3
Implementações

Como já foi dito anteriormente, uma arquitetura orientada a serviços é um conceito ainda não muito bem definido. Por isso, encontrar trabalhos correlatos com um tema similar é complicado. Ao mesmo tempo é bem fácil encontrar diversos trabalhos acadêmicos que tratam de temas que foram abordados no capitulo anterior, porém que sequer citam o termo serviço. Diante deste quadro, este capítulo irá tratar de descrever como funcionam três implementações conhecidas de *frameworks* para a construção de sistemas orientados a serviços que, somados, abrangem todas as características ditas relevantes citadas no capítulo anterior.

3.1.
Vinci

Vinci [Agrawal et al., 2001] [Bayardo et al, 2002] é um *framework* que fornece suporte à construção de aplicações orientadas a serviços baseado na troca de documentos XML e focado na construção de aplicações Web. Requisitos como escalabilidade, robustez e desenvolvimento ágil com rápida passagem da prototipação para implantação também foram considerados no seu projeto. Maximizar o desacoplamento e o desempenho são outros pontos fortes do *framework.* Toda comunicação do Vinci é feita por documentos XML não validados para aumentar o desacoplamento entre clientes e servidores. Por questões de desempenho, caracteristicas como segurança e verificação de mensagens são desconsideradas propositalmente e delegadas para os serviços, o que acaba relegando Vinci a ambientes de redes locais confiáveis. Porém, aplicações criadas com Vinci podem se comunicar com o mundo externo (Internet, WANS, etc...) através de *proxies* e *gateways* que serão explicados mais adiante. Vinci é um *framework* neutro quanto a plataforma de execução ou linguagem de programação, podendo ser executado em ambientes altamente heterogêneos que suportem comunicação via TCP/IP.

3.1.1. Comunicação

Toda a comunicação no Vinci é feita através de documentos *XTalk*. O Quadro 4 introduz uma mensagem XTalk. Esses documentos não são XML puros e sim uma representação pseudo-binária de documentos XML. Na prática o XTalk é um formato proprietário que se fundamenta nos conceitos e padrões dos documentos XML. Tal opção foi feita por questões principalmente de desempenho, tamanho do código e uso de memória. A representação em memória de documentos XTalk são consideravelmente menores e mais rápidas de se percorrer do que documentos XML representados por estruturas como DOM (*Document Object Model*).

```
<QUERY xmlns:vinci="http://vinci.almaden.ibm.com/2000/vinci">
 <vinci:COMMAND>lookup</>
 <TITLE>Zen and the Art of Motorcycle Maintenance</>
</>

<RESPONSE>
 <ISBN>0553277472</>
 <AUTHOR>Robert M. Pirsig</>
 <PUBLISHER>Bantam Books</>
</>
```

Quadro 4 – Exemplo de uma mensagem *XTalk*

Uma característica marcante dos documentos XTalk é que em nenhum momento eles se preocupam com a tipagem dos dados. Diferente de protocolos como SOAP, o XTalk nunca especifica se o dado contido em um documento deve ser visto como um número inteiro, ponto flutuante ou qualquer outro tipo de dado. Segundo os criadores do Vinci, existem duas razões básicas para tal decisão.

A primeira é que incluir informação sobre o tipo de dado no corpo do documento é uma sobrecarga desnecessária, uma vez que o cliente já deve saber qual informação e como ela está disponibilizada no documento que lhe será enviado como resposta ao seu pedido. Se o cliente não tiver o mínimo

conhecimento de tal informação e de como ela está disponibilizada a comunicação entre ambos fica inviável.

A segunda razão é que incluir informações de tipo no corpo do documento dificulta a evolução tanto dos clientes quanto dos serviços uma vez que a cada alteração no formato pré-definido do documento deve ser repassada para todos que o utilizem. A proposta do Vinci e do XTalk é que quanto mais livre forem os documentos, mais fácil é a comunicação entre clientes e serviços, e sua conseqüente evolução.

3.1.2. Ambiente de Execução

O ambiente de execução (*Enviroment)* do Vinci é de suma importância para o *framework.* Ele é responsável por tarefas ditas administrativas que são inerentes a sistemas distribuídos como:

- Localização de serviços,
- Monitoramento de serviços e
- Ciclo de vida dos serviços

O ambiente de execução ainda é responsável por disponibilizar facilidades para a escalabilidade do sistema, balanceamento de carga dos serviços e depuração.

