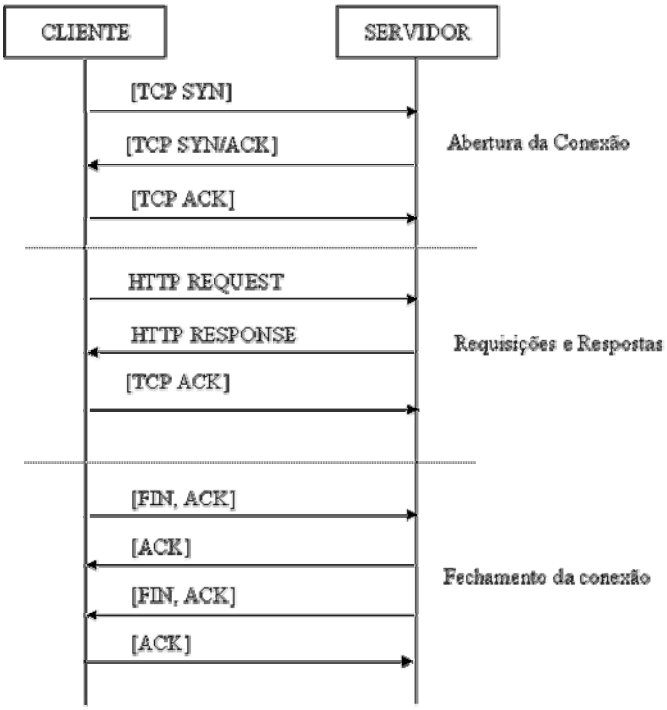

Figura 6.15 - Tráfego de pacotes do *toolkit* SSJ

6.4 Considerações Finais

Nesse capítulo foram realizados vários experimentos com o objetivo de analisar e comparar o desempenho dos *Web Services toolkits* suportados pelo utilitário, aplicando o processo proposto para uniformizar a coleta dos resultados. Uma aplicação-teste simples foi definida, porém suficiente para validar o processo proposto e identificar os gargalos dos *toolkits* analisados.

Os experimentos realizados avaliaram o tempo para instanciar um *stub* de forma dinâmica e estática, a latência, o tempo para tratar uma exceção definida pelo usuário e o tempo de respota médio para enviar e receber mensagens de tamanhos e complexidades diferentes. Além disso, a análise do tráfego de pacotes e o

monitoramento das mensagens SOAP foram realizados para o entender o comportamento dos *toolkits*. Resumindo as características principais identificadas de cada *toolkit*, a Tabela 6.4 foi estruturada para facilitar a comparação entre os mesmos.

Tabela 6.4 - Comparação dos *toolkits*

Características	Axis	JWSDP	SSJ
Parser XML	Xerces	SJSXP	Xerces
Geração para instanciar o *stub*	Estática	Estática	Dinâmica
Latência	Média	Baixa	Baixa
RTT para transamitir tipos básicos	Médio	Baixo	Baixo
RTT para transamitir tipos complexos	Médio	Baixo	Médio
Tempo de (de)serialização	Alto	Baixo	Alto
Tempo para tratar exceções	Alto	Baixo	Baixo
Tamanho da mesnagem para estrutura complexas	Pequeno	Médio	Grande
Cálculo do atributo “Content-Length”	Sim	Sim	Sim
Cabeçalho e o corpo da mensagem enviados no mesmo pacote	Sim	Não	Sim
Conexões Persistentes	Não	Sim	Sim
Streaming dos dados	Não	Sim	Sim
Gargalo para fechar a conexão	Não	Não	Sim

O *toolkit* JWSDP (*Java Web Services Developer Pack*) apresentou o melhor desempenho em todos os experimentos, respondendo mais rapidamente às requisições do cliente. O *toolkit* Axis apresentou um tempo alto de resposta, gastando aproximadamente 7 ms para enviar mensagens compostas por tipos de dados simples. Diferentemente, o *toolkit* SSJ (*Systinet Server for Java*) apresentou um desempenho semelhante ao *toolkit* JWSDP quando mensagens simples eram transportadas. Porém, apresentou problemas de desempenho quando a complexidade das mensagens aumentou.

