# 2
# Arquitetura Orientada a Serviços

Uma *arquitetura de software* é um conceito abstrato que dá margem a uma série de definições. A definição usada pelo ANSI/IEEE afirma que uma arquitetura de software trata basicamente de como os componentes fundamentais de um sistema se relacionam intrinsecamente e extrinsecamente [ANSI/IEEE, 2000]. Uma arquitetura orientada a serviços (*SOA – Service Oriented Architecture*) tem como seu componente fundamental o conceito de serviços.

O conceito de *serviço* é definido de diversas formas, principalmente na literatura não acadêmica. Durante o estudo para esta dissertação este tipo de literatura apresentou algumas definições até certo ponto contraditórias. Na literatura não acadêmica, o termo "serviço" é empregado para definir coisas distintas, o que acaba gerando alguma confusão. Existem artigos não acadêmicos, como [Barry & Associates, 2003] e [Sonera, 2002], que chegam a atrelar serviços aos *XML Web Services (*que serão detalhados no capitulo seguinte) por razões mercadológicas sem nenhuma argumentação técnica convincente. Apesar das distintas definições, na maioria das vezes, serviço é usado para se referenciar a um componente de software binário baseado em um contrato [Bieber e Carpenter, 2001].

Já na literatura acadêmica é mais difícil encontrar referências diretas ao termo "serviço". Em geral o que é conhecido como "serviço" na literatura não acadêmica é conhecido como "componentes" ou "contratos" na literatura acadêmica. Se tratarmos o serviço como uma definição ou descrição de ações a serem executadas, um serviço é um contrato. Porém, se definirmos o serviço como a implementação direta dessas ações, um serviço é um componente. Em [Pernici e Mercella, 2000] é usado o termo *e-service* para referenciar componentes binários executando em ambientes distribuídos, mas não é uma regra geral. Fazendo uso da literatura é possível classificar aplicações orientadas a serviços como sistemas cooperativos abertos distribuídos, como veremos nas seções seguintes, pois apresentam todas características descritas para estes tipos de sistemas.

Portanto, o conceito do que é uma arquitetura orientada a serviços ainda não é consensual, uma vez que serviço não é um termo bem definido e se confunde com os conceitos de componente e de contrato. Através deste trabalho foi possível identificar algumas características relevantes que todas aplicações e *frameworks* que se dizem orientados a serviços possuem. É importante ressaltar que não é necessário que se tenha todas características listadas a seguir para denominar se uma aplicação é orientada a serviços; pelo contrário, a maioria das aplicações estudadas possui apenas um pequeno subconjunto dessas características, e são autoproclamadas como aplicações orientadas a serviços. As características consideradas relevantes são:

- Reuso "Caixa-preta"
- Distribuição
- Heterogeneidade Ambiental
- Composição
- Coordenação
- Dinamismo e Adaptabilidade
- Estado
- Sincronia
- Robustez de Protocolos

O intuito do restante deste capitulo é apresentar e discutir tais características.



## 2.1. Reuso "Caixa-preta"

O reuso de software surgiu inicialmente por uma necessidade de economizar recursos de hardware [Clements, 1995]. Há algumas décadas atrás não havia memória suficiente nos dispositivos para armazenar muitas rotinas (segmento de código) e então, foi observado que era possível executar tarefas similares através de uma única sub-rotina parametrizada, e assim não desperdiçar o uso da escassa memória disponível.

Com a evolução dos componentes de hardware e a conseqüente redução de preço, o reuso de software mudou de foco. Desde então o principal objetivo do

reuso é economizar recursos humanos e não mais de hardware, até por que recursos humanos se tornaram muito mais dispendiosos.

O reuso pode ser divido em duas vertentes principais quanto a necessidade do programador de conhecer o componente de software a ser reutilizado: o reuso caixa-branca e o caixa-preta.

No reuso conhecido como *caixa-branca*, existe a necessidade que a implementação do componente de software a ser reusado seja exposta de alguma forma. Em linguagens orientadas a objetos, como Java e C++, é muito comum o uso de herança para se atingir o reuso de software, o que é um exemplo clássico do reuso caixa-branca, porém, tal uso é muitas vezes equivocado. O equívoco ocorre quando o esforço do desenvolvedor para conhecer a implementação que será reusada desperdiça recursos, justamente o que o reuso tenta reduzir. Além disso, existe o problema da classe base frágil (*fragile base class problem*) [Mikhajlov e Sekerinski, 1998], onde uma pequena alteração em uma classe do topo de uma hierarquia pode comprometer toda árvore de derivação e alterar substancialmente o funcionamento de toda a aplicação.

