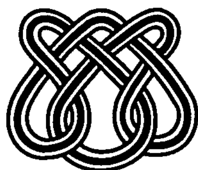
UNIVERSIDADE DE SÃO PAULO

Desenvolvimento de Aplicações Distribuídas Usando a Arquitetura CORBA

Ricardo Ribeiro dos Santos
Mário Meireles Teixeira
Marcos José Santana
Regina Helena Carlucci Santana

Nº 59

NOTAS DIDÁTICAS



Instituto de Ciências Matemáticas de São Carlos

UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

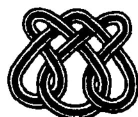
ISSN 0103-2585

Desenvolvimento de Aplicações Distribuídas Usando a Arquitetura CORBA

Ricardo Ribeiro dos Santos
Mário Meireles Teixeira
Marcos José Santana
Regina Helena Carlucci Santana

Nº 59

NOTAS DIDÁTICAS



São Carlos
Dez./2002

Desenvolvimento de Aplicações Distribuídas usando a Arquitetura CORBA

Ricardo Ribeiro dos Santos[1] Mário Meireles Teixeira[2] Marcos José Santana[3]
Regina Helena Carlucci Santana[3]

[1]Centro de Ciências Exatas e da Terra
Universidade Católica Dom Bosco
Av. Tamandaré, 6000, Jd. Seminário
79117-900 Campo Grande, MS

[2]Departamento de Informática
Universidade Federal do Maranhão
Av. dos Portugueses s/n, Campus do Bacanga
65085-580 São Luís, MA

[3]Departamento de Ciências de Computação e Estatística
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo
Caixa Postal 668
13560-970 São Carlos, SP

Novembro de 2002

Sumário

1 Sistemas Distribuídos **5**

1.1 Conceitos 5

1.2 Características de um Sistema Distribuído 6

1.3 Modelos Arquiteturais 10

1.4 Dificuldades na construção e gerenciamento 10

1.5 O modelo Cliente-Servidor 12

1.5.1 Conceitos 12

1.5.2 Vantagens e Desvantagens 13

1.5.3 Comunicação 14

2 A Arquitetura CORBA **21**

2.1 Introdução 21

2.2 A OMA 21

2.3 Componentes da arquitetura CORBA 23

2.3.1 O ORB 23

2.3.2 Os *stubs* 24

2.3.3 Os *Skeletons* 24

2.3.4 O Repositório de Interfaces 25

2.3.5 Interface de Invocação Dinâmica 25

2.3.6 Interface de *Skeleton* Dinâmica 25

2.3.7 Repositório de Implementação 26

2.3.8 Adaptador de Objetos 26

2.3.9 Interface do ORB 26

2.4 Interoperabilidade entre ORBs 26

2.5 Referências de Objetos 27

2.6 A IDL e os Metadados em CORBA 28

2.6.1 Estrutura da IDL 28

2.7 Serviços CORBA 29

3 Exemplos de Aplicações **33**

3.1 Implementações CORBA 33

3.1.1 ORBacus 33

3.1.2 TAO (*The ACE ORB*) 34

3.2 O exemplo Agenda 36

3.3 O exemplo Caixeiro Viajante com *callback* 41

4 Considerações Finais **50**

Referências Bibliográficas **51**

Lista de Figuras

1.1 Modelo estação de trabalho e servidor 10
1.2 Diversidade de elementos em um Sistema Distribuído 11
1.3 Uma interação em um sistema Cliente-Servidor 13
1.4 Aplicações cliente-servidor no sistema RPC 16
1.5 Funcionamento de um sistema RPC 18
1.6 Referência implícita para um objeto remoto 19
1.7 Referência explícita para um objeto remoto 20

2.1 Arquitetura OMA 22
2.2 Arquitetura CORBA 23
2.3 Exemplo de uma Referência de Objeto 28
2.4 Exemplo de IDL utilizando módulos e interfaces 29
2.5 Exemplo de IDL utilizando tipos definidos pelo usuário e métodos com parâmetros de entrada 30
2.6 Exemplo de um serviço de nomes 31

3.1 Estrutura básica da implementação TAO 35
3.2 Exemplo de um arquivo de *makefile* 37
3.3 Interface IDL para a aplicação Agenda 38
3.4 Programa cliente em ORBacus (Parte I) 39
3.5 Programa cliente em ORBacus (Parte II) 40
3.6 Programa servidor em ORBacus 41
3.7 Implementação da interface Agenda 42
3.8 Arquivo IDL para o problema do Caixeiro Viajante 43
3.9 Inicialização de variáveis e implementação do *callback* no cliente 44
3.10 Inicialização do ORB, POA e Serviço de Nomes 45
3.11 Invocação do método *callback* nos servidores 46
3.12 Implementação de métodos específicos para resolução do Problema do Caixeiro Viajante 47
3.13 Implementação do método (PCV) que é invocado pelo cliente 48

3.14 Inicialização do ORB, POA e espera de requisições de clientes 49

Capítulo 1

Sistemas Distribuídos

1.1 Conceitos

O termo "sistema distribuído", como tantos outros no campo da Informática, tem sido usado à exaustão, para descrever desde simples sistemas operacionais de rede até complexas implementações de sistemas distribuídos reais, muitas vezes até como uma estratégia de marketing. Assim, como tantas outras expressões largamente utilizadas, corre o risco de perder seu sentido original.

É preciso então, desde já, delimitar-se bem o escopo deste termo, a fim de se ter certeza de que se está falando sobre as mesmas coisas. Uma definição bastante coerente, embora antiga, é a enunciada por Tanenbaum [Tanenbaum and Renesse, 1985] e aperfeiçoada por Müllender [Mullender, 1993], que diz que um sistema distribuído é aquele que se apresenta aos seus usuários como um sistema centralizado, mas que, na verdade, funciona em diversas CPUs independentes. Além disso, um sistema distribuído não deve ter pontos críticos de falha, ou seja, se um componente do mesmo quebrar, isto não deve fazer com que o sistema como um todo falhe, sendo esta característica de estabilidade uma de suas principais vantagens em relação a um sistema centralizado.

Coulouris [Coulouris et al., 2001] enfatiza que os computadores de um sistema distribuído devem estar conectados através de uma rede, caracterizando bem o fato de que estes não precisam estar localizados em uma única sala, ou mesmo próximos entre si, já que, a princípio, não há limite para a área abrangida por um sistema desse tipo. Além disso, esse autor ressalta o fato de que os computadores devem estar equipados com software de sistemas distribuídos, por ser justamente este componente que fará com que os usuários vejam o sistema como uma entidade única, integrada, embora o mesmo se encontre funcionando em computadores diferentes, situados em locais os mais diversos.

1.2 Características de um Sistema Distribuído

Nesta seção, serão abordadas algumas das características fundamentais de um sistema distribuído, indicando-se porque estas o tornam tão especial e atrativo.

- Compartilhamento de Recursos. O termo "recurso" é bastante abrangente e, neste contexto, refere-se a qualquer entidade do sistema passível de ser compartilhada. Incluem-se aqui tanto componentes de hardware como discos, impressoras, scanners, quanto elementos como arquivos, aplicativos e outros. O compartilhamento de recursos é um ponto fundamental em qualquer sistema, seja ele centralizado ou distribuído. No primeiro caso, o compartilhamento é imediato, uma vez que todos os recursos estão concentrados em único local, ao qual se conectam os terminais. Já no caso distribuído, os recursos encontram-se espalhados pela rede, contidos dentro dos diversos equipamentos e o acesso a eles é gerenciado por meio de softwares servidores, que determinam quais as operações que podem ser realizadas e por quem. Portanto, gerenciar os recursos de um sistema distribuído de forma ótima não é uma tarefa trivial.

- Capacidade de Expansão (*Scalability*) e Modularidade. Em um sistema centralizado, há um limite físico para o número máximo de processadores que podem ser incorporados em uma máquina, para a capacidade máxima de memória e para o número de discos. Já em um sistema distribuído, teoricamente não existe limite para o número de máquinas que podem compor o sistema, nem para a distância e diversidade das mesmas. Isto, com certeza, é uma característica desejável para qualquer sistema, mas complica sobremaneira o desenvolvimento do software distribuído, uma vez que este tem que continuar se comportando satisfatoriamente à medida em que o sistema cresce. Outro ponto de pressão é a rede, que pode vir a ter sua capacidade sobrepujada pelo excesso de mensagens e dados circulando na mesma. A modularidade implica que o sistema pode ser expandido gradualmente, introduzindo-se software e máquinas conforme a necessidade. Isto, de certa forma, facilita a manutenção do sistema e o isolamento de falhas.

- Disponibilidade. Como já foi comentado, uma das características mais interessantes de um sistema distribuído é que a falha de um componente individual não pode fazer com que o sistema como um todo pare. Dessa forma, se uma estação de trabalho falha, os programas sendo executados nela têm que ser migrados para uma outra máquina da maneira mais transparente possível para o usuário, idealmente de forma que este nem mesmo se dê conta da falha que acabou de ocorrer. Problemas em servidores (de arquivos, mail, impressão etc.) também têm que ser tratados de forma transparente, com um outro equipamento tomando o lugar do que falhou imediatamente. A fim de garantir a disponibilidade do sistema a qualquer custo, o projeto do mesmo tem que contemplar dois aspectos: redundância

(de hardware, software e dados) e algoritmos de recuperação.

A redundância, se feita com critério e controlada, é um dos pontos mais importantes para aumentar a disponibilidade de um sistema distribuído. Isto é óbvio, pois se houver um único servidor de arquivos, por exemplo, a falha dessa máquina interromperá a operação de todo o sistema. Portanto, para os serviços mais críticos, é sempre bom ter uma máquina sobressalente, pronta para entrar em ação (é claro que, no intervalo entre as falhas, que se espera ser razoavelmente grande, essa máquina tem que ser usada para outros fins). No caso de redundância de dados e software, é preciso empregar técnicas para evitar a proliferação de inconsistências.

Entretanto, é bom sempre ter em mente que, se por um lado, a redundância ou replicação aumenta a disponibilidade dos recursos e informações, por outro lado ela vai também implicar em uma certa perda de desempenho do sistema, devendo-se analisar, portanto, quando é o momento certo de replicar.

Quanto aos algoritmos de recuperação, estes devem ser capazes de retornar o sistema a um estado correto anterior, no caso de uma falha. Têm-se desenvolvido bastante pesquisas nessa área e muita contribuição tem vindo da área de banco de dados. Aqui, entra em cena o gerenciamento de transações distribuídas.

- Concorrência. Em um sistema centralizado, do tipo time sharing, que possua um só processador, a concorrência dos processos é obtida através do revezamento no uso da CPU. Mesmo no caso em que exista mais de uma CPU, há um limite até mesmo físico e de projeto para o número máximo de processadores que uma máquina pode conter. Já em um sistema distribuído, devido à característica deste possuir várias máquinas diferentes, conectadas por uma rede, o fato de ocorrer paralelismo e concorrência é uma conseqüência natural.

  Considere, por exemplo, o caso de um usuário que submeta vários processos (programas) para execução em uma estação de trabalho. A concorrência, aqui, ocorre entre os seus próprios processos e, tomando-se a visão global do sistema, em outras estações existirão também outros processos executando, pertencentes a outros usuários, o que aumenta ainda mais o grau de paralelismo. E, o que é mais importante, processos de um usuário executando em uma máquina não irão degradar o desempenho dos processos de um outro usuário em uma estação diferente (salvo, talvez, pelo uso que estes possam vir a fazer da rede), diferentemente de um sistema de tempo compartilhado, em que normalmente existe uma única CPU, barramento, memória etc. para ser compartilhados entre vários processos.

  Outra oportunidade de concorrência é o fato de os serviços do sistema serem executados

em diferentes servidores, o que implica dizer que as requisições de serviço por parte dos clientes são efetivamente atendidas em paralelo. Isto é particularmente interessante para serviços muito solicitados, como o acesso a arquivos, porém é necessário dotar as máquinas de um software adequado a fim de sincronizar as requisições e garantir que o paralelismo na execução das operações não venha a provocar o surgimento de inconsistências.

Assim, vê-se que, em um sistema distribuído, é bastante presente a concorrência na execução das tarefas, surgindo como uma característica natural do sistema.

- Informação de Estado Compartilhado. Dado o escopo de um sistema distribuído, o número e a diversidade das estações de trabalho e a própria natureza distribuída dos dados, torna-se muito difícil obter informações globais sobre o estado do sistema. O que normalmente acontece é que determinadas regiões do mesmo possuem informações de âmbito local, que são compartilhadas sob demanda. Como exemplo, tem-se o caso do roteamento entre redes: cada nó roteador contém informações sobre qual o melhor caminho para chegar a um determinado local, a partir dele. Assim, se chega um pacote para aquele destino, este é enviado para o nó indicado na tabela de roteamento, de lá para outro ponto e assim por diante. Não há, em nenhum local, um elemento coordenador que contenha as informações de roteamento para toda a rede, pois isto seria impraticável devido ao enorme volume de dados que teria que ser processado. Esta situação é a que normalmente se encontra em sistemas distribuídos e, se por um lado é vantajosa, porque não dá a nenhum equipamento a tarefa de saber tudo sobre determinado aspecto do sistema (o que criaria pontos críticos de falha), por outro lado torna muito difícil obter informações sobre certos pontos fundamentais para o gerenciamento do sistema, como a carga de trabalho nas estações do mesmo, a utilização da rede, dentre outros.

- Transparência. Transparência, neste contexto, significa esconder dos usuários e programadores a separação inerente aos componentes de um sistema distribuído, de modo que estes o percebam como um todo integrado e não como um conjunto de elementos independentes. A separação dos componentes do sistema é que permite, por exemplo, o paralelismo real na execução dos programas, como já foi comentado, o isolamento de falhas em regiões específicas do sistema e sua recuperação sem afetar outros pontos do mesmo, o monitoramento dos canais de comunicação com o objetivo de empregar políticas de segurança, e também garante a capacidade de crescimento ou redimensionamento do sistema à medida em que isto se faz necessário. Entretanto, é de suma importância, em um sistema distribuído, que essa dispersão dos seus componentes seja escondida dos usuários através do uso de softwares apropriados de gerenciamento. A seguir, serão comentados alguns tipos de transparência que podem ser encontrados em um sistema.

- Transparência de Acesso significa que as operações utilizadas para se ter acesso a um objeto ou recurso (arquivos, dispositivos, processos) são as mesmas, independentemente do fato de este se encontrar na mesma máquina onde foi solicitada a operação ou em um equipamento remoto. Outro aspecto que auxilia no aumento da transparência é o fato de que o nome de um objeto não deve conter nenhuma informação acerca de sua localização, de modo que este possa ser invocado sem conhecimento de seu lugar físico. Isso leva a dois tipos de transparência: a de Localização e a de Migração, que implica que um objeto pode ser movido de um lugar para outro do sistema sem afetar a maneira como os usuários ou programas o referenciam. Se um sistema possui transparência de acesso e localização, diz-se que ele apresenta a chamada Transparência de Rede, que facilita sobremaneira a utilização dos recursos do sistema distribuído, devendo ser, portanto, um dos principais objetivos a serem perseguidos pelos projetistas do mesmo.

- Transparência de Concorrência permite que diversos processos tenham acesso aos mesmos objetos concorrentemente sem que um afete o trabalho do outro e sem que ocorra uma degradação significativa no desempenho.

- Transparência de Replicação garante que diferentes réplicas de um objeto podem ser mantidas pelo sistema a fim de aumentar sua confiabilidade e melhorar seu desempenho sem que os usuários se dêem conta disso.

- Transparência em relação a Falhas significa que estas serão confinadas ao local onde ocorreram, na medida do possível, a fim de que os usuários não diretamente afetados possam continuar trabalhando normalmente, apesar dos erros ocorridos no hardware ou software.

- Transparência de Desempenho permite que o sistema se reconfigure (por exemplo, através da migração de processos de uma máquina para outra) a fim de continuar oferecendo a mesma qualidade de serviço aos seus usuários, apesar da variação de sua carga de trabalho.

- Transparência de Expansão (*Scaling*) garante que o sistema distribuído possa se expandir sem alterar a sua estrutura básica e sem provocar a necessidade de se rescrever os programas existentes.

Para concluir, vale acrescentar que não há, atualmente (pelo menos no âmbito desta revisão bibliográfica), nenhum sistema que apresente todas estas formas de transparência. Na verdade, elas são apenas diretrizes que devem nortear o projeto de qualquer sistema distribuído, a fim de garantir uma maior homogeneidade e coerência no acesso a seus objetos, por parte dos usuários e programas.

1.3 Modelos Arquiteturais

Um modelo arquitetural de um sistema define quais são os diversos componentes do mesmo, como estes estão organizados e como os usuários têm acesso aos vários recursos do sistema. No campo dos sistemas distribuídos, alguns modelos diferentes já foram experimentados, sendo que um deles, o modelo estação de trabalho e servidores (Figura 1.1), firmou-se como o padrão de fato para a organização desse tipo de sistema.

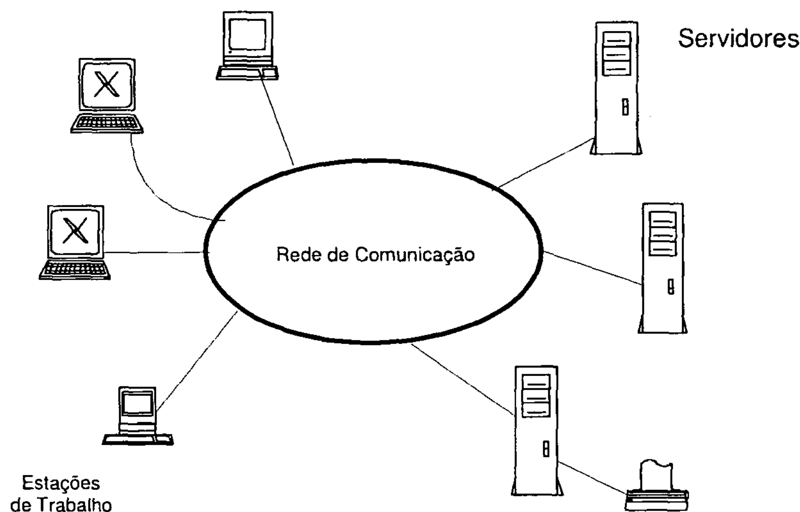


Figura 1.1: Modelo estação de trabalho e servidor

Nesse modelo, clientes executando em uma ou mais estações de trabalho requisitam serviços aos servidores. De forma geral, tem-se firmado como premissa que as estações de trabalho são máquinas utilizadas pelos usuários do sistema e, por isso, possuem capacidades de memória e processamento mais limitadas se comparadas com as máquinas servidoras. Além da relativa facilidade de implantação, esse modelo arquitetural tem sido amplamente utilizado em razão do baixo custo para aquisição de estações de trabalho e até mesmo de servidores.

1.4 Dificuldades na construção e gerenciamento

Pelo que já foi discutido, percebe-se que um sistema distribuído não é algo trivial de se construir e manter. Ainda há um bom caminho a percorrer até que se chegue ao sistema perfeito, totalmente transparente, com desempenho aceitável sob quaisquer condições e que se mantenha em funcionamento por mais sérios que sejam os erros ocorridos.

Nesta seção, serão discutidas as razões da complexidade inerente ao projeto, construção e gerenciamento de sistemas distribuídos.

- Diversidade. Os elementos que compõem um sistema distribuído são muito diversos, sejam eles estações de trabalho, placas de rede, cabos de interconexão ou softwares de gerenciamento. Para repetir uma máxima já consagrada, muitas vezes, coisas conhecidas, que funcionam muito bem sozinhas ou em um ambiente homogêneo, apresentam problemas inesperados quando conectadas em um ambiente tão heterogêneo como é um sistema do tipo distribuído. E aqui, ao contrário dos sistemas do tipo *time sharing* tradicionais, não há um fornecedor único a quem recorrer. Na verdade, há vários, sem nenhuma ligação entre si além de um cliente comum e é este último elemento que tem que resolver os problemas que surgem e manter o sistema funcionando. A Figura 1.2 apresenta um exemplo da diversidade de elementos em um Sistema Distribuído.

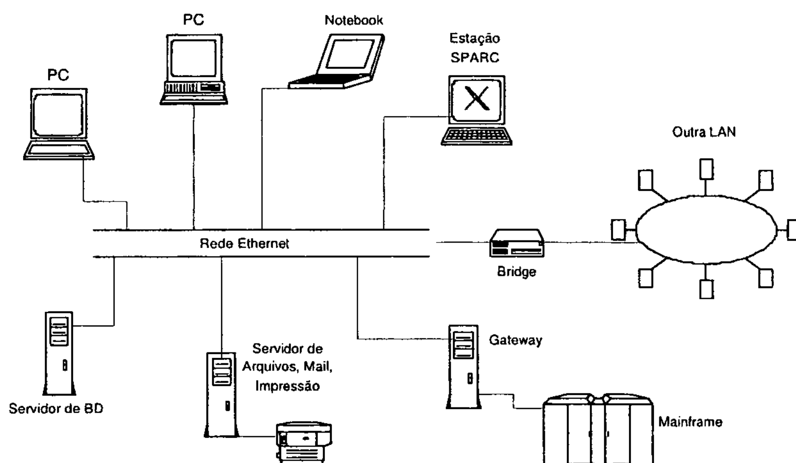


Figura 1.2: Diversidade de elementos em um Sistema Distribuído

- Interconexão. A interconexão de máquinas pode trazer problemas de incompatibilidade, de conectores e outros. Porém, os maiores problemas surgem na interconexão em nível de sistema, por exemplo para a comunicação e o acesso a arquivos entre sistemas UNIX de famílias diferentes. Outro problema pode ocorrer na troca de mensagens de correio eletrônico. Como já foi comentado, esta região de interconexão é justamente o "buraco negro" sobre o qual nenhum dos fornecedores envolvidos quer opinar.

- Propagação e Rastreamento de Erros. Nos sistemas centralizados, encontrar um erro é relativamente fácil se comparado aos sistemas distribuídos, afinal, excluindo-se as linhas de comunicação até os terminais, o erro estará naquela máquina central, já que todo o processamento e recursos se encontram em torno dela. Entretanto, em um ambiente distribuído, dada a diversidade dos elementos envolvidos e considerando-se que cada estação é autônoma no que se refere a sua capacidade de processamento, fica difícil encontrar exatamente qual o ponto em que se deu o erro. Por exemplo, se uma mensagem enviada a

uma aplicação em um outro computador não foi respondida, como saber com exatidão o que ocorreu? O equipamento de destino não está funcionando? A conexão até ele está interrompida? A solicitação se perdeu na rede ou será que foi a resposta a ela? Ou talvez a estação de destino simplesmente estava ocupada demais para responder? Essas e outras perguntas surgem a todo momento e respondê-las não é nem um pouco trivial.

Outro ponto importante é como impedir a propagação dos erros, já que as máquinas do sistema estão interconectadas e, portanto, aptas a receber mensagens de qualquer lugar. Um problema muito comum é uma estação estar com sua tabela de roteamento corrompida e começar a enviar os pacotes para lugares indevidos, o que pode congestionar desnecessariamente certas regiões da rede ou fazer com que pacotes se percam devido à expiração do seu tempo de vida. Uma solução para esse tipo de problema é delegar a certos equipamentos a função de ficar monitorando o funcionamento do sistema periodicamente.

- Evolução do Sistema. Uma das principais vantagens de um sistema distribuído é justamente a sua capacidade de expansão, quase "sem limites", pela simples adição de novas estações de trabalho ou de linhas de comunicação para integrar novas redes ao mesmo. Contudo, essa evolução deve ser cuidadosamente planejada, para que o sistema não venha simplesmente a crescer, mas sim que continue a atender a seus usuários de forma satisfatória. Um erro muito comum é simplesmente começar a conectar estações ao sistema indiscriminadamente, à medida em que elas vão chegando. Nesse sentido, é importante compreender que o sistema deve evoluir, mas é preciso fazê-lo com responsabilidade, analisando bem se não é necessário modificar a topologia da rede ou trocá-la por uma outra de capacidade maior.

1.5 O modelo Cliente-Servidor

1.5.1 Conceitos

Atualmente, o modelo sobre o qual se baseia a maior parte dos sistemas distribuídos adota o paradigma Cliente-Servidor. Basicamente, esse modelo especifica que existe um conjunto de processos servidores, os quais gerenciam o acesso a um determinado tipo de recurso e uma série de processos clientes, os quais necessitam do acesso aos recursos compartilhados para realizar suas tarefas, fazendo, portanto, solicitações aos servidores. Dessa maneira, todo e qualquer recurso existente no sistema distribuído (software, hardware e dados) está sob a guarda de um processo servidor, o qual é o responsável por administrar o acesso concorrente dos diversos clientes ao mesmo. A Figura 1.3 exemplifica o funcionamento básico de um sistema cliente-servidor.

A fim de firmar bem o conceito de cliente e servidor, considere-se o que diz Ullman: um cliente pode ser qualquer computador ou estação de trabalho conectado ao sistema através da rede, que o usuário utiliza para ter acesso aos recursos disponíveis. Um servidor é uma máquina

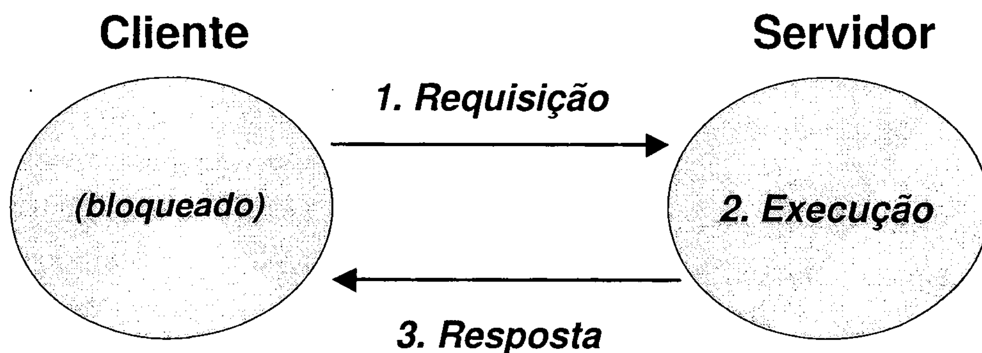


Figura 1.3: Uma interação em um sistema Cliente-Servidor

que provê aos clientes recursos como discos de grande capacidade, bancos de dados, conexões com outras redes, serviços de impressão etc. Os servidores podem ser computadores de grande porte, minicomputadores, estações de trabalho poderosas ou dispositivos de rede.

O modelo cliente-servidor é responsável por muito da flexibilidade e melhor desempenho dos sistemas distribuídos quando comparados aos sistemas centralizados, já que os servidores podem ser distribuídos pela rede a fim de ficarem mais próximos dos clientes que os utilizam e é possível dividir a tarefa de gerenciamento entre várias máquinas. Um ponto interessante a esclarecer é que normalmente haverá mais de um servidor para cada tipo de recurso, justamente a fim de atingir as vantagens mencionadas, sendo importante distinguir entre os serviços que o sistema oferece aos seus clientes e os servidores que dão suporte a eles: o serviço é sempre único; os servidores podem ser vários, inclusive atuando de forma cooperativa. Outro aspecto é que um determinado processo pode ser servidor de outro e cliente de um terceiro, o que torna o sistema como um todo bem flexível.

1.5.2 Vantagens e Desvantagens

O modelo cliente-servidor permite que diversas máquinas, grandes e pequenas, sejam interligadas. Assim, em um sistema desse tipo, não há, normalmente, uma única máquina da qual dependa o funcionamento de todo o sistema, o que aumenta sua confiabilidade. A capacidade de processamento agregada dessas máquinas é incalculável e os usuários possuem poder de processamento local, graças às estações de trabalho. Os sistemas cliente-servidor também obedecem à filosofia de sistemas abertos, o que significa dizer que não há um único fornecedor, mas vários, aumentando a competitividade e baixando os custos. Estes sistemas também podem crescer gradativamente, pela adição de uma ou outra estação de trabalho ou periférico, conforme a necessidade. Finalmente, não há um único ambiente de trabalho e aplicativos aos quais todos os usuários tenham que se submeter. Pelo contrário, devido à capacidade de processamento de cada estação individualmente, cada usuário pode trabalhar com os aplicativos e ambientes com

os quais se sentir mais confortável e que melhor satisfaçam às suas necessidades.

Mas, apesar de todos esses pontos positivos, também há dificuldades. Há muito mais com que se preocupar, ou seja, muito mais coisas podem dar erradas, devido à diversidade dos equipamentos e à área de abrangência do sistema. Assim, quando ocorre algum problema, é sempre mais difícil isolá-lo e descobrir o culpado e, para solucioná-lo, não é possível simplesmente contactar um dado fornecedor que terá resposta para tudo. Nessa nova realidade, o administrador do sistema vai ser, muitas vezes, o responsável por preencher as lacunas deixadas pelos fornecedores. As ferramentas de suporte ao gerenciamento do sistema também não são tão variadas e completas como as existentes para o caso dos sistemas centralizados, embora tenha havido grande evolução nos últimos anos. A forma de se desenvolver softwares para sistemas cliente-servidor é diferente da programação tradicional e, portanto, é considerável o gasto que se terá com o treinamento da equipe de analistas e programadores.

1.5.3 Comunicação

Os componentes de um sistema distribuído encontram-se lógica e fisicamente separados e, portanto, para que desempenhem suas tarefas, é necessário que haja uma troca de informações entre eles. No caso particular do modelo cliente-servidor (Figura 1.3), a comunicação objetiva principalmente a realização de serviços e, basicamente, consiste na transmissão de uma requisição de um processo cliente para um processo servidor, recepção da mensagem e execução da solicitação pelo servidor, e envio de uma resposta ao cliente. O desempenho de um sistema distribuído depende, e muito, da eficiência da comunicação entre suas partes e esta não provém apenas da rede, mas também de quão cuidadosamente o subsistema de comunicação é projetado, em nível de software.

A seguir, são examinadas algumas abordagens geralmente utilizadas para a comunicação em sistemas cliente-servidor. Dessas, recebe destaque neste trabalho a abordagem baseada em objetos distribuídos, em especial a adotada pelo padrão CORBA.

Passagem de mensagens (*Message Passing*)

Esta é a forma mais simples de todas e faz uso de primitivas básicas de programação do tipo *Send* e *Receive*. Acontece entre um par de processos e envolve a transmissão de uma seqüência de valores (uma mensagem) de um processo emissor para um receptor através de um meio físico (a rede) e a conseqüente aceitação dessa mensagem pelo processo receptor.

As primitivas de comunicação podem ser síncronas (bloqueantes) ou assíncronas (não bloqueantes). No caso das primitivas Assíncronas, tão logo o *Send* seja emitido e a mensagem colocada em uma fila para transmissão, o controle retorna ao programa, não havendo necessidade de aguardar até que a mensagem seja efetivamente enviada. No caso do *Receive* assíncrono, o programa informa sua intenção de receber uma mensagem e aloca um buffer para a recepção.

Quando esta chegar, o programa sofre uma interrupção ou, então, periodicamente faz uma pesquisa (*polling*) para verificar se a mensagem já chegou. Na abordagem Síncrona, um *Send* não retornará controle ao programa emissor até que a mensagem tenha sido efetivamente enviada e, se o *Send* for do tipo confiável, o programa deverá até mesmo aguardar pelo *acknowledgement* de recepção da mensagem. Um *Receive* bloqueante paralisa a execução do programa até a chegada de uma mensagem. As primitivas síncronas facilitam a programação, porém implicam em perda de desempenho, já que os programas que as utilizam não podem operar em paralelo com a transmissão e recepção de mensagens.

Chamadas a Procedimentos Remotos (*Remote Procedure Calls*)

As chamadas a procedimentos remotos representam o próximo nível de complexidade em relação ao *message passing* e sua idéia básica é fazer com que a comunicação entre processos em máquinas diferentes seja o mais semelhante possível a uma chamada convencional de procedimentos, conceito que é amplamente dominado pelos programadores em geral.

O mecanismo de RPCs, por sua simplicidade e por esconder dos programadores os difíceis detalhes de comunicação, tornou-se virtualmente o padrão para a transmissão de mensagens e implementação de aplicações no modelo cliente-servidor e nos sistemas distribuídos de um modo geral.

Como já discutido no paradigma cliente-servidor, os usuários interagem com aplicações que podem ser clientes de outros serviços disponíveis no sistema. Cada serviço provê uma série de operações que podem ser invocadas pelos clientes. A comunicação entre clientes e servidores se dá através de um mecanismo do tipo *send/receive* (*request/reply*).

Os clientes, então, obtêm serviços enviando uma mensagem a um determinado servidor, o qual executa a solicitação e envia uma mensagem de resposta ao cliente, comunicando o resultado da mesma e, eventualmente, mais alguns parâmetros. O cliente, normalmente, fica bloqueado enquanto seu pedido está sendo executado, dando prosseguimento às suas tarefas quando chega a mensagem de resposta.

Todo esse processo de enviar mensagens, bloquear-se, aguardar pela conclusão do serviço, etc., é um pouco complicado e exige um certo grau de conhecimento do programador, além de não ser transparente, pois é preciso se preocupar com protocolos utilizados, *acknowledgements*, *sockets* e outras coisas mais, não sendo isto nem um pouco trivial para a maioria das pessoas. O mecanismo de Chamadas a Procedimentos Remotos (*Remote Procedure Calls*- RPC), portanto, vem justamente introduzir uma abstração maior nesse cenário, permitindo que os clientes solicitem tarefas aos servidores simplesmente fazendo chamadas remotas a procedimentos, de uma maneira semelhante à que é utilizada em linguagens de alto nível convencionais.

Dessa forma, um serviço disponível através da rede apresenta-se a seus clientes como um módulo que possui uma interface bem definida de acesso a suas operações, interface esta cons-

tituída pelos procedimentos. Isto não só dá uma maior uniformidade de acesso às suas funções, escondendo detalhes de implementação das mesmas, mas também aumenta a segurança do sistema e, evidentemente, sua transparência.

Um dos objetivos de um sistema de RPC é fazer com que as chamadas remotas a procedimentos sejam, pelo menos do ponto de vista do programador, o mais semelhante possível às chamadas a procedimentos convencionais. Isto, entretanto, não se mostra uma tarefa fácil, pois o ambiente em que é feita a chamada é diferente daquele em que a mesma é executada, o que provoca diversas conseqüências interessantes, algumas das quais serão analisadas a seguir.

A primeira e talvez mais peculiar delas é que não é possível fazer, pelo menos da maneira usual, uma chamada a um procedimento passando parâmetros por referência ou com ponteiros, da mesma forma que não faz sentido utilizar variáveis globais. Devido à separação entre os ambientes de invocação do procedimento e de execução do mesmo, o único tipo possível de comunicação de parâmetros é através de passagem por valor.

Outro aspecto importante de um sistema que suporta RPC é que o mesmo tem que prover uma Linguagem de Definição de Interface (*Interface Definition Language* - IDL), a qual especifica com detalhes a interface de cada procedimento que os servidores disponibilizam aos seus clientes, fornecendo o nome dos procedimentos, sua lista de parâmetros, seus tipos e informando se estes são de entrada ou saída. Estas informações constituem as chamadas *Procedure Signatures* e são de vital importância para a geração dos *stubs* (que escondem a complexidade da comunicação dos programadores), sendo utilizadas, ainda, para viabilizar um mecanismo de passagem de parâmetros "por referência". A Figura 1.4 apresenta o funcionamento básico de aplicações RPC.

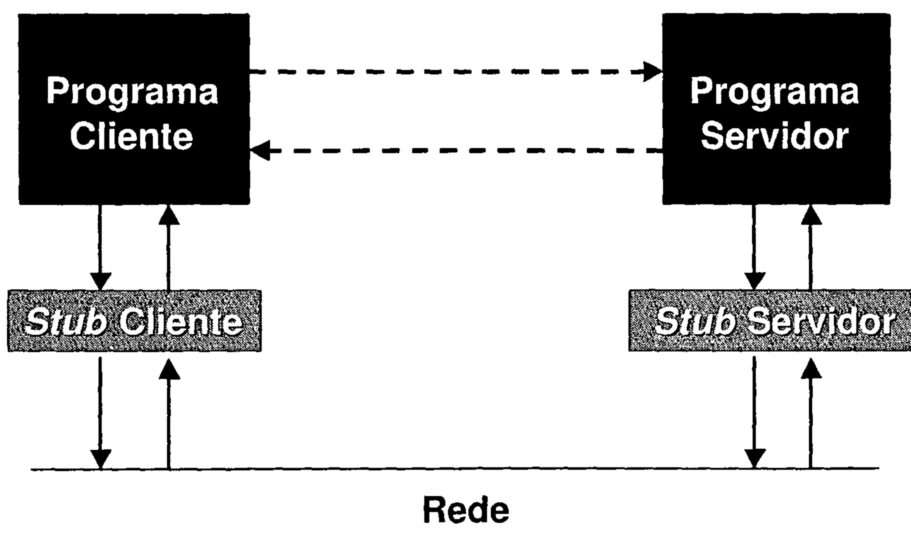


Figura 1.4: Aplicações cliente-servidor no sistema RPC

Existem duas classes de RPC:

- Na primeira classe, o mecanismo de RPC está integrado a uma linguagem de programação. A vantagem disso é que a IDL faz parte das construções da linguagem e aspectos como tratamento de exceções tornam-se mais fáceis. Exemplos são os sistemas Cedar, Argus e Arjuna.
- Na outra classe, é utilizada uma linguagem específica para a definição da interface entre clientes e servidores, tendo a vantagem de não ser necessário aprender uma nova linguagem de programação, aproveitando-se o conhecimento previamente adquirido. Na prática, a linguagem utilizada é a C, que se firmou como um padrão de fato para escrever aplicações desse tipo. Exemplos são o SunRPC, no qual se baseia o NFS (*Network File System*); o Sistema ANSA e o MIG (sistema operacional Mach).

Os procedimentos *stubs* são talvez uma das partes mais importantes de um Sistema de RPC. Sua função é isolar o programador dos detalhes referentes à comunicação através da rede.

Eles se fazem necessários porque, no momento em que o programa "centralizado" é dividido, levando-se algumas de suas subrotinas para uma máquina remota, é preciso introduzir algum elemento que vá assumir o lugar dessa subrotina junto ao módulo que a utiliza e vice-versa. Dessa forma, no lado cliente, o procedimento *stub* "substitui" o procedimento que foi movido para a máquina remota e, no lado servidor, ele toma o lugar do módulo que iniciou a chamada, ambos os *stubs*, então, contribuindo para que a separação dos componentes da aplicação se dê de forma transparente ao programador. Observe-se que as funções propriamente ditas das subrotinas estão contidas no código desenvolvido pelo programador, que não é alterado. Os *stubs* apenas cuidam dos detalhes relativos à troca de mensagens.

Outro aspecto importante de um Sistema de RPC é a Biblioteca RPC (também conhecida como RPC *Library* ou RPC *Runtime* API), a qual implementa um conjunto de funções que são utilizadas tanto pelo código da aplicação quanto pelos *stubs*, reduzindo as chamadas que os procedimentos teriam que fazer a um pequeno conjunto de funções definidas na API, o que facilita a programação distribuída e reduz a ocorrência de erros.

Como já foi comentado, o mecanismo de RPC tem que ser transparente, isto é, para o programador as chamadas têm que parecer locais. Para tanto, é necessário introduzir alguns elementos que vão esconder as complexidades inerentes ao problema. Nesta seção, são analisados cada um deles à medida em que se estudam as etapas envolvidas em uma chamada remota (Figura 1.5).

Ao ser feita uma chamada a um procedimento remoto, da mesma forma que em uma chamada local, os parâmetros são colocados na pilha, assim como é guardado o endereço de retorno. Entretanto, em vez de se chamar o procedimento propriamente dito, é invocado um *stub*, o qual coloca os parâmetros em uma mensagem (*marshalling*), juntamente com uma identificação

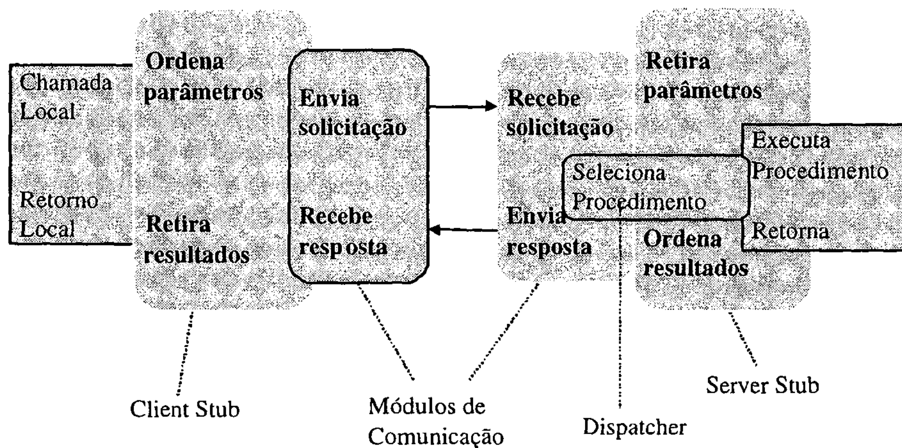


Figura 1.5: Funcionamento de um sistema RPC

do procedimento chamado e envia a mensagem ao servidor, bloqueando-se, em seguida, até que chegue a resposta. Quando esta chega ao lado servidor, o *dispatcher* identifica qual o procedimento está sendo solicitado e passa a tarefa ao *stub* servidor apropriado, o qual retira os parâmetros da mensagem (*unmarshalling*) e chama o procedimento servidor "real" da maneira usual, ou seja, colocando os parâmetros na pilha e passando o controle ao mesmo.

O procedimento servidor, então, executa o trabalho que lhe compete e retorna os resultados ao *stub* servidor, o qual compõe uma nova mensagem e a envia de volta ao cliente. O *stub* cliente irá recebê-la e fazer um retorno da maneira usual à rotina que invocou a RPC.

Observe-se que, durante todo esse processo, o cliente não tem a mínima idéia se o trabalho executado foi local eu remoto, da mesma forma que o servidor não sabe se a solicitação foi feita localmente ou através da rede. Nesse "desconhecimento" consiste toda a elegância do mecanismo de RPC, pois todos os detalhes relativos à comunicação ficam a cargo dos *stubs*.

Os *stubs* são gerados a partir da linguagem de definição de interface (IDL), a qual especifica em detalhes tudo o que se precisa saber para chamar um procedimento, como já foi visto. No lado cliente, as aplicações são compiladas não com os procedimentos reais mas sim com os *stubs* e, no lado servidor, há também um *stub* para cada procedimento definido, sendo que cada um deles recebe um identificador. O responsável pela geração desses procedimentos "falsos" é o compilador de interface, que o faz a partir da IDL.

Objetos distribuídos

A principal característica de um objeto é a de encapsular dados (o estado) e as operações para esses dados (os métodos). Esses métodos estão disponíveis para usuários que queiram acessá-lo através de uma interface. É importante compreender que um processo não deve acessar ou manipular o estado de um objeto a não ser através dos métodos disponíveis em uma interface.

A separação entre interfaces e objetos (que implementam essas interfaces) é crucial para sistemas distribuídos. Essa separação permite que uma interface possa estar em uma máquina enquanto o objeto reside em outra. É devido a essa possibilidade que surgiu o termo **objetos distribuídos** [Tanenbaum and Steen, 2002].

Quando um cliente se associa a um objeto distribuído, uma implementação da interface do objeto, chamada *proxy* é carregada dentro do espaço de endereço do cliente. Um *proxy* é análogo a um *stub* cliente no sistema RPC. O objeto reside em uma máquina servidora onde oferece as mesmas interfaces como na máquina cliente. As requisições do cliente que chegam em um servidor são passadas para um *skeleton* (na abordagem RPC, é o *stub* do lado servidor que executa a operação de *unmarshalling* na mensagem recebida). Esse mesmo *skeleton* é o responsável pela operação de *marshalling* quando as respostas forem retornadas para o cliente.

Uma diferença importante entre sistemas RPC e sistemas de objetos distribuídos é que este último oferece referências de objetos para todo o sistema. Através dessas referências é que um cliente poderá invocar métodos em um servidor. Tais referências de objetos podem ser trocadas por processos em diferentes máquinas através de, por exemplo, parâmetros de métodos. Em razão de esconder a implementação de uma referência de objeto, a transparência de distribuição é melhor que em sistemas baseados em RPC. Um fato importante a lembrar é que antes de qualquer chamada a métodos remotos, deve-se primeiro obter a referência do objeto. Dessa forma, existem duas possibilidades de "ligação" (*binding*) com objetos remotos: referências implícitas e referências explícitas.

As referências implícitas possibilitam que um cliente faça a invocação de métodos diretamente para o objeto referenciado sem a necessidade de nenhuma conversão (explícita) intermediária. Nesse tipo de referência, o cliente é transparentemente associado ao objeto remoto no momento que a referência é resolvida para o objeto atual. Com referências explícitas o cliente deve primeiro chamar uma função especial que "liga" ao objeto distribuído antes que possa invocar seus métodos. Exemplos desses dois tipos de referências são apresentados nas Figuras 1.6 e 1.7.

```
Distributed_Object* ref_obj;
ref_obj = new (obj_remoto);
ref_obj->metodo();
```

Figura 1.6: Referência implícita para um objeto remoto

Como já discutido, depois que um cliente obtém a referência de um objeto, podem ser feitas invocações de métodos remotos (*Remote Method Invocation - RMI*). Uma forma usual de fornecer suporte para RMI é especificar interfaces de objetos em uma linguagem de definição de interfaces, similar à abordagem seguida por RPCs. Essa abordagem de utilizar definições de interfaces predefinidas é geralmente chamada de invocação estática. Nesse tipo de invocação é

```
Distributed_Object ref_obj;
Local_Object* ptr_obj;
ref_obj = new (obj_remoto);
ptr_obj = bind (ref_obj);
ptr_obj->metodo();
```

Figura 1.7: Referência explícita para um objeto remoto

necessário que a interface do objeto seja conhecida quando a aplicação cliente é desenvolvida. Isto também implica que se a interface sofrer alterações, a aplicação cliente deve ser recompilada para fazer uso dessas alterações.

De forma alternativa a essa abordagem, existe a possibilidade de compor uma invocação de método remoto em tempo de execução. Nesse novo tipo de invocação (invocação dinâmica), uma aplicação cliente seleciona em tempo de execução o método que deve ser invocado em um objeto remoto. Um exemplo de invocação dinâmica seria:

```
invoke(objeto, metodo, parametro_entrada, parametro_saida);
```

Onde *objeto* identifica o objeto distribuído que possui o *metodo* que deve ser invocado, *parametro_entrada* é uma estrutura que contém os parâmetros de entrada e *parametro_saida* refere-se a estrutura de dados que deve armazenar os valores de saída.

Embora sejam mais complexas do ponto de vista de implementação, a utilização de invocações dinâmicas pode ser atrativa em situações onde se queira "descobrir" alterações em programas servidores (por exemplo, uma aplicação em um *browser* que exibe ao usuário os métodos existentes em uma interface) e não há possibilidade de enviar as interfaces e recompilar as aplicações clientes.

Capítulo 2

A Arquitetura CORBA

2.1 Introdução

No início, e ainda hoje para muitos tipos de situações, a principal alternativa para o desenvolvimento de aplicações distribuídas era a utilização de plataformas baseadas em passagem de mensagens, que incluíam as primitivas *send/receive.* Na década de 80, foi introduzida, por Birrel e Nelson, uma nova forma de se desenvolver aplicações cliente-servidor, que foram as chamadas a procedimentos remotos (RPCs), as quais generalizaram nos ambientes distribuídos o conceito de chamadas a subrotinas e procedimentos, utilizado na programação estruturada convencional.

Com o crescimento da utilização de orientação a objetos, também na década de 80, deu-se a junção entre os sistemas distribuídos e os objetos, dando origem àquilo que hoje se denomina *objetos distribuídos.* Essa forma de se desenvolver aplicações distribuídas parece bem natural, já que, na maioria das vezes, os serviços, em sistemas desse tipo, são como caixas pretas, ou seja, dos mesmos conhece-se apenas a funcionalidade, sem se preocupar com os detalhes de implementação, tal qual quando se usam objetos na programação convencional.

Neste capítulo, será abordada uma das mais conhecidas arquiteturas para desenvolvimento de aplicações baseadas em objetos distribuídos, a arquitetura CORBA.

2.2 A OMA

O CORBA (*Common Object Request Broker Architecture*) é um padrão promulgado pelo OMG (*Object Management Group*) para o desenvolvimento de aplicações distribuídas em ambientes heterogêneos, com a utilização de objetos distribuídos.

O OMG foi criado em 1989, através de uma associação de empresas da área de desenvolvimento de software, para tratar de problemas como a interoperablidade entre as aplicações distribuídas que estavam sendo desenvolvidas à época, através da adoção de padrões de desenvolvimento e de interação entre essas aplicações. No ano 2000, o OMG já contava com mais de 700 membros, dos mais diversos setores da indústria e é um centro ativo de produção de tecnologia de software distribuído.

Desde o início, havia a preocupação de que o CORBA fosse uma solução aberta para o desenvolvimento de aplicações distribuídas, que pudessem executar em qualquer máquina ou sistema operacional, escritas em qualquer linguagem, desde que, é claro, houvesse um mapeamento adequado para cada situação.

O CORBA deixa transparente às aplicações clientes a localização do objeto, a linguagem em que ele foi escrito, sua forma de implementação e em que tipo de plataforma ele executa, permitindo que o desenvolvimento das aplicações distribuídas se dê de forma semelhante ao das aplicações "monolíticas" tradicionais.

O padrão CORBA se insere dentro da OMA (*Object Management Architecture*), também definida pelo OMG, a qual se divide nos seguintes componentes (Figura 2.1):

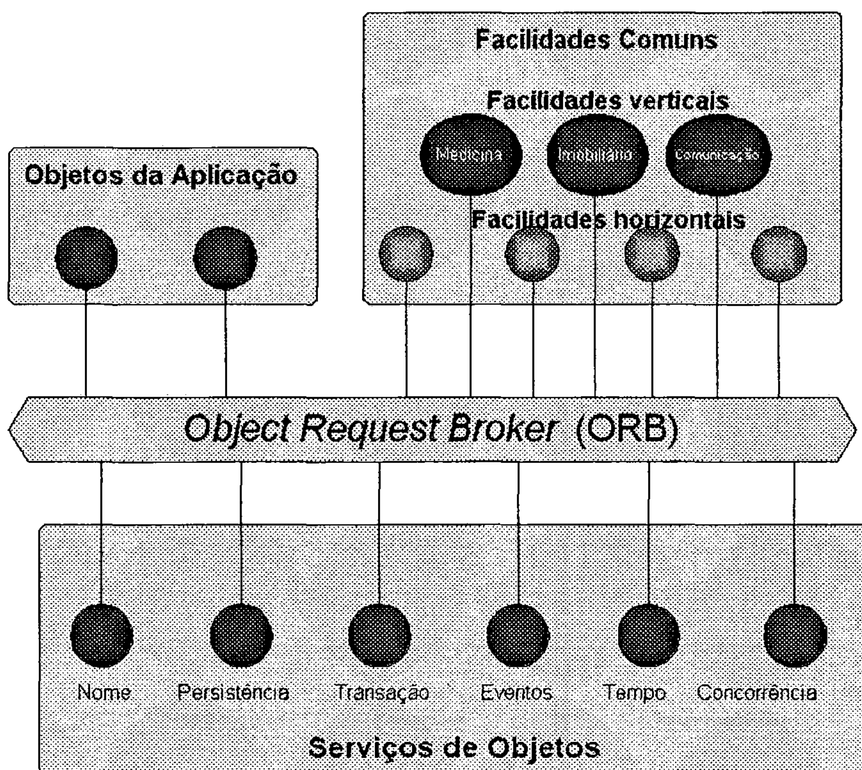


Figura 2.1: Arquitetura OMA

- ORB (*Object Request Broker*) : Intermedia a comunicação entre clientes e servidores. É o barramento de objetos.

- Serviços de Objetos : Serviços de sistema que complementam a funcionalidade do ORB. Ex: serviço de nomes, eventos, ciclo de vida, tempo.

- Facilidades Comuns: Similares ao item anterior, só que orientadas às aplicações. Podem ser de dois tipos:

- Horizontais: voltadas às aplicações de usuário final. Ex: documentos distribuídos, gerenciamento de sistemas, etc.
- Verticais: voltadas a domínios específicos (financeiro, telecomunicações, medicina). Também conhecidas por Interfaces de Domínio.

- Interfaces de Aplicações: Específicas das aplicações desenvolvidas pelos usuários.

2.3 Componentes da arquitetura CORBA

De forma mais detalhada, os componentes definidos na arquitetura CORBA estão organizados como apresentado na Figura 2.2.

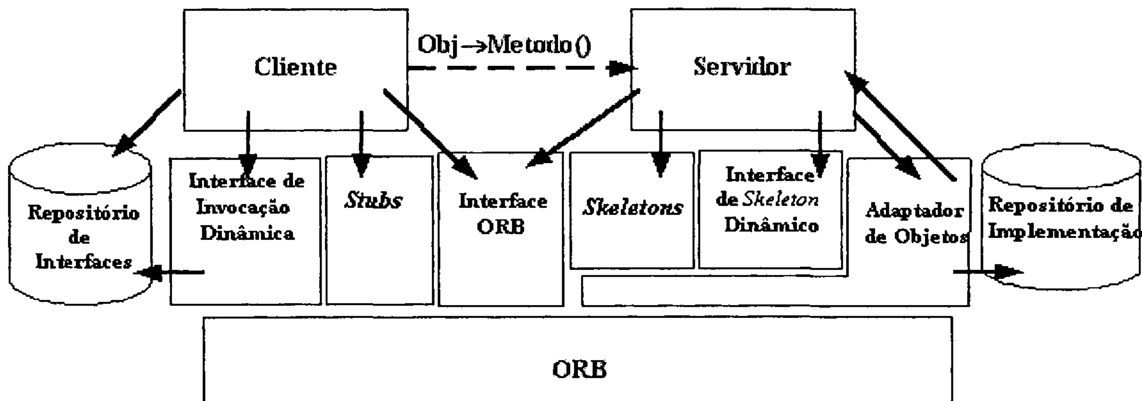


Figura 2.2: Arquitetura CORBA

A linha tracejada indica uma chamada a um método do objeto servidor, na perspectiva do objeto cliente. O objeto cliente invoca o método no servidor exatamente da mesma forma (isto é, com a mesma sintaxe) como faria se fosse uma chamada local, no entanto, o processo "real" ocorre no sentido: Cliente $\longrightarrow$ Interface de Invocação (Dinâmica ou Estática) $\longrightarrow$ ORB $\longrightarrow$ Adaptador de Objetos $\longrightarrow$ *Skeleton* (Dinâmico ou Estático) $\longrightarrow$ Servidor. Os elementos que compõe a arquitetura CORBA são apresentados nas seções a seguir.

2.3.1 O ORB

O ORB é a parte central da arquitetura CORBA. Constitui-se no "barramento virtual" por onde circulam as requisições aos objetos servidores e as respostas dos mesmos aos clientes. É ele que dá aos objetos a possibilidade de se comunicar de forma transparente, independente da distribuição (local ou remota). É importante frisar que, na arquitetura CORBA, cliente e servidor são papéis que se estabelecem em cada interação entre entidades, caracterizando um consumidor e um fornecedor de serviços, respectivamente.

O ORB apresenta várias vantagens em relação a outras formas de *middleware*. Algumas dessas vantagens são:

- Possuir vários serviços agregados;
- Fornecer suporte a diferentes formas de invocação de métodos, síncronas e assíncronas, estáticas e dinâmicas;
- Contemplar mapeamentos para as linguagens de programação mais utilizadas, orientadas ou não a objetos;
- Ser um sistema auto-descritivo;
- Pelo grau de transparência alcançado, em vários níveis;
- Por seus mecanismos de transações e segurança;
- Oferecimento de mensagens polimórficas.

2.3.2 Os *stubs*

Os *stubs* clientes são as interfaces estáticas para os serviços (objetos) e são os responsáveis por tratar dos detalhes da comunicação entre clientes e servidores. Para cada invocação feita a um método remoto, na verdade o cliente está chamando um representante do servidor no ambiente local, uma espécie de *proxy*, que vai compor uma mensagem com a identificação do método invocado e seus parâmetros e enviá-la ao objeto servidor, bloqueando-se localmente (em geral) até que a resposta à solicitação chegue. No retorno, o *stub* retira da mensagem os resultados e os devolve ao programa cliente, que tem a ilusão de que todo o processo se deu localmente. A utilização de *stubs* e *skeletons* permite a realização de invocações estáticas.

2.3.3 Os *Skeletons*

Os *skeletons* são a contrapartida dos *stubs* no ambiente servidor, fornecendo uma interface estática para cada serviço exportado. Ao chegar uma solicitação, o *skeleton* obtém a identificação do método invocado e retira os seus parâmetros da mensagem recebida, fazendo uma chamada local ao objeto servidor. Ao ser completada a solicitação, envia uma mensagem com os resultados ao cliente. É importante observar que durante este processo de solicitação de serviço e resposta, os objetos clientes e servidores se comportam como se tudo ocorresse localmente, pois os detalhes de comunicação ficam a cargo dos *stubs* e *skeletons*, que introduzem a transparência necessária neste esquema. Os mesmos são gerados a partir de uma especificação da intertace do serviço na linguagem IDL, sendo compilados com os módulos cliente e servidor da aplicação, respectivamente.

2.3.4 O Repositório de Interfaces

Todo elemento em um barramento CORBA deve ser auto-descritivo, ou seja, deve ser possível obter, em tempo de execução, informações a respeito de sua interface e a maneira de chamar os seus métodos. Assim, ao se gerar os *stubs* e *skeletons* a partir da descrição do serviço em IDL, geralmente existe a opção de se armazenar a descrição do objeto servidor no Repositório de Interfaces (IR). Este se constitui em uma base de dados dinâmica com informações sobre as interfaces de todos os serviços conhecidos pelo ORB. O IR é geralmente usado quando se deseja fazer invocações dinâmicas.

2.3.5 Interface de Invocação Dinâmica

Embora a invocação estática de métodos, apresente algumas vantagens como a facilidade de programação, melhor desempenho, verificação de tipos em tempo de compilação, auto-documentação, dentre outras, ela exige que se conheçam as particularidades dos objetos invocados em tempo de compilação, o que nem sempre é possível.

Muitas vezes, somente no decorrer da execução de um programa, é possível saber, com certeza, quais são os objetos que se deseja invocar, quais os métodos necessários, os parâmetros que serão passados, etc. Assim, deve existir um mecanismo de invocação dinâmica de métodos em CORBA, que é a *Dynamic Invocation Interface* (DII). Esta permite que o cliente descubra, em tempo de execução, os detalhes de invocação do serviço e construa sua chamada dinamicamente. Os dados sobre o objeto remoto são obtidos a partir da base de dados do Repositório de Interfaces.

A invocação dinâmica permite também que se realizem tipos de chamadas especiais aos objetos, como as assíncronas (também possível com invocações estáticas), síncronas adiadas e *callbacks*.

2.3.6 Interface de *Skeleton* Dinâmica

É a parte correspondente à DII, no lado servidor, permitindo que os servidores sejam escritos sem a necessidade de *skeletons* previamente compilados no código do programa.

A principal vantagem de utilização da *Dynamic Skeleton Interface* (DSI) é na implementação de pontes entre ORBs e também para fazer a comunicação entre CORBA e outras plataformas de computação distribuída. Neste último caso, assim que a outra plataforma de comunicação seja considerada relevante o suficiente, no contexto do desenvolvimento de aplicações distribuídas, geralmente é padronizada a forma de interoperabilidade entre a mesma e o padrão CORBA, haja vista os casos do DCE/ESIOP, CORBA/DCOM e CORBA/RMI.

2.3.7 Repositório de Implementação

O Repositório de Implementação é um repositório, em tempo de execução, que armazena informações sobre as classes que um servidor suporta, os objetos que estão instanciados e suas identificações. É a partir dele que se criam novas instâncias de objetos para atender os clientes. O Adaptador de objetos comumente utiliza os serviços do repositório de implementação.

2.3.8 Adaptador de Objetos

O Adaptador de Objetos é um dos principais elementos no lado servidor de um ORB. É ele que fornece a ligação entre o ORB e as implementações dos objetos no servidor, garantindo uma maior portabilidade das aplicações e um melhor balanceamento de carga. As principais funções do adaptador de objetos são:

- Registrar as classes servidoras no Repositório de Implementação;
- Ativar (instanciar) os objetos chamados, em tempo de execução, segundo a demanda dos clientes;
- Receber as chamadas para os objetos e repassá-las aos mesmos (invocação de métodos);
- Gerar e gerenciar as referências de objetos (ORs).

O CORBA determina que todo ORB deve possuir pelo menos um BOA (*Basic Object Adapter*) e, normalmente, existe um adaptador de objetos para cada linguagem diferente suportada pelo ORB.

2.3.9 Interface do ORB

Um ORB é uma entidade de software e, portanto, possui uma interface (API) para que os elementos conectados a ele possam chamar seus serviços, de uso comum às aplicações. Geralmente, são funções para inicializar o ORB, inicializar o Adaptador de Objetos, para conversão de *strings* em referências de objetos, dentre outras.

2.4 Interoperabilidade entre ORBs

Conforme a complexidade do problema a ser tratado, faz-se necessária a comunicação entre objetos que podem estar separados por longas distâncias, possivelmente por várias redes, fazendo com que a invocação de um método passe por diversos ORBs até chegar ao objeto que execute a requisição. Com isso, um ORB deve comunicar-se com outro ORB para que a mensagem possa ser enviada de um para outro.

Na primeira versão do padrão, o CORBA 1.1, promulgada em 1991, a principal preocupação do OMG era garantir a portabilidade das aplicações, ou seja, aplicações escritas para um

certo ORB deveriam ser facilmente transportadas para outro ORB, com um mínimo de esforço. Porém, cada ORB prosseguia sendo um mundo isolado, já que aplicações que não estivessem conectadas a ele não poderiam se utilizar dos objetos conhecidos pelo mesmo.

O CORBA 2.0 (1994) resolveu este problema, pois se preocupava em garantir a interoperabilidade entre ORBs de diferentes fornecedores. A partir de então, aplicações clientes conectadas a um ORB podiam, de forma fácil e transparente, fazer uso de objetos servidores localizados em ORBs distintos. O trabalho de transporte das solicitações e resultados entre os ORBs fica a cargo da plataforma CORBA.

A solução utilizada foi a criação de um *General Inter-ORB Protocol* (GIOP), que especifica uma representação comum e um conjunto de mensagens padrão para troca de informações entre ORBs, sobre um protocolo orientado à conexão. Este protocolo tem propósitos genéricos e deve ser particularizado para cada solução efetivamente utilizada na rede.

Para o caso de uma rede TCP/IP, foi definido o *Internet Inter-ORB Protocol* (IIOP), que determina como o GIOP funciona sobre o protocolo TCP. Ele é a base para a comunicação entre aplicações CORBA em redes IP, sendo esta solução por vezes chamada de CORBA/IIOP.

A partir do CORBA 2.0, tornou-se possível criar Federações de ORBs, com vários subdomínios de aplicações interligados de forma hierárquica, cada um deles fornecendo serviços relativos a uma área específica ou restritos a um departamento ou unidade de uma organização.

2.5 Referências de Objetos

Cada objeto em um barramento CORBA deve ser identificado de forma única, de modo a evitar ambigüidades na sua invocação. Dessa forma, todo objeto possui um identificador único, chamado de *Object Reference* (OR), que é a chave para acessá-lo dentro de um dado domínio CORBA. Essas referências são como ponteiros para os objetos e são geradas pelo *Object Adapter* quando o mesmo instancia os objetos servidores a partir das descrições de suas classes armazenadas no Repositório de Implementação. Embora a forma de representação interna das ORs fique a cargo da implementação de cada ORB, a experiência tem mostrado que, de forma geral, ORBs adotam o formato apresentado na Figura 2.3 para uma referência de objeto.

Em geral, no início de uma sessão de interação entre um cliente e um servidor, há uma fase em que o cliente precisa obter uma referência para um objeto que implemente o serviço desejado, a qual pode ser conseguida a partir de arquivos, serviços de diretórios, de nomes ou do Repositório de Interfaces.

No CORBA 2.0 foram definidas as *Interoperable Object References* (IORs), que são identificadores universais para os objetos, válidos entre ORBs distintos.

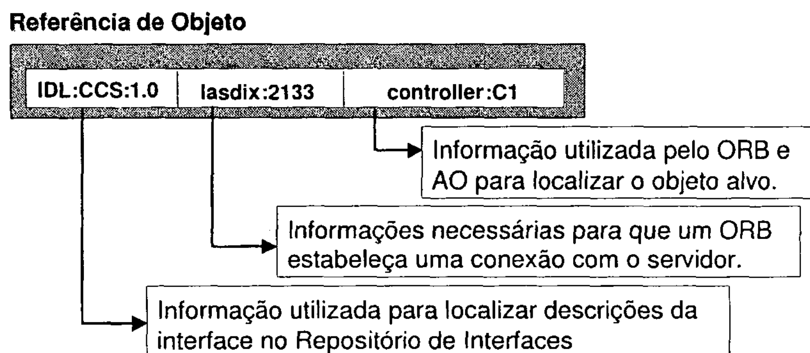


Figura 2.3: Exemplo de uma Referência de Objeto

2.6 A IDL e os Metadados em CORBA

A IDL (*Interface Definition Language*) é usada para especificar a interface dos objetos servidores (dos serviços). Apesar de possuir uma sintaxe parecida com C++, é uma linguagem puramente declarativa, ou seja, não especifica nenhum detalhe de implementação. Assim, através da IDL definem-se quais os métodos disponíveis no servidor, seus parâmetros e tipos, valores de retorno, exceções, herança, atributos de classe e resposta a eventos. A IDL estabelece uma espécie de "contrato" entre o servidor e seus clientes em potencial. Desde que não se mude a interface do serviço, a implementação do mesmo pode ser feita da forma que se achar mais conveniente.

É a partir da especificação da interface do servidor em IDL que são gerados os *stubs* e *skeletons*. Além disso, o resultado da compilação da especificação em IDL pode ser armazenado mo Repositório de Interfaces, permitindo que:

1. Os componentes do sistema "descubram-se" em tempo de execução e possam interoperar;
2. Os clientes sejam escritos sem se preocupar com os servidores que vão ser utilizados;
3. Haja a separação entre interface e implementação;
4. Se faça a comunicação com sistemas mais antigos, não necessariamente orientados a objetos.

Em suma, cria-se um repositório de metadados dinâmicos para o ORB, que contém detalhes sobre todos os objetos conhecidos naquele domínio.

2.6.1 Estrutura da IDL

A IDL tem sintaxe semelhante à especificação de classes em C++, acrescida de mais algumas palavras-chave relativas a conceitos de sistemas distribuídos. Uma especificação em IDL segue a seguinte estrutura (alguns elementos são opcionais):

- Módulo (*module*) - É um *namespace* que agrupa um conjunto de descrições de classes (interfaces).

- Interface - Define um conjunto de métodos (operações) que um cliente pode invocar em um objeto. Uma interface pode: possuir atributos, constantes e tipos pré-definidos; declarar exceções; ser derivada de uma ou mais interfaces (herança).

- Operações - São o equivalente em CORBA a um método. A IDL define a assinatura de uma operação, ou seja, seus parâmetros, tipos e valor de retorno, sendo possível, ainda, especificar se um parâmetro é de entrada, saída ou ambos.

- Tipos de dados:

  - Básicos: *short, long, unsigned, float, double, char, boolean, octet.*
  - Construídos: *string, array, sequence, struct, union, enum, any.*

Na Figura 2.4 é apresentado um exemplo da definição de um módulo com interfaces especificadas em IDL.

```
module Banco
{
        interface Conta{
                op1();
                op2();
        };
        interface Cliente{
                op1();
                op2();
                op3();
        };

};
```

Figura 2.4: Exemplo de IDL utilizando módulos e interfaces

Na Figura 2.5 tem-se a definição de tipos pelo usuário, assim como a criação de *structs*, atributos e métodos com parâmetros de entrada (palavra chave *in*) e valor de retorno.

Para uma completa associação entre a IDL (utilizada para especificar as interfaces) e uma linguagem de programação (utilizada para implementar o cliente e o servidor), deve haver um mapeamento (*language mapping*) da IDL para essa linguagem de programação. Esse mapeamento é conduzido pelo compilador IDL.

2.7 Serviços CORBA

Agregados a um ORB, estão definidos diversos serviços que complementam sua funcionalidade e facilitam o desenvolvimento das aplicações. Os mesmos são brevemente comentados a seguir.

```
typedef float Preco;

struct Lugar {
 char fila;
 unsigned long assento;
};

interface Bilheteria {
 readonly attribute string nomeCinema;
 readonly attribute char ultimaFila;
 Preco obtemPreco ( in Lugar Res, in string dataRes);
 Reserva fazRes( in Lugar lRes, in string dRes, in string Cart);
};
```

Figura 2.5: Exemplo de IDL utilizando tipos definidos pelo usuário e métodos com parâmetros de entrada

- Nomes: Permite que os componentes localizem-se pelo nome. Os objetos podem ser acoplados a serviços como X.500, OSF DCE e NIS. De forma mais específica, possibilita que nomes sejam associados a referências de objetos remotos dentro de contextos de nomes (*naming contexts*). Contexto de nomes é o escopo dentro do qual um conjunto de nomes são agrupados e devem ser únicos. Programas clientes e servidores requisitam o contexto de nomes inicial do ORB, através da invocação do método *resolve_initial_references* com o argumento *"NameService"*. O ORB retorna uma referência para um objeto do tipo *NamingContext*. Esse objeto refere-se para o contexto inicial do servidor de nomes para esse ORB. Pelo fato de que podem existir diversos contextos, objetos não têm nomes absolutos. Nomes são sempre interpretados relativos para um contexto de nomes inicial.

  Aplicações clientes utilizam o método *resolve* para consultar referências de objetos pelo nome. O valor de retorno é do tipo *Object*, de forma que possa retornar referências para qualquer tipo de objeto pertencente às aplicações. O resultado deve ser convertido antes de utilizado.

  Servidores utilizam a operação *bind* para registrar seus objetos em um serviço de nomes e *unbind* para removê-los. A Figura 2.6 apresenta um exemplo de utilização de um serviço de nomes por parte de um servidor (que registra seus objetos) e um cliente (que utiliza o método *resolve* para buscar referências de objetos remotos.

- Eventos: Permite que os objetos registrem seu interesse em eventos específicos, viabilizando uma maneira alternativa de comunicação entre os objetos (acoplamento fraco), não necessariamente do tipo *request-reply*. As notificações são comunicadas como argumentos ou resultados de invocações de métodos síncronos. Notificações podem ser propagadas através da técnica *pushed* (do fornecedor para o consumidor) ou *pulled* pelo consumidor. No primeiro caso, o consumidor implementa a interface *PushConsumer* que inclui um

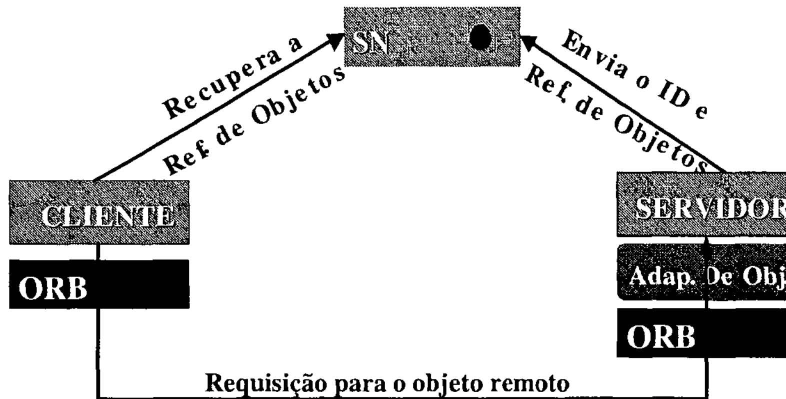


Figura 2.6: Exemplo de um serviço de nomes

método *push*. Consumidores registram sua referência de objeto remoto com os fornecedores. O fornecedor invoca o método *push*, indicando como argumento uma notificação de evento. No segundo caso, o fornecedor implementa a interface *PullSupplier* que inclui um método *pull*. Fornecedores registram suas referências de objetos remotos com os consumidores. Os consumidores invocam o método *pull* e recebem uma notificação como resultado.

- Ciclo de Vida: Define operações para a criação, destruição, cópia e movimentação de objetos no ORB.

- Trader: Permite que um objeto descubra outro baseado em suas características, assim como um objeto também pode anunciar suas funcionalidades e cobrar pelo seu uso.

- Persistência: Fornece uma interface única para o armazenamento de objetos em arquivos comuns, em bancos de dados relacionais ou em bancos de dados orientados a objetos.

- Query: Permite realizar consultas a objetos, através de um superconjunto da SQL: a OQL (*Object Query Language*), do ODMG (*Object Data Management Group*).

- Transações e Controle de Concorrência: Viabiliza a existência de transações distribuídas entre objetos, utilizando o protocolo de *commit* de duas fases. O cliente especifica uma transação como uma seqüência de chamadas RMI que são introduzidas por *begin* e terminadas por *commit* ou *rollback*. O ORB "anexa" um identificador de transação para cada invocação remota e associa isso com requisições *begin*, *commit roolback*. Clientes podem também suspender e retornar transações. O serviço de controle de concorrência utiliza *locks* para controlar o acesso a objetos CORBA. Pode ser utilizado de dentro de transações ou independentemente.

- Relacionamento: Cria relacionamentos entre componentes quaisquer, permitindo a implementação de integridade referencial, *containment relationships* e outros tipos.
- Externalização: Permite converter dados de/para um objeto em uma forma linear (*stream*).
- *Licensing*: Faz a contabilização de uso de um componente, em vários níveis.
- Propriedades: Permite associar propriedades (valores nomeados) a um componente. Por exemplo: um rótulo, uma data.
- Tempo: Permite a sincronização do tempo em um ambiente distribuído, além da definição de eventos dependentes de tempo.
- Segurança: Define um modelo de segurança completo para objetos distribuídos, com suporte à autenticação, controle de acesso, confidencialidade e não-repúdio. Para garantir que a segurança é aplicada corretamente, o serviço de segurança requer a cooperação do ORB. Para fazer uma invocação segura de métodos remotos, as credenciais do cliente são enviadas na mensagem de requisição. Quando o servidor receber essa mensagem, irá verificá-la para se certificar de que as credenciais são válidas e assinadas por uma autoridade aceitável. Essa decisão é tomada através de uma consulta a um objeto contendo informações sobre quem tem permissão para acessar cada método do objeto alvo. Se tiver direitos suficientes, a invocação é transportada e o resultado é retornado para o cliente. O serviço de segurança permite que usuários adquiram suas credenciais e privilégios como resposta ao fornecimento de dados de autenticação (por exemplo, uma senha).

Capítulo 3

Exemplos de Aplicações

Neste capítulo, serão apresentados dois exemplos de aplicações CORBA, usando duas ferramentas distintas. O exemplo *Agenda*, Seção 3.2, foi desenvolvido usando a implementação ORBacus, tendo sido testado nas plataformas Windows e Linux. O exemplo do *Caixeiro Viajante*, Seção 3.3, foi desenvolvido com o uso da ferramenta TAO, em ambiente Linux.

Existem inúmeras implementações CORBA disponíveis atualmente, comerciais e de domínio público, proprietárias e de código aberto, apresentando diferentes graus de compatibilidade com as especificações promulgadas pelo OMG. Cada uma dessas implementações possui suas particularidades, portanto é aconselhável a leitura de um manual antes de se desenvolver uma primeira aplicação usando uma ferramenta com a qual não se está familiarizado.

Independentemente da ferramenta utilizada, ao se desenvolver uma aplicação CORBA devem-se cumprir, em geral, os seguintes passos:

- Descrever a interface do servidor usando a IDL;
- Compilar a interface a fim de gerar os *stubs* e *skeletons*;
- Fornecer uma implementação para cada método definido na interface descrita em IDL;
- Escrever o programa cliente, o qual será constituído basicamente da interface com o usuário e de procedimentos de conexão com o ORB e o programa servidor;
- Escrever o programa servidor, o qual será responsável por conectar-se com o ORB e instanciar as classes definidas.

3.1 Implementações CORBA

3.1.1 ORBacus

O ORBacus é uma implementação CORBA, atualmente de propriedade da IONA Technologies, que é distribuído com seu código fonte, de modo que seus usuários possam estendê-lo e persona-

lizá-lo segundo suas necessidades. Possui mapeamento para as linguagens C++ e Java e opera em diferentes plataformas, entre elas Solaris, HP-UX, Windows NT/2000, Linux e IBM AIX.

A versão 4.1 é totalmente compatível com a especificação CORBA 2.4 e dá suporte a aspectos avançados como o *Portable Object Adapter* (POA) e *Objects by Value*. Atualmente, implementa os seguintes serviços CORBA: Nomes, Eventos, Propriedades, Tempo, *Trader* e Notificação.

O ORBacus não usa *daemons*, sendo necessário apenas compilar os clientes e servidores com as bibliotecas apropriadas. Outras características interessantes são: completo suporte a CORBA IDL, Repositório de Interfaces (IR), *Portable Interceptors*, modelo de concorrência baseado em uma ou múltiplas *threads*, extensões para tolerância a falhas, Interface de Invocação Dinâmica (DII), Interface de *Skeleton* Dinâmica (DSI), suporte ao tipo *Any* e operação com outros protocolos além do IP, tais como: SSL, ATM e IP multicast.

3.1.2 TAO (*The ACE ORB*)

TAO (*The ACE Orb*) é uma implementação CORBA desenvolvida pelo DOC (*Distributed Object Computing*) *Group* da Universidade de Washington. O desenvolvimento dessa implementação CORBA é baseada no *framework* orientado a objetos ACE (*Adaptive Communication Environment*) que, basicamente, implementa os recursos de comunicação necessários para o desenvolvimento de aplicações concorrentes. De maneira geral, ACE provê os recursos de comunicação que podem ser adaptados para diferentes Sistemas Operacionais e arquiteturas de hardware. Para alcançar esse objetivo, suas funcionalidades foram organizadas em classes e divididas em camadas possibilitando, assim, que extensões possam ser efetivadas sem causar detrimento às características já existentes.

A arquitetura do *framework* ACE juntamente com a estrutura da implementação TAO pode ser vista na Figura 3.1.

Conforme pode ser observado na Figura 3.1, além de utilizar o *framework* ACE, TAO implementa o modelo de programação definido pela especificação CORBA através de *stubs* e *skeletons*, além de acrescentar novas funcionalidades visando unir questões como melhor desempenho e qualidade de serviço para aplicações distribuídas. Alguns dos fatores que nortearam o desenvolvimento dessa ferramenta compreendem:

- Fornecer uma implementação gratuita, com código aberto, que possa ser usada por pesquisadores e desenvolvedores;
- Determinar empiricamente as características necessárias para permitir que implementações CORBA tenham suporte para o desenvolvimento de aplicações de missão crítica com qualidade de serviços determinísticos e estatísticos;
- Utilizar padrões de projeto para desenvolver implementações que sejam portáteis, extensíveis, e com garantias de qualidade de serviço;

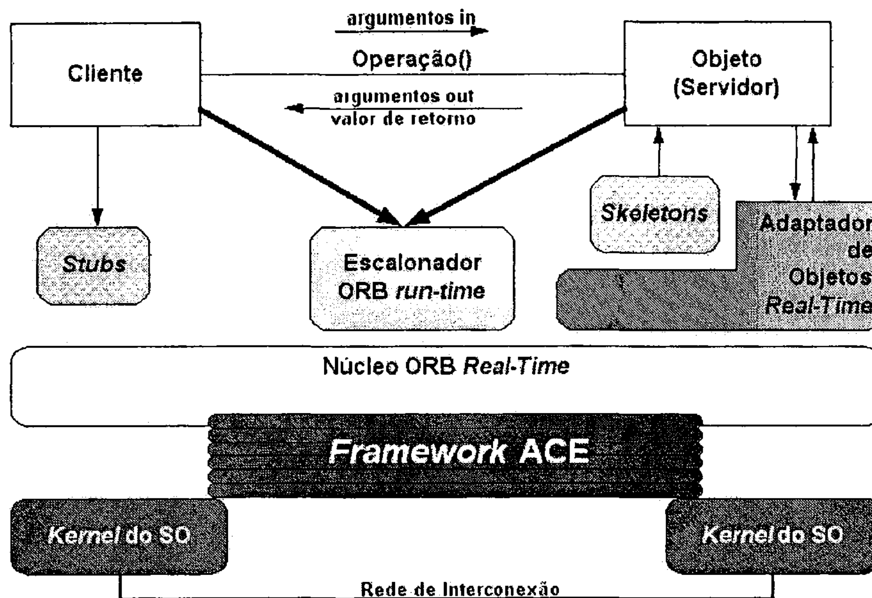


Figura 3.1: Estrutura básica da implementação TAO

- Auxiliar na especificação do padrão *Real-Time* CORBA.

Por ser direcionada principalmente para aplicações de tempo real, em que desempenho e confiabilidade são dois fatores de extrema importância, a implementação TAO tem lançado mão de diversas técnicas e mecanismos na tentativa de melhorar o desempenho das aplicações sem perder a confiabilidade. Um desses mecanismos refere-se às invocações assíncronas através dos modelos *polling* e *callback* da especificação CORBA *Messaging* [OMG, 2001]. Embora não tenha ainda implementado o modelo *polling*, TAO já possui suporte para o modelo *callback*, além dos mecanismos de chamadas unidirecionais (*oneway*) e retardo síncrono (*deferred synchronous*).

Os exemplos apresentados na seção 3.3 utilizam a ferramenta TAO e foram implementados usando a versão 1.1 no sistema operacional Linux (*kernel* 2.2.16). Essa versão contempla os recursos indicados na especificação CORBA 2.3.

Compilando aplicações com TAO

De forma similar ao que acontece com qualquer aplicação CORBA, deve-se inicialmente, definir a interface que será utilizada pelo cliente e implementado pelo servidor. Com isso, é necessário uma ferramenta para compilar essa interface e gerar os *stubs* e *skeletons*. No caso de TAO, o compilador de interfaces é denominado *tao_idl* e sua sintaxe é apresentada a seguir:

```
tao_idl  <options> interface.idl
```

Na versão aqui adotada, o compilador *tao_idl* está em *ACE_wrappers/TAO/TAO_idl*.

Uma listagem das opções que podem ser utilizadas na ferramenta tao_idl pode ser obtida especificando:

```
tao_idl --help
```

Após a compilação da interface, pode-se especificar o cliente e servidor e compilá-los para que os arquivos executáveis sejam gerados. Essa etapa pode ser realizada a partir do utilitário *make* e arquivos de *Makefile* uma vez que a utilização desses arquivos facilitará o processo de compilação. A sintaxe básica para compilação do código fonte e geração de arquivos executáveis é a seguinte:

```
make
```

A Figura 3.2 apresenta um exemplo de um arquivo de *Makefile* levando-se em consideração que o arquivo interface.idl já foi compilado pelo utilitário *tao_idl*.

3.2 O exemplo Agenda

Nesta seção, apresenta-se uma agenda telefônica muito simples desenvolvida em Java e CORBA. Embora tenha uma funcionalidade mínima, o exemplo nos permite acompanhar os detalhes de construção de uma aplicação básica usando a ferramenta ORBacus, versão 4.0.5. A aplicação constitui-se de um cliente CORBA, que possui uma interface com o usuário em modo caracter e invoca operações em um objeto servidor, o qual implementa a agenda em memória principal.

Como em qualquer aplicação CORBA, o passo inicial consiste em descrever a interface do servidor em IDL (Figura 3.3). A interface Agenda define as seguintes operações: consultaNome, insereItem, statusMem e listaItens, respectivamente para procurar um nome na agenda, inserir uma nova entrada, obter o número de nomes corrente e listar os nomes da agenda.

Para compilar a interface, usa-se o jidl, o tradutor de IDL para Java fornecido com o ORBacus:

```
jidl --package agenda Agenda.idl
```

Deve-se observar o uso das letras maiúsculas e minúsculas no comando acima, pois os programas estão escritos em Java. A chamada de um arquivo "agenda.idl", com "a" minúsculo, causaria problemas.

O compilador jidl irá criar um subdiretório "agenda" no diretório corrente e nele gerar diversos programas fonte em Java, os quais serão posteriormente compilados com os programas cliente e servidor fornecidos pelo programador. Neste caso, os programas gerados são: AgendaHelper.java, AgendaHolder.java, AgendaOperations.java, AgendaPOA.java e _AgendaStub.java.

Concluída esta etapa, passa-se para a implementação dos programas cliente e servidor. No caso do programa cliente, segue-se o roteiro:

```
#--------------------------------------------------------------------------
#       Local macros
#--------------------------------------------------------------------------

ifndef TAO_ROOT
  TAO_ROOT = $(ACE_ROOT)/TAO
endif # ! TAO_ROOT

LDLIBS = -lTAO -lorbsvcs

IDLFILES = interfaceC interfaceS
BIN = clientinterface serverinterface
INTERFACEC = interfaceC
INTERFACES = interfaceS

SRC = $(addsuffix .cpp, $(BIN) $(IDLFILES))

CLIENT_OBJS = clientinterface.o $(addsuffix .o, $(IDLFILES))
SERVER_OBJS = serverinterface.o $(addsuffix .o, $(IDLFILES))

TAO_IDLFLAGS += -Ge 1 -GC -GI

#--------------------------------------------------------------------------
#       Include macros and targets
#--------------------------------------------------------------------------

include $(ACE_ROOT)/include/makeinclude/wrapper_macros.GNU
include $(ACE_ROOT)/include/makeinclude/macros.GNU
include $(TAO_ROOT)/rules.tao.GNU
include $(ACE_ROOT)/include/makeinclude/rules.common.GNU
include $(ACE_ROOT)/include/makeinclude/rules.nonested.GNU
include $(ACE_ROOT)/include/makeinclude/rules.local.GNU
include $(TAO_ROOT)/taoconfig.mk

#--------------------------------------------------------------------------
#       Local targets
#--------------------------------------------------------------------------

LDFLAGS += -L$(TAO_ROOT)/orbsvcs/orbsvcs -L$(TAO_ROOT)/tao
CPPFLAGS += -I$(TAO_ROOT)/orbsvcs

.PRECIOUS: interfaceC.h interfaceC.i interfaceC.cpp
.PRECIOUS: interfaceS.h interfaceS.i interfaceS.cpp
.PRECIOUS: interfaceS_T.h interfaceS_T.i interfaceS_T.cpp

serverinterface: $(addprefix $(VDIR),$(SERVER_OBJS))
	$(LINK.cc) $(LDFLAGS) -o $@ $^ $(VLDLIBS) $(POSTLINK)

clientinterface: $(addprefix $(VDIR),$(CLIENT_OBJS))
	$(LINK.cc) $(LDFLAGS) -o $@ $^ $(VLDLIBS) $(POSTLINK)

realclean: clean
	-$(RM) core interfaceC.o interfaceS.o interfaceS_T.o
```

Figura 3.2: Exemplo de um arquivo de *makefile*

1. Ajustar algumas propriedades do ORB, necessárias para o uso do ORBacus com o JDK 1.2 ou posterior;

2. Inicializar o ORB;

3. Tratar quaisquer exceções que possam surgir;

4. Obter uma referência para um objeto da classe implementada no servidor e definida em IDL. Neste caso, a referência é lida a partir de um arquivo; em um cenário mais realista, poderia ser obtida a partir do Serviço de Nomes;

5. Utilizando a referência obtida, inicializar um objeto local para representar o objeto servidor remoto;

```
interface Agenda
{
   string consultaNome(in string nome);
   boolean insereItem(in string nome, in string fone);
   long statusMem();
   void listaItens();
};
```

Figura 3.3: Interface IDL para a aplicação Agenda

6. Incluir o código necessário para a interação com o usuário e a execução da lógica da aplicação cliente.

Os itens 1 a 3 fazem parte da função main() do programa cliente e são exemplificados na Figura 3.4. Os itens 4 a 6, mostrados na Figura 3.5, estão codificados no método run(), que é chamado a partir da função main(). Em termos de implementação, todo este código está contido em único arquivo (Agenda_cli.java), que corresponde ao programa cliente da aplicação CORBA.

A interface com o usuário fornecida pela agenda telefônica é a mais simples possível, em modo caracter. A chamada dos métodos remotos no servidor ocorre da maneira convencional em Java, como se os mesmos tivessem sido implementados localmente; todos os detalhes de comunicação são tratados pelo ORB. O programa cliente poderia também ser um *applet* Java ou ter uma interface gráfica baseada em AWT, sendo necessário para isso fazer algumas modificações no programa relativas à linguagem Java, mas não à utilização do ORBacus.

Quanto à implementação do programa servidor, devem-se seguir os passos:

1. Ajustar as propriedades do ORB;

2. Inicializar o ORB;

3. Tratar quaisquer exceções que possam surgir;

4. Obter uma referência para o Adaptador de Objetos (POA);

5. Criar uma instância do objeto servidor que implementa a interface (classe) definida em IDL;

6. Gravar a referência ao objeto servidor em um arquivo; em um cenário mais realista, a mesma seria registrada no Serviço de Nomes (seção 3.3).

7. Aguardar as requisições dos clientes.

```
package agenda;
import java.io.*;

public class Agenda_cli
{
    public static void main(String args[])
    {
      java.util.Properties props = System.getProperties();
      props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
      props.put("org.omg.CORBA.ORBSingletonClass",
                  "com.ooc.CORBA.ORBSingleton");

      int status = 0;
      org.omg.CORBA.ORB orb = null;
      try
      {
            orb = org.omg.CORBA.ORB.init(args, props);
            status = run(orb, args);
      }
      catch(Exception ex)
      {
          ex.printStackTrace();
          status = 1;
      }
      if(orb != null)
      {
          try
          {
            ((com.ooc.CORBA.ORB)orb).destroy();
          }
          catch(Exception ex)
          {
            ex.printStackTrace();
            status = 1;
          }
      }
      System.exit(status);
    }
    . . .
```

Figura 3.4: Programa cliente em ORBacus (Parte I)

Os itens 1 a 3 são os mesmos do programa cliente, não sendo necessária nenhuma alteração na função main() (Figura 3.4). Os itens 4 a 7 estão implementados no método run(), como mostrado na Figura 3.6. Todo o código do programa servidor está contido em único arquivo (Agenda_srv.java).

Finalmente, é preciso fornecer uma implementação para cada um dos métodos definidos na interface descrita em IDL, como exemplificado na Figura 3.7. A implementação da classe Agenda, contida no arquivo Agenda_impl.java, é igual à de qualquer classe em Java, com a ressalva que a mesma deve ser descendente da classe AgendaPOA, gerada pelo compilador jidl. O programador é livre para implementar os métodos da maneira que desejar, desde que não altere a interface definida em Agenda.idl.

Para compilar os programas cliente e servidor, bem como o arquivo com a implementação

```
static int run(org.omg.CORBA.ORB orb, String[] args)
{
   org.omg.CORBA.Object obj = null;
   try
   {
      String refFile = "Agenda.ref";
      BufferedReader in = new BufferedReader(new FileReader(refFile));
      String ref = in.readLine();
      obj = orb.string_to_object(ref);
   }
   catch(...)

   Agenda agenda = AgendaHelper.narrow(obj);

   System.out.println(">>> AGENDA TELEFONICA <<<");
   String input, nome, fone;
   boolean res;
   do
   {
      System.out.println("\n'1' Consultar, '2' Inserir, " +
           "'3' Status, '4' Listar ou '.' Sair:");
      System.out.print("> ");
      DataInputStream dataIn =
         new DataInputStream(System.in);
      BufferedReader in = new BufferedReader(
         new InputStreamReader(dataIn));
      input = in.readLine();

      // Consultar
      if(input.equals("1")) {
      System.out.print(">> Nome: ");
      nome = in.readLine();

      fone = agenda.consultaNome(nome);

      if (fone.equals("0"))
         System.out.println("Nao encontrado!");
      else
         System.out.println("O telefone e': " + fone);

      // Outros métodos
      . . .

    } while(!input.equals("."));
    return 0;
}
```

Figura 3.5: Programa cliente em ORBacus (Parte II)

da classe Agenda, usa-se o compilador javac:

```
javac agenda/*.java
```

Para executar o cliente e o servidor, faz-se:

```
java agenda.Agenda_cli
java agenda.Agenda_srv
```

```
static int run(org.omg.CORBA.ORB orb, String[] args)
           throws org.omg.CORBA.UserException
{
   org.omg.PortableServer.POA rootPOA =
     org.omg.PortableServer.POAHelper.narrow(
       orb.resolve_initial_references("RootPOA"));

   org.omg.PortableServer.POAManager manager =
      rootPOA.the_POAManager();

   Agenda_impl agendaImpl = new Agenda_impl();
   Agenda agenda = agendaImpl._this(orb);

   try
   {
      String ref = orb.object_to_string(agenda);
      String refFile = "Agenda.ref";
      java.io.FileOutputStream file =
         new java.io.FileOutputStream(refFile);
      java.io.PrintWriter out = new java.io.PrintWriter(file);
      out.println(ref);
      out.close();
   }
   catch(...)

   manager.activate();
   orb.run();
   return 0;
}
```

Figura 3.6: Programa servidor em ORBacus

3.3 O exemplo Caixeiro Viajante com *callback*

O problema do Caixeiro Viajante (*Travelling Salesman*) é comumente encontrado na literatura em razão de ser exemplificado para diversos domínios de aplicação e pertencer à classe de problemas $NP - completo$. De maneira geral, esse problema consiste em determinar a melhor rota (levando-se em consideração algum critério, por exemplo, tempo ou distância) que um viajante deve percorrer de maneira que visite todas as cidades de sua rota e ainda consiga retornar à cidade de onde partiu inicialmente.

Uma abordagem para resolução do PCV consiste em utilizar a programação dinâmica [Terada, 1991], visto que essa técnica reduz significativamente o número de permutações dos n vértices e, com isso, pode propiciar um melhor desempenho para as aplicações. Assim, a resolução implementada considera a programação dinâmica e supõe que uma viagem começa e termina no vértice 1, após visitar cada um dos vértices $2, 3, \ldots, n$ uma e uma só vez. Nesse sentido, qualquer viagem é constituída por uma aresta $(1, k), 2 \leq +k \leq n$, e um caminho M de k até 1, sendo que o caminho M visita cada um dos vértices em $VG - \{1, k\}$ uma e uma só vez. Se a viagem considerada é de custo mínimo, o caminho M deve ser necessariamente de

```
package agenda;

public class Agenda_impl extends AgendaPOA
{
   final int MAX = 20;

   private class Agenda
   {
      String nome;
      String fone;
   };
   private Agenda [] ag;
   private int numItens;

   public Agenda_impl()
   {
      numItens = 0;
      ag = new Agenda[MAX];
      for (int i=0; i < MAX; i++)
         ag[i] = new Agenda();
   }

   public String consultaNome(String chave)
   {
      String res = "0";
      for (int i=0; i < numItens; i++) {
         if ((ag[i].nome).equals(chave)) {
            res = ag[i].fone;
            break;
         }
      }
      return res;
   }
   . . .
}
```

Figura 3.7: Implementação da interface Agenda

custo mínimo.

Para a implementação do problema do caixeiro viajante na ferramenta TAO, foi adotado um processo mestre (o cliente) e o número de escravos (os servidores) é proporcional ao número de cidades menos 1. O mestre envia a rota completa (uma matriz contendo todos os possíveis caminhos) e o ponto (cidade) onde o viajante deve ir a partir da origem. Para cada escravo é enviado um ponto diferente. Após enviar essas informações para todos os escravos, o mestre aguarda pela resposta (a rota percorrida) de cada um deles e determina qual foi a melhor rota. A definição em IDL do problema do Caixeiro Viajante é apresentada na Figura 3.8.

Foi adotado um Servidor de Nomes (/ACE_ wrappers/TAO/orbsvcs/Naming_Service) para que a aplicação servidora registra-se sua identificação e a cliente pudesse obter a referência desse servidor. Uma outra característica da implementação do problema do Caixeiro Viajante na ferramenta TAO refere-se a utilização de invocações assíncronas através do modelo *callback* para que a aplicação mestre não fique "bloqueada" após cada invocação. Em razão disso, na

```
module Cx_Viajante
{
   typedef long mat_cx[4][4];

   struct ret{
     long destino;
     long custo;
     long vet_caixeiro[4];
   };
   typedef ret ret_caix;

   interface Caixeiro{
      ret_caix pcv(in long orig, in long dest, in mat_cax mat_cam);
   };
};
```

Figura 3.8: Arquivo IDL para o problema do Caixeiro Viajante

compilação da interface deve ser utilizado o parâmetro -GC, conforme exemplo a seguir:

```
tao_idl -GC caixeiro.idl
```

Após essa compilação, serão gerados os arquivos *stubs* e *skeletons*:

```
caixeiroC.h caixeiroC.cpp caixeiroS.h caixeiroS.cpp
```

A letra "C" no final do nome do arquivo, indica que o mesmo deve ser compilado junto com a aplicação cliente. A letra "S" indica que o arquivo deve ser compilado junto com a aplicação servidora. Além desses, serão gerados arquivos com a extensão ".i". Esses arquivos são definições de classes e parâmetros utilizados pelos *stubs* e *skeletons.*

A partir do exposto, inicia-se a implementação dos programas cliente e servidor. De forma geral, o roteiro para implementação do servidor não muda em relação aos exemplos apresentados anteriormente. No caso do programa cliente, o seguinte roteiro deve ser seguido:

1. Implementar a funcionalidade desejada no método que será invocada via *callback*;
2. Inicializar o ORB e Adaptador de Objetos;
3. Declarar e inicializar "handlers" (manipuladores) que, posteriormente, realizarão o *callback* para esse mesmo cliente;
4. Declarar um objeto da classe implementada no servidor e definida no arquivo IDL;
5. Obter a Referência do Objeto Servidor;
6. A partir da Referência obtida, inicializar o objeto de acordo com a classe implementada no servidor;
7. Realizar requisições (assíncronas) com o prefixo sendc_ aos servidores; Todas as requisições devem ter como parâmetro o *handler* que realizará o *callback.*

O item 1 do roteiro é exemplificado na Figura 3.9. Os itens 2 à 6 aparecem na Figura 3.10. A implementação do item 7 aparece na Figura 3.11. Deve-se atentar para o fato que em termos de implementação, todos os itens aparecem em um único arquivo.

```
#include <iomanip.h>
#include <iostream.h>
#include "caixeiroC.h"
#include "caixeiroS.h"
#include "ACE_wrappers/TAO/orbsvcs/orbsvcs/CosNamingC.h"

#define MAXNODES 4
#define INFINITY 100
#define FILENAME 100
#define TASKS 3
#define TRUE 1
#define FALSE 0

static int reply_count;
static Cx_Viajante::mat_caixeiro C;
static int viagem;

static int caminho[MAXNODES];
static CORBA::Long origem;

class Cx_Viajante_AMI_CaixeiroHandler_i : public virtual POA_Cx_Viajante::AMI_CaixeiroHandler
{
public:
    virtual void Cx_Viajante_AMI_CaixeiroHandler_i::pcv (const Cx_Viajante::ret_caixeiro &ami_ret,
                                                          CORBA::Environment &ACE_TRY_ENV)
    {
       int i;

       if (ami_ret.custo < viagem)
       {
          for (i=0;i<MAXNODES;i++)
               caminho[i]=ami_ret.vet_caixeiro[i];

          caminho[origem] = ami_ret.destino;
          viagem = ami_ret.custo;
       }
       reply_count--;
    }
};
```

Figura 3.9: Inicialização de variáveis e implementação do *callback* no cliente

Como já comentado, a adoção do mecanismo de *callback* para invocação de métodos remotos, não acarreta alterações quanto ao roteiro de desenvolvimento do servidor. Assim, de maneira geral a implementação do servidor continua seguindo os mesmos passos:

1. Implementar o método remoto;
2. Inicializar ORB e Adaptador de Objetos;
3. Obter a referência do Serviço de Nomes;
4. Registrar-se no Serviço de Nomes;
5. Aguardar por requisições dos clientes.

```
ACE_DECLARE_NEW_CORBA_ENV;

Cx_Viajante_AMI_CaixeiroHandler_i *handlers;
Cx_Viajante::AMI_CaixeiroHandler_var handler_refs;
CORBA::Long no;
int x,i,k,j, pos;
//vetor contendo a referência dos escravos (mestres) nas máquinas da rede
char *servername[]={"caixeiro1-lasdpc16.icmc.sc.usp.br",
                        "caixeiro2-lasdpc16.icmc.sc.usp.br",
                        "caixeiro3-lasdpc16.icmc.sc.usp.br"};

viagem=INFINITY;
//obtendo a referência do ORB e Adaptador de Objetos
CORBA::ORB_var orb=CORBA::ORB_init(argc,argv);
CORBA::Object_var poa_obj = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var rootp = PortableServer::POA::_narrow (poa_obj.in(), ACE_TRY_ENV);
PortableServer::POAManager_var poa_manager = rootp->the_POAManager (ACE_TRY_ENV);

poa_manager->activate (ACE_TRY_ENV);

//obtendo a referência para o Serviço de Nomes
CORBA::Object_var context_obj=orb->resolve_initial_references ("NameService", ACE_TRY_ENV);
CosNaming::NamingContext_var nam_cont = CosNaming::NamingContext::_narrow (context_obj.in());

CosNaming::Name name (1);
name.length(1);
CORBA::Object_var obj;

//declaração de objeto da classe implementada no servidor
Cx_Viajante::Caixeiro_var cv[TASKS];

//declaração dos handlers para fazer callback
handlers = new Cx_Viajante_AMI_CaixeiroHandler_i();
handler_refs = handlers->_this ();

for (i=0;i<(MAXNODES-1);i++)
{ //obtendo a referência dos servidores
  name[0].id = CORBA::string_dup (servername[i]);
  obj=nam_cont->resolve (name);
  if (CORBA::is_nil(obj.in()))
     cerr<<"Nil obj Referece"<<endl;

  cv[i]=Cx_Viajante::Caixeiro::_narrow(obj.in());
    if (CORBA::is_nil(cv[i].in()))
     cerr<< "Argument is not a valid reference"<<endl;
}
```

Figura 3.10: Inicialização do ORB, POA e Serviço de Nomes

O item 1 do roteiro foi dividido em duas Figuras: Figura 3.12 e 3.13. Os demais itens aparecem na Figura 3.14. Após a definição desses arquivos, pode-se compilá-los (conforme exemplo apresentado na Seção 3.1.2) visando a geração dos arquivos executáveis. É importante lembrar que um Servidor de Nomes deve estar "escutando" requisições na rede antes que o servidor e cliente sejam executados.

```
  //inicialização da matriz de caminhos que será enviada para os mestres
  for (i = 0; i < MAXNODES; ++i)
  {
     for ( j = 0; j < MAXNODES; ++j)
     {
        C[i][j] = 10;
        if (i == j)
        {
           C[i][j] = 0;
           if (i == (MAXNODES-1))
              C[i][0] = 1;
        }
        if (i+1 == j)
           C[i][j] = 1;
     }
  }
  pos=0;
  reply_count=TASKS;

  i=0; origem=0;

 for (no=0;no<MAXNODES;++no)
 {
       if (no!=origem)
       {  //cliente invoca método pcv no mestre
          cv[i]->sendc_pcv (handler_refs, origem, no, C);
          i++;
       }
 }
 //cliente fica aguardando por resposta do mestre
 while (reply_count>0)
 {
    if (orb->work_pending())
         orb->perform_work();
 }
 j = 0;
 pos = caminho[origem];
 while (j < MAXNODES-2)
 {
      pos = caminho[pos];
       j++;
 }
 caminho[pos] = origem;

 return 0;
}
```

Figura 3.11: Invocação do método *callback* nos servidores

```
#include "ACE_wrappers/TAO/orbsvcs/orbsvcs/CosNamingC.h"
#include "caixeiroI.h"

#define INFINITY 100
#define MAXNODES 4
#define TRUE 1
#define FALSE 0

static int pertence[MAXNODES];
static int caminho[MAXNODES];
static int vlr_atual[MAXNODES];
static Cx_Viajante::mat_caixeiro C;
static char servername[50];

int todos_sao_falsos(int pertence[])
{
        int i;
        for(i = 0; (i < MAXNODES && !pertence[i]); i++);

        if (i == MAXNODES)
           return(TRUE);
        else
           return(FALSE);
} // fim de todos_sao_true

int pcv_original(int origem, int i, int pertence[])
 {
        int custo, minimo, j, k;
        custo = 0;
        minimo = INFINITY;
        pertence[i] = FALSE;

        if (todos_sao_falsos(pertence)){
           pertence[i] = TRUE;
           return(C[i][origem]);
        }

        for (j = 0; j < MAXNODES ; j++) //Mudei de ++j para j++
        {
           if (j != i && pertence[j])
           {
              custo = C[i][j] + pcv_original(origem, j, pertence);
              if (custo<minimo)
              {
                  minimo = custo;
                  k = j;
              }
           }
        }
        pertence[i] = TRUE;
        if (minimo < vlr_atual[i])
       {
           caminho[i] = k;
           vlr_atual[i] = minimo;
        }
        return(minimo);
}
```

Figura 3.12: Implementação de métodos específicos para resolução do Problema do Caixeiro Viajante

```
Cx_Viajante::ret_caixeiro Cx_Viajante_Caixeiro_i::pcv (CORBA::Long origem, CORBA::Long destino,
                                                      const Cx_Viajante::mat_caixeiro mat_cam,
                                                      CORBA::Environment &ACE_TRY_ENV)
  {
    Cx_Viajante::ret_caixeiro retorno_caixeiro;
    int i,j, pos, viagem;

    for (i = 0; i < MAXNODES; ++i)
    {
       pertence[i] = TRUE;
       caminho[i] = -1;
       vlr_atual[i] = INFINITY;
    }

    for (i=0;i<MAXNODES;i++)
       for (j = 0; j < MAXNODES; j++)
           C[i][j]=mat_cam[i][j];

    pertence[origem] = FALSE;
    viagem = C[origem][destino] + pcv_original(origem, destino, pertence);
    i = 0;
    pos = caminho[origem];

    while ( i < MAXNODES-2 )
    {
         pos = caminho[pos];
         i++;
    }
    caminho[pos] = origem;
    retorno_caixeiro.destino=destino;
    retorno_caixeiro.custo=viagem;

    for (i=0;i<MAXNODES;i++)
        retorno_caixeiro.vet_caixeiro[i]=caminho[i];

    return(retorno_caixeiro);
}
```

Figura 3.13: Implementação do método (PCV) que é invocado pelo cliente

```
int main (int argc, char **argv)
{

  ACE_TRY_NEW_ENV
   {
    int j, i;
    PortableServer::POAManager_var nil_mgr = PortableServer::POAManager::_nil();
    char servername[50];
    int id;
    char *aux[argc-1];

    if (argc<2)
    {
       printf("\nErro...especifique um parametro <id>");
       return;
    }
    id=atoi(argv[1]);
    j=0;
    //Retirando o parametro ID de argv
    for (i=0;i<argc;i++)
    {
      if (i!=1)
      {
        aux[j]=argv[i];
        j++;
      }
    }
     argc--;

     for (i=0;i<argc;i++)
         argv[i]=aux[i];

     //Inicializa o ORB e POA
     CORBA::ORB_var orb=CORBA::ORB_init(argc,argv);
     CORBA::Object_var obj=orb->resolve_initial_references("RootPOA");
     PortableServer::POA_var poa=PortableServer::POA::_narrow(obj.in());
     PortableServer::POAManager_var mgr=poa->the_POAManager();
     mgr->activate();

     //Cria uma Ref. Única com base no nome da máquina para registrar no Serv. De Nomes
     struct utsname host_local;
     struct hostent *hp;
     uname(&host_local);

     if ((hp=gethostbyname(host_local.nodename))==NULL)
     {
       printf("Erro na função gethostbyname...");
       exit(0);
     }

     sprintf(servername,"caixeiro%d-%s",id, hp->h_name);
     Cx_Viajante_Caixeiro_i caix_servant;

     Cx_Viajante::Caixeiro_var caixeiro = caix_servant._this(ACE_TRY_ENV);

     //Obtém uma referência do Serviço de Nomes
     CORBA::Object_var naming_context_object = orb->resolve_initial_references ("NameService");
     CosNaming::NamingContext_var naming_context = CosNaming::NamingContext::_narrow (naming_context_object.in());

     CosNaming::Name name (1);
     name.length (1);

     //Registra-se junto ao Serviço de Nomes
     name[0].id = CORBA::string_dup (servername);
     naming_context->rebind (name, caixeiro.in());

    //servidor fica aguardando por requisições do cliente
    orb->run();
  }
  ACE_CATCHANY
    {
      ACE_PRINT_EXCEPTION (ACE_ANY_EXCEPTION,"Catched exception:");
      return 1;
    }
  ACE_ENDTRY;
   return 0;
}
```

Figura 3.14: Inicialização do ORB, POA e espera de requisições de clientes

Capítulo 4

Considerações Finais

Um sistema distribuído é constituído por um conjunto de elementos de hardware e software independentes, mas interligados de forma a produzir nos seus usuários a sensação de estar utilizando um todo coeso, como se o sistema fosse centralizado.

Desses dois tipos de elementos, hardware e software, este último assume fundamental importância, pois do seu bom projeto e confecção depende o sucesso de um sistema distribuído, tanto em termos da eficiência da comunicação entre as partes quanto do maior ou menor grau de transparência que o sistema venha a apresentar, além de aspectos como confiabilidade, segurança, escalabilidade e outros.

Sendo assim, novos paradigmas ou ferramentas de desenvolvimento de aplicações distribuídas são sempre muito bem-vindos. Na última década, a introdução da abordagem de objetos distribuídos, como "sucessora" e também complementar às chamadas a procedimentos remotos tem ganho muito adeptos e mostrado resultados concretos e viáveis, tornando-se hoje uma realidade.

O presente trabalho procurou apresentar alguns conceitos de sistemas distribuídos, com ênfase no desenvolvimento de aplicações cliente-servidor, abordando as chamadas a procedimentos remotos e os objetos distribuídos. Em particular, foi destacada a arquitetura CORBA, tendo sido detalhados seus principais componentes e apresentados exemplos de desenvolvimento de aplicações nessa plataforma, com uso de duas ferramentas distintas, ORBacus e TAO.

A abordagem empregada não foi exaustiva, mas espera-se que esta monografia sirva como ponto de partida para aqueles que pretendam estudar o desenvolvimento de aplicações distribuídas em CORBA, compreendendo a sua posição no contexto atual de sistemas distribuídos.

Referências Bibliográficas

[Arulanthu et al., 2000] Arulanthu, A. B., Schmidt, D. C., O'Ryan, C., Kircher, M., and Parsons, J. (2000). The design and performance of a scalable orb architecture for corba asynchronous messaging. *Proceedings of the IFIP/ACM Middleware 2000 Conference.*

[Comer, 1995] Comer, D. E. (1995). *Internetworking with TCP/IP - Volume I.* Prentice Hall, New Jersey, 3 edition.

[Coulouris et al., 2001] Coulouris, G., Dollimore, J., and Kindberg, T. (2001). *Distributed Systems - Concepts and Design.* Addison-Wesley, Wokingham-England, 3 edition.

[INPRISE, 1999] INPRISE (1999). *Visibroker for Java 4.0: Programmer's Guide: Using the POA.* http://www.inprise.com/techpubs/books/vbj/vbj40/programmers-guide/poa.html. Visitado em Dezembro de 1999.

[IONA, 2000] IONA (2000). *Orbix 2000.* http://www.iona-iportal.com/suite/orbix2000.htm. Visitado em Setembro de 2000.

[Lamport, 1994] Lamport, L. (1994). *LaTEX: A Document Preparation System.* Addison-Wesley Publising Company.

[Lo et al., 2000] Lo, S. L., Riddoch, D., and Grisby, D. (2000). *The omniORB verison 3.0 User's Guide.* ATT Laboratories, Cambridge.

[Mowbray and Malveau, 1997] Mowbray, T. J. and Malveau, R. C. (1997). *CORBA Design Patterns.* John Wiley Sons, New York.

[Mowbray and Ruh, 1997] Mowbray, T. J. and Ruh, W. A. (1997). *Inside CORBA: Distributed Object Standards and Applications.* Addison-Wesley.

[Mullender, 1993] Mullender, S. (1993). *Distributed Systems.* Addison-Wesley, New York, 2 edition.

[OMG, 1999] OMG (1999). *The Common Object Request Broker: Architecture and Specification.* Document formal/01-02-32, 2.3 edition.

[OMG, 2000] OMG (2000). *The Common Object Request Broker: Architecture and Specification.* Document formal/01-02-32, 2.4 edition.

[OMG, 2001] OMG (2001). *The Common Object Request Broker: Architecture and Specification.* Document formal/01-02-32.

[OOC, 2000] OOC (2000). *ORBacus for C++ and Java User's Manual.* http://www.ooc.com/ob/download4.html. Visitado em Agosto de 2000.

[Orfali, 1996] Orfali, R. (1996). *The Essential Distributed Objects Surival Guide.* John Wiley Sons, New York.

[Orfali et al., 1999] Orfali, R., Harkey, D., and Edwards, J. (1999). *Client/Server Survival Guide.* John Wiley, 3. edition.

[Puder et al., 2000] Puder, A., Römer, K., and Pilhofer, F. (2000). *MiCO: An Open Source CORBA 2.3 Implementation.* http://www.mico.org/doc-book.ps. Visitado em Junho de 2000.

[Santos et al., 2001] Santos, R. R., Souza, P. S. L., Santana, M. J., and Santana, R. H. C. (2001). Performance evaluation of distributed applications development tools from the interprocess communication point of view. *Proceedings of the Parallel Distributed Technics and Applications (PDPTA) Conference.*

[Schmidt and Vinoski, 1995] Schmidt, D. C. and Vinoski, S. (1995). Modeling distributed object applications. *C++ Report Magazine*, 2.

[Schmidt and Vinoski, 1996] Schmidt, D. C. and Vinoski, S. (1996). Distributed callbacks and decoupled communication in corba. *C++ Report Magazine*, 8.

[Schmidt and Vinoski, 1998] Schmidt, D. C. and Vinoski, S. (1998). An introduction to corba messaging. *C++ Report Magazine*, 15.

[Schmidt and Vinoski, 1999] Schmidt, D. C. and Vinoski, S. (1999). Programming asynchronous method invocations with corba messaging. *C++ Report Magazine*, 16.

[Stevens, 1996] Stevens, W. R. (1996). *Advanced Programming in the UNIX Environment.* Addison-Wesley, Reading.

[Tanenbaum, 1995] Tanenbaum, A. S. (1995). *Distributed Operating Systems.* Prentice-Hall, New Jersey.

[Tanenbaum and Renesse, 1985] Tanenbaum, A. S. and Renesse, R. V. (1985). Distributed operating systems. *Computing Surveys*, 17(4).

[Tanenbaum and Steen, 2002] Tanenbaum, A. S. and Steen, M. V. (2002). *Distributed Systems - Principles and Paradigms.* Prentice-Hall, New Jersey.

[Terada, 1991] Terada, R. (1991). *Desenvolvimento de Algoritmos e Estruturas de Dados.* Makron Books, Rio de Janeiro.

[Vinoski, 1997] Vinoski, S. (1997). Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2).

[Vinoski and Henning, 1999] Vinoski, S. and Henning, M. (1999). *Advanced CORBA Programming with C++.* Addison-Wesley, Reading.