FUNDAÇÃO EDSON QUEIROZ
UNIVERSIDADE DE FORTALEZA
UNIFOR

Dissertação de Mestrado:

INTEROPERABILIDADE ENTRE PADRÕES DE OBJETOS DISTRIBUÍDOS ATRAVÉS DE BRIDGES E PROTOCOLOS BASEADOS EM XML

Antonio Augusto Ribeiro Pedroza

Orientador:

Professor Pedro Porfírio Muniz Farias, PHD

FORTALEZA / 2002

Livros Grátis

http://www.livrosgratis.com.br

Milhares de livros grátis para download.

INTEROPERABILIDADE ENTRE PADRÕES DE OBJETOS DISTRIBUÍDOS ATRAVÉS DE BRIDGES E PROTOCOLOS BASEADOS EM XML

Antonio Augusto Ribeiro Pedroza

Aprovado em _____/_____/_____

BANCA EXAMINADORA

____

Professor Pedro Porfírio M. Farias, PHD

____

Professor Nabor das Chagas Mendonça, PHD

____

Professor Javam de Castro Machado, PHD

"Por que será que certas coisas tão bobas

parecem tão inteligentes quando é você que as faz?"

Dennis, The Menace, 1958

Dedicatória

A meus pais, Newton e Elisa,

incansáveis incentivadores e promotores de tudo

de bom em minha vida.

A minha esposa, Thais,

por fazer da nossa casa, em dois anos de

casamento, o melhor lar do mundo.

A minha filha Julia,

primogênita amada, que preenche a minha vida

de tal forma, que desde que ela nasceu,

em 15 de agosto de 2002, para mim

o mundo não é mais o mesmo.

Agradecimentos

Iniciamos a busca pela interoperabilidade entre os padrões COM e CORBA em fevereiro de 2000. Isso para nós foi uma ousadia. Pois apenas seis meses antes havíamos tido o primeiro contato com a tecnologia da computação distribuída baseada em objetos. Foi uma longa curva de aprendizado, sem dúvida, mas que valeu a pena.

Gostaria de agradecer a todos que conviveram comigo neste período, em especial aos professores Pedro Porfírio e Arnaldo Belchior, pelas idéias que me iluminaram, pois sem eles este trabalho, certamente, teria sido outro. Ao professor Porfírio, em particular, meu melhor obrigado pelo suporte sempre presente e por ter, em alguns momentos, acreditado em mim mais do que eu mesmo. Isto me incentivou bastante. Aos professores Nabor Mendonça e Javam Machado que leram este trabalho e ofereceram observações relevantes, meus sinceros agradecimentos.

Agradeço também à Universidade de Fortaleza, Unifor, pelas condições extraordinárias com as quais fui privilegiado no curso deste mestrado, através de bolsa de estudos, incentivo a pós-graduação e no atendimento de múltiplos pleitos. Neste tocante, gostaria de mencionar os professores Randal Pompeu, Nise Sanford Fraga e Plácido Rogério Pinheiro, pelo muito que fizeram por mim.

Índice de Figuras ... **x**

Abstract ... **xi**

Resumo ... **xii**

1 INTRODUÇÃO ... **1**

1.1 RUMO AOS SISTEMAS DISTRIBUÍDOS BASEADOS EM OBJETOS ... 1

1.2 DESCRIÇÃO DO PROBLEMA E PROPOSTA DE SOLUÇÃO ... 16

1.3 CONCLUSÂO ... 18

2 COM, CORBA e BRIDGES ... **20**

2.1 CORBA ... 21

2.1.1 O VOCABULÁRIO CORBA ... 24

2.1.2 FUNDAMENTOS E CARACTERÍSTICAS ... 26

2.2 COM, DCOM E ACTIVEX ... 31

2.2.1 A ARQUITETURA COM ... 38

2.2.2 INTERFACES C++ e INTERFACES COM IDL ... 40

2.3 ANÁLISE DE COM E CORBA ... 53

2.4 BRIDGES COM USO DE COMPONENTES ESPECÍFICOS ... 58

2.4.1 O USO DE C++ PARA INTEROPERAR COM E CORBA ... 59

2.4.2 O USO DA JVM MICROSOFT COMO UMA BRIDGE ... 61

2.4.3 O USO DE COM EM JVM’S NÃO MICROSOFT ... 62

2.4.4 ACTIVEX E JAVABEANS ... 63

2.5 USO DE PRODUTOS COMERCIAIS ... 64

2.6 ENTERPRISE APPLICATION SERVERS ... 65

2.6.1 J2EE ... 67

2.7 A ARQUITETURA DE INTEGRAÇÃO J2EE 70
2.8 CONCLUSÃO 73
3 XML-RPC 75
3.1 DESCRIÇÃO DO PROTOCOLO 75
3.1.1 SINCRONISMO 76
3.1.2 AUSÊNCIA DE ESTADO 77
3.2 TIPOS DE DADOS 78
3.2.1 TIPOS DE DADOS SIMPLES 79
3.2.2 TIPOS DE DADOS COMPOSTOS 82
3.3 XML-RPC REQUEST 84
3.3.1 HTTP HEADERS 84
3.3.2 XML PAYLOAD 86
3.4 XML-RPC RESPONSE 87
3.5 CONCLUSÃO 91
4 SOAP 93
4.1 DESCRIÇÃO DO PROTOCOLO 93
4.1.1 O QUE SOAP NÃO CONTEMPLA 96
4.1.2 ESTRUTURA DE UMA MENSAGEM SOAP 98
4.1.3 TIPOS DE DADOS 99
4.2 A MENSAGEM SOAP 100
4.2.1 SOAP ENVELOPE 101
4.2.2 SOAP HEADER 103
4.2.3 O ATRIBUTO ENCODINGSTYLE 105
4.2.4 SOAP BODY 106
4.3 SOAP e HTTP 108

4.3.1 SOAP HTTP Request ... 109

4.3.2 SOAP HTTP Response ... 110

4.3.3 HTTP Extension Framework ... 110

4.4 ANÁLISE DO PROTOCOLO ... 111

4.5 PROBLEMAS PARA ALCANÇAR A INTEROPERABILIDADE ... 112

4.6 CONCLUSÃO ... 118

5 INTEROPERABILIDADE ... 120

5.1 INTEROPERABILIDADE COM/CORBA ... 121

5.2 ARQUITETURA DA SOLUÇÃO APRESENTADA – A BRIDGE XML ... 123

5.3 INTEROPERABILIDADE COM/CORBA VIA SOAP ... 126

5.3.1 PROBLEMAS DE INTEROPERABILIDADE SOAP ... 127

5.3.2 PROJETO DA SOLUÇÃO ... 129

5.3.3 FLUXO DAS MENSAGENS ... 136

5.3.4 GUI DO CLIENTE COM ... 142

5.4 INTEROPERABILIDADE COM/CORBA VIA XML-RPC ... 143

5.4.1 PROJETO DA SOLUÇÃO ... 144

5.4.2 FLUXO DAS MENSAGENS ... 147

5.4.3 GUI DO CLIENTE COM ... 149

5.5 ARQUITETURA DA BRIDGE XML PARA UMA AGENDA PESSOAL ... 149

5.6 *BRIDGES* CONVENCIONAIS VERSUS *BRIDGES* XML ... 154

6 Conclusão ... 159

Apêndice A ... 163

Apêndice B ... 171

Apêndice C ... 173

Referências **177**

Índice de Figuras

Figura 1 - Aplicação em duas camadas 4
Figura 2 - Aplicação com quatro camadas 8
Figura 3 - Um componente Bridge 11
Figura 4 - Web Service.. 15
Figura 5 - Diagrama OMA [OMG, 2002]... 22
Figura 6 - Fluxo geral de uma requisição CORBA [OMG, 2002] 26
Figura 7 - Arquitetura DCOM [Scribner & Stiver, 2000]....................................... 35
Figura 8 - Tipos de Servidores COM.. 38
Figura 9 - Gateway para interoperar COM e CORBA [Orfali & Harkey, 1998] 60
Figura 10 - Uso da JVM Microsoft para interoperar COM e CORBA.................... 62
Figura 11 - Arquitetura J2EE... 70
Figura 12 - Layout do pacote SOAP [Scribner & Stiver, 2000] 98
Figura 13 - O modelo de bridge usado nas aplicações 124
Figura 14 - A Calculadora Remota 132
Figura 15 - Os componentes envolvidos na solução SOAP *Messaging*............. 133
Figura 16 - Arquitetura da aplicação Calculadora com bridge SOAP *Messaging* 134
Figura 17 - Fluxo Principal 137
Figura 18 - GUI da Calculadora.. 142
Figura 19 - A arquitetura da aplicação Calculadora com bridge XML-RPC......... 145
Figura 20 - Fluxo Principal 148
Figura 21 - Arquitetura da aplicação Agenda 150
Figura 22 - GUI da aplicação Agenda submetendo uma operação de consulta . 151
Figura 23 - GUI da aplicação Agenda mostrando a resposta de uma consulta .. 152
Figura 24 - GUI da aplicação Agenda mostrando a resposta do servidor ASP... 153
Figura 25 - Arquitetura da aplicação Agenda com servidor CORBA 154

Abstract

CORBA and COM are often seen as competing technologies. Each one has its own strengths and the main difference between them resides in Operating Systems support. COM is totally oriented toward the Windows platform, while CORBA, since its inception, has been a multiplatform technology. In this study, we prefer not see them as competing technologies. To the contrary, we prefer to find a way to see them as cooperating. One specific draw back in both distributed computing approaches is that, if an application written with them needs to work within the Internet environment, there is a potential problem with firewalls. Most firewalls only allow HTTP traffic, which uses port 80. This means that a distributed system based on COM or CORBA that uses the Internet requires that network administrators leave open a range of numbers, which poses an unwelcome security risk. One solution to this problem is to make remote calls using HTTP protocol. This way, all traffic between the objects passes through port 80 and this eliminates the firewall problems. Another problem is in the necessity to exist symmetry of the used technologies in the modules that compose a distributed system. This work shows ways of gaining cooperation between the two worlds, COM and CORBA, in the Internet environment, through bridges using protocols based in XML and HTTP. Among all possible similar solutions, we choose working with SOAP and XML-RPC for building our own brige.

Resumo

CORBA e COM são tecnologias freqüentemente vistas como competidoras. Cada uma delas tem seus pontos fortes e a diferença principal entre as mesmas está no suporte que cada uma tem em diferentes Sistemas Operacionais. COM é orientado totalmente à plataforma Windows, enquanto CORBA é uma tecnologia que se propôs, desde a sua especificação inicial, a ser multiplataforma. Neste trabalho, nós não preferimos ver as tecnologias como competidoras. Ao contrário, preferimos encontrar uma forma de vê-las cooperando. Um inconveniente presente em ambas abordagens de computação distribuída é que, se uma aplicação implementada com elas precisar trabalhar usando a Internet há um problema potencial com firewalls. A maioria dos firewalls autoriza passar apenas o tráfego HTTP, que usa a porta 80. Isso significa que um sistema distribuído baseado em COM ou CORBA e que use a Internet necessita que os administradores de rede deixem uma faixa de números de portas abertas, o que abre brechas de segurança. Uma solução para esse problema é fazer chamadas remotas usando o protocolo HTTP. Com isso, todo o tráfego entre os objetos passa pela porta 80 e elimina-se o problema dos *firewalls*. Outro problema está na necessidade de existir simetria das tecnologias usadas nos módulos que compõem um sistema distribuído. Esta dissertação mostra formas de obter cooperação via Internet entre os dois mundos, COM e CORBA, através de *bridges* que usam protocolos baseados em XML e HTTP. Dentre as diversas soluções de

protocolos deste tipo, escolhemos trabalhar com SOAP e XML-RPC na construção de uma *brigde* própria.

1 INTRODUÇÃO

A história da computação foi marcada por mudanças profundas e freqüentes na maneira pela qual as pessoas interagem com os computadores. Um longo caminho foi percorrido desde que foram usadas as primeiras perfuradoras de cartões até chegarmos à interação com os computadores via redes mundiais, incluindo-se o uso de equipamentos móveis e comunicações sem fio. Não é de se admirar, portanto, que a forma na qual os sistemas computacionais são organizados também tenha sofrido mudanças drásticas.

No tópico seguinte, analisaremos etapas dessas mudanças, para contextualizar o nosso trabalho. Em seguida, destacaremos os objetivos desta dissertação.

1.1 RUMO AOS SISTEMAS DISTRIBUÍDOS BASEADOS EM OBJETOS

Os primeiros sistemas computacionais eram, do ponto de vista da arquitetura, monolíticos, pois todo o processamento era centralizado, sendo executado em uma única máquina e operado por seus usuários através de terminais. Nesta arquitetura, o *mainframe* era responsável pelo gerenciamento de tudo: da *interface* do usuário, dos dados e dos processos computacionais implementados. Era ao mesmo tempo o servidor (fazia todo o processamento) e o

cliente (através dos terminais ou de outras interfaces simples como leitoras de cartões).

Com o passar do tempo, tornou-se necessário dividir os elementos constituintes de um sistema computacional em mais de um computador. Além de outros aspectos, analisados a seguir, isso foi motivado e tornou-se possível, pelo surgimento, no início dos anos 70, dos microprocessadores e pelo uso da tecnologia de redes de computadores nos anos 80.

Inicialmente, buscou-se separar, em computadores diferentes, o armazenamento de dados, do processamento desses dados. Os benefícios dessa ação foram (1) o aumento da confiabilidade das aplicações (já que em um sistema monolítico, um problema sério de hardware pode obrigar a parar a execução de uma aplicação), (2) a melhoria da segurança e da performance no acesso às informações, uma vez que o banco de dados estava armazenado em um servidor em separado e (3) a possibilidade de melhorar a escalabilidade das aplicações, na medida em que se diminuía o custo em relação ao uso de grandes e caros *Mainframes* **[Orfali, Harkey & Edwards, 1999]**.

Esse modelo de arquitetura cliente-servidor original foi caracterizado, portanto, por clientes que concentravam o processamento da aplicação, estando os dados a serem processados por esses clientes, mantidos em um servidor de arquivos localizado em uma máquina em separado, estando clientes e servidores interligados por uma rede. Foi uma alternativa mais barata em relação aos *mainframes*.

Um passo adiante foi dado quando se buscou uma nova alternativa para separar os dados do seu processamento, mas que dividisse também o

processamento entre o cliente e o servidor. Substituíram-se, então, os servidores de arquivos pelos servidores de banco de dados. Neste esquema, as aplicações clientes usam SQL, um *driver* de acesso ao banco de dados e um *driver* de rede para gerenciar as informações armazenadas.

A partir daí buscaram-se formas de diminuir o grande tráfego nas redes de computadores, tanto no modelo cliente-servidor que usava servidores de arquivos, quanto naquele que usava os servidores de banco de dados relacionais. Neste último modelo de servidor, o congestionamento era causado, em parte, pela forma não procedural das sentenças SQL. O conjunto de dados resultante de uma consulta SQL devia, pelo fato da SQL não ser uma linguagem de programação, ser transmitido através da rede, do servidor para o cliente e, em caso de atualização, remetido de volta ao servidor, gerando um grande congestionamento no sistema.

O advento das linguagens de consulta e o uso de procedimentos armazenados no servidor (apesar de estes não serem portáveis) amenizaram este problema e caracterizou a arquitetura de duas camadas. Nesse esquema, o processamento está dividido entre o cliente e o servidor de transação, ficando o cliente com a parte do processamento não relacionada ao gerenciamento do banco de dados.

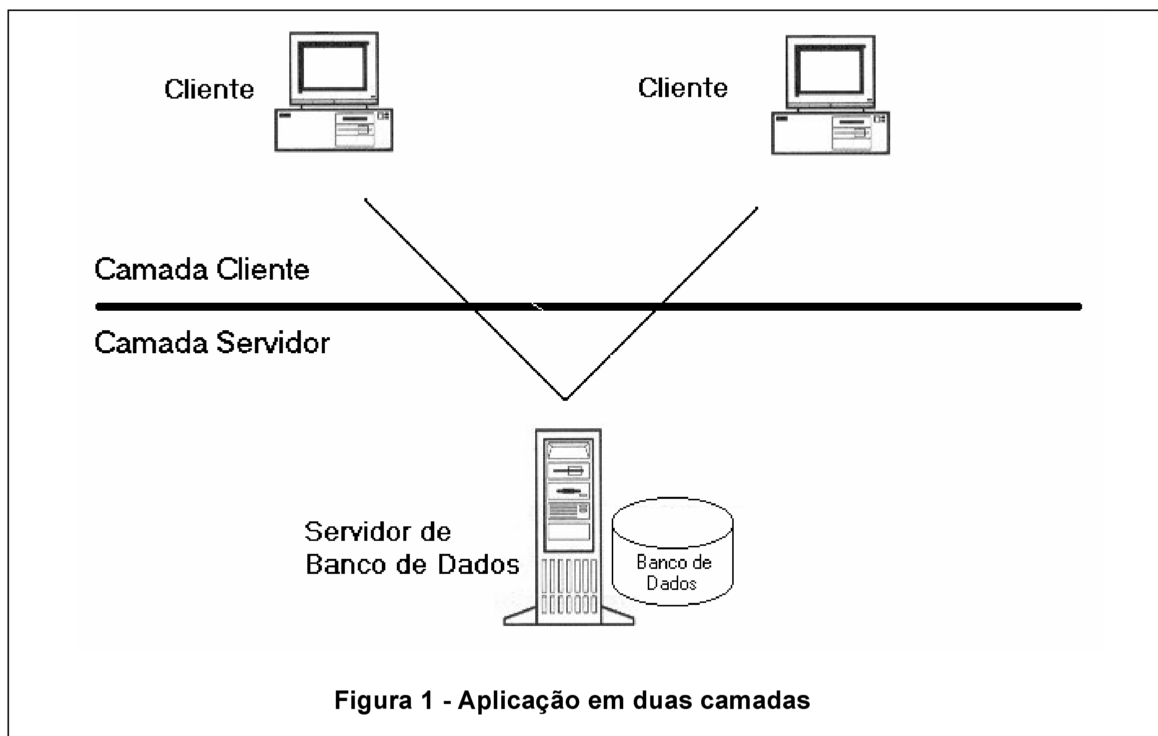

Figura 1 - Aplicação em duas camadas

Essa solução contém problemas afetando a arquitetura cliente-servidor, quer os clientes acessem apenas um ou mais de um servidor. Por exemplo, se um ou mais computadores servidores possuem diferentes sistemas operacionais e/ou diferentes servidores de BD, o cliente precisa ter *drivers* apropriados para acessar todos esses recursos. Como a atualização de *drivers* é uma prática constante na indústria da computação, por causa da evolução constante das tecnologias de hardware e software, os clientes devem não só possuir os *drivers* para estabelecer comunicação, mas a sua versão correta (não necessariamente a mais atual).

Além disso, há diversos outros inconvenientes que podem tornar-se problemas na medida em que a complexidade de uma aplicação cresce **[Orfali, Harkey & Edwards, 1999]**. Por exemplo, dentre outras deficiências da arquitetura de duas camadas encontradas em nossa pesquisa, em relação a outros

esquemas mais evoluídos de arquitetura que hoje conhecemos, consideramos de maior relevância:

- a administração das atualizações da lógica da aplicação, que precisa ser feita em muitos clientes;
- a segurança da aplicação (baixa, apenas no nível de dados);
- o encapsulamento dos dados (baixo, tabelas são expostas);
- a performance (deficiente, muitas sentenças SQL enviadas pela rede; consultas levam ao *download* de dados para processamento do cliente);
- o aumento limitado da escalabilidade;
- reuso da aplicação (baixo, devido a pouca segmentação);
- poucas alternativas para integração de aplicações legadas **[Orfali, Harkey & Edwards, 1999]**.

A solução buscada para os problemas acima, aliada à evolução das aplicações em geral, levou à separação do processamento do cliente em duas partes lógicas: a camada *interface* ou de apresentação e a camada da aplicação propriamente dita ou da lógica do negócio. Esta camada intermediária permitiu que os clientes ficassem mais simples, tendo em vista que a parte da lógica havia sido retirada deles. A comunicação dos clientes com os servidores de BD e seus sistemas operacionais tornou-se problema desses servidores intermediários da camada de aplicação.

As vantagens de se retirar da camada cliente a lógica do negócio e colocá-la em uma camada intermediária trouxe vários benefícios. Podemos enumerar, de forma relevante: (1) aumentar a *performance* dos sistemas, por ensejar maior

proximidade da camada intermediária com os dados do negócio sem importar onde os clientes estejam; (2) aumentar a escala das aplicações que passaram a poder tecnicamente responder a um número bem maior de clientes pela adição de servidores na camada intermediária; (3) tornar as aplicações cliente mais independentes do banco de dados, como já explicamos. O encapsulamento dos dados melhorou pois os mesmos são acessados indiretamente por serviços ou métodos da camada de negócio; (4) centralização na camada intermediária das regras de negócio que deixaram de estar espalhadas entre o servidor de banco de dados e os vários clientes, tornando mais fáceis a manutenção e o desenvolvimento dos produtos; (5) melhorou-se em muito a manutenção dos sistemas, uma vez que mudanças na lógica da aplicação e nos servidores de BD não implicavam mais em alteração do código de todos os clientes; (6) o nível de segurança também foi melhorado pois passou a haver discriminação de nível de segurança por tipo de serviço, método ou objeto. (7) o reuso de código também melhorou devido ao encapsulamento melhorado das partes de uma aplicação (em serviços e objetos divididos pelas camadas), o que (8) melhorou também as facilidades de desenvolvimento, pois o reuso significa menor esforço, menor custo e menor tempo. Além disso, a segmentação lógica das camadas de *interface* e de negócios fomentou o surgimento de ferramentas mais específicas para desenvolver cada uma dessas camadas **[Seely, 2001].** Assim, apareceram ferramentas para desenvolvimento de interfaces, para criação componentes de negócio, para gerenciamento e *tunning* de SGDBs.

Há outros argumentos apontados por alguns pesquisadores para justificar o avanço da arquitetura de três camadas sobre a de duas camadas, mas julgamos que esses acima descritos já são suficientes para embasar essa idéia.

A camada intermediária (também chamada de *middleware* por Jason Pritchard **[Pritchard, 1999]** e por **[Orfali, Harkey & Edwards, 1999]**), por implementar os serviços do negócio, pode ainda ser muito complexa, dependendo, é claro, da natureza da aplicação. De tal forma, que os sistemas podem vir a serem implementados em mais de três camadas, através do desdobramento da camada intermediária.

Um típico exemplo de uma aplicação n-camadas é uma aplicação Web envolvendo *scripts* ASP ou *Servlets*, por exemplo. Tal aplicação é dividida nas seguintes camadas: (1) uma camada cliente, composta pelos *browsers*, como Netscape e Explorer; (2) a camada de apresentação, constituída pelos servidores Web, como Tomcat e IIS, rodando os scripts que fazem o processamento e geram as *home-pages*; (3) a camada de negócio que podem conter os servidores de aplicação, como o JBoss, que executam os componentes ou scripts de negócio residentes (alternativamente esses componentes de negócio podem rodar sobre os servidores de Internet, mas não têm relação direta com a geração de páginas) e (4) o servidor de banco de dados.

Nesse tipo de esquema, mostrado na figura 2, foi a evolução do *middleware* que veio tornar mais fácil a comunicação entre as N camadas de uma aplicação cliente-servidora multicamadas.

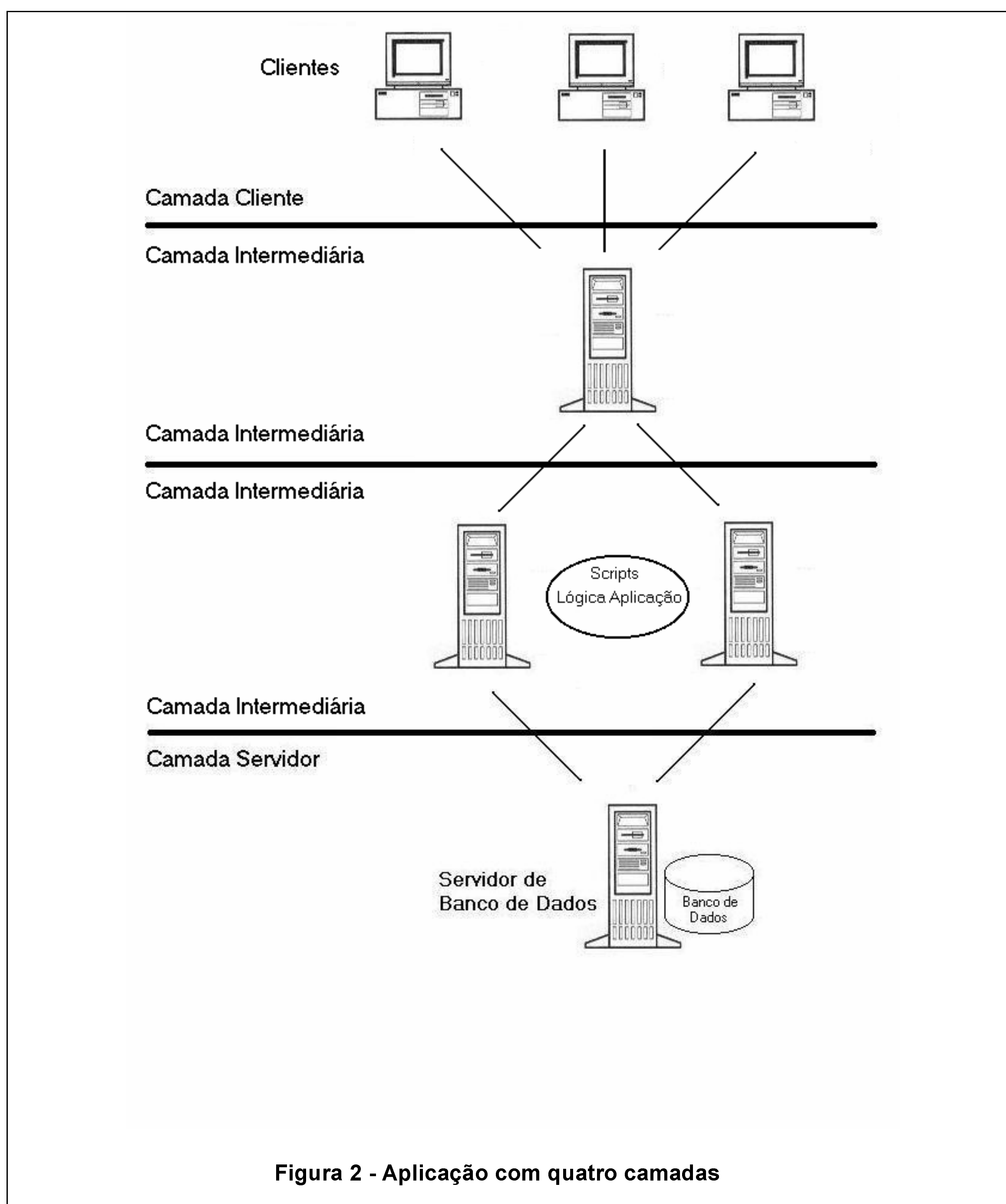

Figura 2 - Aplicação com quatro camadas

O *middleware*, principal objeto dessa dissertação, é um software de conectividade que consiste de um conjunto de serviços que permitem a múltiplos processos, executando em diferentes computadores (ou em um mesmo

computador), interagirem (através de uma rede, no caso dos processos estarem distribuídos em mais de uma máquina). O *middleware* situa-se, portanto, entre o cliente e o servidor, formando, conjuntamente com a lógica da aplicação, a parte intermediária de uma arquitetura cliente-servidor de N camadas. O objetivo do *middleware* é fornecer aos clientes acesso semelhante às aplicações e aos dados, sem importar as plataformas, os sistemas operacionais e servidores de BD em que eles estão situados. Os projetistas podem usá-lo, portanto, para acomodar vários protocolos, plataformas e linguagens diferentes, além de trocar mensagens entre aplicações.

Em sistemas de duas camadas, a tarefa de abstração dos vários protocolos da rede é realizada pelos *drivers* e pelas bibliotecas especializadas fornecidos pelos fabricantes de BD relacionais. Com esse instrumental, as aplicações clientes podem ser escritas sem se importar com detalhes, como a localização real do banco de dados. Já os sistemas com N camadas requerem uma infra-estrutura de comunicação entre as camadas muito mais sofisticada. Isto levou a uma evolução permanente do *middleware* desde o final dos anos oitenta [**Pritchard, 1999**].

A primeira tecnologia *middleware* a ter larga aceitação foi a RPC (*Remote Procedure Call* – chamada a procedimento remoto), usada para fazer chamadas a funções que eram executadas em máquinas remotas. Na medida em que as linguagens orientadas a objetos foram ganhando espaço, o *middleware* evoluiu na direção da semântica da orientação a objetos.

O OMG (Object Managment Group), um grupo formado por vários fabricantes de hardware e software, lançou uma especificação, CORBA (Common Object Request Broker Architecture), que fornece as linhas gerais de uma

arquitetura para distribuir objetos através de linguagens, processos e plataformas. Essa arquitetura permite que qualquer objeto em qualquer plataforma emita mensagens para outro objeto em qualquer outra plataforma. O padrão CORBA foi o primeiro enfoque a permitir, por exemplo, que um objeto Java, rodando no Windows, transparentemente emita mensagens a um objeto Smalltalk em um servidor UNIX ou a um objeto COBOL em um *mainframe*.

A Microsoft, por outro lado, após uma série de evoluções desenvolveu o COM+, uma arquitetura que permite a objetos cruzar limites de linguagens e processos, mas com problemas de configuração e uso em outras plataformas, que não o Windows.

A camada três, da aplicação de quatro camadas descrita acima e representada na figura 2, pode fazer uso das tecnologias COM+ e CORBA, fazendo com que, dentro do universo de cada tecnologia, componentes de negócio possam cooperar transparentemente entre si.

Mas um problema passou a existir com as fusões empresariais e criação de mega-corporações, bem como com os avanços da globalização, que terminou espalhando os departamentos e filiais das empresas por todo o mundo ou dentro de um mesmo território (mesmo as empresas que não são assim tão gigantes).

Ficou difícil integrar as aplicações existentes, muitas vezes usando linguagens, plataformas (hardware mais SO) e *middlewares* diferentes. E esse aspecto (da diferença entre as tecnologias) deu-se em muitos lugares por razões distintas, mas em geral porque nunca se conseguiu uma unanimidade sobre qual tipo de linguagem, plataforma ou *middleware* é o melhor **[McLaughlin, 2001]**. E essa batalha dos componentes tornou muito difícil alcançar a interoperabilidade.

Tanto que Chad Vawter e Ed Roman June **[Vawter, 2001]**, criaram a figura do “*middlemen*”, como instância autêntica de um *middleware* humano, onde a integração mostrou-se impossível ou onde os custos da mesma são injustificáveis.

O passo seguinte na arquitetura multicamadas é a criação de componentes para servir de pontes (*bridges*) entre modelos de objetos distribuídos distintos (ver figura 3, abaixo), de maneira que objetos de arquiteturas diferentes possam interagir. Inclusive usando a WEB.

Como aparece na figura 3, o componente *bridge* atua como um cliente CORBA e como um servidor COM (para efeito de exemplo, poderia ser o inverso). Ou seja, ele precisa ser um componente COM e CORBA (ao mesmo tempo), que consegue ser reconhecido como parte de sistemas distribuídos construídos com ambas as tecnologias, de maneira que possa mapear uma chamada realizada em COM, por exemplo, em uma chamada CORBA (ou vice-versa), a fim de que os sistemas possam interoperar.

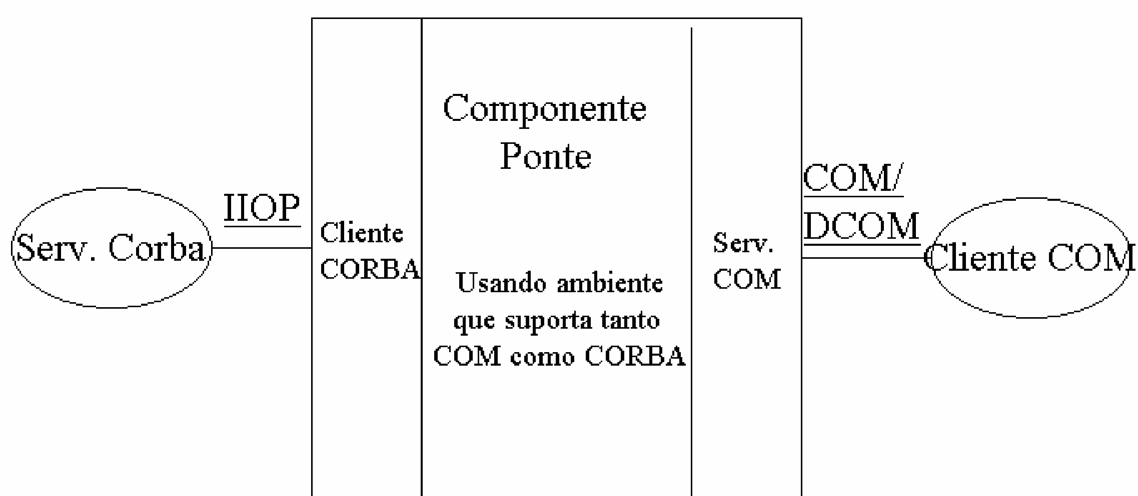

Figura 3 - Um componente Bridge

Mas mesmo usando essa abordagem, que nunca se mostrou simples e eficiente, nem todos os problemas foram resolvidos, pois houve logo o desejo e a necessidade das corporações usarem a internet como camada de transporte para suas aplicações distribuídas. Aspectos econômicos à parte, a internet passou a ser considerada como meio de transporte de mensagens e trocas de dados para as aplicações distribuídas, que muitas vezes precisam burlar as barreiras dos *firewalls*. Além disso, ultimamente a internet passou a ter importância ainda maior como um modo de integração e de compartilhamento de processos e informações em uma escala macro, simplificando a colaboração *business-to-business* (B2B).

Pensando em superar dificuldades com as arquiteturas distribuídas em geral, um consórcio liderado pela Microsoft lançou, em setembro de 99, as bases de uma nova tecnologia, chamada SOAP (Simple Object Access Protocol). Essa tecnologia, que especifica o uso de XML (padrão atual da WEB para troca de dados) e HTTP (SOAP pode ser usado em combinação com qualquer protocolo, embora só tenha sido usado até o momento com HTTP), como infra-estrutura básica para chamadas remotas semelhante a RPC, pode ser usada para fornecer um mecanismo para proporcionar que objetos distribuídos possam conversar via internet.

De forma semelhante a SOAP, outro protocolo que define uma forma para que as chamadas RPC sejam serializadas em documentos XML e assim enviadas através de uma conexão HTTP é o XML-RPC. No entanto, XML-RPC, em geral, é bem mais simples, e limitado, que SOAP. Enquanto o primeiro segue um enfoque procedural, o segundo é firmemente centrado no paradigma de orientação a

objetos. Uma mensagem XML-RPC usa apenas o protocolo HTTP e tem somente o nome do método e a lista de parâmetros. Cada parâmetro inclui um elemento descrevendo o tipo do parâmetro assim como um elemento com o valor do parâmetro. Já uma mensagem SOAP, além de poder usar qualquer protocolo (não só o HTTP), possui uma estrutura bem mais complexa, possibilitando passar um objeto real, com todo seu estado interno, de uma aplicação para outra. Imaginando as possibilidades, por exemplo, um objeto *funcionário* criado em um servidor Python pode ter seu salário aumentado por um cliente Perl ou Java. Há também semelhanças e diferenças sintáticas, como por exemplo, a tag <methodCall> encapsula todo o XML-RPC payload, similar a tag SOAP <Envelope>. Como diferença sintática, o nome do método em XML-RPC é um valor do elemento (tag) <methodName>, enquanto em SOAP o nome do método serve para formar o nome do elemento (nome da tag) que descreve a chamada. Finalmente, os parâmetros, em XML-RPC, são codificados com um elemento de informação de tipo (como <string>) e SOAP, ao contrário, permite o uso de XML *Schemas* para descrever tipos de dados.

Conforme mostraremos no capítulo cinco, pelo fato da complexa especificação SOAP não ser muito específica, em alguns pontos, há mais problemas para interoperar implementações SOAP diferentes do que fazer implementações XML-RPC distintas conversarem.

Também surgiram outras tecnologias baseadas em XML nos últimos dois anos e hoje formam um leque de mais de vinte ferramentas, que permitem criar aplicações distribuídas usando as infra-estruturas existentes para a web (ver site W3C **[XML (W3C), 2001]**).

Nos últimos dois anos, na medida em que essa onda XML avançava, também foi tomando forma um serviço novo na web que ainda está se definindo e evoluindo a cada dia, de maneira que não há ainda uma conceituação formal, amplamente aceita e padronizada sobre ele. Trata-se dos Web Services.

Como explicaremos com mais detalhes ao longo da dissertação, os Web Services não foram criados com esse propósito, mas podem servir de instrumento para proporcionar a interoperabilidade entre COM e CORBA. Isto porque, para se conseguir com que os sistemas cooperem é necessário que falem uma linguagem comum. Uma linguagem na qual ainda que as palavras da comunicação não sejam as mesmas, as aplicações possam ter acesso às palavras e possam traduzi-las em algo que possa ser compreendido.