Já o reuso *caixa-preta* visa eliminar a necessidade do desenvolvedor de um conhecimento da implementação de algum componente de software que fará parte do processo de reuso. Em vez disso, o reuso caixa-preta se dá através da descrição de interfaces ou contratos bem definidos que devem ser respeitados pela implementação a ser elaborada. O esforço é sempre usado na nova implementação e nunca ocorre um desperdício tentando entender implementações de terceiros.

O uso de linguagens de programação que dêem suporte explícito à descrição de interfaces, como Java e C#, facilita o reuso caixa-preta. Em outras linguagens este tipo de reuso fica muito mais difícil, pois são necessários alguns subterfúgios (como as classes virtuais puras de C++) para se atingir tal objetivo, o que acaba levando os desenvolvedores a não adotarem esta opção. Ainda não existe uma linguagem robusta e disseminada que dê suporte completo ao conceito de contratos, o que facilitaria ainda mais o reuso caixa-preta, pois alguns outros elementos como pré e pós-condições poderiam ser utilizados.

Atualmente a disseminação do reuso "caixa-preta", principalmente por causa de linguagens de programação como Java, que fornecem suporte adequado, tem causado o surgimento de aplicações "containers", que recebem componentes em tempo de execução (também conhecidos como *plug-ins)* através de uma

política própria de composição e assim alteram dinamicamente e significativamente seu comportamento. O tema composição será abordado mais profundamente em outra seção.

Aparentemente, com o reuso caixa-preta o esforço é maior, pois existe uma necessidade de se escrever código maior do que no reuso caixa-branca, uma vez que nenhuma parte do código inicial será reaproveitada. Isso pode até ser verdade para o custo inicial, porém a grande vantagem do reuso caixa-preta é em médio prazo, pois as dependências entre os componentes de software da aplicação estão explicitadas em um contrato facilitando adaptações e manutenções que se façam necessárias ao longo do tempo. Já no reuso caixa-branca, as ligações entre os componentes (no caso de herança, entre classe e superclasse) são quase sempre obscuras e implícitas, diretamente no código da implementação [Meijler e Nierstrasz, 1997]. Ao longo do tempo, principalmente em aplicações altamente adaptativas, o reuso caixa-preta se torna mais eficaz.

Arquiteturas orientadas a serviços são essencialmente dirigidas através de contratos. Todos relacionamentos entre os componentes que fazem parte de uma aplicação devem estar descritos de alguma forma em um contrato (seja ele uma interface Java, um arquivo XML ou outra forma qualquer), explicitando assim as dependências e facilitando a manutenção e adaptação.

## 2.2. Distribuição

*Sistemas abertos* (*open-systems*) são definidos como sistemas onde novas entidades podem dinamicamente passar a compor o sistema, deixar de existir ou ainda evoluir [Kielmann, 1996][1] . Já um *sistema aberto distribuído* parte do mesmo principio só que leva em consideração o fato das entidades poderem estar localizadas em máquinas diferentes.

O conceito de orientação a objetos para programação distribuída *(OODP - Object Oriented Distributed Programming)* trata de encapsular o estado de um

[1] Existe uma outra definição para sistemas abertos que também é comumente usada. Por esta outra definição sistema aberto é um sistema composto por diferentes plataformas de hardware e software. Neste trabalho, este tipo de sistema é tratado na questão de heterogeneidade de um ambiente.

determinado objeto e fazê-lo acessível através de uma interface bem definida [Landis e Maffeis, 1997]. Usando este conceito é possível acessar diferentes plataformas de hardware e implementações em diferentes linguagens de programação usando um protocolo bem definido, como por exemplo IIOP (de CORBA), desde que a interface esteja descrita em uma linguagem portável, no caso de CORBA a *IDL (Interface Definition Language*). É importante ressaltar que o conceito de acesso a um estado através de uma interface bem definida não está restrito a linguagens de programação com suporte a orientação a objetos. Linguagens procedurais, como C ou Pascal, possuem implementações de portes de protocolos como IIOP, possibilitando que o mesmo conceito seja aplicado, porém sem que o estado esteja encapsulado em um objeto.

