2

Programação em Sistemas Distribuídos

Rômulo de Oliveira[1] **Joni Fraga**[1] **Carlos Montez**[1]

Resumo:

A redução do preço de computadores e sua grande disseminação vêm impulsionando a criação de redes de computadores. Hoje, essas redes estão em todos os lugares, e a Internet pode ser entendida como um grande e disperso sistema distribuído, que permite aos seus usuários o acesso a uma variedade de serviços. São múltiplos os desafios enfrentados pelos projetistas desses sistemas – frente aos problemas de heterogeneidade, segurança e confiabilidade.

Este curso descreve os principais conceitos relacionados ao desenvolvimento de sistemas distribuídos, tais como: heterogeneidade de representação de dados e *marshaling*; transparências de distribuição e *middleware*; modelo cliente-servidor; Chamada de Procedimento Remoto (RPC); serviços de nome e *binding*. Também serão abordadas duas das principais plataformas utilizadas para esse propósito atualmente: JAVA (com suas tecnologias relacionadas) e CORBA.

[1] Laboratório de Controle e Microinformática – LCMI
Depto de Automação e Sistemas – DAS
Universidade Federal de Santa Catarina – UFSC
e-mail: {romulo, fraga, montez}@das.ufsc.br

2.1. Introdução

A redução do preço dos computadores e sua grande disseminação, a partir do surgimento de microcomputadores na década de 80, impulsionaram a criação de redes de computadores. Essas redes foram criadas, principalmente, devido à necessidade de compartilhamento de recursos entre os computadores. Inicialmente, pequenas redes locais foram sendo instaladas nas instituições, e, posteriormente, a partir de meados da década de 90 no Brasil, elas foram se interligando à Internet. Hoje, as redes de computadores estão em todos os lugares e a Internet pode ser entendida como um grande e disperso sistema distribuído que permite aos seus usuários acessarem serviços, tais como www, email e transferência de arquivos, entre muitos outros.

Um sistema distribuído pode ser definido como aquele "formado por componentes de hardware e software, situados em redes de computadores, e que se comunicam e coordenam suas ações apenas através de trocas de mensagens" [Coulouris 2001]. A separação espacial dos computadores que formam esses sistemas distribuídos pode variar desde poucos metros, em uma rede local, até quilômetros de distância, em uma rede largamente dispersa. Da mesma maneira, podem variar bastante os desafios enfrentados pelos projetistas desses sistemas - frente aos problemas de heterogeneidade, segurança e confiabilidade (tolerância a faltas).

Envios e recepções de mensagens não ocorrem instantaneamente, e a conseqüência imediata do fato da comunicação entre os componentes dos sistemas distribuídos se dar apenas por trocas de mensagens, é a inexistência de um relógio global, no qual todos os computadores possam se basear. Essa característica dificulta a obtenção de um estado global dos componentes do sistema, importante para a resolução de uma série de problemas de sincronização que surgem nas aplicações. Questões de coordenação e sincronização em sistemas distribuídos decorrem exatamente da sua característica de concorrência entre seus componentes e a necessidade de compartilhamento de recursos entre eles. Esses problemas formam uma classe que inclui a necessidade, entre outros, de algoritmos de eleição, detecção de *deadlocks* (impasses) distribuídos, algoritmos de concordância, etc [Lynch 1996].

O objetivo deste texto não é tratar de algoritmos distribuídos, mas sim tratar de plataformas de software que suportam a execução (facilitam a construção) de aplicações distribuídas. As primeiras propostas de plataformas para suportar a construção de aplicações distribuídas, no início da década de oitenta, eram limitadas. Elas tratavam com sistemas homogêneos e estavam baseadas em sistemas operacionais ou linguagens especiais para programação distribuída (ARGUS [Liskov 1988], Emerald [Jul 1991], CONIC [Kramer 1989]). Com a expansão dos sistemas tomando proporções geográficas consideráveis e envolvendo diferentes tecnologias, a necessidade de padrões e plataformas mais flexíveis tornou-se evidente. Nesse sentido houve vários esforços, tais como ODP [ISO 1995], ANSAware [ANSA], DCE [OSF]. Essas propostas serviram para demarcar o

caminho para o conjunto de padrões para ambientes operacionais distribuídos existentes atualmente.

O propósito deste texto é apresentar os conceitos fundamentais presentes no desenvolvimento de sistemas distribuídos, assim como as principais plataformas utilizadas para esse propósito atualmente. O tema em si é vasto, e existem livros inteiros para tratar do assunto, tais como [Tanenbaum 2002], [Coulouris 2001], [Birman 1996], [Lynch 1996]. Em função da limitação de espaço, este texto concentra-se na descrição de duas das tecnologias mais utilizadas atualmente para a construção de sistemas distribuídos. A seção 2.2 apresenta os conceitos elementares da área, presentes em qualquer plataforma que suporte a execução de sistemas distribuídos. Em seguida, a seção 2.3 descreve como aplicações distribuídas podem ser escritas em Java. A seção 2.4 faz o mesmo para o padrão CORBA. Finalmente, a seção 2.5 contém as considerações finais.

2.2. Conceitos básicos

Toda a comunicação entre processos em um sistema distribuído é baseada em trocas de mensagens, pois, conceitualmente, não há memória compartilhada. Quando vários processos precisam se comunicar, é necessária a adoção de protocolos, ou seja, acordos entre as partes que se comunicam, sobre como a comunicação se dará (formatos, conteúdos e significados de mensagens). Cada sistema final na rede executa protocolos que controlam o envio e recepção de informações na rede.

Atualmente, o conjunto de protocolos da Internet – denominado TCP/IP [RFC 1180] – se tornou um padrão *de facto* em ambientes de sistemas distribuídos. Seus dois protocolos mais importantes são o TCP (*Transmition Control Protocol*) – orientado a conexões –, e o IP (*Internet Protocol*), motivo pelo qual esse conjunto de protocolo é conhecido como TCP/IP. A Figura 2.1 mostra a pilha de protocolos TCP/IP, comparando-a com as sete camadas do modelo ISO OSI [Stevens1998, Kurose 2001].

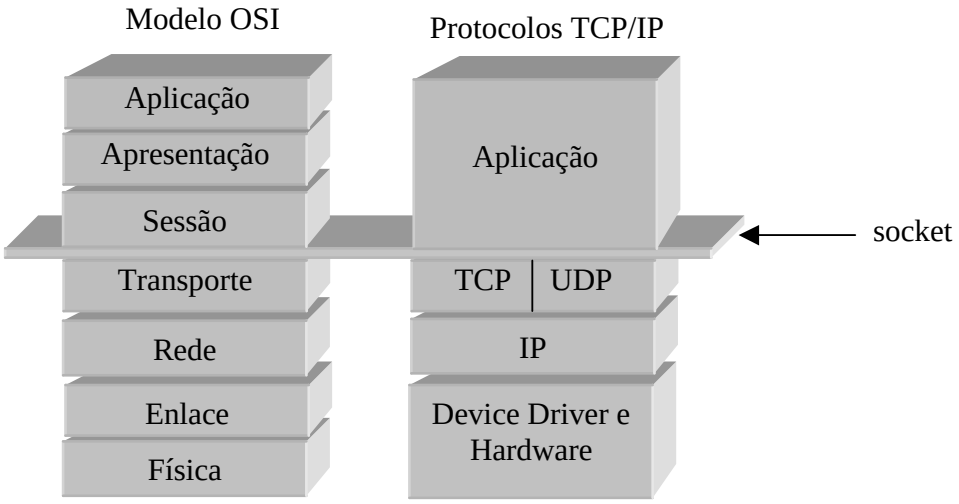

Figura 2.1. Pilha de protocolos do Modelo OSI e TCP/IP.

As aplicações usualmente têm acesso aos protocolos de transporte da rede usando *sockets*. Um *socket* pode ser concebido como um portão de entrada e de saída de cada processo. Cada processo envia e/ou recebe mensagens através de seus *sockets*. Devido ao fato que, usualmente, os protocolos de transporte da rede são implementados e controlados pelo sistema operacional, um *socket* também pode ser concebido como uma interface entre a aplicação e o substrato de comunicação do sistema operacional (Figura 2.2).

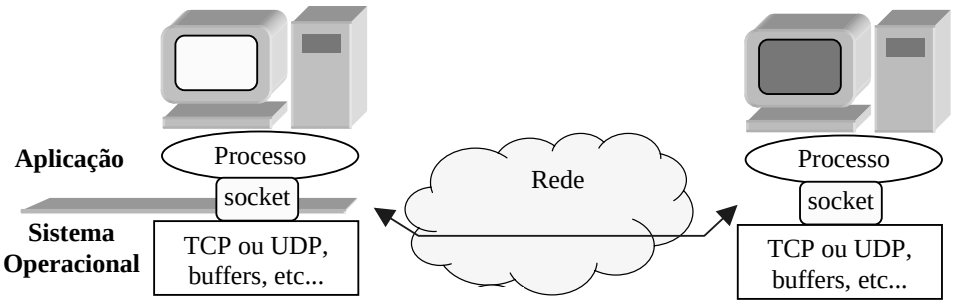

Figura 2.2. Socket como interface de acesso à rede.

A programação de sistemas distribuídos pode ser feita diretamente em uma linguagem de programação (ex. C, C++ ou Java), usando uma API (*Application Program Interface* – Interface de Programação de Aplicações) *socket*. Na seção 2.3.1, este texto apresenta os principais conceitos relacionados à programação de aplicações distribuídas usando a API *socket* do Java [Harold 1997].

2.2.1. Heterogeneidade de representação de dados e marshaling

É viável a construção de grandes sistemas distribuídos através da API *socket*. Entretanto, o programador de tais sistemas precisa lidar diretamente com uma série de questões, tais como heterogeneidade de representação de dados e falta de transparências de distribuição. Essas questões exacerbam a complexidade do sistema, desviando a atenção dos programadores do problema que deveria ser originalmente resolvido.

Suponha, por exemplo, um sistema heterogêneo, onde processos clientes e servidores residam em computadores com diferentes tipos de processadores. Ao trocar mensagens entre si, é possível que alguns dos processos representem um valor inteiro, com 2 bytes, usando a representação *little-endian*, enquanto outros processos usem a representação *big-endian*. A representação *little-endian*, usada pela Intel, representa o valor inteiro considerando o byte mais significativo antes do menos significativo. Já a representação *big-endian*, usada, por exemplo, pelos processadores SPARC, é o inverso. A Figura 2.3 mostra as duas formas de representação do valor inteiro 256 (100 na base 16).

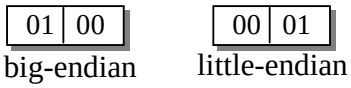

Figura 2.3. Representação big-endian e little-endian.

Problemas semelhantes surgem com representações diferentes para números de ponto flutuantes, e até mesmo com representações de códigos de caracteres (ASCII ou EBCDIC). Envio de dados pelo suporte de comunicação, implica em empacotá-los na forma de mensagens no momento do envio, e desempacotá-los no momento de suas recepções. Representações diferentes de dados podem gerar interpretações diferentes dos valores dos dados. Para superar essas questões, um sistema distribuído pode adotar duas estratégias. A primeira é a de converter todos os dados transmitidos para um formato comum, previamente conhecido por todos os processos. A segunda estratégia é a de enviar junto com os dados, informações de qual tipo de representação está sendo adotada naquela mensagem.

Ambas as soluções demandam trabalho adicional do programador e aumentam a complexidade do sistema. Com objetivo de facilitar a programação de sistemas distribuídos, seria interessante que as questões da heterogeneidade dos dados, e o trabalho adicional gerado por esses problemas, no envio e na recepção das mensagens, não precisassem ser tratados diretamente pelos programadores. Veremos adiante nesse texto que os *stubs*, implementados em suportes de *middleware*, são os componentes de sistemas distribuídos que buscam ocultar dos programadores toda complexidade envolvida com o empacotamento (*marshaling*) e desempacotamento de mensagens (*unmarshaling*), respectivamente, nos envios e recepções de mensagens. (Empacotamento e desempacotamento de mensagens também costumam ser chamados de serialização e deserialização de parâmetros)

2.2.2. Transparências de distribuição e middleware

Para complicar ainda mais esse cenário, suponha que um programador resolva implementar um serviço em dois servidores diferentes (replicação) por motivo de tolerância a faltas. Nesse caso, o programador teria de se preocupar em implementar cada cliente fazendo requisições para os dois servidores. Ou ainda, é possível que vários processos do sistema distribuído precisem fazer acesso simultâneo a um recurso compartilhado (por exemplo, um arquivo). Nesse caso, o programador precisa implementar um código que controle a competição no acesso a esse recurso (por exemplo, garantindo a exclusão mútua no acesso ao recurso).

Seria interessante que essas questões de replicação e concorrência fossem transparentes (invisíveis) para o programador, ou seja, que este não precisasse se preocupar com esses temas, e se concentrasse no problema principal a ser resolvido. Assim, o ideal é que existisse, do ponto de vista do programador do sistema, transparências de replicação e de concorrência. Basicamente, costuma-se reconhecer oito tipos de transparências de distribuição [ISO 1995]:

- *de acesso*: quando o acesso aos recursos é feito de forma idêntica, independentemente se estes se encontram localmente ou remotamente localizados;
- *de localização*: quando os recursos são acessados sem o conhecimento de suas localizações;

- *de concorrência*: permite que vários processos concorrentes compartilhem recursos sem interferência entre eles;
- *de replicação*: quando o acesso às múltiplas instâncias de um recurso é feito sem o conhecimento explícito do programador;
- *de falha*: permite que usuários e as aplicações completem suas tarefas apesar de falhas em componentes de hardware ou software;
- *de migração*: permite a movimentação de recursos e de clientes sem afetar a operação ([Coulouris 2001] denomina essa transparência como "de mobilidade", devido à importância crescente da computação móvel nos dias atuais);
- *desempenho*: permite que o sistema seja reconfigurado automaticamente quando a carga varia no tempo;
- *de escalabilidade*: permite que o sistema se expanda sem a mudança na estrutura do sistema ou nos algoritmos de suas aplicações.

A forma usual de se obter essas transparências de distribuição é através do uso de *middlewares*. Em um sistema distribuído, o software que suporta um modelo de programação atuando sobre serviços de sistema operacional e de rede é chamado de *middleware*.

Basicamente, *middleware* é uma camada de *software*, residente acima do sistema operacional e do substrato de comunicação, que oferece abstrações de alto nível, fornecendo uma visão uniforme na utilização de recursos heterogêneos existentes, com objetivo de facilitar a programação distribuída (Figura 2.4).

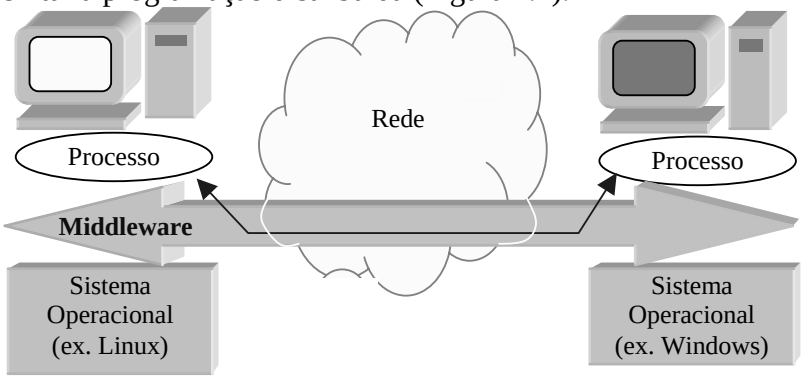

Figura 2.4. *Middlewares* ocultam a heterogeneidade em sistemas distribuídos.

A seção 2.3 desse texto apresenta o *middleware* Java RMI, e a seção 2.4 os padrões CORBA, um *middleware* para programação de sistemas distribuídos na forma de objetos distribuídos. Ambos buscam ocultar a heterogeneidade de hardware e sistemas operacionais hospedeiros. CORBA vai além, no sentido que também permite que o sistema seja desenvolvido usando linguagens de programação diferentes.

2.2.3. Modelo cliente-servidor

Interações em um sistema distribuído ocorrem, usualmente, através do modelo cliente-servidor. No modelo cliente-servidor, um sistema é estruturado em processos cooperantes que assumem papeis de clientes ou de servidores. Um processo assume o papel de servidor quando ele oferece serviços aos outros processos clientes. Um mesmo processo pode ora fazer papel de cliente, ora papel de servidor. Por exemplo, um processo servidor de nomes (*ex.* DNS [RFC 1035]) pode receber uma requisição para resolução de um nome o qual ele não possa resolver. Nesse caso, o servidor, temporariamente, assume papel de cliente fazendo uma requisição para outro servidor de nomes. Servidores de tempo NTP [Mills 1991] também funcionam de forma hierárquica, onde alguns servidores sincronizam seus relógios, periodicamente, fazendo requisições a outros servidores.

A maneira mais simples de implementar um modelo cliente-servidor é através de um protocolo simples do tipo requisição/resposta (*request/reply*). A Figura 2.5 apresenta um exemplo de implementação de um protocolo requisição/resposta usando três primitivas *fazOperação(),* e *aguardaPedido() e* enviaResposta*()*. Essas primitivas, por sua vez, podem ser implementadas através de operações de envio e recepção de mensagens existentes, por exemplo, na API *socket*.

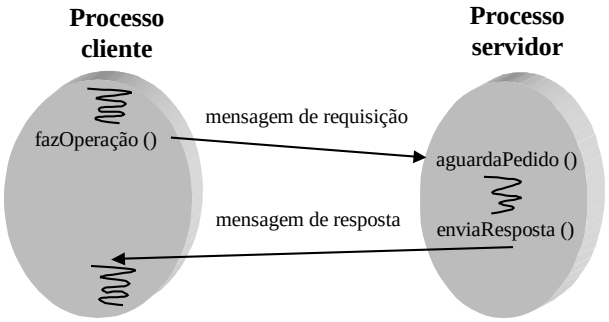

Figura 2.5. Modelo cliente-servidor usando um protocolo requisição-resposta.

Nessa implementação de protocolo, um processo cliente, envia uma mensagem de requisição para um servidor requisitando algum serviço (*fazOperação()*). Essa operação bloqueia o processo cliente enquanto não é retornada uma mensagem de resposta. No outro ponto final da comunicação, o servidor já estava bloqueado aguardando alguma requisição de serviço (*aguardaPedido()*). Ao receber o pedido de requisição, o servidor se desbloqueia, extrai a mensagem, executa a operação pedida pelo cliente, e retorna uma mensagem para o cliente (*enviaResposta()*). Um exemplo tradicional de implementação de um protocolo requisição/resposta é o HTTP, usado em interações entre navegadores e servidor web.

É interessante notar que toda a complexidade decorrente do tratamento de possíveis perdas de mensagens, da heterogeneidade de representação de dados, e questões de empacotamento (*marshaling*) e desempacotamento (*unmarshaling*) de dados em mensagens não é representada na Figura 2.5. O conceito de chamada de procedimento remoto (RPC)

apresentado a seguir busca, exatamente, ocultar essa complexidade em sistemas distribuídos, fornecendo ao programador um ambiente de programação mais confortável.