Um resultado importante foi a confirmação que o tamanho das mensagens não é único gargalo de desempenho de *Web Services*, pois o *toolkit* Axis apresentou o maior tempo para enviar tipos de dados simples e complexos mesmo gerando as menores

mensagens e o *toolkit* SSJ gerou mensagens maiores e apresentou um tempo de resposta menor que o Axis.

Os *toolkits* SSJ e JWSDP, em relação ao Axis, aplicam duas otimizações que influenciam no seu bom desempenho. A primeira é o uso de conexões persistentes, onde as várias requisições são transmitidas numa mesma conexão. E a segunda é a técnica de *streaming*, chamada *Chunked Transfer Coding*, utilizada pelo servidor para enviar as respostas ao cliente. Além dessas otimizações, apenas o *toolkit* JWSDP adota o modelo *Pull Parsing*, que é mais eficiente, utilizando o *parser* SJSXP (*Sun Java Streaming XML Parser*). Essas otimizações tornam o JWSDP mais eficiente.

7 Conclusões e Trabalhos Futuros

Essa dissertação abordou um problema em aberto na área de *Web Services*: seu desempenho. A decisão de adotar a tecnologia *Web Services* tem sido tomada apenas em função da sua interoperabilidade. Porém, o seu desempenho deve ser avaliado, pois sua ineficiência limita o desenvolvimento de aplicações que demandam por desempenho.

O foco desse trabalho foi avaliar o desempenho de *Web Services toolkits*, propondo uma estratégia de avaliação para selecionar o *toolkit* "ideal" para desenvolver e expor um serviço, pois existem várias implementações de *toolkits* disponíveis no mercado.

A primeira atividade para montar essa estratégia foi estudar e organizar os gargalos de desempenho de *Web Services*. Os principais gargalos detalhados nessa dissertação foram o tamanho e a complexidade das mensagens, o tempo gasto no cálculo do tamanho da mensagem, a escolha do *parser* XML, os custos de serialização e deserialização, o estilo de codificação, os custos do estabelecimento da conexão e os gargalos de comunicação como o atraso na troca de pacotes e o número de pacotes.

Além desses, a escolha do estilo de codificação também afeta diretamente a eficiência de *Web Services*. Conforme apresentado na Seção 3.4, o estilo *Document/Literal* oferece melhor desempenho que o estilo *RPC/Encoded*, porque

resulta em mensagens menores e menos complexas, minimizando os custos de transmissão na rede e a latência. Além de desempenho, o estilo *Literal/Wrapped* apresenta melhores resultados de interoperabilidade.

De forma geral, o desempenho de uma aplicação *Web Services* dependem do projeto e implementação do *toolkit* utilizado para implementá-la e dos gargalos introduzidos pelos protocolos SOAP e de transporte, onde o protocolo HTTP é o mais comumente adotado.

A partir desses gargalos, foram publicadas diretrizes para guiar a avaliação de desempenho de *Web Services toolkits* antes de desenvolver as aplicações, a fim de identificar o mais apropriado para atender os seus requisitos não-funcionais como eficiência, baixa latência e alta vazão.

As diretrizes desenvolvidas são simples e fáceis de aplicar, focando desde a análise da documentação de um *toolkit*, a fim de verificar o *parser* adotado e o estilo de codificação suportado, à quantificação de métricas de desempenho. Além disso, as diretrizes propõem o monitoramento das mensagens SOAP e do tráfego de rede. Com isso, as mesmas representam os principais pontos que devem ser observados durante a avaliação.

Do ponto de vista operacional, foi proposto um processo para uniformizar a avaliação de desempenho de diferentes *toolkits*. De forma geral, o processo representa um guia prático, definindo um passo a passo para executar a avaliação, porém sempre embasado pelas diretrizes.

Uma vez que o processo de avaliação pode demandar muito tempo, verificou-se a necessidade de automatizar algumas de suas tarefas, principalmente os passos referentes à parte de implementação, compilação, execução da aplicação cliente e coleta de métricas.