Então, mais uma vez, XML é parte dessa solução. Mas a chave para a interoperabilidade não é apenas isso, mas também permitir responder questões do tipo: que informação pode-se conseguir de um certo sistema? Onde se pode encontrá-lo? Ou seja, deve existir um meio de saber quais serviços uma aplicação está tornando disponível na WEB. Algo parecido com os serviços de "paginas amarelas" **[Sharma, 2001]** dos catálogos telefônicos. UDDI (*Universal, Discovery, Description and Integration*) é uma tecnologia recente que fornece meios de registrar e descobrir serviços na WEB. A WSDL (*Web Services Description Language*) é uma linguagem de descrição que oferece uma maneira de disponibilizar a informação necessária sobre um serviço, a fim de permitir a um cliente interagir com ele. Uma visão geral do processo envolvendo Web Services pode ser visto na figura abaixo:

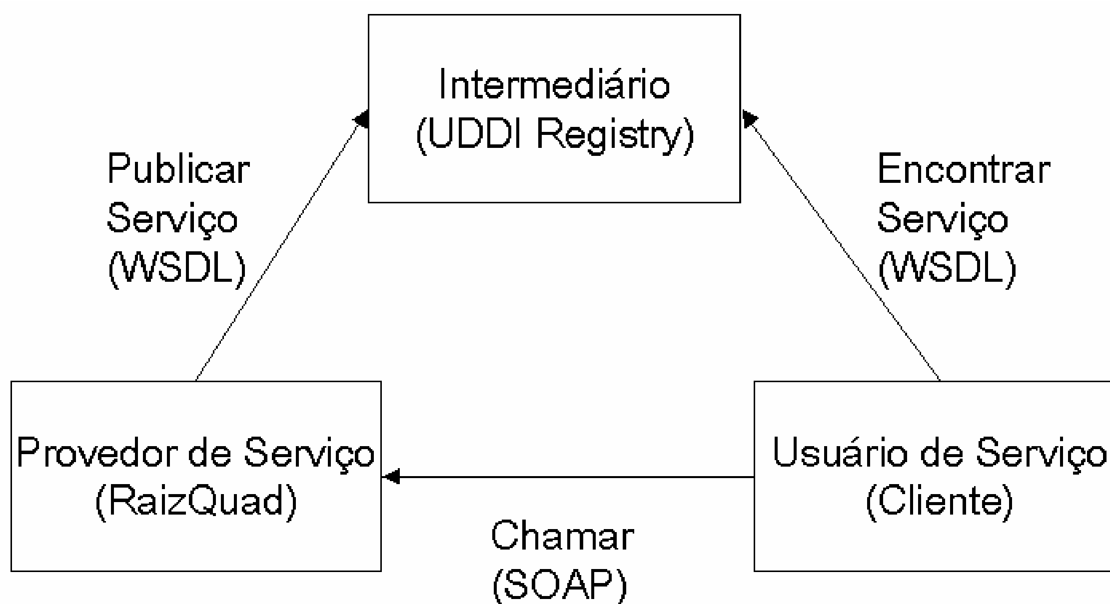

Figura 4 - Web Service

Como retrata a figura 4, suponhamos que existe um certo serviço implementado e disponibilizado na WEB, como um cálculo simples de raiz quadrada, por exemplo. Através de UDDI pode-se registrar o serviço, usando-se uma descrição do mesmo no formato WSDL. Também através de UDDI, um cliente pode pesquisar por um serviço específico, fornecendo uma certa descrição. UDDI, como resposta da pesquisa, devolve todos os registros (no formato WSDL) dos serviços que combinam com a descrição enviada. Uma vez que o cliente encontrou o serviço desejado, ele pode interagir com esse serviço, usando o URL do serviço e os parâmetros apropriados, devidamente codificados em uma chamada SOAP.

Uma vez que já discorremos sobre o histórico da arquitetura dos sistemas computacionais, resta-nos enquadrar nosso trabalho neste contexto.

Nesta dissertação, mostraremos uma nova maneira de construir uma *bridge* entre modelos de objetos distribuídos distintos, COM e CORBA, usando esses protocolos baseados em XML, especificamente SOAP e XML-RPC, obtendo, assim, os benefícios que essas abordagens oferecem.

1.2 DESCRIÇÃO DO PROBLEMA E PROPOSTA DE SOLUÇÃO

Toda comunicação entre objetos requer que um objeto possa encontrar um outro objeto, solicitar a execução de um serviço e receber uma resposta. A informação trocada entre os objetos necessita ser colocada em um formato apropriado, a fim de que possa ser compreendida por ambos.

As arquiteturas COM e CORBA oferecem vantagens distintas para se criar aplicações baseadas em objetos distribuídos. Mas, uma vez que elas são ricas em termos de funcionalidade, são, conseqüentemente, difíceis de implementar. Elas também tendem a ser simétricas, significando que, quando aplicações distribuídas são construídas usando-se uma dessas arquiteturas, o mesmo modelo de objeto distribuído é requerido em todos os pontos.

Além desses inconvenientes, se a aplicação distribuída necessitar funcionar via internet há um problema potencial com os *firewalls*. Geralmente, sistemas que usam TCP/IP associam uma porta diferente para cada serviço que o sistema aceita. Cada requisição (*request*) que usa um serviço particular, carrega consigo um correspondente número de porta. HTTP, por exemplo, está associado à porta 80, enquanto FTP usa a porta 21. Os *firewalls* operam, ao bloquear um serviço específico, rejeitando todo o tráfego enviado à porta que o serviço está usando.

Em acréscimo, também bloqueiam, em algumas situações de risco potencial, todas as portas que não estão em uso e as portas especiais, como, por exemplo, a porta 135 (RPC *Endpoint Mapper*) **[Scribner, 2000]**, por razões de segurança. Por isso, o *firewall* consegue impedir todo o tráfego. Isso cria um problema para os protocolos de objetos distribuídos (IIOP, GIOP, ORPC etc), porque eles não têm um número de porta padrão único. Geralmente, esses protocolos usam portas associadas dinamicamente e escolhidas aleatoriamente quando é necessário.

Portanto, uma solução distribuída baseada em RPC, como CORBA ou COM, que requeira que um número arbitrário de portas seja aberto, não funcionará quando um objeto servidor se posicionar atrás de um *firewall*, uma vez que o funcionamento da mesma traria um risco indesejável de segurança.

Há soluções propostas para resolver esse problema, como uma em que o *firewall* permitiria tráfego através de portas em uma certa faixa de numeração **[Adam, 2001]**, o que, obviamente, abre uma brecha de segurança.

Outra opção é colocar o objeto remoto COM/CORBA em um servidor *farm* e usar HTTP para acessar o objeto, a partir de uma máquina cliente remota (ver site Sun **[www1]**). Mas, claro que um enfoque mais genérico é preferível.

A solução para esse problema, bem como para a necessidade da simetria tecnológica pode ser alcançada com as tecnologias de Web Services, em geral. Ou seja, em vez de se usar o protocolo específico de cada tipo de objeto distribuído para se obter a comunicação entre eles, podemos fazê-los comunicar-se usando o protocolo HTTP-POST para enviar mensagens empacotadas em XML. Portanto, todo o tráfego passa pela porta 80, o que significa que os protocolos dos Web Services podem ser usados, sem problemas, na internet e

com *firewalls,* além de dispensarmos a simetria tecnológica. Tal fato, se sistematizado, pode tornar viável a qualquer par de objetos do universo COM+ e CORBA conversar entre si, transparentemente.

Esta dissertação vai mostrar um modelo de solução que pode ser empregado para esses problemas de interoperabilidade em sistemas distribuídos. Esse modelo proposto leva à integração dos diferentes modelos de objetos, bem como mostra como que esses objetos podem comunicar–se via internet. Isto proporciona um mecanismo, através do qual podem-se integrar sistemas heterogêneos, com serviços, componentes e aplicações diferentes.

Esta dissertação assume que os leitores estão familiarizados com as tecnologias CORBA, COM e XML, apesar de dedicar um espaço para descrever e resumir as duas primeiras.

No capítulo dois abordaremos as estratégias clássicas para se construir *bridges* entre COM e CORBA. Mostraremos soluções baseadas em mapeamento de chamadas, no uso e construção de componentes padrões, no uso de soluções comerciais e enfocaremos as principais ferramentas de integração. No capítulo três, baseado principalmente na especificação, discutiremos a tecnologia XML-RPC. No quarto capítulo, mostraremos o estágio atual de SOAP, apontando alguns usos atuais da tecnologia e no capítulo cinco mostraremos soluções implementadas com as estratégias aqui comentadas, que fazem objetos distribuídos de tecnologias e plataformas distintas, interoperarem via Internet.

1.3 CONCLUSÂO

Os protocolos baseados em XML alcançaram uma posição de destaque na computação nos últimos quatro anos, tornando-se uma ferramenta flexível para ser usada na solução de um grande número de problemas. Mostraremos que um dos problemas que ela é capaz de resolver é o da interoperabilidade entre objetos de sistemas distribuídos de arquiteturas diferentes, como COM e CORBA, funcionando através da Internet. Esta monografia propõe um modelo para interoperar COM e CORBA e, através da descrição dos mecanismos de implementações distintas dessa mesma idéia, prova que isso é possível.

2 COM, CORBA e BRIDGES

Uma vez que é consensual que é melhor reusarmos componentes de software pré-existentes em vez de criarmos novos a partir do zero **[Bachmann, 2000]**, mostraremos, neste capítulo, as diferentes maneiras de integrarmos COM e CORBA ao criarmos soluções baseadas em objetos distribuídos.

Em princípio, descreveremos as arquiteturas de COM (principalmente a parte DCOM) e CORBA. Veremos que as diferenças entre essas arquiteturas são, em maior parte, sintáticas, pois os seus mecanismos e funcionalidades gerais são bastante similares, já que ambos servem ao mesmo propósito. Depois enfocaremos as *bridges* COM/CORBA sob três perspectivas: *bridges* baseadas no uso de componentes especializados (feitos sob medida para uma certa aplicação), bridges comerciais de uso geral e finalizaremos abordando os servidores de aplicação *Enterprise* e suas estratégias de integração, dando destaque à especificação J2EE.

Apesar de nós não termos encontrado nenhuma categorização formal, verificamos que as *bridges* podem ser uni ou bi-direcionais, significando que há *bridges* específicas para clientes/COM-servidores/CORBA e vice-versa, além de *bridges* que servem para interoperar ambos, ao mesmo tempo. Neste trabalho estamos preocupados com as *bridges* bi-direcionais, apesar de perceber que na prática isso pode não ser uma grande vantagem em muitos casos. Além desta

característica, nossa proposta de *bridge* COM/CORBA é diferente das anteriores mencionadas, porque usa os serviços de tecnologias baseadas em RPC e XML.

2.1 CORBA

Formando um conjunto que conta com mais de 800 das maiores organizações da industria de software, o OMG (*Object Management Group*) nasceu em 1989, com o objetivo de definir uma infra-estrutura que permitisse o desenvolvimento de aplicações distribuídas em sistemas heterogêneos. Como um resultado dos muitos trabalhos deste grupo, surgiu a especificação CORBA (*Common Object Request Broker Architecture*), elemento central da OMA (*Object Managment Architecture*).

CORBA especifica como um programa, a partir de quase qualquer organização, computador, linguagem, sistema operacional ou rede, pode interagir, usando os protocolos padronizados GIOP e IIOP, com outro programa da mesma ou de outra organização, computador, linguagem, sistema operacional ou rede **[OMG, 2002]**.

A OMA descreve como objetos distribuídos e as interações entre eles podem ser especificadas, através de dois modelos relacionados: o Modelo de Objeto, que define como as interfaces de objetos distribuídos são descritas através de um ambiente heterogêneo; e o Modelo de Referência que caracteriza as interações entre esses objetos.

O Modelo de Objeto define um objeto como uma entidade encapsulada com uma identidade imutavelmente distinta **[Hennig & Vinoski, 1999]**, da mesma

forma que objetos convencionais, cujos serviços podem ser acessados através de uma interface. A implementação e a localização do objeto são transparentes para os seus clientes.

O Modelo de Referência define quatro categorias de interfaces, cada uma com serviços distintos, integrados por um ORB (*Object Request Broker*): *Application Objects, Vertical CORBA Facilities, Horizontal CORBA Facilities, CORBA Services*.

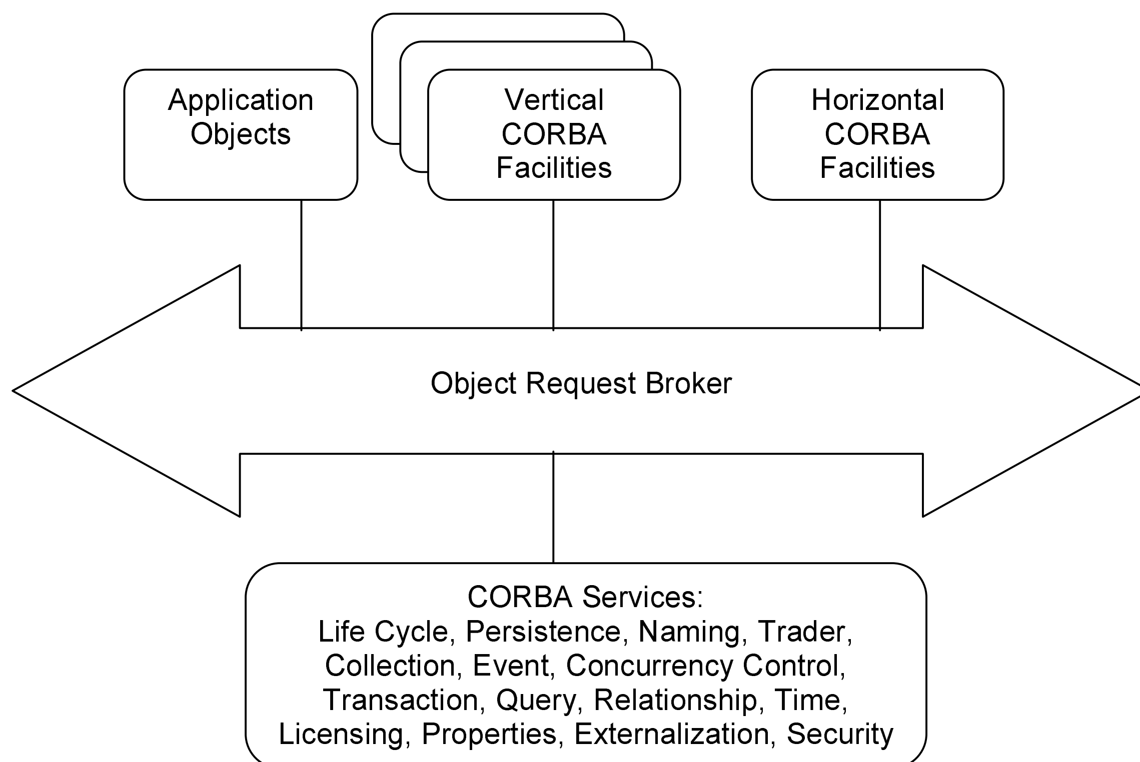

Figura 5 - Diagrama OMA [OMG, 2002].

Um ORB permite a comunicação entre clientes e objetos servidores, ativando, transparentemente, os objetos remotos que não estão executando quando as chamadas são feitas. O ORB também possui uma interface que pode

ser usada diretamente por objetos clientes e servidores. É um pseudo-objeto **[Liberty, 19999]** que deve estar sempre disponível nos ambientes de execução cliente e servidor. As categorias de interface mostradas acima usam os serviços de comunicação e ativação do ORB **[Orfali, Harkey & Edwards, 1999]**. Alguns produtos ORB incluem serviços CORBA (*CORBA services*) básicos nos seus pacotes, mas outros não. Outras empresas de software oferecem apenas os serviços, sem um ORB **[OMG, 2002]**.

Os *Application Objects* são desenvolvidos especificamente para uma aplicação. Os *CORBA Services* são coleções de interfaces de serviços de sistema usadas por muitas aplicações distribuídas, independentemente do domínio da aplicação, como o *OMG Naming Service*, que permite obter a localização de componentes através do nome, e o *OMG Trader Service*, que possibilita aos objetos publicar seus serviços. *Vertical CORBA Facilities* são interfaces usadas por aplicações distribuídas que fazem parte e são específicas de um certo domínio de aplicação (saúde, telecomunicações, indústria etc). Na figura 5, acima, os múltiplos retângulos indicam a multiplicidade dos domínios. *Horizontal CORBA Facilities* são interfaces cujas funcionalidades situam-se entre *CORBA services* e *Application Objects*, sendo, portanto, serviços potencialmente úteis a vários domínios de aplicação, como *Printing Facility, Secure Time Facility, Internationalization Facility,* e *Mobile Agent Facility*. Esta é a única categoria da OMA que não possui uma força tarefa trabalhando nela **[OMG, 2002]**.

2.1.1 O VOCABULÁRIO CORBA

Como toda tecnologia, a arquitetura CORBA tem uma terminologia associada a ela. Os significados de alguns desses termos têm relação com termos de outras tecnologias, enquanto outros são novos e outros têm sentidos diferentes. A relação de termos que se segue não pretende ser exaustiva e tem a intenção de conceituar alguns elementos envolvidos na tecnologia CORBA e que usaremos nas explicações seguintes. Esses conceitos serão refinados ao longo deste capítulo.

Um objeto CORBA (*CORBA object*) é especificado por uma interface, implementada em uma linguagem padrão chamada IDL. Essa interface define um contrato que descreve as capacidades do objeto distribuído no sistema em questão. É, portanto, uma entidade virtual **[Hennig & Vinoski, 1999]**, no sentido de que ele realmente não existe até que o mesmo seja implementado por sentenças de uma linguagem de programação. Um objeto CORBA deve ser localizado por um ORB em um sistema distribuído, a fim de receber chamadas remotas de clientes.

Um cliente (*client*) é a entidade que faz uma requisição (chamada) a um objeto CORBA, podendo o primeiro existir no mesmo espaço de endereçamento que o segundo, ou em outro completamente em separado.

Uma requisição (*request*) é uma chamada de uma operação em um objeto CORBA por um cliente.

Um objeto alvo (*target object*) é o objeto CORBA que é o receptor de uma certa requisição. O modelo de objeto CORBA é um modelo de expedição única

(*single-dispatching model*) no qual o objeto alvo de uma requisição é determinado unicamente pela referência (*object reference*) usada para fazer o pedido.

Um servidor (*server*) é uma aplicação na qual um ou mais objetos CORBA, invocados em uma certa requisição, existem. Requisições seguem de um cliente para um objeto alvo em um servidor e o objeto alvo envia os resultados de volta em uma resposta (*response*), caso a requisição requeira uma.

Uma referência de um objeto (*object reference*) é um ponteiro usado para identificar e localizar um objeto CORBA. Para os clientes, as referências são entidades opacas **[Orfali & Harkey, 1998]**. Clientes usam referências para dirigir requisições a objetos, mas eles não podem criar referências das suas partes constituintes, nem podem acessar ou modificar o conteúdo de uma referência. Cada referência está associada somente a um único objeto CORBA.

Um *stub* é um *proxy* local para um objeto servidor remoto, cujo código é gerado por um compilador IDL, a partir da interface do objeto remoto. As chamadas do cliente para o objeto servidor são feitas através de um *stub* que as repassa para o ORB cliente. Deve existir um *stub* para cada objeto que um cliente usa no servidor.

Skeletons são classes servidoras e, da mesma forma que os *stubs*, são também geradas por um compilador IDL, que fornecem interfaces estáticas para cada objeto exportado do servidor. Estende-se o *skeleton* (através de herança), a fim de poder fornecer a funcionalidade necessária.

Um *servant* é uma entidade que implementa um ou mais objetos CORBA e que existe dentro do contexto de uma aplicação servidora. Em Java e C++,

servants são instâncias de classe, que estendem os *skeletons* e sobrescrevem as funções definidas na interface IDL do objeto CORBA.

Marshaling é o nome que se dá ao processo de empacotar dados em um formato padrão, de tal forma que esses dados possam ser transmitidos para um objeto distribuído, garantindo assim a sua integridade. *Unmarshaling* é o nome que se dá ao processo de desempacotar os dados transmitidos. *Marshaling* e *unmarshaling* são processos similares a serialização e desserialização que ocorre com os objetos Java.

2.1.2 FUNDAMENTOS E CARACTERÍSTICAS

A figura 6 mostra os relacionamentos entre a maioria das características de CORBA. Nela, a aplicação cliente faz requisições e a aplicação servidora a recebe e trata apropriadamente.

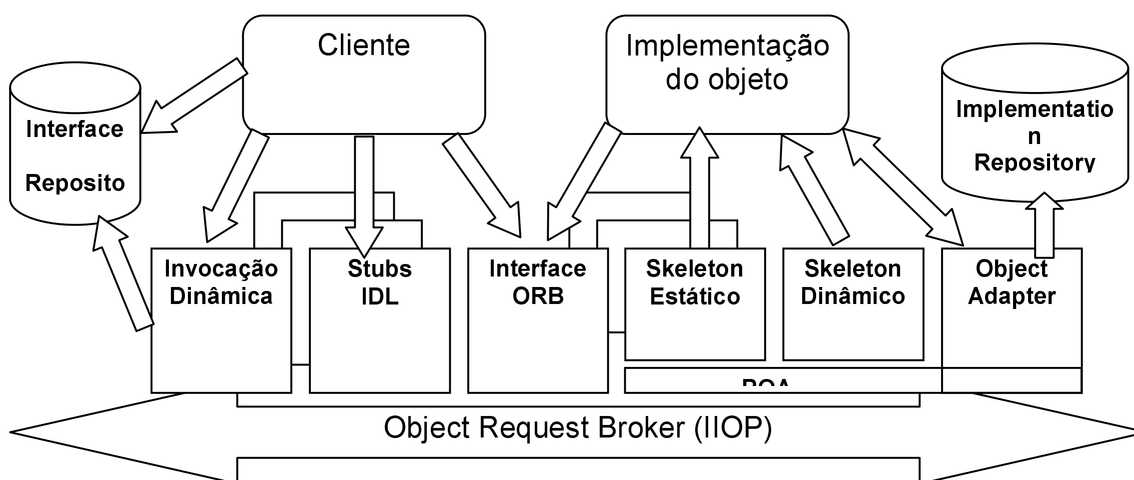

Figura 6 - Fluxo geral de uma requisição CORBA [OMG, 2002]

O cliente dirige suas requisições para o ORB vinculado ao seu processo. O cliente pode tanto escolher fazer as requisições usando *stubs* estáticos, gerados a partir da interface de definição do objeto e implementado em uma linguagem de programação, quanto usando a DII (*Dynamic Invocation Interface*). A DII permite descobrir métodos (que serão invocados) em tempo de execução, através do uso de API´s padronizadas para a busca dos metadados (no *Interface Repository*) que definem a interface do objeto servidor. A DII também fornece um conjunto de API´s para gerar dinamicamente os parâmetros, fazer a chamada remota e receber os resultados.

O *Interface Repository* é um banco de dados distribuído **[Orfali, Harkey & Edwards, 1999]** que contém versões das interfaces (IDL) de objetos CORBA. As API´s do *Interface Repository* permitem obter e modificar as descrições de todas as interfaces dos objetos registrados, os métodos que eles suportam e os seus respectivos parâmetros. A especificação CORBA chama essas descrições de *method signature*. As API´s ainda permitem que novas informações sejam acrescentadas no *Repository*. Para isso, o compilador IDL associa um ID a cada tipo em uma especificação. Esse ID consiste em um identificador único para cada tipo IDL (componente, objeto) e é usado como uma chave no *Interface Repository*, onde a correspondente definição de tipo está armazenada.

A interface ORB consiste de poucas API´s para serviços locais que podem ser de interesse para uma aplicação (como converter uma referência em uma string e vice-versa). Essas chamadas podem ser úteis se é necessário armazenar e enviar referências.

Continuando a descrição da figura 6, o ORB cliente transmite a requisição para o ORB da aplicação servidora, que por sua vez a despacha para o *object adapter* (que criou o objeto alvo). O *object adapter* despacha a requisição para o *servant* que está implementando o objeto alvo. Da mesma forma que o cliente, o servidor pode escolher entre o mecanismo de despacho estático (*skeletons*) ou dinâmico (DSI – *Dynamic Skeleton Interface*) para os seus *servants*. Este, entretanto, não diferencia entre uma chamada estática ou dinâmica, pois ambas têm a mesma semântica **[Orfali, Harkey & Edwards, 1999]**. Depois que o *servant* processa a requisição, ele retorna uma resposta pra a aplicação cliente.

É o *object adapter* quem aceita as requisições de serviço em nome do objeto alvo. Conforme descrito pelo *Adapter Design Pattern* **[Gamma et al, 1995]**, que é independente de CORBA, um *object adapter* é um objeto que adapta a interface de um objeto a uma interface diferente esperada pelo chamador.

Ou seja, um *object adapter* é um objeto interposto que usa delegação para permitir a um chamador passar requisições para um objeto sem saber qual a verdadeira interface do objeto. Ele fornece o ambiente para instanciar os *servants*, passando as requisições para eles e associando-os aos IDs (a especificação CORBA chama-os de *IDs object references* **[OMG, 2002]**).

O *object adapter* também registra as classes que ele suporta e as suas instâncias no *Implementation Repository*. A especificação ainda diz que cada ORB pode suportar mais de um *adapter*, mas obrigatoriamente tem que suportar o *object adapter* padrão chamado POA (*Portable Object Adapter*).

O POA é o segundo *object adapter* que o OMG especifica. O primeiro foi o BOA (Basic Object Adapter), projetado em 1990, mas agora obsoleto. Entre outras

coisas, o POA é capaz de instanciar objetos que não estão ativos quando há uma necessidade, e desativá-los quando não há nenhum trabalho a fazer, gerenciando os recursos do servidor, visando adequá-los à escala da aplicação. Também é possível fixar os padrões de alocação e desalocação de recursos através da configuração dos *POA Policys* (existem sete ao todo – quatro deles foram institucionalizados como alternativas pré-codificadas: *Service, Session, Process e Entity*). Através deles pode-se determinar se os objetos criados são persistentes ou transientes; se a ativação é por chamada e qual a relação entre os objetos CORBA e os *servants* **[Hennig & Vinoski, 1999]**.

Sem object adapters, o ORB teria que diretamente efetuar as suas funções, além de todas as suas responsabilidades e, como resultado, teria uma interface muito complexa, de difícil gerenciamento pelo OMG e o número de possíveis estilos de implementação dos *servants* seria limitado **[Orfali, Harkey & Edwards, 1999]**.

A DSI (*Dynamic Skeleton Interface*) fornece um mecanismo de ligação em tempo de execução para servidores que necessitam tratar chamadas para componentes que não têm *skeletons* estáticos (gerados pelo compilador IDL). O *skeleton* dinâmico procura, nos parâmetros da chamada, a identificação do objeto alvo e do método envolvidos na requisição. *Skeletons* dinâmicos são muito úteis para implementar *bridges* genéricas entre ORBs **[Orfali, Harkey & Edwards, 1999]** e eles também podem ser usados por interpretadores e linguagens de script para gerar, dinamicamente, implementações de objetos **[Hennig & Vinoski, 1999]**. A DSI é o equivalente servidor da DII e pode receber tanto chamadas estáticas quanto dinâmicas.

O *implementation repository* fornece um repositório dinâmico de informações sobre as classes que o servidor suporta, sobre os objetos que estão instanciados e os seus IDs. Ele também serve como um lugar para armazenar informações associadas com a implementação dos ORBs. A especificação CORBA não padroniza o *implementation repository* e somente sugere algumas funções que ele deve implementar **[OMG, 2002]**. Isto é proposital, uma vez que o repositório está intimamente relacionado à plataforma onde o objeto foi implementado, tendo o mesmo que lidar com a criação e finalização de processos, *threads* e outros aspectos ligados ao software básico. Também não é viável propor uma especificação que cubra todos os possíveis ambientes, porque a exata funcionalidade oferecida pelo *implementation repository* varia para diferentes ambientes **[Hennig & Vinoski, 1999].**

Além disso, características como migração, escalabilidade, performance e *load balancing* dependem do *implementation repository*, fornecendo, assim, o principal aspecto no qual os provedores de ORBs podem acrescentar características extras aos seus produtos além de adaptá-los aos ambientes servidores **[Pritchard, 1999]**.

Apesar da falta de padronização, a interoperabilidade entre ORB´s de diferentes fornecedores é assegurada, pois a especificação mostra precisamente como se processa a interação dos clientes com o *implementation repository*. Ou seja, a solução proprietária existe apenas entre o servidor e o repositório.

Descreveremos agora a arquitetura COM e, em seguida, faremos uma análise dos dois modelos de arquitetura distribuída.

2.2 COM, DCOM E ACTIVEX

Poucos assuntos, em todo o espectro da programação de computadores, são tão vastos e tão amplos como COM. É bem possível que COM seja o maior esforço de programação já empreendido para os computadores da plataforma Intel – IBM PC **[Reisdorph, 1999]**. Além disso, é o modelo mais usado no mundo para a construção de componentes de software **[Microsoft, 2002]**. Mas, apesar do tamanho e da sofisticação, essa arquitetura não é universalmente aplaudida, mesmo após a Microsoft ter melhorado a estrutura e a forma de implementar os níveis mais baixos de COM.

Quando começamos a estudar e desenvolver componentes COM, nós tínhamos um preconceito negativo muito forte. No momento atual, mesmo com as restrições e com as críticas que recebe, a nossa experiência pessoal, com este tipo de componente, é que esta arquitetura funciona. E funciona muito bem. Até melhor do que os seus críticos mais ferrenhos admitem.

COM é a abreviatura de *Component Object Model*. Apesar de não termos encontrado nenhuma definição formal, pudemos inferir que as principais conquistas de COM são:

1. Proporcionar uma forma de definir uma especificação para a criação de objetos independente de linguagem;
2. Proporcionar uma forma de permitir a implementação de objetos que podem ser chamados entre processos distintos, mesmo que esses processos estejam rodando em máquinas diferentes (essa característica de COM o fez ser chamado pela Microsoft, até há pouco tempo, de DCOM,

mas hoje é referenciado, por razões comercias, simplesmente como COM+);

Em outras palavras, COM permite ao programador (1) escrever um código que pode ser usado por múltiplas linguagens de programação, (2) criar controles ActiveX, que interagem com algum código que esteja executando dentro ou fora do mesmo processo que eles; (3) controlar outros programas através da automação OLE, além de (4) trocar mensagens com objetos ou programas em outras máquinas.

COM está por trás da maior parte do novo código desenvolvido para os sistemas operacionais Windows XP e Windows 2000 **[Microsoft, 2002]**. Na prática, pode-se pensar no Windows como um conjunto de interfaces COM, que uma aplicação pode chamar quando precisar de um serviço específico. O Windows Explorer, do Sistema Operacional Windows, por exemplo, é acessado através de COM. Se alguém deseja fazer mudanças na Barra de Tarefas (componente da GUI do Windows), o faz através de COM. A programação da interface do usuário do Windows, usa um conjunto de interfaces COM, chamado Shell API. COM permite que os softwares Word e Excel, por exemplo, possam ser completamente automatizados, fazendo com que, trabalhando com um deles, seja possível executar funções do outro, de maneira transparente para o usuário.

Além do Windows, o mesmo conceito se aplica também a outros softwares da Microsoft, como o Internet Information Server (IIS), Internet Explorer, Microsoft Transaction Server, Microsoft Message Queue etc.

A história do COM começa com a OLE (Object Linking and Embedding). Esta tecnologia apareceu, inicialmente, no Windows 3.1. Usando o OLE original

(1.0), podia-se criar um documento composto. Um documento composto contém dados de duas ou mais aplicações. Por exemplo, um documento criado com Word poderia incluir um conjunto de células de uma planilha de Excel, a fim combinar fontes e tipos de informação diferentes. Os dados de uma aplicação podem ser ligados ou podem ser embutidos dentro do outro. No exemplo anterior, se o dados são ligados, então mudanças na planilha são visíveis quando o documento do Word é aberto. Se os dados estiverem embutidos, mudanças para a planilha não são atualizadas no documento do Word, ou sejam não há nenhum link entre estes dois arquivos. No exemplo citado, dentro da tecnologia OLE 1.0, os dados ligados e embutidos podiam ser editados apenas por suas aplicações nativas: o Excel (aplicação servidora) era usado para editar a parte da tabela e o Word (aplicação cliente) era usado para processar o texto.

OLE evoluiu para OLE 2.0, que passou a permitir a edição *in-place*. Esta característica permite que o menu e as *toolbars* do programa servidor substituam, temporariamente, aqueles itens do programa cliente quando o usuário editar objetos ligados ou embutidos. Com a edição *in-place*, o usuário pode editar a parte dos dados da outra aplicação (servidora), sem deixar a aplicação do cliente.

Um outro aspecto do OLE é automação OLE. Usando a automação OLE, um programa cliente pode invocar um servidor e então usar a sua funcionalidade. O servidor pode rodar como um processo escondido do usuário, ou pode ser visível, para promover a interação do usuário. Com o Network OLE, o programa servidor pôde passar a residir em uma máquina diferente daquela do programa cliente.

A Microsoft, ultimamente, se refere a OLE 1.0, OLE 2.0, Automação OLE e Network OLE, simplesmente como OLE (OLE deixou de ser uma sigla e passou a ser um nome). Todas essas variações do OLE se baseiam na arquitetura conhecida como COM (ou COM+).

COM é uma especificação desenvolvida (e implementada primordialmente) pela Microsoft, que descreve como objetos devem se comunicar entre si. Componente é um termo genérico que se aplica a muitos tipos diferentes de software: programas cliente, programas da camada intermediária de uma aplicação n-camadas, programas servidores, objetos em geral e mesmo uma aplicação completa. COM descreve como Objetos Componentes devem se conectar, mas não descreve como o código que implementa a interface deve ser escrito. Esta flexibilidade permite que os programas sejam desenvolvidos usando linguagens diferentes, de diferentes fabricantes, tais como C++, Visual Basic, Delphi, ou o MicroFocus Cobol. Por exemplo, o software da camada cliente de uma aplicação pode ser escrito em VB, componentes da camada intermediária podem ser implementados em C++ e outros podem ser implementados em Delphi.

DCOM é o COM distribuído. Usando DCOM, componentes COM implementados em linguagens distintas, de forma independente, sem um ter prévio conhecimento do outro, podem ser colocados em máquinas diferentes e se comunicarem um com o outro sem nenhuma mudança ou adaptação nas suas implementações.

Usando RPC's, o DCOM, executando em uma máquina pode estabelecer uma conexão com o DCOM executando em uma outra máquina. Depois que os

objetos estão conectados, o DCOM age como um autêntico *middleware* entre as camadas de COM nas duas máquinas.

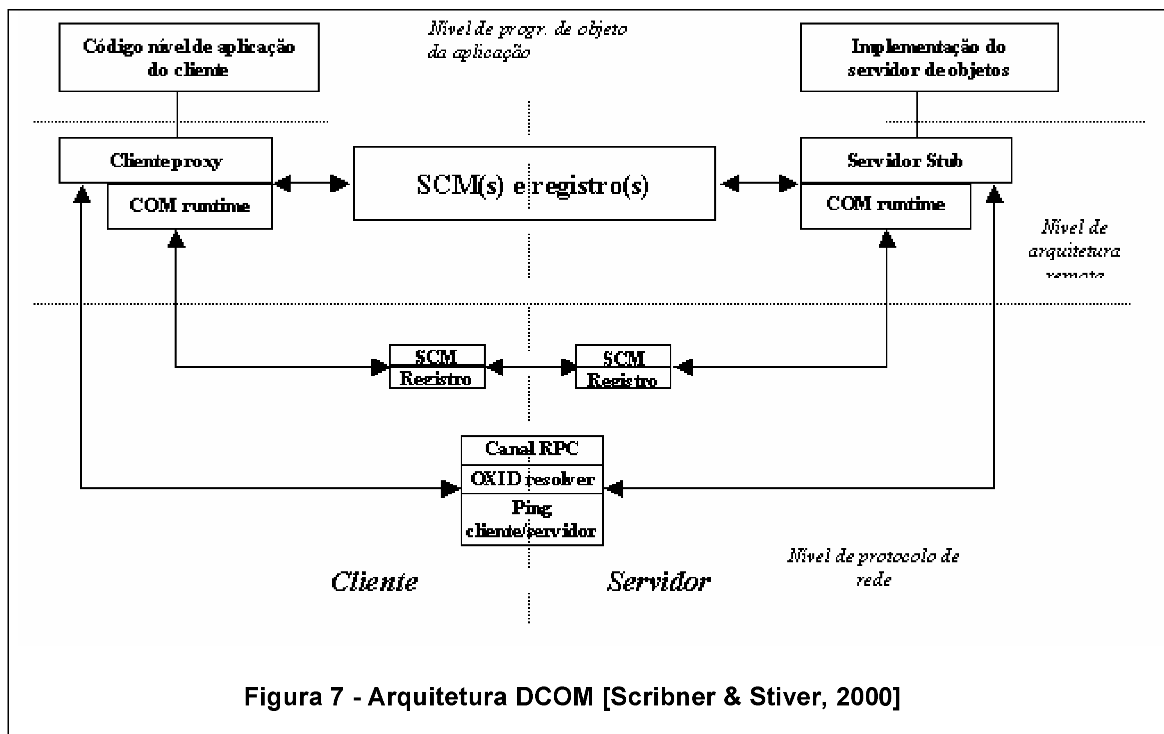

Figura 7 - Arquitetura DCOM [Scribner & Stiver, 2000]

Como pode ser observado na figura 7, um cliente, ao requisitar a criação de um objeto remoto, faz com que, no cliente, haja o recebimento de uma referência para o *proxy* local. O SCM (*Service Control Manager*) local pega as informações do sistema remoto, apresentadas pelo cliente quando o *proxy* foi criado (ou as lê do banco de dados de registro do sistema) e contata o SCM no computador remoto. O SCM ativa o objeto remoto e devolve uma referência desse objeto para o sistema local. Essa referência é dada para o *proxy* do objeto local que o cliente pode usar para criar instâncias do objeto remoto. As funções detalhadas dos

componentes presentes na figura acima e as particularidades das chamadas estão descritas mais adiante neste capítulo.