2.2.4. Chamada de Procedimento Remoto

Uma das principais atividades em sistemas distribuídos é a da busca de paradigmas para programação distribuída que facilitem ao máximo os programadores desses sistemas. Programadores estão acostumados a desenvolver sistemas fazendo chamadas de procedimentos (no paradigma de programação procedural) ou invocações em operações em objetos (no paradigma de programação orientada a objetos). O desenvolvimento de sistemas distribuídos unicamente através de trocas de mensagens, usando, por exemplo, API *sockets*, torna a programação tediosa e com bastante tendência a introdução de erros durante sua codificação.

A implementação de modelos cliente-servidor, usando protocolos requisição/resposta, representa "um passo a mais" para o programador desses sistemas. Contudo, conforme já foi dito, toda a complexidade com relação ao empacotamento e desempacotamento de mensagens precisa ser tratada pelo programador. Assim, em meados da década de 80 foi proposta a idéia do RPC (*Remote Procedure Call* – Chamada de Procedimento Remoto).

A Figura 2.6 esquematiza como são efetuadas chamadas usando o RPC. A idéia básica é, do ponto de vista dos programadores, tornar as interações entre clientes e servidores semelhantes a uma chamada de procedimento. Para o programador existe uma *transparência de acesso*, e toda a complexidade relacionada às tarefas de serialização/deserialização de parâmetros também são ocultadas do programador.

A complexidade dessas tarefas é ocultada do programador, e encapsulada dentro dos *stubs*. Os *stubs* são gerados durante a compilação dos programas que compõem o sistema, e uma descrição mais detalhada desse processo de geração desses *stubs*, é apresentada mais adiante, nas seções 2.3.3 (Java RMI) e 2.4.9 (CORBA).

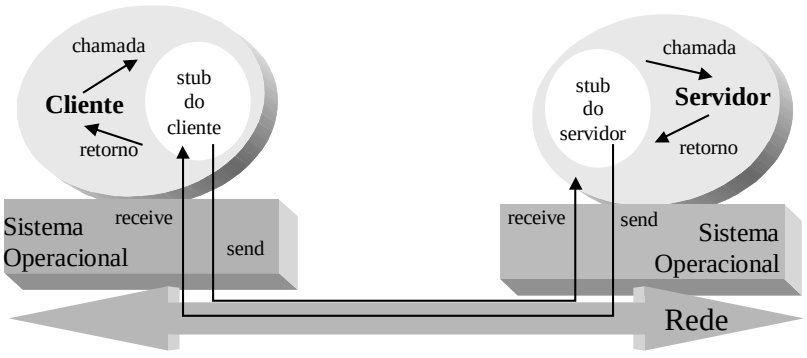

Figura 2.6. Chamada de Procedimento Remoto.

Basicamente, o cliente executa uma chamada de procedimento. O *stub* do cliente empacota a requisição e seus parâmetros (*marshaling*) e envia a mensagem (*send*). Do

outro lado, a mensagem é recebida (*receive*) pelo *stub* do servidor, que desempacota (*unmarshaling*) a mensagem, executando uma chamada de procedimento ao servidor. Após o servidor executar o procedimento, o valor de retorno é devolvido ao *stub* do servidor, o qual empacota a resposta em uma mensagem, que é enviada (*send*) de volta para o cliente. O *stub* do cliente recebe (*receive*) a mensagem de retorno, desempacota a resposta e a retorna para o processo cliente.

Para o processo cliente, todas essas etapas ocorrem de forma transparente. Ele simplesmente faz uma chamada de procedimento, aguarda que ela seja executada, e recebe a resposta.

2.2.5. Serviços de Nome e binding

Aplicações tradicionais (não distribuídas) não têm necessidade de esquemas especiais para identificar e localizar seus provedores de serviço. Nesse caso, os serviços utilizados usualmente são oferecidos pelos sistemas operacionais, que executam no mesmo computador das aplicações clientes. A forma de obter acesso a esses serviços já é conhecida previamente, em tempo de compilação, e normalmente se dá através de chamadas de procedimentos.

Por outro lado, aplicações distribuídas necessitam de esquemas elaborados para localizar os seus fornecedores de serviço. Uma forma convencional e imediata é o uso de endereços IP e de portas de comunicação. Por exemplo, a porta 80 do endereço IP 200.135.48.81 poderia identificar um determinado servidor web na Internet. Entretanto, essa forma de localização e identificação é inadequada em sistemas distribuídos por vários motivos. Uma delas é o fato do par "porta-endereço IP" não identificar de forma inequívoca um serviço. Ou seja, apesar da tentativa de padronização no uso de valores numéricos de portas para alguns serviços, nada garante que a porta 80, por exemplo, seja usada exclusivamente em servidores web. Outro serviço na Internet poderia usar esse valor. Além disso, é possível, por exemplo, que alguns servidores web possam fazer uso de outras portas tais como a 8080.

Contudo, a principal desvantagem do uso dessa abordagem, é o fato dela dificultar a obtenção das *transparências de distribuição*. Nesse sentido, visando resolver esses problemas, atualmente são empregados serviços de nomes. Serviços de nomes (ou *location brokers*) permitem que clientes tenham acesso a serviços remotos através de nomes – identificadores de "alto-nível" –, em contraste com a identificação "baixo-nível" oferecida pelo par "porta-endereço IP". Caso o serviço seja replicado, ou migre, o cliente continua a acessar o serviço de forma transparente. Na Internet o DNS [RFC 1035] é o padrão adotado para serviço de nomes. Nesse texto, iremos apresentar: na seção 2.3.3, a classe `java.rmi.Naming;` na seção 2.3.4, as funcionalidades do *Servidor de Lookup* do Jini; e, na seção 2.4.6, o Serviço de Nomes do CORBA.

A seguir são apresentadas algumas tecnologias Java usadas para programação de sistemas distribuídos: *socket*, *servlet*, *RMI* e *Jini*.

2.3. Tecnologias Java

Java é uma linguagem de programação criada pela *Sun Microsystems*, anunciada oficialmente em 1995, com o propósito de se tornar a linguagem de programação preferida para a Internet. Essa pretensão está baseada em dois aspectos de Java. Primeiramente, o compilador Java gera código para uma máquina hipotética, chamada *Máquina Virtual Java*. Este código apresenta a característica de ser interpretado. Dessa forma, o código-objeto gerado pela compilação de um programa Java é altamente portável através de diferentes arquiteturas e sistemas operacionais. Basta que exista uma implementação da máquina virtual Java em um computador para que qualquer código Java possa executar nele, sem a necessidade de uma recompilação. No contexto da Internet, onde a heterogeneidade das plataformas é a regra, a portabilidade de código-objeto, sem necessidade de recompilações, é uma vantagem importante.

Outro aspecto que torna Java atraente e adequada para Internet é o da própria sintaxe da linguagem. Ela é orientada a objetos, inclui suporte nativo para programação *multithread* e tratamento de exceções, possui uma imensa biblioteca de classes que ajuda no momento de criar os aplicativos e possui um tratamento de tipos mais rigoroso do que, por exemplo, C++. Este texto não tem a pretensão de ensinar Java, algo capaz de ocupar milhares de páginas. O leitor interessado pode consultar os vários livros existentes que tratam do assunto, como [Arnold 1997] e [Deitel 2001].

Java não é apenas uma linguagem de programação. Ela é cercada de diversas tecnologias que atendem a diferentes propósitos. Cada uma dessas tecnologias aparece na forma de bibliotecas e/ou programas auxiliares, criadas com um propósito específico. Uma visão geral das tecnologias associadas com Java pode ser obtida na página http://java.sun.com/products. Das várias tecnologias associadas com Java, três delas são especialmente importantes para a programação de sistemas distribuídos: Servlets, RMI e Jini. Cada uma delas será tratada aqui. Porém, inicialmente será mostrado como a linguagem suporta de maneira simples a construção de programas distribuídos, usando um recurso de mais baixo nível: a comunicação através de *sockets*.

2.3.1. Sockets em Java

O *socket* é uma abstração que representa um ponto de comunicação em uma rede de computadores [Harold 1997]. Para dois computadores trocarem informações, cada um utiliza um *socket*. Tipicamente, um computador atua como servidor, abrindo o *socket* e ficando na escuta, a espera de mensagens ou pedidos de conexões. O outro computador assume o papel de cliente, enviando mensagens para o *socket* do servidor.

Existem dois modos de operação: orientado a conexão, baseado no protocolo TCP (*Transport Control Protocol*) e sem conexão, empregando o protocolo UDP (*User Datagram Protocol*). Para usar *sockets* orientados a conexão, antes do envio dos dados é necessário o estabelecimento de uma conexão entre o cliente e o servidor. A conexão deve ser terminada ao final da comunicação e os dados chegam na mesma ordem que foram enviados. Quando *sockets* sem conexão são empregados, a entrega não é garantida. Dados

podem chegar em ordem diferente da que foram enviados. O modo usado depende das necessidades da aplicação. Via de regra, o uso de conexão implica em maior confiabilidade, porém com maior *overhead*.

Cada computador em uma rede TCP/IP possui endereço IP único. Portas representam conexões individuais com esse endereço. Quando o *socket* é criado, ele deve ser associado com uma porta específica, isto é, efetuar o seu *binding*. Para estabelecer uma conexão o cliente precisa conhecer o endereço IP da máquina onde o servidor executa e o número da porta que o servidor está escutando. Existe a tentativa de padronização de alguns números de portas com relação aos serviços oferecidos pelo servidor, tais como: 7-echo, 13-daytime, 21-ftp, 23-telnet, 25-smtp, 79-finger, 80-http, 110-pop3. Estes números de portas são definidos pela IANA (*Internet Assigned Numbers Authority*).

A Figura 2.7 mostra o algoritmo básico para clientes e servidores quando TCP é usado. Em Java, clientes e servidores adotam procedimentos diferentes. Um *socket* cliente é criado e conectado pelo construtor da classe `Socket`. Por exemplo:

```
clientSocket = new Socket("merlin.das.ufsc.br", 80);
```

A comunicação em si é feita com o auxílio de classes tipo *streams*, que são classes do pacote `java.io`. Os clientes seguem sempre a mesma receita básica: cria um *socket* com conexão cliente e utiliza classes que implementam *streams* para efetuar a comunicação. Existem diversas opções para *sockets* clientes, tais como definir o *timeout* usado no protocolo de desconexão, *timeout* utilizado em operações de leitura, etc.

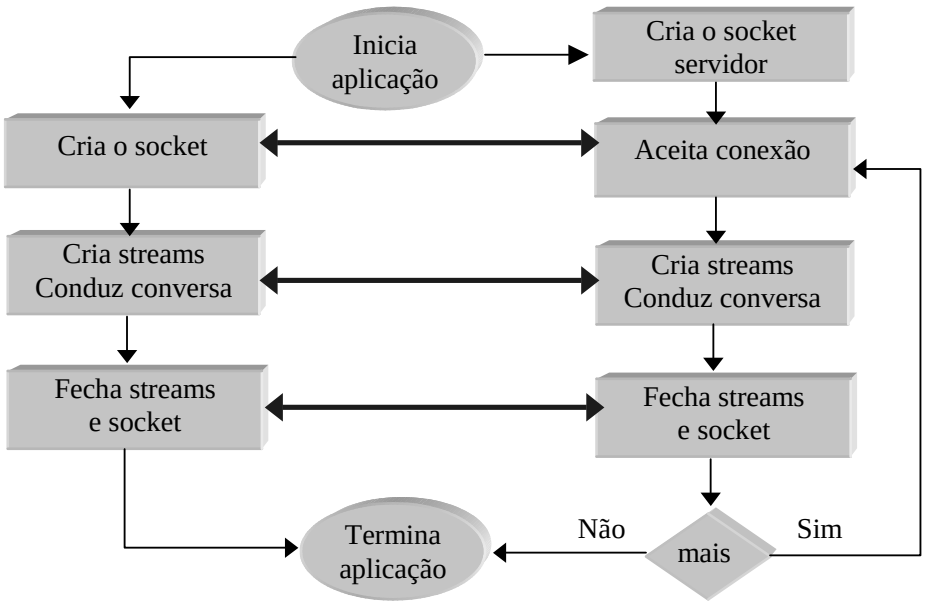

Figura 2.7. Comunicação utilizando *sockets* orientados à conexão.

Servidores não criam conexões, eles esperam pelo pedido de conexão de um cliente. O construtor de `ServerSocket` é usado, como em:

```
ServerSocket serverSocket = new ServerSocket(80, 5);
```

Um servidor pode receber várias requisições de conexão ao mesmo tempo e as requisições são processadas uma a uma. A fila de requisições ainda não atendidas é denominada *pilha de escuta*. O tamanho da fila de escuta indica quantas requisições de conexão simultâneas são mantidas. Assim, na construção acima, o primeiro parâmetro é o número da porta a ser escutada e o segundo parâmetro é o tamanho da pilha de escuta. O valor *default é 50.*

O método `accept()` é chamado para retirar requisições da fila. Ele bloqueia o processo até a chegada uma requisição, e após isso, retorna um *socket* conectado com o cliente. O objeto *socket* do servidor não é usado, um novo é criado para a transferência dos dados. O *socket* do servidor continua enfileirando pedidos de conexão. Finalmente, *streams* são usados para a troca de dados. Um servidor simples processa uma conexão de cada vez. Entretanto, utilizando múltiplas *threads*, um servidor concorrente vai criar uma nova *thread* para cada conexão aberta e assim consegue processar várias conexões simultaneamente. Por exemplo, servidores web comerciais são todos concorrentes.

No caso de comunicação baseada em datagramas, a classe `DatagramSocket` é usada tanto por clientes como por servidores. O servidor deve especificar sua porta, e a omissão do parâmetro significa "use a próxima porta livre". O cliente sempre utiliza a próxima. O algoritmo básico é mostrado na Figura 2.8.

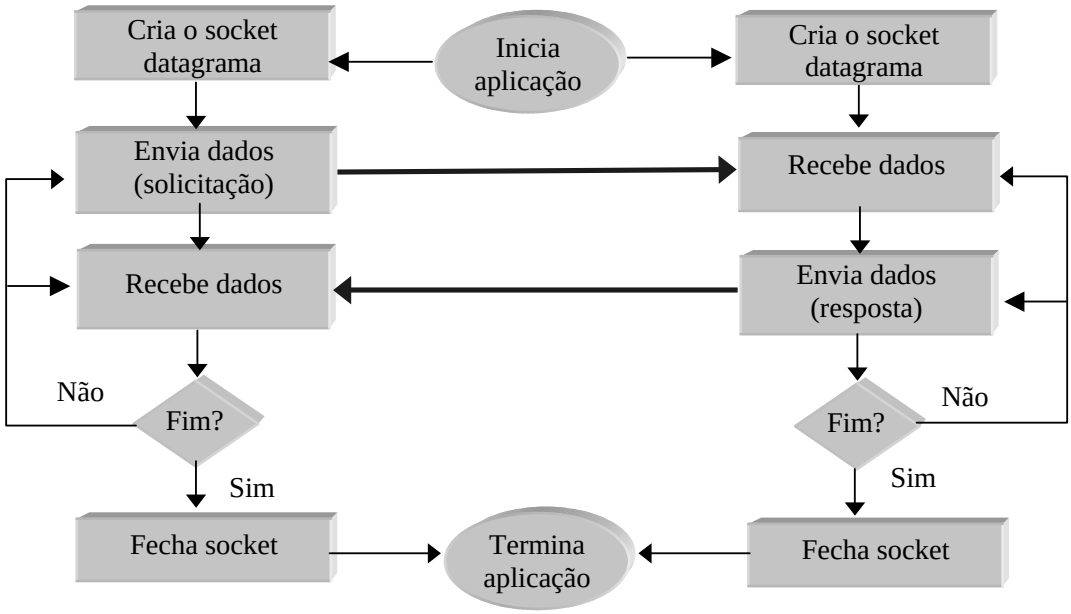

Figura 2.8. Comunicação utilizando *sockets* orientados a datagrama

A classe `DatagramPacket` é usada para enviar e receber dados. Ela contém informações para conexão e os próprios dados. Para receber dados de um *socket* datagrama, o método `receive()` bloqueia até a recepção dos dados.

Para enviar dados para um *socket* datagrama é necessário um endereço, a classe `InetAddress` representa endereços na Internet. Por exemplo:

```
InetAddress addr = InetAddress.getByName(“das.ufsc.br”).
```

O envio de um *string* para um *socket* destinatário é feito através da montagem de um objeto `DatagramPacket`, o qual conterá a mensagem e o endereço destino. Seu envio através do método `send()` de objeto da classe `DatagramSocket` criado anteriormente pelo cliente.

2.3.2. Servlets

Servlets é uma forma de programação distribuída em Java que utiliza abstrações de mais alto nível do que o emprego direto de *sockets*. O *servlet* segue o modelo cliente-servidor e estende a funcionalidade de um servidor. Em particular, estende a funcionalidade de servidores HTTP. Neste caso, são utilizados para interagir com bancos de dados, gerar páginas dinâmicas e manter informações sobre uma sessão web em andamento. A Figura 2.9 ilustra a sua utilização. Embora tenham sido concebidos com este contexto, é possível utilizá-los em outras situações. Os pacotes relacionados com esta tecnologia são `javax.servlet` e `javax.servlet.http`.

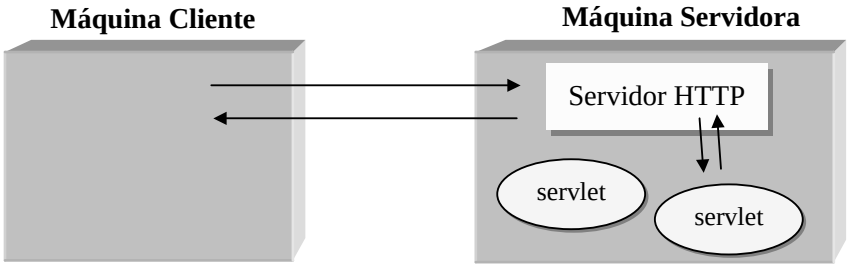

Figura 2.9. Esquema básico dos servlets.

Servlets são semelhantes aos *applets*, mas executam do lado do servidor. São suportados pelos servidores HTTP mais conhecidos, como o servidor web da Netscape, o IIS da Microsoft e o servidor Apache. Na situação típica, o cliente (provavelmente um navegador web) envia uma requisição HTTP para o servidor HTTP, o servidor recebe a solicitação e a repassa para o *servlet* apropriado. O *servlet* processa a requisição, incluindo aqui um possível acesso a bancos de dados, e retorna os dados para o cliente, os quais envolvem documentos HTML, imagens, ou outros tipos de documentos usados no ambiente web.