Dentre as tarefas destinadas ao ambiente de execução, a mais importante é referente à localização de serviços. Para tanto, o Vinci possui uma implementação de provedor de serviços (*ServiceProvider*) própria denominada *VNS* (*Vinci Name Service*).

O VNS pode ser considerado como uma versão mais avançada, e desenvolvida especificamente para redes locais, de protocolos para localização de serviços. Um cliente pode consultar o VNS de duas formas básicas: diretamente através do nome do serviço desejado, ou definindo uma expressão XPath [Clark et al, 1999] que é aplicada aos metadados dos serviços registrados no VNS.

Quando um novo servidor é inicializado, ele se comunica diretamente com o VNS (através de um protocolo proprietário sobre TCP/IP) e negocia uma porta por onde o serviço implementado estará disponível para os clientes. Nessa fase inicial de comunicação, o VNS então determina ao serviço a porta na qual serão

enviados os pedidos. O serviço pode especificar uma faixa de portas e determinar ao VNS que só deseja atuar em uma porta que pertença a tal faixa. O conceito por de trás dessa negociação de portas é abstrair a decisão de qual porta operar, evitando assim possíveis conflitos por serviços que utilizem a mesma porta. O VNS também suporta rodízio de portas para que serviços problemáticos possam ser imediatamente reiniciados sem a necessidade de esperar o *timeout* padrão do protocolo TCP/IP que é demasiadamente longo.

Quando um cliente solicita ao VNS um determinado serviço, este responde com um conjunto de servidores e portas por onde podem ser acessados serviços que atendam as necessidades do cliente. Além disso, o VNS pode distribuir os pedidos por várias portas e servidores diferentes em casos de uma grande quantidade de operações simultâneas, disponibilizando assim um balanceamento de carga rudimentar.

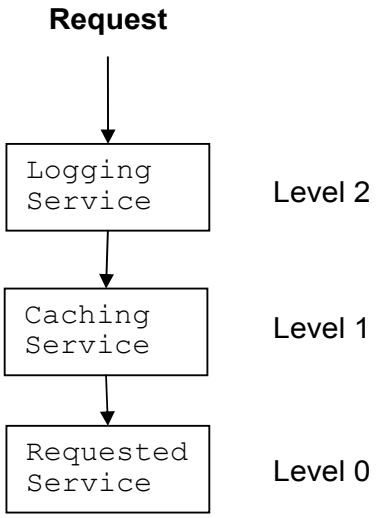

Figura 5 – Encadeamento dos serviços segundo a prioridade registrada pelo VNS

O VNS também estabelece prioridades entre serviços. Com isso, a ordem que os serviços são disponibilizados para os clientes pode seguir uma ordem pré-definida de acordo com as prioridades atribuídas. Assim, é possível, por exemplo, criar meta-serviços como um serviço de cache de dados. Nesse caso tal serviço receberia uma prioridade de maneira que seria sempre consultado antes que o pedido chegasse efetivamente ao serviço original requisitado, e assim poderia

decidir se os dados solicitados estão disponíveis em cache e determinar o fim da requisição. Esse processo é totalmente transparente para os clientes, ficando a cargo do VNS o fluxo da requisição. Vários serviços e meta-serviços podem ser chamados em cascata usando o mecanismo de prioridades, fornecendo para os clientes funcionalidades como cache, log, autenticação, etc. A Figura 5 mostra o encadeamento em cascata de 3 serviços, sendo que dois deles são meta-serviços executados antes do serviço requisitado.

Finalmente, o ambiente de execução do Vinci prevê conexões com o mundo exterior (fora da rede local) através de *gateways* e *proxies*. Os *proxies* tem por função receber pedidos internos usando o protocolo do Vinci e reformatá-los para algum outro padrão que seja aceito no mundo externo, como SOAP ou IIOP. Depois, deve receber as respostas desses serviços externos e traduzi-los de volta ao protocolo proprietário do Vinci.

Já os *gateways* tem função inversa, eles recebem os pedidos de clientes externos a rede local onde está sendo executado o Vinci e convertem esses pedidos de maneira que os serviços do Vinci possam entendê-los. Finalmente eles enviam de volta a resposta aos clientes que requisitaram as informações inicialmente.