Os Sistemas Operacionais Windows, UNIX, Macintosh, e muitos outros sistemas que rodam em mainframe suportam RPC e potencialmente podem usar DCOM. Por exemplo, o True65, da Compaq, é um Sistema Operacional Unix que o suporta. Usando DCOM, portanto, os componentes podem residir não somente em máquinas diferentes, mas também em plataformas diferentes e ainda assim comunicarem-se entre si. Entretanto, um dos problemas de DCOM é ser centrado no Windows, havendo pouquíssimas implementações desse modelo de objetos em outros Sistemas Operacionais.

ActiveX é um termo que se aplica tanto aos controles quanto aos servidores (ou aos componentes do código). A forma mais difundida de uso desse tipo de controle, mas não a única, é a sua inclusão em páginas dinâmicas da web. Neste caso, se um controle não estiver instalado na máquina do usuário que solicitar uma página que faça uso desse controle, o *web browser* pode fazer o *download* e instalá-lo para o usuário. Os controles desenvolvidos usando a especificação COM completa são relativamente grandes e os *downloads* tornam-se demorados. Os controles de ActiveX, por outro lado, usam um subconjunto da especificação completa de COM, permitindo que o tamanho do controle seja menor e, conseqüentemente, o *download* seja mais rápido.

Com relação às teorias de projetos de sistema, que em geral visam melhorar o desenvolvimento e a manutenção de programas através da segmentação, COM oferece um modelo que permite aos programadores fazerem de seus programas, um conjunto de componentes (servidores COM). Esses

componentes podem, além de fornecer serviços, ser vinculados a um programa principal quando for necessário. Por exemplo, um depurador de um ambiente de programação, poderia ser uma aplicação isolada, armazenada em algum lugar no disco do sistema, sendo integrada ao IDE, quando necessário, via COM. É o caso dos depuradores dos ambientes de desenvolvimento da Borland. Ou ainda, um sistema financeiro poderia usar, através de COM, a calculadora do Windows (ou uma outra mais específica), quando uma conta tivesse que ser feita pelo usuário, a fim de que o mesmo pudesse que tomar uma decisão.

Resumindo as características gerais de COM, podemos classificar esse modelo de objetos, mostrando que ele suporta três tipos de servidores para implementar componentes (ver figura 8):

- Servidor *in-process*. Um servidor *in-process* é implementado como uma DLL (*Dynamic Linking Library*) que executa dentro do espaço do processo de uma aplicação. O overhead de performance para invocar um servidor *in-process* é pequeno, devido ao mesmo estar localizado no mesmo processo do cliente. Este tipo de servidor é comumente referenciado como um controle ou componente ActiveX.
- Servidor Local. Um servidor local executa em um processo em separado no mesmo computador. A comunicação entre uma aplicação e um servidor local é alcançada através do COM Runtime (explicado no texto a seguir – item 2.2.1), usando um protocolo de comunicação interprocesso de alta velocidade **[Microsoft, 2002]**. Por motivos óbvios, o overhead de

performance e recursos para invocar um servidor local é bem maior que no caso do servidor *in-process*.

- Servidor Remoto. Executa em um computador remoto. DCOM estende a estrutura COM, provendo uma infra-estrutura baseada em RPC que é usada para gerenciar a comunicação entre a aplicação e o servidor remoto. Também por razões claras, o overhead de *performance* de uma aplicação para invocar um servidor remoto é bem maior do que seria se a mesma aplicação invocasse um servidor local.

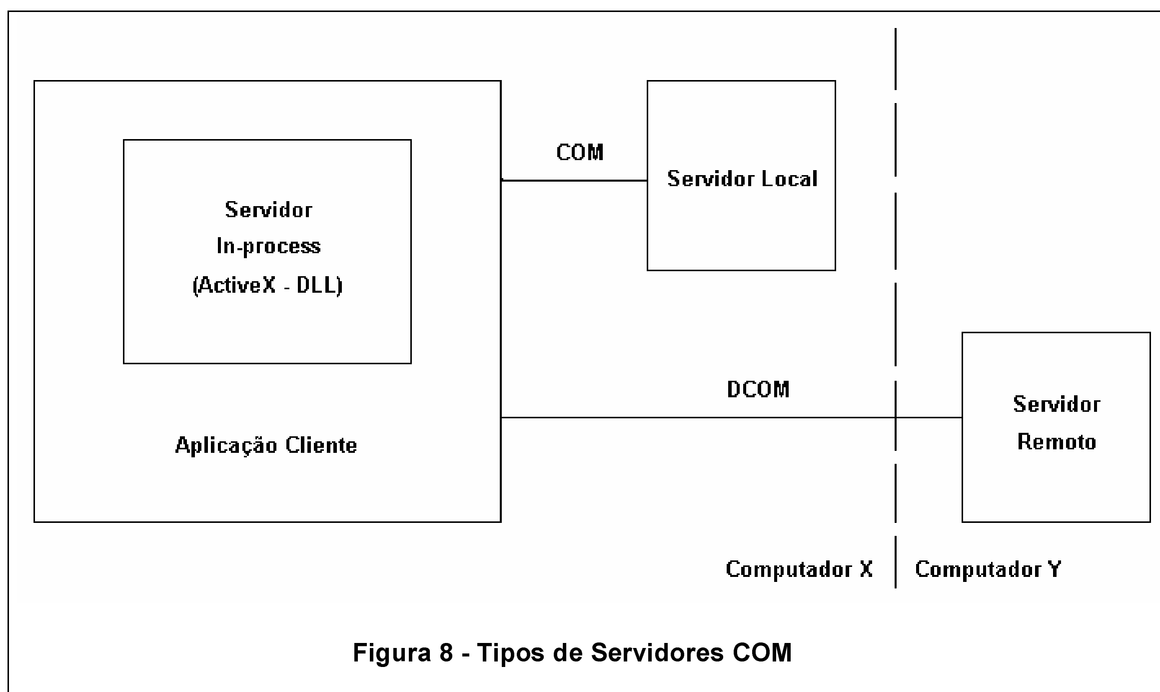

Figura 8 - Tipos de Servidores COM

2.2.1 A ARQUITETURA COM

O modelo COM, em essência, é constituído pelos seguintes elementos:

1. Object: este elemento se parece muito com uma classe C++. Os objetos implementam toda a funcionalidade e contêm a parte central do COM.
2. Interface: Este elemento fornece a definição das funcionalidades comuns a muitos objetos e o conjunto das interfaces de um componente COM define as regras para interação com os seus objetos.
3. *COM runtime*: Este elemento contém um pequeno conjunto de rotinas que mantêm os serviços do objeto.

Todo objeto implementa uma ou mais interfaces através das quais, cada objeto fica acessível para o mundo. As interfaces que os objetos implementam representam a suas funcionalidades.

Cada objeto pode optar por criar outro objeto como uma saída de uma chamada em uma das suas interfaces (a implementação de um método de interface cria um objeto e retorna um ponteiro para uma das suas interfaces através de um parâmetro de saída). Este processo é parte da semântica da interface. É raramente útil um objeto instanciar outro e então este segundo objeto produzir outro objeto instância da classe do primeiro objeto **[Liberty, 2000]**. Na maioria das vezes, existe um objeto criado diretamente que cria outros objetos, os quais criam outros objetos etc, resultando em uma estrutura de árvore de objetos. Nessa estrutura, os nós (objetos) de mais alto nível podem produzir objetos (dos nós) de mais baixo nível. No jargão de *COM*, dá-se o nome de componente a toda esta coleção hierárquica de objetos criados pelo objeto raiz e seus descendentes.

Um tipo particular de objetos COM que podem ser criados diretamente a partir do *COM Runtime* é a “COM class”, ou *coclass*. Estes objetos estão associados com outros tipos de objetos COM, as fábricas de classes (*class*

factories), que os instanciam. O *class factory* tem um método em uma das suas interfaces que produz uma instância de *coclass*. Este método com os seus parâmetros, portanto, pode ser considerado como uma construção, independente de linguagem, análoga ao operador C++/Java *new*, uma vez que provê alocação dinâmica de memória.

Objetos COM podem ser usados pelos seus clientes de dois modos distintos: direto ou indireto (*marshaled*). Quando um objeto e seus clientes residem no mesmo contexto de execução (chamado estranhamente de apartamento), os clientes usam ponteiros simples (diretos) para as interfaces dos objetos. Quando o objeto e seus clientes residem em diferentes apartamentos, o COM se interpõe entre eles e provê o suporte necessário para chamar os métodos da interface e retornar os resultados. Dois objetos, em especial, fornecem os meios necessários para isso. (1) O *proxy* reside no contexto do cliente e atua como o objeto para aquela interface em particular. Isto tudo de forma transparente, ou seja, o cliente não pode diferenciar entre o objeto e o seu *proxy*. (2) O *stub* reside no contexto do objeto e atua como cliente daquela interface de objeto. A conexão entre os objetos usa protocolos apropriados para aquela conexão (mensagens do windows, RPC ou outro qualquer).

2.2.2 INTERFACES C++ e INTERFACES COM IDL

Em COM, assim como em CORBA, são entidades conhecidas como interfaces que definem a funcionalidade que os objetos implementam. Elas se

relacionam muito proximamente a classes C++ que contenham somente métodos virtuais puros e nenhum item de dado.

Descreveremos, a seguir, como as interfaces COM são implementadas em C++, tendo em vista que, apesar de ser a linguagem que implementa o modelo de componentes COM, não existe em C++ uma estrutura que dê suporte a esse conceito, como existem em outras linguagens de programação, tais como Java, delphi e C#, ou como na IDL CORBA. Note-se claramente que essa restrição de C++ não representa um empecilho ao uso dessa linguagem, tendo em vista que, como já frisamos, as principais estruturas (ORB, COM Runtime) das arquiteturas COM e CORBA são implementadas em C++ **[OMG, 2002][Microsoft 2002]**.

Além disso, a maior parte das *bridges*, de componentes desenvolvidos especificamente para uma certa interação, são implementadas usando-se C++, conforme abordaremos no item 2.3.1.

A palavra-chave *interface*, usada no código para descrever uma interface, é substituída por um *struct* em uma diretiva de pré-processamento C++

```
#define interface struct
```

Todos os métodos de interface são declarados como sendo métodos virtuais puros em um *struct* C++. Como exemplo, mostraremos (inclusive ao longo da explicação que se segue), elementos do objeto cliente COM, Calculadora, que será usado em implementações que demonstraremos no capítulo cinco:

```
Interface ICalculadora
{
  void Somar(double Valor);
};
```

Esta definição é traduzida para isto, em COM:

```
Struct ICalculadora
{
  virtual void Somar(double Valor) = 0;
};
```

Porque um *struct*, em C++, torna todos os seus métodos e membros públicos, uma simples diretiva de pré-processamento é suficiente. Se uma classe for usada, será necessária a adição da palavra *public*, para definir este tipo de visibilidade. Se uma classe for usada, o uso da palavra chave *public* é obrigatório.

A semelhança entre interfaces e classes C++ é notadamente intencional e tem origem do fato de que todos os compiladores C++ geram a mesma tabela de métodos virtuais que, no caso das classes que possuem apenas métodos virtuais, é a definição binária exata de uma interface. Completando o quadro, todos os métodos devem efetuar uma chamada padrão (*stdcall*) para os subprogramas, a fim de alcançar compatibilidade binária com outras linguagens. Isto porque a linguagem C++ carrega os parâmetros de subprogramas na pilha de execução, da direita para a esquerda, enquanto linguagens como Delphi, Cobol e Fortran o fazem da esquerda para a direita.

Interfaces estão sujeitas às seguintes restrições:

- Elas não têm implementações, apenas métodos virtuais puros;
- Elas não podem conter dados;
- Elas podem ser derivadas de outras interfaces, mas as heranças podem apenas acrescentar novos métodos (virtuais puros);
- Somente herança simples é permitida;

A restrição sobre dados membros é crucial na arquitetura COM. Os dados não podem ser manipulados diretamente. Em vez disso, o cliente deve ajustar-se às interfaces expostas. Isto é necessário para preservar a independência de linguagem e de código entre o cliente e o objeto, além de prover a base para a independência das trocas de contexto de execução. A restrição sobre herança simples é necessária porque não existe um padrão para o layout binário das VTables (as tabelas de métodos virtuais usadas para chamadas com ligação dinamica em C++) das classes bases quando a herança múltipla está envolvida.

Embora a VTable seja uma excelente identidade binária para uma interface, deve existir uma forma de identificação no código fonte, ou seja, de alguma forma, um programa-fonte em C++ ou em outra linguagem qualquer, deve poder referenciar-se a ela em algum lugar do código. Uma alternativa seria usar o nome textual da interface usado na sua definição. Embora este enfoque seja ótimo para fontes C++, ele não se sustenta quando olhamos para os objetivos de COM: não está na forma binária, não pode ser compreendido por outras linguagens e não tem a garantia de ser único (dois programadores podem dar um mesmo nome às suas interfaces).

A solução vem na forma de um identificador binário denominado IID (*interface identifier*). IIDs são variações das GUID's ou UUID's (arrays de 16 bytes que são garantidos serem únicos pelo algoritmo usado na sua geração). *Globally Unique Identifier* (GUID) é o nome que a Microsoft deu aos UUIDs (*Universally Unique Identifiers*), entidades definidas pelo consórcio OSF DCE (Open Software Foundation Distributed Computing Environment). O IID é o GUID associado ao

nome de uma interface. A função COM *CoCreateGuid()* gera um novo GUID cada vez que é chamada.

O GUID é gerado combinando-se o número único encontrado na placa de rede com outra informação, como a data atual do sistema **[Reisdorph, 1999]**.

Pela especificação COM, as interfaces são descritas em uma linguagem própria chamada IDL (*Interface Definition Language*). A IDL é parte da OSF DCE *Remote Procedure Call* e é usada também para definir uma visão transparente das chamadas remotas e para gerar o código apropriado para elas.

A Microsoft acrescentou algumas extensões específicas para COM. O resultado é que interfaces COM podem ser usadas remotamente através de RPC (isto significa entre processos, da mesma forma que entre máquinas).

A sintaxe IDL é similar à sintaxe das declarações C++, com alguns *tokens* adicionais. O exemplo anterior da definição da interface ICalculadora escrita em IDL fica assim:

```
[object, uuid(55A2EEE1-E158-11D4-B28D-000021638ACB)]

interface ICalculadora

{

  void Somar([in] double Valor);

};
```

A palavra chave *object* indica que isto é uma interface COM, *uuid()* fornece o IID da interface, *in* mostra que o argumento é um parâmetro de entrada. Existem ferramentas que geram, a partir de um arquivo com a definição de uma interface IDL, um arquivo *header* C++ que deve ser usado quando se implementa ou usa-se a interface. Essas ferramentas também geram o código *proxy/stub* para acesso remoto e um arquivo que define o IID (GUIDs são representadas como estruturas e precisam ser alocadas). Mais de uma interface pode ser definida em um único arquivo IDL.

Da forma que a interface ICalculadora está definida até agora, ela não irá compilar corretamente. Dois erros serão apontados pelo compilador (MIDL): (1) todas as interfaces devem herdar de *IUnknown* e (2) todos os métodos das interfaces COM devem retornar HRESULT. Fazendo as alterações, temos:

```
import "unknown.h"

[object, uuid(55A2EEE1-E158-11D4-B28D-000021638ACB)]

interface ICalculadora : IUnknown
{
  HRESULT Somar([in] double Valor);
};
```

O arquivo header contém, após a compilação, além de outros debris, a seguinte definição de interface:

Interface

```
DECLSPEC_UUID("55A2EEE1-E158-11D4-B28D-000021638ACB")

interface ICalculadora : public IUnknown
{
public:
  virtual HRESULT STDMETHODCALLTYPE Somar( /*[in]*/ double Valor) = 0;
};
```

A classe C++ resultante, portanto, contém apenas métodos públicos virtuais puros.

HRESULT é um inteiro de 32 bits que descreve o status da chamada do método. O nome implica que o valor será tratado como uma referência (ponteiro), o que, apesar das implicações históricas apontarem neste sentido, já não é mais o caso, tendo em vista que 32 bits são suficientes para descrever o resultado em si **[Rector et al 1999]**. O padrão COM obriga todos os métodos a devolver HRESULTs para que os erros sejam reportados de maneira uniforme. Olhando os bits individualmente temos:

SRRFFFFFFFFFFFCCCCCCCCCCCCCCCC

S (1 bit) – Código de severidade: SEVERITY_SUCCESS ou SEVERITY_FAIL

R (2 bits) – Reservado, deve ser zero

F (11 bits) – *Facility*, diz qual subsistema reportou o erro

C (16 bits) – Código de status (aponta o erro)

O código de status em si ocupa os primeiros 16 bits de HRESULT. Depois vem a parte de facility, que aponta para o subsistema que retornou o erro. Todos os códigos principais são vinculados para FACILITY_ITF (4), que significa (ITF) específico de interface. O bit mais importante é o de severidade:

SEVERITY_SUCCESS (0) significa que a operação foi realizada, enquanto SEVERITY_FAIL (1) significa que a operação falhou. Há ainda duas macros FAILED(hr) e SUCCEEDED(hr) que testam o HRESULT hr retornado para sucesso ou falha, respectivamente.

Nomes simbólicos correspondentes a constantes HRESULTs contém três partes na seguinte ordem: facility, severidade e código. Algumas poucas constantes HRESULTs usadas omitem a parte facility, como, por exemplo: CO_E_NOTINITIALIZED, DRAGDROP_S_DROP, E_OUTOFMEMORY, E_FAIL, S_OK e S_FALSE.

HRESULTs não devem ser usadas para reportar exceções, já que não é permitido a elas ultrapassar os limites da interface (elas são específicas da implementação C++, ou têm implementação diferente, ou mesmo estão ausentes em outras linguagens). As exceções que ultrapassam os limites da interface forçam a RPC a retornar RPC_E_SERVERFAULT (0x80010105) se a interface é remota. Se o objeto está rodando, uma exceção, provavelmente, derrubará o cliente.

As interfaces podem ser herdadas, o que faz com que a interface derivada tenha todos os métodos da interface base e o objeto que implemente a interface derivada tenha que implementá-los também. E se duas ou mais interfaces são herdadas de uma mesma interface base e as interfaces são implementadas por um único objeto, os métodos da interface base podem, provavelmente, ter mais de uma implementação.

Em resumo, não é aconselhável (embora permitido) que se use herança de interface, com exceção de circunstâncias especiais. COM oferece uma melhor

maneira de se obter o polimorfismo (o método QueryInterface() da interface IUnknown). A herança de interface é necessária somente se uma interface não pode existir sem implementar métodos da sua interface base.

Um exemplo é IUnknown, que deve ser a base de todas as interfaces COM:

```
[
    local,
    object,
    uuid(00000000-0000-0000-C000-000000000046),
    pointer_default(unique)
]
interface Iunknown
{
    HRESULT QueryInterface(
        [in] REFIID riid,
        [out, iid_is(riid)] void **ppvObject );
    ULONG AddRef( void );
    ULONG Release( void );
};
```

A palavra chave *local*, significa que essa interface não é remota (ou seja, nenhum código de manobra deve ser gerado pelo compilador IDL). Isto soa estranho porque as interfaces COM são geralmente remotas (isto é, elas podem ser chamadas de fora dos limites do contexto de execução).

Na verdade IUnknow é remota (ou seja, ela pode receber uma chamada remota). O seu esquema para as chamadas remotas reside em outra interface,

IRemUnknown, usada internamente, a qual tem os mecanismos para permitir que otimizações de chamadas sejam executadas. A declaração *pointer_default (unique)* informa que ponteiros NULL podem ser retornados nas chamadas de métodos.

iid_is é usado para tratar o ponteiro como um ponteiro de interface e atribuir a ele um IID. É aqui que os IIDs entram em cena pela primeira vez no código da interface. *REFIID* é um IID e é o único *token* não compatível com o C usado em métodos de interface. Ele explora o fato de que os compiladores C++ passam argumentos de referência como ponteiros.

QueryInterface() é usado para perguntar ao objeto, através do parâmetro *riid*, se ele suporta uma interface em particular, especificada pelo IID. O resultado é retornado no segundo parâmetro, um ponteiro genérico tipo void **, o que retira a possibilidade de verificações de tipo. *QueryInterface()* deve retornar S_OK ou E_NOINTERFACE, dependo se a interface pesquisada é ou não implementada pelo objeto, respectivamente. Em caso negativo, o parâmetro ppv deve ser atribuído a NULL.

AddRef() e *Release()* são usados para controlar o tempo de vida do ponteiro da interface, por meio de contagem. Todo ponteiro de interface tem um contador de referências. Sempre que ele for obtido, deve-se seguir uma chamada a *AddRef()*.

Quando o cliente finaliza seu processamento com o objeto, deve usar o ponteiro para chamar *Release()*. Toda vez que um ponteiro for duplicado (como, por exemplo, quando for atribuído a outra variável ponteiro), *Addref()* deve ser chamado para incrementar o contador de referências. Antes de destruir a cópia,

como no final de um escopo, *Release()* deve ser chamado. A implementação de *Release()* deve verificar quando o contador de referências alcançar o zero para liberar quaisquer recursos alocados pela interface.

Todo objeto gerencia seu próprio contador de referência. A implementação real é livre para usar um contador de referência único para todas as interfaces. Apesar disso, quando todas as interfaces de um objeto alcançam zero referência, o objeto destrói a si mesmo.

Porque todas as interfaces derivam de IUnknown, elas possuem *QueryInterface()* nas suas VTBLs e os objetos, geralmente, possuem apenas uma implementação de *QueryInterface()* na VTBL.

Há quatro maneiras diferentes de obter-se um ponteiro inicial para um objeto:

- Através das funções genéricas de criação de objetos COM, como CoGetClassObject() ou CoCreateInstance();
- Através de uma chamada a um método de uma interface em outro objeto que retorna a interface em outro objeto;
- Quando um cliente de um objeto passa um ponteiro de interface de outro objeto para ser o primeiro objeto a fazer uma chamada a um método da interface. Então, o primeiro objeto recebe um ponteiro do objeto novo para uso interno (o primeiro objeto é o cliente);
- Através de outra função API que produz um objeto específico e retorna uma das suas interfaces para o chamador. Um exemplo é a função CreateStreamOnHGlobal() que cria o objeto OLE padrão que

representa um stream (bloco) de memória e retorna a sua interface Istream.

O último método raramente é usado **[Rector et al 1999]**. Era padrão nos anos iniciais da OLE (que tinha o background Win32 API). O segundo método representa a navegação da hierarquia de objetos. O terceiro método apresenta os objetos com uma forma para se comunicar pró-ativamente com os seus clientes. O primeiro método é a mais amplamente forma aceita de distribuir componentes COM. Ela funciona nas *coclasses*. Cada *coclass* tem um objeto associado chamado objeto classe ou fábrica de classe. Embora não seja obrigatório, este objeto expõe a interface IClassFactory:

```
[
    local,
    object,
    uuid(00000001-0000-0000-C000-000000000046),
    pointer_default(unique)
]
interface IclassFactory : Iunknown
{
    HRESULT CreateInstance(
        [in, unique] Iunknown *pUnkOuter,
        [in] REFIID riid,
        [out, iid_is(riid)] void **ppvObject );
    HRESULT LockServer ( [in] BOOL block );
};
```

O método LockServer() é usado para impedir o servidor que implementa os objetos de interesse de finalizar quando nenhum objeto estiver existindo. Então os pedidos de criação submetidos posteriormente são atendidos consideravelmente mais rápidos. Todas as chamadas para *LockServer(TRUE)* devem ser associadas com um número apropriado de chamadas a *LockServer(FALSE)* (da mesma forma que *AddRef()* e *Release()* para as interfaces).

CreateInstance() recebe três parâmetros, sendo os dois últimos semelhantes àqueles passados para o método *QueryInterface()* e permitem ao cliente obter diretamente um ponteiro para uma interface específica, não somente *IUnknown*. O primeiro parâmetro é o novo ponteiro de objeto IUnknown e é usado para agregação. Para clientes normais ele será sempre NULL. Este método tem um conhecimento íntimo do objeto que ele deve criar. O ponteiro retornado deve ser controlado com um contador de referência (*AddRef()* deve ser chamado).

O *COM runtime* tem mecanismos para localizar as fábricas de classes (que não discutirei neste texto).

2.3 ANÁLISE DE COM E CORBA

CORBA fornece ótimo suporte (inter-plataforma, inter-linguagem) para criação de objetos servidores de uma aplicação de arquitetura remota N-camadas, pois se pode criar esses servidores usando linguagens como C, C++, Java, Smalltalk, COBOL, ADA, Delphi, Phyton, C# **[OMG, 2002]**, além de existirem produtos CORBA em diversas plataformas: *mainframes*, na maioria das plataformas Unix, nas versões do Windows, para OS/2 e também para plataformas Vax/VMS **[Dr. Dobbs, 2001]**.

Além disso, a especificação CORBA é um padrão aberto que foi criado e aceito por um grande número de instituições que formam o OMG e a arquitetura CORBA está bastante amadurecida, após quase uma década (desde a primeira implementação, em 1993) de aperfeiçoamentos. Por tudo o que foi descrito, nós podemos afirmar que CORBA é a arquitetura dominante de objetos distribuídos e está bem à frente das principais concorrentes: COM (Microsoft), SOM (IBM), Voyager (ObjectSpaces) e RMI (Sun).

Apesar disso, encontramos algumas fraquezas. CORBA é um padrão aberto e consensual entre centenas de instituições, o que dificulta e torna demorada a aprovação de novas versões da especificação **[OMG, 2002]**. Some-se a isso, o fato de que, apesar da especificação CORBA 3.0 conter um modelo para componentes CORBA, ainda não apareceu nenhuma implementação comercial do mesmo. Talvez por causa do sucesso do modelo de componentes COM. Registre-se que até alguns componentes CORBA presentes em vários ambientes de

implementação usados por nós durante esta pesquisa (como o C++ Builder 6, da Borland, por exemplo), são componentes COM.

COM também é uma arquitetura de componentes bastante amadurecida e muito usada. Só para se ter uma idéia, na mesma ferramenta de desenvolvimento que mencionamos anteriormente, o C++ Builder 6, contamos 359 componentes COM que vêm com o produto. Já o Visual C++ .NET, da Microsoft, possui 621 objetos COM. Isso sem mencionar os componentes implementados por terceiros, para cada uma dessas ferramentas. Ou seja, fazendo uma projeção a partir de buscas pela internet, estamos falando de dezenas de milhares de componentes COM já construídos em dez anos.

Essa quantidade de componentes deu-se por alguns importantes fatores, já comentados, presentes na história desses componentes. Ou seja, por suas múltiplas características (*in-process*, *out-of-process* e remoto), pela sua fácil integração a ferramentas e aplicações (via Windows), bem como pelas facilidades de criação desses componentes, nos diversos ambientes de desenvolvimento da plataforma Windows.

O próprio Windows fornece todos os seus serviços através de objetos COM, dos mais simples (objetos de interface) aos mais avançados (como o Microsoft Transaction Service, Microsoft Message Queueing Service e o .NET framework **[Microsoft, 2002]**). Esse conjunto formado pelas ferramentas de desenvolvimento, serviços do Windows e o modelo de componentes COM tornam bastante ágil a construção e manutenção de sistemas multicamadas complexos.

As principais fraquezas de COM incluem as limitações de plataforma, com pouquíssimas implementações para outros sistemas operacionais, como o Unix (e

também para mainframes **[Microsoft 2002]**) e o seu uso em linguagens que não aquelas suportadas pela Microsoft (como o Java). Além disso, existe o fator de risco de depender de um único e principal fornecedor da tecnologia (e responsável pela sua especificação e desenvolvimento) na plataforma Windows, que é a Microsoft. Na plataforma Unix, apenas Software AG e Compaq criaram modelos de objetos COM em suas implementações.

Baseando-se nas características de COM e CORBA, podemos afirmar que CORBA parece ser uma escolha adequada para se criar servidores para todas as camadas (exceto a camada de interface) de uma aplicação multicamadas. Já COM pode ser apontado como a escolha ideal para a camada cliente (camada de interface). Isso faz sentido, porque as camadas intermediárias e a camada servidora de uma aplicação multicamadas precisam ser construídas usando-se a melhor arquitetura distribuída disponível e não necessariamente precisam se basear em uma arquitetura de componentes, aspecto no qual COM é muito superior no momento.

De acordo com as atuais especificações de cada arquitetura, COM e CORBA têm que lidar com um conjunto semelhante de funcionalidades e características, a fim de que possam desempenhar seus papeis. Analisemos:

- Em um sistema distribuído, interfaces são usadas para estabelecer contratos que descrevem as funcionalidades de cada objeto remoto. Então, ambos possuem uma linguagem para definir interfaces;
- Ambos suportam inúmeros tipos de dados primitivos e estruturas de dados, com o objetivo de transmitir informações;

- Ambos procuram garantir a integridade dos dados, enquanto os mesmos são transmitidos. Eles conseguem isso através dos mecanismos de *marshaling* e *unmarshaling*;
- Como um sistema distribuído precisa tornar transparente o acesso aos objetos remotos, não importando o local em que esses estejam, tanto COM quanto CORBA se utilizam dos mesmos mecanismos, mas com nomes diferentes: *proxies*, *stubs* e *skeletons*;
- *Handles* são usados por ambos para que, através deles, um objeto possa acessar outro objeto remoto. Em COM, os *handles* são chamados de ponteiros de interface e em CORBA de referências de objetos;
- A criação de novas instâncias de objetos distribuídos também é proporcionada. COM tem duas *factories* destinadas para tal, *IClassFactory* e *IClassFactory2*. Em CORBA, é o programador quem define as interfaces das *factories*;
- Tanto COM como CORBA definem os mecanismos para chamadas estáticas e dinâmicas de objetos distribuídos;
- Ambos também fornecem mecanismos para remover instâncias que não estão sendo mais usadas;

Sumarizamos na tabela abaixo as funcionalidades e características comuns de sistemas distribuídos, mostrando como COM (obviamente a parte DCOM) e CORBA se comportam em relação a elas.

Aspectos	Similaridades	Diferenças
Interfaces	DCOM e CORBA usam uma linguagem própria.	CORBA IDL é mais simples. COM tem melhor suporte do SO
Tipos de Dados	DCOM e CORBA suportam um rico conjunto de tipos primitivos de dados, além de constantes, tipos enumerados, estruturas e arrays.	DCOM identifica um subconjunto de tipos de dados conhecidos como Tipos de Automação. As interfaces DCOM não compatíveis com esse tipo não possuem a garantia de funcionarem em outros ambientes que não o C++. Qualquer interface CORBA pode ser usada de qualquer cliente CORBA.
Marshaling e Unmarshaling	DCOM e CORBA o fazem através de *stubs* cliente e servidor, de forma transparente.	DCOM permite que interfaces compatíveis com o tipo automação usem a biblioteca de tipos para as operações, eliminando a necessidade de *stubs* padronizados.
Proxies, stubs skeletons e Receivers	DCOM e CORBA baseiam-se em stubs cliente e servidor para gerenciar chamadas remotas. DCOM e CORBA geram stubs clientes e servidores a partir das interfaces.	DCOM – (proxies – stubs). CORBA – (stubs – skeletons). As DLL's que contém DCOM proxy-stub são usadas por todos os ambientes de linguagens. Um CORBA stub-skeleton em separado deve ser criado para cada combinação ORB/Linguagem (até a versão 2.1).
Object Handles	DCOM e CORBA suportam contadores de referência nas instâncias.	DCOM os chama de ponteiros de interface. CORBA os chama de referências. CORBA suporta herança múltipla de interfaces, DCOM não. DCOM suporta mais de uma implementação de interfaces, CORBA não.
Criação de objetos	DCOM e CORBA usam factories para criar instâncias de objetos.	DCOM tem interfaces padronizadas (IClassFactory). CORBA usa objetos específicos, sob medida para cada caso.

Invocação de objetos	Métodos similares às chamadas locais.	Em CORBA, os erros são suportados por exceções definidas pelo usuário. DCOM usa um mecanismo baseado nos valores de retorno de HRESULT.
Destruição do objeto	DCOM e CORBA baseiam-se em contagem de referências para determinar quando um objeto pode ser destruído.	CORBA mantém contadores de referências no cliente e no servidor. DCOM suporta garbage collection.

A seguir analisaremos formas já existentes de fazer interagir componentes dessas duas plataformas.

2.4 BRIDGES COM USO DE COMPONENTES ESPECÍFICOS

Há vantagens e desvantagens no uso de componentes criados manualmente para fazer a ponte entre DCOM e CORBA, que são dependentes do número de componentes que devem ser criados. Uma vantagem clara é a independência de produtos de terceiros. A desvantagem principal é o custo potencialmente elevado (tempo) associado com projeto, implementação e manutenção de tais componentes.

Ao criarmos um componente *bridge*, deve-se usar um ambiente de desenvolvimento que suporte ambas as tecnologias. Ao estudar as bridges baseadas em componentes desenvolvidos, examinamos três desses ambientes: o C++ que fornece excelente suporte a COM e CORBA; a JVM da Microsoft que fornece suporte embutido para COM e uma implementação de pacotes Java que dão suporte a COM em JVM's não Microsoft. Como, desde que iniciamos esse

estudo, o produto Microsoft foi descontinuado (Visual J++) e não representa mais uma solução viável e atualizada de implementação de *bridges*, não mostraremos detalhes dessa implementação.

2.4.1 O USO DE C++ PARA INTEROPERAR COM E CORBA

A linguagem C++ apresenta o ambiente mais maduro e o melhor instrumental para a construção de um componente *bridge* COM/CORBA. Isso porque, conforme descrevemos, a arquitetura interna de COM é obviamente norteada por C++ e CORBA também tem a sua implementação baseada nas estruturas dessa linguagem desde a sua criação (atualmente, pelo que pesquisamos, apenas Iona e Borland fornecem ORBs implementados em Java), o que proporciona um ambiente robusto para construir-se servidores e clientes em ambas as plataformas.

A principal desvantagem em usar C++ para implementar a *bridge* é que a linguagem, além de difícil de ser usada, não é um bom exemplo de simplicidade, como, por exemplo, quando se quer gerenciar com sucesso o *lifetime* dos objetos (comparando-se com Java, que tem gerência automática de memória). Essa dificuldade pode levar, em casos especiais não bem planejados, a vazamento de memória, bem como a presença de ponteiros oscilantes.

O enfoque mais direto para construir *bridges* COM/CORBA é escrever componentes especializados, que usam ambos, internamente, e mapear um modelo em outro.

Em 1995, logo após a aprovação da especificação CORBA 2.0, o OMG lançou a idéia da criação de um mecanismo capaz de permitir a comunicação entre COM e CORBA, essa idéia recebeu o nome de COM/CORBA RFP. Poucos meses depois, treze companhias interessadas na idéia, propuseram algumas intenções. O OMG, então, pressionou essas empresas a adotarem o IIOP como protocolo padrão de comunicação entre COM e CORBA **[Orfali & Harkey, 1998]**. A partir daí, foi concebido um *gateway* que provê mecanismos capazes de interagir as duas tecnologias. Esses mecanismos estão descritos na própria especificação CORBA, capítulos 17 a 20 **[OMG, 2002]**.