Todo *servlet* deve implementar a interface `javax.servlet.Servlet`. Muitos dos métodos especificados nessa interface serão chamados pelo servidor que hospeda o *servlet*.

A interface é composta pelos métodos `init()` (automaticamente chamado uma vez para permitir a inicialização do *servlet*), `service()` (método chamado com a solicitação de um cliente), `getServletConfig()` (deve retornar uma referência para o objeto que implementa a interface `ServletConfig`, o qual fornece informações de configuração), `getServletInfo()` (deve retornar um String que contém informações tais como autor e versão) e `destroy()` (método responsável por liberar quaisquer recursos que tenham sido alocados pelo *servlet* durante sua execução). O método mais importante é o `service()`, pois ele é responsável por implementar a funcionalidade do *servlet*. Seus parâmetros são dois objetos *stream* que permitem receber dados do cliente (a requisição) e enviar dados para o cliente (a resposta).

Existem duas classes abstratas incluídas nos pacotes relativos a *servlets* que fornecem uma implementação padrão para facilitar a programação dos *servlets*. O programador pode partir dessas classes, e criar uma subclasse na qual sobrepõe alguns dos métodos da interface. As classes abstratas são `GenericServlet` e `HttpServlet`.

A classe `HttpServlet` sobrescreve o método `service` para fazer distinção entre as operações mais freqüentes do protocolo HTTP. Dessa forma, o método `service` analisa a requisição recebida e, conforme a operação, chama o método apropriado. Por exemplo, a classe define métodos como `doGet()`, `doPost()`, `doDelete()`, `doOptions()`, `doPut()` e `doTrace()`. Se o *servlet* a ser criado vai operar no contexto da web (quase sempre é o caso), então estender a classe `HttpServlet` é mais vantajoso do que estender a classe `GenericServlet`.

A partir do mecanismo geral descrito neste texto, os detalhes de programação incluem as interfaces e classes auxiliares, além dos métodos e parâmetros utilizados. Por exemplo, várias classes específicas são utilizadas como parâmetros dos métodos citados antes. Também existe suporte ao manuseio de "*cookies*", através da classe `javax.servlet.http.Cookie`, e suporte ao manuseio de informações referentes à sessão em andamento sem a utilização de *cookies*, através da classe `javax.servlet.http.HttpSession`. Observe que o *servlet* pode utilizar os recursos do pacote JDBC de Java para acessar bancos de dados, quando isso for necessário para atender as requisições recebidas. Informações detalhadas de como programar *servlets* foge do objetivo deste texto. Elas podem ser encontradas, inclusive com exemplos de código completos, em livros como [Deitel 2001].

2.3.3. RMI (Remote Method Invocation)

A invocação remota de métodos permite que um objeto Java em um computador invoque o método de outro objeto, em outro computador, como se fosse um objeto local. A tecnologia RMI está baseada na Chamada Remota de Procedimento (RPC - *Remote Procedure Call*), apresentada antes. A princípio, mecanismos para RMI podem ser implementados para qualquer linguagem orientada a objetos. No caso de Java, a grande vantagem está na homogeneidade. Por exemplo, como todas as máquinas virtuais Java representam os dados da mesma maneira, não existe a necessidade de conversão de dados.

Além disso, o mecanismo de serialização de objetos em Java permite que objetos completos sejam passados como parâmetros de invocações remotas. O principal propósito do RMI é o mesmo do RPC: facilitar a programação de uma aplicação distribuída na medida que esconde do programador a maioria dos detalhes relacionados com a comunicação em ambiente distribuído. Uma vantagem do RMI-Java sobre outras soluções, como CORBA, é a construção de interfaces usando a própria linguagem Java, sem a necessidade de uma linguagem separada para a descrição de interfaces.

Em resumo, os passos para construir uma aplicação simples que utiliza RMI são três. Inicialmente é necessário definir uma interface remota que descreve como o cliente e o servidor se comunicam um com o outro. Em seguida, é necessário criar o código do programa servidor que implementa a interface remota definida antes. Finalmente, deve-se criar o programa cliente que, utilizando uma referência da interface remota, invoca os métodos da implementação remota da interface definida antes. Obviamente, uma vez definida a interface remota, os programas cliente e servidor podem ser desenvolvidos em paralelo.

A interface remota define os métodos remotos que o programa cliente poderá invocar. Toda interface remota estende a interface `java.rmi.Remote` e inclui a declaração dos métodos que serão suportados pelo objeto remoto. Basta um objeto implementar uma interface derivada da interface `Remote` para tornar-se apto a ter seus métodos invocados remotamente, a partir de uma outra máquina virtual, com as devidas verificações de segurança.

Métodos remotos estão sujeitos a problemas na comunicação entre diferentes computadores ou mesmo problemas na máquina que executa o objeto servidor. Em função disso, todo método remoto pode lançar uma exceção `RemoteException` e o objeto cliente deve estar preparado para isto. Por exemplo:

```
import java.rmi.*;
public interface ServidorTempo extends Remote{
   public java.util.Date getDate( )
       throws RemoteException;
   }
```

A classe servidora deverá implementar a interface remota. Ela deve estender a classe `java.rmi.server.UnicastRemoteObject`, a qual serve de modelo para todos os objetos que atuam como servidores remotos. O construtor dessa classe cuida dos detalhes de baixo nível para permitir que o objeto receba conexões remotas através de *sockets*, e, portanto deve ser sempre chamado.

Como última tarefa do lado servidor, é necessário definir o nome do objeto servidor, o qual será usado pelos clientes para localizá-lo no momento de efetuar as invocações remotas. O nome tem a forma:

```
//computador:porta/nome_objeto_remoto
```

onde computador e porta dizem respeito ao ponto de contato com o `rmiregistry` responsável pela sua localização e o `nome_objeto_remoto` refere-se ao nome do objeto servidor.

O `rmiregistry` é o programa servidor de nomes responsável por fazer o mapeamento entre "`nome_objeto_remoto`" de um computador e o respectivo objeto executando sobre a máquina virtual local. O `rmiregistry` faz parte do pacote Java e está normalmente vinculado à porta 1099 de cada computador onde existam objetos remotos executando.

O objeto servidor deve fazer o seu registro perante o `rmiregistry` escolhido fornecendo o seu nome. A classe `java.rmi.Naming` é usada para estabelecer o vínculo (*bind*) entre o nome escolhido e o objeto servidor propriamente dito, perante o `rmiregistry` em questão.

O utilitário `rmic`, parte do pacote Java, é responsável por criar uma classe *stub* apropriada para o objeto servidor em questão. Esta classe *stub* deverá executar junto com o objeto cliente. Ela receberá as invocações de método locais do objeto cliente e fará a comunicação com a máquina servidora, usando o suporte RMI para fazer a invocação chegar ao objeto servidor. A versão 1.2 de Java gera apenas um *stub* para executar no lado cliente. Versões anteriores de Java utilizavam uma abordagem diferente.

Antes de o objeto cliente poder fazer invocações remotas ao objeto servidor, ele deve contactar o `rmiregistry` apropriado para localizar o objeto servidor. O `rmiregistry` apropriado, aquele que contém o registro do objeto servidor, é identificado através do endereço de máquina (DNS ou IP) e o número da porta escutada (normalmente a porta 1099). O objeto servidor é identificado por seu nome.

A classe `java.rmi.Naming` fornece suporte para a localização do objeto servidor. A partir do endereço e porta do `rmiregistry` e do nome do objeto servidor, ela não só localiza o objeto servidor em questão como também cria um *proxy* local que, deste momento em diante, será usado pelo objeto cliente para obter os serviços do objeto servidor. Todo o processo de envio de parâmetros, retorno de resultados, comunicação via *sockets*, etc é implementado pelas classes de suporte e pelo *stub*, sendo invisível para o programador da aplicação. O grande poder do RMI reside na simplicidade do mecanismo usado pelo programador de aplicação para criar uma interface de comunicação entre objetos distribuídos. A Figura 2.10 resume todo o processo.

Uma observação importante deve ser feita. Quando um dos parâmetros do método remoto é uma referência para objeto, o que é enviado para o método remoto não é uma referência para o objeto na máquina cliente, mas sim uma cópia completa do objeto em questão. Para isto, o objeto passado como parâmetro deve ser serializável.

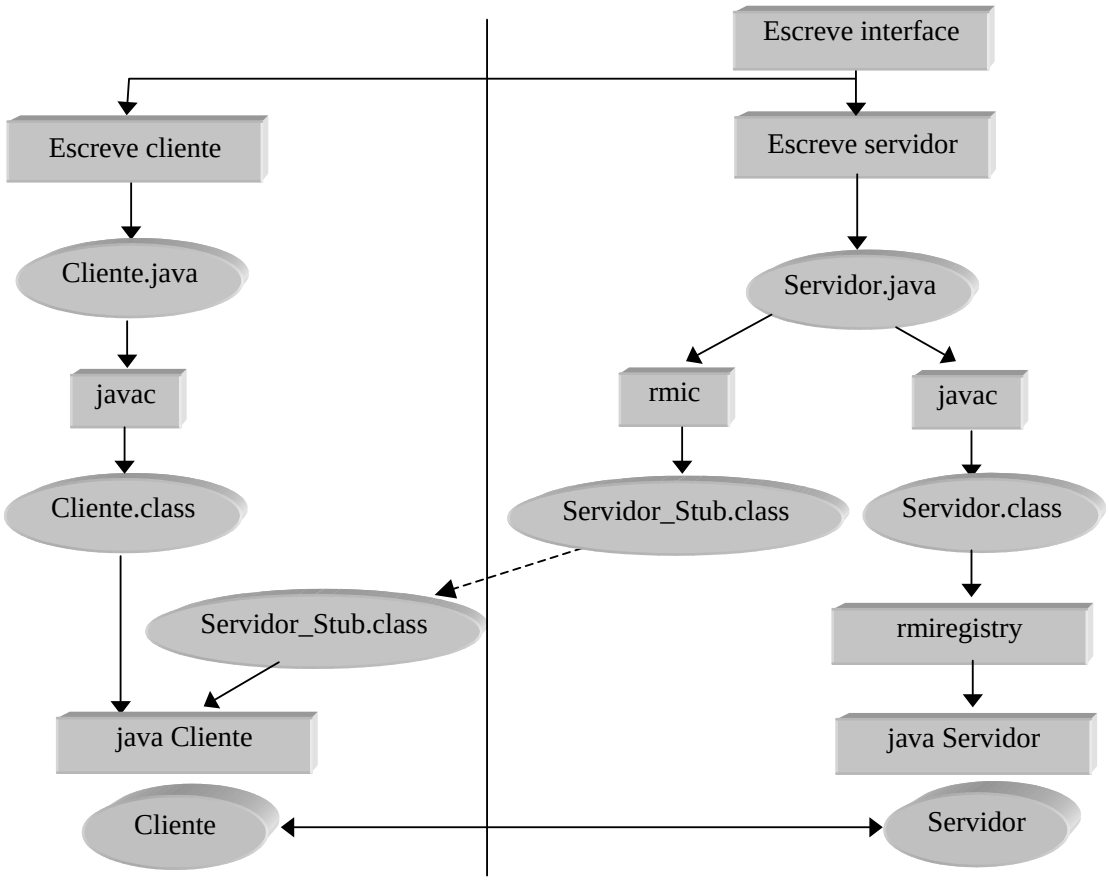

Figura 2.10. Mecanismo geral do RMI.

2.3.4. Jini

A concepção de Jini ([Sun 1999], [Sun 2001], [Edwards 2000]) partiu de uma visão do futuro que está se tornando realidade rapidamente. Esta visão do futuro está baseada na constatação que processadores continuarão a ficar mais rápidos, menores e mais baratos, ao menos pelos próximos anos.

Memória também continuará a ficar mais rápida e mais barata. Dispositivos para entretenimento continuarão a aumentar a sua parcela digital. Como conseqüência, nos próximos anos, até mesmo os dispositivos mais simples poderão conter um processador razoavelmente poderoso. Poderoso o bastante para executar uma versão simples de Java. Ao mesmo tempo, a velocidade das redes locais continuará a aumentar, e elas se farão presentes em todos os lugares, permitindo a conexão em rede de todos os dispositivos com computação embutida.

É possível vislumbrar um mundo onde tudo estará conectado em rede. Embora isso já aconteça em parte hoje nos escritórios, no futuro pode-se esperar este nível de

conectividade também em hotéis, lares, veículos, etc, além de tornar-se mais intenso em ambientes de escritório. É possível imaginar uma casa onde o aparelho de som, a televisão, o vídeo-cassete, o aparelho de DVD, o equipamento de segurança, o ar condicionado, os telefones fixo e celular, até mesmo a iluminação, o rádio-relógio e, é claro, os microcomputadores e impressoras, formarão uma rede local.

Entretanto, usuários buscam soluções em rede que sejam simples e confiáveis. Atualmente, a configuração de redes locais é trabalhosa. Imagine, atualmente, o que é necessário para conectar uma impressora e um microcomputador na rede local do escritório e fazer a configuração necessária para o computador usar a impressora. Projete agora esse esforço para a variedade de dispositivos citados antes, e sem qualquer ajuda de um especialista em informática. A solução atual não é apropriada para o consumo em massa.

O objetivo da tecnologia Jini é simplificar a construção de sistemas distribuídos. Os consumidores simplesmente conectam dispositivos físicos na rede, ou instalam programas em seu computador e os serviços que eles oferecem ficam automaticamente disponíveis para todos os outros equipamentos e aplicações da rede. Embora o espectro de utilização de Jini seja imenso, aparentemente os primeiros segmentos de mercado que serão beneficiados por esta tecnologia são os fabricantes de dispositivos periféricos.

Basicamente, Jini é uma arquitetura para construção de aplicações distribuídas baseada em Java que, acima de tudo, facilita a integração entre provedores de serviços e consumidores de serviços em uma rede local. Serviço aqui possui uma conotação ampla. Vai desde um serviço de impressão – provido por uma impressora –, até um algoritmo de tradução de línguas – implementado puramente em software. Assim, Jini pode ser concebido como um mecanismo de "*plug-and-play*" para serviços em geral no ambiente distribuído.

Em termos práticos, Jini aparece como um conjunto de classes e interfaces Java, acrescidos de alguns programas utilitários e protocolos, cujo objetivo é criar uma federação (ou comunidade) de máquinas virtuais Java. Pessoas, dispositivos, dados e aplicações dentro da federação podem ser dinamicamente conectados uns aos outros para o fornecimento de serviços (compartilhar informações ou realizar tarefas).

2.3.4.1. Protocolos Jini

A Figura 2.11 ilustra a infraestrutura associada com Jini. Um ambiente com diferentes arquiteturas de computadores e sistemas operacionais é tornado homogêneo através da execução de máquinas virtuais Java. A tecnologia RMI fornece o mecanismo de comunicação entre programas executando em diferentes máquinas virtuais através da rede.

Em seguida temos os três mecanismos fundamentais de Jini, os protocolos para "Discovery", "Join" e "Lookup". Os serviços (impressão, etc) são oferecidos acima desses, no nível de aplicação.

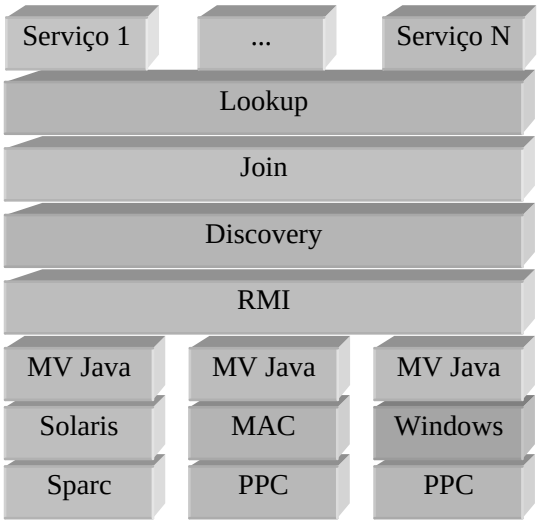

Figura 2.11. Arquitetura Jini.

O elemento que define uma federação Jini é o Serviço de Lookup. O Servidor de Lookup deve executar em algum computador da federação e ele é responsável por manter a informação de "*quais os serviços disponíveis nesta federação*". Qualquer entidade, dispositivo, ou programa, querendo oferecer algum tipo de serviço para os demais membros da federação deve se cadastrar no Servidor de Lookup. Esta operação de cadastramento recebe o nome de "join". Os clientes, por sua vez, consultam o Servidor de Lookup a procura de um servidor específico. Esta consulta pode ser parametrizada de diferentes formas e corresponde a operação "lookup". Para que clientes e servidores possam efetuar respectivamente as operações de "join" e "lookup", eles devem saber como contactar o Servidor de Lookup. A operação de descoberta do Servidor de Lookup é chamada de "discovery". Nas próximas seções, cada uma dessas três operações fundamentais será detalhada.

2.3.4.2. Discovery

O propósito de Jini é a formação *ad-hoc* de federações. Clientes e servidores devem ser colocados em contato sem que exista nenhuma forma prévia de configuração. Por exemplo, ao conectar uma nova impressora ou um novo forno microondas na rede local eles devem automaticamente localizar o Servidor de Lookup dessa rede e fazer o seu cadastramento, oferecendo dessa forma os seus serviços para os demais membros da federação. Da mesma forma, quando uma máquina fotográfica é conectada na rede e ela deseja contactar um servidor de impressão para imprimir algumas fotos, ela deve antes localizar um Servidor de Lookup para através dele localizar os serviços de impressão disponíveis naquela federação. A pergunta aqui é: "*como um dispositivo ou um software recém conectado na rede local localiza o Servidor de Lookup ?*"

Existem 3 formas. Quando uma aplicação ou serviço é ligado, ela pode utilizar o **protocolo de difusão de pedido** ("*Multicast Request Protocol*") que utiliza a capacidade de

difusão da rede local para mandar uma mensagem específica indagando onde naquela rede local estão os Servidores de Lookup. Uma forma alternativa é usar o **protocolo de difusão de anúncio** ("*Multicast Announcement Protocol*"), quando o próprio Servidor de Lookup utiliza a capacidade de difusão da rede local para anunciar sua localização. Finalmente, o **protocolo de discovery direto** ("Unicast Discovery Protocol") é usado quando uma aplicação ou dispositivo conhece o endereço de rede do Servidor de Lookup a ser contactado. Este tipo de "discovery" é especialmente útil quando o servidor sendo cadastrado não se encontra na mesma rede local do Servidor de Lookup. O endereço do Servidor de Lookup é definido na forma de uma URL, por exemplo:

```
jini://servlookup.das.ufsc.br
```

Vários Servidores de Lookup podem existir simultaneamente na mesma rede, com o propósito de prover redundância e um certo grau de tolerância a faltas. É importante observar que o Serviço de Lookup é, por si só, um serviço. Logo, o próprio Serviço de Lookup aparece como um serviço cadastrado no Servidor de Lookup. O mecanismo de "discovery" nada mais é que um método simplificado de efetuar a operação "lookup" quando o serviço procurado é o próprio Serviço de Lookup.

Como resposta ao pedido de "discovery", o Servidor de Lookup envia uma classe Java que será executada na máquina da entidade que fez o pedido. Esta classe será usada pela entidade para acessar os serviços do Servidor de Lookup. Em termos práticos, esta classe permitirá a entidades do tipo servidor realizar a operação "join" junto ao Servidor de Lookup. Ao mesmo tempo, entidades do tipo cliente usarão esta c**l**asse para realizar a operação "lookup". Observe que, o código original das entidades cliente e servidor não precisa conter o código que implementa a comunicação com o Servidor de Lookup, pois este código será fornecido pelo próprio Servidor de Lookup como resposta ao pedido de "discovery". Este tipo de código é chamado na literatura de *proxy*. Tudo que as entidades cliente e servidor precisam conhecer antecipadamente é a interface Java do *proxy* do Servidor de Lookup, a qual é definida em sua forma mais básica na especificação de Jini. Servidores de Lookup mais sofisticados que o básico podem ser criados, mas sua interface Java deverá sempre estender a interface padrão. Dessa maneira, entidades que esperam encontrar apenas a interface básica não terão problemas em usar uma classe que suporta uma interface mais elaborada, pois a interface básica também será suportada. A Figura 2.12 ilustra a operação "discovery".

Espera-se que um dia a tecnologia Jini seja usada para conectar inclusive dispositivos simples como interruptores de luz. No caso de dispositivos simples demais para executar uma máquina virtual Java, é possível implementar a operação "discovery" e, em seguida, a própria operação "join" de uma forma mais simples. Por exemplo, um programa escrito em C, gravado em ROM e embutido no dispositivo é responsável por conversar com o Servidor de Lookup e realizar as operações. Ou ainda, um outro dispositivo com maior capacidade computacional pode fazer o cadastramento em nome de um dispositivo mais simples.

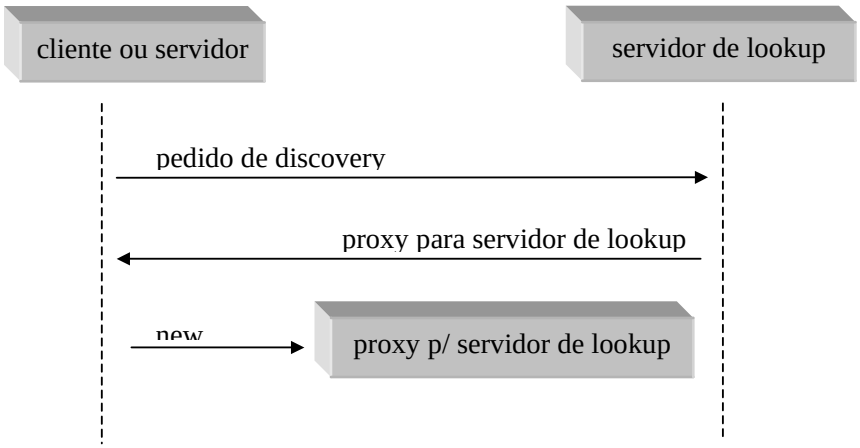

Figura 2.12. Operação "discovery".

2.3.4.3. Join

Uma vez concluída a operação de "discovery", uma entidade que deseja oferecer seus serviços aos membros da federação pode fazer o seu cadastro no Servidor de Lookup, ou nos Servidores de Lookup, caso a operação de "discovery" tenha descoberto mais de um. A operação "join" consiste em efetuar o cadastro do serviço oferecido, o qual será identificado por um nome, por atributos, por uma interface Java suportada, e será associado com um endereço de rede e uma ou mais classes Java, as quais funcionarão como um *proxy*, a ser executado nas máquinas virtuais das futuras entidades clientes. A Figura 2.13 ilustra a operação "discovery" seguida pela operação "join" por parte de um "servidor x" qualquer.

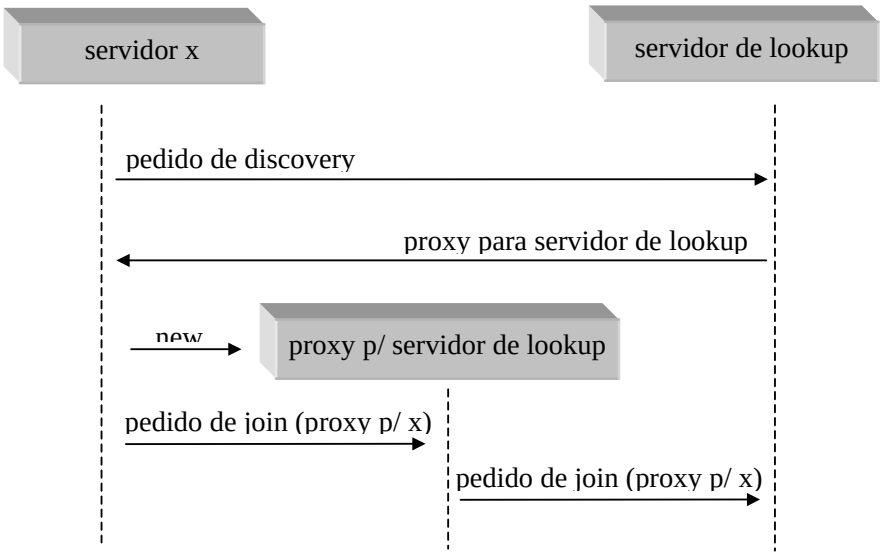

Figura 2.13. Operação "discovery" seguida da operação "join".

2.3.4.4. Lookup

Quando um cliente deseja contactar um servidor capaz de realizar determinado serviço, ele realiza uma operação "lookup" junto ao Servidor de Lookup. A operação "lookup" é basicamente o mapeamento de uma descrição do serviço desejado, fornecida pelo cliente, para uma classe *proxy*, previamente fornecida pelo servidor capaz de realizar aquele serviço. A descrição do serviço desejado pelo cliente pode ter a forma atributos descritivos, tais como nome do serviço e versão, ou ter a forma da interface Java a ser suportada pelo *proxy* e usada para acessar o serviço em questão.

Uma vez que o cliente obteve uma classe *proxy* para o serviço desejado, ela é instanciada na máquina virtual Java do cliente e a aplicação cliente passa a chamar os métodos desse objeto local para obter o serviço. Existem diversas formas pelas quais o *proxy* pode atender às requisições do cliente. A Figura 2.14 ilustra a operação "lookup" por parte de um "cliente y", e o subseqüente acesso a um "servidor x" qualquer.

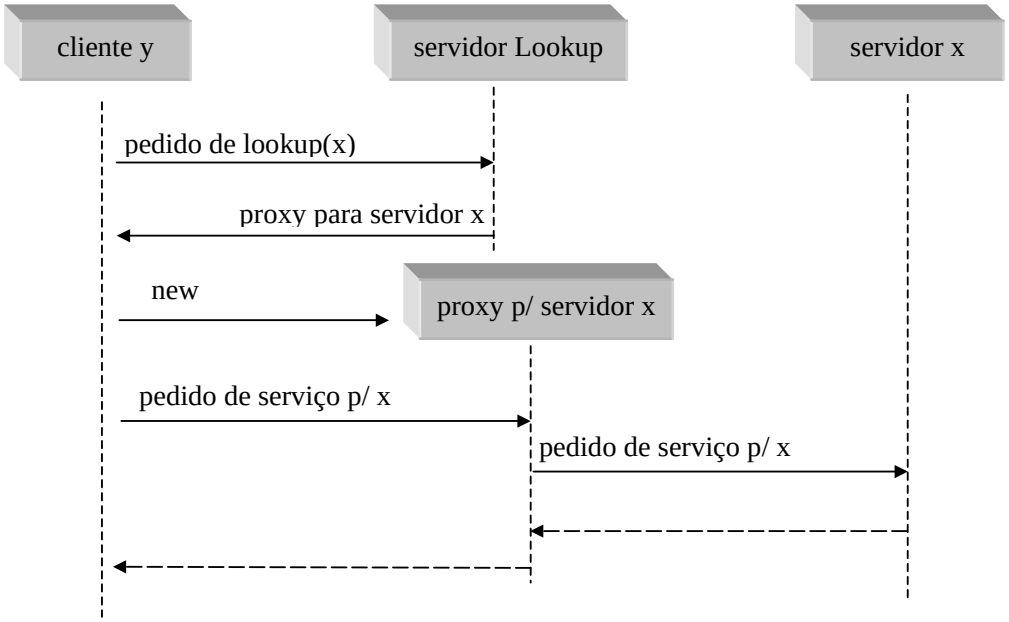

Figura 2.14. Cliente usa "lookup" e *proxy* para acessar servidor.

O *proxy* pode realizar ele mesmo o serviço, quando o serviço em questão for uma computação que não depende de dispositivos físicos específicos e pode ser realizada na máquina cliente. O *proxy* pode ser um simples *stub* RMI, fornecendo uma porta para o acesso de métodos Java que, executando na máquina servidora, são capazes de fornecer o serviço solicitado. Esses métodos podem acessar recursos físicos específicos presentes na máquina servidora. Por fim, o *proxy* pode ser uma classe complexa, que executa parte do serviço na própria máquina cliente e solicita parte do serviço para métodos executando na máquina servidora. O protocolo entre o *proxy* na máquina cliente e o servidor na máquina servidora é completamente ignorado pelo cliente, que conhece apenas a interface do

serviço, suportada pelo *proxy*, uma classe local para o cliente. Neste último caso o *proxy* pode até mesmo ser uma porta para acesso a objetos escritos em outras linguagens de programação, sendo usado CORBA como meio entre o *proxy* e o servidor que ele representa. As possibilidades são ilimitadas.

Como o protocolo entre o *proxy* e o servidor é ignorado pela aplicação cliente, diferentes soluções podem ser criadas para um mesmo tipo de serviço. Por exemplo, uma interface "Impressora" pode ser criada para definir os serviços suportados por uma impressora de rede. A partir disso cada fabricante constrói as suas impressoras, completamente diferentes entre si. Cada fabricante também escreve um *proxy* apropriado para cada impressora e cria um protocolo proprietário para o seu *proxy* trocar informações com a sua impressora. Uma ampla variedade de soluções é ignorada pela aplicação cliente, que se preocupa apenas em obter junto ao Servidor de Lookup um *proxy* capaz de suportar a interface "Impressora" e depois em chamar os métodos dessa interface. A garantia de compatibilidade entre o *proxy* e a impressora é obtida pelo fato da própria impressora ter registrado o *proxy* (operação "join") junto ao Servidor de Lookup.

Alguns *proxy* podem oferecer, além de uma interface de programação, uma interface para o usuário. O *proxy*, ao executar na máquina cliente, pode apresentar uma interface gráfica ao usuário e permitir que este usuário acesse diretamente os serviços remotos. Neste caso, o programa cliente pode desconhecer completamente a funcionalidade associada com o serviço, e simplesmente servir como um hospedeiro para o *proxy*, que faz todo o trabalho. Nesta configuração, o *proxy* é muito semelhante a um *applet*, que cumpre o mesmo papel ao executar no navegador (cliente) e comunicar-se diretamente com a sua máquina de origem para suprir sua funcionalidade. A Figura 2.15, a Figura 2.16 e a Figura 2.17 ilustram todo o processo básico de colocar um cliente e um servidor em contato.

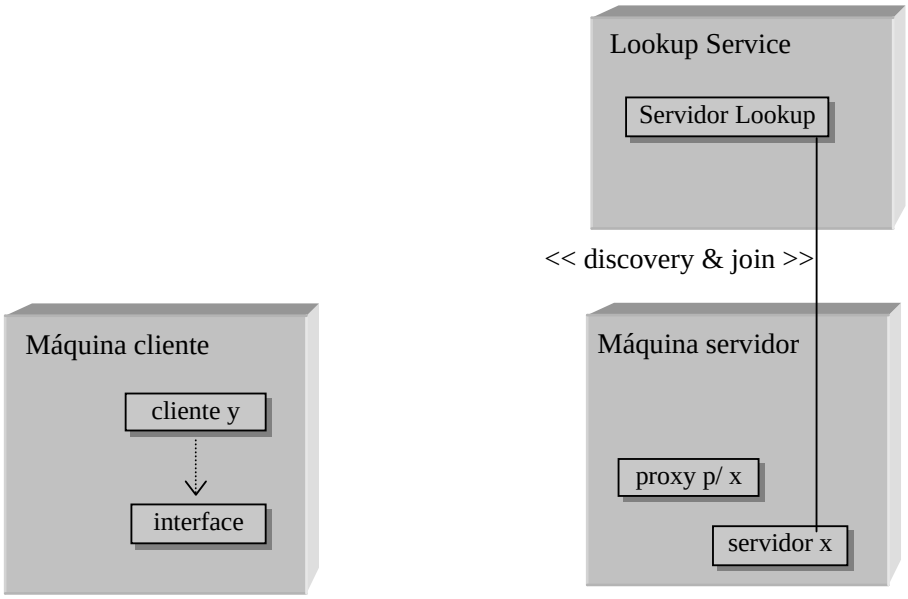

Figura 2.15. Situação antes do "discovery" e "join" do servidor.

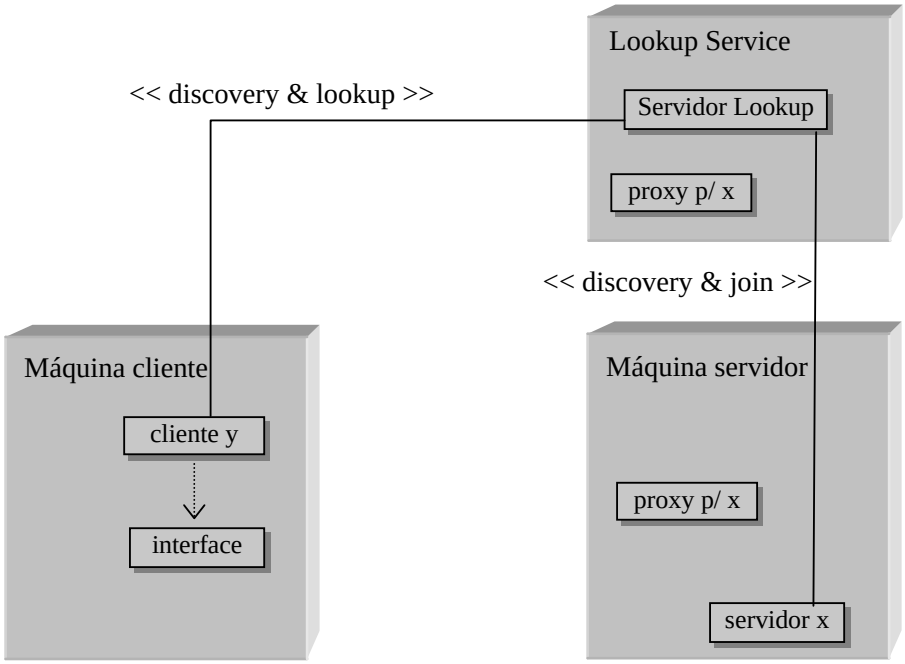

Figura 2.16. Situação após o "join" e antes do "lookup".

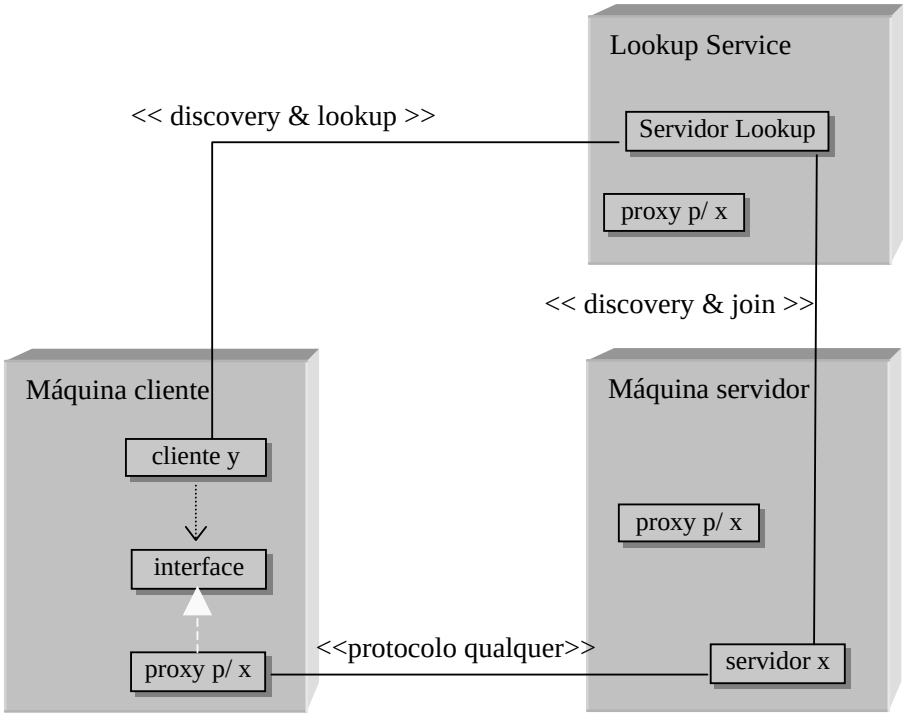

Figura 2.17. Situação após o "lookup".

2.3.4.5. Outros mecanismos de Jini

Além das operações de "discovery", "join" e "lookup", a infra-estrutura Jini disponibiliza outros recursos para o desenvolvedor de aplicações distribuídas. Esta seção descreve rapidamente as facilidades para "leasing", "eventos distribuídos" e "transações distribuídas", depois conclui com alguns comentários gerais.

Através do mecanismo denominado "leasing" um dispositivo ou programa servidor registra-se no Serviço de Lookup por tempo limitado, especificado no momento da operação "join". Quando o prazo definido esgota-se, o dispositivo ou programa deve renovar o seu cadastramento. Se, por alguma razão, o dispositivo ou programa torna-se indisponível, seu registro será automaticamente removido do sistema após o prazo especificado. O fato das entradas sem valor do Serviço de Lookup serem removidas depois de algum tempo torna o sistema "auto-limpante". O mecanismo tem esse nome pois é considerado que o servidor "aluga" espaço no Servidor de Lookup para fazer o seu registro por um período de tempo. Se ele não renovar o aluguel, será despejado, ou seja, seu registro será removido e o espaço liberado.

Jini também inclui uma API para facilitar a construção de aplicações distribuídas baseadas na idéia de eventos. O mesmo mecanismo de eventos usado na construção de JavaBeans e presente nas *Java Foundation Classes* é estendido para o ambiente distribuído. Eventos podem ser usados, por exemplo, para permitir que um Servidor de Lookup notifique um dado cliente quando o servidor que aquele cliente estava esperando finalmente executou a operação "join" e ingressou na federação.

O pacote Jini também inclui uma API para gerenciar transações distribuídas, o qual usa a técnica clássica de confirmação em duas fases (*two-phase commit*). Transações facilitam a construção de aplicações distribuídas com relacionamentos complexos entre vários objetos distribuídos.

Jini não inclui nenhuma preocupação com segurança e controle de acesso. A filosofia básica é que qualquer máquina virtual Java que tenha acesso ao Servidor de Lookup pode usufruir os serviços registrados nele. É claro que esquemas de autenticação e controle de acesso podem ser empregados pelos Servidores de Lookup ou diretamente por servidores específicos, mas esta questão não é abordada pelo pacote Jini e deve ser tratada como algo a mais, parte da aplicação. Na verdade, Jini perde muito do seu apelo quando considerações sobre controle de acesso devem ser feitas. Jini funciona melhor em ambientes onde uma rede local define a federação e todos os serviços registrados estão disponíveis para todas as aplicações dentro dos limites da federação, como em um escritório ou uma residência. Embora existam críticas sobre a escalabilidade de Jini em termos, por exemplo, da Internet, o seu propósito principal sempre foi facilitar a conexão entre componentes distribuídos no contexto de uma rede local, o que é feito sem problemas.

Na API de Jini existe suporte para o conceito de grupo. Durante a operação "discovery" um servidor ou cliente pode especificar os grupos que ele gostaria de localizar. O resultado da operação "discovery" retornará apenas os Servidores de Lookup que suportam os grupos solicitados. O conceito de grupo pode ser usado para criar uma divisão

interna no caso de federações muito grandes. Entretanto, não existe nenhum controle de acesso embutido em Jini e na maioria das vezes espera-se que a divisão em grupos seja ignorada e todos os componentes refiram-se ao grupo *default* (sem nome).

Finalmente, Jini é uma tecnologia Java, isto é, feito para o mundo Java, e não para ser usado em outros ambientes. Jini implementa algo similar a um "*plug-and-play*" para aplicações distribuídas, onde os elementos conectados podem ser dispositivos físicos como impressoras e videogames, ou aplicações como tradutores e compactadores. Todo o mecanismo está baseado na capacidade do cliente obter dinamicamente um *proxy* que será executado localmente, fato que é enormemente facilitado pelo ambiente confortável criado por uma única linguagem de programação e por um conjunto homogêneo de máquinas virtuais.

2.4. Especificações CORBA

2.4.1. Introdução

Nos últimos anos, a área de sistemas distribuídos tem avançado na direção da padronização aberta. Esta tendência vem no sentido de combater soluções proprietárias que determinam sempre altos custos em desenvolvimento de sistemas. Sistemas abertos abrangem um significante leque de tecnologias e especificações que, de forma diversa aos ambientes proprietários, permitem soluções mais adaptadas à complexidade e à heterogeneidade de sistemas distribuídos.

As especificações CORBA (*Commom Object Request Broker Architecture*) foram criadas com o objetivo de permitir que as dificuldades existentes na programação em ambientes distribuídos sejam superadas. Seu conjunto de especificações abertas padronizadas pela OMG[2] (*Object Management Group*) [OMG 1998] já se consolidou como um padrão para sistemas distribuídos. A utilização das especificações CORBA cobre às necessidades de redução de custos, portabilidade, interoperabilidade e flexibilidade nos sistemas contemporâneos.

Esta seção introduz uma revisão geral de objetos distribuídos em sistemas abertos, dando ênfase às especificações OMG.

2.4.2. Arquitetura CORBA

As especificações CORBA definem um conjunto de mecanismos padronizados e de conceitos para construir objetos distribuídos. Segundo essa arquitetura, métodos de objetos remotos podem ser ativados de forma transparente em ambientes distribuídos heterogêneos através de um ORB (*Object Request Broker*). O ORB, num sentido mais genérico, é um canal de comunicação para objetos distribuídos que interagem entre si usando o modelo de interações cliente-servidor. Objetos distribuídos encapsulam estados e se fazem acessíveis

[2] É uma organização formada por centenas de empresas com o objetivo de especificar um conjunto de padrões e conceitos para a programação orientada a objetos em ambientes distribuídos abertos.

através de uma interface bem definida, construída a partir de uma linguagem de definição de interfaces (IDL: *Interface Definition Language*). Usando CORBA é possível também obter interoperabilidade entre objetos desenvolvidos em diferentes linguagens de programação (C, C++, Ada, Cobol, Java, etc.), onde as diferenças entre as mesmas são ocultadas através do mapeamento adequado da IDL para a linguagem de programação desejada.

A IDL é uma linguagem declarativa, sem nenhuma estrutura algorítmica, com sintaxes e tipos predefinidos, baseados na linguagem C++. O uso desta linguagem de definição de interface permite tratar das heterogeneidades do ambiente computacional que incluem, além das diferenças de linguagens de programação, também o trato com diferentes tipos de máquinas e sistemas operacionais.

2.4.3. O modelo de objetos

Esta seção apresenta a terminologias e os aspectos conceituais do *modelo de objetos* definido pela OMG [OMG 1998]. Um sistema de objetos é uma coleção de objetos que isola os requerentes do serviço (os clientes) dos provedores dos serviços (os servidores) através de interfaces bem definidas. Um *cliente* — não necessariamente um objeto — faz requisições a um objeto *servidor*. Uma *requisição* pode ser interpretada como um evento que ocorre em um instante de tempo particular, provocando um processamento no servidor. Estão associadas a uma requisição, as informações sobre o endereço do objeto alvo, sobre a operação requisitada e sobre os parâmetros associados à operação.

Neste modelo, os objetos só podem ser criados ou destruídos por meio de requisições. Isso se deve ao fato de não existirem mecanismos de construtores e destrutores nas especificações IDL/CORBA — como em algumas linguagens de programação orientadas a objetos. A criação de um objeto por um cliente é materializada na forma de uma *referência de objeto* que denota o novo objeto. Uma *referência de objeto* é, basicamente, um identificador associado a um objeto particular. Especificamente, uma referência de objeto identificará o mesmo objeto cada vez que a mesma é usada numa requisição. Contudo, um objeto pode ser identificado por diferentes referências de objeto.

Todo objeto deste modelo é disponível a seus clientes através de sua *interface*, que descreve um conjunto de operações (métodos) que podem ser requisitados pelos clientes. Um objeto satisfaz a uma dada interface se especificado de forma que possa atender a cada requisição de execução de métodos definidos na sua interface. O uso da interface permite ocultar do cliente os aspectos relacionados à implementação do serviço (métodos do objeto servidor). Mudanças na forma de implementar um serviço não implicam, necessariamente, na alteração da interface deste serviço, não provocando, dessa forma, qualquer ônus ao cliente. O conceito de *interface* está diretamente relacionado ao conceito de *classe* no modelo tradicional da programação orientada a objetos. De forma similar às classes, no modelo de objetos da OMG é permitido aplicar técnicas de encapsulamento, herança e polimorfismo.

2.4.4. A arquitetura OMA

O objetivo da OMG é permitir o crescimento da tecnologia a objetos e influenciar sua direção no sentido de cumprir os requisitos e conceitos estabelecidos na OMA (*Object Management Architecture*). A arquitetura OMA fornece uma infra-estrutura conceitual para todas as especificações da OMG. As regras definidas por esta arquitetura de referência visam auxiliar a construção de componentes de software interoperáveis, reutilizáveis e portáveis. Esta arquitetura é composta de quatro principais elementos (Figura 2.18): o ORB, os objetos de serviço, as facilidades comuns e os objetos da aplicação.

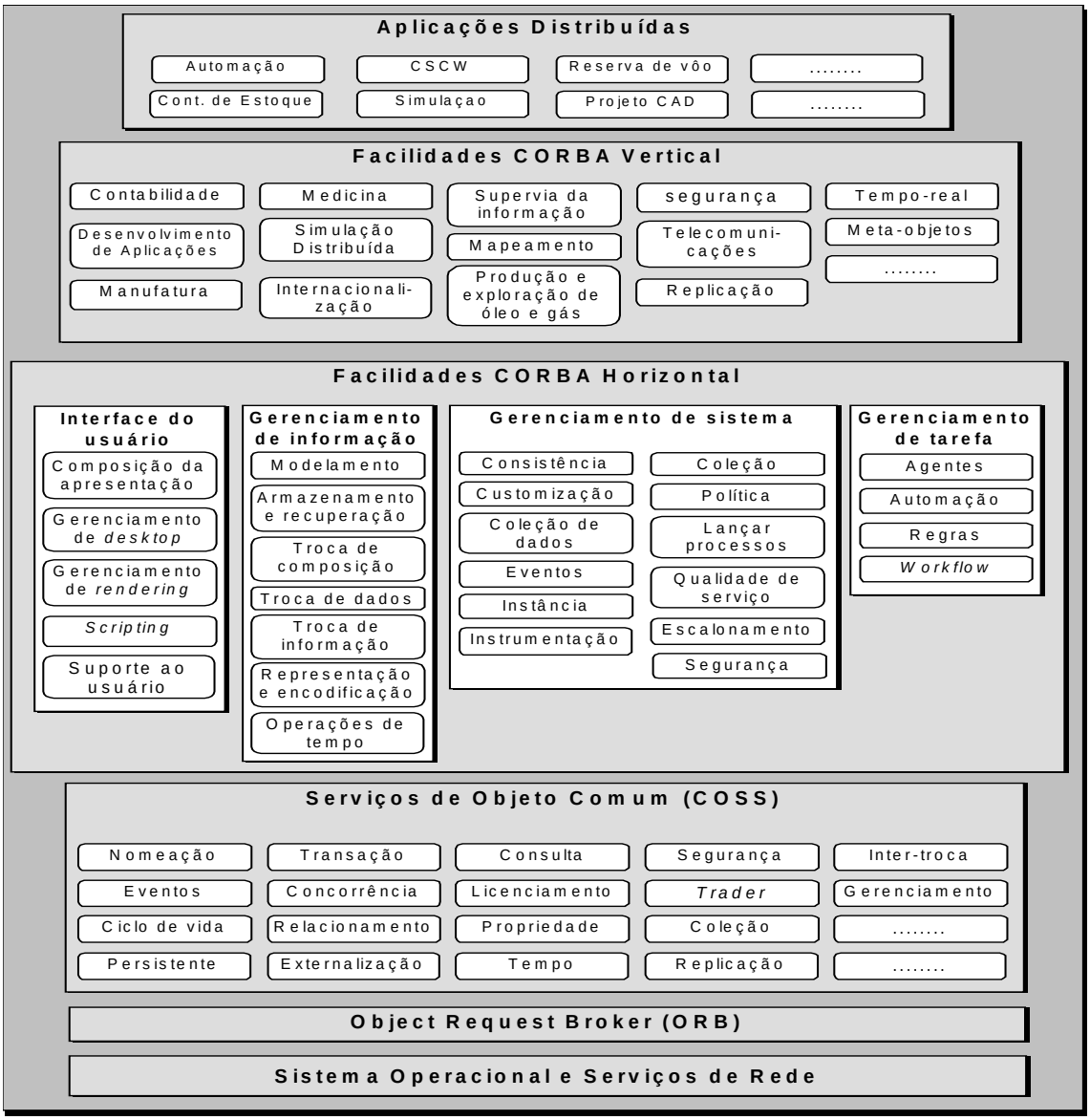

Figura 2.18. Arquitetura OMA (*Object Management Architecture*).

2.4.5. Estrutura do ORB

As especificações CORBA definem um *broker*: o ORB que implementa abstrações e semânticas da comunicação entre objetos em um sistema distribuído (Figura 2.19). O ORB fornece os mecanismos de representação de objetos e de comunicação para que seja possível o processamento das requisições no lado servidor em um ambiente computacional aberto. O ORB, num sentido mais genérico, pode ser entendido como uma via de comunicação para que objetos heterogêneos possam interoperar.

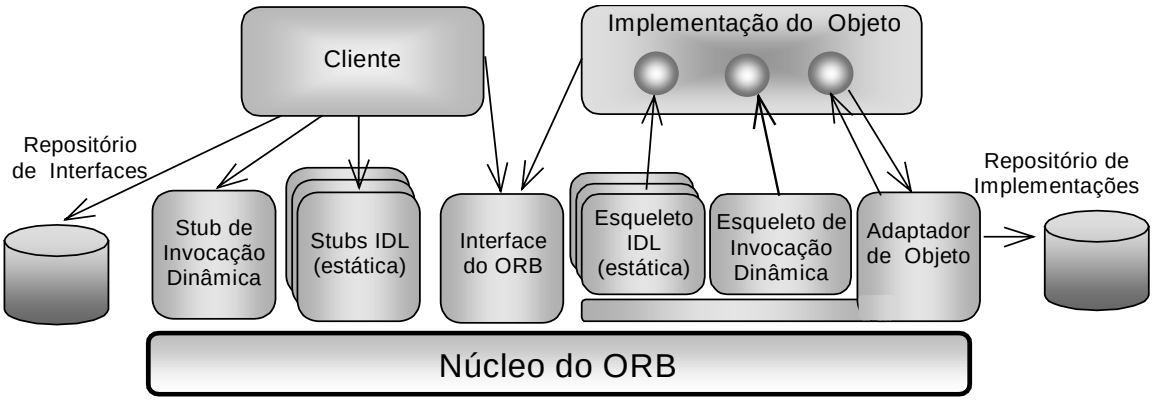

Figura 2.19. Arquitetura CORBA (Common Object Request Broker Architecture).

As interações no ambiente CORBA, seguindo o modelo cliente/servidor, utilizam primitivas de um mecanismo de RMI (*Remote Method Invocation*) nas interações deste modelo. A grande diferença do RMI em relação ao tradicional RPC (*Remote Procedure Call*) está no fato que métodos remotos devem dar suporte a passagem de objetos como argumentos e resultados de métodos.

As *stubs* (*stubs* e esqueletos IDL no modelo CORBA) necessárias que implementam o RMI são geradas a partir da tradução da interface IDL do servidor. Uma chamada do cliente de um método de um servidor remoto (requisição do cliente) é então transmitida através da rede usando estas *stubs* que executam a serialização/deserialização (*marshalling*/*unmarshalling*) de parâmetros e objetos, lidando com as transformações de uma chamada em uma mensagem correspondente e vice-versa.

O ORB, em uma chamada de cliente, através de seu núcleo, localiza o objeto remoto, transporta os dados e transfere o controle para a *Implementação do Objeto* servidor através do *Esqueleto* IDL correspondente. Quando o processamento da requisição termina, os resultados são retornados para o ORB, que os entrega ao cliente de forma transparente da localização do serviço.

Para executar a requisição, a Implementação de Objeto é suportada por alguns serviços do ORB, disponíveis através do *Adaptador de Objetos*. O Adaptador de Objetos se situando no topo dos serviços de comunicação do *núcleo de ORB*, além de passar requisições, fornece os serviços ORB: geração e interpretação de referências de objetos, invocação de métodos, ativação e desativação de objetos, mapeamento de referências de

objetos para implementações de objetos, registros de implementações, etc. As especificações definem um *Adaptador de Objetos Portável* (POA - *Portable Object Adapter*), que pode ser usado pela maioria dos objetos que possuem implementações convencionais. Algumas implementações de objeto podem ter requisitos especiais, exigindo adaptadores de objeto específicos. Nada impede implementadores de estender o POA ou de desenvolverem adaptadores mais adequados às suas necessidades.

A *Interface do ORB*, segundo as especificações CORBA, deve ser idêntica para as diferentes implementações CORBA. Devido ao fato que a maioria das funcionalidades necessárias ao funcionamento das interações da aplicação serem oferecidas através dos adaptadores, *stubs* e dos *skeletons*, nesta interface é oferecida apenas um pequeno número de operações que são comuns tanto a clientes como a implementações de objetos. Esta interface do ORB é acessada diretamente por objetos cliente e servidor.

Na requisição de um serviço, o cliente pode realizar uma invocação no servidor de duas formas: estática através de *Stubs* IDL ou dinâmica através da interface de invocação dinâmica (DII). Em ambas abordagens a implementação do objeto (o servidor) desconhece o tipo de invocação utilizada na requisição pelo cliente. Na invocação estática, o cliente faz a invocação de um método no objeto servidor se utilizando de *Stubs* IDL apropriados, gerados no processo de compilação do arquivo de definição de interface IDL correspondente. Esta *stub* pode ser entendida como uma representação local do objeto remoto.

Para invocações dinâmicas, as interfaces de objetos disponíveis via CORBA devem estar armazenadas no sistema (*repositório de interfaces*). Uma invocação dinâmica permite ao cliente dinamicamente construir e invocar métodos no servidor usando interfaces DII. Este tipo de chamada oferece uma grande flexibilidade para sistemas dinâmicos: o cliente determina o objeto a ser invocado, o método a ser executado e o conjunto de parâmetros desse método, em tempo de execução, através de uma seqüência de chamadas. A Figura 2.20 ilustra esta seqüência de chamadas feitas no repositório de interfaces e na interface DII.

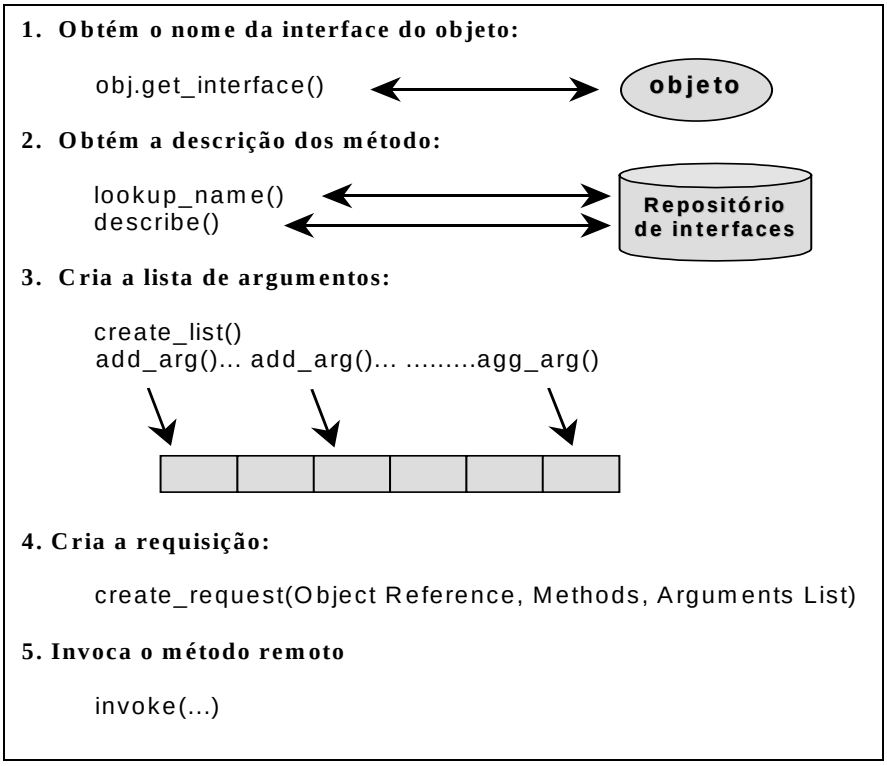

Figura 2.20. Construção de uma chamada através da DII

O *Repositório de Interfaces* contém informações de interfaces IDL em uma forma disponível em tempo de execução para as invocações dinâmicas. Usando a informação disponível no repositório é possível a um programa encontrar um objeto remoto mesmo desconhecendo sua interface, seus métodos e parâmetros, e sua forma de ativação. Já o *Repositório de Implementações* (Figura 2.19) contém informações que permitem ao ORB localizar e ativar implementações de objetos.

2.4.6. Objetos de serviço – *COSS*

Para o desenvolvimento de aplicações de objetos distribuídos, a OMG oferece um conjunto de serviços oferecendo funcionalidades básicas para compor ou dar suporte aos objetos da aplicação. No padrão CORBA, essas funcionalidade adicionais são providas como objetos de serviço, isto é, objetos CORBA com interfaces IDL bem definidas. Por exemplo, a concorrência e a persistência que são necessárias em algumas aplicações não são incluídas pela OMG como mecanismos do núcleo do ORB. Esses serviços e muitos outros são padronizados pela OMG, dentro da arquitetura OMA (Figura 2.18), na forma de objetos de serviço de serviços ou *COSS* (*Common Object Services Specifications*) [OMG 1997a].

Esta coleção de serviços (interfaces de objetos) pode ser entendida como uma extensão ou uma complementação das funcionalidades do ORB. A definição destes serviços padronizados foi a forma encontrada pela OMG para não estender o ORB ou

introduzir novos adaptadores de objetos quando da necessidade de aumentar funcionalidades de *middleware*.

Uma especificação de objeto de serviço usualmente consiste de um conjunto de interfaces e de uma descrição do comportamento do serviço, que podem ser utilizados por objetos de aplicação e outros objetos de serviço. A sintaxe usada para especificar as interfaces é a OMG IDL. A semântica que especifica o comportamento do serviço é expressa, em geral, em termos de objetos, operações e tipos de dados padronizados [OMG 1997a]. A OMG tem definido até agora vários *COSS*, tais como:

- **Serviço de ciclo de vida**, que define serviços e convenções para criar, destruir, copiar e mover objetos. Devido aos ambientes baseados no CORBA suportar objetos distribuídos, este serviço define serviços e convenções que permitem ao cliente realizar operações nesse serviço remotamente;
- **Serviço de persistência**, que introduz uma interface única para salvar estados persistentes dos objetos em servidores de armazenamento, incluindo objetos de base de dados. O estado do objeto é considerado composto de duas partes: *o estado dinâmico* que está tipicamente na memória e que envolve informações que não necessariamente devem ser preservadas e o *estado persistente* que contém as informações necessárias para reconstruir um estado dinâmico.
- **Serviço de nomes**, que determina as necessidades para a implementação e administração de nomes e seus contextos em ambientes CORBA. Objetos a partir de um ORB localizam objetos remotos por nomes, mesmo que localizados em outros ORBs. Uma associação entre nome (nome simbólico) e uma referência de objeto é chamada de *name binding*. Neste caso, um *name context* pode ser visto como um objeto que contém na sua representação um conjunto de *name bindings*. Em um contexto de nomes cada nome é único. Diferentes nomes podem designar um mesmo objeto no mesmo ou em diferentes contextos simultaneamente. A Figura 2.21 ilustra a organização de um objeto `NamingContext` definido na especificação *Cosnaming* – serviço de nomes do CORBA.

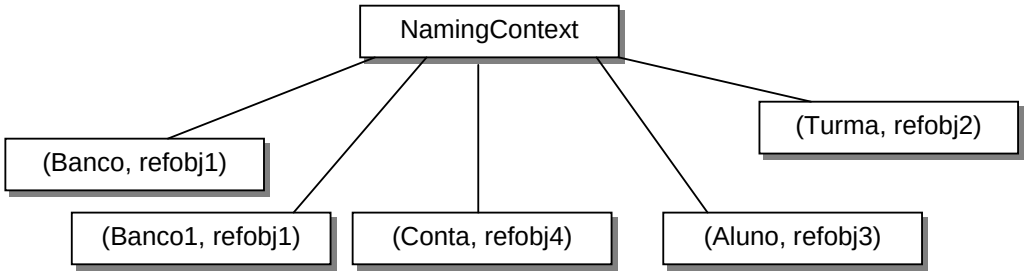

Figura 2.21. NamingContext no CosNaming

O serviço de nomes no CORBA pode usar contextos federados (*federated naming contexts*), fornecendo desta maneira, um suporte hierarquizado para gestão de contextos e nomes em sistemas distribuídos.

Nas especificações CORBA uma referência de objeto é definida numa forma padrão conhecida como IOR (*Interoperable Object Reference*). A IOR [OMG 1998] é uma seqüência de caracteres que quando convertida para o formato adequado fornece as informações de endereço IP da máquina, a porta, um ponteiro local para o objeto a ser acessado e algumas informações de controle relacionadas com a própria IOR. Cada objeto distribuído deve ter sua própria IOR registrada em um *NamingContext*. Com isto o cliente necessitando da IOR de um objeto servidor deve, através do envio do nome simbólico do mesmo, obtê-lo no *NamingContext* onde este objeto foi registrado. A Figura 2.22 apresenta um diagrama temporal separando em fases os procedimentos no suporte dado pelo serviço de nomes às interações cliente/servidor.

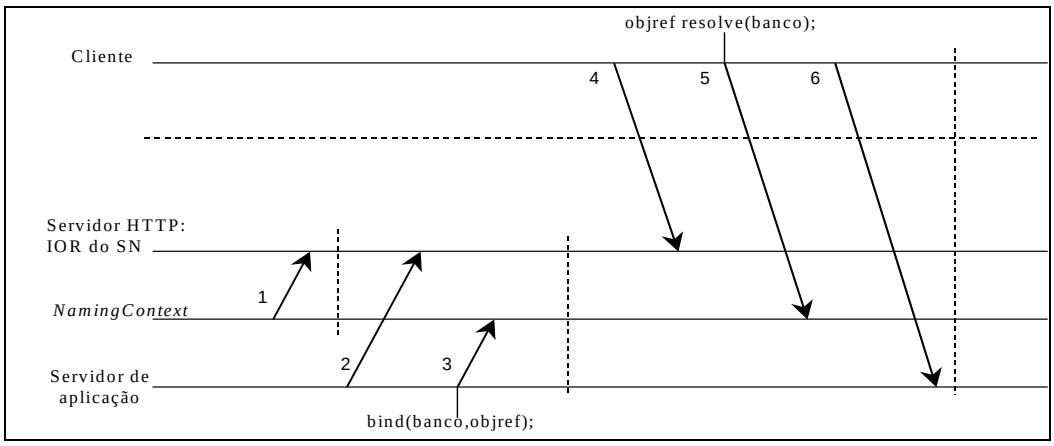

Figura 2.22. Interações para resolução de nomes no CORBA.

Um objeto *NamingContext* na sua iniciação (passo 1 da Figura 2.22) deve colocar a sua própria IOR num repositório (por exemplo, serviço de diretórios, servidor http, ou ftp), tornando-a disponível a objetos CORBA em geral para que possam ter acesso ao mesmo. No exemplo da figura, o *NamingContext* tem a sua IOR disponível através de um servidor http, o que facilitaria o acesso ao mesmo via *Internet*. Uma vez o *NamingContext* em execução, objetos para se tornarem visíveis via CORBA registram suas IORs neste serviço de nomes. Todo objeto servidor então, quando de sua iniciação registra a sua IOR em um *NamingContext*. Para isto é necessário buscar a IOR do *NamingContext* (passo 2 da figura), o que é feito com o método resolve_initial_reference do serviços da interface ORB. Após obter a IOR do *NamingContext* o servidor de aplicação pode invocá-lo para se registrar (passo 3).

Os procedimentos de um cliente para viabilizar a sua comunicação com o servidor de aplicação devem iniciar com a obtenção da IOR do *NamingContext* usando o método `resolve_initial_reference` (passo 4 da Figura 2.22). Com a posse dessa IOR (disponível em servidor http), o cliente usando o nome do objeto servidor invoca o *NamingContext*, pedindo a resolução do nome (passo 5 da figura). Se nome existir dentro do contexto indicado, o serviço de nomes retorna a IOR associada ao nome enviado. A partir daí o cliente obtém o *binding* desejado podendo fazer as invocações ao servidor de aplicação (passo 6 da figura).

- O **serviço de notificação de eventos** (*CosEventComm*) define uma outra forma de comunicação. O serviço define um objeto *canal de eventos* que coleta e distribui eventos entre os componentes através do ORB, que desacopla as comunicações entre objetos cliente e servidor (comunicação assíncrona). O serviço define duas funções: a de fornecedor de eventos e a de consumidor de eventos. Os objetos podem registrar dinamicamente seus interesses em termos da recepção de eventos específicos, assumindo então o papel de consumidores. Os eventos são produzidos pelo fornecedor de eventos.
  Neste serviço são definidas duas abordagens: os modelos *Pull* e *Push*. O modelo *Push* permite ao fornecedor do evento iniciar a transferência do evento ao consumidor. Já o modelo *Pull* caracteriza as transferências a partir de requisições dos consumidores. A Figura 2.23 ilustra o modelo *Push*.

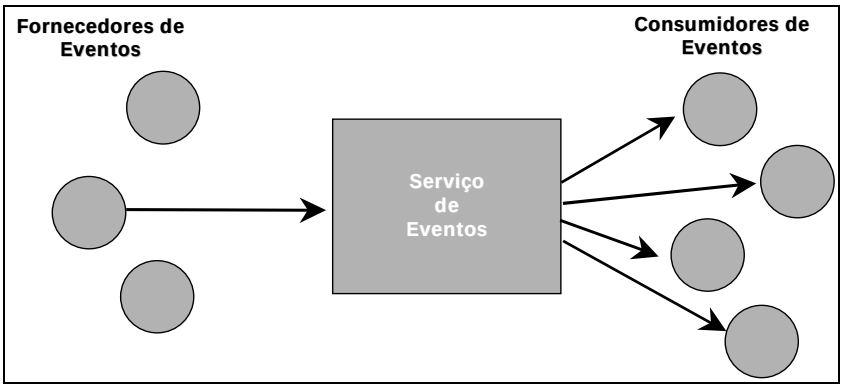

Figura 2.23. Serviço de eventos atuando segundo a abordagem *Push*

- **Serviço de tempo**: este COSS define as interfaces de um serviço de tempo global para ambientes normalmente distribuídos e heterogêneos. Este serviço é importante, por exemplo, na ordenação dos eventos que ocorrem no sistema. Pode ser usado também para disparar eventos em valores de tempo especificados através de mecanismos de temporização e alarmes ou ainda, para computar intervalos de tempo entre eventos. A representação de tempo deste serviço segue o padrão UTC (*Universal Time Coordinated*).

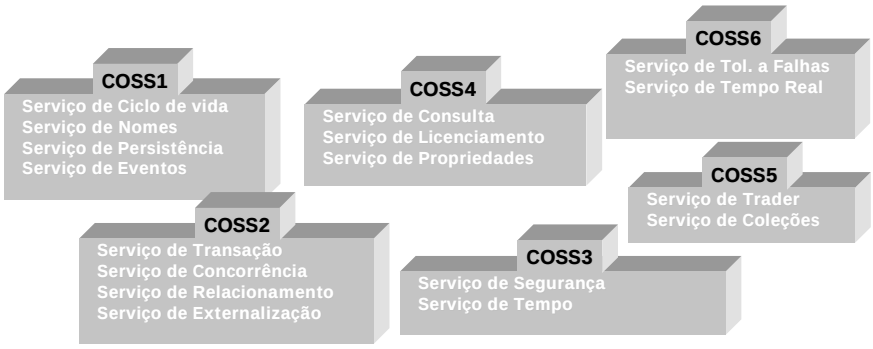

Figura 2.24. A coleção dos COSS definidos até o presente momento pela OMG.

Existem outras especificações de serviços COSS a Figura 2.24 apresenta os grupos destes serviços. Algumas destas serão retomados na seqüência deste texto.

2.4.7. Facilidades comuns

As facilidades comuns são uma coleção de serviços de propósitos gerais utilizados pelas mais diversas aplicações. As facilidades comuns são divididas segundo dois aspectos, as facilidades comuns horizontais e as facilidades comuns verticais (Figura 2.18). As facilidades horizontais podem ser utilizadas por diferentes aplicações, independente da área da aplicação, são divididas segundo quatro categorias: interface de usuário, gerenciamento de informação, gerenciamento de sistema e gerenciamento de tarefa. As facilidades verticais são utilizadas em áreas de aplicação especificas, por exemplo: gerenciamento e controle de imagens, supervias de informação, manufatura integrada por comutador, simulação distribuída, contabilidade, etc.

2.4.8. Interoperabilidade entre ORBs

Nas primeiras especificações CORBA não estavam claras a forma de como garantir a interoperabilidade entre objetos pertencentes a diferentes implementações de ORBs. A política da OMG é a padronização do CORBA e seus serviços, na maioria das vezes, apenas em nível de interface, deixando em aberto aos programadores de *middleware* as escolhas referentes a parte da implementação. Este fato tornou difícil garantir a interoperabilidade entre objetos em diferentes implementações de ORB sem a padronização de um protocolo de comunicação e da sintaxe de transferência das mensagens que deveriam circular entre estes ORBs. Para suprir esta necessidade, a OMG publicou o padrão CORBA 2.0, nestas especificações foram introduzidos os seguintes protocolos (Figura 2.25):

- *Protocolo Inter-ORB GERAL* (GIOP): que especifica um conjunto de formatos para dados a serem transferidos nas comunicações entre ORBs;
- *Protocolo Inter-ORB Internet* (IIOP): que especifica como mensagens GIOP são transmitidas numa rede TCP/IP (GIOP + TCP/IP = IIOP);
- *Protocolo Inter-ORB para Ambientes Específicos* (ESIOPs): que é uma especificação feita para permitir a interoperabilidade de um ORB com outros ambientes de *middleware* (por exemplo, DCE).

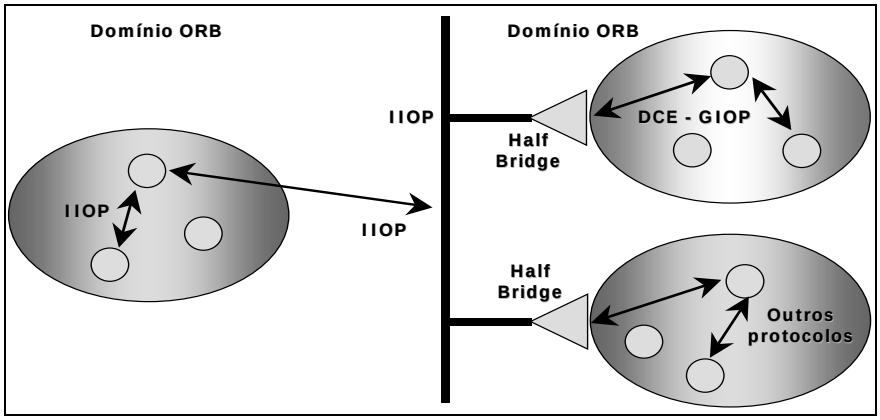

Figura 2.25. Interoperabilidade entre ORBs

Para comunicações de um objeto em ambientes CORBA com outro ambiente não-CORBA foi introduzida a noção de *Half Bridge*. Este mecanismo faz o mapeamento que transforma uma requisição expressa em termos do modelo origem para o modelo do domínio destino, permitindo que chamadas atravessem estes domínios. A idéia deste mapeamento está centrada na conversão do formato de mensagens de um domínio não-CORBA para o GIOP permitindo, então, que estas possam ser transmitidas através do protocolo IIOP (domínio CORBA), e vice-versa.

Na versão 2.0 do CORBA também foi acrescentada a interface *Skeleton* dinâmica (DSI) que é utilizada para a interconexão de ORBs distintos. Quando o POA recebe uma invocação para uma implementação de objeto que não tem um *Skeleton* pré-compilado (abordagem estática), esta invocação é passada para a interface de *Skeleton* dinâmica que ativa a DII de um outro ORB que possua este *Skeleton* onde será feita a invocação; isto caracteriza a interoperabilidade chamada nas especificações de *Bridge Request-Level* (Figura 2.26) . A outra forma de interoperabilidade é através da abordagem *Bridge In-Line* na qual a "ponte" é implementada dentro do núcleo do ORB (Figura 2.27).

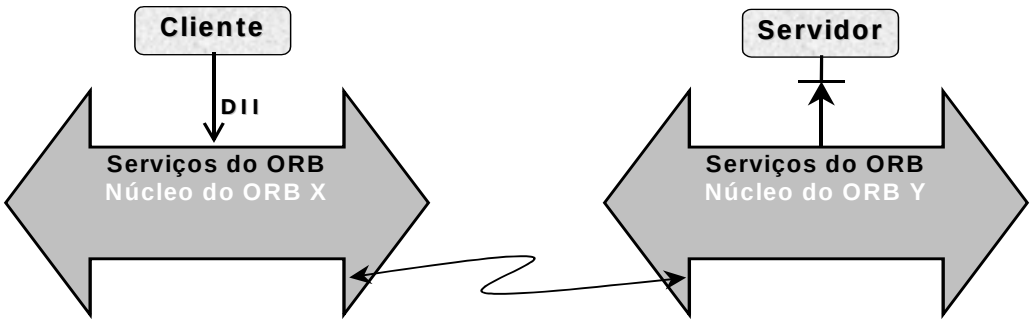

Figura 2.26. Formas para interoperabilidade entre ORBs: Bridge request-level.

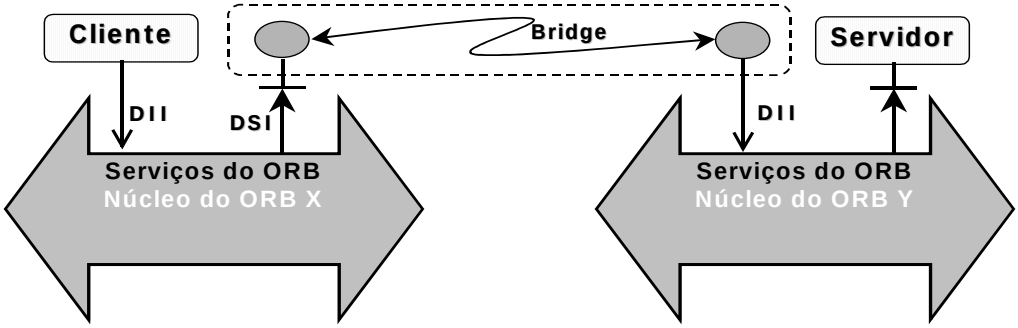

Figura 2.27. Formas para interoperabilidade entre ORBs: Bridge In-Line.

2.4.9 Programando com CORBA IDL

Como citado anteriormente, a IDL (*Interface Definition Language*) é uma linguagem declarativa, com sintaxes e tipos predefinidos, baseada na linguagem C++. Sendo uma linguagem de descrição de interface a sua função é gerar todo o suporte RMI que permitirá a chamada de métodos remotos. Os principais elementos que compõem uma IDL são: módulos (*modules*), interfaces, tipos de dados, constantes, atributos, operações e exceções.

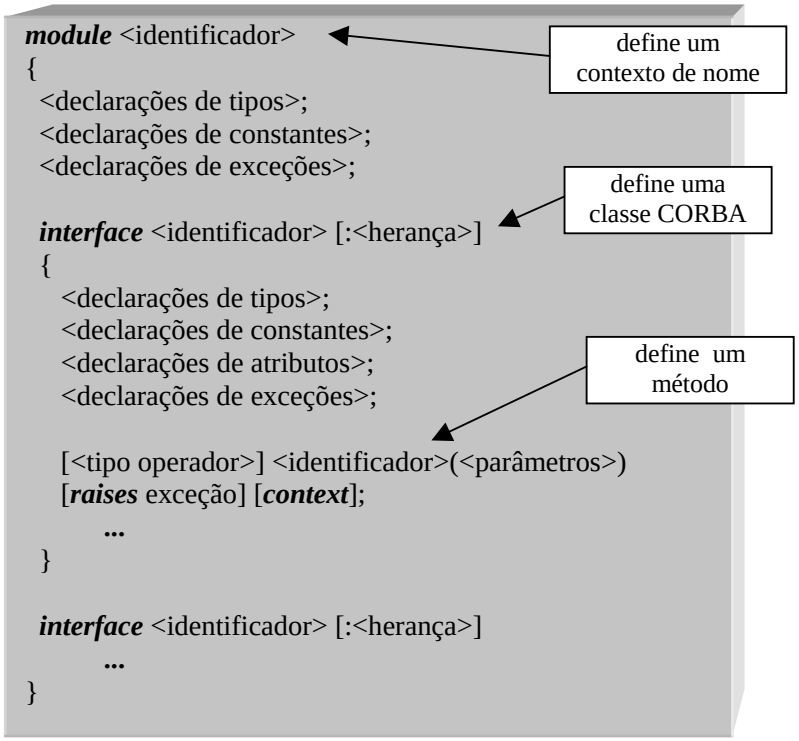

Figura 2.28. Declarações IDL.

Os tipos de dados da IDL devem ser mapeados nos tipos da linguagem alvo, ou seja, mapear nos tipos da linguagem usada na programação da aplicação. A IDL suporta tipos básicos (*short*, *long*, *float*, *boolean* ,...), tipos derivados (*array*, *sequences* e *string*; construídos com o uso da palavra chave *typedef*), tipos estruturados (*enum*, *struct*, *union*) e tipos variáveis (*arrays* dinâmicos, *string*) e o tipo *Any* que aporta uma certa flexibilidade.

A declaração `module` descreve os objetos (interfaces) que compõem a aplicação. Cada objeto terá suas interfaces descritas, com seus atributos e métodos sendo declarados segundo a sintaxe apresentada na Figura 2.28.

A Figura 2.29 ilustra a obtenção dos binários do cliente e servidor a partir de uma declaração de interface IDL e dos códigos dos objetos cliente e servidor. A tradução da especificação de interface pelo compilador IDL gera os *stubs* na linguagem alvo que ligados com os códigos de aplicação ao serem compilados devem gerar os binários correspondentes ao cliente e servidor. O compilador IDL gera também na linguagem alvo as interfaces dos objetos, cabendo ao programador preencher o corpo dos métodos.

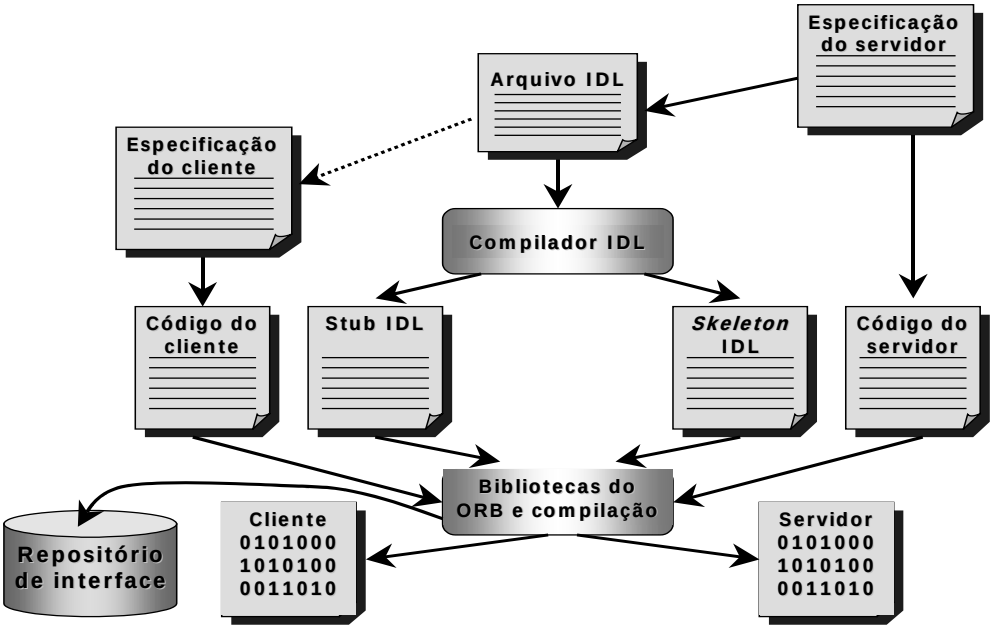

Figura 2.29. Geração dos binários do cliente e servidor

Basicamente, a partir de uma especificação do servidor, o programador escreve as classes de objetos do servidor em IDL [Orfali 1998]. Esses arquivos IDL formam uma espécie de "contrato" entre os clientes do serviço e o provedor do serviço, pois descrevem todos os serviços (métodos e atributos) que são exportados e visíveis para os potenciais clientes, e descrevem a forma que esses serviços podem ser acessados. Esses arquivos IDL precisam ser compilados pelos clientes e servidor antes de serem usados. O compilador IDL irá mapear as descrições em IDL para a linguagem usada pelo cliente ou servidor. Por exemplo, o programador do servidor pode usar um compilador IDL-Java, e o programador de um cliente (dentre os vários que podem existir no sistema) pode usar um compilador

IDL-C++. Os resultados dessas compilações são os *stubs* (que no caso do servidor recebem o nome de *skeletons*) Os *stubs* ocultam dos programadores a complexidade das tarefas necessárias na serialização/deserialização dos parâmetros em/de mensagens da rede. Finalmente, os programadores codificam o resto do código de seus sistemas; ou seja, o programador do servidor irá codificar as classes que ele descreveu no arquivo IDL, e o programador do cliente irá codificar a parte que faz chamadas ao servidor.

2.4.10. Integrando CORBA, HTTP e *Java*/RMI

A grande disseminação e a portabilidade da linguagem Java adicionada de seus mecanismos que facilitam a programação de aplicações na Internet levou a OMG a lançar especificações CORBA tentando unir as duas tecnologias. A primeira iniciativa foi definir o mapeamento IDL para a linguagem Java que permite então a geração de *stubs* e *skeletons* em Java, e o mapeamento inverso do Java para IDL que permite programadores Java gerarem *stubs* para atuarem em ambientes CORBA a partir de código Java.

A Sun, por sua vez, adotou o protocolo IIOP no Java/RMI. A Netscape em seus navegadores já oferece suporte para IIOP. Considerando que o HTTP abre uma nova conexão para cada requisição no mesmo servidor Web e o IIOP utiliza uma mesma conexão para várias requisições em um servidor CORBA, a união destas três ferramentas cria novas perspectivas na programação de aplicações na Internet. A Figura 2.30 ilustra as possibilidades da combinação destas ferramentas. Clientes em servidores *Web* podem ser ativados a partir de um navegador via http. Uma vez ativados podem interagir com servidores CORBA, Java ou mesmo DCOM e DCE. Estas combinações trazem bem mais alternativas que os usuais comandos CGI em um servidor *Web*. Um cliente na forma de um *applet* pode com suporte CORBA a partir de qualquer navegador na Internet pode enviar requisições a servidores CORBA, por exemplo.

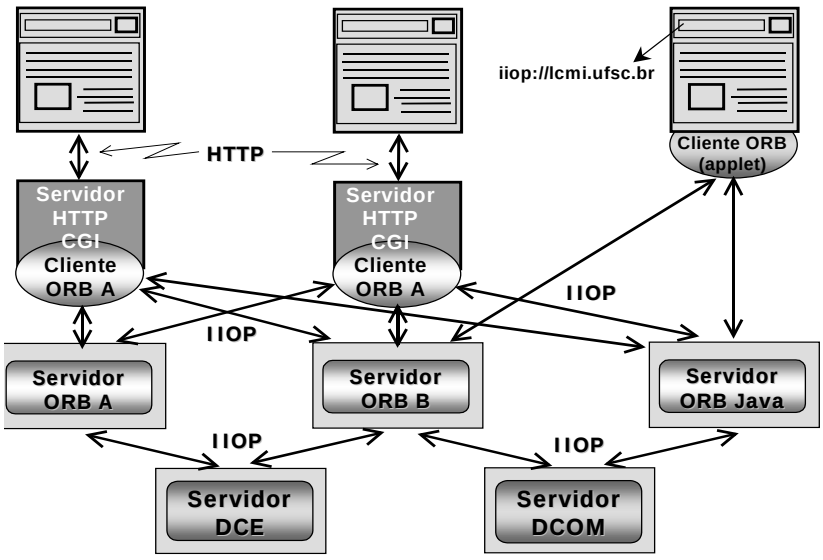

Figura 2.30. Integração CORBA/*Java*/Web

2.4.11. Suporte para Aplicações Críticas em CORBA

A OMG, dentro de sua dinâmica de manter as especificações CORBA evoluindo, vem estimulando a criação de muitos grupos de discussão ligados a interesses específicos de aplicações. Com isto nos últimos anos têm surgido novos *COSS* (*Common Object Services Specifications*) atendendo características de determinadas áreas de aplicações. É neste quadro que se encaixam as especificações *Real-Time CORBA* (RT-CORBA) [OMG 1999b], *Fault-Tolerant CORBA* (FT-CORBA) [OMG 2000a] e *CORBA Security* (CORBASec) [OMG 2000b] que são COSS voltados para aplicações críticas.

Na seqüência, são apresentadas descrições resumidas das especificações OMG sobre o *Real-Time* CORBA, o *Fault-Tolerant* CORBA e o CORBA *Security*.

4.11.1. As Especificações Real-Time CORBA

A falta de suporte de *middleware* para aplicações de tempo real levou o consórcio OMG a criar em 1995 um grupo de trabalho com objetivo de estender os padrões CORBA. Um documento RFI (*Request for Information*) [OMG 1996] foi lançado especificando requisitos que deveriam ser mantidos por um ORB que suporta aplicações com restrições temporais. O documento definiu um conjunto de requisitos para o ambiente operacional, para o ORB e serviços. Os requisitos de ambiente operacional podem ser resumidos em: *multithreading*, filas ordenadas por prioridade, sincronização de relógios e latência máxima de comunicação delimitada. Já os requisitos de ORB e serviços envolvem certas facilidades para expressar e implementar restrições temporais, a noção de prioridade global, invocações com polimorfismo temporal, etc.

Em setembro de 1997, um documento RFP (*Request for Proposal*) [OMG 1997b] foi publicado solicitando propostas de padronização de um CORBA para aplicações de tempo real (RT CORBA). Esse primeiro documento RFP, o Real-Time CORBA 1.0 RFP, definiu alguns requisitos para escalonamento baseado em prioridades.

Em outubro de 1998, cinco propostas apresentadas previamente foram reunidas em uma única proposta – o RT CORBA 1.0 [OMG 1999b]. Esse documento enfatiza a definição de mecanismos de ORB, permitindo projetistas selecionarem as políticas para escalonamento de tempo real. Não existem suposições prévias sobre o ambiente operacional subjacente, mas o documento reconhece que um sistema operacional em conformidade com POSIX é adequado para a obtenção de previsibilidade. Algumas das características e abstrações introduzidas nesta especificação são descritas na seqüência.

Threads, Pool de Threads e *Filas de Requisições*: O RT CORBA 1.0 descreve *threads* como as entidades escalonáveis do sistema. Algumas interfaces são especificadas para gerir *threads Pool de threads* e *filas de requisição* o que permite uma grande portabilidade às aplicações. *Threads* podem ser definidas e controladas de forma uniforme, independentemente se a plataforma de execução possui *threads* Solaris, NT, ou POSIX.1c. São introduzidos também *Pool de threads* e *filas de requisição* como recursos que aplicações podem manipular diretamente. Um *pool de threads* pode ser criado para

processar requisições de métodos recebidas por um servidor. Uma fila de requisição pode ser criada e associada com o *pool de threads* (Figura 2.31).

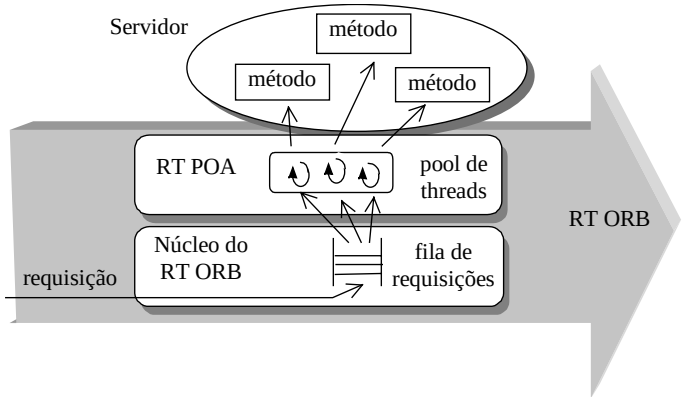

Figura 2.31. Exemplo de Uso dos Mecanismos do RT CORBA

Prioridade CORBA: A prioridade CORBA é um tipo de prioridade global que pode atravessar diferentes ORBs, fazendo uso de mensagens de invocação. Dependendo da abordagem de escalonamento, essa prioridade pode ser usada pelos servidores para ordenar as requisições recebidas. Portanto, o documento RFP introduz dois modos de utilização: os modelos *client-priority propagation* e o *server-set priority*. Em ambas as estratégias, a prioridade CORBA da invocação deve ser mapeada para as prioridades do sistema operacional antes da execução do método invocado.

Binding: A ligação entre cliente e servidor RT-CORBA é feita através de um *binding* explícito que contém vários passos a serem tomados para a interconexão dos objetos, tais como: localizar objetos destino e iniciar estruturas de dados que suportem comunicação entre objetos. No lado do cliente, o *binding* pode ser usado para garantir que seus requisitos temporais sejam respeitados (por exemplo, através da seleção de um protocolo de transporte apropriado e de um valor de *timeout* para detecção de faltas). No lado do servidor, o *binding* permite a alocação dos recursos necessários na execução das requisições, tais como threads e filas.

Interceptadores: O principal mecanismo proposto nas especificações CORBA, que introduz flexibilidade na programação da aplicação é o *interceptador*. Interceptadores são mecanismos geralmente usados para implementar de maneira transparente as políticas que regulam o comportamento das aplicações. A especificação RT CORBA 1.0 não trata especificamente de interceptadores, porque estes já foram padronizados nas especificações CORBA [OMG 1998].

Os mecanismos e recursos definidos nas especificações RT-CORBA devem permitir a implementação de diferentes abordagens de escalonamento de tempo real sobre as requisições principalmente as baseadas em prioridades fixas.

4.11.2. As Especificações Fault-Tolerant CORBA

Atualmente, muitas aplicações com requisitos de tolerância a faltas estão seguindo o paradigma de orientação a objetos e considerando o CORBA como uma alternativa de se adequar a sistemas abertos. Como resultado deste interesse, a OMG apresentou recentemente a especificação *Fault-Tolerant CORBA* [OMG 2000a]. As especificações FT CORBA, que ainda devem passar por varias revisões (e extensões), definem um conjunto de interfaces de serviços e facilidades úteis para a implementação de técnicas de replicação em ambientes distribuídos heterogêneos.

As especificações atendem apenas aos requisitos básicos para tolerância a faltas, definindo algumas interfaces bastante genéricas, de fácil entendimento, e aplicáveis em praticamente todas aplicações que requerem tolerância a faltas. Um padrão que tenta atender a todos os requisitos de um largo espectro de aplicações pode fazê-lo de forma insatisfatória ou, pode ser muito complexo para implementar. Portanto, o comitê da OMG responsável pela padronização do FT CORBA optou, para esse primeiro momento, por assumir apenas alguns compromissos.

Nesta primeira especificação foram apresentados um conjunto de interfaces de serviços comuns (COSS) e facilidades e ainda protocolos para a interoperabilidade que compõem algumas funcionalidades como: *gerenciamento de replicações, gerenciamento de faltas, gerenciamento de recuperação e logging*. Na Figura 2.32, é apresentada a arquitetura do FT-CORBA. Nela são apresentados os serviços comuns de objetos que, no *middleware*, fornecem funcionalidades básicas para a construção de servidores tolerantes a faltas.

Gerenciamento de Replicações

O gerenciamento de replicação é constituído pelos serviços de gerenciamento de propriedades, de grupos de objetos e fábrica genérica. Segundo as especificações, o gerente de replicações é responsável pelo gerenciamento de grupo, exercendo um controle dinâmico nas entradas e saídas (normal ou por falha) de objetos de um grupo pela manutenção de listas atualizadas de seus membros (*membership*).

Na criação ou remoção de réplicas, o serviço de objeto gerente de replicações usa o objeto fábrica genérica para criar e remover membros do grupo de objetos da aplicação. O objeto fábrica genérica negocia com os objetos fábrica locais para a criação/remoção de réplicas nas diferentes estações de um sistema distribuído. No processo de criação de réplicas são utilizados mecanismos de *logging* e *checkpoint* para o registro e a atualização de estados, que a partir de uma réplica *primária* (objeto servidor 1 na figura), fazem transferências de estados para novas réplicas que se juntam na replicação.

O serviço de gerenciamento de propriedades define e manipula as propriedades de tolerância a faltas dos grupos de objetos mantidos pelo serviço de gerenciamento de replicações.

Gerenciamento de Faltas

No *gerenciamento de faltas*, o serviço de detecção de falhas é dividido em três níveis, detectores de falhas em nível de objeto, processo e *host*. Os detectores de falhas nos diferentes níveis são baseados em mecanismos de *timeout* (as hipóteses de falha assumidas são de *crashes* de processadores). O detector de falha em nível de *host* se utiliza de replicação. O serviço notificador de falhas é também replicado e tem a função de enviar mensagens de notificação ao gerente de replicações, a partir dos registros de falhas enviados pelos detectores de falhas dos três níveis. Esses registros são necessários ao gerente de replicações para atualização das listas de membros de um grupo de objetos.

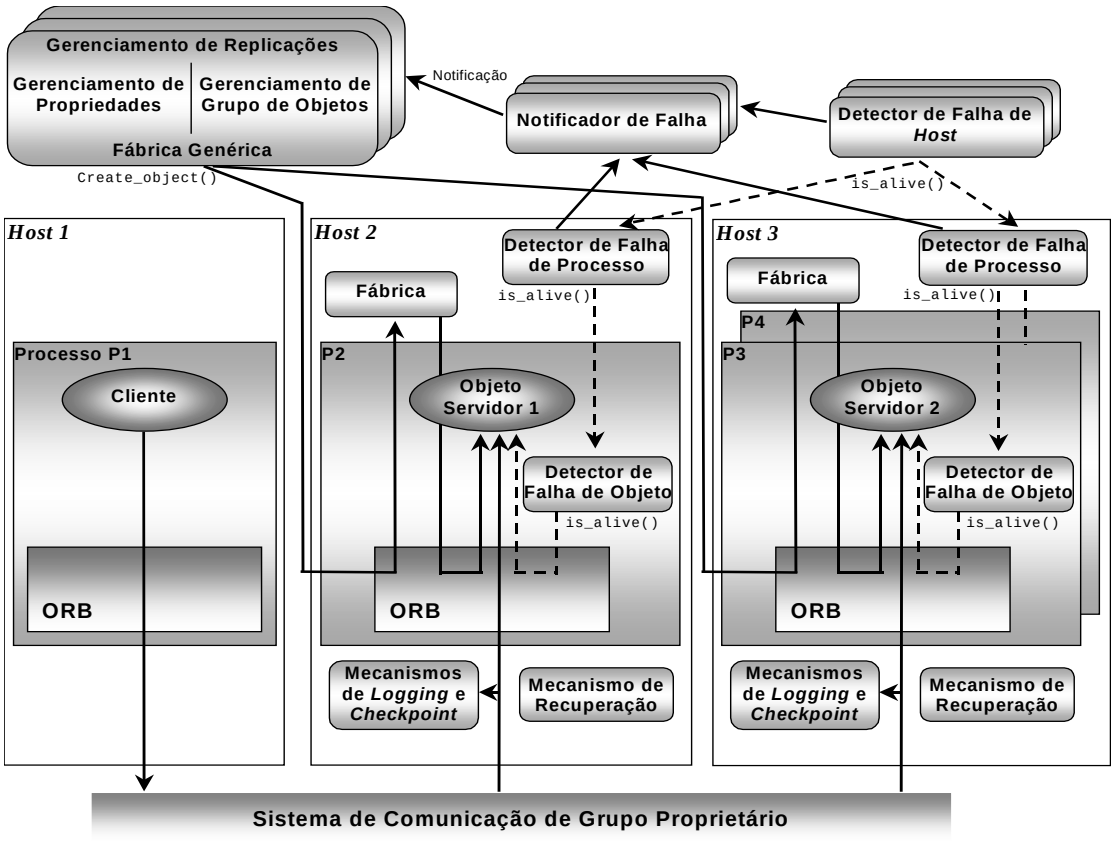

Figura 2.32. Arquitetura de Tolerância a Faltas do CORBA

As entidades e mecanismos descritos nas especificações foram introduzidos para suportarem as técnicas de replicação que podem ser suportadas pela arquitetura FT-CORBA, (*Replicação ativa*, *Replicação passiva fria* e *Replicação passiva quente* [OMG 2000a]).

A OMG, como é de sua característica, se limita em definir interfaces genéricas de serviços, tentando atender diferentes abordagens de tolerância a faltas. Por exemplo, protocolos de comunicação de grupo confiável, base fundamental para a implementação de técnicas de replicação, não são padronizados – como era de se esperar, pela

complexidade dos mesmos e pela quantidade de algoritmos envolvidos. A OMG enfatiza, neste caso, o uso de soluções proprietárias (Figura 2.32). Todavia, para garantir um mínimo grau de interoperabilidade, os sistemas de comunicação de grupo devem adotar a IOGR (*Interoperable Object Group Reference*), a referência de grupo de objetos definida em [OMG 1999a]. A IOGR é uma extensão da IOR de um simples objeto, formando uma referência interoperável de grupo de objetos. Uma IOGR permite a um cliente referenciar a um grupo de objetos como a uma entidade única. De forma genérica, a IOGR contém o conjunto de IORs de cada membros de um grupo de objeto.

4.11.3. As Especificações CORBA Security

A complexidade inerente aos sistemas de objetos distribuídos causa vulnerabilidades adicionais que devem ser contidas por uma arquitetura de segurança. Dentre estas vulnerabilidades, o controle de acesso em sistemas de larga escala é problemático, já que sistemas de objetos distribuídos podem crescer sem limites, com componentes sendo constantemente adicionados, removidos e modificados nestes ambientes. No sentido de minimizar estes problemas a OMG introduziu o *CORBAsec* que é um modelo de segurança que, quando implementado e administrado corretamente, pode prover um alto nível de segurança para informações e aplicações de objetos distribuídos em ambientes de larga escala [OMG 2000b].

O modelo de segurança descrito neste documento estabelece vários procedimentos envolvendo a autenticação, a verificação da autorização na invocação de um método remoto, a segurança da comunicação entre os objetos, além de aspectos relacionados com esquemas de delegação de direitos, não-repudiação, auditoria e administração da segurança.

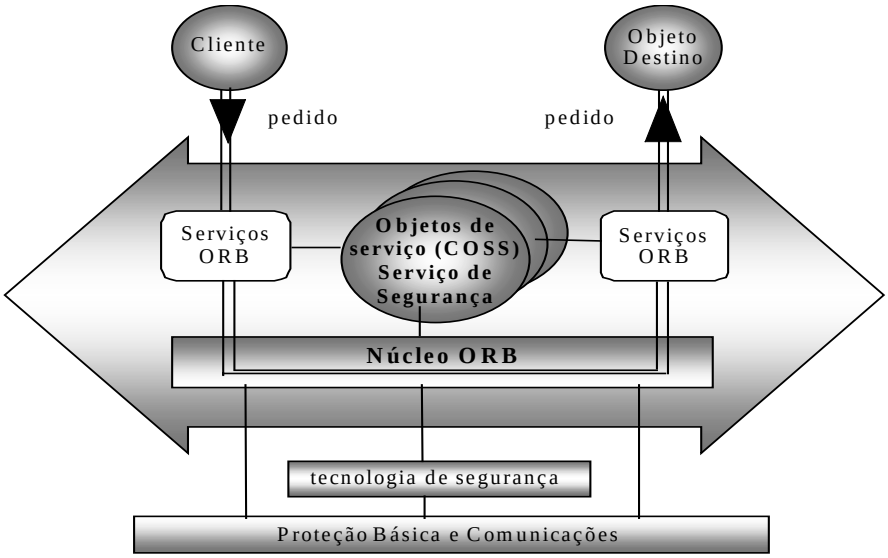

Figura 2.33. O Modelo CORBA de Segurança

O modelo CORBA de segurança relaciona objetos e componentes de quatro níveis (Figura 2.33). Em nível de aplicação, objetos cliente e servidor; serviços COSS, serviços ORB e o núcleo do ORB, todos em nível de *middleware*; componentes de tecnologia de segurança em nível de serviços de segurança subjacentes e componentes de proteção básica, fornecida por uma combinação de hardware e sistemas operacionais locais.

4.11.4. Interceptadores

No modelo CORBA de segurança, são definidos dois tipos de interceptadores que atuam durante uma invocação de método: o interceptador de controle de acesso (*Access Control Interceptor*), que em nível mais alto provoca um desvio para realizar o controle de acesso na chamada, e o interceptador de chamada segura (*Secure Invocation Interceptor*), que faz uma interceptação de mais baixo nível no sentido de fornecer propriedades de integridade e de confidencialidade nas transferências de mensagens correspondentes à invocação. Esses interceptadores atuam tanto no lado do cliente como do objeto servidor de aplicação.

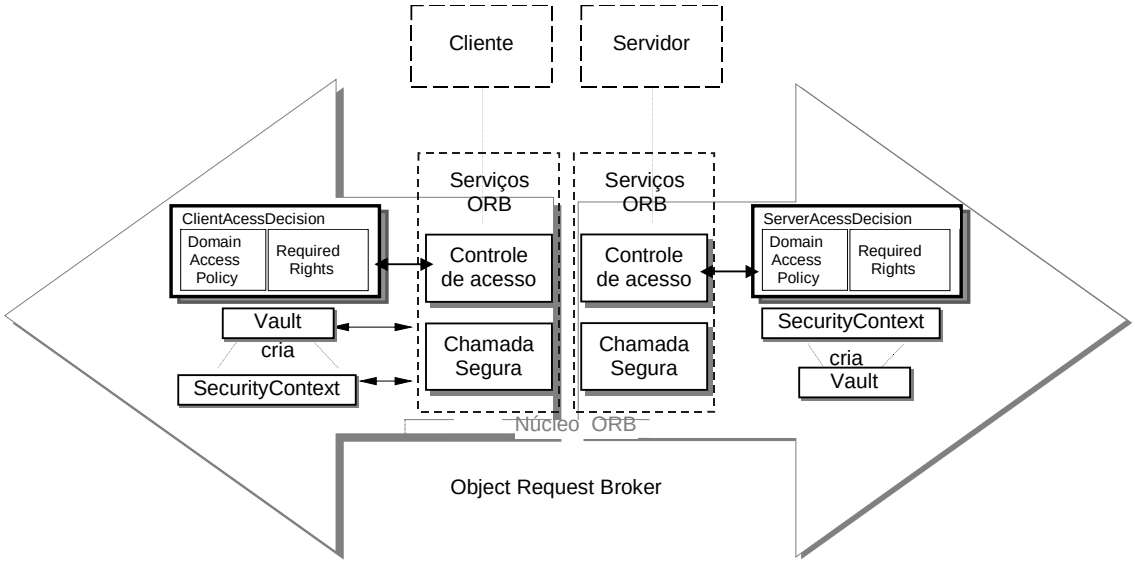

Figura 2.34. Componentes Internos do Modelo de Segurança do CORBA

A Figura 2.34 mostra todos os objetos e mecanismos definidos pelo CORBA*Sec* que atuam a nível de *middleware*. Observa-se que no interior das setas estão contidos o interceptador de controle de acesso que está representado pelo objeto *Controle de Acesso* e o interceptador de chamada segura que está representado pelo objeto *Chamada Segura*.

Serviços de Objetos Comuns

Os serviços de objetos comuns que implementam os controles de segurança nas especificações CORBA são:

- o *Principal Authenticator* que corresponde ao serviço de autenticação de principais no CORBA;
- o *Vault* que estabelece as associações seguras entre os clientes e servidores com os respectivos contextos de segurança;
- o *SecurityContext* que representa o contexto da associação segura, o *Credential* que representa as credenciais ou direitos do cliente na sessão;
- o *DomainAccessPolicy* que representa a interface de gerenciamento de políticas de autorização discricionárias e concede a um conjunto de principais direitos específicos para executar operações sobre todos os objetos do domínio [Karjoth98];
- o *RequiredRights* que armazena as informações sobre os direitos (*rights*) necessários para executar cada método de cada interface do sistema;
- os objetos *AccessDecision* responsáveis por interagir com os objetos *DomainAccessPolicy* e *RequiredRights* para determinar se uma dada operação sobre um objeto destino específico é permitida.

Existem, também, outros objetos de serviço relacionados com a não-repudiação e auditoria. Os serviços de objetos comuns no modelo CORBA de segurança isolam aplicações e o ORB da tecnologia de segurança (Figura 2.34). Esta última consiste em uma camada subjacente que implementa várias funcionalidades dos serviços de objetos comuns relacionados com a segurança. A tecnologia de segurança inclui serviços de autenticação, serviços de associação segura (distribuição de chaves, certificados, cifragens/decifragens), etc. De acordo com a especificação do *CORBASec*, as tecnologias que podem ser utilizadas para fornecer esses serviços são [OMG 2000b]: SPKM (*Simple Public-Key GSS-API Mechanism*), Kerberos, CSI-ECMA (baseado no SESAME e no mecanismo ECMA GSS-API) e o SSL.

2.4.12. Considerações sobre as especificações CORBA

A OMG, dentro de sua linha de fornecer padrões abrangentes que atendam as necessidades de programação de aplicações distribuídas em sistemas cada vez mais caracterizados pela distribuição geográfica e heterogeneidade, continua a estimular a produção de novas especificações. Além das citadas neste texto, existem várias outras propostas em fase de padronização ou já padronizadas na OMG. Neste processo, estão as propostas de: UML (*Unified Modeling Language*); protocolo Meta-objeto; suporte para mídia contínua; comercio eletrônico; agentes móveis e centenas de outras especificações.

O CORBA corresponde a um conjunto de especificações bastante completo atendendo a um espectro amplo de aplicações. Por ser completo é considerado muito complexo por seus críticos. Mas no sentido de justificar a sua aceitação diríamos que a força destas especificações está na sua constante evolução que procura mantê-la adaptada sempre às novas tecnologias emergentes. A nossa opinião é de que a aceitação destes padrões deve continuar ainda por muitos anos.

2.5. Conclusão

Este texto descreveu plataformas para a execução de aplicações distribuídas. Inicialmente a seção 2.2 apresentou os conceitos fundamentais da área. Em seguida, a seção 2.3 mostrou como aplicações distribuídas podem ser escritas através da tecnologia Java. Em particular, foram apresentadas quatro maneiras de usar Java com este propósito: *sockets*, *servlets*, RMI e Jini. A seção 2.4 tratou do padrão mais importante atualmente para o desenvolvimento de aplicações distribuídas em arquiteturas heterogêneas, o padrão CORBA.

Esta é uma área com enorme quantidade de material publicado. O leitor interessado no assunto poderá ampliar os seus conhecimentos através da literatura citada ao longo do texto. Os autores esperam que, após a leitura deste texto, fique mais fácil para alguém interessado na área identificar os tópicos relevantes e localizar o material apropriado a ser estudado.

No momento que este texto é escrito, surgem diversos novos desafios para a comunidade que trabalha na área, ao mesmo tempo em que velhos desafios continuam indomados. A necessidade premente é atacar os problemas de segurança existentes hoje. Todos nós conhecemos bem os problemas advindos dos vírus de computador e dos acessos não autorizados. A disseminação da computação móvel também traz novos problemas para a confiabilidade da operação de aplicações distribuídas. Finalmente, sistemas distribuídos com requisitos de tempo real exigem novas considerações no momento do projeto. A dimensão tempo deixa de ter um papel secundário, como uma questão de desempenho, e passa a ter um papel central, relacionado com a própria funcionalidade da aplicação. Com certeza nos próximos anos muitas novidades continuarão a surgir nesta área.

2.6. Referências

[ANSA] ANSAware, URL: http://www.ansa.co.uk

[Arnold 1997] K. Arnold, J. Gosling, *Programando em Java*, Makron Books, 1997.

[Birman 1996] K. Birman, *Building Secure and Reliable Network Applications*, Manning Publications, 1996.

[Coulouris 2001] G. Coulouris, J. Dollimore, T. Kindberg, *Distributed Systems: Concepts and Design*, Third Edition, Addison-Wesley, 2001.

[Deitel 2001] H. Deitel, P. Deitel, *Java Como Programar*, 3a edição, Bookman, 2001.

[Edwards 2000] W. Keith Edwards, *Core Jini*, 2nd edition, Prentice Hall, 2000.

[Harold 1997] E. Harold, *Java Network Programming*, O'Reilly, 1997.

[ISO 1995] ISO, *Reference Model of Open Distributed Processing - Part 1: Overview*, Document 10746-1, May 1995.

[Jul 1991] Eric Jul, et. al., *Emerald: A General-Purpose Programming Language*, Software Practice and Experience, vol. 21, number 1, pages 91-118, Jan. 1991.

[Kramer 1989] J. Magee, J. Kramer, M. Sloman, *Constructing Distributed Systems in Conic*, IEEE Transactions on Software Engineering, vol 15, no 6, 1989.

[Kurose 2001] J. Kurose, K. Ross, *Computer Network: A Top-Down Approach Featuring the Internet*, Addison Wesley, 2001.

[Liskov 1988] B. Liskov, *Distributed Programming in Argus*, Communications of the ACM, Vol. 31, Num. 3, Mar. 1988, pages 300-312.

[Lynch 1996] N. Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers, 1996.

[Mills 1991] D. L. Mills, *Internet Time Synchronization: the Network Time Protocol*, IEEE Trans. on Communications 39, 10, pp.1482-1493, Oct. 1991.

[OMG 1996] OMG Realtime Platform Special Interest Group, Realtime Technologies - Request for Information, Object Management Group (OMG), Document realtime/96-08-02 96, Aug. 1996.

[OMG 1997a] Object Management Group, *CORBAservices: Common Object Services Specification*, OMG Document. Mar. 1997.

[OMG 1997b] OMG Realtime Platform Special Interest Group, *Realtime CORBA 1.0 RFP*, Object Management Group (OMG), Document realtime/97-09-31, Sep. 1997.

[OMG 1998] OMG, *The Common Object Request Broker: Architecture e Specification - Revision 2.2*, Object Management Group (OMG), Feb. 1998.

[OMG 1999a] Object Management Group, ***ORB Interface***, OMG Document Number 99-07-08, June 1999.

[OMG 1999b] OMG, *Realtime CORBA - Joint Revised Submission*, Object Management Group (OMG), Document orbos/99-02-12, Mar. 1999.

[OMG 2000a] Object Management Group, *Fault-Tolerant CORBA Specification V1.0*, OMG document: ptc/2000-04-04, Apr. 2000.

[OMG 2000b] Object Management Group, *Security Service:v1.5*, OMG Document Number 00-06-25, Jun. 2000.

[Orfali 1998] R. Orfali, D. Harkey, *Client/Server Programming with JAVA anda CORBA*, Second Edition, John Wiley & Sons, Inc., 1998.

[OSF] DCE, URL: http://www.opengroup.org/dce/

[RFC 1035] P. Mockapetris, *Domain Names – Implementation and Specification*, Nov. 1987.

[RFC 1180] T. Socolofsky, C. Kale, *A TCP/IP Tutorial*, Jan 1991.

[Stevens 1998] W. Stevens, *UNIX Network Programming Networking APIs: Socket and XTI*, Volume 1, Second Edition, Prentice-Hall Inc., 1998.

[Sun 1999] Sun, *Jini Architectural Overview: Technical White Paper*, Sun Microsystems, Inc., 1999.

[Sun 2001] Sun, *Jini Network Technology: An Executive Overview*, Sun Microsystems, Inc., 2001.

[Tanenbaum 2002] M. Van Steen, A. Tanenbaum, *Distributed Systems: Principles and Paradigms*, Prentice Hall, 2002.