Invocação de Serviços *Web* Utilizando uma Linguagem de Domínio Específico Embutida em Java

Wagner Salazar Pires
Marco Túlio de Oliveira Valente

Departamento de Ciência da Computação
Pontifícia Universidade Católica de Minas Gerais
E-mail: wagner@pucmg.br, mtov@pucminas.br

Resumo

Serviços *Web* são aplicações associadas a um servidor *Web*, as quais têm como entrada e saída mensagens estruturadas em XML. Neste artigo, descreve-se uma linguagem para invocação de serviços *Web*, denominada WSIL, a qual possui três características principais: (i) WSIL permite que serviços *Web* sejam invocados em aplicações clientes por meio de uma sintaxe bastante similar àquela tradicionalmente usada em uma chamada local de método; (ii) WSIL permite a utilização de combinadores de serviço na invocação de serviços *Web*, com o objetivo de tornar tais invocações robustas a falhas de comunicação típicas do ambiente da Internet e (iii) WSIL é uma linguagem de domínio específico embutida em Java, isto é, programas em WSIL são chamados e interpretados a partir de programas Java. Além de apresentar WSIL, o artigo descreve um exemplo de aplicação desenvolvida na linguagem e a implementação de seu ambiente de execução.

Palavras-chave: Serviços *Web*, Linguagens de Domínio Específico, Combinadores de Serviços, Programação Distribuída.

1 Introdução

A *World Wide Web* é um grande sistema de informações distribuído, o qual permite que usuários localizados em qualquer ponto do planeta tenham acesso aos mais diversos tipos de serviços e informações. Passado pouco mais de uma década de sua criação, é inquestionável a influência exercida pela *Web* nos mais diversos ramos de atividades econômicas e sociais. Em essência, a *Web* é um grande hipertexto distribuído cujo funcionamento é apoiado em aplicações servidoras e clientes. Aos servidores, cabe armazenar e disponibilizar páginas escritas na linguagem de marcação HTML. Aplicações clientes, chamadas de *browsers*, permitem que os usuários naveguem pela estrutura de hipertexto, recuperando e exibindo páginas armazenadas nos diversos servidores do sistema. Comunicação entre clientes e servidores é realizada por meio do protocolo HTTP.

A infra-estrutura de informações disponibilizada pela *Web* foi, portanto, planejada e implementada para consumo humano. Tradicionalmente, aplicações clientes são sempre *browsers*, os quais são as ferramentas utilizadas por usuários humanos para recuperar e acessar informações de interesse dos mesmos. Em geral, estas informações são textuais, possuindo pouco ou nenhum grau de estruturação. O modelo tradicional da *Web*, portanto, dificulta que outras aplicações, que não *browsers*, acessem e manipulem informações disponibilizadas na rede. Suponha, por exemplo, que uma empresa tenha que diariamente consultar o preço de uma mercadoria em

um certo número de fornecedores e então comprar determinada quantidade da mesma junto ao fornecedor que tenha o menor preço. Certamente, um funcionário da empresa poderia realizar diariamente esta tarefa, acessando o sistema *Web* de comércio eletrônico dos fornecedores. Como é uma tarefa diária, seria, no entanto, interessante que a mesma fosse automatizada e se transformasse em uma rotina do sistema de compras desta empresa. Para tanto, seria necessário que esse sistema de compras estivesse integrado, via *Web*, aos sistemas dos fornecedores. No entanto, utilizando os protocolos e linguagens originais da *Web* esta integração é praticamente inviável. A grande dificuldade deriva do fato de as saídas de um sistema *Web* tradicional serem documentos textuais, os quais são manipulados facilmente por usuários humanos utilizando um *browser*, mas não por outras aplicações.

Mais recentemente, uma extensão do modelo tradicional da *Web* encontra-se em desenvolvimento com o objetivo de disponibilizar informações que serão "consumidas" a partir de programas escritos em qualquer linguagem de programação. Esta extensão, conhecida pelo nome de *serviços Web* [8, 10], propõe dotar servidores e clientes *Web* da capacidade de processar mensagens XML. Basicamente, serviços *Web* são programas associados a um servidor *Web*. Tais programas têm como entrada e saída mensagens estruturadas em XML. Por exemplo, no caso do sistema de compras descrito acima, cada fornecedor poderia disponibilizar em seu servidor *Web* um programa capaz de: (i) ler um documento XML de entrada descrevendo a mercadoria de interesse, seu código de identificação, quantidade etc; (ii) obter o preço da mercadoria solicitada e (iii) devolver como resposta um outro documento XML contendo o código da mercadoria e o seu preço. Para que serviços *Web* sejam viáveis, alguns novos protocolos e linguagens são utilizados na implementação dos mesmos. Conforme já afirmado, as mensagens trocadas entre clientes e implementações de serviços *Web* são estruturadas utilizando XML. Além disso, um novo protocolo, denominado SOAP [16], define o formato destas mensagens. Por fim, uma linguagem denominada WSDL [21] é utilizada para descrever a interface de um serviço *Web*, isto é, as mensagens que o mesmo é capaz de processar. Em comum com a *Web* tradicional, o protocolo HTTP é usado para transporte de mensagens entre aplicações cliente e servidoras.