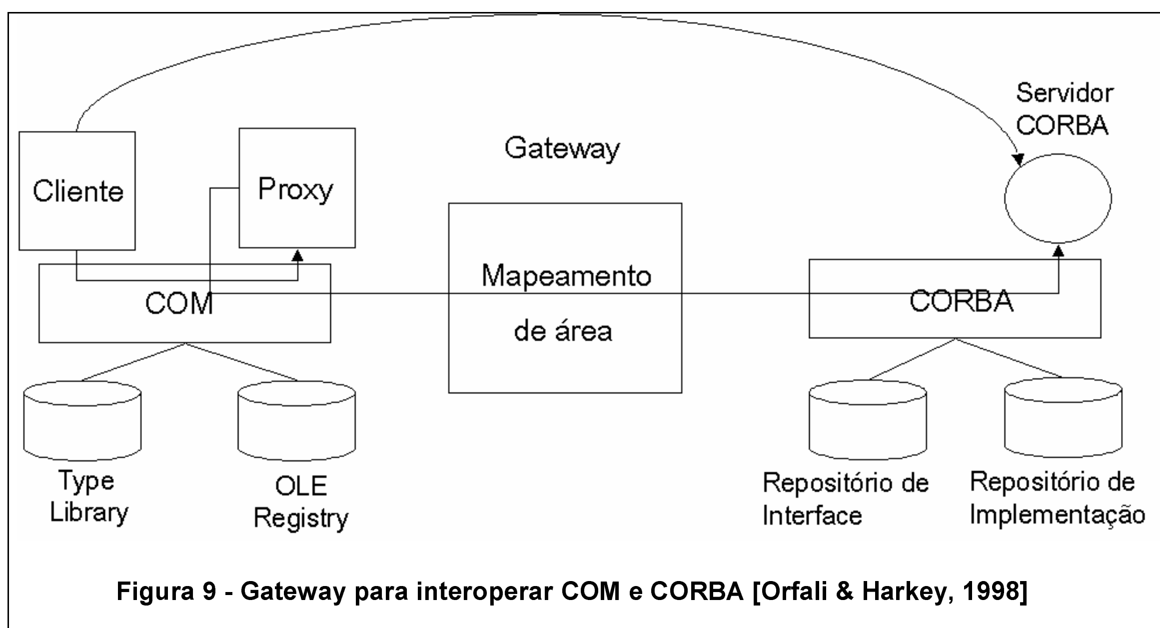

Figura 9 - Gateway para interoperar COM e CORBA [Orfali & Harkey, 1998]

Em uma chamada remota de um objeto COM para um objeto CORBA, como está ilustrado acima, o gateway deve fornecer ao proxy COM a referência do objeto remoto, para que o cliente COM possa invocar seus serviços. Além disso, esse *gateway* deve possuir várias outras funcionalidades, como:

- Mapas de tipos de dados: permite a conversão de dados de uma das tecnologias para dados equivalentes à outra tecnologia;

- Mapa de inicialização: provê uma relação que permita que clientes de um padrão acesse o primeiro serviço de uso do outro padrão;

- Mapa de fabricação de objetos: permite que um objeto de um dos padrões crie um objeto do outro padrão;

- Mapa de registros: permite que objetos de uma das tecnologias sejam registrados no registro de objetos da outra tecnologia;

- Mapa de domínios seguros: possui um mapa entre a segurança de uma tecnologia e de outra.

Com a disponibilização das implementações, por parte do *gateway*, de todos esses mapas, se faz possível a interligação entre COM/CORBA.

2.4.2 O USO DA JVM MICROSOFT COMO UMA BRIDGE

O enfoque mais direto para criar um componente *bridge* baseado em Java é usar a JVM da Microsoft, pelo suporte embutido que a mesma apresenta para COM e porque pode também ser usado com a maioria dos ORB´s Java. Atualmente, existe suporte CORBA para todas as versões da JVM Microsoft, que

segue o padrão das classes Java IDL para a versão 1.1.8 do jdk. O grande problema é a descontinuidade do produto que não mais foi atualizado.

Ao iniciarmos esse estudo de bridges, implementamos a primeira bridge usando este enfoque. Com o passar do tempo, e a desatualização da JVM Microsoft, abandonamos esta via. Um enfoque centrado nesta tecnologia pode ser encontrado em **[PRITCHARD, 1999]**.

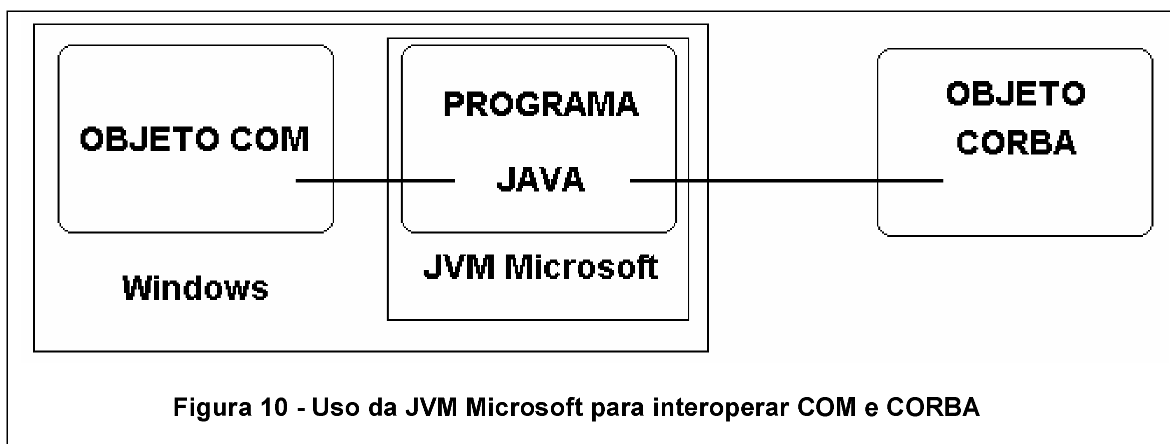

Figura 10 - Uso da JVM Microsoft para interoperar COM e CORBA

2.4.3 O USO DE COM EM JVM'S NÃO MICROSOFT

Para se poder usar COM em JVM não Microsoft deve-se recorrer a pacotes de terceiros, como o *J-Integra* da Intrinsyc (ver **[www2]**), implementado originalmente pela Linar, atualmente na versão 1.5 ou então o Java2COM, da Neva Object Technology (ver **[www3]**). Muitas empresas já entraram neste

mercado e saíram (ou faliram), como a Siemens Nixdorf, a Sapiens, a ExpertSoft, dentre outras.

Nos nossos ensaios e testes construímos uma *bridge* usando o J-Integra. Optamos por ele por dois motivos. Primeiro, o produto tinha um período de avaliação de dois meses (o dobro do Java2COM) e segundo, possui suporte técnico on-line muito melhor que descreve muito bem suas características (embora não seja tão bom assim para os procedimentos de instalação e configuração).

2.4.4 ACTIVEX E JAVABEANS

Muitos produtos que servem de ponte entre componentes Activex e Javabeans podem ser usados também para interoperar COM e CORBA. Os seguintes produtos são complementares, no sentido de que o primeiro pode ser usado para ligar clientes COM com servidores CORBA, enquanto o outro pode ser usado para clientes CORBA e servidores COM (*bridges* unidirecionais).

1 - A Sun desenvolveu um produto chamado Javabeans Bridge for Activex (ver **[www4]**), que permite que javabeans sejam usados como componentes ActiveX de aplicações clientes COM. Para interoperar de COM para CORBA, a funcionalidade de um servidor CORBA deve ser colocada no Javabean e o mesmo pode ser exposto como um controle ActiveX para um cliente COM.

Esse componente possui um utilitário que cria e empacota no Bean escolhido, informações sobre a Type Library OLE e sobre o registro do windows. Isto permite que os containeres OLE/COM analisem corretamente o Bean e o apresentem em um ambiente VB, por exemplo, como seria feito com qualquer componente da MFC. Esta interação permite ainda ao Bean disparar eventos que podem ser reconhecidos por uma rotina VB, além de poder ser usado como servidor, porque reconhece o modelo de invocação de métodos ActiveX/COM/OLE.

2 – A Gensym Corporation criou um produto chamado BeanXPorter, atualmente parte de um produto (solução *Enterprise*) chamado G2, o qual permite que componentes ActiveX sejam usados como Javabeans a partir de aplicações clientes Java.

Outra ferramenta nessa linha que permite a componentes ActiveX serem usados como servidores Java/CORBA é a Bridge2Java, da IBM, que faz uso da Java Native Interface (JNI) para criar proxies Java, a partir da Type Library de controles ActiveX servidores.

Esta abordagem não foi explorada neste trabalho que priorizou as bridges COM/CORBA usadas para interoperar objetos via WEB.

2.5 USO DE PRODUTOS COMERCIAIS

Muitos fabricantes de software produziram bridges que estão de conformidade com a especificação de interconexão da OMG (ver **[www5]** - capítulos 17, 18 e 19 da especificação CORBA 2.6). A maioria deles são também

implementadores de ORB´s e essas bridges são, geralmente, construídas sob medida para trabalharem com seus respectivos ORB´s. A exceção à regra é a Visual Edge Software (ver **[www6]**).

A *bridge* da Visual Edge, ObjectBridge, não trabalha com um ORB especificamente, mas, ao contrário, pode ser usada com uma gama de middlewares para CORBA, COM e Java, porque opera tanto com ORBs primários, como suporta um IIOP (Internet Inter-Orb Protocol) genérico. Isso garante seu uso com qualquer ORB CORBA padrão. Algumas empresas bem conhecidas licenciaram o produto da Visual Edge, como a BEA, Visigenic e Expersoft.

Outras empresas também criaram suas próprias *bridges* como a notável Iona (OrbixCOMet, **[www7]**), que está presente no mercado de objetos distribuídos desde os primeiros dias (e que também criou duas implementações de SOAP, ver apêndice B). Apesar de não oferecer nenhuma *bridge* COM/CORBA a Microsoft demonstrou interesse nessa área e licenciou o COM para a Iona e a Visual Edge.

Um produto especialmente focado em mapear COM/OLE Automation e CORBA é a *bridge* DAIS COM2CORBA, da PeerLogic (adquirida pela Critical Path – **[www8]**).

2.6 ENTERPRISE APPLICATION SERVERS

Como mencionamos no primeiro capítulo, há alguns anos, ao criar-se a camada intermediária em um sistema com arquitetura de duas camadas, retirou-se a lógica do negócio da parte cliente do sistema, para fixá-la em servidores

intermediários, nos quais a lógica da aplicação pode ser mais bem gerenciada. Ao longo desse tempo, a disseminação de aplicações baseadas em linguagens e plataformas distintas em uma mesma organização, isolou software e dados em ilhas de domínios de problemas e elevou os custos e os riscos para a integração desses sistemas. Isto levou a um novo desafio para as equipes de desenvolvimento, que tiveram que passar a se preocupar com questões de infra-estrutura (chamadas remotas, suporte, transações, segurança, *load balancing*, escalabilidade etc) e em como resolver problemas de integração entre aplicações, que muitas vezes foram projetadas para rodar em plataformas distintas e não possuem nenhum protocolo de comunicação com outros softwares que não fazem parte de seu domínio de aplicação.

O objetivo principal dos servidores de aplicação *enterprise* que apareceram em meados de 1998, é justamente livrar o programador dos tipos de preocupações citados acima, de maneira que ele pode centrar seus esforços apenas no objetivo do sistema, deixando que a infra-estrutura da arquitetura de integração forneça os serviços necessários. Mas isso não quer dizer que não seja penoso criar esse tipo de sistema, pois isto ainda envolve o gerenciamento de muitas complexidades.

Como solução para se estabelecer uma arquitetura *enterprise,* que possibilitasse gerenciar adequadamente as complexidades desses tipos de aplicação, algumas propostas surgiram. Dentre elas, as soluções que mais se destacam são as plataformas J2EE (Sun) e *dot NET* (mais nova versão do Microsoft DNA). Ambas têm vantagens e desvantagens, uma em relação à outra. Optamos por usar J2EE nas implementações que descreveremos nesta

dissertação, devido à possibilidade de integração mais facilitada das aplicações Java que rodam em cima de J2EE com servidores CORBA, através de RMI-IIOP. A plataforma *dot NET*, recém-lançada, ainda não possui a devida maturidade e os serviços próprios de comunicação remota (*NET Remoting*) ou de integração via *Web Services* ainda são limitados a softwares da mesma plataforma.

2.6.1 J2EE

J2EE é um conjunto de especificações de uma plataforma tecnológica, não é, portanto, um produto (não se faz um *download* de J2EE, mas de algumas ferramentas e servidores que fornecem serviços às aplicações). Este conjunto de especificações foi projetado para simplificar problemas com o desenvolvimento, distribuição e gerência de soluções *enterprise* multicamadas, baseadas ou não na web. Essas soluções são aplicações distribuídas que, portanto, requerem um ambiente que seja seguro, escalável e baseado em transações. Em um ambiente J2EE, as camadas podem estar fisicamente colocadas em um mesmo local ou podem estar separadas. Como já mencionamos, a camada intermediária em si não está restrita a uma única camada, mas pode ser dividida em subcamadas e as mesmas podem ser distribuídas entre nós distintos.

No estabelecimento desse padrão, a Sun trabalhou em colaboração com outras grandes empresas que estão no mercado de *e-business*, tais como a BEA, IBM e Oracle **[Vawter, 2001]**, em uma estratégia bem sucedida de garantir

mercado para o padrão, através do envolvimento de parceiros na definição do mesmo.

A plataforma J2EE é baseada na linguagem de programação Java, o que é uma boa estratégia, dado às características de independência de plataforma da linguagem. Ela consiste de muitas tecnologias diversas, algumas operando ao nível de sistema (serviços de infra-estrutura, como comunicação, segurança etc), outras operando ao nível de aplicação. Dentre essas últimas tecnologias estão a arquitetura EJB (*Enterprise Java Beans*), JSP (*Java Server Pages*) e Servlets. A arquitetura EJB define um modelo de componentes para aplicações *enterprise* e oferece mecanismo para estabelecer portabilidade entre servidores de aplicação. Servlets e JSP são ferramentas usadas na gerência de sessões e geração de conteúdo dinâmico para a web.

Como tecnologias e API´s da arquitetura J2EE que operam ao nível de sistema estão JDBC (*Java Database Connectivity*), JMS (*Java Message Service*), JTS (*Java Transaction Service*) e JTA (*Java Transaction API*), dentre outras. Cada uma apresenta interfaces de programação de tal forma que aplicações Java possam ter acesso aos serviços da plataforma.

Conceitualmente, a arquitetura J2EE divide-se em *containers*. Um *container* é um ambiente de execução padronizado que fornece serviços específicos aos componentes. Um componente é uma porção de software, um módulo, que é suportado por algum *container*, ou seja, cada container disponibiliza vários serviços especificados de sua função.

No ambiente J2EE há dois tipos principais de *containers*, o Web Container (que oferece serviços para aplicações dinâmicas da web – protocolos e API´s de

comunicação e serviços de rede) e o EJB Container (que oferece serviços como controle de transação, segurança, multi-threading, computação distribuída etc).

Assim, uma aplicação J2EE é um conjunto de *containers* de aplicações Java, na qual seus componentes são transformados em *bytecodes* e interpretados por uma JVM, independente de plataforma. Até os próprios *containers* são escritos em Java.

Há menos de dois anos, a arquitetura J2EE, tradicionalmente usada para construir aplicações *server-side*, foi estendida para incluir suporte para a construção de *Web Services* baseados em XML. Esses serviços podem interoperar com outros, embora demande mais esforço fazê-lo se a comunicação desejada for para serviços escritos em outras plataformas que não a J2EE.

2.7 A ARQUITETURA DE INTEGRAÇÃO J2EE

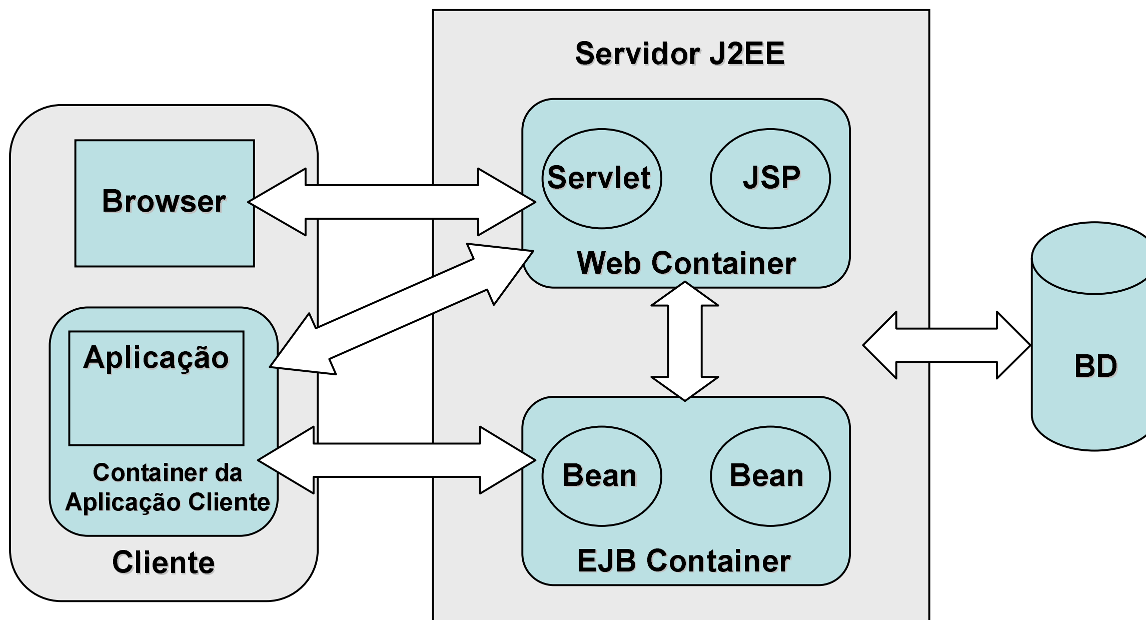

Figura 11 - Arquitetura J2EE

Como se observa na figura 11, acima, a aplicação J2EE é suportada por dois *containers* que provêem os serviços necessários para aplicações *enterprise*, como transações e segurança, dentre outros.

A camada de negócio, em aplicações J2EE corporativas, tem sua lógica embutida em componentes EJB´s (Enterprise Java Beans). Esta camada conecta-se a bancos de dados usando JDBC, SQL/J ou a sistemas existentes usando o JCA (*Java Connector Architecture*). Pode também se conectar com outros parceiros de negócio usando tecnologias de Web Services, como XML-RPC, SOAP, WSDL, UDDI, ebXML, através das API´s Java para XML (JAX´s), usadas por um servlet.

Alguns clientes não precisam de Web Services, já que normalmente conectam diretamente a camada EJB via IIOP, pois via de regra são escritos pela mesma organização que construiu a aplicação J2EE. Web browsers e alguns dispositivos *wireless* podem conectar-se diretamente a JSP´s/Servlets, os quais devolvem, como resposta, interfaces baseadas em HTML, XHTML e WML.

Como já frisamos, a plataforma J2EE especifica tecnologias que suportam o desenvolvimento de aplicações distribuídas e multicamadas. Essas tecnologias estão divididas em três categorias: componentes, serviços e comunicação.

Um componente é uma unidade de software (um tipo abstrato de dado – um objeto) que pode ser usado por aplicações, para lhes fornecer um certo serviço. Componentes usados em um ambiente J2EE incluem componentes EJB, componentes web (servlets e Java Server Pages) e aplicações clientes (incluindo applets). Estes últimos componentes rodam na plataforma cliente, e os outros, EJBs, Servlets e JSPs, na plataforma servidora.

As tecnologias de serviço abrangem as API´s relativas a acessos a BD, transações, serviços de diretório e nomes (para proporcionar mecanismo de busca), serviços de mensagens, comunicação entre clientes e servidores, protocolos de uso em aplicações distribuídas, segurança e mecanismos de distribuição de carga entre servidores.

JNDI (*Java Naming and Directory Interface*) proporciona um meio para que uma aplicação localize componentes (objetos do sistema e da aplicação) que existem em um *namespace* (ambiente) de acordo com certos atributos.

Um arquivo de descrição de distribuição (*Deployment Descriptor File*) é um documento XML, cujos elementos descrevem como distribuir os componentes e

partes da aplicação em um determinado ambiente. Este documento, descrito pelo usuário, é processado pela implementação do J2EE.

Os serviços de transação, providos pelas API´s JTA, possibilitam que os usuários criem transações em um ambiente distribuído, dando suporte às ações de recuperação de falhas e multiprogramação, garantindo que cada transação seja completada sem interferência de outros processos. Os *containers* controlam todos os aspectos do processo, iniciando, propagando, mantendo e, baseado no descritor de distribuição, confirmando (*commiting*) ou não (*rolling back*) uma transação.

A plataforma J2EE ainda fornece serviços de segurança que garantem acessos autorizados aos recursos, através de um processo, em duas etapas, de autenticação e autorização. Neste processo, um programa autentica a si mesmo fornecendo os dados de autenticação (ID e *password*) de seu cliente. Depois de autenticado, o programa recebe autorização para acessar os recursos que são permitidos ao seu grupo (a autorização é baseada no conceito padrão de *security roles*).

As API´s JDBC 3.0 fornecem a conectividade necessária entre a plataforma J2EE e os bancos de dados, podendo realizar funções como conectar a um servidor de bando de dados, gerenciar transações, executar *stored procedures* e sentenças SQL e ter acesso a dados armazenados.

Mensagens assíncronas podem ser passadas na plataforma J2EE, graças as API´s JMS. A plataforma J2EE suporta comunicação direta (ponto a ponto com mensagens *callbacks*) ou indireta (via *mailboxes* **[Tanembaum, 2001]**).

Outro serviço suportado por J2EE é o JavaMail, que é uma coleção de API´s que dão suporte ao armazenamento, formatação e transporte de correio eletrônico. O JavaMail utiliza o JFA (*Javabeans Activation Framework*) para integrar o suporte para os tipos MIME na plataforma Java.

Há ainda os serviços de mensagens remotas via RMI (*Remote Method Invocation*), que é o modelo padrão Java para se implementar aplicações distribuídas, através do uso de suas API´s.

RMI-IIOP é uma implementação das API´s RMI sobre o IIOP (Internet Inter-ORB Protocol). RMI-IIOP permite que programadores escrevam interfaces remotas em Java. A interface remota pode ser convertida em IDL (Interface Definition Language), a qual pode ser usada em uma outra linguagem de programação que suporte os protocolos da OMG (Object Management Group) e inclua uma biblioteca ORB (Object Request Broker). RMI-IIOP fornece, portanto, a interoperabilidade necessária de objetos Java com objetos CORBA implementados em qualquer outra linguagem.

2.8 CONCLUSÃO

Discutimos acima as arquiteturas de objetos distribuídos COM e CORBA, bem como diferentes formas e tecnologias para interopera-los. Essas são as formas padrão de promover a interoperabilidade e cada uma tem seus pontos fortes e suas fraquezas.

De uma maneira geral, quem precisa de uma *bridge* COM/CORBA, ou investe em implementações de *enterprise application servers*, ou investe em

componentes proprietários, ou desenvolve arduamente seu próprio componente de integração.

Desde que foi estabelecido o padrão, a XML permitiu que ambientes heterogêneos compartilhassem informação pela WWW. Agora, através dela e da combinação com outra gama de tecnologias, também se pode compartilhar processos via web (mas, ao contrário das informações, os processos não são transportados via web). Esse compartilhamento de processos através da internet, que se convencionou chamar de Web Services é uma evolução das aplicações da XML na construção de sistemas, que agora passa a representar não somente a estrutura das informações, mas também a estrutura das mensagens trocadas entre aplicações distintas. Utilizamos esta faceta da XML para propor e implementar, de acordo com nossa proposta de dissertação, um tipo de *bridge* diferente das anteriores descritas, que promove a interoperação entre objetos COM/CORBA e ainda pode atravessar a barreira dos *firewalls* (ver capítulo cinco). Descreveremos mais os Web Services nos capítulos três, quatro e cinco desta dissertação.

3 XML-RPC

Em uma definição simples de suas funções, XML-RPC permite fazer chamadas remotas de funções, tornando possível estabelecer comunicações entre programas executando em computadores distintos. A XML, que é parte da infra-estrutura usada por XML-RPC, fornece o vocabulário para descrever chamadas remotas, as quais são transmitidas entre computadores usando o protocolo HTTP. Nesta dissertação, é explorada esta capacidade de XML-RPC para estabelecer comunicações (chamadas) nas pontes entre objetos de arquiteturas distintas. Explicaremos, a seguir, a especificação XML-RPC, que é encontrada em **[www9]**.

3.1 DESCRIÇÃO DO PROTOCOLO

Uma chamada XML-RPC é conduzida entre um processo cliente e outro servidor, disponível em um URL. Inicialmente, uma mensagem é passada por uma aplicação cliente a um processo cliente XML-RPC e nela são enviados o nome do método e os seus parâmetros (dados da chamada), além da especificação do servidor. Nesse processo, o cliente XML-RPC empacota os dados como XML e os envia ao servidor via HTTP POST. O servidor HTTP recebe a requisição POST e passa o seu conteúdo XML para um XML-RPC *Listener* (que é um manipulador de chamadas). Esse manipulador analisa o XML, obtém os dados e então chama o

método apropriado, com os devidos parâmetros. O método retorna uma resposta ao processo XML-RPC, que a empacota como XML e retorna essa estrutura como resposta à requisição HTTP POST. O Cliente XML-RPC analisa o XML recebido como resposta, obtém o valor retornado e então remete o dado para o programa cliente. O uso de HTTP significa que as requisições XML-RPC são síncronas e não há guarda de estado entre uma chamada e outra.

A implementação de aplicações servidoras XML-RPC é um pouco mais trabalhosa do que a implementação de clientes. Além de construir a lógica da aplicação, é necessário publicar os serviços em uma biblioteca XML-RPC, de tal forma que a mesma possa gerenciar as respostas. A aplicação servidora pode ser colocada no contexto de um servlet container, como o Tomcat, ou pode usar a classe *WebServer*, que vem com as implementações de XML-RPC, que cria um pequeno servidor para exclusivamente responder às chamadas XML-RPC.

3.1.1 SINCRONISMO

Uma requisição XML-RPC é sempre seguida de uma resposta XML-RPC, pois essa resposta precisa ocorrer na mesma conexão HTTP do pedido. Além disso, um processo cliente, que faz uma chamada XML-RPC, fica bloqueado até que a resposta de um processo servidor, que trate essa chamada, chegue. Isso pode ocasionar problemas para a aplicação cliente, se um determinado método do servidor tiver um tempo de resposta demasiado alto e o mesmo for freqüentemente chamado. É sugerido que o código seja escrito de tal forma que o

bloqueio não afete a sua execução normal e que, em alguns casos, o cliente restrinja ou proíba as chamadas a esses métodos custosos **[St. Laurent, 2001]**.

Mesmo sendo a chamada XML-RPC síncrona, é possível implementar um sistema assíncrono. Se houver necessidade de construir uma aplicação desse tipo é necessário usar a mesma solução para mensagens *callbacks* dos sistemas tradicionais. Ou seja, cria-se um método no objeto chamado que retorna *NULL* (caso de C++, por exemplo), recebendo esse mesmo método uma referência ao objeto chamado. O fato de retornar *NULL* garante que o primeiro processo prossiga sua execução e a referência permite que o objeto receptor estabeleça uma chamada ao objeto emissor da mensagem, posteriormente, para eventuais respostas. Claro que isso inclui mais overhead de comunicação na solução, pois seria necessário ter dois clientes e dois *listeners* em ambos os lados da chamada. Isso ainda vai aumentar em muito a complexidade dos códigos do cliente e do servidor envolvidos em uma chamada assíncrona. Uma saída possível seria fazer o método servidor retornar um identificador único para cada resposta e o processo cliente implementar um método XML-RPC especial que permita a recepção de múltiplos resultados e a identificação do pedido correspondente a cada um.

3.1.2 AUSÊNCIA DE ESTADO

Porque XML-RPC é baseado no HTTP, nenhum contexto é preservado entre uma requisição e outra. Ou seja, não há memória das operações anteriores. As aplicações construídas para a Web que precisam manter algum estado, o

fazem através de alguns recursos, como o uso combinado de cookies e valores armazenados no lado servidor, ou transmitem o estado de uma requisição a outra via URL.

Quando a manutenção de estado também é necessária entre chamadas XML-RPC, pode-se usar os parâmetros das chamadas, empacotados em XML, para o envio de dados ao servidor que, por sua vez, necessita de sistema de gerenciamento de estado (um banco de dados), a fim de poder identificar uma chamada como sendo a seqüência de outra e até usar o histórico armazenado para montar corretamente as respostas. É basicamente a mesma solução usada por *web designers* para manter estado entre uma operação e outra.

3.2 TIPOS DE DADOS

Uma chamada XML-RPC passada para um método remoto pode conter muitos valores passados como parâmetros, mas, como qualquer chamada feita a um subprograma, deverá ter apenas um único valor de retorno. Para representar esses valores, XML-RPC define um formato XML para cada tipo de dados básico, como inteiros, reais e strings, além de formatos para tipos estruturados, como arrays e registros. Todo item de dado, em uma chamada ou resposta XML-RPC, deve estar contido entre as tags *<value> ... </value>*.

3.2.1 TIPOS DE DADOS SIMPLES

XML-RPC tem apenas um tipo de dado para representar inteiros. Este tipo pode usar as tags <int> ou <i4>, representando um inteiro de 32 bits com sinal. Exemplos:

```
<value><int>inteiro</int></value>
ou
<value><i4>inteiro</i4></value>
```

Os valores que representam inteiros não devem conter espaços ou outros caracteres, apenas dígitos (0 – 9) e os sinais – ou +.

Os valores reais são representados como tipos de dados ponto flutuante, <double>, usando a representação padrão de 64 bits do IEEE (padrão IEEE 754). Não há representações de tipos reais abreviados (como float, por exemplo), como acontece em muitas plataformas. Um *double* é um valor que pode iniciar com um + ou -, e é seguido de dígitos, um ponto decimal e mais dígitos de precisão decimal. Exemplos:

```
<value><double>10.0</double></value>
<value><double>3.1416</double></value>
<value><double>-50.3456</double></value>
```

Valores lógicos são representados usando-se a tag <boolean> e tem dois valores possíveis, 1 (verdade) e 0 (falso):

```
<value><boolean>1</boolean></value>

<value><boolean>0</boolean></value>
```

As cadeias de caracteres podem se representados de duas formas distintas, com ou sem a tag <string>. Quando não estiver presente essa tag dentro de uma tag <value>, assume-se o tipo *default* como string. Exemplos:

<value><string>Interoperabilidade entre COM+ e CORBA</string></value>

<value>Interoperabilidade entre COM+ e CORBA</value>

O espaço em branco é significativo dentro dos strings. Caracteres delimitadores especiais usados por XML-RPC (tais como & e <) podem ser incluídos em uma string usando referências predefinidas (como & e <, respectivamente).

A especificação XML-RPC diz claramente que, apesar do padrão ser baseado em XML, que suporta vários conjuntos de caracteres, como o *Unicode*, por exemplo, os caracteres do tipo string estão restritos aos caracteres ASCII. Nem mesmo os caracteres acentuados de *Latin-1* estão disponíveis e o uso de outro grupo de caracteres sujeitará as aplicações a erros na codificação. Problemas similares podem surgir ao se usar referências de caracteres XML que não são explicitamente suportadas pela especificação XML-RPC.

Data e hora são codificados usando-se o tipo **dateTime.iso8601** (ver **[www10]**). Um valor *dateTime* em XML-RPC é representado precisamente com este formato

```
<value><dateTime.iso8601>aaaammddThh:mm:ss</dateTime.iso8601></value>
```

Por exemplo, a data

```
<value><dateTime.iso8601>20021231T23:59:00</dateTime.iso8601></value>
```

representa o último minuto do ano de 2002, ou seja, às 23 horas, 59 minutos e zero segundo, do dia 31 de dezembro de 2002. Como observado, esse tipo de dado é representado em XML-RPC sem indicar o GMT (*Greenwich Mean Ti*me).

Certos caracteres são proibidos em XML, como aqueles caracteres com valores ordinais menores que o ASCII do espaço em branco (ASCII 32), o que se torna um problema quando se deseja transportar arquivos binários, que podem conter esses caracteres. A solução para esse problema é usar a codificação *base-64*, freqüentemente presente em aplicações internet, como quando se precisa transportar e-mails com arquivos de imagem ou som anexados, por exemplo. A tag <*base64*>, portanto, é usada para englobar um objeto binário.

3.2.2 TIPOS DE DADOS COMPOSTOS

Um array XML-RPC não obriga que os seus elementos sejam do mesmo tipo, nem os numera, como em um array convencional. Para representar o array, se usa duas tags, <array.> e <data>, segundo o seguinte modelo:

```
<value>
        <array>
                <data>
                        <value>valor</value>
                <data>
        </array>
</value>
```

Um item de um array pode ser de qualquer tipo, simples ou composto. Dessa forma pode-se criar arrays multidimensionais pela adição de um array a um item de dado.

O tipo struct permite a representação de dados de uma forma similar aos arrays associativos de Perl (no qual os itens de array podem ser de tipos distintos e são armazenados com as chaves associadas a ele, formando um conjunto não ordenado; os seus elementos são recuperados e armazenados com funções de hash - sendo estruturas pouco apropriadas quando se deseja acessar todos os seus elementos seqüencialmente **[SEBESTA, 2001]**). Um struct é construído como uma série de membros (<member>), sendo cada um deles, um par

nome/valor (<name>, <value>). O nome deve ser uma string ASCII e o valor pode ser qualquer valor XML-RPC, incluídos aí os arrays e structs.

Exemplo:

```
<value><struct>
        <member><name>nome</name><value><string>Newton</string></value></member>
        <member><name>idade</name><value><int>64</int></value></member>
        <member><name>filhos</name><value>
                <array><data>
                        <value><string>Augusto</string></value>
                        <value><string>Luiza</string></value>
                        <value><string>Cristina</string></value>
                </data></array>
        </value></member>
</struct></value>
```

O equivalente em Perl:

```
%Pedroza = (‘nome’ => “Newton”, ‘idade’ => 64, ‘filhos’ => [“Augusto”, “Luiza”, “Cristina”] );
```

Ao contrário de Perl, XML-RPC não restringe que as chaves, ou seja, os nomes, dos elementos sejam únicos (mas na prática eles devem ser e algumas implementações sobrepõem membros anteriormente processados com o mesmo nome **[St.Laurent, 2001]**).

3.3 XML-RPC REQUEST

Uma requisição XML-RPC tem duas partes: os cabeçalhos HTTP que identificam o servidor e o XML *payload*, que contém informações sobre a mensagem a ser passada.

3.3.1 HTTP HEADERS

XML-RPC requer um número mínimo de cabeçalhos HTTP para serem colocados em conjunto com a mensagem. Exemplo:

```
POST /augustopedroza/serviço.jsp HTTP/1.0
User-Agent: helma XMLRPC 1.0
Host: member.isavvix.com
Content-Type: text/xml
Content-Length: 126
```

O método HTTP POST indica para o servidor que dados estão sendo enviados a ele. O *path* indica o caminho para o script que deve tratar os dados. Se o servidor é dedicado unicamente aos serviços XML-RPC, como o *XmlRpcServer* do pacote *helma.xmlrpc* (incluído na única implementação XML-RPC para Java), o path pode ter algum significado para o servidor **[McLaughin, 2001]**, dependendo da implementação. Especificamente, o path "/RPC2" é usado freqüentemente em algumas implementações XML-RPC (como Zope e Perl – **[St.Laurent, 2001]**),

mas não há necessidade especial de colocar um servidor em algum lugar específico do host.

O cabeçalho User-Agent contém um *id* correspondendo ao tipo de implementação XML-RPC que está sendo usada. É um item em geral de pouca importância, mas que pode apresentar variações interessantes, que podem entrar em cena quando for desejado lidar com níveis extras de controle ou informação. Como nenhuma dessas variações faz parte das implementações mais populares **[http://www.xmlrpc.com/spec]**, nem serve aos propósitos desta monografia, nós não abordaremos neste texto.

HOST especifica o nome do servidor que deve atender a requisição e também permite o uso de um servidor virtual, que compartilha o mesmo endereço IP com outros servidores, para prover políticas de segurança ou tratamentos diferenciados às requisições que necessitarem.

Como XML-RPC apareceu antes da RFC 3023 (ver **[www11]**) que especifica um mecanismo para criar identificadores mais específicos para XML, o valor de Content-Type é constante e deve ser “text/xml”. Outras implementações XML-RPC que aceitam outros tipos de XML precisam identificar as mensagens através de methodCall (que é o elemento raiz do *payload* de XML-RPC).

Content-Lenght deve conter o número de bytes presentes no corpo da mensagem XML-RPC, não o número de caracteres. Pois algumas implementações podem usar outro conjunto de caracteres que não o ASCII (o UTF de 16 bits, por exemplo).

Nenhum outro cabeçalho é exigido por XML-RPC. Mas isso não significa que não se possa usá-los. Por exemplo, apesar da autenticação básica HTTP,

como a RFC 1945, oferecer um baixo nível de segurança, pode-se usar os cabeçalhos *WWW-Authenticate* e *Authorization* para autenticação simples. Outros cabeçalhos podem ser úteis para manter sessão entre uma chamada e outra, passando-se dados da sessão ou mesmo *cookies*. Pode-se ainda usar cabeçalhos próprios usando-se o prefixo “X-“, conforme definido pela RFC 1521, para indicar sua natureza não oficial (X-DadosExtras: Informacao).

3.3.2 XML PAYLOAD

É a parte XML de uma chamada XML-RPC que contém os dados da mensagem remota (nome do método remoto e os parâmetros). O formato geral é o seguinte:

```
<?xml version=”1.0”?>
<methodCall>
        <methodname>nome_do_método</methodName>
        <params>
                <param><value><tipo>parâmetro1</tipo></value></param>
                <param><value><tipo>parâmetro2</tipo></value></param>
        </params>
</methodCall>
```

A mensagem fica dentro das tags <methodCall>...</methodCall>, sendo o restante do esquema auto-explicativo.

A lista de parâmetros pode conter zero ou mais valores e a tag <parms> deve sempre estar presente, mesmo na ausência de parâmetros. Não há formas de passar parâmetros referenciados por nome (como em Fortran e Python), nem criar métodos com número de parâmetros variáveis (como em C++). Mas pode-se simular esses e outros elementos de linguagens, usando-se o tipo <struct>, descrito anteriormente.