Para facilitar essa tarefa, o utilitário JWSPerf (*Java Web Services* Performance) foi desenvolvido. JWSPerf automatiza uma parte do processo proposto, pois o desenvolvedor apenas configura alguns parâmetros antes de executar o utilitário. JWSPerf suporta três *toolkits* Java bastante conhecidos e usados – Apache Axis, JWSDP (*Sun Java Web Services Developer Pack*) e SSJ (*Systinet Server for Java*). Para automatizar a geração de código, o JWSPerf é composto por dois módulos com tarefas

bem definidas – o de geração de classes de teste e o de invoção – e essa estruturação permite que novos *toolkits* sejam facilmente incorporados.

Mesmo utilizando uma aplicação-teste simples, os resultados obtidos foram suficientes para validar a importância da aplicação das diretrizes e do utilitário para viabilizar a avaliação.

O presente trabalho apresentou a importância de fazer uma análise detalhada do desempenho do *Web Services toolkit* antes de adotá-lo, pois as questões sobre sua eficiência devem ser resolvidas antes ou durante a implementação da aplicação, dado que o *toolkit* é responsável por boa parte do desempenho da mesma. O objetivo da estratégia proposta é simplificar a exposição de aplicações *Web Services* usando um *toolkit* eficiente. Além de buscar o bom desempenho, é necessário entender o negócio da aplicação para projetar sua interface e escolher os tipos de dados apropriados, pois a natureza da aplicação afeta sua eficiência.

7.1 Principais Contribuições

A seguir são resumidas as principais contribuições deste trabalho:

- Organização dos gargalos de desempenho de *Web Services*.
- Publicação de diretrizes guiam a avaliação, focando nos principais aspectos de um *toolkit* que devem ser analisados.
- Elaboração de um processo, a fim de uniformizar a avaliação de desempenho de *toolkits* e facilitar a escolha do *toolkit* "ideal" para desenvolver uma aplicação.
- Desenvolvimento do utilitário JWSPerf (*Java Web Services* Performance) de código aberto que automatiza parte das tarefas do processo proposto para avaliação de desempenho de *Web Services toolkits*.
- Usando o utilitário JWSPerf, reduz-se o tempo gasto na avaliação e no estudo da documentação dos *toolkits* e na implementação do código em

si, uma vez que os desenvolvedores apenas necessitam configurar algumas propriedades nos arquivos de configuração e rodar os comandos.

- Desenvolvimento de um *benchmark* simples que pode ser utilizado para avaliar o desempenho de outros *Web Services toolkits*.

7.2 Trabalhos Futuros

Entre os principais tópicos possíveis de extensão em trabalhos futuros, podemos citar as seguintes melhorias:

- Suportar *toolkits* implementados em linguagens diferente de Java, como C#, C, C++.
- Incoporar outros *toolkits* Java ao utilitário JWSPerf, como por exemplo, o Glue da webMethods [webMethods, 2004].
- Incoporar ao JWSPerf outras métricas, como por exemplo, o cálculo do intervalo de confiança.
- Elaborar interfaces gráficas para melhorar a interação do usuário com o JWSPerf.
- Automatizar a construção do código do lado do servidor, passando a ser responsável pela geração da camada de comunicação *Web Services* da aplicação para diferentes *toolkits*;
- Avaliar a interoperabilidade de diferentes *Web Services toolkits*, usando o utilitário JWSPerf para automatizar essa tarefa uma vez que o mesmo gera o código a partir da interface WSDL.
- Projetar o utilitário para utilizar vários computadores, a fim de distribuir a carga e gerar carga intensa para avaliar a escalabilidade dos *Web Services toolkits*.

Referências Bibliográficas

[Abu-Ghazaler et al., 2004] N. Abu-Ghazaleh, M. J. Lewis and M. Govindaraju, “Differential Serialization for Optimized SOAP Performance", Proceedings of the 13th International Symposium on High Performance Distributed Computing (HPDC), Honolulu, Hawaii, pp. 55-64, 2004.

[Abu-Ghazaler et al., 2004a] N. Abu-Ghazaleh, M. J. Lewis and M. Govindaraju, “Performance of Dynamically Resizing Message Fields for Differential Serialization of SOAP Messages", Proceedings of the International Symposium on Web Services and Applications (ISWS), Honolulu, Hawaii, pp. 783-789, 2004.