Tipicamente, o cliente possui de alguma maneira um identificador (endereço IP, porta, entre outros) estático para o servidor que deverá receber suas requisições e as envia diretamente ao servidor. Isso caracteriza um *sistema distribuído fechado* devido a ser altamente estático, onde novas entidades não podem passar a compor o sistema. O principal exemplo para este tipo de sistema é a implementação de RPC (*Remote Procedure Call*) da *Sun Microsystems,* que data dos anos 70.

Já no caso de *sistemas distribuídos abertos*, o cenário é mais complexo. É preciso, como dito anteriormente, prever a entrada e saída de componentes de maneira transparente para os clientes. O identificador não é mais estático e sim dinâmico, pois o servidor pode ser alterado, desde a sua implementação até sua localização.

A maneira usada atualmente de resolver esta questão é através de um registro de serviços (*Service Provider*). Antes da execução o cliente sabe apenas que tipo de serviço ele deseja acessar, ou seja, qual interface deve ser implementada para satisfazer as suas necessidades. O processo de execução então segue os seguintes passos:

1. O cliente acessa o registro de serviços através de um protocolo especifico, requisitando alguma implementação para uma determinada interface. Neste caso estamos considerando o acesso ao registro como um acesso estático; mais à frente veremos que é possível ter acesso dinâmico para registros de serviços também.

2. O registro responde dando uma identificação para o servidor que implementa tal interface.
3. Finalmente, o cliente acessa o serviço fazendo uso de seu identificador e recebe os resultados, caso existam.

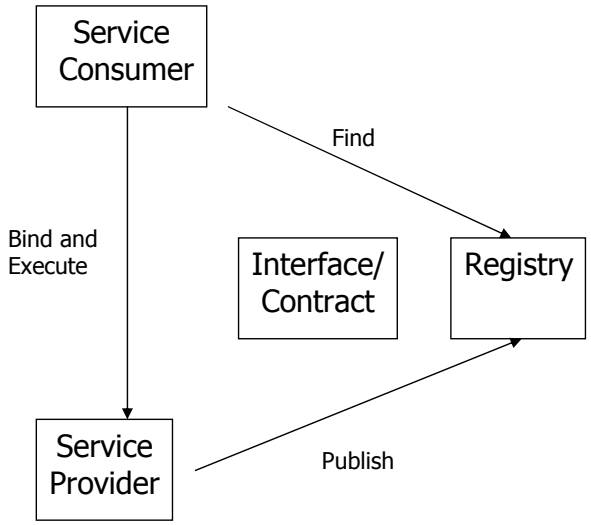


**Figura 1- Esquema do paradigma find , bind e execute**

Este processo é conhecido como o paradigma *find, bind and execute.* A Figura 1 apresenta um esquema para este paradigma. Existem algumas implementações deste paradigma como o serviço de *trading* do CORBA e o UDDI (*Universal Descriptor, Discovery and Integration*) [Oasis, 2002]. Tipicamente, quando um serviço entra em funcionamento ele se registra no registro de serviços, informando quais interfaces ele implementa para que possa ser contactado no futuro. Esta etapa é conhecida como publicação (*publish*).

Algumas implementações ainda possibilitam que a identificação para o registro de serviços seja obtida dinamicamente (ex. JINI). Para tanto, o cliente precisa obter uma identificação para o registro de nomes antes de chamá-lo. É colocada na rede uma requisição usando *broadcast* e todos serviços de nomes que estiverem ativos devem responder. O cliente então armazena esta identificação para chamadas posteriores. Esse processo é conhecido como descoberta (*discover*) e só funciona de maneira adequada em redes locais devido ao uso de *broadcast* (onde não existem *firewalls* barrando esse tipo de requisição).

Aplicações orientadas a serviços são por natureza aplicações distribuídas que fazem uso exaustivo dos conceitos de distribuição, seja por que são formadas

por sistemas legados executando em diferentes máquinas ou por questões de escalabilidade, onde o desempenho do sistema é vital para a resolução do problema proposto. Mais do que isso, geralmente aplicações orientadas a serviços são aplicações distribuídas abertas que fazem uso do registro de serviços.