Os nomes de métodos podem conter apenas caracteres alfanuméricos e mais ".", ":", "_", e "/". Um exemplo completo de uma solicitação XML-RPC:

```
POST /augustopedroza/serviço.jsp HTTP/1.0
User-Agent: helma XMLRPC 1.0
Host: member.isavvix.com
Content-Type: text/xml
Content-Length: 165

<?xml version="1.0"?>
<methodCall>
	<methodName>raizQuadrada</methodName>
	<params>
		<param><value><double>25.0</double></value></param>
	</params>
</methodCall>
```

3.4 XML-RPC RESPONSE

A resposta a uma chamada XML-RPC também tem duas partes, os cabeçalhos e um corpo (XML Payload), não importando se o tratamento da mensagem recebida foi bem sucedido ou se houve erro.

Uma chamada de um procedimento remoto bem sucedida deve retornar apenas um elemento (que até pode ser de um tipo composto e aí conter muitos valores).

Uma mensagem de resposta precisa estar delimitada pela tag <methodResponse> e conter uma lista de parâmetros <params> com um único parâmetro <param>. Por exemplo:

```
<?xml version="1.0"?>
<methodResponse>
        <params>
                <param><value><double>5.0</double></value></param>
        </params>
</methodResponse>
```

Mesmo quando um método não retorna valor algum, que é caso de funções do tipo *void*, um valor precisa, obrigatoriamente, também ser retornado. Recomenda-se, nesses casos, que se retorne um valor <nil/> (presente em caráter experimental em implementações Java de XML-RPC – ver **[www12]**), ou um valor lógico para indicar se a ação pretendida foi bem sucedida (ou ainda retorne um valor qualquer padrão previamente determinado **[St.Laurent, 2001]**).

XML-RPC fornece um mecanismo similar às exceções de C++ e Java para tratar situações de erro. Quando isso acontecer, uma estrutura pré-definida <fault>..</fault> é colocada no método *response*, no lugar do resultado normal. Essa tag contém um valor único do tipo struct com dois membros, faultCode e faultString, que reportarão, respectivamente, o número do erro e uma mensagem com a descrição do mesmo. Exemplo:

```
<?xml version="1.0"?>
<methodResponse>
<fault>
        <value><struct>
        <member><name>faultCode</name><value><int>3</int></value></member>
        <member><name>faultString</name><value>Incorrect parameter</value></member>
        </struct></value>
 </fault>
</methodResponse>
```

Infelizmente, os códigos dos erros não são padronizados e, dependendo da implementação de XML-RPC usada, obtém-se códigos diferentes para um mesmo tipo de erro. O código acima corresponde à implementação de XML-RPC para Java (pacote *helma.xmlrpc*). Se fosse a de Perl, o código 3 seria relativo à mensagem

```
<value><string>No such method.</string></value>
```

Pode-se, entretanto, criar os códigos de erros próprios e usá-los para tratar os erros lógicos da aplicação, como em

```
<member><name>faultCode</name><value><int>800</int></value></member>
<member><name>faultString</name><value>numero negativo</value></member>
```

podendo as linguagens que suportam o conceito mapear as *faults* em exceções . Por exemplo, em Java, o pacote helma.xmlrpc usa a exceção XmlRpcException para manipular erros com XML-RPC. Em linguagens que não suportam exceções, como C, uma seleção simples pode ser utilizada para verificar o valor de retorno, observando se ele representa ou não uma condição de erro.

Os cabeçalhos que o servidor envia em uma resposta são semelhantes àqueles que são enviados por uma requisição:

```
HTTP/1.1 200 OK
Date: Tue, 19 Mar 2002 21:19:44 GMT
Server: Apache/1.3.12 (Unix) Debian/GNU JSP/2.1.1
Connection: close
Content-Type: text/xml
Content-Length: 818
```

O código de resposta deve ser sempre “200 OK” para uma chamada XML-RPC bem sucedida, mas o cliente deve estar apto a lidar com o conjunto de códigos de resposta HTTP, pois pode haver problemas no pedido, como um path incorreto, restrições de segurança etc. Assim como na requisição, o *Content-*

Lenght precisa conter o número de bytes da resposta. E isso significa que toda a resposta deve ser computada antes de ser enviada, o que ocasiona uma restrição ao uso de XML-RPC, pois não é possível criar respostas baseadas em *streams*. Isto é um problema especial quando a resposta é grande demais e há limites na memória disponível para a operação **[McLaughin, 2001]**. É possível criar uma implementação não padrão e omitir o *Content-Lenght* através do uso de HTTP 1.1, mas há o custo de ter que lidar com problemas de interoperabilidade.

3.5 CONCLUSÃO

XML-RPC é um protocolo simples que leva a tecnologia web a uma nova direção, no sentido de possibilitar a criação de simples e ao mesmo tempo poderosas conexões entre diferentes tipos de programas. Isto porque XML-RPC fornece uma camada de abstração que torna simples conectar diferentes tipos de sistemas computacionais sem a necessidade de criar um novo padrão para cada aplicação.

Além disso, pelas tecnologias envolvidas, HTTP e XML, os custos da implementação são baixos. E como se não bastasse, porque XML-RPC tem o foco restrito em resolver um único problema, o de fazer chamadas remotas, é até certo ponto fácil de se aprender e de implementar.

Como aspecto positivo final a se ressaltar sobre XML-RPC, as implementações vêm com servidores simples embutidos, o que facilita ainda mais a utilização desta tecnologia, desde que uma aplicação, ou uma chamada

específica dessa aplicação, não necessite de controles de segurança muito rígidos.

4 SOAP

SOAP (Simple Object Access Protocol), como o seu nome diz, é um protocolo simples (de usar, nem tanto de escrever **[McLaughin, 2001]**), projetado para facilitar as trocas de informações e para fazer chamadas remotas em uma arquitetura distribuída. O protocolo é leve, requerendo pouco overhead de um sistema e SOAP se destaca, atualmente, como a principal ferramenta usada para disponibilizar Web Services. Diante de tais características, pode-se afirmar que SOAP tem como objetivo de projeto “simplicidade e extensibilidade” (SOAP v1.1 Specification, Section 1.1 – ver **[www13]**).

Neste capítulo, iniciaremos fazendo uma descrição analítica das características gerais do protocolo. Depois mostraremos a estrutura de uma mensagem SOAP, apontando detalhes do cabeçalho HTTP e mostrando os elementos constituintes da mesma (*SOAP Payload*). Em seguida, faremos uma análise das características do protocolo e finalizaremos mostrando problemas de compatibilidade entre as mensagens de três diferentes implementações de SOAP: Axis, Microsoft SOAP Toolkit e Apache SOAP.

4.1 DESCRIÇÃO DO PROTOCOLO

De forma similar que XML-RPC, SOAP possibilita realizar as trocas de informações e fazer as chamadas remotas com uso do HTTP, o que permite contornar questões relativas a *firewalls* (uma chamada SOAP transpassa facilmente a maioria das barreiras) e detalhes do mecanismo de comunicação, que não fazem parte da lógica das aplicações, mas muitas vezes o programador precisa lidar com elas. Além disso, os documentos HTTP são apenas texto, o que permite fácil diagnóstico de problemas usando uma ferramenta baseada em texto. Há um amplo suporte para HTTP na internet, ele é independente de plataforma e, pelo fato de não guardar informação de estado (*stateless*) e não ser orientado a conexão, permite aos servidores SOAP proporcionarem alta escalabilidade, em relação a servidores que processam chamadas CORBA ou DCOM, que mantêm estado, contêm contadores de referências e até gerência automática de memória (DCOM).

A dúvida sobre qual protocolo utilizar na nossa pesquisa levou-nos a tentar responder a uma questão primordial: qual, dos protocolos baseados em XML, é o melhor para se usar em qual situação? Num primeiro instante, observamos que o mercado estava apontando para SOAP. No site da W3C, nós acompanhamos, desde março de 2000, o número desses protocolos baseados em XML crescer de quatro para quase trinta. XML-RPC, SOAP, WDDX e XMI formavam a lista inicial (este último, tem um propósito diferente daqueles protocolos que analisamos nesta dissertação).

Como observamos no capítulo anterior, XML-RPC (assim como WDDX) é um protocolo simples. Bem mais simples que SOAP, o que poderia apontá-lo

como um bom candidato para ser usado nas chamadas remotas de uma aplicação distribuída.

No entanto, XML-RPC e WDDX apresentam problemas, que acabam por deixá-los um passo atrás de SOAP, em questões de representação de tipos de dados (confiabilidade), de extensibilidade e suporte. No primeiro aspecto, nós observamos que XML-RPC tem que lidar com questões de descrição de tipos de dados através de DTDs. Já SOAP faz a descrição de dados através de *schemas*, combinando a sintaxe descritiva dos dados com informação de tipo dos mesmos, o que é uma vantagem. Quanto ao segundo aspecto, sempre que alguma alteração é feita no protocolo, XML-RPC tem que atualizar todas as implementações do protocolo em todas as linguagens e plataformas que o suportam. SOAP resolve o problema de atualizações simplesmente através da adoção de um novo *namespace*. E no aspecto de suporte, claramente as gigantes Microsoft e IBM adotaram a tecnologia SOAP, dando um impulso vitorioso nesse aspecto de quem se destaca mais (ou que protocolo domina o mercado). XML-RPC é mais humilde, sendo mantida por Dave Winer, da Userland, que "*claims a copywrite on the spec, but disclaims ownership of the protocol itself*" (ver **[www13]**).

Há ainda a se considerar dois aspectos que não têm paralelos em XML-RPC, que são o suporte a mensagens assíncronas (o termo técnico é *SOAP-Based Messaging*) e a possibilidade de uma mensagem SOAP sair de um ponto de origem, passar por vários pontos intermediários, ser processada nesses pontos intermediários (ativando serviços) até chegar a um ponto final.

Em uma mensagem SOAP, há três componentes básicos definidos na especificação: um modelo de empacotamento de mensagens (SOAP Envelope), um mecanismo de serialização chamado de *Encoding Rules* e um mecanismo para fazer RPC (SOAP RPC Representation).

O grupo responsável pela especificação SOAP projetou essas três peças para trabalharem juntas, mas nada impede um programador de usar somente um desses itens. Por exemplo, um programa pode usar as regras de codificação (SOAP Encoding) para salvar arquivos de configuração **[McLaughlin, 2001]**. Um sistema de mensagem instantânea poderia usar o SOAP Envelope para enviar dados entre o emissor e o receptor **[Seely, 2001]**. Poderia-se, também, ignorar o SOAP Envelope e usar as regras de codificação e o mecanismo de RPC (SOAP RPC Representation) para fazer as chamadas.

Com relação aos aspectos da simplicidade de SOAP e o fato de ele ser um protocolo e não uma arquitetura de sistema distribuído (portanto não especificando os serviços que tal estrutura geralmente apresenta), há alguns elementos da especificação que merecem ser comentados, pois estão relacionados com o foco desta dissertação.

4.1.1 O QUE SOAP NÃO CONTEMPLA

Vejamos o que diz a seção *Design Goals* da especificação SOAP 1.1 **[SOAP 1.1 W3C, 2000]**:

"A major design goal for SOAP is simplicity and extensibility. This means that there are several features from traditional messaging systems and distributed object systems that are not part of the core SOAP specification. Such features include: Distributed garbage collection, Boxcarring or batching of messages, Objects-by-reference (which requires distributed garbage collection), Activation (which requires objects-by-reference)".

Ou seja, SOAP não contém muito daquelas características que normalmente são encontradas em sistemas distribuídos tradicionais:

- *Distributed garbage collection.* Se um objeto é criado em um servidor por cliente SOAP e esse cliente finaliza antes de liberar o objeto, SOAP não diz o que fazer;
- *Boxcarring.* Significa que, se é necessário enviar múltiplas mensagens, tem-se que criar múltiplas mensagens com SOAP;
- *Objects-by-reference.* Ou seja, SOAP não diz que não se pode retornar uma referência para um objeto em uma resposta SOAP, o que significa que se uma referência for retornada é também necessário especificar como gerenciar o *distributed garbage collection*;
- *Activation.* SOAP não especifica como se pode obter uma referência para uma função ou objeto contido em uma requisição SOAP. Isto reduz os requisitos para qualquer sistema que quer usar SOAP.

Esses serviços são oferecidos comumente por arquiteturas distribuídas, como CORBA e DCOM. Ao contrário dessas arquiteturas, SOAP é meramente um

protocolo de comunicação. Por exemplo, SOAP não especifica, ao contrário de DCOM e CORBA, a semântica de transporte das mensagens (*one-to-one*, *request/reply*, *broadcast* etc **[Adam, 2001]**), nem existe ainda nenhum suporte para transações remotas (via Internet).

Também não há suporte direto para as questões de segurança mas, porque SOAP é baseado em HTTP, ele permite que se use alguns elementos de segurança de nível de aplicação, aliados a uma conexão HTTPS ou SSL.

4.1.2 ESTRUTURA DE UMA MENSAGEM SOAP

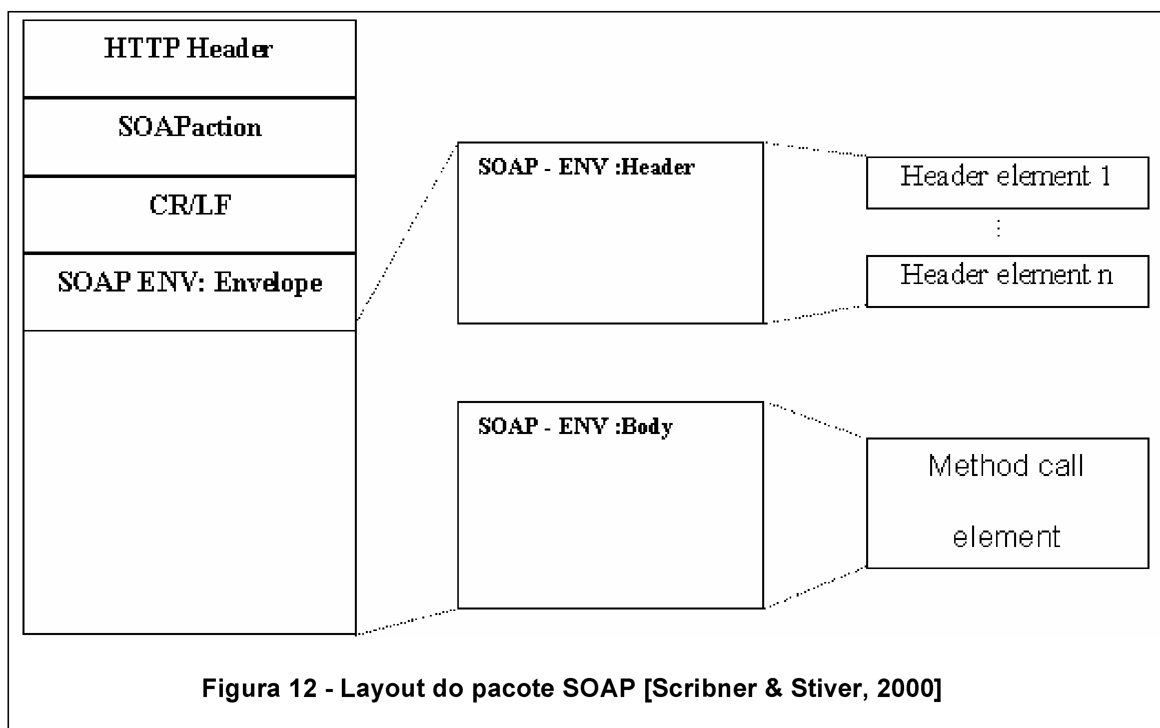

Figura 12 - Layout do pacote SOAP [Scribner & Stiver, 2000]

Como se pode ver na figura acima, o SOAP payload é encapsulado dentro do SOAP envelope e ele mesmo é parte do HTTP payload, assumindo-se que o

HTTP é o protocolo de transporte (em teoria, pode-se usar outros). O envelope SOAP consiste de um cabeçalho e um corpo. O cabeçalho é opcional, mas, se estiver presente, deve imediatamente seguir a abertura de uma tag XML no envelope. Se existir, portanto, haverá um ou mais elementos de cabeçalho, que proporcionarão meta-informações, relativas à chamada do método. Segundo a própria especificação de SOAP, essas meta-informações podem conter dados diversos, contudo, até o momento, a especificação SOAP usa a inclusão de um ID de transação. O corpo do SOAP Envelope (*body*) contém o nome do método remoto e seus argumentos serializados. Na seção seguinte, faremos uma descrição detalhada dessa estrutura e das regras de codificação de tipos em SOAP.

As implementações SOAP que atualmente estão sendo usadas referem-se à especificação 1.1 e nela se baseia todo o nosso trabalho com SOAP para esta dissertação.

4.1.3 TIPOS DE DADOS

Da mesma forma que XML-RPC, a especificação SOAP define como codificar os valores de diferentes tipos de dados. Mas as aplicações SOAP não precisam, necessariamente, usar a codificação SOAP, embora a especificação encoraje os programadores a usá-la. Ou seja, pode-se usar outros modelos de dados e estilos de codificação (o que a XML permitir) e ainda assim obter-se uma mensagem SOAP válida.

No apêndice A, apresentamos uma terminologia para os tipos de dados , definida pela especificação SOAP e que está relacionada com as regras de codificação do protocolo (ver **[www13]**). Mostramos também as regras para serialização do SOAP.

4.2 A MENSAGEM SOAP

Para formular o conteúdo de uma mensagem SOAP, os autores do protocolo escolheram a XML, porque ela contém um grande número de características, bem mais do que SOAP usa ou precisa. Por exemplo, a especificação SOAP diz, *ipsis litteris*, que "*A SOAP message MUST NOT contain a Document Type Declaration. A SOAP message MUST NOT contain Processing Instructions*" **[http://www.w3.org/TR/SOAP/#_Toc478383492]**. Dos elementos de XML adotados por SOAP, todos são detalhadamente especificados sobre como os mesmos serão usados pelo protocolo. Essas características tornam simples implementar soluções usando SOAP, porque os programadores não precisam contar com um XML *parser* muito desenvolvido **[Seely, 2001]**.

Assim, todas as mensagens SOAP são documentos XML, que sempre contêm um *Envelope*, contendo o mesmo um *body* (obrigatório) e podendo também conter um *header* (opcional).

4.2.1 SOAP ENVELOPE

O envelope é o elemento (TAG) que envolve toda a mensagem SOAP. Ele apresenta as seguintes regras sintáticas:

- O elemento sempre tem como nome *Envelope*;
- Cada mensagem SOAP deve contê-lo envolvendo a mensagem;
- O elemento pode conter declarações de *namespace* e atributos adicionais. Todos os atributos contidos dentro do elemento devem ser qualificados;
- Cada elemento também pode conter sub elementos. Como os atributos adicionais, esses elementos devem ser qualificados e devem estar após o elemento SOAP Body.

Como todos os elementos em uma mensagem SOAP, o *Envelope* deve especificar o estilo da codificação, através do atributo *encodingStyle*, que pode estar presente em qualquer um dos três elementos SOAP (Envelope, Header ou Body).

SOAP não define o modelo de versão tradicional baseado em número de versão. Ao invés disso a mensagem SOAP tem um elemento envelope associado com o *namespace* "http://schemas.xmlsoap.org/SOAP/envelope". Se uma aplicação SOAP receber uma mensagem na qual o elemento envelope esteja associado com um *namespace* diferente ocasionará num erro com a mensagem "*Version Mismatch error*".

Além dos elementos header e body do envelope se pode adicionar outros elementos (sub elementos) de acordo com a necessidade. Por exemplo, imaginemos que é preciso recriar a funcionalidade *causality* que é usada no

DCOM para evitar *deadlock*. Seria preciso inserir na mensagem SOAP uma informação adicional *causality*. Para fazer isso, deve-se criar um novo *schema* que especifique o layout do elemento *causality* e incluir o *namespace* para o elemento, para assim se conseguir implantar a funcionalidade.

```
<SOAP-ENV:Envelope>
        <SOAP-ENV:Header />
        <SOAP-ENV:Body>
                <- - Serializar a  informação aqui... - ->
        </SOAP-ENV:Body>
        <!- -CausalityID representa algum valor temporariamente
             O tipo de dado seria especificado no schema...- ->
        <CID:CausalityID
                xmlns:CID="urn:schemas-casodeuso.com:causalityid">
                1234
        </CID:CausalityID >
</SOAP-ENV:Envelope>
```

Apesar de *Header* e *Body* serem definidos como elementos independentes, eles apresentam alguma relação. O relacionamento entre uma entrada do *Body* e uma do *Header* é que uma entrada *Body* é equivalente, semanticamente, a uma entrada do *Header* dirigida ao *actor default* e com um atributo SOAP mustUnderstand com o valor "1". O *actor default* é indicado ao não se usar o atributo actor.

4.2.2 SOAP HEADER

Como já foi dito antes, o *header* de uma mensagem SOAP é opcional. Entretanto, se ele estiver incluso em uma mensagem, precisa ser a primeira estrutura presente no Envelope (a especificação chama uma estrutura interna de *child*, seguindo o padrão XML). O *header* estende a mensagem SOAP de forma modular, através de atributos, proporcionando um mecanismo para se acrescentar informações sobre a mensagem na própria mensagem.

Uma mensagem SOAP trafega de uma aplicação cliente a um destino final, podendo passar através de um conjunto de nós intermediários pelo caminho. Cada nó é uma aplicação que pode receber e repassar mensagens SOAP. O cabeçalho SOAP pode ser usado para indicar algum processamento adicional no nó (independente do processamento feito no destino final), como autenticação do cliente, gerenciamento de transação etc. Para protocolos que não provêem um caminho de retorno (como FTP, que não pode iniciar um *get request* **[Seely, 2001]**), o *header* pode também comunicar o caminho de retorno para a resposta.

No exemplo, o *header* indica que essa é uma transação (um URI especifica o *namespace* para a transação), mas poderia ter especificado atributos para outro tipo de processo, como verificação de permissão, por exemplo. O atributo *mustUnderstand="1"* significa que o nó inicial no caminho da mensagem SOAP precisa processar o *header*. O valor 10 é passado para o nó inicial como entrada.

Observe-se que o filho imediato de Header precisa usar um nome de elemento totalmente qualificado. Esse nome consiste do *namespace URI* e o nome local. Seguindo as regras XML, a aplicação assume que qualquer elemento contido no filho imediato de *Header* está no mesmo *namespace*.

O atributo *mustUnderstand* pode ter apenas dois valores, zero (a entrada é opcional, pode pular esse elemento se ele não fizer sentido) e um (a entrada é obrigatória, precisa reportar erro se o elemento não fizer sentido). O fazer sentido refere-se à capacidade do nó de processar aquele elemento. A ausência do atributo é semanticamente equivalente à sua presença com valor zero. Isto permite que os servidores que não sabem o fazer com o *Header*, o ignorem e ainda forneçam uma resposta **[McLaughin, 2001]**.

Mencionamos anteriormente que os nós intermediários, por onde passam as mensagens SOAP, podem realizar algum processamento adicional. Ou seja, nem todas as partes de uma mensagem SOAP são dirigidas ao destino final e partes podem ser dirigidas a um ou mais intermediários no caminho da mensagem. Esses recipientes intermediários (aplicações, na verdade) não podem repassar esse elemento de cabeçalho (a eles destinado) para a próxima aplicação no caminho da mensagem SOAP e devem removê-lo. As aplicações intermediárias e o destino final são identificados através de um URI e são tratados como atores (*actors*) pela especificação SOAP. Esses atores podem determinar que cabeçalho é necessário processar, procurando pelo atributo *actor* com o URI *http://schemas.xmlsoap.org/soap/actor/next*. Quando este atributo estiver ausente em um *Header*, é porque a aplicação que recebeu a mensagem é o recipiente final.

Resumindo, todos os elementos filhos do *Header* são chamados de entradas do *Header*. As regras que se aplicam para as entradas do *Header* são:

- As entradas do *header* são qualificadas pelo nome do elemento, que consiste de um *namespace* e um nome local. Todos elementos filhos do *header* devem ser qualificados por *namespaces*.
- O *header* deve obedecer a especificação SOAP, a menos que tenha um atributo *encodingstyle*.
- Os atributos *MustUnderstand* e *actor* devem ser usados para indicar como processar as entradas e por quem.

4.2.3 O ATRIBUTO ENCODINGSTYLE

O atributo global *EncodingStyle* pode ser usado para indicar regras de serialização de dados aplicadas nas mensagens SOAP. O valor do atributo é uma lista ordenada de um ou mais URI´s que especifica a regra (ou as regras) que pode ser usada para desserializar uma mensagem SOAP, indicada na ordem da mais específica para a menos específica.

O padrão usado para o atributo *EncodingStyle* é "*http://schemas.xmlsoap.org/SOAP/encoding/*", porém pode-se mudar o valor padrão usando-se um algoritmo próprio de serialização.

4.2.4 SOAP BODY

O SOAP Body contém informações obrigatórias e direcionadas para o destino final da mensagem. O seu uso padrão inclui envio de chamadas RPC e mensagens de erro. O SOAP Body é o local onde se pode encontrar o nome do método, argumentos serializados e seus valores, além de informações de resposta dos métodos serializadas.

A especificação SOAP indica três tipos de elementos do *body*:

- Chamada (Call): pacote de chamada dos métodos;
- Resposta (Response): resposta do método remoto;
- Erro (Fault): pacote de erro;

CALL BODY

O primeiro elemento tem o mesmo nome que o nome do método que está sendo chamado. Os elementos que estão contidos dentro do elemento método representam os argumentos serializados e cada argumento tem o nome de acordo com a assinatura do método. Se não se conhece os nomes dos argumentos pode-se substituí-los por *paramxxx*, onde *xxx* é o número da ordem do parâmetro de acordo com a ordem na qual ele aparece na assinatura do método. O primeiro parâmetro é *param0*, o segundo é *param1* e assim por diante.

RESPONSE BODY

A estrutura é semelhante a da chamada, só que o primeiro elemento do *body* tem o nome formado pelo nome do método concatenado com a palavra *response*, por convenção. O valor de retorno do método é serializado dentro das tags <return></return>. Se o método tem parâmetros [out] ou [in,out], os valores de retorno devem vir no elemento de resposta usando as mesmas regras de nome para a chamada.

FAULT BODY

O elemento *fault* é usado para carregar erros e/ou informação de status. Se presente ele não deve aparecer mais do que uma vez no elemento *Body*.

Fault define os seguintes elementos:

- *faultcode*: Código usado pelo software para prover algum meio de, programaticamente, identificar o erro que aconteceu. Todos os elementos *fault* precisam incluir um *faultcode*;
- *faultstring*: Proporciona uma explicação do erro em forma de texto, de maneira a ser compreendido por seres humanos. Também é cláusula obrigatória no *fault*;
- *faultactor*: Aponta qual foi a aplicação SOAP presente no caminho da mensagem que reportou o erro. A obrigação de incluir esse elemento é das aplicações intermediárias. A aplicação final tem a opção de escrevê-lo ou não. Quando enviado, o valor é um URI indicando a fonte do *fault*;

- *detail*: Mostra informações de erros específicas da aplicação relacionando-os com o elemento *Body*. Deve estar presente se o conteúdo de *Body* não puder ser processado com sucesso. Esse elemento não pode ser usado para reportar erros do *Header*. Fazendo isso, sempre que ele estiver presente significa que o problema aconteceu no Body. Na sua ausência, o problema aconteceu em outro lugar qualquer;

A especificação SOAP define quatro códigos de erros:

- *VersionMismatch*. A aplicação encontrou um namespace inválido para o SOAP Envelope;
- *MustUnderstand.* Uma aplicação que deveria efetuar algum processamento (indicado pelo valor de URI do actor), não entendeu o elemento do Header que teve o atributo mustUnderstand colocado como valor “1” (processamento obrigatório);
- *Client.* Essa classe de erro indica que um problema existiu em algum lugar dentro da mensagem original. Esses erros aparecem em mensagens mal formadas, bem como em mensagens que não contêm informações suficientes;
- *Server.* Esses erros se relacionam ao processamento da mensagem pelas aplicações, não necessariamente na mensagem em si (falta de memória, queda de conexão ao receber streams etc);

4.3 SOAP e HTTP

SOAP não altera a semântica de HTTP que tem suas próprias regras para transmitir vários tipos de dados. Tanto SOAP como HTTP têm o conceito de intermediários, mas com características distintas. SOAP usa *actors* para atuarem nesta funcionalidade. Um servidor intermediário HTTP deve realizar atividades HTTP: autenticação, tradução **[St.Laurent, 2001]**, o que for. E aplicações SOAP baseadas em HTTP precisam indicar que os dados que estão sendo transmitidos são do tipo “text/xml”.

4.3.1 SOAP HTTP Request

Pode-se usar SOAP com HTTP usando diferentes métodos (HTTP request). A especificação, no entanto, somente define um “encaixe” (binding) em relação a HTTP POST, sendo *“...waste of time creating a new way to do SOAP over HTTP”* **[Seely, 2001]**, porque não haveria suporte para a nova solução no mercado.

Pode-se indicar a intenção de um SOAP Request através de uma nova variável chamada *SOAPAction*, cujo valor é um URI. O URI não necessita estar em um formato específico e não precisa ser real (*resolvable*). Essa nova variável, na presença de um SOAP Request, pode informar uma das três seguintes coisas:

1. Se estiver presente e com valor, esse valor fornece a intenção (o objeto e o método que será executado). Por exemplo, *SOAPAction: "http://Pedroza02/Calculadora.soma"*;
2. Se estiver presente mas sem valor, a intenção necessita ser fornecida pelo URI. Por exemplo, *SOAPAction = “”*;

3. Se estiver ausente, significa que não existe indicação do intento da mensagem;

4.3.2 SOAP HTTP Response

Aqui a semântica dos códigos de status HTTP é seguida. Ou seja, se a resposta vier com um código 2xx, a mensagem foi bem compreendida, recebida e processada. Se um erro ocorrer no processamento da mensagem, a aplicação SOAP deve retornar um HTTP 500 "*Internal Server Error*", com um *Body* contendo um elemento SOAP Fault indicando a natureza do erro.

4.3.3 HTTP Extension Framework

O HTTP Extension Framework possibilita muitas coisas diferentes. SOAP o usa para identificar a presença de um SOAP Request dentro de uma mensagem assim como o intento desse *request*. Uma aplicação SOAP cliente somente usa esse mecanismo quando ambas as entidades sabem como usá-lo.

Se um cliente quer usar o Extension Framework ele envia a mensagem como um "M-POST" e uma declaração obrigatória. Se o servidor não suporta esse tipo de pedido, ele retornará o código 510 ("Not Extended"). A mensagem pode ser re-enviada pelo cliente sem usar o *framework*. O identificador usado para a mensagem SOAP usando o Extension Framework é o "*http://schemas.xmlsoap.org/soap/envelope/*".

4.4 ANÁLISE DO PROTOCOLO

A especificação SOAP é bastante vaga e proporciona muita flexibilidade na maneira em que se escolhe montar as mensagens.

Por exemplo, na seção 5 da especificação, *Data Encoding Section*, a nona regra diz que valores *null* podem ser representados tanto pela omissão do elemento XML que representa esse valor, como por um elemento com o atributo *xsi:nil* presente e seu valor atribuído a *true*. Suponhamos que um servidor SOAP disponibilize serviços, utilizando em uma chamada a um serviço, como muitas linguagens de programação o fazem, a leitura dos argumentos de maneira posicional, sem se importar com o nome dos argumentos do método que implementa esse serviço. Um cliente para esse serviço pode preferir montar as chamadas omitindo os argumentos com valor null, gerando problemas na chamada.

Outro caso em que e especificação também não é clara, é sobre como responder a uma chamada que apresenta um tipo de retorno void e nenhum parâmetro passado por resultado. Por exemplo, essa resposta pode conter um envelope vazio, ou um envelope com um elemento de resposta do método vazio, ou mesmo uma resposta HTTP 204 (sem resposta) **[Graham et al, 2002]**, sem envelope.

Já o cabeçalho HTTP *SOAPAction* também é definido vagamente na seção 6.1.1 da especificação, como representando a intenção da mensagem. Alguns pacotes, como o Microsoft SOAP Toolkit e o Apache SOAP, não usam este valor e

outros requerem o mesmo para acionar o serviço correto. Por essa razão, *SOAPAction* tem sido fonte de problemas de interoperabilidade entre implementações SOAP.

A especificação também diz que podem existir muitas entradas BODY, como no caso de mensagens serializadas, mas não está claro como saber, exatamente, qual é a principal entrada que deve ser processada como uma chamada remota a um método (serviço) oferecido por um servidor.

Um último problema que podemos citar com a especificação, é que ela diz ser possível fazer mensagens auto-descritivas com SOAP, mas não explicita uma maneira uniforme de fazê-lo.

Os metadados sobre o conteúdo da mensagem, que descrevem, por exemplo, quais são os tipos dos atributos (*xsi:type*) de uma chamada, estão embutidos dentro da mensagem. É igualmente possível assumir que tais metadados são informados usando WSDL, a linguagem de descrição de Web Services. Este é o caso do Microsoft SOAP Toolkit, que utilizamos nas aplicações descritas no capítulo cinco. É até possível que não haja descrição dos tipos de dados, como no Apache SOAP que também utilizamos nessas aplicações descritas.

Essas questões, além de outras semelhantes, proporcionam desafios quando se tenta interoperar implementações distintas de SOAP. No capítulo cinco, descreveremos nossa experiência neste particular.

4.5 PROBLEMAS PARA ALCANÇAR A INTEROPERABILIDADE

Nossa proposta nesta dissertação visa promover a interoperabilidade entre COM e CORBA através de uma *bridge* baseada em protocolos XML. Para demonstrar alguns problemas gerados pelo pouco rigor da especificação SOAP e também para justificar a solução adotada por nós no capítulo cinco dessa dissertação, na implementação de uma *bridge* baseada em SOAP, vamos analisar os problemas de compatibilidade entre as mensagens de três diferentes implementações de SOAP: o novo Apache Axis 1.0, o Microsoft SOAP Toolkit 2.0 SP 2 e o Apache SOAP 2.3.1.

Todas as mensagens mostram um cliente SOAP chamando a operação *soma* em uma calculadora remota. As mensagens enviam um argumento, de valor 10.

A primeira mensagem que analisaremos, é gerada pelo Microsoft SOAP Toolkit.

```
POST /Calculadora HTTP/1.1
Content-Type: text/xml; charset="utf-8"
Content-Length: 398
SOAPAction: "http://Pedroza02/Calculadora.soma"

<?xml version="1.1" encoding=”UTF-8”?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
     <SOAPSDK1:soma xmlns: SOAPSDK1=" http://schemas.xmlsoap.org/message">
       <valor>10</valor>
```

```
    </SOAPSDK1:soma>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Na mensagem acima, o nome do método chamado aparece tanto no cabeçalho HTTP (SOAPAction), como na parte *BODY* do *Envelope*. Além disso, os parâmetros são nomeados (da mesma forma que em algumas Linguagens de Programação, como Fortran 90 e Ada 95). Ou seja, há uma tag com o nome do parâmetro (e se há mais de um parâmetro, não importa a ordem em que os mesmos são colocados na mensagem). Um arquivo WSDL, gerado pelo *toolkit* (embora pudesse ser codificado manualmente), que vai junto com a mensagem, também especifica o *SOAPAction* (na seção *Binding* do documento), além de relacionar os *labels* das tags dos argumentos. Se a mensagem contém tags de argumentos incorretas ou não contém um valor para *SOAPAction*, a mesma irá falhar quando chegar ao servidor. Logo abaixo, estão em destaque, no arquivo WSDL gerado pelo MS SOAP Toolkit, as descrições de *SOAPAction* e das tags dos parâmetros. Dado o tamanho do arquivo, mostramos apenas as partes de mais interesse para esta explicação.

```
<?xml version='1.0' encoding='UTF-8' ?>
<definitions  name ='ClienteCalculadora'   targetNamespace = 'http://tempuri.org/wsdl/'
          xmlns:wsdlns='http://tempuri.org/wsdl/'
          xmlns:typens='http://tempuri.org/type'
          xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
          xmlns:xsd='http://www.w3.org/2001/XMLSchema'
```

```
            xmlns:stk='http://schemas.microsoft.com/soap-toolkit/wsdl-extension'
            xmlns='http://schemas.xmlsoap.org/wsdl/'>
  <types>  ………   </types>
  <message name='Calculadora.soma'>
    <part name='valor' type='xsd:double'/>
  </message>
   ……….
  <portType name='CalculadoraSoapPort'>
    <operation name='soma' parameterOrder='valor'>
      <input message='wsdlns:Calculadora.soma' />
      ………..
    </operation>
  </portType>
  <binding name='CalculadoraSoapBinding' type='wsdlns:CalculadoraSoapPort' >
    <stk:binding preferredEncoding='UTF-8'/>
    <soap:binding style='rpc' transport='http://schemas.xmlsoap.org/soap/http' />
    <operation name='soma' >
      <soap:operation soapAction=' http://Pedroza02/Calculadora.soma' />
      <input>
        <soap:body use='encoded' namespace='http://tempuri.org/message/'
                    encodingStyle='http://schemas.xmlsoap.org/soap/encoding/' />
      </input>
      ………
    </operation>
  </binding>
  <service name='CalculadoraSoapService' >
    <port name='CalculadoraSoapPort' binding='wsdlns:CalculadoraSoapBinding' >
      <soap:address location='http://Pedroza01/Dissert/MSApache/CalculadoraSoapService.exe' />
    </port>
  </service>
</definitions>
```

Um arquivo WSDL pode conter até sete elementos. *Types* é o elemento que contém definições de tipos de dados. *Schemas* podem ser usados para descrever os tipos. *Message* define os dados que estão sendo passados entre emissor e receptor. Inclui os dados de entrada e saída. *Operation* é usado para a definição de uma ação que um serviço pode executar. *Port* define a localização (*endpoint*) de um serviço definido em *Binding*. *PortType* descreve o conjunto de operações que um ou mais *ports* (URLs) suportam (no exemplo acima, há apenas uma operação, soma). Cada operação tem pelo menos uma entrada e uma saída, que estão mapeadas no elemento *message*. A operação de saída (não mostrada) tem um sufixo "*Response*" acrescentado ao nome da mensagem pelo Microsoft SOAP Toolkit WSDL File Generator. A associação do protocolo com a descrição dos dados para uma certa porta define o *Binding*. *Service* é uma coleção de *ports* relacionadas.

A mensagem abaixo mostra a mensagem SOAP gerada pelo Apache SOAP. Em destaque estão as diferenças para a chamada gerada pelo *toolkit* da Microsoft.

