CAPÍTULO 1

Java Web Services Uma Rápida Introdução

O Que São Web Services?

Embora o termo *web service* tenha vários imprecisos e desenvolvidos significados, uma olhada em algumas características típicas de web services será o suficiente para nos introduzir na codificação de um web service e um cliente, também conhecidos como um consumidor ou solicitante. Como o nome sugere, um web service é um tipo de aplicação para a web, isto é, uma aplicação tipicamente oferecida através de *HTTP* (Hyper Text Transport Protocol). Um web service é, então, uma aplicação distribuída, cujos componentes podem ser aplicados e executados em dispositivos distintos. Por exemplo, um web service de escolha de ações pode consistir de diversos componentes de código, cada um armazenado em um servidor de grau de negócio separado, e o web service pode ser consumido em PCs, handhelds e outros dispositivos.

Web services podem ser divididos em dois grupos, baseados em *SOAP* e de estilo *REST*. A distinção não é precisa porque, como um futuro exemplo de código ilustra, um serviço baseado em SOAP fornecido através de HTTP é um caso especial de um serviço de estilo REST. *SOAP* originalmente significa Simple Object Access Protocol (Protocolo de Acesso a Objetos Simples), mas agora pode significar Protocolo de Arquitetura Orientada a Serviço em inglês Service Oriented Architecture (SOA). Desconstruir SOA não é trivial, mas um ponto é indiscutível: seja lá o que SOA for, web services possuem um papel central na abordagem SOA para design e desenvolvimento de software. (Isto é parcialmente certo. SOA oficialmente não é mais uma sigla, e SOAP e SOA podem viver separadas uma da outra). No momento, SOAP é apenas um dialeto *XML* (EXtensible Markup Language), no qual documentos são mensagens. Em web services baseados em SOAP, SOAP é a infraestrutura menos vista. Por exemplo, em um típico cenário, chamado padrão de troca de mensagem pedido/resposta, em inglês, *Message Exchange Parttern* (MEP) solicitação/resposta, a biblioteca SOAP fundamental do cliente envia uma mensagem SOAP como uma solicitação de serviço, e a biblioteca SOAP subjacente do web service envia outra mensagem SOAP como a resposta correspondente ao serviço. O código fonte do cliente e do web service pode oferecer algumas dicas sobre o SOAP subjacente (veja Figura 1-1).

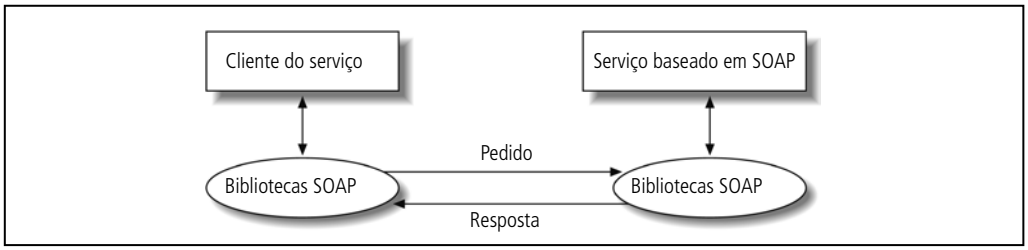

Figura 1-1. Arquitetura de um típico web service baseado em SOAP.

Representational State Transfer em português REST significa (Transferência de Estado Representacional). Roy Fielding, um dos principais autores da especificação HTTP, criou a sigla em sua dissertação de PhD para descrever um estilo arquitetural no desenvolvimento de web services. SOAP tem padrões no World Wide Web Consortium [W3C], toolkits e nas inúmeras bibliotecas de software. REST não tem padrões, possui poucos toolkits, e escassas bibliotecas de software. O estilo REST é freqüentemente visto como um antídoto para a assustadora complexidade de web services baseados em SOAP. Este livro aborda web serviços baseados em SOAP e de estilo REST, começando com os baseados em SOAP.

Exceto no modo de teste, o cliente de um web service baseado em SOAP ou de estilo REST raramente é um navegador web, mas sim uma aplicação sem uma interface gráfica do usuário. O cliente pode ser escrito em qualquer linguagem com as apropriadas bibliotecas de suporte. Na verdade, a principal atração de web services é a transparência de linguagens: o serviço e seus clientes não precisam ser escritos na mesma linguagem. Transparência de linguagem é a chave para a ***interoperabilidade*** de web service; isto é, a habilidade de web services e solicitantes interagirem apesar das diferenças nas linguagens de programação, bibliotecas de suporte e plataformas. Para enfatizar este atrativo, clientes nos nossos web services Java serão escritos em várias linguagens como C#, Perl e Ruby, e clientes Java irão consumir serviços escritos em outras linguagens, incluindo linguagens desconhecidas.

É claro que não há mágica na transparência de linguagem. Se um web service baseado em SOAP, escrito em Java, pode ter um consumidor Perl ou Ruby, deve haver um intermediário que gerencia as diferenças de tipos de dados entre as linguagens do serviço e do solicitante. Tecnologias XML, que suportam troca e processamento de documento estruturado e agem como o intermediário. Por exemplo, em um típico web service baseado em SOAP, um cliente transparentemente envia um documento SOAP como uma solicitação para um web service, que transparentemente retorna outro documento SOAP como uma resposta. Em um serviço de estilo REST, um cliente pode enviar uma solicitação HTTP padrão para um web service e recebe um documento XML apropriado como uma resposta.

Diversas características distinguem web services de outros sistemas de software distribuídos. Aqui estão três:

Infraestrutura aberta

Web services são implantados que usam protocolos-padrão da indústria e independentes do fornecedor como HTTP e XML, são onipresentes e bem compreendidos.

Web services vão suportar a rede, formatação de dados, segurança e outras infraestruturas já configuradas, o que diminui os custos de entrada e promove interoperabilidade entre os serviços.

Transparência de linguagem

Web services e seus clientes podem interoperar, mesmo se forem escritos em diferentes linguagens de programação. Linguagens como C/C++, Java, Perl, Python, Ruby e outras oferecem bibliotecas, utilitários e até frameworks em suporte dos web services.

Design modular

Web services são designados para serem modulares em design, para que novos serviços possam ser gerados através da integração e agrupamento de serviços existentes. Imagine, por exemplo, um serviço de monitoramento de estoque integrado com um serviço de pedidos online, para gerar um serviço que automaticamente ordena os produtos apropriados em resposta aos níveis do estoque.

Para Que Servem os Web Services?

Esta pergunta óbvia não tem uma única e simples resposta. No entanto, os benefícios-chefe e promessas do web services são claros. Modernos sistemas de software são escritos em uma variedade de linguagens – uma variedade que parece crescer. Estes sistemas de software vão continuar sendo armazenados em uma variedade de plataformas. Grandes e pequenas instituições possuem significativo investimento em sistemas de software legado, cuja funcionalidade é útil e talvez crucial para missões críticas; e poucas destas instituições têm a vontade e os recursos, humanos ou financeiros, de reescrever seus sistemas legado.

É raro que um sistema de software execute em esplêndido isolamento. O típico sistema de software deve interoperar com outros, que podem residir em diferentes hospedeiros e serem escritos em diferentes linguagens. Interoperabilidade não é apenas um desafio a longo prazo, mas também um requisito atual de software em produção.

Web services resolvem estas questões diretamente porque tais serviços são, antes de tudo, neutros de linguagem e plataforma. Se um sistema de legado COBOL é exposto através de um web service, o sistema é então interoperável com clientes de serviço escritos em outras linguagens de programação.

Web services são inerentemente sistemas ***distribuídos*** que se comunicam principalmente através de HTTP, mas que podem se comunicar também através de outros transportes populares. O conteúdo de comunicação de web services são textos estruturados (isto é, documentos XML), que podem ser inspecionados, transformados, persistidos e processados com ferramentas disponíveis amplamente e até gratuitamente. Quando eficiência é demandada, porém, web services também podem fornecer conteúdos binários. Finalmente, web services são um trabalho em progresso, com sistemas distribuídos em tempo real como sua plataforma de teste. Por todas estas razões, web services são uma ferramenta essencial em qualquer caixa de ferramentas do programador moderno.

Os exemplos que seguem, neste capítulo e em outros, são simples o suficiente para isolar características importantes de web services, mas também são realistas o suficiente para ilustrar o poder e flexibilidade que tais serviços trazem ao desenvolvimento de software. Que comecem os exemplos.

Um Primeiro Exemplo

O primeiro exemplo é um web service baseado em SOAP em Java e clientes em Perl, Ruby e Java. O web service baseado em Java consiste de uma interface e uma implementação.

A Interface Endpoint do Serviço e o Bean de Implementação do Serviço

O primeiro web service em Java, como quase todos os outros neste livro, pode ser compilado e implementado usando Java SE 6 (Java Standard Edition 6), ou superior, sem nenhum software adicional. Todas as bibliotecas exigidas para compilar, executar e consumir web services estão disponíveis no Java 6, que suporta *JAX-WS* (Java API for XML - Java para Web Services XML). JAX-WS suporta serviços baseados em SOA e de estilo REST. JAX-WS é comumente abreviado para *JWS* de Java Web Services. A versão atual de JAX-WS é 2.x, que é um pouco confuso, porque a versão 1.x tem um rótulo diferente: JAX-RPC. JAX-WX preserva, mas também estende significativamente as capacidades de JAX-RPC.

Um web service baseado em SOAP poderia ser implementado como uma única classe Java, mas, seguindo as melhores práticas, deve haver uma interface que declare os métodos, que são as operações de web service, e uma implementação, que defina os métodos declarados na interface. A interface é chamada de SEI: Interface Endpoint do Serviço (Service Endpoint Inteface). A implementação também é chamada de SIB: Bean de Implementação do Serviço (Service Implementation Bean). O SIB pode ser um POJO ou um EJB de Sessão sem estado (Enterprise Java Bean). O Capítulo 6, que lida com o Servidor de Aplicação GlassFish, mostra como implementar um web service como um EJB. Até então, os web services baseados em SOAP serão implementados como POJOs, isto é, como instâncias de classes Java regulares. Estes web services serão publicados, usando classes de biblioteca que vêm com o Java 6 e, um pouco depois, com o Tomcast standalone e Glassfish.

Core Java 6, JAX-WS e Metro

Java SE 6 lança o JAX-WS. Porém, JAX-WS tem uma vida fora do Java 6 e uma equipe de desenvolvimento separada. A tecnologia de ponta do JAX-WS é o *Metro Web Services Stack (https://wsit.dev.java.net)*, que inclui Project Tango para promover interoperabilidade entre a plataforma Java e *WCF* (Windows Communication Foundation), também conhecida como Indigo. A iniciativa de interoperabilidade atende pela sigla *WSIT* (Tecnologias de Interoperabilidade de Web Services em inglês Serwice Interoperability Technologies). Em qualquer caso, a atual versão Metro de JAX-WS, daqui em diante *Metro release*, está tipicamente à frente de JAX-WS, que é lançado com o Java 6 SDK. Com Update 4, o JAX-WS, no Java 6, foi de JAX-WS 2.0 para JAX-WS 2.1, uma melhora significativa.

Os frequentes releases Metro corrigem bugs, adicionam características, aliviam a carga para o programador e, em geral, fortalecem JAX-WS. No começo, meu objetivo é introduzir JAX-WS com a menor complicação possível; por enquanto, então, o JAX-WS que vem com Java 6 está bom. De tempos em tempos, um exemplo pode envolver mais trabalho do que necessário sob o atual release do Metro; nestes casos, a idéia é explicar o que realmente está acontecendo antes de introduzir um atalho ao Metro.

A página do Metro oferece download fácil. Depois de instalado, o Metro reside em um diretório chamado jaxws-ri. Exemplos subsequentes que usam release Metro assumem uma variável de ambiente `METRO_HOME`, cujo valor é o diretório de instalação para *jaxws-ri*. O *ri*, a propósito, é a abreviação de implementação de referência (*reference inplementation)*.

Finalmente, o release Metro baixada é uma maneira de fazer JAX-WS sob Java 5. JAX-WS exige no mínimo Java 5, porque suporte à anotações começa em Java 5.

Exemplo 1-1 é a SEI para um web service, que retorna a hora atual, seja como uma string ou os milésimos de segundos decorridos da época Unix, meia-noite Janeiro 1, 1970 GMT.

Exemplo 1-1. Interface de Endpoint de Serviço para TimeServer

```
package ch01.ts; // time server

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;

/**
* A anotação @WebService indica que esta é a
* SEI (Interface de Endpoint de Serviço).
* @WebMethod indica que cada método é uma operação de serviço.

* A anotação @SOAPBinding impacta a construção
* por-debaixo-dos-panos do contrato de serviço, o documento
  WSDL Web Service Definition Languange
* (Linguagem de Definição de Web Services). Style.RPC
* simplifica o contrato e tornar a implementação mais fácil.
//mais sobre isto mais tarde
@WebService
@SOAPBinding(style = Style.RPC) // mais sobre isto mais tarde
public interface TimeServer {
  @WebMethod String getTimeAsString();
  @WebMethod long getTimeAsElapsed();
}
```

Exemplo 1-2 é o SIB, que implementa a SEI.

Exemplo 1-2. Bean de Implementação do Serviço para o TimeServer

```
package ch01.ts;

import java.util.Date;
import javax.jws.WebService;
```

```
/**
*  A propriedade endpointInterface ou @WebService linka o
*  SIB (esta classe) à SEI.
*  Note que as implementações do método não são anotadas
*  como @WebMethods.
*/
@WebService(endpointInterface = "ch01.ts.TimeServer")
public class TimeServerImpl implements TimeServer {
   public String getTimeAsString() { return new Date().toS-
   tring(); }
   public long getTimeAsElapsed() { return new Date().getTime();
   }
}
```

Os dois arquivos são compilados da forma usual, a partir do diretório de trabalho atual, que, neste caso, está imediatamente acima do subdiretório ch01. O símbolo % representa o prompt de comando:

```
% javac ch01/ts/*.java
```

Uma Aplicação Java para Publicar o Web Service

Depois de SEI e SIB serem compilados, o web service está pronto para ser publicado. Em modo de produção, um Servidor de Aplicação Java, como BEA WebLogic, GlassFish, JBoss ou WebSphere pode ser usado; mas em modo de desenvolvimento e até de leve produção, uma simples aplicação Java pode ser usada. Exemplo 1-3 é aplicação publicadora para o serviço TimeServer.

Exemplo 1-3. Publicador endpoint para o TimeServer

```
package ch01.ts;

import javax.xml.ws.Endpoint;

/**
* Esta aplicação publica o web service cuja
* SIB é ch01.ts.TimeServerImpl. Por enquanto,
* o serviço é publicado no endereço de rede 127.0.0.1.,
* que é localhost, e na porta número 9876, pois esta
* porta está provavelmente disponível em qualquer máquina desktop.
* O caminho da publicação é /ts, * um nome arbitrário.
*
* A classe Endpoint tem um método de publicação sobrecarregado.
* Nesta versão de dois argumentos, o primeiro argumento é a
* URL de publicação como uma string e o segundo argumento é
* uma instância do SIB de serviço, neste caso
* ch01.ts.TimeServerImpl.

* A aplicação roda indefinidamente, esperando solicitações de serviço.
* Ela precisa ser finalizada no prompt de comando com control-C
* ou o equivalente.

* Depois da aplicação ser iniciada, abra um navegador para a URL.
*
```

```
 * http://127.0.0.1:9876/ts?wsdl
 *

 * Para visualizar o contrato de serviço, use o documento WSDL. Este é um
 * teste fácil para determinar se o serviço foi implementado com
 * sucesso. Se o teste obtiver sucesso, um cliente então pode ser
 * executado no serviço.
*/
public class TimeServerPublisher {
  public static void main(String[ ] args) {
    //primeiro argumento é a URL de publicação
    //segundo argumento é uma instância SIB
    Endpoint.publish("http://127.0.0.1:9876/ts", new TimeServerImpl());
  }
}
```

Depois de compilado, o publicador (publisher) pode ser executado de maneira usual:

```
% java ch01.ts.TimeServerPublisher
```

Como o Publicador (Publisher) de Endpoint Manipula Requisições

Out of the Box (fora da caixa), o publicador `Endpoint` lida com uma solicitação de cliente por vez. Isto é o suficiente para colocar os web services em funcionamento em modo de desenvolvimento. Porém, se o processamento de uma dada solicitação deve esperar, então todas as solicitações de outros clientes estão efetivamente bloqueadas. Um exemplo, no final deste capítulo, mostra como `Endpoint` pode gerenciar solicitações ao mesmo tempo para que uma solicitação em espera não bloqueie as outras.

Testando o Web Service com um Navegador

Nós podemos testar o serviço implementado, abrindo um navegador e visualizando o documento *WSDL* (Linguagem de Definição de Web Service), que é um contrato de serviço gerado automaticamente. O navegador é aberto para uma URL que tem duas partes. A primeira parte é a URL publicada na aplicação Java `TimeServerPublisher`: *http://127.0.0.1:9876/ts*. Anexa a esta URL é a string de consulta ?wsdl em letras maiúsculas, minúsculas ou mistas. O resultado é *http://127.0.0.1:9876/ts?wsdl*. Exemplo 1-4 é o documento WSDL que o navegador exibe.

Exemplo 1-4. Documento WSDL para o serviço TimeServer

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://ts.ch01/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    targetNamespace="http://ts.ch01/"
    name="TimeServerImplService">
  <types></types>

  <message name="getTimeAsString"></message>
  <message name="getTimeAsStringResponse">
```

```
    <part name="return" type="xsd:string"></part>
  </message>
  <message name="getTimeAsElapsed"></message>
  <message name="getTimeAsElapsedResponse">
    <part name="return" type="xsd:long"></part>
  </message>

  <portType name="TimeServer">
    <operation name="getTimeAsString" parameterOrder="">
     <input message="tns:getTimeAsString"></input>
     <output message="tns:getTimeAsStringResponse"></output>
    </operation>
    <operation name="getTimeAsElapsed" parameterOrder="">
     <input message="tns:getTimeAsElapsed"></input>
     <output message="tns:getTimeAsElapsedResponse"></output>
    </operation>
  </portType>

  <binding name="TimeServerImplPortBinding" type="tns:TimeServer">
    <soap:binding style="rpc"
         transport="http://schemas.xmlsoap.org/soap/http">
    </soap:binding>
    <operation name="getTimeAsString">
     <soap:operation soapAction=""></soap:operation>
     <input>
       <soap:body use="literal" namespace="http://ts.ch01/"></soap:body>
     </input>
     <output>
       <soap:body use="literal" namespace="http://ts.ch01/"></soap:body>
     </output>
  </operation>
  <operation name="getTimeAsElapsed">
     <soap:operation soapAction=""></soap:operation>
     <input>
       <soap:body use="literal" namespace="http://ts.ch01/"></soap:body>
     </input>
     <output>
       <soap:body use="literal" namespace="http://ts.ch01/"></soap:body>
     </output>
    </operation>
  </binding>

  <service name="TimeServerImplService">
       <port name="TimeServerImplPort" binding="tns:TimeServerImplPortBinding">
       <soap:address location="http://localhost:9876/ts"></soap:address>
    </port>
  </service>
</definitions>
```

O Capítulo 2 examina o WSDL em detalhes e introduz utilitários Java associados com o contrato de serviço. Por enquanto, duas seções do WSDL (ambas exibidas em negrito) merecem uma rápida olhada. A seção `portType`, próxima ao topo, agrupa as operações que o web service oferece, neste caso as operações `getTimeAsString` e `getTimeAsElapsed`, que são os dois métodos Java declarados na SEI e implementados no SIB. O `portType` do WSDL

é como uma interface Java em que o `portType` apresenta as operações de serviço de forma abstrata, mas não oferece nenhum detalhe de implementação. Cada operação no web service consiste de uma mensagem `input` e `output`, onde input significa *entrada para web service*. Em tempo de execução, cada mensagem é um documento SOAP. A outra seção WSDL de interesse é a última, a seção `service` e, em particular, o serviço `location`, neste caso a URL *http://localhost:9876/ts*. A URL é chamada de *service endpoint* e informa os clientes sobre onde o serviço pode ser acessado.

O documento WSDL é útil para criar e executar clientes em um web service. Várias linguagens possuem utilitários para gerar código de suporte ao cliente a partir de um WSDL. O utilitário Java é agora chamada de *wsimport*, mas os nomes mais antigos, *wsdl2java* e *java2wsdl*, eram mais descritivos. Em tempo de execução, um cliente pode consumir o documento WSDL, associado com um web service, para obter informações importantes sobre os tipos de dados associados com as operações incluídas no serviço. Por exemplo, um cliente pode determinar a partir do nosso primeiro WSDL que a operação `getTimeAsElapsed` retorne um inteiro e não espera argumentos.

O WSDL também pode ser acessado com vários utilitários como *curl*. Por exemplo, o comando:

```
% curl http://localhost:9876/ts?wsdl
```

também exibe o WSDL.

Evitando um Problema Sutil na Implementação do Web Service

Este exemplo parte da prática muito comum de ter o SIB do serviço (a classe TimeServerImpl) conectado à SEI (a interface TimeServer) apenas através do atributo endpointInterface na anotação @WebService. Não é incomum ver isto:

```
@WebService(endpointInterface = "ch01.ts.TimeServer")
public class TimeServerImpl { // implements TimeServer removed
```

O estilo é popular mas não é seguro. É muito melhor ter a cláusula implements para que o compilador verifique se o SIB implementa os métodos declarados na SEI. Remova a cláusula implements e comente as definições das duas operações web service:

```
@WebService(endpointInterface = "ch01.ts.TimeServer")
public class TimeServerImpl {
    // public String getTimeAsString() { return new Date().toString();
    }
    // public long gettimeAsElapsed() { return new Date().getTime();
    }
}
```

O código ainda compila. Com a cláusula implements configurada, o compilador envia um erro fatal, porque o SIB falha, ao definir os métodos declarados na SEI.

Um Solicitante Perl e Ruby do Web Service

Para ilustrar a transparência da linguagem de web services, o primeiro cliente no web service baseado em Java não está em Java, mas sim em Perl. O segundo cliente está em Ruby. Exemplo 1-5 está no cliente Perl.

Exemplo 1-5. Cliente Perl para o cliente TimeServer

```
#!/usr/bin/perl -w

use SOAP::Lite;
my $url = 'http://127.0.0.1:9876/ts?wsdl';
my $service = SOAP::Lite->service($url);

print "\nCurrent time is: ", $service->getTimeAsString();
print "\nElapsed milliseconds from the epoch: ", $service->getTimeAsElapsed();
```

Em uma execução de amostra, o resultado foi:

```
Current time is: Thu Oct 16 21:37:35 CDT 2008
Elapsed milliseconds from the epoch: 1224211055700
```

O módulo Perl `SOAP:: Lite` oferece a funcionalidade que permite ao cliente enviar a solicitação SOAP apropriada e processe a resposta SOAP resultante. A URL de solicitação, a mesma URL usada para testar o web service no navegador, termina com uma string de consulta, que solicita o documento WSDL. O cliente Perl obtém o documento WSDL a partir do qual a biblioteca `SOAP:: Lite,` então gera o objeto de serviço apropriado (em sintaxe Perl, a variável escalar `$service)`. Ao consumir o documento WSDL, a biblioteca `SOAP:: Lite` obtém as informações necessárias, em particular, os nomes das operações do web service e tipos de dados envolvidos nestas operações. A Figura 1-2 descreve a arquitetura.

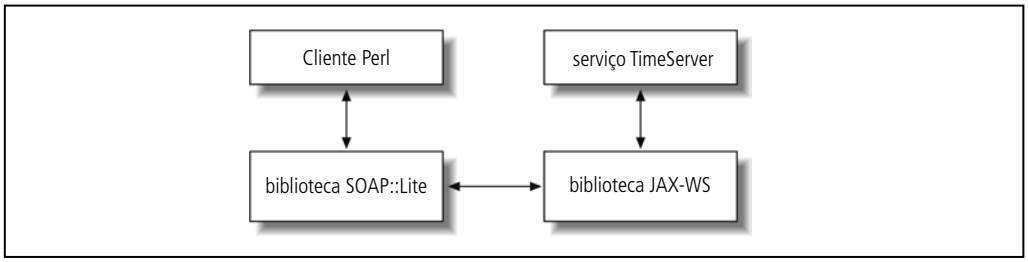

Figura 1-2. Arquitetura do cliente Perl e serviço Java

Depois da configuração, o cliente Perl chama as operações web service sem nenhuma confusão. As mensagens SOAP continuam não sendo vistas.

Exemplo 1-6 é um cliente Ruby que é funcionalmente equivalente ao cliente Perl.

Exemplo 1-6. Cliente Ruby para o cliente TimeServer

```
#!/usr/bin/ruby

# Um pacote Ruby para serviços baseados em SOAP
require 'soap/wsdlDriver'

wsdl_url = 'http://127.0.0.1:9876/ts?wsdl'

service = SOAP::WSDLDriverFactory.new(wsdl_url).create_rpc_driver

# Salva solicitação/resposta em arquivos chamados '...soapmsgs...'
service.wiredump_file_base = 'soapmsgs'

# Chama operações de serviço
result1 = service.getTimeAsString
result2 = service.getTimeAsElapsed

# Exibir resultados
puts "Current time is: #{result1}"
puts "Elapsed milliseconds from the epoch: #{result2}"
```

O SOAP Oculto

Em web services baseados em SOAP, um cliente tipicamente faz uma chamada de procedimento remoto no serviço invocando uma das operações do web service. Como mencionado anteriormente, esta ida e vinda entre o cliente e o serviço é o padrão de troca de mensagem solicitação/resposta, e as mensagens SOAP trocadas neste padrão permitem que o web service e um consumidor sejam programados em linguagens diferentes. Agora observamos mais atentamente o que acontece no nosso primeiro exemplo. O cliente Perl gera uma solicitação HTTP, que é uma mensagem formatada, cujo *corpo* é uma mensagem SOAP. Exemplo 1-7 é a solicitação HTTP de um exemplo de execução.

Exemplo 1-7. Solicitação HTTP para o service TimeServer

```
POST http://127.0.0.1:9876/ts HTTP/ 1.1
Accept: text/xml
Accept: multipart/*
Accept: application/soap
User-Agent: SOAP::Lite/Perl/0.69
Content-Length: 434
Content-Type: text/xml; charset=utf-8
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
    soap:encodingStyle="http:// schemas.xmlsoap.org/soap/encoding/"
    xmlns:soap="http://schemas.xmlsoap.org/soap/ envelope/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tns="http://ts.ch01/"
```

```
    xmlns:xsd ="http://www.w3.org/2001/XMLSchema">
   <soap:Body>
    <tns:getTimeAsString xsi:nil="true" />
   </soap:Body>
</soap:Envelope>
```

A solicitação HTTP é uma mensagem com uma estrutura própria. Em particular:

- A *linha inicial* HTTP vem primeiro e especifica o método de solicitação, neste caso o método POST, que é típico de solicitações para recursos dinâmicos, como web services ou outro código de aplicação web (por exemplo, um servlet Java) como o oposto de solicitações para uma página HTML estática. Neste caso, uma solicitação POST, ao invés de uma GET, é necessária porque apenas uma solicitação POST tem um corpo, que encapsula a mensagem SOAP. A seguir vem a URL de solicitação pela versão HTTP, neste caso 1.1, que o solicitante entende. HTTP 1.1 é a versão atual.
- A seguir vem os *headers* (cabeçalhos) HTTP, que são os pares chave/valor, nos quais dois pontos (:) separam a chave do valor. A ordem dos pares chave/valor é arbitrária. A chave *Accept* ocorre três vezes, como um tipo/subtipo *MIME* (Multipurpose Internet Mail Extencions) como o valor: `text/xml, multipart/*,` e `application/soap`. Estes três pares sinalizam que o solicitante está pronto para aceitar uma resposta XML arbitrária, uma resposta com arbitrariamente muitos anexos de qualquer tipo (uma mensagem SOAP pode ter arbitrariamente muitos anexos), e um documento SOAP, respectivamente. A chave HTTP `SOAPAction` está frequentemente presente no cabeçalho HTTP de uma solicitação web service e o valor da chave pode ser a string vazia, como neste caso; mas o valor também pode ser o nome da operação web service solicitada.
- Dois caracteres *CRLF* (Retorno de Carro com Avanço de Linha), que correspondem a dois caracteres Java \n, separam os cabeçalhos HTTP do corpo HTTP, que é exigido para o verbo POST mas pode estar vazio. Neste caso, o corpo HTTP contém o documento SOAP, comumente chamada de envelope SOAP, porque o elemento mais externo ou *document* é chamado de *Envelope*. Neste envelope SOAP, o corpo SOAP contém um único elemento cujo *nome local (local name)* é `getTimeAsString`, que é o nome da operação web service que o cliente quer solicitar. O envelope de solicitação SOAP é simples neste exemplo, porque a operação solicitada não requer argumentos.

No lado do web service, as bibliotecas Java fundamentais processam a solicitação HTTP, extraem o envelope SOAP, determinam a identidade da operação de serviço solicitada, chamam o método Java `getTimeAsString` correspondente e, então, geram a mensagem SOAP apropriada para levar o valor de retorno do método de volta para o cliente. Exemplo 1-8 é a resposta HTTP da solicitação de serviço Java `TimeServerImpl` mostrada no Exemplo 1-7.

Exemplo 1-8. Resposta HTTP do serviço TimeServer

```
HTTP/1.1 200 OK
Content-Length: 323
Content-Type: text/xml; charset=utf-8
```

```
Client-Date: Mon, 28 Apr 2008 02:12:54 GMT
Client-Peer: 127.0.0.1:9876
Client-Response-Num: 1

<?xml version="1.0" ?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <soapenv:Body>
    <ans:getTimeAsStringResponse xmlns:ans="http://ts.ch01/">
      <return>Mon Apr 28 14:12:54 CST 2008</return>
    </ans:getTimeAsStringResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

Novamente o envelope SOAP é o corpo de uma mensagem HTTP, neste caso, a resposta HTTP ao cliente. A *linha inicial* HTTP agora contém o código de status como o inteiro 200 e o texto correspondente `OK`, que sinaliza que a solicitação do cliente foi gerenciada com sucesso. O envelope SOAP no corpo da resposta HTTP contém a hora atual como uma string entre as tags XML de início e fim chamadas de *return*. A biblioteca SOAP Perl extrai o envelope SOAP da resposta HTTP e, por causa de informações no documento WSDL, espera que o valor de retorno desejado da operação web service ocorra no elemento XML *return*.

Um Solicitante Java de Web Service

Exemplo 1-9 é um cliente Java funcionalmente equivalente aos clientes Perl e Ruby exibidos nos Exemplos 1-5 e 1-6, respectivamente.

Exemplo 1-9. Cliente Java para o Java web service

```
package ch01.ts;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import java.net.URL;
class TimeClient {
    public static void main(String args[ ]) throws Exception {
       URL url = new URL("http://localhost:9876/ts?wsdl");

       //Nome qualificado do serviço
       //Primeiro argumento é o URI do serviço
       //Segundo é o nome do serviço publicado no WSDL
       QName qname = new QName("http://ts.ch01/", "TimeServerImplService");

       //Cria, de fato, uma fábrica para o serviço
       Service service = Service.create(url, qname);

       //Extrai a interface endpoint, o serviço "port".
       TimeServer eif = service.getPort(TimeServer.class);
```

```
        System.out.println(eif.getTimeAsString());
        System.out.println(eif.getTimeAsElapsed());
    }
}
```

O cliente Java usa a mesma URL com uma string de consulta como fazem os clientes Perl e Ruby, mas o cliente Java explicitamente cria um *nome qualificado* XML, que tem a sintaxe *namespace URI:local name*. Um *URI* é um Identificador de Recurso Uniforme em inglês (Uniform Resource Identifier) e difere da comum URL, que especifica uma *localização,* porque URI não precisa especificar uma localização. Resumindo, um URI não precisa ser uma URL. Por enquanto, é suficiente entender que a classe Java `java.xml.namespace.QName` representa um nome qualificado XML. Neste exemplo, o namespace URI é fornecido no WSDL, e o nome local é o nome de classe SIB `TimeServerImpl` com a palavra Service anexada. O nome local ocorre na seção service, a última seção do documento WSDL.

Depois dos objetos `URL` e `QName` serem construídos e do método *Service.create* ser chamado, a expressão que interessa:

```
TimeServer eif = service.getPort(TimeServer.class);
```

executa. Lembre que, no documento WSDL, a seção `portType` descreve, no estilo de uma interface, as operações incluídas no web service. O método `getPort` retorna uma referência para um objeto Java que pode chamar as operações `portType`. A referência de objeto `eif` é do tipo `ch01.ts.TimeServer`, que o tipo SEI. O cliente Java, como o cliente Perl, chama os dois métodos web service; e as bibliotecas Java, como as bibliotecas Perl e Ruby, geram e processam as mensagens SOAP trocadas transparentemente, para possibilitar as solicitações de método com sucesso.

Monitoramento de Baixo Nível em Mensagens HTTP e SOAP

Exemplo 1-7 e Exemplo 1-8 mostram uma mensagem de solicitação HTTP e uma mensagem de resposta HTTP, respectivamente. Cada mensagem HTTP encapsula um envelope SOAP. Estes rastros de mensagem foram feitos com o cliente Perl alterando a diretriz Perl *use,* no Exemplo 1-5:

```
use SOAP::Lite;
```

para

```
use SOAP::Lite +trace;
```

O cliente Ruby no Exemplo 1-6 contém uma linha:

```
service.wiredump_file_base = 'soapmsgs'
```

que faz os envelopes SOAP serem salvos em arquivos no disco local. É possível capturar o tráfego de baixo nível diretamente em Java também, como exemplos futuros ilustram. Várias opções estão disponíveis para monitorar mensagens SOAP e HTTP no baixo nível. Aqui está uma pequena introdução de algumas delas.

O utilitário *tcpmon* (disponível em *https://tcpcmon.dev.java.net*) é free e é baixado como um arquivo JAR executável. Sua interface gráfica de usuário (GUI) é fácil de usar. O utilitário exige apenas três configurações: o nome do servidor, que no padrão é `localhost`; a porta do servidor, que deve ser configurada em `9876` para o exemplo `TimeServer`, porque esta é a porta na qual `Endpoint` publica o serviço; e a porta local, que, por padrão, é `8080` e é a porta que *tcpmon* escuta. Com *tcpmon* configurada, o `TimeClient` vai enviar suas solicitações à porta `8080`, ao invés da porta `9876`. O utilitário *tcpmon* intercepta o tráfego HTTP entre o cliente e web service, exibindo as mensagens completas na sua GUI.

O release do Metro tem classes de utilitários para monitorar tráfego HTTP e SOAP. Esta abordagem não exige nenhuma alteração no código do serviço ou cliente; porém, um pacote adicional deve ser incluído no classpath e uma propriedade de sistema deve ser configurada na linha de comando ou em código. O pacote exigido está no arquivo *jaxws-ri/jaxws-rt.jar*. Supondo que a variável de ambiente `METRO_HOME` aponta para o diretório *jaxws-ri*, aqui está o comando que monitora o tráfego HTTP e SOAP entre o `TimeClient`, que conecta o serviço na porta 9876, e o serviço `TimeServer`. (Em Windows, `$METRO_HOME` se torna `%METRO_HOME%)`. O comando está em três linhas para legibilidade:

```
% java -cp ".":$METRO_HOME/lib/jaxws-rt.jar \
  -Dcom.sun.xml.ws.transport.http.client.HttpTransportPipe.dump=true \
  ch01.ts.TimeClient
```

O dump resultante mostra todo o tráfego SOAP, mas não todos os cabeçalhos HTTP. Monitoramento de mensagem também pode ser feito no lado do servidor.

Há muitos outros produtos comerciais e open source disponíveis para monitoramento de tráfego. Entre os produtos que valem a pena conferir estão SOAPscope (*http://www.home.mindreef.com*), NetSniffer (*http://www.miray.de*) e Wireshark (*http://www.wireshark.org*). O utilitário *tcpdump* vem com a maioria dos sistemas do tipo Unix, incluindo Linux e OS X, e está disponível em Windows, como WinDump (*http://www.winpcap.org*). Além de ser gratuito, *tcpdump* não é intrusivo, pois não exige nenhuma alteração em um web service ou um cliente. O utilitário *tcpdump* faz o dump do tráfego de mensagem para a saída padrão. O utilitário companheira *tcptrace* (*http://www.tcptrace.org*) pode ser usada para analisar o dump. O restante desta seção aborda brevemente *tcpdump* como o utilitário de monitoramento flexível e poderoso.

Em sistemas do tipo Unix, o utilitário *tcpdump* tipicamente deve ser executado com o *superusuário*. Há muitos argumentos assinalados que determinam como o utilitário funciona. Aqui está uma chamada simples:

```
% tcpdump -i lo -A -s 1024 -l 'dst host localhost and port 9876' | tee
  dump.log
```

O utilitário pode capturar pacotes em qualquer interface de rede. Uma lista de tais interfaces está disponível com o *tcpdump-D* (em Windows, WinDump-D), que é equivalente ao comando *ifconfig-a* em sistemas como Unix). Neste exemplo, o par flag/valor `-i lo` significa *capturar pacotes da interface lo*, sendo *lo* o diminutivo da interface de rede localhost em muitos sistemas como Unix. O flag `-A` significa que os pacotes capturados devem ser

apresentados em ASCII, que é útil para pacotes web, pois estes tipicamente contêm texto. Os flags `-s 1024` configuram o *tamanho da cópia*, o número de bytes que deve ser capturado de cada pacote. A flag *-1* força o a saída padrão a ser bufferizado em linha e mais fácil de ser lido; e, no mesmo tema, a construção `| tee dump.log` nos pipes finais a mesma saída exibe na tela (a saída padrão) em um arquivo local chamado *dump.log*. Finalmente, a expressão:

```
'dst host localhost and port 9876'
```

age como um filtro, capturando apenas pacotes cujo destino seja *localhost* na porta 9876, a porta na qual `TimeServerPublisher` do Exemplo 1-3 publica o serviço `TimeServer`.

O utilitário *tcpdump* e a aplicação `TimeServerPublisher` pode ser iniciada em qualquer ordem. Depois que ambas estiverem rodando, o `TimeClient` ou um dos outros clientes pode ser executado. Com o exemplo de uso de *tcpdump* mostrado acima, os pacotes fundamentais de rede são salvos no arquivo *dump.log*. O arquivo não exige editoração para tornálo facilmente legível. Em qualquer caso, o arquivo *dump.log* captura os mesmos envelopes SOAP exibidos nos Exemplos 1-7 e 1-8.

O Que Está Claro Até Agora?

O primeiro exemplo é um web service com duas operações, cada uma oferece a hora atual mas em diferentes representações: em um caso é uma string legível por humanos, em outro caso, como os milésimos de segundo decorridos da época Unix. As duas operações são implementadas como independentes, métodos autocontidos. Da perspectiva do solicitador do serviço, qualquer método pode ser solicitado independentemente do outro e uma solicitante de um método de serviço não tem impacto em nenhuma solicitação subsequente do mesmo método de serviço. Os dois métodos Java não dependem nem um do outro nem de qualquer outro campo de instância para qual ambos têm acesso; na verdade, a classe SIB *TimeServerImpl* não tem absolutamente nenhum campo. Resumindo, as duas solicitações de método são stateless (sem estado).

No primeiro exemplo, nenhum método espera argumentos. Em geral, operações de web service podem ser parametrizadas para que a informação apropriada possa ser passada para a operação como parte da solicitação de serviço. Independente se as operações web service forem parametrizadas, elas ainda devem aparecer ao solicitante como independente e autocontidas. Este princípio de design irá guiar todos os exemplos que consideramos, até aqueles mais ricos que o primeiro.

Características Importantes do Primeiro Código de Exemplo

A classe *TimeServerImpl* implementa um web service com um padrão distintivo de troca de mensagem (MEP) – solicitação/resposta. O serviço permite que um cliente faça uma chamada de procedimento remoto de linguagem neutra, chamando os métodos `getTimeAsString` e `getTimeAsElapsed`. Outros padrões de mensagem são possíveis. Imagine, por exemplo, um web service que monitora novas quantidades de neve para áreas de esqui. Alguns clientes participantes, talvez aparelhos elétricos de medição de neve estrategicamente colocados ao redor das ladeiras de neve, podem usar o padrão, enviando uma

quantidade de neve a partir de uma localização particular, mas sem esperar uma resposta do serviço. O serviço pode exibir o padrão de notificação múltipla, distribuindo aos clientes assinantes (por exemplo, escritórios de viagem) informações sobre atuais condições de neve. Finalmente, o serviço pode periodicamente usar o padrão solicitação/resposta para perguntar a um cliente assinante se ele desejar continuar a receber notificações. Em resumo, web services baseados em SOAP suportam vários padrões. O padrão solicitação/resposta de RPC continua sendo o dominante. A infraestrutura necessária para suportar este padrão em particular vale a pena resumir:

Transporte de mensagem (Message Transport)

SOAP é designado para servir de transporte neutro, um objetivo de design que complica tudo porque mensagens SOAP não podem confiar em informações específicas ao protocolo incluídas na infraestrutura de transporte. Por exemplo, SOAP fornecido através de HTTP não deve se diferenciar de SOAP fornecido através de algum outro protocolo de transporte como *SMTP (Simple Mail Transfer Protocol)*. Na prática, porém, HTTP é o transporte comum para serviços baseados em SOAP, um ponto enfatizado no nome comum: ***web*** services baseados em SOAP.

Contrato de serviço (Service Contract)

O cliente de serviço exige informações sobre as operações de serviço para solicitá-las. Em particular, o cliente precisa de informações sobre a sintaxe de solicitação: o nome da operação, a ordem e tipos de argumentos passados para a operação e o tipo de valor retornado. O cliente também exige o endpoint do serviço, tipicamente a URL do serviço. O documento WSDL fornece estas partes de informação e outras. Embora um cliente possa chamar um serviço sem acessar primeiro o WSDL, isto pode tornar as coisas mais difíceis do que precisavam ser.

Sistema de tipo (Type System)

A chave para a neutralidade da linguagem e, então, interoperabilidade serviço/consumidor é um sistema de tipo compartilhado para que os tipos de dados usados na solicitação do cliente coordenem com os tipos usados na operação do serviço. Considere um simples exemplo. Suponha que um Java web service tenha a operação:

```
boolean bytes_ok(byte[ ] some_bytes)
```

A operação `bytes_ok` realiza alguns testes de validação nos bytes passados para a operação como um argumento do tipo, retornando `true` ou `false`. Agora suponha que um cliente escrito em C precise chamar `bytes_ok`. C não tem tipos nomeados `boolean` e `byte`. C representa valores boolean com inteiros com não-zero, como `true`, e zero, como `false`; e o tipo C `char com sinal` corresponde ao tipo Java `byte`. Um web service pode ser complicado de consumir, se clientes tiverem que mapear tipos de linguagem do cliente para tipos de linguagem de serviço. Em web services baseados em SOAP, o sistema de tipo de Esquema XML (XML Schema) o sistema de tipo padrão que media entre os tipos do cliente e os tipos do serviço. No exemplo acima, o tipo de Esquema XML `xsd:byte` é o tipo que media entre o C `char com sinal` e o Java `byte`; o tipo Esquema XML `xsd:boolean` é o tipo de mediação para os inteiros C não-zero e zero os valores boolean Java `true` e `false`. Na notação `xsd:byte`, o prefixo `xsd`

(XML Schema Definition em português, Definiçaõ de Esquema XML) enfatiza que este é um tipo de Esquema XML, porque `xsd` é a extensão usual para um arquivo que contém uma Definição de Esquema XML; por exemplo, *purchaseOrder.xsd*.

API SOAP do Java

Um grande apelo de web services baseados em SOAP é que SOAP geralmente continua oculto. Apesar de tudo, pode ser útil dar uma olhada no suporte fundamental de Java para gerar e processar mensagens SOAP. O Capítulo 3, que introduz handlers (manipuladores) SOAP, põe a API SOAP em uso prático. Esta seção oferece um primeiro olhar para a API SOAP através de um exemplo de simulação. A aplicação consiste de uma classe, `DemoSOAP`, mas simula o envio de uma mensagem SOAP como uma solicitação e o recebimento de outra como uma resposta. Exemplo 1-10 mostra a aplicação completa.

Exemplo 1-10. Uma demonstração de API SOAP de Java

```
package ch01.soap;

import java.util.Date;
import java.util.Iterator;
import java.io.InputStream;
import java.io.ByteArrayOutputStream;
import java.io.ByteArrayInputStream;
import java.io.IOException;
import javax.xml.soap.MessageFactory;
import javax.xml.soap.SOAPMessage;
import javax.xml.soap.SOAPEnvelope;
import javax.xml.soap.SOAPHeader;
import javax.xml.soap.SOAPBody;
import javax.xml.soap.SOAPPart;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPException;
import javax.xml.soap.Node;
import javax.xml.soap.Name;

public class DemoSoap {
    private static final String LocalName = “TimeRequest”;
    private static final String Namespace = “http://ch01/mysoap/”;
    private static final String NamespacePrefix = “ms”;

    private ByteArrayOutputStream out;
    private ByteArrayInputStream in;

    public static void main(String[ ] args) {
      new DemoSoap().request();
    }

    private void request() {
      try {
        // Criar uma mensagem SOAP para enviar para um stream de saída
        SOAPMessage msg = create_soap_message();
```

```
        //Injetar a informação apropriada na mensagem
        //Neste caso, apenas o cabeçalho da mensagem (opcional) é usado
        //e o corpo está vazio.
        SOAPEnvelope env = msg.getSOAPPart().getEnvelope();
        SOAPHeader hdr = env.getHeader();

        // Adicionar um elemento ao cabeçalho SOAP.
        Name lookup_name = create_qname(msg);
        hdr.addHeaderElement(lookup_name).addTextNode("time_request");

        // Simular o envio da mensagem SOAP ao sistema remoto
        // escrevendo-o em um ByteArrayOutputStream.
        out = new ByteArrayOutputStream();
        msg.writeTo(out);

        trace("The sent SOAP message:", msg);

        SOAPMessage response = process_request();
        extract_contents_and_print(response);
    }
    catch(SOAPException e) { System.err.println(e); }
    catch(IOException e) { System.err.println(e); }
}

private SOAPMessage process_request() {
    process_incoming_soap();
    coordinate_streams();
    return create_soap_message(in);
}

private void process_incoming_soap() {
    try {
    // Copiar stream de output em stream de input para simular
    //streams coordenadas através de uma conexão de rede.
    coordinate_streams();

    // Criar a mensagem SOAP "recebida" a partir do
    // stream de entrada..
    SOAPMessage msg = create_soap_message(in);

    // Inspecionar o cabeçalho SOAP para a palavra-chave 'time-request'
    // e processar a solicitação se a palavra-chave ocorrer.
    Name lookup_name = create_qname(msg);

    SOAPHeader header = msg.getSOAPHeader();
    Iterator it = header.getChildElements(lookup_name);
    Node next = (Node) it.next();
    String value = (next == null) ? "Error!" : next.getValue();

    // Se a mensagem SOAP contiver solicitação para a hora, criar uma
    // nova mensagem SOAP com a hora atual no corpo.
    if (value.toLowerCase().contains("time_request")) {
```

```
    // Extrair o corpo e adicionar a hora atual como um elemento.
    String now = new Date().toString();
    SOAPBody body = msg.getSOAPBody();
    body.addBodyElement(lookup_name).addTextNode(now);
    msg.saveChanges();

    // Escrever na stream de saída.
    msg.writeTo(out);
    trace("The received/processed SOAP message:", msg);
   }

  }
  catch(SOAPException e) { System.err.println(e); }
  catch(IOException e) { System.err.println(e); }
}

private void extract_contents_and_print(SOAPMessage msg) {
  try {
    SOAPBody body = msg.getSOAPBody();

    Name lookup_name = create_qname(msg);
    Iterator it = body.getChildElements(lookup_name);
    Node next = (Node) it.next();

    String value = (next == null) ? "Error!" : next.getValue();
    System.out.println("\n\nReturned from server: " + value);
  }
  catch(SOAPException e) { System.err.println(e); }
}

private SOAPMessage create_soap_message() {
  SOAPMessage msg = null;
  try {
    MessageFactory mf = MessageFactory.newInstance();
    msg = mf.createMessage();
  }
  catch(SOAPException e) { System.err.println(e); }
  return msg;
}

private SOAPMessage create_soap_message(InputStream in) {
  SOAPMessage msg = null;
  try {
    MessageFactory mf = MessageFactory.newInstance();
    msg = mf.createMessage(null, // ignore cabeçalhos MIME
                           in); // stream de origem
  }
  catch(SOAPException e) { System.err.println(e); }
  catch(IOException e) { System.err.println(e); }
  return msg;
}

private Name create_qname(SOAPMessage msg) {
  Name name = null;
```

```
   try {
     SOAPEnvelope env = msg.getSOAPPart().getEnvelope();
     name = env.createName(LocalName, NamespacePrefix, Namespace);
    }
    catch(SOAPException e) { System.err.println(e); }
    return name;
   }
private void trace(String s, SOAPMessage m) {
    System.out.println("\n");
    System.out.println(s);
    try {
    m.writeTo(System.out);
    }
    catch(SOAPException e) { System.err.println(e); }
    catch(IOException e) { System.err.println(e); }
   }
private void coordinate_streams() {
     in = new ByteArrayInputStream(out.toByteArray());
     out.reset();
   }
}
```

Aqui está um resumo de como a aplicação funciona, com ênfase no código envolvendo mensagens SOAP. O método `request` da aplicação `DemoSoap` gera uma mensagem SOAP e adiciona a string `time_request` ao cabeçalho do envelope SOAP. O segmento de código, com comentários removidos, é:

```
SOAPMessage msg = create_soap_message();
SOAPEnvelope env = msg.getSOAPPart().getEnvelope();
SOAPHeader hdr = env.getHeader();
Name lookup_name = create_qname(msg);
hdr.addHeaderElement(lookup_name).addTextNode("time_request");
```

Há duas maneiras básicas de criar uma mensagem SOAP. A maneira mais simples é ilustrada neste segmento de códi,go:

```
MessageFactory mf = MessageFactory.newInstance();
SOAPMessage msg = mf.createMessage();
```

Na maneira mais complicada, o código `MessageFactory` é o mesmo, mas a chamada de criação tornar-se:

```
SOAPMessage msg = mf.createMessage(mime_headers, input_stream);
```

O primeiro argumento para `createMessage` é uma coleção dos cabeçalhos de camada de transporte (por exemplo, os pares chave/valor que constituem um cabeçalho HTTP), e o segundo argumento é um stream de entrada que fornece os bytes para criar a mensagem (por exemplo, o stream de entrada encapsulado em uma instância de um `Socket` java).

Depois que a mensagem SOAP é criada, o cabeçalho é extraído do envelope SOAP e um nó (node) de texto XML é inserido com o valor `time_request`. A mensagem SOAP resultante é:

```
<SOAP-ENV:Envelope
     xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Header>
      <ms:TimeRequest xmlns:ms="http://ch01/mysoap/">
         time_request
      </ms:TimeRequest>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body/>
</SOAP-ENV:Envelope>
```

Não há necessidade de examinar imediatamente todos os detalhes desta mensagem SOAP. Aqui está um resumo de alguns pontos-chave. O corpo SOAP é sempre exigido mas, como neste caso, o corpo pode estar vazio. O cabeçalho SOAP é opcional mas, neste caso, o cabeçalho contém o texto `time_request`. Conteúdos de mensagem como `time_request` normalmente são colocados no corpo SOAP e informações especiais de processamento (por exemplo, dados de autenticação do usuário) são colocadas no cabeçalho. A questão aqui é ilustrar como o cabeçalho SOAP e o corpo SOAP podem ser manipulados.

O método `request` escreve a mensagem SOAP em um `ByteArrayOutputStream`, que simula o envio da mensagem através de uma conexão de rede para um receptor em um host diferente. O método `request` chama o método `process_request`, que, então, delega as tarefas restantes para outros métodos. O processamento segue. A mensagem SOAP recebida é criada a partir de um `ByteArrayInputStream`, que simula um stream de entrada no lado do receptor; este stream contém a mensagem SOAP enviada. A mensagem SOAP agora é criada a partir da stream de entrada:

```
SOAPMessage msg = null;
try {
  MessageFactory mf = MessageFactory.newInstance();
  msg = mf.createMessage(null, // ignore cabeçalhos MIME
                     in); // stream de origem (ByteArrayInputStream)
}
```

e então a mensagem SOAP é processada para extrair a string `time_request`. A extração ocorre como segue. Primeiro, o cabeçalho SOAP é extraído da mensagem SOAP e então há uma iteração sobre os elementos com o nome de tag:

```
<ms:TimeRequest xmlns:ms="http://ch01/mysoap"/>
```

é criado. Neste exemplo, há um elemento com este nome de tag e o elemento deve conter a string `time_request`. O código de busca é:

```
SOAPHeader header = msg.getSOAPHeader();
Iterator it = header.getChildElements(lookup_name);
Node next = (Node) it.next();
String value = (next == null) ? "Error!" : next.getValue();
```

Se o cabeçalho SOAP contiver a string de solicitação apropriada, o corpo SOAP é extraído da mensagem SOAP de entrada e um elemento contendo a hora atual como uma string é

adicionado ao corpo SOAP. A mensagem SOAP revisada então é enviada como uma resposta. Aqui está o segmento de código com os comentários removidos:

```
if (value.toLowerCase().contains("time_request")) {
 String now = new Date().toString();
 SOAPBody body = msg.getSOAPBody();
 body.addBodyElement(lookup_name).addTextNode(now);
 msg.saveChanges();

 msg.writeTo(out);
 trace("The received/processed SOAP message:", msg);
}
```

A mensagem SOAP de saída em uma execução de exemplo foi:

```
<SOAP-ENV:Envelope
      xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Header>
       <ms:TimeRequest xmlns:ms="http://ch01/mysoap/">
          time_request
       </ms:TimeRequest>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
       <ms:TimeRequest xmlns:ms="http://ch01/mysoap/">
          Mon Oct 27 14:45:53 CDT 2008
       </ms:TimeRequest>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Este exemplo fornece um primeiro olhar para a API de Java. Exemplos futuros ilustram o uso a nível de produção da API SOAP.

Um Exemplo com Tipos de Dados Mais Ricos

As operações no serviço `TimeServer` não requer argumentos e retornam tipos simples, uma string e um inteiro. Esta seção oferece um exemplo mais rico, cujos detalhes são esclarecidos no próximo capítulo.

O web service `Teams`, no Exemplo, 1-11 difere do serviço `TimeServer` em diversas maneiras importantes.

Exemplo 1-11. O web service de estilo de documento Teams

```
package ch01.team;

import java.util.List;
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class Teams {
   private TeamsUtility utils;
```

```
    public Teams() {
       utils = new TeamsUtility();
       utils.make_test_teams();
    }
    @WebMethod
    public Team getTeam(String name) { return utils.getTeam(name); }

    @WebMethod
    public List<Team> getTeams() { return utils.getTeams(); }
}
```

Por um lado, o serviço `Teams` é implementado como uma única classe Java ao invés de uma SEI e SIB separados. Isto é feito simplesmente para ilustrar a possibilidade. Uma diferença mais importante está nos tipos de retorno das duas operações `Teams`. A operação *getTeam* é parametrizada e retorna um objeto do tipo `Team` definido pelo programador, que é uma lista de instâncias de `Player`, outro tipo definido pelo programador. A operação `getTeams` retorna uma `List<Team>`, isto é, uma `Collection` Java.

A classe utilitária `TeamsUtility` gera os dados. Em um ambiente de produção, este utilitário pode recuperar um time ou lista de times (teams) de um banco de dados. Para tornar este exemplo simples, o utilitário cria os times (teams) e seus jogadores (players) rapidamente. Aqui está:

```
    package ch01.team;

    import java.util.Set;
    import java.util.List;
    import java.util.ArrayList;
    import java.util.Map;
    import java.util.HashMap;

    public class TeamsUtility {
        private Map<String, Team> team_map;

        public TeamsUtility() {
           team_map = new HashMap<String, Team>();
        }

        public Team getTeam(String name) { return team_map.get(name); }
        public List<Team> getTeams() {
           List<Team> list = new ArrayList<Team>();
           Set<String> keys = team_map.keySet();
           for (String key : keys)
             list.add(team_map.get(key));
           return list;
        }

        public void make_test_teams() {
           List<Team> teams = new ArrayList<Team>();
           ...
           Player chico = new Player("Leonard Marx", "Chico");
           Player groucho = new Player("Julius Marx", "Groucho");
```

```
        Player harpo = new Player("Adolph Marx", "Harpo");
        List<Player> mb = new ArrayList<Player>();
        mb.add(chico); mb.add(groucho); mb.add(harpo);
        Team marx_brothers = new Team("Marx Brothers", mb);
        teams.add(marx_brothers);

        store_teams(teams);

      }
      private void store_teams(List<Team> teams) {
        for (Team team : teams)
            team_map.put(team.getName(), team);
      }
   }
```

Publicando o Serviço e Escrevendo um Cliente

Lembre-se de que a SEI para o serviço `TimeServer` contém a anotação:

```
@SOAPBinding(style = Style.RPC)
```

Esta anotação exige que o serviço use apenas tipos muito simples como string e inteiro. Por contraste, o serviço *Teams* usa tipos de dados mais ricos, o que significa que `Style.DOCUMENT`, o padrão, deve substituir `Style.RPC`. O estilo do documento exige mais configuração, que é dada abaixo, mas não é explicada até o próximo capítulo. Aqui, então, estão os passos exigidos para empregar o web service e um exemplo de cliente escrito rapidamente:

1. Os arquivos fontes são compilados da maneira usual. A partir do diretório de trabalho, que tem *ch01* como um subdiretório, o comando é:

```
% javac ch01/team/*.java
```

Além da classe `Teams` anotada com `@WebService`, o diretório *ch01/team* contém as classes `Team`, `Player`, *TeamsUtility* e `TeamsPublisher` exibidas abaixo todas juntas:

```
package ch01.team;
public class Player {
   private String name;
   private String nickname;
   public Player() { }

   public Player(String name, String nickname) {
      setName(name);
       setNickname(nickname);
   }

   public void setName(String name) { this.name = name; }
   public String getName() { return name; }
   public void setNickname(String nickname) { this.nickname = nickname; }
   public String getNickname() { return nickname; }
}
```

```
// end of Player.java

package ch01.team;

import java.util.List;
public class Team {
    private List<Player> players;
    private String name;

    public Team() { }
    public Team(String name, List<Player> players) {
    setName(name);
    setPlayers(players);
    }

    public void setName(String name) { this.name = name; }
    public String getName() { return name; }
    public void setPlayers(List<Player> players) { this.players =
    players; }
    public List<Player> getPlayers() { return players; }
    public void setRosterCount(int n) { } // no-op but needed for pro-
    perty
    public int getRosterCount() { return (players == null) ? 0 :
    players.size(); }

    }
}
// end of Team.java

package ch01.team;
import javax.xml.ws.Endpoint;
class TeamsPublisher {
    public static void main(String[ ] args) {
       int port = 8888;
       String url = "http://localhost:" + port + "/teams";
       System.out.println("Publishing Teams on port " + port);
       Endpoint.publish(url, new Teams());
    }
}
```

2. No diretório de trabalho, chame o utilitário *wsgen*, que vem com Java 6:

```
% wsgen -cp . ch01.team.Teams
```

   Este utilitário gera vários *artefatos*; isto é, tipos Java necessários pelo método `Endpoint.publish` para gerar o WSDL do serviço. O Capítulo 2 analisa estes artefatos e como eles contribuem para o WSDL.
3. Execute a aplicação `TeamsPublisher`.
4. No diretório de trabalho, chame o utilitário *wsimport*, que também vem com Java 6:

```
% wsimport -p teamsC -keep http://localhost:8888/teams?wsdl
```

   Este utilitário gera várias classes no subdiretório *teamsC* (a flag `-p` significa `package`) Estas classes tornam mais fácil escrever um cliente para o serviço.

Passo 4 promove a codificação de um cliente, como é exibido aqui:

```
import teamsC.TeamsService;
import teamsC.Teams;
import teamsC.Team;
import teamsC.Player;
import java.util.List;

class TeamClient {
   public static void main(String[ ] args) {
       TeamsService service = new TeamsService();
       Teams port = service.getTeamsPort();
       List<Team> teams = port.getTeams();
       for (Team team : teams) {
          System.out.println("Team name: " + team.getName() +
                       " (roster count: " + team.getRosterCount() +
                       ")");
          for (Player player : team.getPlayers())
             System.out.println(" Player: " + player.getNickname());
       }
   }
}
```

Quando o cliente executa, a saída é:

```
Team name: Abbott and Costello (roster count: 2)
  Player: Bud
  Player: Lou
Team name: Marx Brothers (roster count: 3)
  Player: Chico
  Player: Groucho
  Player: Harpo
Team name: Burns and Allen (roster count: 2)
  Player: George
  Player: Gracie
```

Este exemplo mostra o que é possível em um web service baseado em SOAP e de nível comercial. Tipos definidos pelo programador como `Player` e `Team`, junto com coleções arbitrárias destes, podem ser argumentos passados para ou valores retornados de um web service como diretrizes a serem seguidas. Uma diretriz entra em ação neste exemplo. Para as classes `Team` e `Player`, as propriedades JavaBean são dos tipos `String` ou `int`; e um `List`, como qualquer `Collection` Java, tem um método `toArray`. No final, um `List<Team>` reduz a arrays de tipos simples; neste caso, instâncias `String` ou valores `int`. O próximo capítulo aborda detalhes, em particular, como os utilitários *wsgen* e *wsimport* facilitam o desenvolvimento de serviços e clientes JWS.

Tornando Multithread o Publicador (Publisher) Endpoint

Nos exemplos até agora, o publicador `Endpoint` tem sido single-thread e, então, capaz de lidar com uma requisição de cliente por vez: o serviço publicado completa o processamento de uma solicitação, antes de começar o processamento de outra. Se o processamento da solicitação atual ficar esperando, então nenhuma solicitação subsequente pode ser processada, a menos e até que a solicitação em espera seja completamente processada.

Em modo de produção, o publicador (publisher) `Endpoint` vai precisar lidar com solicitações concorrentes, para que várias solicitações pendentes possam ser processadas ao mesmo tempo. Se o sistema de computador subjacente for, por exemplo, um multiprocessador simétrico (SMP), então CPUs separadas podem processar diferentes solicitações ao mesmo tempo. Em uma máquina com uma única CPU, a concorrência vai ocorrer através de compartilhamento de tempo; isto é, cada solicitação obtém uma parte dos ciclos de CPU disponíveis, para que diversas solicitações estejam em algum estágio de processamento a qualquer momento. Em Java, concorrência é obtida através de multithread. A questão é, então, como tornar o publicador *Endpoint* multithread. O framework JWS suporta multithread do *Endpoint* sem forçar o programador a trabalhar com construções difíceis e propensas ao erro, como o bloco `synchronized` ou as chamadas dos métodos `wait` e `notify`.

Um objeto `Endpoint` tem uma propriedade Executor definida com os métodos padrões `get/set`. Um `Executor` é um objeto que executa tarefas `Runnable`; por exemplo, instâncias Java *Thread* padrões. (A interface `Runnable` declara apenas um método, cuja declaração é `public avoid run ().`) Um objeto `Executor` é uma boa alternativa para instâncias `Thread`, pois o `Executor` oferece construções de alto nível para submeter e gerenciar tarefas que devem ser executadas concorrentemente. O primeiro passo para tornar o publicador `Endpoint` multithread é criar uma classe `Executor` como a seguinte, muito básica:

```
package ch01.ts;

import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;

public class MyThreadPool extends ThreadPoolExecutor {
    private static final int pool_size = 10;
    private boolean is_paused;
    private ReentrantLock pause_lock = new ReentrantLock();
    private Condition unpaused = pause_lock.newCondition();

    public MyThreadPool(){
       super(pool_size, // core pool size
            pool_size, // maximum pool size
            0L, // keep-alive time for idle thread
            TimeUnit.SECONDS, // time unit for keep-alive setting
            new LinkedBlockingQueue<Runnable>(pool_size)); // work queue
    }

    // some overrides
    protected void beforeExecute(Thread t, Runnable r) {
        super.beforeExecute(t, r);
        pause_lock.lock();
        try {
           while (is_paused) unpaused.await();
        }
```

```
        catch (InterruptedException e) { t.interrupt(); }
        finally { pause_lock.unlock(); }
    }

    public void pause() {
        pause_lock.lock();
        try {
            is_paused = true;
        }
        finally { pause_lock.unlock(); }
    }

    public void resume() {
        pause_lock.lock();
        try {
            is_paused = false;
            unpaused.signalAll();
        }
        finally { pause_lock.unlock(); }
    }
}
```

A classe `MyThreadPool` cria um pool de 10 threads, usando uma fila de tamanho fixo para armazenar as threads que são criadas por debaixo dos panos. Se as threads em pool estiverem todas em uso, então a próxima tarefa em fila deve esperar até que uma das threads ocupadas esteja disponível. Todos estes detalhes de gerenciamento são lidados automaticamente. A classes `MyThreadPool` sobrescreve alguns dos métodos disponíveis para dar o sabor.

Um objeto `MyThreadPool` pode ser usado para fazer um publicador `Endpoint` multithread. Aqui está o publicador revisado, que agora consiste de diversos métodos para dividir o trabalho:

```
package ch01.ts;

import javax.xml.ws.Endpoint;

class TimePublisherMT { // MT for multithreaded
    private Endpoint endpoint;

    public static void main(String[ ] args) {
        TimePublisherMT self = new TimePublisherMT();
        self.create_endpoint();
        self.configure_endpoint();
        self.publish();
    }
    private void create_endpoint() {
        endpoint = Endpoint.create(new TimeServerImpl());
    }
    private void configure_endpoint() {
        endpoint.setExecutor(new MyThreadPool());
    }
```

```
private void publish() {
   int port = 8888;
   String url = "http://localhost:" + port + "/ts";
   endpoint.publish(url);
   System.out.println("Publishing TimeServer on port " + port);
   }
}
```

Depois que `ThreadPoolExecutor` for codificada, tudo que resta é configurar a propriedade executora do publicador `Endpoint` em uma instância da classe executor. Os detalhes de gerenciamento de thread não atrapalham o publicador.

O publicador `Endpoint` multithread é adequado para produção leve, mas este publicador não é um *contêiner* de serviço no sentido verdadeiro; isto é, uma aplicação de software que prontamente pode implementar muitos web services na mesma porta. Um contêiner web como Tomcat, que é a implementação de referência, é mais adequado para publicar múltiplos web services. Tomcat é introduzido em exemplos futuros.

O Que Vem a Seguir?

Um web service baseado em SOAP deve fornecer, como um documento WSDL, um contrato de serviço para seus clientes potenciais. Até agora vimos como um cliente Perl, Ruby e Java pode solicitar o WSDL em tempo de execução para uso em bibliotecas SOAP fundamentais. O Capítulo 2 estuda WSDL mais profundamente e ilustra como pode ser usado para gerar artefatos do lado do cliente, como classes Java, que, por sua vez, facilitam a codificação de clientes de web service. Os clientes Java, no Capítulo 2, não serão escritos do zero, como nosso primeiro cliente Java. Em vez disso, tais clientes serão escritos com a considerável ajuda do utilitário *wsimport*, como foi o `TeamClient` exibido anteriormente. O Capítulo 2 também introduz *JAX-B* (Java API for XML-Binding), uma coleção de pacotes Java que coordenam tipos de dados Java e tipos de dados XML. O utilitário *wsgen* gera artefatos JAX-B que possuem função importante nesta coordenação; então, *wsgen* também terá uma análise mais profunda.