## 2.3. Heterogeneidade Ambiental

O fenômeno da Internet apresentou uma nova realidade para os desenvolvedores de sistemas, principalmente os *sistemas de informação cooperativos* onde diferentes fontes de informação são agregadas com o objetivo de aumentar a quantidade de informações disponíveis para o usuário (vide Figura 2). Por ser uma rede de escala gigantesca existem uma infinidade de recursos distintos que podem ser acessados através da rede. Tais recursos podem ser não só informações puras mas também na forma de acesso a outros sistemas de informação [Meijler e Nierstrasz, 1997]. Como atualmente até mesmo pequenas redes locais são heterogêneas, pois são compostas de plataformas de hardware e software distintas, a dificuldade da criação de sistemas de informação cooperativos aumenta consideravelmente.

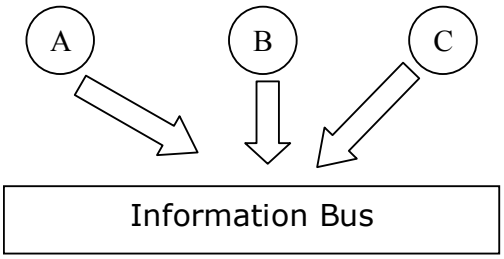


**Figura 2 - Sistemas de Informação Cooperativos**

O cenário traçado tenta mostrar a necessidade de um sistema se comunicar mesmo em ambientes altamente heterogêneos. Além disso, os ambientes podem ser dinâmicos, onde não é possível saber durante a elaboração do sistema quais serão os componentes (e em que plataformas estarão sendo executados) que irão compor o sistema no futuro.

Sistemas baseados em uma arquitetura orientada a serviços são tipicamente sistemas cooperativos abertos. Dessa natureza decorre a necessidade de que sistemas baseados em SOA ofereçam suporte a ambientes heterogêneos. Em geral

os componentes dos sistemas (ou serviços) estão executando em máquinas distintas, ainda que sejam máquinas virtuais distintas (no caso de linguagens interpretadas como Java), e precisam se comunicar de alguma forma. Esta comunicação pode ser feita através de um protocolo padronizado ou sobre um protocolo proprietário, o importante é que esteja bem definido. Atualmente tem se optado por usar protocolos padronizados sobre TCP/IP. Ainda neste capitulo serão mais aprofundadas as questões de distribuição e de protocolos.

Uma das áreas onde arquitetura orientada a serviços mais vem sendo empregada é a integração de aplicações corporativas (*EAI – Enterprise Application Integration*), onde as aplicações legadas já constituídas estão executando em diversas plataformas, desde linux em micro computadores até OS/390 em *mainframes*. Para tanto, criam-se adaptadores que acessem as aplicações legadas e se comuniquem através do protocolo definido pelo sistema. O sistema por sua vez, disponibiliza as informações agregadas fornecidas pelas diversas aplicações legadas, preferencialmente sem ter que reescrever uma só linha de código do legado.



## 2.4. Composição

Os sistemas desenvolvidos até pouco tempo eram tipicamente fechados, de maneira que caso fosse necessária uma extensão era preciso criar outra aplicação e compô-la de maneira a ficar transparente ao usuário [Szyperski, 1996]. Mas esta outra aplicação era de fato uma outra aplicação independente. O conjunto de aplicações independentes podia ser visto como um sistema, tipicamente unidas por um menu único. Com o uso da orientação a objetos e o encapsulamento, passou a ser possível compor distintas aplicações com alguma alteração de código (alteração menos significativa que no desenvolvimento com linguagens procedurais).

Como atualmente os requisitos de um sistema mudam muito rapidamente [Schneider e Nierstrasz, 1999], a orientação a objetos por si só não resolve o problema da composição de aplicações. Mesmo uma pequena alteração de código pode causar um grande impacto no sistema. O ideal perseguido é compor ou recompor aplicações sem tocar em código já constituído e devidamente testado.

Desenvolvimento orientado a componentes disponibiliza uma maior flexibilidade, separando claramente algumas partes de uma aplicação (componentes) através de interfaces bem definidas (caixa-preta). Os primeiros casos de sucesso da composição de componentes foram os construtores de aplicação visuais como Delphi e Visual Basic, porém eles geralmente eram restritos a um certo domínio de aplicação. Delphi, por exemplo, focava em aplicações de banco de dados para o sistema operacional Windows [Schneider e Nierstrasz, 1999].

Como compor esses componentes de maneira simples e eficiente com a menor quantidade de código passou a ser uma área da ciência da computação bastante explorada. Alguns modelos formais e *frameworks* conceituais foram propostos e deram origem a uma série de linguagens especificas para composição de componentes como Piccola [Schneider e Nierstrasz, 1999], CLAM [Sample et al, 1999] e MANIFOLD [Arbab et al, 1993].

Com o uso cada vez maior de linguagens reflexivas, como Java ou C#, a tarefa de composição de componentes tornou-se menos complexa, pois passou a ser razoável em tempo de execução carregar novos componentes e através de reflexão invocar métodos específicos de uma interface definida anteriormente.

## 2.5. Coordenação

Em uma arquitetura orientada a serviços, tão importante quanto a composição é a coordenação. A diferença entre composição e coordenação é que a coordenação está mais intimamente ligada com sincronização, ordenação e temporização, enquanto composição trata mais da combinação de elementos em um todo. Porém, a composição apropriada dos elementos é um pré-requisito para a coordenação [Sample et al, 1999]. Em sistemas abertos e distribuídos, como tipicamente são os ditos orientados a serviços, um modelo de coordenação é altamente recomendável, facilitando o dinamismo e a adaptabilidade (serão vistos mais adiante). Para facilitar a criação do modelo de coordenação, foram criadas algumas linguagens de coordenação. Uma *linguagem de coordenação* tem o propósito de permitir que duas ou mais partes (componentes) se comuniquem com

o objetivo de coordenar operações visando um mesmo objetivo compartilhado por ambas as partes.

Na literatura acadêmica, as linguagens de coordenação têm sido usadas principalmente para a coordenação de agentes paralelos e concorrentes. Exemplos de tal aplicação são linguagens como Linda [Carriero e Gelenter, 1989] ou Bonita [Rowstron e Wood, 1997]

Na indústria, à coordenação está mais relacionada a modelagem de processos de negócios (*BPM – Business Process Management*), onde a linguagem usada tenta descrever intimamente como funciona o processo da corporação, sempre executando uma coordenação entre os agentes que executam cada passo no processo todo. A Figura 3 apresenta um exemplo de um processo de negócios. Exemplos de linguagens com esse intuito são BPEL/BPEL4WS [IBM, 2003].



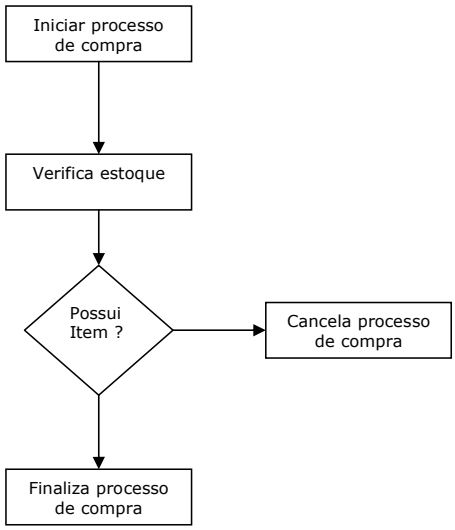


**Figura 3 - Exemplo simplificado de um processo de negócio.**

Esse tipo de linguagem trata principalmente de questões como distribuição de tarefas e sincronia na execução das mesmas. Geralmente, todas requisições enviadas para um determinado sistema são endereçadas a um controlador (*engine*) que utiliza a descrição do processo (na linguagem apropriada) para coordenar as requisições enviadas aos serviços, bem como receber as respostas e repassá-las ao cliente que iniciou o processo.

## 2.6. Dinamismo e Adaptabilidade

Criar sistemas dinâmicos sempre foi um dos grandes desafios tecnológicos encontrados por desenvolvedores. O isolamento de parte das aplicações em bibliotecas de ligação dinâmicas, como DLLs e SOs, foi um grande passo para dinamização dos sistemas. Caso algum requerimento fosse alterado, alteravam-se estas bibliotecas e o sistema estava evoluído.

O uso da carga dinâmica de bibliotecas, aliado a informações de tipo em tempo de execução, como RTTI (*Runtime Type Information)* de C++, sempre possibilitou um dinamismo e adaptabilidade. Porém, a complexidade de escrever código para sistemas com carga dinâmica praticamente excluía esta possibilidade. É muito difícil encontrar sistemas aptos a se adaptar a diferentes bibliotecas em tempo de execução.

O advento das linguagens reflexivas alterou o panorama da dinamicidade dos sistemas. A invocação de métodos de forma dinâmica, e a introspecção dos dados de um objeto e dos metadados de uma classe, facilitou e muito a construção de sistemas dinâmicos. Claro que essa "mágica" não ocorre sozinha. É preciso que o sistema seja construído sabendo da possibilidade da alteração em tempo de execução. Um exemplo simples de um sistema elaborado levando em consideração a possibilidade de evolução pode ser alcançado usando uma linguagem como Java onde existe o conceito de interfaces implementado diretamente na linguagem. Neste caso é necessário que o desenvolvedor sempre utilize as interfaces como tipos de dados, em vez da própria classe que implementa a interface referida.

```
public void doSomething() {
    Vector vet = new Vector();
    vet.add(2);
}
public void doSomething2() {
   List vet = new Vector();
   vet.add(2);
}
public void doSomething3() {
   List vet = (List)getDesiredClass().newInstance();
```

```
   vet.add(2);
}
```

**Quadro 1 – Uso de Interfaces em Java**

O Quadro 1 mostra um exemplo de como se aplicar o conceito de interfaces usando a linguagem Java. Neste exemplo existem três métodos distintos que fazem a mesma ação. O primeiro instancia um objeto da classe *Vector* e tem seu tipo amarrado. O segundo usa omo tipo a interface *List* e instancia um objeto da classe *Vector*. Já o último, busca a classe que implemente a interface *List* através de um método. Para efeito de evolução o último método é mais razoável. Por questões de desempenho os dois primeiros são mais recomendáveis.

Os sistemas abertos são dinâmicos por definição. Como aplicações orientadas a serviços são sistemas abertos, pode-se dizer que também são dinâmicas. Aplicações orientadas a serviços conseguem se adaptar às mudanças de requerimentos com facilidade. Os serviços podem entrar e sair do sistema em tempo de execução, tipicamente se comunicando com o registro de serviços através de mensagens de publicação ou de "despublicação". Além disso, geralmente os sistemas nunca fazem referência direta ao serviço e sim a uma interface para o serviço, possibilitando dinamismo.

## 2.7. Estado

O estado de um componente pode ser dividido em duas classes distintas: *persistente* ou *de sessão*.

*Estado Persistente* é definido quando existe a possibilidade de pelo menos uma das partes de persistir dados trocados na comunicação entre as partes. Quando dados de uma mensagem enviada a um determinado componente podem ser recuperados após a troca de mensagens em qualquer instante de tempo, esse dado foi persistido. Um exemplo clássico é uma mensagem enviada a um servidor de banco de dados. Um cliente pode enviar uma mensagem contendo alguns dados, que posteriormente podem ser recuperados através de uma outra mensagem em um outro momento. No Quadro 2 são apresentadas dois tipos de mensagens na forma de comandos em SQL, uma para persistência de dados e outra para recuperação dos mesmos dados.

```
INSERT INTO Tabela VALUES(1, 25, 'Joao', 1200)

SELECT * FROM Tabela WHERE is = 1
```

**Quadro 2 – Exemplo de duas mensagens (comandos em SQL).**

Já o *Estado de Sessão* diz respeito ao estado dos componentes somente durante a comunicação ou *sessão*. Uma vez finalizada a sessão entre os componentes, o estado é desfeito e seus dados são liberados. Este tipo de estado é importante para manter o andamento de operações mais complexas como, por exemplo, troca de mensagens assíncronas (a próxima seção irá explicar melhor este tema).

O estado de sessão tipicamente é obtido através da geração de um identificador único para a sessão, compartilhado as partes. Ao se iniciar a sessão, esse identificador é gerado e deve ser usado em chamadas subseqüentes. O final da sessão deve ser explicitado para que o estado possa ser desfeito. Em ambientes distribuídos esse tipo de estado é bem mais complicado de ser implementado, pois envolve a questão da coleta de lixo distribuída, uma vez que não é possível garantir que ambas as partes estejam conectadas até o final da conversação.

Tipicamente, sistemas baseados em arquiteturas orientadas a serviços são construídos apenas com componentes que não possuem estado de sessão. Um dos motivos para essa escolha é que geralmente essas aplicações são distribuídas e por isso quanto menos mensagens forem trocadas entre os componentes, melhor desempenho o sistema vai obter. Em uma aplicação que possui estado de sessão, a quantidade de mensagens trocadas é tipicamente maior. Outro fator essencial para que os serviços não possuam estados, é o acoplamento. Uma sessão com estado implica em um acoplamento maior, uma vez que mais mensagens são trocadas.

Como atualmente grande parte das aplicações SOA são construídas por *XML Web Services,* o uso do estado de sessão é pequeno. Isto por que normalmente os componentes destes sistemas se comunicam através de SOAP sobre HTTP, que é um protocolo sem suporte a conexão. Com isso, para se obter estado através desse protocolo é necessário uma terceira parte envolvida na comunicação para controlar, se possível de forma transparente, a identificação das sessões que possam estar acontecendo. Além disso, como citado anteriormente, o

problema da coleta de lixo distribuída afeta diretamente a escalabilidade desses sistemas que são por natureza construídos para serem escaláveis.

## 2.8. Sincronia

A troca de mensagens, seu modo de transmissão e a semântica são partes fundamentais em sistemas distribuídos, pois são a única maneira que os componentes possuem de se comunicar e sincronizar suas ações [Charron-Bost et al, 1996]. É possível classificar a troca de mensagens quanto a sincronia entre os componentes em: comunicação assíncrona e comunicação síncrona.

*Comunicação Assíncrona* é usualmente definida quando o processamento do componente que enviou a mensagem (*sender*) não é bloqueado (*non-blocking*) a espera do desfecho da mensagem. Já *Comunicação Síncrona* ocorre quando o processamento é bloqueado (*blocking*) até que o desfecho ou retorno da mensagem seja completado.

Nenhuma das abordagens pode ser considerada superior a outra. Apesar da comunicação assíncrona ter menos possibilidade de *deadlocks,* e em geral oferecer maior paralelismo (pois não há bloqueio do processamento), ela precisa de tratamento de fluxo e buferização de dados complexos, o que dificulta sua implementação. Além disso, é possível simular um tipo de comunicação usando a outra, usando *buffers* intermediários (simulação de comunicação assíncrona) ou bloqueando explicitamente o processamento até o recebimento da resposta (simulação de comunicação síncrona)

A discussão de sincronia em ambientes distribuídos passa pela necessidade da existência de um estado de sessão (como definido na seção anterior). As aplicações orientadas a serviços atuais são geralmente síncronas devido à inexistência do estado de sessão na maioria dos protocolos usados para comunicação, principalmente protocolos sobre HTTP. Existem algumas aplicações que usam comunicação assíncrona simulada por comunicação síncrona, onde geralmente um servidor atua como intermediário criando um buffer de mensagens e identificando de uma forma proprietária (isto é não padronizada) a conversação. O problema neste caso é que esta intermediação e identificação proprietária acabam restringindo o uso dos componentes por terceiros, diminuindo

assim a heterogeneidade que é uma das características importantes de sistemas baseados na arquitetura orientada a serviços.

## 2.9. Robustez de Protocolos

Recorrendo à definição de arquitetura de software descrita no início deste capítulo, vemos que o relacionamento entre os componentes, tanto interna quanto externamente ao sistema, é fundamental para o desenvolvimento de um software. Este relacionamento tipicamente ocorre através de uma comunicação entre as partes baseadas em um protocolo de comunicação. Um *protocolo* é um conjunto de regras formais que definem como os dados devem ser transmitidos de uma parte para a outra [Webmaster, 2003]. Qualquer tipo de comunicação entre partes necessita de um protocolo, até mesmo a invocação de um método local é regida por um protocolo (neste caso trata questões como ordem dos parâmetros na pilha, localização do valor de retorno, etc...). Porém, em ambientes distribuídos (principalmente se forem heterogêneos), onde as partes estão conectadas através de uma rede de comunicação, os protocolos são mais relevantes. Questões como desempenho, robustez e segurança são muito importantes de serem consideradas nestes cenários. Esta seção irá se ater a questões da robustez dos protocolos.

*Robustez de protocolos* é definida neste trabalho como a liberdade dada aos componentes envolvidos em uma comunicação de evoluir sem quebrar a comunicação. Por exemplo, um cliente acessando um determinado dado em um servidor deve continuar a funcionar mesmo que o servidor evolua e passe a fornecer mais informações, ou altere um ou mais dados.

Protocolos baseados em texto, como XML-RPC ou SOAP, levam vantagem em relação a protocolos binários, como IIOP ou JRMP (um dos protocolos usados pelo RMI de Java), na questão da robustez. Uma pequena alteração em uma mensagem binária pode levar a uma mudança total no entendimento da mesma. No caso de protocolos baseados em texto, isso também pode acontecer, mas a probabilidade é bem menor. Os protocolos binários são ditos mais sensíveis a pequenos ruídos em suas mensagens. Além disso, protocolos baseados em texto tipicamente são autodescritivos no que diz respeito a tipagem dos seus dados, tornando mais fácil uma alteração no tipo dos dados. O Quadro 3 mostra uma

mensagem XML-RPC, totalmente tipada que exemplifica isso. Nesta mensagem, o tipo do parâmetro é descrito pelo tag <i4>, que define o dado (41) como um inteiro de quatro bytes com sinal.

```
<methodCall>
   <methodName>examples.getStateName</methodName>
   <params>
      <param>
         <value><i4>41</i4></value>
         </param>
      </params>
 </methodCall>
```

**Quadro 3 - Exemplo de mensagem XML-RPC.**



**Pedido**

```
<QUERY>
  <ISBN>0595250777</ISBN>
</QUERY>
```

**Resposta 1**

```
<RESPONSE>
  <TITLE>Realities of Foreign Service Life</TITLE>
</RESPONSE>
```

**Resposta 2**

```
<RESPONSE>
  <TITLE>Realities of Foreign Service Life</TITLE>
  <AUTHOR>Patricia Linderman</AUTHOR>
</RESPONSE>
```

**Figura 4 – Exemplo da possibilidade de evolução de protocolos baseados em texto com mensagens descritas por XML.**

Outro ponto a ser considerado, no que diz respeito à evolução, é a inclusão ou exclusão de dados (e não somente a alteração de tipo). Metalinguagens de descrição de dados como XML auxiliam neste ponto. Usando XML para a troca de informações, é possível evoluir um servidor de maneira que os clientes

continuem acessando normalmente sem que notem qualquer diferença. A Figura 4 apresenta esta possibilidade de evolução, a partir de um protocolo baseado em XML.

O pedido para o componente que irá fazer uma consulta a um banco de dados sobre livros é sempre o mesmo. É passado como parâmetro um identificador único para um livro. Em determinado instante o servidor devolve como resultado da consulta o título do referido livro (resposta 1). Eventualmente, o componente que executa tal ação evolui de forma a passar a retornar como resultado da consulta, além do título, o autor do livro (resposta 2). Os clientes que foram construídos para receber a resposta 1 irão continuar a funcionar normalmente, e novos clientes podem ser construídos para que considerem a resposta 2, onde consta o autor do livro. Esta possibilidade de evolução é que torna os protocolos autodescritivos mais robustos.

É importante ressaltar que protocolos autodescritivos possuem o grande problema da falta de desempenho, pois as mensagens são necessariamente maiores que em protocolos não autodescritivos. Por tal razão, para aplicações onde o desempenho é relevante, a escolha de protocolos descritos externamente é mais apropriada. Tipicamente, protocolos binários não são autodescritivos e recomendáveis para aplicações que necessitam de desempenho.

Aplicações orientadas a serviços necessitam que os protocolos de comunicação sejam robustos, pois são altamente dinâmicas e podem ser alteradas constantemente. A grande maioria das aplicações orientadas a serviços atuais utiliza como protocolo de comunicação o SOAP, que é um protocolo baseado em XML, o que comprova a preferência desta classe de protocolos apesar do baixo desempenho.