```
POST /Calculadora HTTP/1.1
Content-Length: 426
Host: Pedroza02
Content-Type: text/xml; charset="utf-8"
<!-- não há SOAPAction -->
<?xml version="1.1" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
```

```
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <soma>
      <arg0>10</arg0>
    </soma>
  </SOAP-ENV:Body;
</SOAP-ENV:Envelope>
```

Observa-se que não há um *SOAPAction* no cabeçalho HTTP e o argumento da chamada está codificado não como *valor*, mas como *arg0* (se houvesse outro haveria um *arg1* e assim por diante). As aplicações SOAP servidoras baseadas no Apache SOAP somente procuram pelo nome da operação no *Envelope*, ignorando SOAPAction. Além disso, lêem os parâmetros baseados apenas na ordem em que estão posicionados. Também não há a necessidade de um arquivo WSDL, nem há a definição de tipos de dados dos parâmetros.

Mas a diferença entre as tecnologias está diminuindo graças aos esforços dos fabricantes de software e à nova especificação SOAP 1.2. Para dar uma idéia, observemos a mesma chamada SOAP, gerada pelo no Apache Axis (laçado em novembro de 2002), com os mesmos destaques nos mesmos pontos já analisados na mensagem do Apache SOAP, mostrada anteriormente:

```
POST /Calculadora HTTP/1.1
Content-Length: 458
Host: Pedroza02
Content-Type: text/xml; charset="utf-8"
SOAPAction: ""
```

```
<?xml version="1.1" encoding=”UTF-8”?>
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:xsd=”http://www.w3.org/2001/XMLSchema”
 xmlns:xsi=” http://www.w3.org/2001/XMLSchema-instance” >
  <SOAP-ENV:Body>
    <soma>
      <arg0 xsi:type=”xsd:string”>10</arg0>
    </soma>
  </SOAP-ENV:Body;
</SOAP-ENV:Envelope>
```

Ou seja, existem o cabeçalho SOAPAction, com valor vazio, o que pode ser usado para melhorar a segurança. Os argumentos das chamadas continuam com a característica de serem posicionais, mas já há informação de tipo.

Além dessas características, o Apache Axis 1.0 já é capaz de gerar um arquivo WSDL a partir de uma ferramenta gratuita criada pela IBM, o Java2WSDL. O Axis, no entanto, ainda não processa, na sua versão inicial, o WSDL recebido em uma chamada (simplesmente despreza-o). Essas mudanças melhoram muito, mas ainda não resolvem todos os problemas de interoperabilidade.

4.6 CONCLUSÃO

SOAP é um protocolo que faz um uso inovador e convincente de tecnologias disponíveis que não estão ligadas a um único fornecedor de software.

O seu propósito é especificar um protocolo que pode ser usado para facilitar aplicações distribuídas, proporcionando a realização de chamadas remotas, via internet. Para esse fim, SOAP usa um protocolo de transporte, HTTP (ou outro), para levar mensagens serializadas codificadas em XML entre uma aplicação e outra. As principais motivações para o uso de SOAP envolvem (1) base tecnológica para uso em sistemas distribuídos orientados a objetos (que usam a internet); (2) o protocolo não é vinculado a nenhum fabricante; (3) o encorajamento para a implementação de aplicações fracamente acopladas **[Scribner & Stiver, 2000]**; como é a arquitetura de *bridge* que propomos neste trabalho; (4) Simples, porque é baseado em texto e (5) extensível, porque as modificações não têm alterado as aplicações, visto que o XML das mensagens SOAP usa *schemas* e *namespaces*, ao contrário de XML-RPC que usa DTDs.

5 INTEROPERABILIDADE

Neste capítulo, nós descreveremos como obter a interoperabilidade COM/CORBA através de *bridges* que usam protocolos baseados em XML. Primeiramente discutiremos aspectos gerais da interoperabilidade entre COM e CORBA para, em seguida, passar a analisar a arquitetura da solução proposta utilizando XML.

Em seguida compararemos a solução que propomos, utilizando XML, com as soluções existentes, analisadas no capítulo dois. Enfatizaremos as restrições existentes nestas soluções que não utilizam XML e que foram melhoradas por nossa proposta.

Com o objetivo de mostrar a orquestração e verificar as dificuldades práticas da interoperabilidade utilizando XML, foram construídas duas bridges e, além disto, para testar as *bridges*, foram implementadas também duas aplicações.

A primeira bridge utiliza a tecnologia SOAP como elemento central e a segunda *bridge* utiliza o protocolo XML-RPC. Uma das virtudes da arquitetura utilizada é permitir a substituição do protocolo de comunicação sem alterações substanciais na solução proposta. Portanto, seria simples construir uma terceira *bridge* utilizando outras tecnologias de web services.

O lado cliente de ambas as *bridges* (denominado emissor) foi implementado para interagir com um objeto COM e o lado servidor das duas *bridges* (denominado receptor) foi implementado para interagir com um objeto CORBA. A solução proposta também poderia ter sido implementada da forma inversa, ou seja, emissor interagindo com CORBA e receptor interagindo com COM.

Duas aplicações foram construídas para testar as bridges: Uma máquina de Calcular e uma Agenda.

Na primeira aplicação, o cliente da Calculadora (implementado em C++) utiliza as brigdes para invocar o servidor da Calculadora (implementado em Java) que realiza as operações matemáticas.

Na segunda aplicação, o cliente da Agenda (implementado em ASP com interface Web) usa a mesma arquitetura da *bridge* proposta para pesquisar informações e solicitar operações de atualização em um banco de dados a um servidor da Agenda (implementado na plataforma J2EE).

5.1 INTEROPERABILIDADE COM/CORBA

COM e CORBA compartilham muitas características (ver capítulo dois). De um modo geral, as interfaces COM têm funcionalidades equivalentes às interfaces CORBA. Além disso, os ponteiros COM são equivalentes às referências CORBA e existe muita similaridade nos métodos de chamada e nos tipos de dados, o que torna viável a interconexão entre as duas arquiteturas, via mapeamento.

Assim, um objeto cliente COM pode interagir com um objeto servidor CORBA através de um objeto visão COM (em verdade, um *proxy* para o objeto

CORBA) que fornece os métodos e interfaces necessárias para estabelecer a comunicação de COM para CORBA. O contrário também é possível, ou seja, um objeto cliente CORBA pode acessar um objeto servidor COM através de um objeto visão CORBA. Esse objeto visão pode ser um componente único em *bridges* bidirecionais, conforme explicado nos capítulos um e dois desta dissertação (ver figuras 3 e 8).

O objeto cliente trata o objeto visão como se ele fosse o objeto real, da mesma forma que os objetos distribuídos tratam os *stubs* como se fossem os objetos remotos. O objeto visão, por sua vez, proporciona a comunicação com o objeto servidor, de maneira transparente para o objeto cliente, via RPC.

A especificação CORBA há muito reconheceu a importância da interconexão. A especificação inclui mapeamento bidirecional entre objetos CORBA e objetos COM, que abrange as interfaces COM padrão (chamada *IUnknown*) e inclusive a interface Automation (usada para os tipos de automação OLE – chamada *IDispatch*). A especificação inclusive recomenda que a *bridge* obtida por mapeamento fique localizada na plataforma COM, o que é natural, uma vez que, apesar de existirem soluções COM para outros sistemas operacionais, como o Tru65 Unix, da Compaq **[Microsoft, 2002]**, é difícil outro ambiente suportar o padrão.

Algumas empresas implementaram *bridges* de acordo com o padrão OMG, como Iona, Digital BEA Systems e a Visual Edge Software (ver item 2.5 desta dissertação e links no apêndice C).

A solução mais direta para interoperar as arquiteturas CORBA e COM, usando protocolos baseados em XML, seria implementar os componentes já

empacotando chamadas usando o formato SOAP ou o formato XML-RPC (solução mais simples). Mas, reescrever o código fonte para tais componentes *brigdes* sempre esteve fora dos objetivos e da realidade deste trabalho, tendo em vista as restrições de tempo e de número de pessoas envolvidas nesta pesquisa.

5.2 ARQUITETURA DA SOLUÇÃO APRESENTADA – A BRIDGE XML

A solução usada nas aplicações que desenvolvemos foi fundamentalmente a mesma, ou seja, criar uma *bridge*, com protocolos baseados em XML, entre diferentes sistemas distribuídos baseados em objetos. Pelo uso dessa *bridge*, um objeto de um sistema pode fazer uma chamada remota para um objeto de um outro sistema, valendo-se de protocolos baseados em XML. Ou seja, a *bridge*, em si, é composta:

1. pela própria infra-estrutura e pelos mecanismos usados nas chamadas remotas SOAP e XML-RPC;
2. por artefatos de software acrescentados a cada solução distribuída, de modo a permitir que a mesma possa fazer (receber) uma chamada remota para (de) um objeto de um outro sistema distribuído (baseado em objetos);

Essa solução tem um esquema geral que se apresenta assim delineado, de modo simplificado:

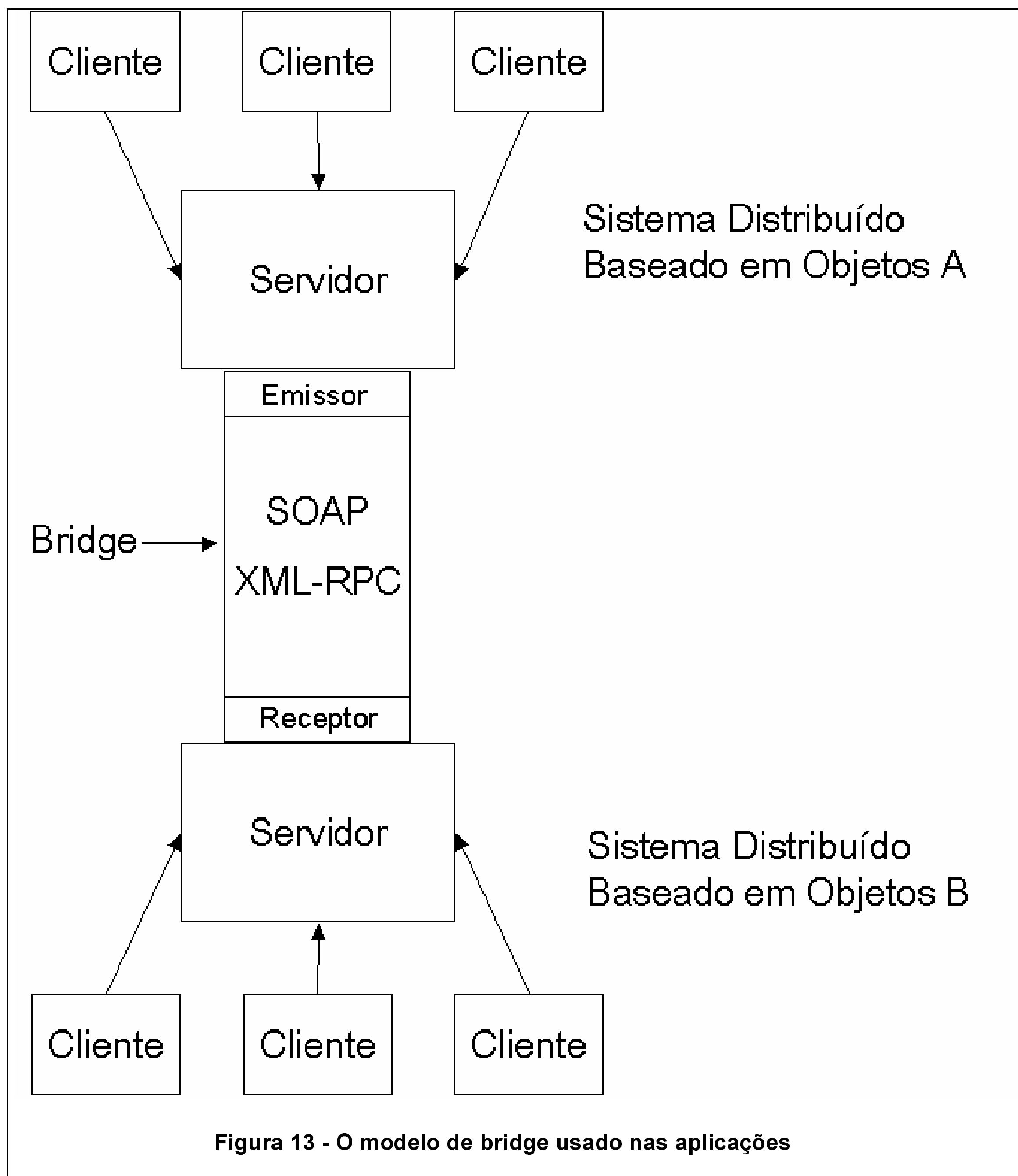

Figura 13 - O modelo de bridge usado nas aplicações

A figura acima mostra um sistema distribuído A que pode obter serviços do sistema distribuído B. Observe-se que os clientes do sistema A que desejam um serviço do sistema B, devem solicitá-lo ao servidor do sistema A. Este, por sua vez, providencia a solicitação do cliente e, ao receber alguma resposta, a repassa

para o cliente solicitante. Observe-se, ainda, que há uma *bridge* entre os dois sistemas que possui, em cada extremidade, um objeto (Emissor de um lado e Receptor do outro). Do lado cliente, o Emissor, atuando como um *proxy*, empacota e envia uma chamada remota, quando for de interesse de um cliente do sistema. Do lado servidor, o Receptor, atuando como um servidor à parte, espera, indefinidamente pela chegada de uma chamada e a processa quando a mesma chega. Essa mensagem pode trafegar de um ponto a outro usando uma rede local ou pela internet.

Em uma chamada que use esse modelo de *bridge*, o Emissor é um artefato de software com ações e características diferentes do receptor. Se desejarmos construir uma *bridge* bi-direcional, é necessário ter os dois tipos de artefatos em cada lado. Neste caso, cada artefato é implementado de acordo com as idiossincrasias de cada ambiente onde o mesmo funcionará e não necessariamente é um software portável.

A parte central da *bridge* é composta pela infra-estrutura e pelas funcionalidades das implementações das tecnologias de Web Services. Um dos aspectos dessa solução que julgamos relevantes é a sua pluralidade. Ou seja, como a tecnologia de Web Services é utilizada apenas para fazer a chamada remota entre o objeto Emissor e o objeto Receptor, pode-se utilizar qualquer uma das tecnologias de Web Services existentes, desde que a mesma seja capaz de suportar a comunicação entre programas.

Na figura 13, acima, o propósito era tão somente representar a idéia da *bridge*, de maneira que não estão representados todos os elementos que

compõem o *middleware*, ou que fornecem suporte a ele, como sistemas operacionais, serviços de rede, segurança etc.

Usaremos esta solução para interoperar COM e CORBA de *bridge* XML.

5.3 INTEROPERABILIDADE COM/CORBA VIA SOAP

A tarefa de interoperar COM e CORBA através de SOAP fica mais simplificada se usarmos uma mesma implementação do protocolo e uma mesma linguagem de programação. Por exemplo, se implementarmos um objeto cliente COM e um objeto servidor CORBA em C++, usando o software C++ Builder como ambiente de programação para ambos os objetos e o Microsoft SOAP Toolkit como ferramenta SOAP no cliente e no servidor, a ponte entre as duas arquiteturas de objetos distribuídos pode ser implementada com menor esforço, do que se as linguagens, os pacotes de implementação e as plataformas do cliente e do servidor são distintas.

A arquitetura da *bridge* XML proposta acima, no item 5.2, independe se utilizamos ou não a mesma implementação de SOAP no Emissor e no Receptor. Para demonstrar isso, projetamos uma solução para uma situação em que as implementações SOAP do lado cliente e do lado servidor não são compatíveis. E mesmo assim, a arquitetura proposta para a *bridge* será mantida. Inicialmente vamos analisar os problemas de interoperabilidade e, em seguida, mostraremos a solução.

5.3.1 PROBLEMAS DE INTEROPERABILIDADE SOAP

Na aplicação que descreveremos a seguir, usamos, no lado cliente, o Microsoft SOAP Toolkit 2.0 SP 2 (uma implementação gratuita de SOAP), com o software C++ Builder. Do lado servidor, usamos o Apache SOAP 2.3.1 (uma implementação também gratuita de SOAP), com Java.

A Microsoft criou de tal maneira o MS Toolkit que utilizamos, que o mesmo é completamente independente da arquitetura DNA/dot NET. Isto representa uma vantagem, pois para usá-lo não é obrigatório lidar com nenhuma das outras tecnologias Microsoft. Essa implementação baseia-se em WSDL (*Web Services Description Language*). Um arquivo WSDL descreve, em formato XML e na seção *PortType*, as operações proporcionadas por um serviço. Esse arquivo pode ser (e geralmente é) gerado por uma ferramenta do pacote que implementa SOAP e que usa esse tipo de arquivo nas mensagens. Outras ferramentas e métodos de vários pacotes SOAP também empacotam, automaticamente, chamadas SOAP.

O pacote Apache SOAP é baseado no SOAP4J, uma implementação SOAP da IBM. O SOAP4J foi repassado no final de 2000 ao Apache Group e é um *open source*. Houve muitos problemas para configurar o Apache SOAP, pois o pacote tem um grande número de dependências. Muitos itens precisam ser instalados ou configurados e a documentação, afora a das API´s, é muito resumida e, em alguns itens da instalação, omissa.

Como mencionamos ao final do capítulo anterior, a especificação SOAP é complexa e deixa muitos itens abertos à interpretação. Como resultado, o envelope SOAP gerado em uma implementação pode não ser apropriadamente

compreendido por outra. Uma leitura mais atenta das mensagens do arquivo da lista de discussão SOAP, no site da W3C, mostra que não é fácil conseguir a interoperabilidade entre implementações SOAP distintas e, ainda assim, as situações não são ainda muito confiáveis. Por exemplo, um problema de compatibilidade entre o Apache SOAP e o Microsoft SOAP Toolkit é que os clientes MS não enviam informação de tipo em cada parâmetro e o servidor Apache SOAP vai rejeitar tais pedidos. Já na versão do Toolkit da Microsoft, para a ativação de um objeto cliente SOAP é necessário passar um arquivo WSDL e o servidor também precisa ter um em um diretório virtual criado para a aplicação no IIS. Assim, o esforço de interoperabilidade fica prejudicado.

Portanto, o que seria ideal, além da especificação mais precisa (o que está sendo corrigido, em parte, na versão SOAP 1.2, mas ainda não foi concluído), era conseguir uma API única para SOAP, de maneira que houvesse uma uniformidade nos processos de inicialização e nas chamadas aos serviços. Nós conseguimos relacionar onze pacotes (os mais relevantes) com implementações diferentes, para linguagens diferentes e sistemas operacionais diferentes (ver apêndice B). Há alguns pontos semânticos e sintáticos semelhantes, mas ainda não é possível simplesmente transferir a lógica do código escrito em um pacote para outro.

Novas implementações de SOAP que começam a chegar ao mercado, como o Axis, da Apache, lançado em novembro de 2002, apesar de ainda não contemplarem todos os aspectos da versão SOAP 1.2 (que ainda está sendo discutida), já proporcionam melhor interoperabilidade devido à maior precisão da nova especificação. No capítulo quatro, ao realizar uma análise mais detalhada dos problemas de compatibilidade, nós mostramos as diferenças entre as

chamadas SOAP do toolkit da Microsoft, do pacote Apache SOAP e do novo Apache Axis.

5.3.2 PROJETO DA SOLUÇÃO

Após a análise das principais implementações SOAP disponíveis, nós enumeramos pelo menos dois aspectos a serem considerados quando se deseja interoperar aplicações (baseadas em arquiteturas distribuídas ou não) com implementações distintas do protocolo SOAP: o uso ou não de um documento WSDL e o formato da implementação das mensagens.

Quanto ao uso de um documento WSDL com a implementação SOAP, pode-se dispor de dois tipos de operações: mensagens orientadas a documentos (*document-centric*) e mensagens que fazem chamadas remotas (*RPC-oriented*).

O primeiro tipo, também chamado simplesmente de *messaging*, serve para enviar dados codificados em documentos XML, de um emissor para um receptor, sendo os mesmos responsáveis por gerar e analisar, respectivamente, esses documentos, com base em um XML-Schema conhecido. Através desse modelo de mensagem também se pode construir mensagens assíncronas.

Com relação ao segundo tipo de mensagem, através dele podemos invocar um processo remoto, passar-lhe seus argumentos e receber os dados de retorno do processamento, de maneira similar à que ocorre quando realizamos uma chamada remota usando o mecanismo padrão de RPC. Diferentemente do primeiro tipo de operação, neste caso, nem o cliente nem o servidor precisam

gerar e analisar o XML das mensagens remotas, ficando a cargo da implementação SOAP fazer as traduções dos tipos nativos de dados (das linguagens usadas nas aplicações cliente e servidor) para o formato padrão do protocolo e vice-versa.

Ambos os tipos de operações podem ser usados no modelo de *bridge* que estamos propondo.

Quanto ao formato da mensagem, ela varia em alguns aspectos. Algumas variações são decorrentes do uso ou não de WSDL. Outras são decorrentes das falhas da especificação SOAP, já apontadas no capítulo quatro desta dissertação.

Resumindo o que foi detalhado no capítulo quatro, as principais diferenças entre as chamadas SOAP geradas pelo Toolkit da Microsoft e pelo Apache SOAP são:

- O Apache SOAP não faz uso de WSDL na mensagem, apenas para disponibilizar um serviço. O Toolkit da Microsoft sempre requer o uso de um arquivo WSDL;
- No Toolkit Microsoft, o nome do método chamado aparece tanto no cabeçalho HTTP (SOAPAction), como na parte *BODY* do *Envelope*, além da descrição do arquivo WSDL. No Apache SOAP, a entrada SOAPAction não é gerada;
- No Toolkit Microsoft, os parâmetros dos métodos são nomeados, e no Apache SOAP eles são posicionais;
- A informação de tipo dos parâmetros está presente nas chamadas do Toolkit Microsoft e ausente nas chamadas do Apache SOAP;

- O Apache SOAP não é capaz de processar o *Byte Order Mark* (BOM) do tipo Unicode (UTF-16, mais precisamente) enviado pelo Microsoft SOAP Toolkit. Assim as mensagens devem trafegar em formato de oito bits, mas sem enviar qualquer informação de BOM.

Resta-nos, então, mostrar que estratégia pode ser usada para contornar os problemas de interoperabilidade entre as implementações de SOAP, de maneira a tornar operacional e viável a *bridge* proposta em 5.2 com o uso de SOAP. Essa solução é trabalhosa, porque em vez de usar os mecanismos de RPC de SOAP, usa o formato *messaging*. Isto quer dizer que todo o processamento de montar a mensagem no cliente e analisá-la no servidor será feito codificando todos os detalhes necessários. Como benefício, essa solução para esse problema de interoperabilidade funciona não só para as implementações SOAP acima mencionadas, mas também para quaisquer outras que sigam a especificação oficial.

A seguir, nós vamos exibir os componentes da arquitetura da *bridge* XML e os seus relacionamentos. Além desses componentes, a fim de proporcionar uma compreensão geral do processo, mostraremos também outros objetos que estão presentes no exemplo e as GUIs.

Para mostrar a aplicação da tecnologia de *bridges* com SOAP, implementamos um servidor CORBA, Calculadora, com seis métodos (*soma, sub, mult, div, retornaTotal, mudaTotal*), além do construtor da classe.

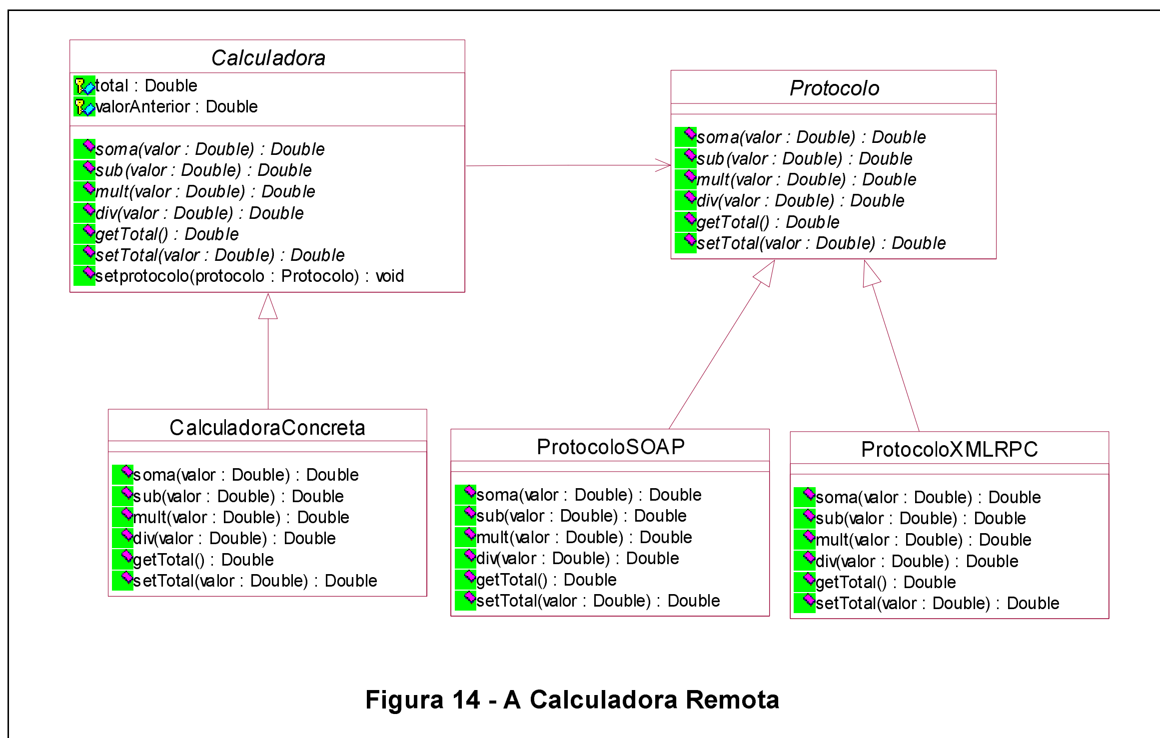

Figura 14 - A Calculadora Remota

Observe-se na figura 14, acima, que a calculadora, a fim de estar preparada para ser implementada em mais de um protocolo, delega o envio da mensagem a uma estrutura hierárquica paralela. Com isso, a injeção da dependência desejada pode ser configurada através do método *setProtocolo*.

Portanto, se outro protocolo além dos dois previstos precisar ser usado no futuro, a solução está extensível, bastando que outra classe concreta de protocolo seja implementada e que a nova dependência seja injetada.

Um cliente COM deverá usar essa calculadora remota. O objeto cliente COM foi implementado em C++, mas poderia também ter sido em VB, C# ou qualquer outra linguagem que suporte COM.

Para contornar os problemas de compatibilidade, portanto, descreveremos como fazer uma chamada remota baseada em *messaging*, a partir de aplicação C++, gerando o próprio XML necessário para a chamada SOAP.

Do lado cliente, a configuração de hardware e software inclui um PC com Windows XP, tendo o software sido implementado com o C++ Builder 5.0 e o Microsoft SOAP Toolkit 2.0, *service pack* 2. Do lado servidor, a configuração inclui um PC com Conectiva Linux 8, tendo o software sido implementado com o Java 1.3.1 e o Apache SOAP 2.3.1.

A arquitetura dos componentes da aplicação pode ser observada no diagrama abaixo:

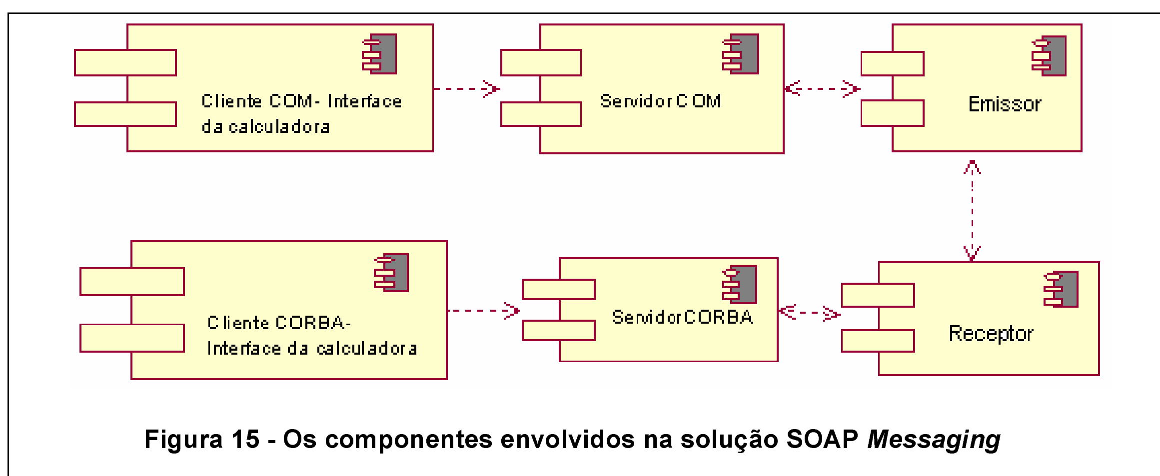

Figura 15 - Os componentes envolvidos na solução SOAP *Messaging*

A visão de distribuição (deployment) dos componentes está mostrada logo a seguir:

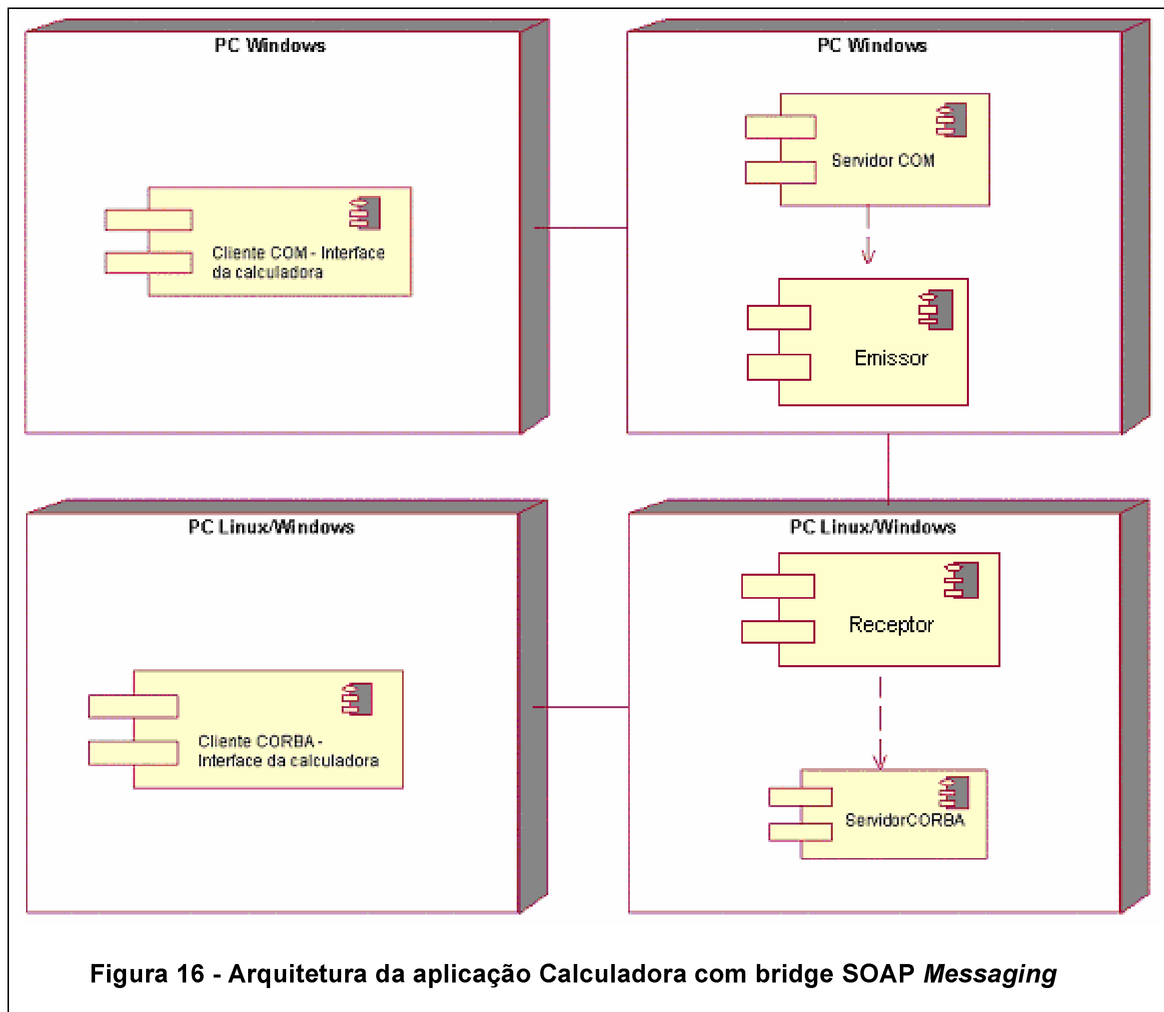

Figura 16 - Arquitetura da aplicação Calculadora com bridge SOAP *Messaging*

Como mostram as figuras acima, na arquitetura dessa aplicação, seguindo o modelo de arquitetura proposto para a *bridge* em 5.2, criamos um servidor COM, no lado cliente da aplicação. Assim que o servidor inicia, ele cria o objeto Emissor, seguindo o padrão *Singletown* **[Gamma at al, 1995]**, tendo em vista que apenas uma única instância desse servidor deve existir em um dado instante.

O objeto Emissor funciona como um *proxy* e será responsável por:

- receber as requisições do servidor COM (que por sua vez as recebe de seus clientes);

- montar a mensagem SOAP, se encarregando de todos os detalhes da mesma;
- enviar a mensagem de maneira síncrona, via *messaging*, seguindo o padrão de interação *request-response* **[Orfali, Harkey & Edwards, 1999]**;
- ao receber resultados, analisar a resposta e passar os valores para o servidor COM.

As comunicações entre o objeto Emissor e o servidor COM, a fim de devolver os valores das respostas das chamadas, podem ser realizadas de duas formas: ou através de *callbacks*, no caso de uma chamada assíncrona, ou através do mecanismo comum de retorno de chamadas a subprogramas, tendo em vista que o Emissor é criado no mesmo espaço de endereçamento do servidor COM. Na solução que agora detalhamos, optamos pela segunda forma.

No lado servidor da aplicação, há um servidor CORBA que fornece os serviços da Calculadora para seus clientes e para os clientes do servidor COM.

O objeto Receptor tem as seguintes responsabilidades:

- receber as chamadas SOAP;
- analisar o XML, obter o nome do método a ser executado e os seus parâmetros;
- atuar como cliente CORBA realizando a chamada requerida no servidor CORBA;
- receber o resultado da operação efetuada pelo servidor CORBA, montar o XML da resposta SOAP e enviá-la ao Emissor;

Pela arquitetura da *bridge*, assim como o servidor COM, o servidor CORBA também é único. Essa limitação não é exclusiva da solução proposta, mas é compartilhada por todas as *bridges* baseadas em componentes (ver capítulo dois). Assim, livramos os clientes de ambos os servidores desse *overhead* de código, tornando mais simples os clientes de cada lado (tanto os clientes do servidor COM, como os clientes do servidor CORBA).

5.3.3 FLUXO DAS MENSAGENS

Vamos mostrar agora, como é a seqüência das chamadas que acontecem entre um cliente COM e o servidor CORBA, ao se usar a *bridge* implementada de acordo com a arquitetura proposta em 5.2 e usando SOAP. O processo pode ser melhor compreendido ao se acompanhar as chamadas, durante toda a explicação, nas figuras 16 e 17.

Ao iniciar a aplicação Calculadora, o usuário (cliente COM) submete ao servidor COM uma operação. O Servidor COM, por sua vez, chama essa operação no Emissor. Para efeito de descrição da operação, suponhamos que foi submetida a operação *soma*. O perfil do método *soma* no Emissor deve respeitar o perfil do método *soma*, segundo a hierarquia da classe Calculadora, presente no servidor CORBA. A seguir, apresentamos um diagrama de seqüência das chamadas realizadas:

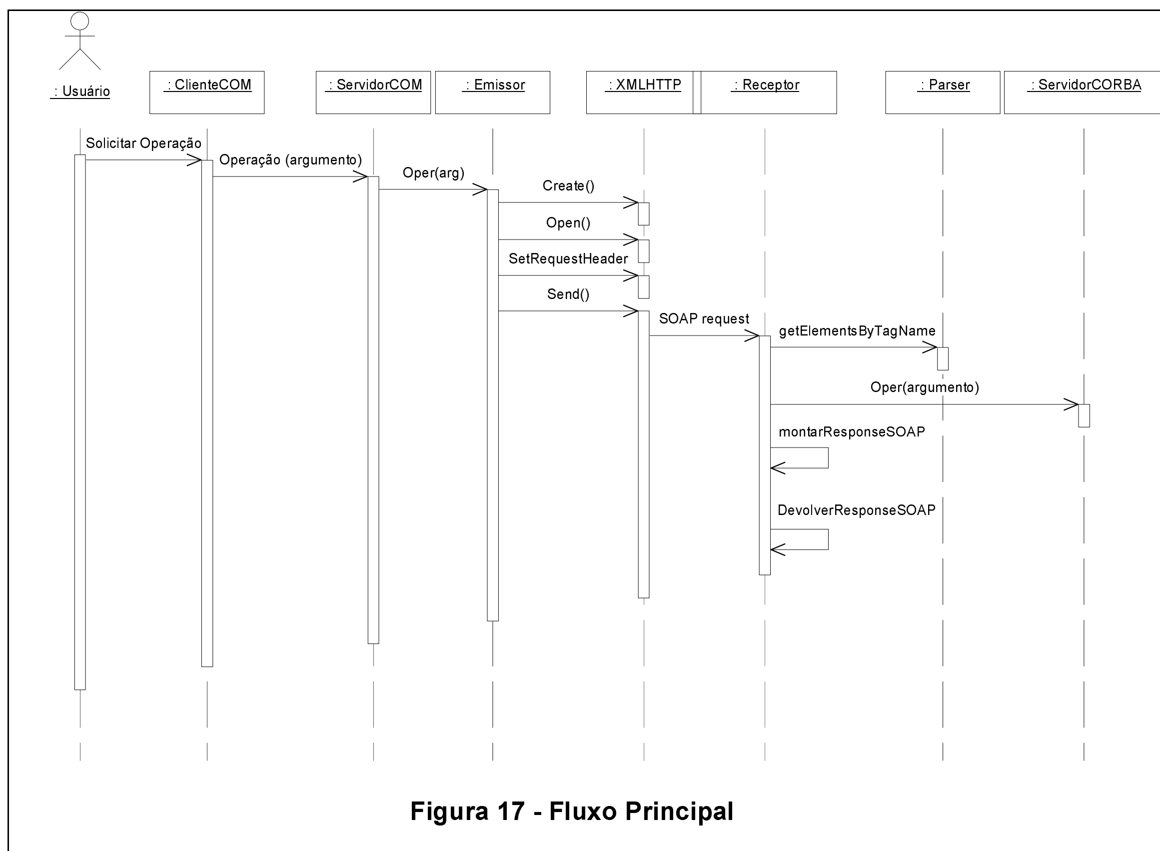

Figura 17 - Fluxo Principal

No objeto Emissor, o método *soma* constrói o XML da mensagem e chama o método *empacota* que acrescenta o cabeçalho HTTP à mensagem e a envia. O método empacota pode acrescentar, dependendo da mensagem, algum cabeçalho SOAP. O código para gerar o XML da mensagem está mostrado abaixo. Observe que a mensagem SOAP montada é similar àquela descrita anteriormente no capítulo quatro, quando nós analisamos a estrutura de uma mensagem SOAP gerada pelo Apache SOAP.