[Abu-Ghazaler et al., 2004b] N. Abu-Ghazaleh, M. Govindaraju and M. J. Lewis, "Optimizing Performance of Web Services with Chunk-Overlaying and Pipelined-Send”, Proceedings of the International Conference on Internet Computing (ICIC), pp. 482-485, 2004.

[Apache Axis, 2004] Apache Software Foundation, “Apache Axis”, http://ws.apache.org/axis/, 2004.

[Apache SOAP, 2004] Apache Software Foundation, “Apache SOAP”, http://ws.apache.org/soap/, 2004.

[Austin et al., 2004] D. Austin, A. Barbir, C. Ferris and S. Garg, “Web Services Architecture Requirements”, http://www.w3.org/TR/wsa-reqs/, 2004.

[Berners et al., 1996] T. Berners-Lee, R. Fielding and H. Frystyk. “Hypertext Transfer Protocol – HTTP/1.0”, IETF RFC 1945, http://www.ietf.org/rfc/rfc1945, 1996.

[Box et al., 2000] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte and D. Winer, “Simple Object Access Protocol (SOAP) 1.1”, http://www.w3.org/TR/soap11/, 2000.

[Cai et al., 2002] M. Cai, S. Ghandeharizadeh and S. Song, “A Comparison of Alternative Encoding Mechanism for Web Services”, Proceedings of the 13th International Conference on Database and Expert Systems Applications (DEXA), Aix en Provence, France, pp. 93-102, 2002.

[Chiu et al., 2002] K. Chiu, M. Govindaraju and R. Bramley, “Investigating the Limits of SOAP Performance for Scientific Computing”, Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC 2002), Edinburgh, Scotland, pp. 246-254, 2002.

[Cohen, 2003] F. Cohen, “Discover SOAP encoding’s impact on Web service performance”, http://www-128.ibm.com/developerworks/ webservices/library/ws-soapenc/, 2003.

[Davis and Parashar, 2002] D. Davis and M. Parashar, “Latency Performance of SOAP Implementations”, Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID), Berlin, Germany, pp. 407-412, 2002.

[Devaram and Anresen, 2003] K. Devaram and D. Anresen, “SOAP Optimization Via Parameterized Client-Side Caching”, Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, Marina Del Rey. CA, pp. 785-790, 2003.

[Elfwing et al., 2002] R. Elfwing, U. Paulsson and L. Lundberg, “Performance of SOAP in Web Service Environment Compared to CORBA”, Proceedings of the Ninth Asia-Pacific Software Engineering Conference (APSEC), Queensland, Australia, pp. 84, 2002.

[Engelen, 2003] R. A. Van Engelen, “Pushing the SOAP Envelope With Web Services for Scientific Computing”, Proceedings of the International Conference on Web Services (ICWS), Las Vegas, pp. 346-352, 2003.

[Engelen and Gallivan, 2002] R. A. van Engelen and K. A. Gallivan, “The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks”, Proceedings of the 2nd IEEE/ACM International Symposium on

Cluster Computing and the Grid (CCGrid), Berlim, Germany, pp. 128-135, 2002.

[Ethereal, 2004] Ethereal, "A Network Protocol Analyser", http://www.ethereal.com, 2004.

[Extreme!, 2004] Extreme! Laboratory of Indiana University, "Grid Web Services", http://www.extreme.indiana.edu/xgws/#projects, 2004.

[Fielding et al., 1999] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee, "Hipertext Transfer Protocol -- HTTP/1.1", IETF RFC 1626, http://www.ietf.org/rfc/rfc2616.txt, 1999.

[FIX, 2005] FIX Protocol Ltd, "The Financial Information Exchange Protocol (FIX)", http://www.fixprotocol.org/specification/fix-43-pdf.zip, 2005.

[Govindaraju et al., 2000] M. Govindaraju, A. Slominski, V. Chopella, R. Bramley and D. Gannom, "Requirements for and Evaluation of RMI Protocols for Scientific Computing", Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM), Dallas, Texas, pp. 61, 2000.