3.2. Jini

Tradicionalmente os sistemas operacionais modernos são construídos assumindo o fato que um computador possui um processador, alguma memória e uma unidade de disco. Quando um computador se inicia, a primeira providência tomada pelo sistema operacional é procurar uma unidade de disco; caso esta não seja encontrada, o computador simplesmente não funciona. Porém, com o avanço tecnológico surgiram os pequenos dispositivos como celulares. Esses dispositivos possuem um processador, alguma memória e, principalmente, uma conexão com uma rede. A primeira coisa que um celular faz quando é ligado é procurar uma rede; caso esta não seja encontrada ele simplesmente não funciona. Atualmente estão sendo projetados computadores que funcionam sem unidade de disco e que como os celulares, dependem principalmente de uma conexão de rede. Essa mudança no cenário tecnológico que altera o principal dispositivo de um

computador, de uma unidade de disco para uma conexão de rede, afeta como devemos projetar, desenvolver e executar os novos sistemas. Para isso a Sun Microsystems resolveu criar a plataforma Jini [Jini, 1999] [Jini, 2003] [Arnold et Al, 1999].

Jini é uma tentativa de repensar as arquiteturas dos computadores dando uma importância maior a rede na qual um computador está conectado. O caráter dinâmico e heterogêneo de uma rede onde dispositivos são conectados e removidos regularmente é o ponto principal desta nova arquitetura.

A plataforma Jini é uma coleção de APIs (Application Program Interfaces) e protocolos de comunicação construídos sobre a base da linguagem Java. Jini é freqüentemente denotado como uma extensão da linguagem Java. A Figura 6 mostra o desenho esquemático das camadas que compõem a arquitetura proposta para o Jini.

O uso de RMI é de alta importância na arquitetura de Jini. Com isso é possível transportar objetos através da serialização, e executar métodos (pela reflexividade de Java), entre diferentes máquinas virtuais que podem estar executando em diferentes computadores sobre diferentes sistemas operacionais.

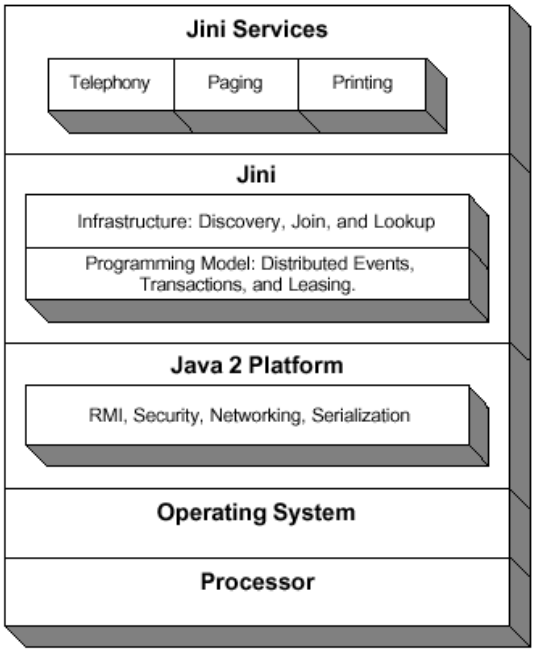

Figura 6 – As camadas da plataforma JINI

Em Jini todos componentes de um sistema são considerados serviços, desde componentes de software binários até dispositivos diversos com como, por exemplo, uma impressora ou uma torradeira. Os serviços são agrupados seguindo o conceito de federação. Uma federação de serviços é um conjunto de serviços disponíveis na rede que um cliente (que pode ser um programa, outro serviço ou um usuário) pode utilizar.

A idéia central por trás do conceito de federação é não existir uma unidade central controladora e autenticadora dos serviços que compõem a federação. Ao contrário disso, a plataforma através de componentes do ambiente de execução (*runtime*) fornece mecanismos de localização e autenticação, de maneira que os clientes podem se encontrar sem passar por uma autoridade central.

Após um cliente encontrar um serviço, este serviço passa a estar incorporado ao cliente. Na prática, o cliente executa o download de código de um p*roxy* para o serviço requerido. O *proxy* então passa a estar no mesmo espaço de memória do cliente, que se comunica com ele através de chamadas locais. O cliente então não necessita mais do ambiente de execução do Jini para utilizar o serviço requerido.

3.2.1. Funcionamento

A plataforma Jini possui três processos como mecanismos principais em seu ambiente de execução:

- Descoberta (*Discovery*),
- Junção (*Join)* e
- Busca (*Lookup*)

Um serviço quando é iniciado deve se anunciar, para que os clientes tenham como acessar suas funcionalidades. Para isso ele deve se registrar em um serviço de localização (*Service Provider*). O processo de descoberta trata de encontrar na rede local, serviços de localização através do envio de um pacote de informações para todas máquinas na rede (*broadcast*). Feito isso, os serviços de localização existentes respondem com informações para a identificação única de cada um dos serviços de localização na rede com endereço IP e porta.

De posse desta identificação o novo serviço que está se registrando deve se unir à rede de serviços através do processo de junção (*Join*), onde basicamente o serviço envia para o registro um objeto p*roxy* que é responsável por se comunicar diretamente com o serviço. Assim quando clientes solicitarem alguma das funcionalidades deste novo serviço, irão receber esse *proxy* para efetuar a comunicação entre o cliente e o serviço. Na terminologia do Jini esse p*roxy* é chamado de *service object*.

O último processo trata da interação de um cliente qualquer com um serviço de localização de serviços. O cliente para utilizar o serviço de localização deve antes encontrá-lo através do processo de descoberta. Uma vez com acesso a algum serviço de localização, o cliente envia a requisição contendo uma ou mais interfaces que deverão ser atendidas pelo serviço a ser encontrado. Caso exista ao menos um serviço que atenda a tal requisição, o serviço de localização responde com o *proxy* para acesso direto ao serviço requerido. A partir deste ponto a interação é feita diretamente entre cliente e serviço remoto através do p*roxy* local.

3.2.1.1. Leasing

A plataforma Jini utiliza o conceito de *lease* para controlar o acesso aos serviços. Um *lease* é o direito de um cliente para utilizar durante um determinado período de tempo um serviço desejado. Cada *lease* é negociado entre o cliente e o serviço propriamente dito. Tal negociação faz parte do protocolo de comunicação definido no Jini.

O *lease* dever ser renovado antes que o período pelo qual o cliente obteve o direito se encerre; caso contrário o cliente não conseguirá mais utilizar as funcionalidades do serviço. Os *leases* podem ser exclusivos ou não exclusivos. Caso sejam exclusivos apenas o cliente que obteve direito ao *lease* conseguirá utilizar o serviço até que seu período se encerre. Tal mecanismo é comumente utilizado para acessos concorrentes a recursos compartilhados.

Uma importante função do *lease* é permitir a evolução dos serviços sem que ocorra um erro de comunicação não previsto. Isso ocorre em sistemas de longa duração onde o acesso ao serviço de localização ocorre no ínicio da execução e não mais. Se, depois de um tempo, o serviço desejar evoluir de maneira que

proxies antigos não mais funcionem, o serviço precisa notificar os clientes de alguma forma. Como clientes geralmente não são notificáveis, o mecanismo de *lease* obriga os clientes a, de tempos em tempos, re-localizar os serviços e assim possibilitar uma evolução do p*roxy*.

O *lease* é uma característica importante do Jini que poucos ambientes de execução distribuídos possuem.

3.2.2. Considerações Finais

A plataforma Jini remete para o ambiente de rede o modelo de programação orientada a objetos. A arquitetura da plataforma, que concentra grande parte de seus esforços na separação entre implementação e interface, traz uma série de benefícios para o desenvolvedor de sistemas baseados em serviços. Outro ponto importante é o fato dos clientes não precisarem saber como o *proxy* para um serviço e o cliente se comunicam. Todo esse protocolo de comunicação é exclusivo do serviço. Uma vez que o cliente efetua o mecanismo de *lookup* ele passa a ter localmente um *proxy* para o serviço sendo acessado, com assim a comunicação entre eles p*roxy* e serviço é trasparente para o cliente como mostra a Figura 7. Com isso qualquer protocolo de comunicação como RMI, IIOP, SOAP, entre outros, pode ser utilizado sem maiores problemas.

A plataforma Jini oferece uma série de recursos para a construção de aplicações baseadas em uma arquitetura orientada a serviços, por isso é quase uma referência para sistemas de tal categoria. O conceito de serviços é muito forte e usado constantemente. Finalmente, os mecanismos de *discovery, lookup e join* fornecem uma base para o baixo grau de acoplamento necessário em sistemas compostos de serviços, pois consegue lidar bem com o caráter dinâmico de uma rede local com os serviços sendo criados e destruídos o tempo todo.

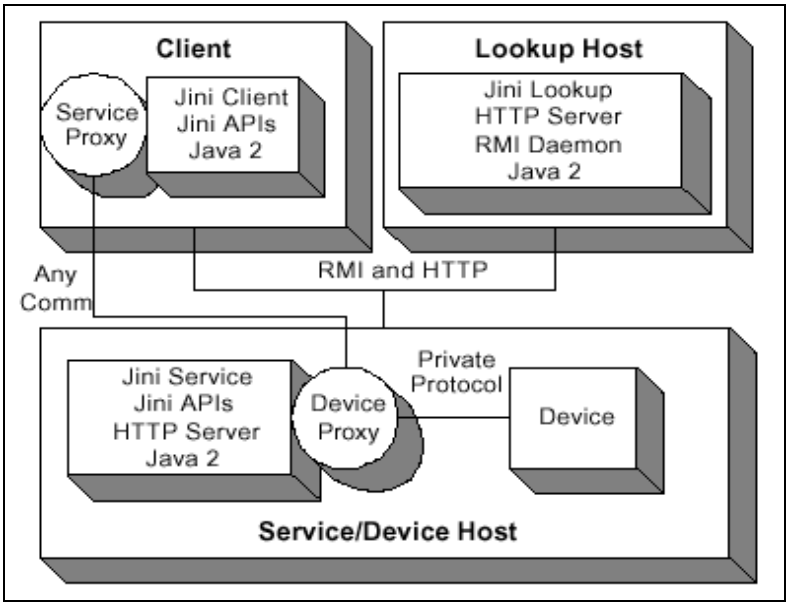

Figura 7- Esquema de comunicação do Jini

3.3. XML Web Services

Nos últimos anos, uma série de tecnologias para computação distribuída tem surgido. Iniciativas como CORBA, COM/DCOM e Java RMI, entre outras, acabaram tendo diversos tipos de problema. CORBA tem o problema de possuir protocolos complexos. COM/DCOM é uma tecnologia totalmente proprietária e dependente de plataforma. Já Java RMI é muito presa a linguagem Java e dificilmente se comunica com outras linguagens de programação.

Por outro lado, a *World Wide Web* é uma iniciativa intimamente relacionada com a computação distribuída, que obteve grandes resultados com rápida disseminação no mundo tudo. A WWW é apoiada em um conjunto de padrões simples e abertos, o que facilita sua adoção. Na WWW toda comunicação é feita através do protocolo HTTP e a informação é sempre representada através de HTML. Ambos são simples e suportados por inúmeras plataformas. Uma tecnologia para computação distribuída é muito mais facilmente adotada quando baseada em protocolos simples e abertos, se possível padronizados. Porém, a WWW foi projetada para que seus documentos sejam lidos por seres humanos.

Em conseqüência, a comunicação entre sistemas através de HTTP e HTML não é uma tarefa simples.

Os *XML Web Services* tentam resolver esse problema criando protocolos que sejam entendidos pelas máquinas, se apoiando na infra-estrutura disponibilizada pela WWW e principalmente, na sua experiência bem sucedida. Todos protocolos utilizados são padronizados e chancelados pelo *W3C (World Wide Web Consortium*) uma entidade independente e consensual. Os *XML Web Services* são baseados em três protocolos textuais, que são mais simples que protocolos binários, e tipicamente se utilizam de HTTP para comunicação. Isso faz com que potencialmente os *XML Web Services* possam, em um curto espaço de tempo, estar acessíveis em todas plataformas. Atualmente diversas plataformas possuem implementações dos protocolos que compõem a infra-estrutura necessária para os *XML Web Services*.

Os três protocolos textuais utilizados nos *XML Web Services* são baseados na linguagem XML, daí a opção deste trabalho de chamá-los de *XML Web Services* e não somente de *Web Services* [2]. São eles:

- *SOAP – Simple Object Access Protocol*
- *WSDL – Web Services Definition Language*
- *UDDI – Universal Description, Discovery and Integration*

SOAP é o protocolo de comunicação de um sistema distribuído. Baseado em XML, SOAP define o formato das mensagens que devem ser trocadas entre os componentes de uma comunicação remota. WSDL é a linguagem que descreve a interface entre estes componentes que deve ser obedecida para que a comunicação seja bem sucedida. E finalmente, o UDDI é o protocolo utilizado para a comunicação com o registro de serviços.

[2] O termo "Web Services" é comumente usado para se referenciar a sistemas distribuídos utilizando os protocolos SOAP, WSDL e UDDI. Como no entendimento deste trabalho, todas implementações de arquiteturas orientadas a serviços podem ter seus componentes chamados de "Web Services", preferiu-se se usar o termo "XML Web Services" para se referenciar a serviços se utilizando SOAP, WSDL e UDDI.

3.3.1. SOAP

SOAP surgiu para preencher uma lacuna existente nos padrões de sistemas distribuídos. Através de um formato razoavelmente simples e legível, aplicações executando em diferentes máquinas podem acessar procedimentos remotamente. A simplicidade objetivada pelos criadores de SOAP não foi atingida como deveria. O resultado é um protocolo mais simples que os protocolos binários como IIOP, e mais complicado que outros protocolos baseados em texto como XML-RPC. Atualmente a versão existente é a 1.1, mas já está sendo elaborada a versão 2.0. O Quadro 5 mostra a estrutura básica de um documento SOAP, enquanto o Quadro 6 exemplifica uma mensagem usando este protocolo.


```
<SOAP:Envelope
         xmlns:SOAP=”http://schemas.xmlsoap.org/soap/envelope”>
  <SOAP:Header>
     <!– Header content -->
  </SOAP:Header>
  <SOAP:Body>
     <!– Body content -->
  </SOAP:Body>
</SOAP:Envelope>
```

Quadro 5 – Estrutura básica de uma mensagem SOAP

```
<SOAP:Envelope
         xmlns:SOAP=”http://schemas.xmlsoap.org/soap/envelope”>
  <SOAP:Body>
    <et:eTicket
         xmlns:et=”http://acme-travel.com/eticket/schema”>
      <et:passengerName first=”Joao” last=”Machado” />
      <et:flightInfo airlineName=”AA”
                     flightNumber = ”222” />
    </et:eTicket>
  </SOAP:Body>
</SOAP:Envelope>
```

Quadro 6 – Exemplo de uma mensagem SOAP sem o elemento SOAP:*Header*

A especificação de SOAP não descreve sobre qual protocolo as mensagens devem ser transportadas, porém as implementações de SOAP mais usadas foram construídas em cima do protocolo HTTP. Como o objetivo de SOAP é se disseminar utilizando a infra-estrutura de rede disponível atualmente, o uso de HTTP como meio de transporte é muito importante para que os pacotes de dados contendo as mensagens SOAP possam atravessar *firewalls*.

3.3.1.1. WSDL

Enquanto a especificação de SOAP trata diretamente do conteúdo das mensagens a serem trocadas, WSDL descreve, através de documentos XML, quais e como as mensagens devem ser trocadas entres as partes. WSDL foi criada diretamente pela indústria de desenvolvimento de software e mais tarde padronizada pelo W3C.

WSDL tem por objetivo descrever interfaces de componentes de software de uma forma neutra, ou seja, sem se prender a nenhuma plataforma, para que os componentes possam ser comunicar em ambientes heterogêneos. Assim, é necessário que haja uma tradução para localizar a descrição neutra da interface para uma implementação especifica de software e hardware. Isto tipicamente é feito através de ferramentas de auxilio ao desenvolvimento.

Os criadores da WSDL criaram uma linguagem genérica, de maneira que ela pode ser usada para descrever qualquer tipo de componente de software e não somente *XML Web Services*. O [WSIF, 2004], por exemplo, usa WSDL para descrever outros tipos de componentes de software como EJBs e classes Java normais com acesso remoto através de RMI. A estrutura de um arquivo WSDL é apresentada no Quadro 7.

```
<definitions>
  <types>
    definition of types........
  </types>
  <message>
    definition of a message....
  </message>
```

```
  <portType>
    definition of a port.......
  </portType>
  <binding>
    definition of a binding....
  </binding>
</definitions>
```

Quadro 7 - Estrutura de um arquivo WSDL

Porém, a grande flexibilidade e generalidade da linguagem acabaram trazendo problemas para sua legibilidade. Uma interface descrita através de WSDL é complexa e quase sempre necessita do auxilio de uma ferramenta para seu entendimento. O resultado é que o processo de desenvolvimento acaba se tornando altamente dependente de ferramentas auxiliares. O Quadro 8 apresenta a descrição em WSDL do serviço *HelloService*, que possui apenas uma operação denominada *sayHello*. A complexidade de WSDL poderia ter sido evitada se a linguagem tivesse sido projetada para não ser tão genérica quanto é na prática. Linguagens de descrição de interface como a IDL de CORBA resolvem grande parte dos problemas que WSDL resolve, porém com uma legibilidade muito maior.

No capítulo 4 será apresentada uma linguagem para a descrição de interfaces, baseada na simplificação da WSDL, criada especialmente para o *framework* XMLTalk.

```
<definitions name="HelloService"
 targetNamespace="http://www.ecerami.com/wsdl/HelloService.wsdl"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://www.ecerami.com/wsdl/HelloService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <message name="SayHelloRequest">
      <part name="firstName" type="xsd:string"/>
   </message>
   <message name="SayHelloResponse">
      <part name="greeting" type="xsd:string"/>
   </message>
   <portType name="Hello_PortType">
```

```
      <operation name="sayHello">
         <input message="tns:SayHelloRequest"/>
         <output message="tns:SayHelloResponse"/>
      </operation>
   </portType>
   <binding name="Hello_Binding" type="tns:Hello_PortType">
      <soap:binding style="rpc"
         transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="sayHello">
         <soap:operation soapAction="sayHello"/>
         <input>
            <soap:body
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               namespace="urn:examples:helloservice"
               use="encoded"/>
         </input>
         <output>
            <soap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               namespace="urn:examples:helloservice"
               use="encoded"/>
         </output>
      </operation>
   </binding>

   <service name="Hello_Service">
      <documentation>WSDL File for HelloService</documentation>
      <port binding="tns:Hello_Binding" name="Hello_Port">
         <soap:address
location="http://localhost:8080/soap/servlet/rpcrouter"/>
      </port>
   </service>
</definitions>
```

Quadro 8 – Exemplo completo de um serviço descrito em WSDL

3.3.1.2. UDDI

Enquanto SOAP é usado para efetuar a comunicação entre as partes em um sistema distribuído, e WSDL descreve como essa comunicação deve ser feita, UDDI tem a responsabilidade de fornecer um mecanismo para localização de serviços (*Service Provider*). UDDI define um registro de serviços centralizado porém, diferentemente das implementações mais especificas como as fornecidas pelo Vinci ou JINI, o UDDI deve poder ser acessado por qualquer plataforma. Um registro de serviços que aceita pedidos no formato definido pela especificação do UDDI é tipicamente conhecido como um *servidor UDDI*.

UDDI é composto por duas especificações básicas. Uma das especificações trata do registro de serviços e seu modelo de dados, definindo que tipos de informações devem ser fornecidos por cada serviço para serem armazenadas pelo servidor. A outra especificação se preocupa com as operações disponibilizadas pelo servidor e define uma API para consulta e atualização dos dados armazenados.

A estrutura de um servidor UDDI é muito parecida com qualquer registro de serviços, não apresentando nenhuma grande inovação. A diferença maior está que o servidor UDDI é acessado sempre através de mensagens SOAP, como mostra o Quadro 9. Além disso, o modelo de dados usado é bem mais completo, e por isso complexo, que o utilizado normalmente em registros de serviços, como exemplificado no Quadro 10.

Os servidores UDDI foram propostos para serem um grande repositório universal de serviços, por onde todos clientes iriam buscar dinamicamente os serviços. Atualmente esse modelo não é utilizado, e tipicamente os clientes não passam pelo servidor de UDDI para acessarem o serviço requerido, relegando os servidores de UDDI a uma função secundária. Além disso, principalmente devido a restrições de segurança, cada vez mais está se tornando uma tendência o uso de servidores UDDI privados, onde apenas clientes internos (no sentido de estarem na mesma rede local) possuem acesso ao registro de serviços.

```
<?xml version="1.0" encoding="UTF-8"?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <find_service  businessKey="*"  generic="1.0"  xmlns="urn:uddi-
org:api">
      <name>delayed stock quotes</name>
    </find_service>
  </Body>
</Envelope>
```

Quadro 9 – Exemplo de uma mensagem SOAP que deve ser entendida por um servidor UDDI.

```
<businessEntity businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64"
                operator="www.ibm.com/services/uddi"
                authorizedName="0100001QS1">
  <discoveryURLs>
    <discoveryURL
useType="businessEntity">http://www.ibm.com/services/uddi/uddiget?
businessKey=BA744ED0-3AAF-11D5-80DC-002035229C64
    </discoveryURL>
  </discoveryURLs>
  <name>XMethods</name>
  <description xml:lang="en">
       Web services resource site
   </description>
  <contacts>
    <contact useType="Founder">
      <personName>Tony Hong</personName>
      <phone useType="Founder" />
      <email useType="Founder">thong@xmethods.net</email>
    </contact>
  </contacts>
  <businessServices>
    <businessService
         serviceKey="d5921160-3e16-11d5-98bf-002035229c64"
         businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
      <name>XMethods Delayed Stock Quotes</name>
      <description xml:lang="en">
           20-minute delayed stock quotes
```

```
      </description>
      <bindingTemplates>
        <bindingTemplate
             bindingKey="d594a970-3e16-11d5-98bf-002035229c64"
             serviceKey="d5921160-3e16-11d5-98bf-002035229c64">
          <description xml:lang="en">
             SOAP binding for delayed stock quotes service
          </description>
          <accessPoint URLType="http">
             http://services.xmethods.net:80/soap
          </accessPoint>
          <tModelInstanceDetails>
            <tModelInstanceInfo
       tModelKey="uuid:0e727db0-3e14-11d5-98bf-002035229c64" />
          </tModelInstanceDetails>
        </bindingTemplate>
      </bindingTemplates>
    </businessService>
  </businessServices>
</businessEntity>
```

Quadro 10 – Exemplo do modelo de dados armazenado por um servidor UDDI

3.3.2. Conclusões

SOAP não traz nenhuma grande novidade tecnológica. Desde os tempos do RPC (*Remote Procedure Call*) na década de 70, que se utiliza com sucesso chamadas remotas de procedimentos. Já há algum tempo existe CORBA, que conquistou uma fatia considerável do mercado, mas não obteve maior sucesso principalmente pela falta de apoio das empresas que dominam o mercado de software.

Um ponto importante para o SOAP é a legibilidade de suas mensagens. Tanto os pedidos quanto as respostas são documentos baseados em texto e XML, o que facilita a compreensão. Desenvolver um software que use SOAP é muito mais fácil que em CORBA, por exemplo, pois depurar as mensagens trocadas entre os diferentes objetos remotos é factível. Outros protocolos baseados em

texto, como o XML-RPC, também são altamente legíveis, até mesmo mais do que SOAP, porém não obtiveram o apoio maciço da indústria que SOAP obteve.

Apesar da adoção massiva da tecnologia, os *XML Web Services* ainda tem muito que evoluir. O uso desta tecnologia ainda é prematuro na maioria dos casos, pois questões já resolvidas em outras tecnologias, como manutenção do estado transacional e composição dos serviços, ainda não estão bem resolvidas. A indústria tem trabalhado em novas especificações que resolvam essas questões, como *WS-Transaction* e *WS-Coordination*. Além disso, a segurança da troca de mensagens de SOAP ainda é pequena, por enquanto se restringindo ao suporte do protocolo de transporte SSL (*Secure Socket Layer*). Finalmente, o desempenho é muito aquém das outras tecnologias de comunicação distribuída, principalmente por usar um protocolo baseado em texto relativamente complicado que deve ser devidamente interpretado. O baixo desempenho tem frustrado iniciativas do uso de SOAP na Internet, quase sempre relegando seu uso a redes locais onde o problema é atenuado.

As especificações, apesar de serem de comum acordo entre toda indústria, apresentam vários pontos opcionais que podem gerar implementações distintas, fazendo com que componentes construídos em plataformas distintas não se comuniquem adequadamente. O problema é tão grave que a indústria resolveu se reunir e criar um novo consórcio chamado WS-I (*Web Services Interoperability*), responsável apenas por limitar as especificações do W3C, para finalmente atingir a desejada interoperabilidade entre plataformas através dos *XML Web Services*.

A principal vantagem de SOAP é a padronização que uma especificação seguida por toda uma indústria de software pode trazer. Ambientes altamente heterogêneos como a Internet tiram proveito máximo da padronização, facilitando o trabalho da construção de sistemas distribuídos. WSDL e UDDI são coadjuvantes de SOAP e, assim como SOAP, possuem alguns problemas, porém também tem como sua principal vantagem a padronização de suas especificações.

3.4. Quadro Comparativo

Uma vez detalhado o funcionamento de cada uma das implementações estudadas, é possível traçar um quadro comparativo em relação às características relevantes apresentadas no capítulo 2 deste trabalho. Como é possível, com maior ou menor complexidade, que uma aplicação construída com qualquer um dos *frameworks* apresentados tenha qualquer uma das características listadas, optou-se por traçar um perfil das aplicações típicas construídas. O quadro a seguir lista quais características uma aplicação típica possui quando construída a partir de cada uma das implementações de *frameworks* com suporte a construção de aplicações orientadas a serviços.

	Vinci	**JINI**	**XML Web Services**
Reuso "Caixa-preta"	Sim	Sim	Sim
Distribuição	Sim	Sim	Sim
Heterogeneidade Ambiental	Sim	Não	Sim
Composição	Sim (1)	Não	Não
Coordenação	Não	Não	Não
Dinamismo e Adaptabilidade	Sim	Sim	Sim
Estado	Não	Sim	Não (2)
Sincronia	Não	Sim	Não (2)
Robustez de Protocolos	Sim	Sim	Não (3)

(1) Através da atribuição de níveis de prioridades entre os serviços

(2) Considerando aplicações típicas utilizando o protocolo HTTP.

(3) Considerando aplicações típicas usando SOAP e WSDL.