```
chamada  = '<?xml version="1.1"?>';
chamada += '<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope">';
chamada += "<SOAP-ENV:Body>";
chamada += "<m:soma xmlns:m=\"http://www.augustopedroza.com\">";
chamada += "<parameters>";
chamada += "<param0>" + valor + "</param0>";
chamada += "</parameters>";
chamada += "</m:soma>";
chamada += "</SOAP-ENV:Body>";
chamada += "</SOAP-ENV:Envelope>";
```

O código C++ acima é trivial, como se observa, constituindo-se apenas de concatenação de strings que formam a mensagem. No método *empacota*, o objeto XMLHTTP, da *Type Libray* do MS SOAP Toolkit é iniciado e uma conexão síncrona (representada no código abaixo pela opção *false*, como parâmetro de *open*) é aberta, como está descrito no código logo a seguir:

```
XMLHTTP *httpObj = new XMLHTTP();
HttpObj->Open("POST", "http://pedroza02/soap/servlet/ReceiveSoap",false,"","");
HttpObj->setRequestHeader("SOAPAction", sSOAPAction);
HttpObj->setRequestHeader("Content-Type","text/xml");
HttpObj->Send (soapRequest);
```

No código C++ acima, os cabeçalhos HTTP são configurados com o método de instância *setRequestHeader* da classe XMLHTTP e o método de instância *send* da mesma classe envia uma chamada HTTP Post para um servidor SOAP (Receptor). Receptor faz parte da *bridge* e, em verdade, é implementado

como um cliente CORBA que recebe a chamada SOAP enviada por Emissor e a processa, como está mostrado no código abaixo.

```
try {
   BufferedReader recebe = request.getReader();
   StringBuffer documento = new StringBuffer();
   String xml;
   while ((xml = recebe.readLine()) != null ) {
      documento.append(xml);
   }
   recebe.close();
  mensagemSoap = documento.toString().trim();
} catch (IOException e) {
...
}
```

No código Java acima, a variável *recebe* obtém o *stream* com o XML da chamada SOAP. Cada linha do *stream* é adicionada ao conteúdo da variável documento (um StringBuffer). Após ser acrescentado todo o XML à variável documento, a mesma é atribuída à String mensagem. A seguir, o XML é analisado em um método chamado *analisa* e são obtidos o nome do método a ser invocado no servidor CORBA e os seus parâmetros.

Observe-se que essa solução não trata problemas de chamadas concorrentes porque prevê, de maneira simplificada mas eficiente, a comunicação de um único objeto de um lado da *bridge* (servidor COM) com um único objeto do

outro lado da *bridge* (servidor CORBA). Isso faz com que, na possibilidade de, em algum caso específico, haver simultaneidade de chamadas, recorra-se a mecanismos específicos de sistemas distribuídos baseados em objetos, como o padrão Lock **[Mowbray, 1997]**, que diz como gerenciar o acesso concorrente a uma operação em um objeto distribuído, quando essa operação só pode ser acessada por um único cliente por vez.

Mostraremos agora, como é implementada a chamada realizada entre o Receptor e o servidor CORBA. Como Receptor e o servidor CORBA são codificados em Java, nós tínhamos a opção de usar RMI-IIOP ou Java IDL. Optamos por este último.

```
Try {
  ORB orb = ORB.init("iiop://Pedroza02:9000", null);
  Org.omg.CORBA.Object objRef = orb.resolve_inicial_references("CorbaPedroza02");
  NamingContext ncRef = NamingContextHelper.narrow(objRef);
  NameComponent nc = new NameComponent("Calculadora", " ");
  NameComponent path[] = nc;
  // calcref é do tipo CalculadoraInterface e está definido como variável de instância
  calcRef = CalculadoraInterfaceHelper.narrow(ncRef.resolve(Path));
}
catch (Exception e) { ... }
```

No momento em que o servidor SOAP for iniciado ele deve executar os comandos acima. O código inicia criando e iniciando o ORB. Depois obtém uma referência para o serviço de nome do Corba (COSNaming – Corba Object service)

e finalmente obtém a referência do objeto remoto calculadora. O código seguinte mostra a invocação de *soma*.

```
.....
  Double total = calcRef.soma(valor);
.....
```

A partir de então, o Receptor está pronto para pegar o valor total, convertê-lo em String, montar o XML da resposta SOAP e enviá-la ao Emissor.

Como já comentamos, a chamada feita pelo Emissor, no lado COM da *bridge*, ao servidor SOAP, no lado CORBA da *bridge*, é feita de maneira síncrona. Sendo uma chamada do tipo *request-response*, olhemos agora, como o Emissor recebe a resposta da chamada.

```
try {
AnsiString resposta = URLDecode(httpObj.ResponseText);
XMLDOM *soapXML = new XMLDOM();
SoapXML->async = false;
SoapXML->loadXML(Resposta);
total = dyanamic_cast<double>(SoapXML->selectSingleNode("return"));
} catch (EConvertError &e ) {
 total = 0; throw;
} catch (Exception &e) {
 total = -1;  throw;
}
```

Quando o servidor CORBA responder à chamada feita pelo Receptor (que é um cliente CORBA), ele (o receptor) a repassará para o Emissor, que é o cliente

da chamada síncrona SOAP. Especificamente, o XML da resposta é devolvido ao método *analisa* de Emissor. O XML da resposta será armazenado em uma variável string C++. Processa-se, então o XML recebido e atribui-se o resultado à variável *total*, como mostra o código C++ acima.

O valor de *total* pode ser enviado para o servidor COM de duas formas: através de um simples "return" no método do Emissor que foi chamado pelo servidor COM, ou através de uma mensagem *callback*, pois o Emissor tem um ponteiro do objeto que o criou. O servidor COM, finalmente, repassa o valor de resposta ao seu cliente que solicitou a operação *soma*.

5.3.4 GUI DO CLIENTE COM

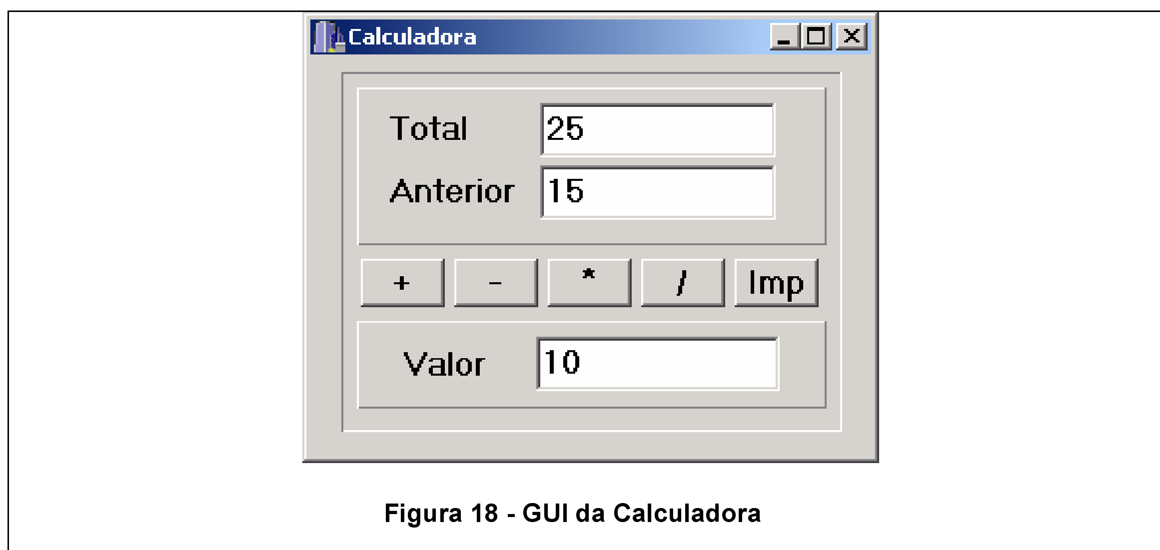

Figura 18 - GUI da Calculadora

A figura acima mostra a interface do usuário COM após duas operações de soma. Ao ser iniciada a calculadora mostra zero no campo Valor e os valores atuais de total e anterior. As operações subsequentes mudam os valores desses

campos. Para saber qual o valor atual deles, chama-se o método *retornaTotal* através do botão Imp.

5.4 INTEROPERABILIDADE COM/CORBA VIA XML-RPC

As principais diferenças entre a implementação de uma *bridge* XML, usando SOAP, e uma bridge XML, usando XML-RPC, são derivadas das diferentes características dos dois protocolos.

De um modo geral, as diferenças entre os protocolos são:

- XML-RPC somente possui a capacidade de enviar mensagens RPC, não sendo possível utilizar o estilo *messaging,* como no SOAP;
- Ao contrário de SOAP, não há suporte a chamadas assíncronas em XML-RPC (embora as mesmas possam ser feitas acrescentando-se várias linhas de código próprio, observando todos os detalhes que uma chamada desse tipo exige e contornando todos os problemas, sem o suporte das API´s das implementações XML-RPC);
- o XML das mensagens SOAP baseia-se em *namespaces* e *schemas*, enquanto o XML das mensagens XML-RPC baseia-se em DTDs. Por isso SOAP é claramente mais extensível;
- as mensagens XML-RPC só podem conter letras, números e mais quatro caracteres especiais (_ . / ;). Pode ser útil para uma grande parte das aplicações em geral, mas é ineficiente em alguns casos, como quando é necessário enviar um objeto como argumento de uma

chamada, por exemplo (Há uma nova refrência XML-RPC para dados binários, mas ainda não há implementações disponíveis);

- uma mensagem XML-RPC somente tem um destino, não possuindo a capacidade de ser processada em vários pontos de uma trajetória, como as mensagens de extensibilidade horizontal **[Graham, 2002]** de SOAP (chamadas de *intermediaries*) ;

Apenas um subconjunto das funcionalidades de SOAP está disponível em XML-RPC. Esse protocolo é fácil de usar e a sua simplicidade talvez seja sua maior limitação. Uma das razões que nos motiva a apresentar a arquitetura da *bridge* XML usando o protocolo XML-RPC é justamente mostrar que, mesmo com um protocolo mais simples, a solução funciona. De maneira mais limitada, mas funciona.

5.4.1 PROJETO DA SOLUÇÃO

A arquitetura da aplicação Calculadora usando a *bridge* XML-RPC é igual à arquitetura da aplicação Calculadora que usa a *bridge* SOAP. Ou seja, as arquiteturas são iguais porque as diferenças entre as aplicações resumem-se a *bridge*.

No lado cliente da *bridge* XML, objeto Emissor, utilizamos as API´s C++ da biblioteca XML-RPC-C, implementada pela SourceForge. No objeto Receptor da *bridge* XML, utilizamos as API´s do pacote helma.xmlrpc, implementado pela Helma.

Ambas as API´s tem que ser totalmente compatíveis, em relação à versão do protocolo, para uma chamada XML-RPC poder funcionar.

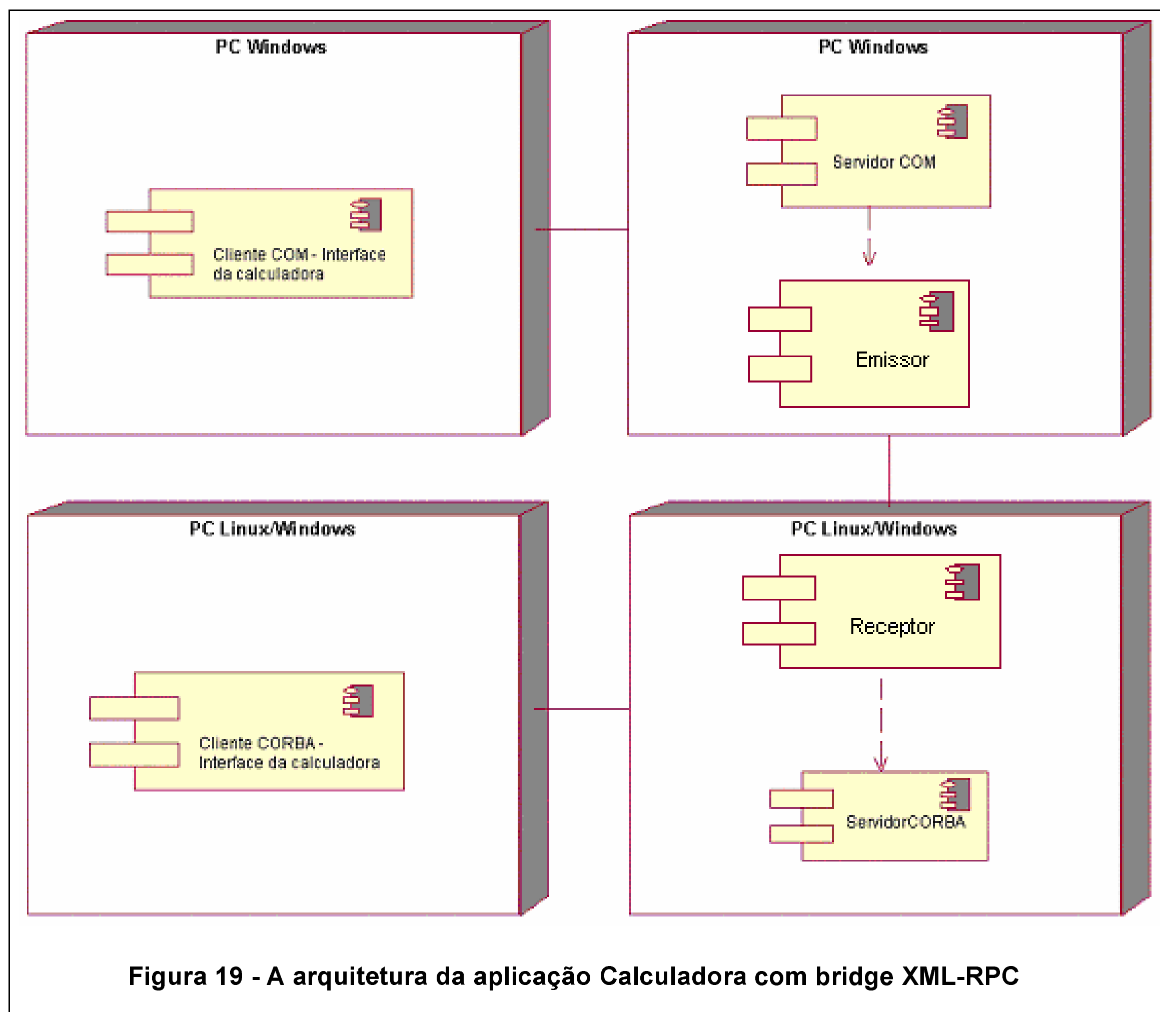

Figura 19 - A arquitetura da aplicação Calculadora com bridge XML-RPC

A figura acima mostra a arquitetura da aplicação, que segue o modelo de arquitetura proposto para a *bridge* XML. No lado cliente da aplicação há um servidor COM. Assim que o servidor inicia, ele cria o objeto Emissor, seguindo o padrão *Singleton* **[Gamma at al, 1995]**, tendo em vista que apenas uma única instância desse servidor deve existir em um dado instante.

O objeto Emissor funciona como um *proxy* e será responsável por:

- receber as requisições do servidor COM (que por sua vez as recebe de seus clientes);
- montar os argumentos da chamada XML-RPC em um vetor;
- enviar a mensagem de maneira síncrona;
- ao receber resultados, analisar a resposta e passar os valores para o servidor COM.

As comunicações entre o objeto Emissor e o servidor COM, a fim de devolver os valores das respostas das chamadas, podem ser realizadas de duas formas: ou através de *callbacks*, no caso de uma chamada assíncrona (entre o servidor COM e o Emissor), ou através do mecanismo comum de retorno de chamadas a subprogramas, tendo em vista que o Emissor é criado no mesmo espaço de endereçamento do servidor COM. Na solução que agora detalhamos, optamos pela segunda forma.

No lado servidor da aplicação, há um servidor CORBA que fornece os serviços da Calculadora para seus clientes e para os clientes do servidor COM.

O objeto Receptor tem as seguintes responsabilidades:

- receber as chamadas XML-RPC;
- executar o método passando-lhe seus parâmetros;
- atuar como cliente CORBA realizando a chamada requerida no servidor CORBA;
- receber o resultado da operação efetuada pelo servidor CORBA e devolvê-lo para o Emissor;

5.4.2 FLUXO DAS MENSAGENS

Vamos mostrar agora, como é a seqüência das chamadas que acontecem entre um cliente COM e o servidor CORBA, ao se usar a *bridge* implementada de acordo com a arquitetura proposta em 5.2 e usando SOAP. O processo pode ser mais bem compreendido ao se acompanhar as chamadas, durante toda a explicação, nas figuras 19 e 20.

Ao iniciar a aplicação Calculadora, o usuário (cliente COM) submete ao servidor COM uma operação. O Servidor COM, por sua vez, chama essa operação no Emissor. Para efeito de descrição da operação, suponhamos que foi submetida a operação *soma*. O perfil do método *soma* no Emissor deve respeitar o perfil do método *soma*, segundo a hierarquia da classe Calculadora, presente no servidor CORBA. A seguir, apresentamos um diagrama de seqüência das chamadas realizadas:

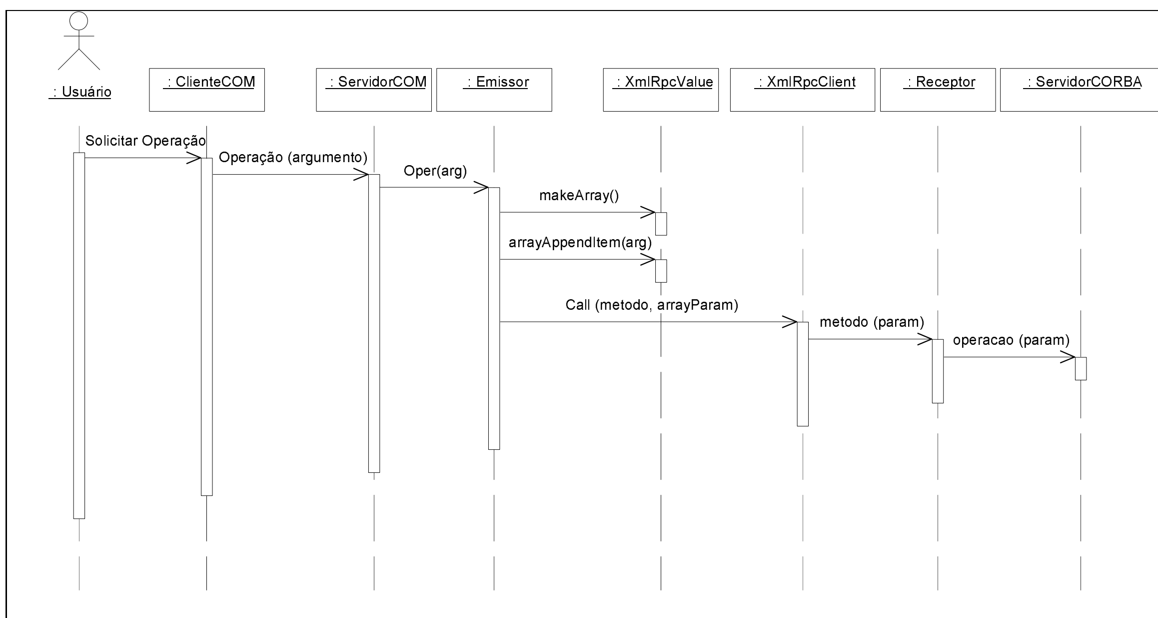

Figura 20 - Fluxo Principal

No objeto Emissor, o método *soma* monta os parâmetros da mensagem com a ajuda de *XmlRpcValue* e a envia para o Receptor, através de *XmlRpcClient*. Logo abaixo, estão os detalhes do código envolvido nessa operação.