[Govindaraju et al., 2004] M. Govindaraju, A. Slominski, K. Chiu, P. Liu, R. van Engelen and M. J. Lewis, "Toward Characterizing the Performance of SOAP Toolkits", Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, Pittsburgh, USA, pp. 365-372, 2004.

[Gray, 2004] N. A. B. Gray, "Comparison of Web Services, Java-RMI, and CORBA service implementations", Proceedings of the 5th Australasian Workshop on Software and System Architectures (ASWEC 2004), Melbourne, Australia, pp. 52-63, 2004.

[Gray, 2005] N. A. B. Gray, "Performance of Java Middleware – Java RMI, JAXRPC, and CORBA", Proceedings of the 6th Australasian Workshop on Software and System Architectures (AWSA 2005), Brisbane, Australia, pp. 31-39, 2005.

[gSOAP, 2004] gSOAP Toolkit. "gSOAP: C/C++ Web Services and Clients", http://www.cs.fsu.edu/~engelen/soap.html, 2004.

[Head et al., 2005] M. Head, M. Govindaraju, A. Slominski, P. Liu, N. Abu-Ghazaleh, R. Engelen, K. Chiu and M. Lewis, "A Benchmark Suíte for SOAP-based Communication in Grid Web Services", Proceedings of the 2005 ACM/IEEE conference on Supercomputing, Seattle, pp. 19, 2005.

[Hericko et al., 2003] M, Hericko, M. Juric, I. Rozman and A. Zivkovic, “Object Serialization Analysis and Comparison in Java and .NET”, ACM SIGPLAN Notices, Vol 38, Nº 8, pp. 44-54, 2003.

[Juric et al., 2004] M. B. Juric, B. Kezmah, M. Hericko, I. Rozman and I. Vezocnik, “Java RMI, RMI Tunneling and Web Services Comparison and Performance Analysis”, Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Vol. 39, Nº 5, Venice, Italy, pp. 58-65, 2004.

[Kennington, 2005] A. Kennington, “Network Traffic Monitoring”, http://www.topology.org/comms/netmon.html, 2005.

[Kohlhoff and Steele, 2003] C. Kohlhoff and R. Steele, “Evaluating SOAP for High Performance Business Applications: Real-Time Trading Systems”, Proceedings of the 12th International World Wide Web Conference (WWW2003), Budapest, Hungary, 2003.

[Machado and Ferraz, 2005] A. C. C. Machado and C. A. G. Ferraz, “Guidelines for Performance Evaluation of Web Services”, Proceedings of the 11th Brazilian Symposium on Multimedia and the web (WebMedia '05), Poços de Caldas, Minas Gerais, Brazil, pp. 1-10, 2005.

[Machado and Ferraz, 2006] A. C. C. Machado and C. A. G. Ferraz, “JWSPerf: A Performance Benchmarking Utility with Support to Multiple Web Services Implementations”, Proceeding of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT/ICIW 2006), Guadalupe, French Caribbean, pp. 159, 2006.

[Manes, 2004] A. Manes, “The wrapped document/literal convention”, http://atmanes.blogspot.com/2005/03/wrapped-documentliteral-convention.html, 2004.

[McGoven et al., 2003] McGovern, J., Tyagi, S., Stevens, M., Mathew, S., “Java Web Services Architecture, Morgan Kaufmann Publishers, 2003.

[Meyers, 2005] N. Meyers, “PerfAnal: A Performance Analisys Tool”, http://java.sun.com/developer/technicalArticles/Programming/perfanal/, 2005.

[Microsoft, 2004] Microsoft Corporation, “.NET Framewrok Developer Center”, http://msdn.microsoft.com/netframework/, 2004.

[Microsoft, 2004b] Microsoft Corporation, “Web Services Performance: Comparing Java™ 2 Enterprise Edition (J2EE™ platform) and .NET Framework. A Response to Sun Microsystem’s Benchmark”,

http://www.gotdotnet.com/team/compare/Benchmark_response.pdf, 2004.

[Ng et al., 2003] A. Ng, S. Chen and P. Greenfield, “Evaluation of Contemporary Commercial SOAP”, Proceedings of the 5th Australasian Workshop on Software and System Architectures (AWSA), Melbourne, Australia, pp. 64-71, 2003.

[Qworks, 2004] Qworks, “Web Services Performance: Comparing JWSDP (Java Web Service Developer Pack) TM, AXIS and .NET Framework TM, Version 0.1”, http://groups-beta.googe.com/group/qworks, 2004.

[Roubtsov, 2004] V. Roubtsov, “My kingdom for a good timer”, http://www.javaworld.com/javaworld/javaqa/2003-01/01-qa-0110-timing.html, 2004.

[SAX, 2004] SAX, “Official Website for SAX”, http://www.saxproject.org/, 2004.

[Shirasuma et al., 2002] S. Shirasuma, H. Nakata, S. Matsuoka and S. Sekiguchi, “Evaluating Web Services Based Implementations of GridRPC”, Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HDPC 2002), Edinburgh, Scotland, pp. 237-245, 2002.

[SOAP Builders, 2004] SOAP Builders, “SOAP Builders Forum”, http://groups.yahoo.com/group/soapbuilders/, 2004.

[SoapWare.Org, 2004] SoapWare.Org, “The Leading Directory for SOAP 1.1 Developers”, http://www.soapware.org/directory/4/implementations, 2004.

[Suciu and Liefke, 2004] D. Suciu and H. Liefke, “Xmill: An Efficient Compressor for XML Data”, http://www.research.att.com/sw/tools/xmill/, 2004.

[Sun, 2004] Sun Microsystems, “Java Web Services Developer Pack (Java WSDP)”, http://java.sun.com/webservices/jwsdp/index.jsp, 2004.

[Sun, 2004b] Sun Microsystems, “Web Services Performance: Comparing Java™ 2 Enterprise Edition (J2EE™ platform) and .NET Framework”, http://java.sun.com/performance/reference/whitepapers/WS_Test-1_0.pdf, 2004

[Systinet, 2004] Systinet, “Systinet Server for Java”, http://www.systinet.com/products/ssj/overview, 2004.

[webMethods, 2004] webMethods, “Glue Evaluation”, http://www.webmethods.com/meta/default/folder/0000006047, 2004.

[Ying et al., 2004] Y. Ying, Y. Huang and D. W. Walker, “Using SOAP with Attachment for e-Science”, Proceedings of the UK e-Science All Hands Meeting, Nottingham, UK, pp. 1061-1064, 2004.

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

Ata de Defesa de Dissertação de Mestrado do Centro de Informática da Universidade Federal de Pernambuco, 28 de agosto de 2006.

Ao vigésimo oitavo dia do mês de agosto do ano dois mil e seis, às catorze horas e trinta minutos, no Centro de Informática da Universidade Federal de Pernambuco, teve início a **qüingentésima qüinquagésima oitava** defesa de dissertação de Mestrado em Ciência da Computação intitulada **"Diretrizes e um Utilitário para Avaliação de Desempenho de Toolkits Web Services"** da candidata **Ana Carolina Chaves Machado**, a qual já havia preenchido anteriormente as demais condições exigidas para a obtenção do grau de mestre. A Banca Examinadora, composta pelos professores André Luis de Medeiros Santos, pertencente ao Centro de Informática desta Universidade, Walfredo Costa Cirne Filho, pertencente ao Departamento de Sistemas e Computação da Universidade Federal de Campina Grande e Carlos André Guimarães Ferraz, pertencente ao Centro de Informática desta Universidade, sendo o primeiro presidente da Banca Examinadora e o último orientador do trabalho de dissertação, resolveu: **Aprovar por unanimidade e dar o prazo de trinta dias para entrega da versão final do trabalho**. E para constar lavrei a presente ata que vai por mim assinada e pela Banca Examinadora. Recife, 28 de agosto de 2006.

Maria Lília Pinheiro de Freitas
(secretária)

Prof. André Luis de Medeiros Santos
(primeiro examinador)

Walfredo Costa Cirne Filho
(segundo examinador)

Prof. Carlos André Guimarães Ferraz
(terceiro examinador)