Diversas ferramentas e sistemas têm sido propostos para apoiar o desenvolvimento desta nova geração de aplicações *Web*. Como exemplo, temos os sistemas Apache SOAP [4] e Apache Axis [3]. Em geral, estas ferramentas, tornam mais simples o desenvolvimento de serviços *Web*, encapsulando detalhes dos protocolos de comunicação e das linguagens de formatação utilizados. No caso de sistemas desenvolvidos para linguagens orientadas por objetos, uma abstração disponibilizada com freqüência é a capacidade de se associar a servidores *Web* objetos cujos métodos poderão ser chamados remotamente, a partir de aplicações clientes. A implementação destes sistemas torna transparente aos usuários finais todos os detalhes de processamento de mensagens, tanto nos clientes como em servidores. Em linhas gerais, tais sistemas implementam na *Web* o mesmo tipo de abstração disponibilizado em sistemas de objetos distribuídos para redes tradicionais, como CORBA [15] e Java RMI [17].

No entanto, sendo disponibilizados sobre a infra-estrutura da *Web*, serviços *Web* devem lidar com características e com a qualidade de serviço típicas do ambiente Internet. Assim, os mesmos são sujeitos a flutuações na largura de banda disponível, a elevadas latências e a falhas devido a problemas na rede ou a queda de servidores. Por exemplo, estudos já demonstraram que falhas diversas na camada de roteamento da Internet tornam clientes incapazes de acessar a *Web* por, em média, cerca de 14 minutos por dia [7]. Qualquer linguagem ou sistema projetado para *Web* deve ser capaz de lidar com estes níveis de falhas e, se possível, prover construções para atenuar os efeitos das mesmas sobre as aplicações. Um exemplo de tais construções são os chamados *combinadores de serviços* [6]. Propostos por Cardelli, combinadores de serviços constituem um conjunto de operadores para tratar falhas comuns na *Web*. Combinadores de serviços foram pro-

postos originalmente para serem usados em linguagens e bibliotecas com recursos para recuperar páginas *Web* tradicionais, isto é, para recuperar um documento HTML dada a sua URL. Os mesmos permitem especificar estratégias para tratamento de falhas quando da recuperação de uma página HTML, para especificar estratégias para recuperação concorrente de páginas etc. As linguagens WebL [13] e XL [9] são exemplos de linguagem que provêm suporte a combinadores de serviços.

Neste trabalho, descreve-se uma linguagem para invocação de serviços *Web*, chamada WSIL (*Web Service Invocation Language*), a qual apresenta três características principais:

- WSIL permite que serviços *Web* sejam invocados com uma sintaxe bastante similar àquela de uma chamada local de métodos. Basicamente, as únicas informações necessárias para se invocar um serviço *Web* são a sua localização e a sua interface descrita em WSDL, as quais são normalmente oferecidas pelo implementador do serviço. Esta característica distingue WSIL de outras bibliotecas para invocação de serviços *Web*, como Apache SOAP [4].

- WSIL permite a utilização de combinadores de serviço na invocação de serviços *Web*. Assim, WSIL oferece aos clientes de serviços *Web* um recurso poderoso para tratar falhas de comunicação e para definição de invocações concorrentes de serviços. Ressalte-se que WSIL difere-se de linguagens como WebL, onde combinadores de serviços são usados na recuperação de páginas *Web* tradicionais. Em outras palavras, WSIL estende o conceito de combinadores de serviços para a nova geração de serviços *Web* que está sendo disponibilizada na Internet.

- WSIL é uma linguagem embutida em Java, isto é, programas em WSIL são chamados e interpretados a partir de uma linguagem hospedeira. Até o momento a única linguagem hospedeira disponível é Java. Assim, programas WSIL são usados apenas para invocar um serviço *Web* e obter o seu resultado. Considera-se que esta abordagem possui duas vantagens principais: (i) evita-se que programadores tenham que utilizar uma nova linguagem para desenvolver clientes de serviços *Web* (conforme é a proposta, por exemplo, da linguagem XL) e (ii) viabiliza-se a invocação de serviços *Web* por meio de uma sintaxe mais simples e natural, sem, no entanto, requerer modificações na sintaxe e nas ferramentas de desenvolvimento da linguagem hospedeira.

Este trabalho encontra-se dividido em 7 seções. Na Seção 2, são apresentados os principais conceitos e protocolos utilizados na implementação de serviços *Web*. Na Seção 3, descreve-se as principais características e construções da linguagem WSIL. A Seção 4 contém um exemplo de uso da linguagem. Na Seção 5, descreve-se a implementação de um ambiente para execução de programas em WSIL. A Seção 6 compara a linguagem proposta neste trabalho com outras linguagens e sistemas para construção de serviços *Web*. Por fim, na Seção 7, apresentam-se as conclusões do artigo.

2 Serviços *Web*

Serviços *Web* são aplicações distribuídas que se comunicam por meio de mensagens que são codificadas em XML, formatadas e encapsuladas segundo o protocolo SOAP [16] e usualmente transportadas via HTTP[1]. Complementando esta definição, serviços *Web* têm suas interfaces definidas em uma linguagem denominada WSDL [21] e são registrados em servidores UDDI [18]. No restante desta seção, procura-se explicar com um pouco mais de detalhes cada um dos itens

[1]Outros protocolos podem ser usados no transporte de mensagens, como, por exemplo, SMTP.

da definição acima. Antes disso, na Tabela 1, compara-se aplicações baseadas no conceito de serviços *Web* com aplicações *Web* tradicionais:

	Web tradicional	**Serviços Web**
Informações	Não estruturadas, para "consumo humano"	Estruturadas, para "consumo por aplicações"
Linguagem	HTML	XML, WSDL
Protocolos	HTTP	HTTP, SOAP, UDDI
Clientes	*Browser*	Aplicações escritas em quaisquer linguagens

Tabela 1: Diferenças entre a *Web* tradicional e os Serviços *Web*

Para ilustrar as definições que se seguem, suponha o serviço *Web* codificado em Java pelo seguinte método:

```
String sayHello(String name) {
   return "Hello, " + name;
}
```

Quando um cliente solicita a invocação do método acima, uma mensagem deve ser transmitida ao servidor, contendo os argumentos da chamada. O protocolo SOAP define como gerar um documento em XML representando esta mensagem, isto é, como "empacotar" argumentos da chamada em XML. Além disso, SOAP define o formato da mensagem de resposta enviada pelo servidor. Mensagens geradas de acordo com este protocolo são denominadas de envelopes SOAP e, sendo documentos textuais, podem ser transportadas por meio de um protocolo como HTTP. Mostra-se abaixo o envelope SOAP gerado quando de uma invocação do método `sayHello`, tendo a string `Wagner` como argumento:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <SOAP-ENV:Header> </SOAP-ENV:Header>
  <SOAP-ENV:Body>
     <ns1:sayHello xmlns:ns1="Hello"
         SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <name type="xsd:string">Wagner</name>
     </ns1:sayHello>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

A criação e processamento de envelopes SOAP fica a cargo da biblioteca de desenvolvimento de serviços *Web* utilizada, de forma que os programadores de aplicações clientes e servidoras não precisam conhecer detalhes deste protocolo.

A interface de um serviço *Web*, isto é, a especificação das mensagens recebidas e produzidas pelo mesmo, é definida em uma linguagem específica, baseada em XML e denominada de WSDL. Mostra-se abaixo a definição WSDL do serviço Web definido pelo método `sayHello`:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="urn:hello"
```

```
   ......
   <wsdl:message name="sayHelloResponse">
      <wsdl:part name="sayHelloReturn" type="xsd:string"/>
   </wsdl:message>
   <wsdl:message name="sayHelloRequest">
      <wsdl:part name="in0" type="xsd:string"/>
   </wsdl:message>
   <wsdl:portType name="HelloServer">
      <wsdl:operation name="sayHello" parameterOrder="in0">
         <wsdl:input message="intf:sayHelloRequest" name="sayHelloRequest"/>
         <wsdl:output message="intf:sayHelloResponse" name="sayHelloResponse"/>
      </wsdl:operation>
   </wsdl:portType>
   ..........
   </wsdl:binding>
   <wsdl:service name="HelloServerService">
      <wsdl:port binding="intf:rpcrouterSoapBinding" name="rpcrouter">
         <wsdlsoap:address location="http://localhost8080/soap/servlet/rpcrouter"/>
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

Como se pode ver neste exemplo, a criação de uma especificação WSDL não é trivial, já que a mesma é extensa e requer definições bastante primitivas. Assim, existem ferramentas que a partir de uma interface Java geram uma interface WSDL equivalente. Com isso, evita-se que o programador de um servidor tenha que conhecer detalhes da sintaxe de WSDL. Um provedor de serviços pode ainda publicar a especificação WSDL de seu serviço junto a um serviço de diretório. Clientes podem consultar este diretório para descobrir a localização e a especificação de eventuais serviços *Web* que desejem acessar. Um protocolo chamado UDDI especifica as mensagens necessárias para registrar e consultar por serviços *Web* em tais diretórios.

Programando um cliente: Definidos e caracterizados os protocolos de uma pilha de serviços *Web*, o trecho de código a seguir realiza uma invocação do método `sayHello`, passando como parâmetro a string `Wagner`. O código mostrado encontra-se em Java e foi implementado usando-se o pacote Apache SOAP.

```
01:  try {
02:    URL url= new URL("http://localhost:8080/soap/servlet/rpcrouter");
03:    String name= "Wagner";
04:
05:    Call call= new Call();
06:    call.setTargetObjectURI("urn:hello");
07:    call.setMethodName("sayHello");
08:    call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
09:
10:    Vector args= new Vector();
11:    args.addElement(new parameter ("name", String.class, name, null));
12:    call.setParams(args);
13:
14:    Response resp= null;
15:    try {
16:      resp= call.invoke(url, "");
17:      Parameter ret= resp.getReturnValue();
```

```
18:       System.out.println(ret.getValue());
19:     }
20:     catch(SOAPException e ){
21:       System.erro.println("Erro: "+e.getMessage());
22:     }
23:   }
24:   catch(Exception e ){
25:     e.printStackTrace();
26:   }
```

Na linha 2, instancia-se a URL do serviço a ser chamado. Nas linhas 5 a 8, cria-se um objeto da classe `Call`, o qual referencia a chamada a ser realizada. Nas linhas 10 a 12, cria-se a lista de argumentos da chamada a ser realizada. Esta lista é associada ao objeto da classe `Call`. Após configurar todas estas propriedades, na linha 16 invoca-se o serviço remoto e, na linha 17, obtém-se o resultado do mesmo. Possíveis exceções ocorridas durante a invocação do serviço são tratadas nas linhas de 20 a 25. Analisando este código, conclui-se que a invocação de um serviço demanda uma quantidade de código considerável.

3 A Linguagem WSIL

WSIL é uma linguagem que permite a invocação de serviços *Web* por meio de um sintaxe similar àquela de chamadas de métodos locais. WSIL oferece ainda um conjunto de combinadores de serviços usados para lidar com falhas de comunicação inerentes a aplicações distribuídas construídas sobre a infra-estrutura da Internet. A fim de facilitar o aprendizado e a integração com programas escritos em outras linguagens, WSIL é uma "linguagem embutida", isto é, programas WSIL são chamados e interpretados a partir de programas escritos em uma linguagem hospedeira.

A única construção disponibilizada por WSIL é denominada de *serviço Web*, a qual é declarada pela palavra-reservada `webservice`. Todo `webservice` é identificado por um nome. Serviços *Web* podem ser *primitivos* ou *compostos*. Um serviço primitivo, além de um nome obrigatório, é associado a uma especificação WSDL. Tais serviços são ditos primitivos porque as mensagens que o mesmo aceita são aquelas definidas em sua especificação WSDL. Mostra-se abaixo a declaração de dois serviços primitivos, denominados de `servidor1` e `servidor2`:

```
webservice servidor1 { "servidor1.wsdl" }
webservice servidor2 { "servidor2.wsdl" }
```

Um `webservice` composto, além de um nome, possui uma lista de mensagens. Toda mensagem possui uma lista de parâmetros e um corpo. O corpo de uma mensagem é composto por uma invocação a outro serviço, possivelmente utilizando-se combinadores de serviços. O resultado de uma mensagem é definido como sendo o resultado da invocação da mensagem contida em seu corpo. O trecho de código a seguir mostra a declaração de um serviço *Web* composto:

```
webservice servidor3 {
  mensagem_1 (lista_parametros_1) is corpo_1
  mensagem_2 (lista_parametros_2) is corpo_2
  ..............
  mensagem_n (lista_parametros_n) is corpo_n
}
```

No corpo de uma mensagem, uma invocação de serviço é definida por meio da seguinte sintaxe (onde * é o operador de Kleene):

```
<invocacao> -> <invocacao_simples> ( <combinador> <invocacao_simples> )*
<invocacao_simples> ->  <nome_servico>.<nome_mensagem>(<lista_parametros>)
```

Suponha que a especificação WSDL de `servidor1` e `servidor2` definam o método `sayHello`, usado como exemplo na Seção 2. Suponha ainda que S1 e S2 são duas invocações de serviços *Web*. Define-se abaixo a sintaxe e a semântica dos combinadores de serviços existentes em WSIL:

- *Seqüenciamento*, com a seguinte sintaxe: `S1 ; S2`. O serviço S1 é invocado e, na seqüência, invoca-se S2. Retorna-se como resposta uma lista contendo os resultados da execução de S1 e de S2. Caso a execução de um dos serviços falhe, armazena-se nesta lista um objeto que denota a falha ocorrida.
  Exemplo: `servidor1.sayHello("pedro") ; servidor2.sayHello("maria")`

- *Escolha não-determinística*, com a seguinte sintaxe: `S1 | S2`. Um dos dois serviços é não-deterministicamente escolhido e então executado. Caso o serviço escolhido falhe, o outro serviço será invocado. O resultado da invocação é aquele retornado pelo serviço invocado com sucesso. No caso de falhas de ambos serviços, retorna-se uma lista com objetos que denotam as falhas ocorridas. Basicamente, este combinador é usado para balanceamento de carga no acesso a serviços *Web* replicados.
  Exemplo: `servidor1.sayHello("pedro") | servidor2.sayHello("maria")`

- *Execução concorrente com único resultado*, com a seguinte sintaxe: `S1 || S2`. Ambos serviços são executados concorrentemente. O resultado da invocação é aquele fornecido pelo serviço que retornar primeiro. No caso de falhas de ambos serviços, retorna-se uma lista com objetos que denotam as falhas ocorridas. Basicamente, este combinador é usado para reduzir o tempo de resposta de um serviço provido por múltiplos servidores.
  Exemplo: `servidor1.sayHello("pedro") || servidor2.sayHello("maria")`

- *Execução concorrente com múltiplos resultado*, com a seguinte sintaxe: `S1 + S2`. Ambos serviços são executados concorrentemente. O resultado da invocação é uma lista contendo os resultados da execução de S1 e de S2. Caso a execução de um dos serviços falhe, armazena-se nesta lista um objeto que denota a falha detectada. Basicamente, este combinador é usado para reduzir o tempo de resposta de uma seqüência de invocações de serviços, no caso em que uma invocação é independente do resultado da outra.
  Exemplo: `servidor1.sayHello("pedro") + servidor2.sayHello("maria")`

- *Execução alternativa em caso de falha*, com a seguinte sintaxe: `S1 ? S2`. Invoca-se o serviço S1 e, em caso de falha do mesmo, invoca-se o serviço S2. O resultado da invocação é aquele retornado pelo serviço executado com sucesso. No caso de falhas de ambos serviços, retorna-se uma lista com objetos que denotam as falhas ocorridas. Basicamente, este combinador é usado para aumentar o grau de tolerância a falhas de uma aplicação.
  Exemplo: `servidor1.sayHello("pedro") ? servidor2.sayHello("maria")`

Gramática: A seguir, mostra-se a gramática da linguagem WSIL. Nesta gramática, não-terminais são escritos entre < e > e terminais são escritos entre aspas duplas. Na notação usada, $A*$ equivale a $\epsilon \mid A \mid AA \mid AAA \mid \ldots$ e $A+$ equivale a $A* - \{\epsilon\}$.

```
<programa>                -> (<serviços_web>)+
<serviços_web>            -> <serviços_web_primitivo>
                           | <serviços_web_composto>
<serviços_web_primitivo> -> "webservice" <identificador> "{" <literal> "}"
<serviços_web_composto>  -> "webservice" <identificador> "{" (<mensagem>)+ "}"
<mensagem>                -> <identificador> "(" <lista_parametros> ")" "is" <invocacao>
<lista_parametros>        -> <identificador> ( "," <identificador> )*
                           | epsilon
<invocacao>               -> <invocacao_simples> ( <combinador> <invocacao_simples> )*
                           | "(" <invocacao> ")"
<invocacao_simples>       ->  <identificador>.<identificador>(<lista_parametros>)
<combinador>              ->  "||"  |  ";"  |  "?"  |  "+"  |  "|"
<identificador>           ->  <letra> ( <letra> | <digito> )*
<digito>                  ->  "0" | "1" | ...| "9"
<letra>                   ->  "a" | "b" | ...| "z" | "A" | "B" | ...| "Z" | "_"
```

4 Exemplo de Uso

Descreve-se, nesta seção, um exemplo de uso da linguagem WSIL. Suponha uma universidade *multi-campi*, com três *campus* em cidades diversas de um determinado estado. Nos exemplos que se segue, estes *campi* são denominados de `CampusA`, `CampusB` e `CampusC`. Suponha ainda que tais *campi* são integrados via Internet e que nos mesmos encontram-se implementados os serviços *Web* descritos nas Tabelas 2 e 3. Neste exemplo, assume-se ainda que o diretório de artigos DBLP (`http://www.informatik.uni-trier.de/~ley/db/`) disponibilizou o acesso a suas bases via serviços *Web*. A interface WSDL deste serviço está contida em `dblp.wsdl`. Ainda para enriquecer o exemplo, assumiu-se que uma réplica deste serviço foi instalada no `CampusA`.

Serviços Web do `CampusA`	
Mensagens	**Descrição**
consultaLivro(ISBN)	Retorna dados de livro disponível na biblioteca deste *campus*, dado seu ISBN
consultaNota(matr, disciplina)	Retorna nota referente a uma matrícula de aluno e a uma disciplina
consultaLab(lab, hInicio, hFim)	Retorna código de computador disponível no laboratório e horário informados
consultaArtigo(titulo)	Retorna dados de artigo catalogado em uma réplica local do serviço DBLP

Tabela 2: Serviços disponibilizados no `CampusA`. Estes serviços estão replicados em dois servidores deste *campus*. As especificações WSDL de cada uma destas réplicas encontram-se nos arquivos `a1.wsdl` e `a2.wsdl`.

Mostra-se a seguir um exemplo de programa WSDL que faz uso dos serviços *Web* disponíveis nesta universidade. Inicialmente, declaram-se um série de serviços primitivos, associados às especificações WSDL de cada um dos serviços descritos:

```
01: webservice campusA1 {"a1.wsdl"}  -- servidor principal do campus A
02: webservice campusA2 {"a2.wsdl"}  -- servidor secundário do campus A
03: webservice campusB {"b.wsdl"}
04: webservice campusC {"c.wsdl"}
```

Serviços Web dos `CampusB` e `CampusC`	
Mensagens	**Descrição**
consultaLivro(ISBN)	Retorna dados de livro disponível na biblioteca deste *campus*, dado seu ISBN

Tabela 3: Serviços disponibilizados nos `CampusB` e `CampusC`. O serviço é provido por único servidor de cada *campus* e sua especificação está contida nos arquivos `b.wsdl` e `c.wsdl`.

```
05: webservice dblp {"dblp.wsdl"}    -- servidor do diretório DBLP
```

Em seguida, declara-se um serviço composto, o qual utiliza combinadores de serviços para implementar um serviço mais robusto, incluindo estratégias de concorrência e tolerância a falhas.

```
06: webservice WebUniversidade {
07:
08:   consultaNota(matr,discipl) is
09:      campusA1.consultaNota(matr,discipl) |
10:      campusA2.consultaNota(matr,discipl)
11:
12:   consultaLivro(ISBN) is
13:      campusA1.consultaLivro(ISBN) +
14:      campusB.consultaLivro(ISBN) +
15:      campusC.consultaLivro(ISBN)
16:
17:   consultaLab(lab1,lab2,lab3,hInicial,hFinal) is
18:      campusA1.consultaLab(lab1,hInicial,hFinal) ||
19:      campusA1.consultaLab(lab2,hInicial,hFinal) ||
20:      campusA1.consultaLab(lab3,hInicial,hFinal)
21:
22:   consultaArtigo(titulo) is
23:      campusA1.consultaArtigo(titulo) ?
24:      dblp.consultaArtigo(titulo).
25: }
```

O serviço composto `WebUniversidade` possui quatro mensagens:

- A primeira mensagem, chamada `consultaNota`, utiliza o combinador `|` (escolha não-determinística) para fins de balanceamento de carga. Como esta mensagem é bastante utilizada pelos alunos da universidade para consultar suas notas, este combinador viabiliza a divisão do processamento da mesma entre os dois servidores do `campusA`.

- A segunda mensagem, chamada `consultaLivro`, utiliza o combinador `+` (execução concorrente com múltiplos resultados) para executar concorrentemente a pesquisa pelo livro solicitado nas bibliotecas dos `campusA`, `campusB` e `campusC`. Com isso o solicitante pode obter mais rapidamente informações sobre o livro desejado em todas as três bibliotecas da universidade.

- A terceira mensagem, chamada `consultaLab` utiliza o combinador `||` (execução concorrente com único resultado) para descobrir um microcomputador dos laboratórios informados que esteja disponível no horário especificado. Supondo que o usuário não possui predileção por laboratório, a primeira resposta obtida pode ser retornada ao mesmo.

- A quarta mensagem, chamada `consultaArtigo`, utiliza o combinador `?` (execução alternativa em caso de falha) para consultar os dados de um artigo inicialmente na réplica local do serviço DBLP e, no caso de falha, na implementação original do serviço. A réplica local é consultada primeiro, visto que a mesma, em geral, oferece um tempo de resposta menor.

5 Implementação

O ambiente de execução WSIL inclui um compilador e um interpretador. O interpretador é chamado a partir de um programa Java, usando uma API definida pela linguagem. O compilador basicamente lê e analisa um arquivo WSIL e gera uma estrutura de dados que irá dar suporte ao processo de interpretação, a qual é salva em um arquivo com a extensão `.wrt` (*WSIL run time*).

A implementação do compilador, chamado `wsilc`, foi baseada em técnicas tradicionais de compilação. O mesmo utiliza um autômato finito para apoiar o processo de análise léxica e um *parser* descendente recursivo para apoiar o processo de análise sintática. O "código gerado" por este compilador é uma estrutura de dados descrevendo cada um dos serviços *Web* especificados no arquivo de entrada. Esta estrutura de dados é organizada como uma "tabela *hash* de tabelas *hashes*". A tabela *hash* principal mapeia cada serviço do arquivo WSIL de entrada em uma segunda tabela *hash*, a qual associa, por sua vez, cada mensagem de um serviço a uma árvore binária que descreve a invocação da mesma. Suponha, por exemplo, um programa WSIL com três serviços s_1, s_2 e s_3. Suponha ainda que o serviço s_1 defina as mensagens m_1, m_2 e m_3. Assumindo que o corpo da mensagem m_2 é formado pela invocação `a ? (b | c)`, a Figura 1 ilustra a estrutura de dados gerada pelo compilador `wsilc` após a compilação deste programa.

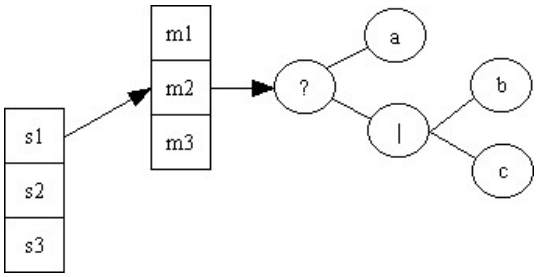

Figura 1: Estrutura de dados gerada pelo compilador `wsilc` ao compilar um arquivo de entrada com serviços s_1, s_2 e s_3. O serviço s_1 define as mensagens m_1, m_2 e m_3. O corpo da mensagem m_2 é formado pela invocação `a ? (b | c)`

Além de gerar a estrutura de dados descrita, o compilador `wsilc` utiliza a ferramenta `WSDL2Java`, integrante do pacote Apache Axis [3], para gerar para cada serviço primitivo do arquivo de entrada um conjunto de classes Java que serão utilizadas para invocar mensagens de tais serviços. Estas classes são utilizadas, portanto, em tempo de execução, pelo interpretador de WSIL. Este interpretador basicamente avalia a árvore de invocação da mensagem chamada. Particularmente, a biblioteca de *threads* de Java é utilizada pelo interpretador para avaliar combinadores cuja semântica requer a invocação concorrente de serviços *Web*. Além disso, a biblioteca de reflexividade de Java é usada para invocar serviços primitivos do arquivo fonte. Como afirmado, estes serviços são chamados utilizando-se as classes geradas pela ferramenta `WSDL2Java`. Basicamente, ao invocar um serviço primitivo, o interpretador instancia objetos de tais classes. Ressalte-se que o nome das mesmas somente é conhecido em tempo de execução.

Daí o emprego de reflexividade para viabilizar a instanciação destes objetos.

API para utilização de WSIL em Java: Para acessar um serviço *Web* a partir de um programa Java, utilizando a linguagem WSIL, basta instanciar um objeto da classe `WSILCall`, a qual pertence ao pacote `wsil`. A partir do método `invoke` deste objeto, o usuário terá acesso a todos os serviços descritos no arquivo. O método `invoke` possui três parâmetros: o nome do serviço *Web* a ser chamado, o nome da mensagem e um vetor com os argumentos da mesma. O trecho de código a seguir demonstra o acesso a um serviço *Web* de nome `Tradutor`, declarado no arquivo `exemplo.wsil`. Este serviço possui um método chamado `TraduzPt_Eng` que traduz palavras do português para inglês.

```
WSILCall call = new WSILCall("exemplo.wsil");
Object[] args = new Object[] {"mesa","livro"};
String result= call.invoke("Tradutor","TraduzPt_Eng",args);
```

6 Trabalhos Relacionados

Linguagens de programação para domínios específicos (DSL, ou *Domain Specific Languages*) já foram propostas para diversos tipos de problemas [19]. Como exemplo, temos linguagens para geração de compiladores (como Lex [14] e Yacc [12]), para processamento de textos (como AWK [2]), para desenho de figuras (como PIC [5]), para controle de robôs (como Yampa [11]) ou para acesso a bancos de dados relacionais (como SQL).

Não temos, no entanto, conhecimento de nenhuma outra DSL, além daquela descrita neste artigo, que tenha sido desenvolvida com o intuito de permitir a invocação de serviços *Web*. As soluções existentes para invocação de serviços *Web* podem ser classificadas da seguinte forma:

- Baseadas em bibliotecas ou *frameworks*, os quais oferecem classes que encapsulam diversos detalhes inerentes à invocação de um serviço *Web*. Como exemplo, temos os sistemas Apache SOAP [4], Apache Axis [3] e WSIF (*Web Service Invocation Framework*) [20]. Apache SOAP foi uma das primeiras bibliotecas desenvolvidas e, por isso, oferece recursos bastante primitivos, conforme mostrado na Seção 2. A biblioteca Apache Axis, sendo mais recente, incorpora recursos comuns em *middlewares* orientados por objetos, como *stubs* e *skeletons*, e com isso oferece um maior nível de abstração. Já o *framework* WSIF utiliza WSDL como linguagem padrão para descrição de serviços providos ao longo de uma rede. WSIF utiliza WSDL para descrever serviços acessados não só via SOAP, mas também via RMI/IIOP ou JMS. Assim, a idéia é fornecer uma interface de programação única para serviços acessados por meio de diferentes protocolos.

  Quando compara-se WSIL com as bibliotecas e *frameworks* mencionados, consideramos que WSIL oferece um maior de nível de abstração que todos eles, permitindo que o código de invocação de um serviço *Web* seja simples e de fácil entendimento. Além disso, WSIL incorpora a idéia de combinadores de serviços.

- Baseadas em linguagens de propósito geral, às quais são incorporadas construções para invocação de serviços *Web*. Como exemplo, temos a linguagem XL [9], a qual é uma linguagem de programação projetada especificamente para o desenvolvimento de serviços *Web*. XL possui três características importantes. Primeiro, trata-se de uma linguagem de programação de alto nível, com os mais variados tipos de comandos, operadores, valores etc. Segundo, o sistema de tipos da linguagem é baseado em XML. Os valores primitivos da

linguagem são documentos XML, isto é, são estruturados no formato de árvores ordenadas e rotuladas. A terceira característica é o suporte a combinadores de serviços.

A abordagem adotada no projeto e na implementação de WSIL possui duas vantagens em relação àquela adotada na linguagem XL. Primeiro, WSIL é uma linguagem de domínio específico embutida em Java. Esta característica facilita o uso e o aprendizado de WSIL, uma vez que não se exige que os programadores tenham que aprender integralmente uma nova linguagem de programação. Segundo, a adoção de um sistema de tipos integralmente baseado em XML torna a programação em XL bastante distinta da programação usual em linguagem orientadas por objetos, onde basicamente existem valores de tipos primitivos (inteiros, reais, booleanos etc) e de tipos estruturados (objetos). Assim, esta característica dificulta o uso e aprendizado de XL.

Por fim, ressalte-se que serviços *Web* oferecem funcionalidades similares àquelas existentes em *middlewares* orientados por objetos, como CORBA [15] e Java RMI [17]. Basicamente, a principal abstração oferecida por tais *middlewares* é o conceito de chamada remota de método. Uma aplicação cliente pode chamar um método de um objeto localizado em uma aplicação remota com uma sintaxe similar à de uma chamada local de método. Assim, serviços *Web* são muitas vezes considerados como sendo uma extensão de tais *middlewares* para a *Web*. Por exemplo, a linguagem WSDL desempenha papel equivalente à linguagem IDL em CORBA. O protocolo SOAP padroniza o procedimento de *marshalling* e *ummarshalling* realizado por *stubs* e *skeletons* em CORBA. O protocolo de transporte HTTP desempenha numa arquitetura de serviços *Web* papel semelhante ao do protocolo IIOP em CORBA. Por fim, serviços UDDI são funcionalmente similares ao serviço de nomes de CORBA. Uma comparação mais detalhada entre CORBA e serviços *Web* pode ser encontrada em [1].

7 Conclusão

Serviços *Web* são considerados atualmente a principal novidade em relação ao modelo tradicional da *Web*, baseado em hipertexto. Atualmente, por exemplo, já é possível, por meio de serviços *Web*, acessar o catálogo de livros à venda na livraria eletrônica Amazon ou pesquisar por alguma informação no Google. Acredita-se que serviços *Web* desempenharão ainda um papel importante na integração de sistemas de informação na Internet, principalmente no caso de sistema de comércio eletrônico B2B (*Bussiness-to-Bussines*). Assim, neste trabalho, foi proposta uma linguagem de programação de domínio específico para construção de aplicações clientes de serviços *Web*.

Em relação a sistemas e linguagens com objetivos semelhantes, WSIL apresenta três vantagens importantes: (i) WSIL permite que serviços *Web* sejam invocados em aplicações clientes por meio de uma sintaxe bastante similar àquela tradicionalmente usada em uma chamada local de método; (ii) WSIL permite a utilização de combinadores de serviço na invocação de serviços *Web*, com o objetivo de tornar tais invocações robustas a falhas de comunicação típicas do ambiente da Internet e (iii) WSIL é uma linguagem de domínio específico embutida em Java, isto é, programas em WSIL são chamados e interpretados a partir de programas Java.

Como trabalhos futuros, pretende-se implementar um maior número de exemplos práticos em WSIL, de forma a validar as construções e o poder de expressão da linguagem. Pretende-se também investigar a adoção de mecanismos de segurança em WSIL, já que esta questão é essencial em aplicações distribuídas construídas na Internet.

Referências

[1] A. Gokhale A, B. Kumar, and A. Sahuguet. Reiventing the Wheel? CORBA vs. Web Services. In *Eleventh International World Wide Web Conference*, 2002.

[2] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison Wesley, 1988.

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

[4] Apache SOAP. http://ws.apache.org/soap/.

[5] J. L. Bentley. Programming Pearls: Little Languages. *Communications of the ACM*, 29(8):711–721, 1986.

[6] Luca Cardelli and Rowan Davies. Service Combinators for Web Computing. In *Proceedings of the Conference on Domain-Specific Languages (DSL-97)*, pages 1–10. USENIX Association, October 1997.

[7] Bharat Chandra, Mike Dahlin, Lei Gao, Amjad-Ali Khoja, Amol Nayate, Asim Razzaq, and Anil Sewani. Resource management for scalable disconnected access to Web services. In *Tenth World Wide Web Conferrence*, pages 245–256, may 2001.

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

[9] Daniela Florescua, Andreas Grünhagen, and Donald Kossmann. XL: An XML Programming Language for Web Service Specification and Composition. In *International World Wide Web Conference*, May 2002.

[10] K. Gottschalk, S. Graham, H. Kreger, and J. Snell. Introduction to Web Services Architecture. *IBM Systems Journal*, 41(2):170–177, 2002.

[11] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, Robots, and Functional Reactive Programming. In *Summer School on Advanced Functional Programming, Oxford University*, 2002.

[12] S.C. Johnson. Yacc - Yet Another Compiler Compiler. Technical Report 32, Bell Labs, 1974.

[13] Thomas Kistler and Hannes Marais. WebL - A programming language for the Web. *Computer Networks and ISDN Systems*, 30:259–270, April 1998.

[14] M.E. Lesk. Lex - a Lexical Analyzer Generator. Technical Report 39, Bell Labs, 1975.

[15] Object Management Group. The Common Object Request Broker: Architecture and Specification, October 2000. 2.4.

[16] Simple Object Access Protocol (SOAP). http://www.w3c.org/TR/wsdl.

[17] Sun Microsystems. Java Remote Method Invocation Specification, revision 1.8, 2002.

[18] Universal Description, Discovery and Integration (UDDI). http://www.uddi.org.

[19] Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[20] Web Service Invocation Framework (WSIF). http://ws.apache.org/wsif/.

[21] Web Services Description Language (WSDL). http://www.w3c.org/TR/wsdl.