```
......
 // obtém o ponteiro para a calculadora (essa instrução fica no construtor de Emissor)
XmlRpcClient calculadora = new XmlRpcClient (http://Pedroza02/xmlrpc/Receptor);
.....
// essas instruções ficam no método soma de emissor
XmlRpcValue param_array = XmlRpcValue::makeArray(); // array de parâmetros
param_array.arrayAppendItem(valor); // parâmetro da chamada
XmlRpcValue apps = calculadora.call("Calculadora.soma", param_array); // faz a chamada remota
XmlRpcValue total = dynamic_cast<double>(apps.arrayGetItem(0)); // obtém o valor de total devolvido
......
```

No código acima, do objeto Emissor, estão mostrados os comandos para obter um ponteiro para o objeto, montar o array de parâmetros, fazer a chamada ao Receptor e, após receber a resposta, retirar do vetor de resultados o valor de *total*.

O código do Receptor é trivial, porque ele apenas repassa, na qualidade de cliente CORBA, a requisição recebida de Emissor para o método devido no servidor CORBA.

5.4.3 GUI DO CLIENTE COM

A GUI do cliente COM é rigorosamente a mesma mostrada em 5.3.3 e suporta os mesmos casos de uso.

Analisemos, a seguir, a arquitetura de outra aplicação, uma Agenda pessoal, que também usa esse modelo de *bridge* XML.

5.5 ARQUITETURA DA BRIDGE XML PARA UMA AGENDA PESSOAL

A fim de demonstrar a versatilidade da *bridge* XML, mostraremos como essa solução pode ser usada em outro caso, no qual nós integramos um cliente ASP (interface web), com um servidor JSP.

Verifica-se como este exemplo é útil e oportuno, quando se leva em conta a crescente necessidade de integração de sistemas, muitos deles com tecnologias muito heterogêneas. Neste aspecto, o modelo de *bridge* XML também mostra sua utilidade.

Olhando o diagrama 5.9, vemos que a aplicação ASP, um servidor ActiveX (COM), roda no servidor IIS 5.0 e permite a um usuário, através de uma interface Web, pesquisar informações em uma agenda pessoal, além de proporcionar operações comuns de alteração, inclusão e exclusão de informações em um banco de dados.

A aplicação JSP roda em um servlet container (o Tomcat 4.0.3) e é cliente do objeto Dadosbean. Este, por sua vez, apresenta as funcionalidades necessárias para administrar um SGBD MySQL.

A arquitetura está mostrada logo abaixo, através de um diagrama de blocos:

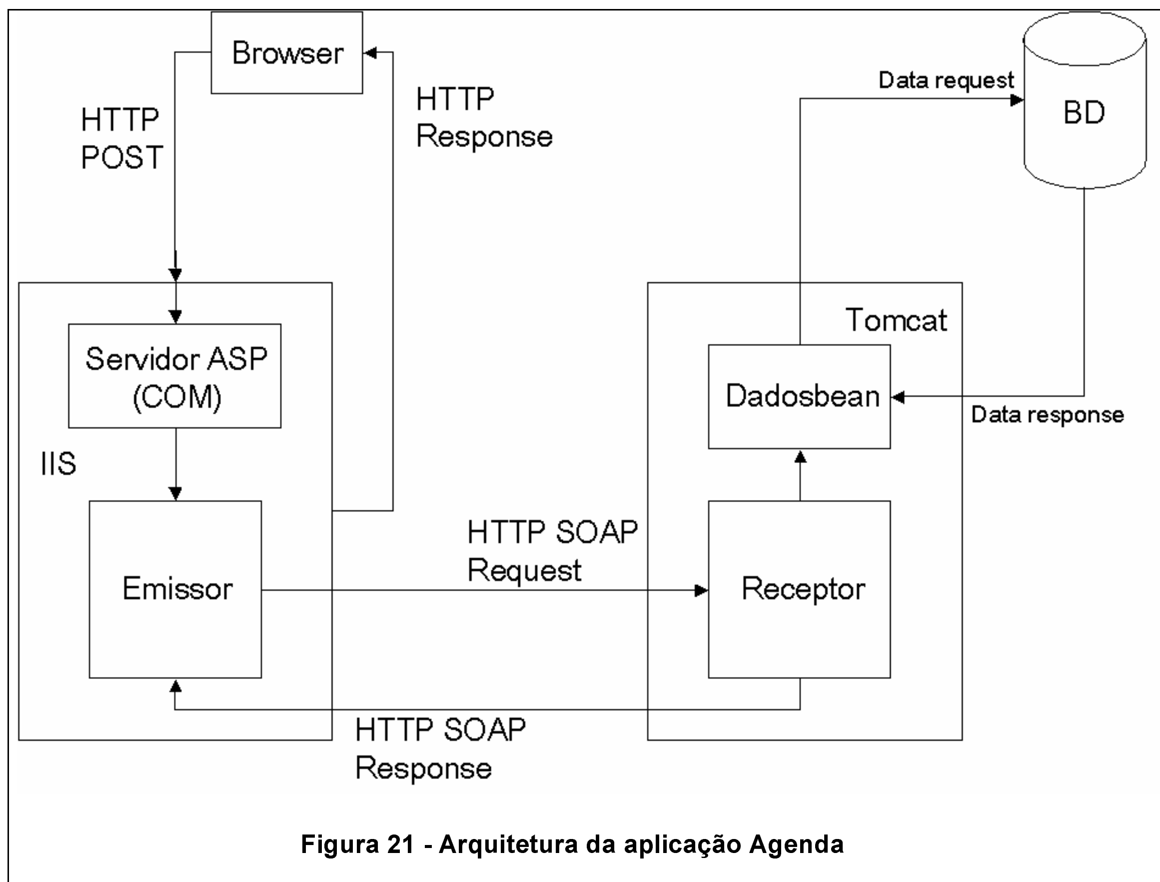

Figura 21 - Arquitetura da aplicação Agenda

Podemos identificar a *bridge* XML nesta arquitetura, ao observarmos o par Emissor-Receptor na parte de baixo da figura, ligados por chamadas SOAP.

Como na aplicação Calculadora, o Emissor é criado pelo Servidor ASP, o qual submete-lhe as requisições. As requisições do servidor ASP foram geradas por um usuário a partir de uma interface web.

Como exemplo, suponhamos que foi submetida uma operação de consulta por nome, como mostra a figura abaixo.

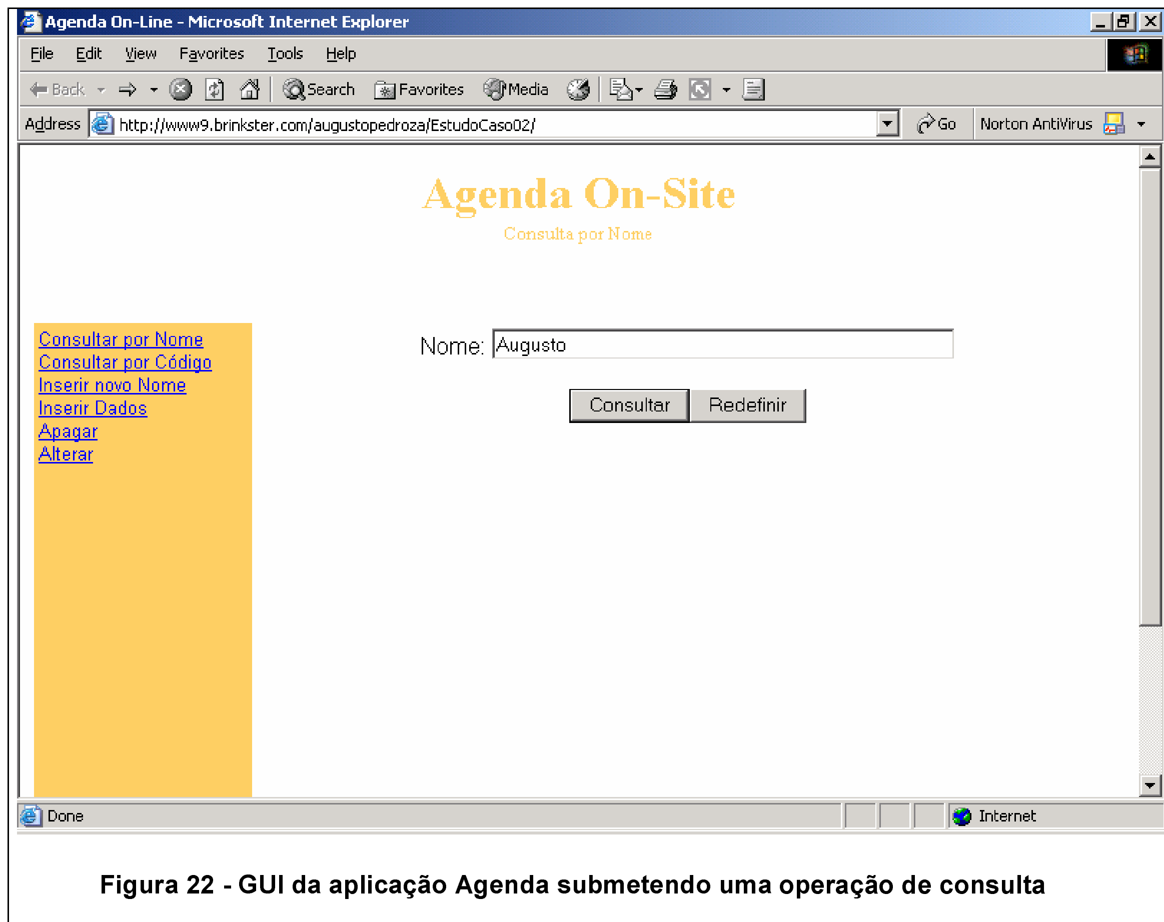

Figura 22 - GUI da aplicação Agenda submetendo uma operação de consulta

Como a chamada SOAP nesta aplicação segue o estilo *messaging* usado na calculadora SOAP, o Emissor recebe o pedido, gera o XML da mensagem SOAP e a envia para o Receptor.

O Receptor analisa a mensagem, pega os seus parâmetros e submete a operação a um objeto Java, Dadosbean (um javabean). Após receber os dados da consulta, monta a mensagem de resposta e envia-a ao Emissor.

O Emissor, então, repassa os dados para o servidor ASP que gera a página de resposta vista na figura abaixo.

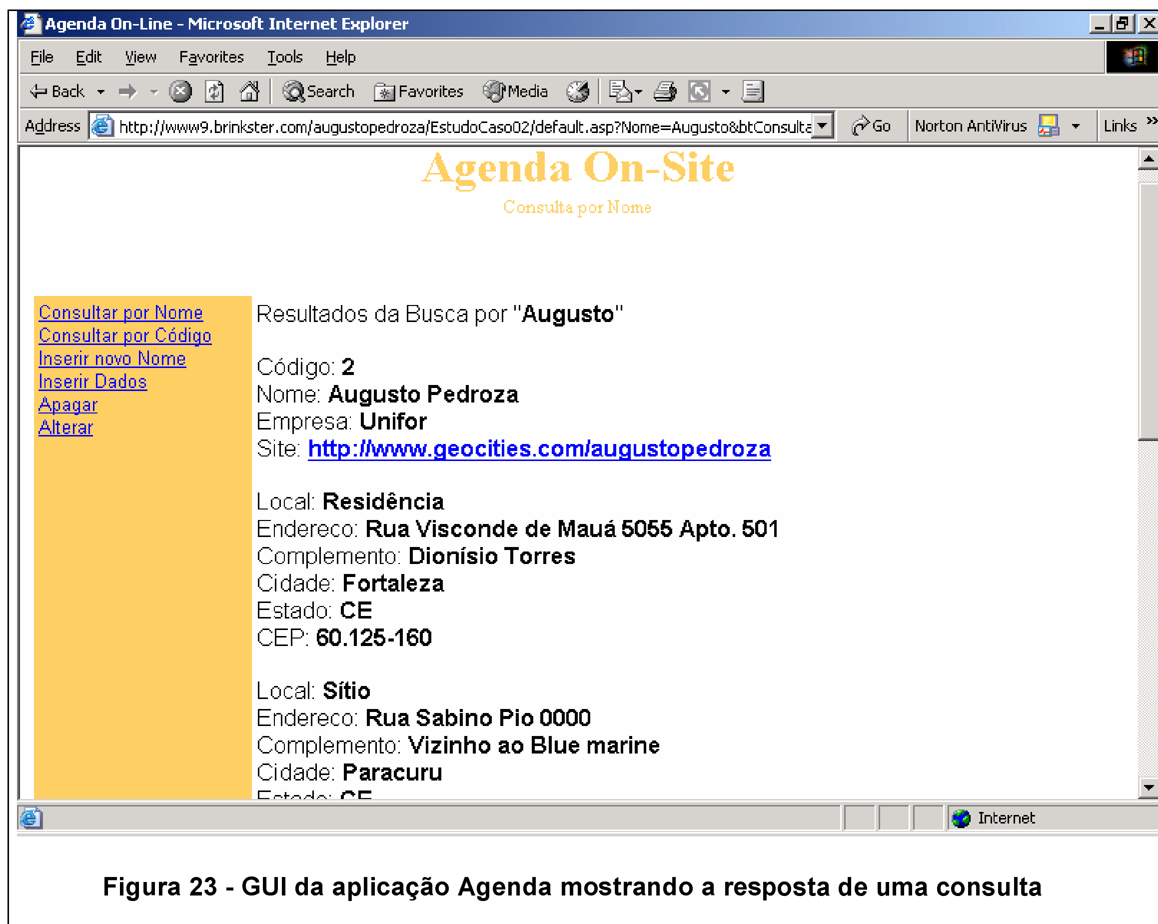

Figura 23 - GUI da aplicação Agenda mostrando a resposta de uma consulta

Mas nem todas as vezes que o usuário interagir na interface web a *bridge* XML entra em ação. Isto porque algumas vezes a interação não solicitará um serviço de banco de dados.

Por exemplo, o usuário pode estar querendo abrir a interface de inclusão de número de telefone. Nesse caso, o próprio servidor ASP trata a requisição e gera a página abaixo:

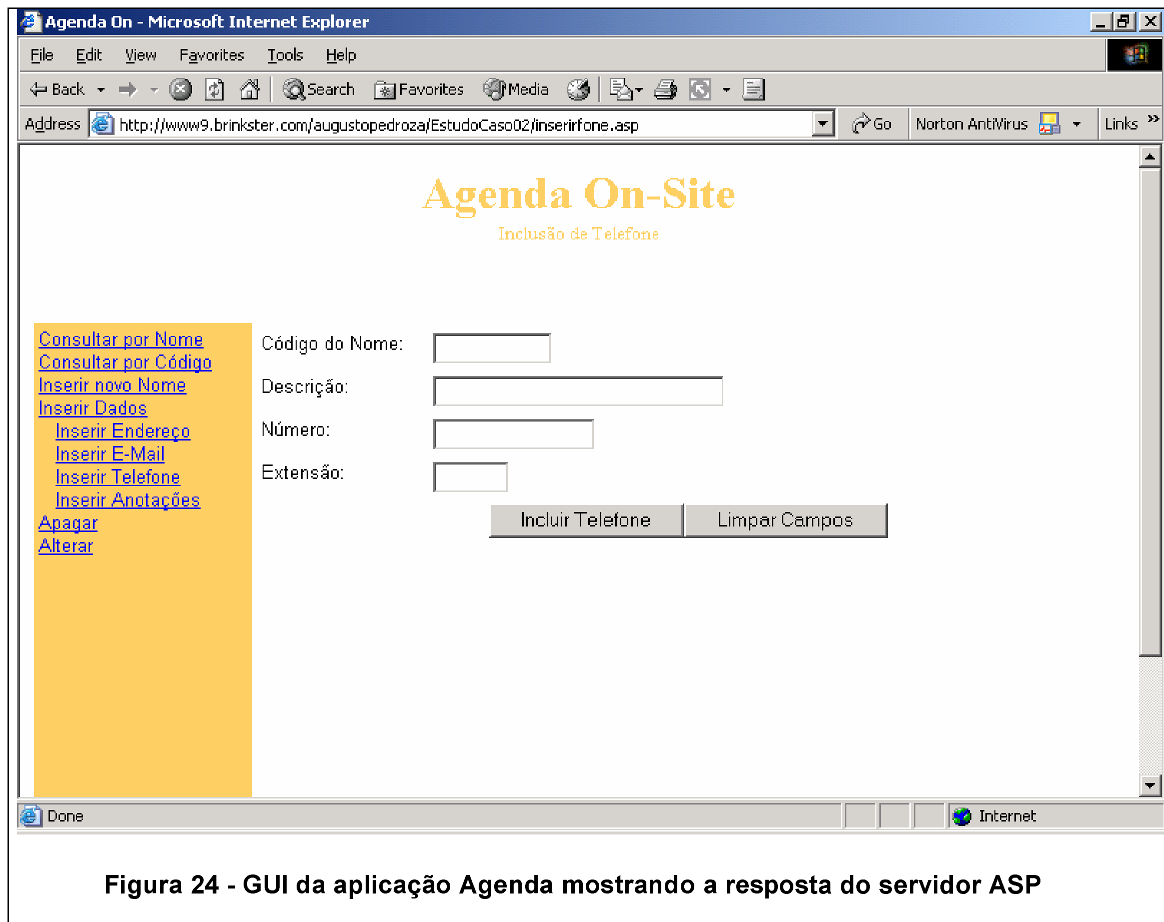

Figura 24 - GUI da aplicação Agenda mostrando a resposta do servidor ASP

Observemos agora, no diagrama abaixo, que poucas mudanças na arquitetura seriam necessárias para colocar o Receptor interagindo com um servidor CORBA. Vejamos:

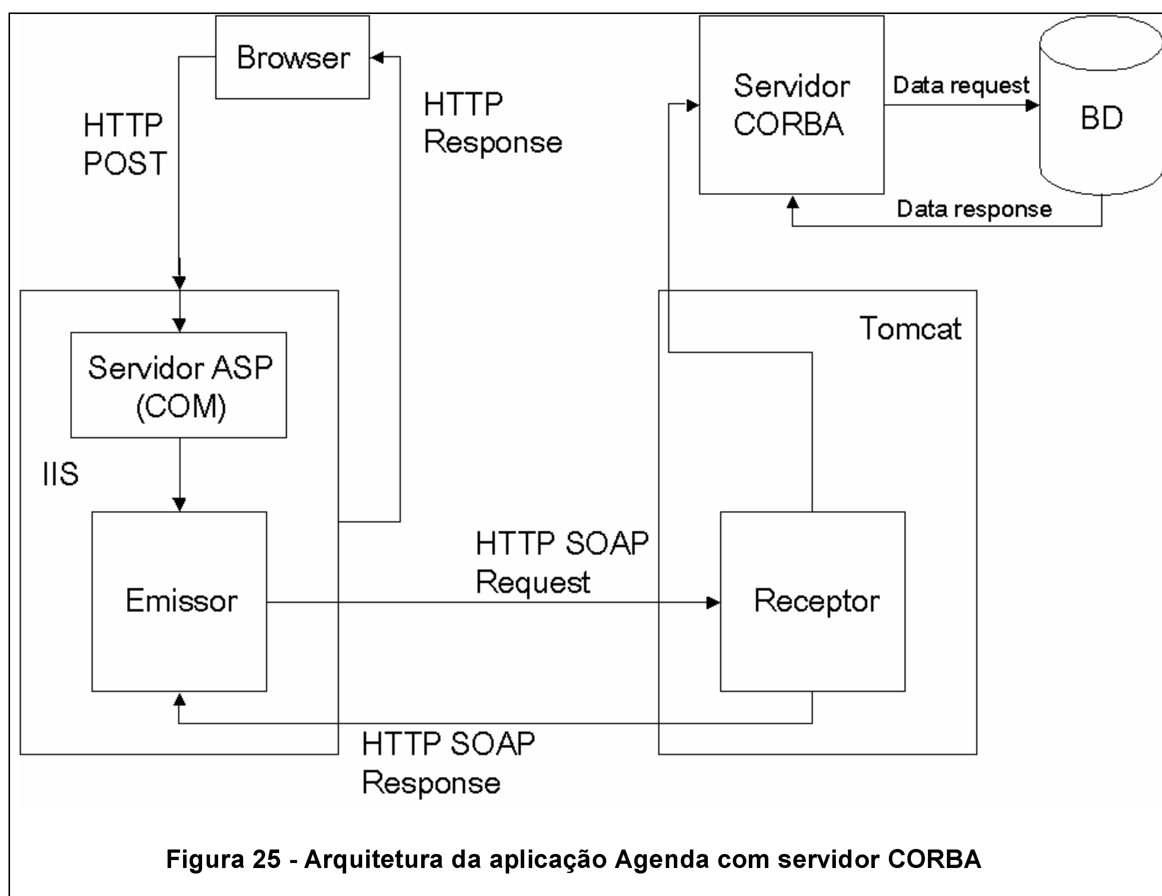

Figura 25 - Arquitetura da aplicação Agenda com servidor CORBA

Neste diagrama, figura 25, o servidor CORBA substitui Dadosbean, da figura 21, oferecendo serviços relacionados à persistência. A outra alteração que necessita ser feita é tornar o Receptor um cliente do servidor CORBA.

5.6 *BRIDGES* CONVENCIONAIS VERSUS *BRIDGES* XML

Qualquer um dos modelos de *bridge* mostrado nesta dissertação torna a aplicação distribuída mais lenta. É o preço que os sistemas pagam pela interoperabilidade. No entanto, ao contrário das *bridges* convencionais (aquelas baseadas em um único componente e que mapeiam uma chamada de um sistema

em uma chamada de outro), as *bridges* baseadas em XML são mais ortogonais do que as primeiras.

Constata-se isso ao se observar as restrições de localização das primeiras (praticamente apenas na plataforma Windows, que é a única que dá suporte aos três tipos de objetos COM), bem como as suas limitações de mapeamento, que podem não atender igualmente a todos os ORBs CORBA (tendo em vista haver muita incompatibilidade entre ORBs de fabricantes distintos), muitas vezes necessitando de adaptações quando se deseja usar a mesma *brigde* em situações diferentes.

Já as *bridges* baseadas em XML, por usarem recursos totalmente, ou em grande parte, padronizados (ou seja, XML, texto em formato UTF, HTTP), apresentam-se para nós como uma solução melhorada, porque é uma tecnologia mais simples e pode ser usada para a integração de sistemas distribuídos.

O *overhead* de processamento dos dados das chamadas em formato XML não aparece como um destaque que vá impactar bastante o tempo de resposta da solução, porque o principal gargalo de performance dos sistemas, aonde a solução venha a ser empregada, é o tempo gasto na comunicação entre dois pontos de uma rede, o que releva a segundo plano esse *overhead* de processamento.

Ainda sob a perspectiva de performance, o fato do Emissor da *bridge* funcionar como um *proxy*, obriga-o a ter os mesmos métodos do servidor do outro sistema distribuído (o servidor CORBA das nossas aplicações), facilitando o modo de devolver a resposta. Assim, o *thread* do servidor COM que fez a chamada ao Emissor pode ter duas opções:

1. ficar bloqueado, esperando a resposta (que é mais fácil de implementar), ou
2. fazer uma chamada assíncrona, prosseguir a sua linha de execução, fazendo com que o Emissor se encarregue de realizar a chamada *callback*.

No início da pesquisa, nós montamos o Emissor como sendo composto de um único método que receberia chamadas para todos os serviços desejados pelos clientes COM. Quando optamos pelo Emissor funcionar como um *proxy*, isso simplificou (e agilizou, em termos de tempo, porque o código está dividido por vários métodos) a montagem do XML da mensagem SOAP que será enviada ao receptor. Além disso, melhorou-se também a manutenção, tendo em vista que as alterações no código ficaram mais localizadas.

As *bridges* convencionais são componentes fortemente acoplados, enquanto a *bridge* proposta por nós tem baixo acoplamento, estando divida em partes que se relacionam através de mensagens. Isso permite, da mesma forma que em um sistema modular, que um componente da bridge (por exemplo o Receptor) possa ser evoluído ou alterado sem que seja necessário alterar o outro componente da bridge (Emissor). Ou vice-versa.

Por exemplo, suponha que é necessário, por alguma circunstância, gerenciar chamadas concorrentes ao Receptor. Pode-se realizar alterações neste, sem mexer em sequer uma linha de código do Emissor.

Ainda no campo das alterações, se for necessário trocar a tecnologia de web service envolvida na *bridge*, de XML-RPC para SOAP, por exemplo, são necessárias alterações substanciais em apenas um método do Emissor e em um

método do Receptor, a fim de fazer a *bridge* XML funcionar, pois seu modo de operação continua o mesmo.

As *bridges* convencionais requerem muito conhecimento técnico de COM e CORBA, o que as torna difíceis de implementar. Já a *bridge* XML requer menos esforço de construção, menos conhecimento das tecnologias COM e CORBA e nenhum conhecimento do complexo mapeamento entre elas. Logo, as *bridges* XML são mais simples e também economicamente mais baratas de se implementar e manter.

Como já analisamos nos capítulos um e dois, os componentes *bridges* convencionais são geralmente baseados em C++, em sua maioria. As bridges XML não estão ligadas a uma única linguagem específica, o que as torna um bom instrumento para interoperar COM e CORBA em um ambiente heterogêneo. Além disso, se a operação requerer enviar uma chamada remota pela internet, a *bridge* XML ultrapassa a barreira dos *firewalls* com muito mais facilidade (ver capítulo um).

Outro aspecto dos componentes *bridge* convencionais é que, como já observamos, os mesmos são fortemente acoplados (uma única peça de software) e, para interoperar COM e CORBA, precisam estar localizados ou do lado COM ou do lado CORBA. E ainda precisam estar localizados em uma plataforma (Windows) que suporte ambos (COM e CORBA). Suponhamos, por exemplo, que há duas empresas (concorrentes ou não) que desejam compartilhar alguns serviços dos seus sistemas. Em uma empresa o sistema é baseado em COM e na outra o sistema é baseado em CORBA.

A *bridge* XML possibilita que as partes da mesma possam ser desenvolvidas por equipes diferentes, sem a necessidade de uma equipe conhecer todos os detalhes do outro sistema distribuído, precisando saber apenas qual o nome do objeto e quais os métodos que podem ser chamados. Com um protocolo mais bem elaborado, como SOAP, podem até criar mecanismos de segurança, autenticação e encriptação para proteger os seus sistemas e disciplinar o acesso a eles. Com as bridges convencionais não é possível dividir a tarefa de construção da bridge dessa forma.

Resumindo, em relação às *bridges* convencionais, as *bridges* XML são mais simples e mais baratas para desenvolver. São também constituídas de partes menos acopladas, por isso são mais fáceis de desenvolver e de manter. Além disso, são independentes de linguagem (C++, Java, Phyton, Delphi VB etc), independentes de plataforma (Windows, Unix Linux etc) e apresentam a possibilidade de trocar a tecnologia XML quando for mais conveniente, sem impactar, com re-teste ou re-distribuição, os clientes já existentes.

6 Conclusão

Nos últimos anos, as estratégias de *business-to-business* levaram a uma crescente necessidade de integração de sistemas, muitos deles heterogêneos, a fim de que pudessem compartilhar serviços e cooperarem em tarefas mais complexas. As soluções apresentadas têm que desenvolver estratégias de integração as quais envolvem, em algumas situações, tecnologias como CORBA, DCOM e Internet.

Tais estratégias contêm alguns inconvenientes, como o fato de que as tecnologias baseadas em objetos distribuídos sempre requerem simetria da tecnologia usada. Ou seja, para que um objeto distribuído cliente possa se comunicar com um servidor, é necessário haver um objeto servidor de mesma tecnologia do outro lado do canal de comunicação, o que nem sempre é possível. Há ainda os problemas de segurança na comunicação via Internet, pelas barreiras proporcionadas pelos *firewalls*.

Mostramos, neste trabalho, como obter a interoperabilidade entre arquiteturas distintas de objetos distribuídos, COM e CORBA, através de *bridges* que usam protocolos baseados em XML, atuando como uma estratégia de *tunneling*. O design da *bridge* mostrou-se eficaz no seu propósito de estabelecer a comunicação de objeto distribuídos que fazem chamadas estáticas. Acreditamos que a comunicação entre objetos remotos com tecnologias diferentes via

chamadas dinâmicas também pode ser realizada de acordo com a mesma abordagem, mas não foi objeto desta dissertação, constituindo-se em desafios para trabalhos futuros. A crença baseia-se no fato de que bastaria gerar os elementos da *bridge* também dinamicamente para obter-se o mesmo efeito da *bridge* estática, embora os mecanismos que permitem chamadas dinâmicas nos modelos CORBA e DCOM não sejam os mesmos, o que limitaria o conjunto de operações disponíveis que pudessem operar pela *bridge*. Mas essa limitação tem relação com as limitações dos modelos de objetos distribuídos, não com a *bridge*.

Em relação as *bridges* convencionais, a *bridge* XML, que deve ser montada especificamente para atender a uma certa solução de integração, mostrou-se mais simples e mais barata para se construir, quando comparada a soluções proprietárias genéricas. Ela é constituída de partes menos acopladas, por isso é mais fácil de desenvolver e de manter, em um cenário no qual se deseja interoperar, pontualmente, sistemas heterogêneos, dentro de uma solução geral de arquitetura do software. Além disso, é independente de linguagem e independente de plataforma.

Uma outra virtude do design utilizado na *bridge* é que ele permite a substituição do protocolo de comunicação sem alterações substanciais na solução proposta. Portanto, é viável construir o mesmo modelo de *bridge* XML utilizando outras tecnologias de comunicação que suportem chamadas do tipo *request-response* entre softwares de diferentes tecnologias, como DCOM e CORBA. A solução baseada em SOAP, entretanto, apresenta maiores possibilidades, pois, como foi demonstrado no capítulo 5, pode ser aplicada mesmo em situações nas quais não há compatibilidade entre os *frameworks* de comunicação, caso do

Apache SOAP (que deve ser descontinuado, em favor do Axis) e do Microsoft SOAP Toolkit. Nesses casos, o SOAP possui um formato de chamada, *document*, em que o cliente e o servidor da *bridge* XML constroem e analisam as chamadas remotas de forma manual, contornando as eventuais falhas de impedância entre os formatos das chamadas, conforme foi mostrado no capítulo anterior.

Ou seja, o design da *bride* XML não permite que as diferenças dos *frameworks* atuais, causadas por algumas imprecisões na especificação do protocolo, impeça que a comunicação possa se estabelecer em todos os casos estudados.

O ponto fraco, já devidamente anotado, da *bridge* proposta, em seu estado atual - e que se relaciona com o escopo deste trabalho, é o fato de que a mesma não foi devidamente automatizada e há partes que podem ser geradas automaticamente, com o uso de ferramentas de geração de código, como o XDoclet, por exemplo.

Analisando o trabalho desenvolvido e apresentado nesta dissertação com outras iniciativas, podemos afirmar que, há algum tempo, algumas gigantes do mercado de TI têm se movimentado no sentido de desenvolver uma tecnologia que permita a integração de Web Services. Trabalhando focado nesse problema, em junho de 2002, a Sun Microsystems criou a Web Services Choreography Interface (WSCI), em parceria com outras organizações. Em agosto do mesmo ano, foi a vez da Microsoft e a IBM juntarem esforços para criar a Business Process Execution Language for Web Services (BPEL4WS). Há também diversos *frameworks* arquiteturais, como o Zachman Framework **[Zachman, 2002]** e o E-Gif **[E-GIF, 2002]**, do governo britânico, que, dentre outros objetivos, procuram

documentar padrões e processos de integração arquiteturais. Some-se a isso, os trabalhos de outros pesquisadores que tentam mapear padrões de integração, como aqueles presentes no campo da EAI - Enterprise Application Integration, bem como os adeptos da arquitetura orientada a serviço, que tem como base encapsular funcionalidades em componentes acessados via interface de serviço.

Quando iniciamos esta pesquisa, no final de 1999, havia apenas cinco protocolos baseados em XML, cujas especificações estavam publicadas no site W3C **[XML (W3C), 2001]**. Dentre esses cinco protocolos estavam XML-RPC e SOAP, ambos usados no desenvolvimento deste trabalho. Ainda não existia o termo Web Service, nem as especificações para publicar e descobrir serviços.

Apesar da iniciativa das instituições acima citadas ter uma ambição maior ou ter um foco diferente do qual nos propusemos ao longo deste trabalho, essa busca por integração é um sintoma da necessidade crescente, não somente da integração entre sistemas heterogêneos, mas de uma forma estável e consistente de interoperá-los. O nosso trabalho, ao desenvolver a *bridge* XML, deslocou-se nesse mesmo sentido.

Apêndice A

SOAP - Regras para codificação de tipos

TERMINOLOGIA

- **Value**. É um Item de dado. Todo valor tem um tipo definido. Pode ser um tipo enumerado, tipo data, número, string ou tipos similares ou também pode ser um agregado de muitos tipos primitivos;
- **Simple value**. Item de dado baseado em um tipo primitivo;
- **Compound value**. Item de dado baseado em *single values* e/ou em outros *compound values*. Os seus elementos são nomeados;
- **Accessor.** Dentro de um valor composto pode-se distinguir cada valor individualmente usando-se um ou mais *accessors*, ou seja, um nome (uma referência amplamente qualificada), um ordinal ou ambos. Pode-se ter valores compostos que tem muitos *acessors*, todos usando o mesmo nome (como em um *array*, por exemplo).
- **Array.** Um valor composto com um *accessor* formado por um nome único e com um ordinal diferente associado a cada elemento;
- **Struct.** Um valor composto com um nome único (*accessor*) para cada um de seus elementos;
- **Simple type**. Usado para distinguir a categoria de um *single value*. Ou seja, descreve o tipo de um elemento único.

- **Compound type**. Valores compostos são definidos em termos de *compound types*. Para se definir um tipo composto pode-se usar, dentre outros, um XML *schema* ou um *array*.
- **Locally scoped**. Este termo refere-se a um nome que é único em um item composto, mas não necessariamente dentro de uma mensagem SOAP;
- **Universally scoped.** Esse termo refere-se a um nome (*accessor*), em uma mensagem SOAP, que só pode ser identificado através de um URI;
- **Independent element.** Esse elemento aparece no nível mais alto da serialização;
- **Embedded element.** Se um elemento não é independente, então ele está embutido;

REGRAS PARA SERIALIZAÇÃO

Pode-se serializar mensagens SOAP apenas usando o atributo xsi:type (descreve o tipo e a estrutura) em todos os seus elementos, sem requerer mais nada. As regras de serialização afirmam que o tipo do valor é determinado somente pela referência a um esquema (XML schema) associado. Mas os esquemas não precisam seguir a notação especificada por XML, pois a especificação SOAP permite o uso de uma notação diferente, se for mais apropriada à estrutura que se pretende definir. Esta prática, por razões óbvias, não deve ser usada quando se pretende passar mensagens em um esquema de interoperabilidade, como o do tipo que nós estamos abordando neste texto.

Vamos às regras:

- **Representação de valores.** Todos os valores são representados como conteúdos de elementos (*element content*). Se um valor multi-referência (*multi-reference*) existir no documento, ele precisa ser representado como um elemento independente. Um valor de referência simples não deve ser representado como independente, mas é aceitável se ele o for;
- **Determinar o tipo do valor**. O tipo do valor de um elemento pode ser determinado através do atributo xsi:type (de acordo com a especificação do XML schema), ou através do atributo *SOAP-ENC:arrayType* presente na declaração de um elemento que contém o elemento em questão, ou ainda através do nome do elemento (que indica a relação com o tipo e o tipo pode ser determinado via esquema);
- **Representação de valores simples**. Valores simples são representados como uma seqüência de caracteres que formam o dado, sem sub-elementos. Todo valor simples tem que usar um tipo de dado de um esquema XML válido ou um tipo de dado padrão derivado desse conjunto de tipos de dados. SOAP adota todos os tipos de dados simples descritos na seção "*Built-in datatypes*" da especificação "*XML Schema Part 2: Datatypes*". Ou seja, adota os tipos *int, float, negativeInteger* e *string*, além de tipos derivados desses. Exemplo:

```
<element name="idade" type="int"/>
<element name="altura" type="float"/>
<element name="deslocamento" type="negativeInteger"/>
<element name="cor">
```

```
  <simpleType base="xsd:string">
    <enumeration value="Verde"/>
    <enumeration value="Azul"/>
  </simpleType>
</element>

<idade>35</age>
<altura>1.84</height>
<deslocamento>-80</deslocamento>
<cor>Azul</cor>
```

- **Multireference values**. Se a mensagem contém valores *multireference* (simples ou compostos) esses valores precisam ser codificados como elementos independentes, os quais precisam ter um atributo local, não qualificado, chamado id do tipo ID. Para usar esse valor, codifica-se os *accessors* como elementos vazios que tem um atributo local, não qualificado chamado *href* e de tipo *uri-reference*;
- **Strings**. A especificação define um tipo de dado string, *SOAP-ENC:string*, o qual é igual ao tipo descrito na especificação "*XML Schema Part 2: datatypes*" (http://www.w3.org/TR/xmlschema-2/#string) e não é idêntico ao tipo string definido em muitas linguagens de programação e SGBD´s, por não permitir todos os tipos de caracteres que esses softwares permitem. Um tipo string pode ser codificado como *multi-reference* ou *single-reference*. No primeiro caso, o elemento pode ter um atributo *id* e os outros *accessors* podem fazer referência via atributo *href*.

```
<universidade id="String-0">UNIFOR</universidade>
```

```
<faculdade href="#String-0"/>
```

- **Enumerations**. Podem ser definidos com a mesma semântica e os mesmos propósitos das linguagens de programação, ou seja, definindo nomes para valores.

```
<element name="Automovel" type="tns:automovel"/>
<simpleType name="Automovel" base="xsd:string">
  <enumeration value=Celta"/>
  <enumeration value="Polo Classic"/>
  <enumeration value="Clio Sedan"/>
</simpleType>

<Pessoa>
  <Nome>Augusto Pedroza</Nome>
  <Idade>35</Idade>
  <Automovel>Polo Classic</Automovel>
</Pessoa>
```

- **Array de Bytes**. A exemplo do tipo string, um array de bytes pode ser codificado como *multi-reference* ou *single-reference*. Através deste tipo (similar ao XML), pode-se enviar imagens, sons, animações e dados similares. A especificação (seção 5.2.3) diz claramente que as restrições de tamanho, presentes na codificação base64 (usada para anexos de e-mail, por exemplo), não se aplica a SOAP.

```
<somMP3 xsi:type="SOAP-ENC:base64">
  kdsuri93jgctwvjkcjdbkcvugdb9f673yt3noi8e3bf
</somMP3>
```

- **Polymorphic Accessor**. Muitas linguagens permitem que referências acessem valores de tipos diversos, disponíveis em tempo de execução. Pois bem esse tipo de *accessor* pode ser especificado em SOAP e a instância do mesmo precisa conter o atributo *xsi:type* para descrever o tipo do valor real.

  <resposta xsi:type="xsd:float">3.1416</resposta>

  <resposta xsi:type="SOAP-ENC:string">Unifor</resposta>

- **Valores Null**. Para representar os valores *null*, três opções podem ser usadas. Na primeira, o elemento *accessor* pode ser omitido da mensagem (valor default). Ou o elemento deve ser incluído e o atributo *xsi:null* configurado para 1 (mais recomendável, pois deixa a aplicação mais legível), ou ainda pode-se usar alguns outros atributos e valores independentes da aplicação.

  <resposta xsi:null="1" />

- **Representação de valor composto**. Valores compostos são codificados como uma seqüência de elementos. Cada elemento embutido representa um *accessor*, que se tem um nome único dentro do escopo do elemento que o contem, então esse elemento não precisa ter uma referencia qualificada.
- **Arrays**. São elementos compostos no qual a posição ordinal de um valor serve como único elemento de distinção do mesmo em relação aos outros. Ao se codificar um array, define-se o tipo como *SOAP-ENC:Array* ou como

derivado dele. O array precisa conter um *SOAP-ENC:arrayType* especificando o tipo dos elementos do array.

```
<element name="Fibonacci" type="SOAP-ENC:Array"/>

<Fibonacci SOAP-ENC:arrayType="xsd:int[5]">
  <number>1</number>
  <number>1</number>
  <number>2</number>
  <number>3</number>
  <number>5</number>
</Fibonacci>
```

Apêndice B

PACOTES COM IMPLEMENTAÇÕES SOAP

- **Apache** (open source). Linguagem: Java. SO´s: Unix e Windows. Site: http://xml/apache.org/soap
- **IdooXoap** (open source). Linguagens: Java e C++. SO: Unix. Site: http://www.idoox.com
- **Iona iPortal e Iona Orbix 2000**. Linguagens Java e C++. SO´s: Unix, Windows e OS/390. Site: http://iona.com
- **Microsoft SOAP Toolkit 2** (free). Linguagens: Todas as que suportam COM. SO: Windows. Site: http://msdn.microsoft.com/soap
- **Microsoft Visual Studio .NET**. Linguagens: todas as baseadas em CLR da plataforma .NET. SO: Windows. Site: http://www.microsoft.com
- **PocketSOAP** (open source). Lingaugens: todas as baseadas em COM. SO: Windows. Site: http://www.pocketsoap.com
- **Rogue Wave**. Linguagens: Todas baseadas em CORBA e SQL. SO´s: Unix e Windows. Site: http://www.roguewave.com
- **Soap::Lite** (open source). Linguagem: Perl. SO´s: Unix e Windows. Site: http://www.soaplite.com
- **White Mesa** (open source). Linguagem: C++. SO: Windows. Site: http://www.whitemesa.com

- **Zope** (open source). Linguagem: Python. SO´s: Unix e Windows. Site: http://www.zope.org
- **Axis** (open source). Linguagem: Java. SO´s Unix, Windows, Linux e Solaris. Site: http://www.apache.org

Apêndice C

LINKS RELEVANTES POR ASSUNTO USADOS NA PESQUISA

BRIDGES

- **Critical Path COM2CORBA**

http://www.cp.net

- **Iona bridges COM, CORBA**

http://news.cnet.com/news/0-1003-200-327442.html

- **Iona OrbixCOMet**

http://www.iona.com

- **The JavaBeans Bridge for ActiveX**

http://java.sun.com/products/javabeans/software/bridge/

- **Visual Edge Software, ObjectBridge**

http://vedge.com

COM

- **COM, quick and dirty**

http://www.devx.com/upload/free/features/entdev/1999/02feb99/com0299/com0299.asp

- **Dr. GUI's Gentle Guide to COM**

http://www.microsoft.com/com/news/drgui.asp

- **The Best Feature of COM+**

http://www.objectwatch.com/Issue_22.htm

- **Training Classes in COM+ and Classic COM**

http://www.rollthunder.com/trainframe.htm

- **Using DCOM with Firewalls**

http://www.microsoft.com/com/wpaper/dcomfw.asp

CORBA

- **Distributed Object Computing with CORBA Middleware**

http://www.cs.wustl.edu/~schmidt/corba.html

- **OMG CORBA Website**

http://www.corba.com/

SOAP

- **Ballinger, 2001**

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsoap/html/soapinteropbkgnd.asp

- **Converging Protocols**

http://www.xml.com/pub/a/2000/12/20/ebXML.html

- **Is COM+ Cleaner with SOAP?**

http://msdn.microsoft.com/library/periodic/period00/vb00k1.htm

- **Returning ADO Recordsets with SOAP Messaging**

http://msdn.microsoft.com/workshop/xml/articles/soapguide_ado.asp

- **SOAP Articles & White Papers**

http://www.perfectxml.com/soaparticles.asp

- **SOAP for Java**

http://www.alphaworks.ibm.com/tech/soap4j

- **SOAP: Platform-Independent Server Communication**

http://www.4guysfromrolla.com/webtech/070300-2.shtml

- **Using Microsoft's SOAP Toolkit for remote object access**

http://www.west-wind.com/presentations/soap/

XML-RPC

- **Helma XML-RPC**

http://www.classic.helma.at/bannes/xmlrpc

- **Using XML for Inter-Machine Function Calls**

http://www.rollthunder.com/newslv2n4.htm

- **XML-RPC to SOAP Translator**

http://soap.develop.com/xmlrpc/

OUTROS

- **Integrating J-Integra Tools with Forte for Java**

http://forte.sun.com/ffj/articles/J-Integra.html

- **Intrinsyc**

http://www.intrinsyc.com

- **Introducing the J-Integra® Suite**

http://www.intrinsyc.com/support/jintegrasuite/online_docs/

- **Java RMI Tutorial**

http://www.ccs.neu.edu/home/kenb/com3337/rmi_tut.html

- **Java RMI-IIOP Documentation**

http://java.sun.com/j2se/1.3/docs/guide/rmi-iiop/

- **Linux DCE, CORBA and DCOM Guide**

http://linas.org/linux/corba.html

- **Neva Object Technology**

http://www.nevaobject.com

- **Modular implementation of XML-RPC for C and C++**

http://xmlrpc-c.sourceforge.net/

Referências

[Adam, 2001] - Colin J. Adam. **Whitepaper: XML as a basis for remote procedure calls**, Webservices.org, http://www.webservices.org, 2001.

[Allamaraju et al, 2001] - Subrahmanyam Allamaraju et al, **"Professional Java Server Programming J2EE, 1.3 Edition"**, Wrox, 2001.

[Eddon & Eddon, 1998], Guy Eddon and Henry Eddon. **"Understanding the DCOM Wire Protocol by Analysing Network Data Packets"**, Microsoft System Journal, http://www.microsoft.com/msj/0398/dcom.htm, 1998.

[E-GIF, 2002], The Cabinet Office, e-Government Unit, Technology Policy Team, Interoperability Policy Advisor, **"e-Government Interoperability Framework"**, http://www.govtalk.gov.uk/schemasstandards/egif.asp, 2002

[Gamma et al, 1995]. Erich Gamma et al. **"Design Patterns"**. Addison Wesley, 1995.

[Grahan et al, 2002] – Steve Grahan et al. "Building Web Services with Java", Sams, 2002.

[Henning & Vinoski, 1999] – Michi Henning & Steve Vinoski, **“Advanced CORBA Programming with C++”**, Addison Wesley, 2000.

[Marcato, 2000] – David Marcato, **“Distibuted Computing With Soap”**, text from “Visual C++ Developers Journal”, Vol 3, No 3, April 2000, Fawcette Technical Publication, 2000.

[McLaughlin, 2001] – Brett McLaughlin, **“Java & XML”**, O´Reilly, 2001.

[Mowbray, 1997] – Thomas J. Mowbray, **“Corba Design Patterns”**, Willey & Sons, 1997.

[OMG, 2002] - Object Management Group, Inc, http://www.omg.org, 2001.

[Orfali & Harkey, 1998] - Robert Orfali e Dan Harkey, **“Client/Server Programing with Java and Corba”**, second edition, 1998.

[Orfali, Harkey & Edwards, 1999] - Robert Orfali, Dan Harkey & Jeri Edwards, **“Client/Server Survival Guide”**, third edition, 1999.

[Pritchard, 1999] - Jason Pritchard, **"COM and CORBA Side by Side: Architectures, Strategies, and Implementations"**, Addison-Wesley, 1999.

[Rector et al 1999] - Brent Rector, "ATL Internals", The Addison-Wesley Object Technology Series, Addison-Wesley, 1999.

[Raj, 2000] - Gopalan Suresh Raj, **"A Detailed Comparison of CORBA, DCOM and Java/RMI"**, http://www.execpc.com/~gopalan/misc/compare.html, 2000.

[Rana, 2002] - Ajaz Rana e Albie Collins, **"Enterprise application integration with J2EE connectors"**, CMP Media LLC, 2002.

[Rector et al 1999] - Brent Rector, **"ATL Internals"**, The Addison-Wesley Object Technology Series, Addison-Wesley, 1999.

[Resnick, 1997] - Ron I. Resnick, **"Bringing Distributed Objects to the World Wide Web"**, http://www.interlog.com/~resnick/javacorb.html, 1997.

[Scribner & Stiver, 2000] – Kennard Scribner e Mark Stiver, **"Understanding SOAP, The Authoritative Solution"**, SAMS, 2000.

[Sebesta, 2001] - SEBESTA, Robert W.. **"Concepts of Programming Languages"**, fifth edition. Addison Wesley, 2001.

[Seely, 2001] – Scott Seely, **"SOAP, Cross Plataform Web Service development using XML"**, Prentice Hall, 2001.

[Sharma, 2002] - Rahul Sharma, Beth Stearns, Tony Ng, Scott Dietzen. **"J2EE Connector Architecture and Enterprise Application Integration"**, Addison Wesley, 2002.

[SOAP 1.1 W3C, 2000], **"Simple Object Access Protocol (SOAP) 1.1, Technical Report"**, W3C, 2000.

Strahl, 2001 - Rick Strahl, **"Using Microsoft's SOAP Toolkit for remote object access"**, http://www.west-wind.com/presentations/soap/, West Wind Tecnology, 2001.

[St.Laurent, 2001] – Simon St.Laurent, Joe Johnston & Edd Dumbill, **"Programing Web Services with XML-RPC"**,

[Vawter, 2001], Chad Vawter and Ed Roman, **"J2EE vs. Microsoft.NET, 2001 – A comparison of building XML-based web services"**, The Middleware Company, http://www.theserverside.com/resources/articles/J2EE-vs-DOTNET/article.html, 2001.

[XML (W3C), 2001] - World Wide Web Consortium (W3C) - XML, **"Extensible Markup Language (XML)"**, http://www.w3.org/XML/, 2001.

[XML-RPC Home-Page, 2002] - Userland Software, Inc., "", http://www.xmlrpc.com/, 2002.

[Zachman, 2002], Zachman, John; Holfman, Samuel. **“The Zachman Institute for Framework Advancement (ZIFA)”**, http://www.zifa.com/, 2002.

[www1] Sun. http://java.sun.com/products/jdk/1.2/docs/guide/idl/index.html.

[www2] Intrinsyc. http://www.intrinsyc.com.

[www3] Neva Object Technology. http://www.nevaobject.com.

[www4] Sun. http://java.sun.com/products/javabeans/software/bridge/.

[www5] OMG. http://www.omg.org/technology/documents/formal/corba_2.htm

[www6] Visual Edge Software. (http://www.vedge.com).

[www7] Iona. http://www.iona.com.

[www8] Critical Path. http://www.cp.net.

[www9] Especificação XML-RPC. http://www.xmlrpc.com/spec.

[www10] http://www.cs.tut.fi/%7Ejkorpela/

[www11] http://www.ietf.org/rfc/rfc3023.txt.

[www12] http://www.ontosys.com/xml-rpc/extensions.html.

[www13] SOAP Specification. http://www.w3.org/TR/SOAP/#_Toc478383487

// Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Administração
Baixar livros de Agronomia
Baixar livros de Arquitetura
Baixar livros de Artes
Baixar livros de Astronomia
Baixar livros de Biologia Geral
Baixar livros de Ciência da Computação
Baixar livros de Ciência da Informação
Baixar livros de Ciência Política
Baixar livros de Ciências da Saúde
Baixar livros de Comunicação
Baixar livros do Conselho Nacional de Educação - CNE
Baixar livros de Defesa civil
Baixar livros de Direito
Baixar livros de Direitos humanos
Baixar livros de Economia
Baixar livros de Economia Doméstica
Baixar livros de Educação
Baixar livros de Educação - Trânsito
Baixar livros de Educação Física
Baixar livros de Engenharia Aeroespacial
Baixar livros de Farmácia
Baixar livros de Filosofia
Baixar livros de Física
Baixar livros de Geociências
Baixar livros de Geografia
Baixar livros de História
Baixar livros de Línguas

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo