RELATÓRIO TÉCNICO

"UMA ABORDAGEM PARA O DESENVOLVIMENTO DE APLICAÇÕES DISTRIBUÍDAS EM CORBA USANDO C++ E ORBIX"

José Helvécio Teixeira Jr [1] helveciot@ravel.ufrj.br

Luís Felipe M. de Moraes moraes@ravel.ufrj.br

Suzana Ramos Teixeira [2] suzana@ravel.ufrj.br

Laboratório de Redes de Alta Velocidade - RAVEL
Programa de Engenharia de Sistemas e Computação
Universidade Federal do Rio de Janeiro
COPPE/UFRJ
Cx. Postal 68511 - CEP 21945-970
Rio de Janeiro/RJ

(1) Aluno de Doutorado. Bolsista FAPERJ.
(2) Aluna de Doutorado. Bolsista CAPES.

RESUMO

O CORBA fornece as abstrações e os serviços necessários ao desenvolvimento de aplicações distribuídas e portáveis, sem a necessidade de que o programador se preocupe com detalhes de baixo nível. Ele fornece suporte para vários modelos do tipo pedido-resposta facilitando a localização e a ativação transparente de objetos, e para a independência de linguagens de programação e de sistemas operacionais, o que propicia uma base sólida tanto para a integração de sistemas legados quanto para o desenvolvimento de novas aplicações distribuídas.

1. Introdução

As redes de computadores são tipicamente heterogêneas devido aos seguintes motivos:

- a rapidez em que ocorrem as inovações tecnológicas;
- as diferentes necessidades ("*one size does not fit all*");
- à necessidade de uma maior confiabilidade da rede;
- porque é inevitável.

Assim, projetistas de sistemas distribuídos reais, gostem eles ou não, precisam lidar com a heterogeneidade. Desenvolver software para um sistema distribuído envolve níveis elevados de esforços, e desenvolver software para sistemas distribuídos heterogêneos às vezes chega a ser quase impossível. Um software desse tipo deve lidar com quase todos os problemas normalmente encontrados na programação de sistemas distribuídos, como: falha de algum dos sistemas na rede; o particionamento da rede; problemas ligados ao controle de uso e ao compartilhamento de recursos na rede; e também com riscos de nível de segurança. Se adicionarmos heterogeneidade a um cenário como esse, muito desses problemas passam a ser críticos e começam a aparecer novos tipos de problemas.

Por exemplo, problemas encontrados quando se migra uma aplicação em rede para uso em outra plataforma na rede irá resultar em duas ou mais versões da mesma aplicação. Se forem feitas alterações em qualquer uma das duas versões, as outras versões devem ser alteradas apropriadamente, e depois testadas individualmente para que se possa assegurar que elas estão funcionando adequadamente.

O nível de dificuldade que surge com uma situação como esta aumenta significativamente quando aumenta o número de diferentes plataformas na composição da rede.

Observe que quando falamos de heterogeneidade, neste contexto, não estamos nos referindo somente ao hardware de computação e aos sistemas operacionais. Para escrever uma aplicação distribuída robusta de cima para baixo (por exemplo, desde uma interface gráfica de usuário sob medida até os protocolos de rede) é extremamente complicado para qualquer aplicação do mundo real devido à tremenda complexidade e ao mínimo de detalhes envolvidos. Desta forma, os desenvolvedores de aplicações distribuídas costumam fazer uso pesado de ferramentas e bibliotecas disponíveis. Isso implica dizer que os sistemas distribuídos são implicitamente heterogêneos, e geralmente compostos por várias aplicações em diferentes camadas e por várias bibliotecas. Infelizmente, e muitos casos, à medida que o sistema distribuído cresce, as chances de que todas as aplicações e bibliotecas que o compõem terem sido especificamente projetadas para trabalharem em conjunto diminuem drasticamente.

De maneira geral, duas regras devem ser observadas no projeto de aplicações para sistemas distribuídos heterogêneos:

- empregar modelos e abstrações independentes de plataforma; e
- esconder ao máximo as complexidades de baixo nível sem que se sacrifique demais o nível de desempenho.

Utilizar abstrações e modelos corretos pode, essencialmente, fornecer uma nova camada de desenvolvimento de aplicações que seja homogênea e situada no topo de toda a complexidade causada pela heterogeneidade. Tal camada irá esconder detalhes de baixo nível e permitirá que os desenvolvedores de aplicações solucionem seus problemas mais urgentes sem precisarem tratar de imediato de detalhes de baixo nível de rede para todas as diferentes plataformas usadas por suas aplicações.

A especificação CORBA fornece um conjunto equilibrado de abstrações flexíveis e de serviços concretos necessários para viabilizarem soluções práticas para os problemas associados com a computação heterogênea distribuída.

2. O Grupo de Gerenciamento de Objetos OMG (*Object Management Group*)

O OMG foi criado em 1985 para tentar resolver os problemas relacionados com o desenvolviemento de aplicações distribuídas para sistemas heterogêneos. O OMG é atualmente o maior consórcio de software do mundo, com mais de 800 membros. As primeiras especificações produzidas pelo OMG foram a OMA (*Object Management Architecture*) e o seu núcleo (a especificação CORBA), que fornecem uma infra-estrutura arquitetural completa que suficientemente rica e flexível para acomodar uma grande variedade de sistemas distribuídos.

A OMA emprega dois modelos inter-relacionados para descrever como os objetos distribuídos e a interação entre eles pode ser especificada de forma independente de plataforma. O Modelo de Objetos descreve como são descritas as interfaces de objetos distribuídos através de um ambiente heterogêneo, e o Modelo de Referência caracteriza as interações entre esses objetos.

O Modelo de Objetos define um objeto como uma entidade encapsulada com uma identidade distinta umutável cujos serviços são acessados somente através de interfaces bem definidas. Os clientes utilizam os serviços de um objeto emitindo pedidos para este objeto. Os detalhes da implementação do objeto e a sua localização são mantidos escondidos dos clientes.

O Modelo de Referência fornece categorias de interfaces que são agrupamentos gerais de interfaces para objetos. Como ilustrado na Figura 1, todas as categorias de interfaces são conceitualmente ligadas por meio de um ORB (Object Request Broker). Geralmente, um ORB possibilita a comunicação entre clientes e objetos, ativando de forma transparente aqueles objetos que não estiverem rodando quando os pedidos são encaminhados para eles. O ORB fornece também uma interface que pode ser usada diretamentente tanto pelos clientes quanto pelos objetos.

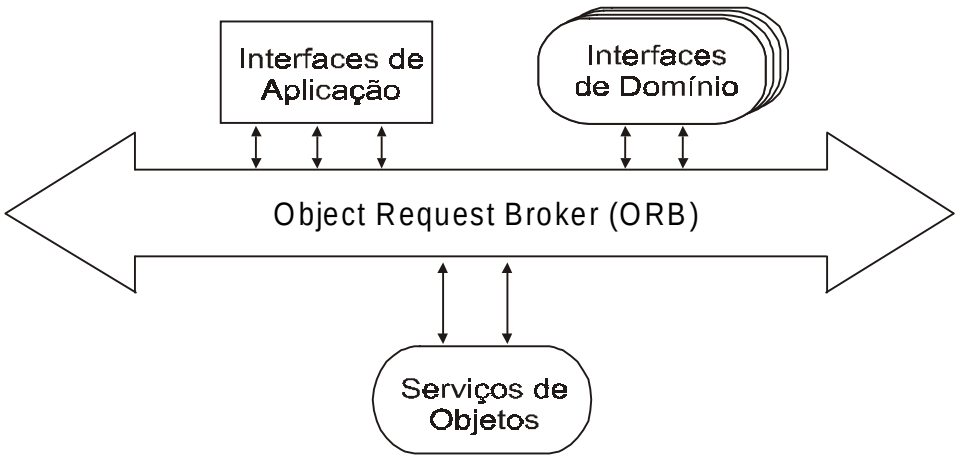

Figura 1. Categorias de Interfaces OMA

A Figura 1 mostra as categorias de interfaces que utilizam as facilidades de ativação e comunicação de um ORB: serviços de objetos, interfaces de domínios e interfaces de aplicações.

Serviços de Objetos são interfaces independentes de domínio, ou orientadas horizontalmente, usadas por muitas aplicações de objetos distribuídos. Por exemplo, todas as aplicações devem obter referências a respeito dos objetos que elas pretendem utilizar. Tanto o serviço de nomes quanto o serviço de negociação da OMG representam serviços de objetos que permitem que as aplicações possam procurar e localizar referências a objetos. Os serviços de objetos são normalmente considerados como parte da infra-estrutura central de computação distribuída.

As Interfaces de Domínios desempenham uma função similar àquelas da categoria de Serviços de Objetos, exceto pelo fato de que as Interfaces de Domínios são específicas para um determinado domínio, ou seja, são orientadas verticalmente. Por exemplo, existem Interfaces de Domínios empregadas em aplicações de planos de previdência privada que são únicas para esta indústria, como por exemplo o serviço de identificação de pessoas (*Person Identification Service*) [28]. Outras Interfaces de Domínios são específicas de outros domínios, como os domínios de finanças, manufatura, telecomunicações dentre outros. Na Figura 1, as várias ocorrências para a Interface de Domínio representam esta multiplicidade de domínios.

Interfaces de Aplicação são desenvolvidas especificamente para uma dada aplicação. Elas não são padronizadas pela OMG. No entanto, se certas Interfaces de Aplicação começarem a aparecer em diferentes aplicações, elas podem se tornar candidatas para padronização em alguma das outras categorias de interfaces.

À medida que a OMG for completando gradualmente as categorias de interfaces, a maior parte de seus esforços de padronização se deslocarão para cima, partindo da infra-estrutura ORB e dos níveis de Serviços de Objetos para infra-estruturas de objetos de domínios específicos.

O conceito de infra-estrutura de objetos, ilustrado na Figura 2, constroi-se a partir das categorias de interfaces descritas há pouco, reconhecendo e promovendo a noção de que os programas baseados em CORBA são compostos a partir de componentes multi-objetos que suportam uma ou mais das categorias de interfaces OMA. A Figura 2 representa tais componentes como círculos, alguns com somente uma categoria de interface e outros com múltiplas categorias. Infelizmente, o termo infra-estrutura é usado de maneira geral, porém, se usado neste contexto ele segue a definição clássica para infra-estrutura de software: uma solução parcial para um conjunto de problemas similares que requerem a adequação das aplicações para que ela seja uma solução efetiva. O OMG espera poder padronizar as especificações para infra-estruturas de objetos para o uso em indústrias representadas pelas suas *Domain Task Forces* (Força Tarefa de Domínios).

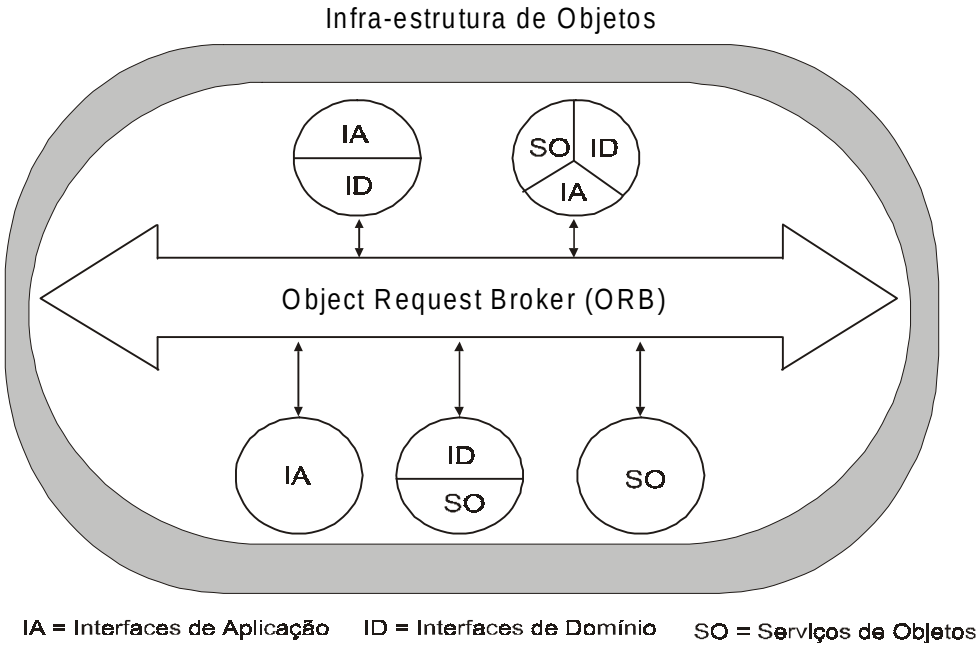

Figura 2.: Infra-estruturas de Objetos OMA

Estes modelos podem não parecer muito complicados ou profundos, porém a sua aparente simplicidade é enganadora.

2.1 Conceitos Básicos e Terminologia

CORBA fornece interfaces de programação e modelos independentes de plataformas para o desenvolvimento de aplicações de computação orientada a objetos distribuídos e portáveis. Sua independência com relação a linguagens de programação, plataformas de computação, e protocolos de rede o tornam extremamente adequado para o desenvolvimento de novas aplicações facilitando a sua integração com os sistemas distribuídos já existentes.

CORBA tem uma terminologia própria. Os termos mais importantes desta terminologia estão explicados abaixo:

- Um objeto CORBA representa uma entidade virtual passível de ser localizada por um ORB e de ter pedidos de clientes dirigidos para ela. Ela é virtual no sentido de que ela realmente não existe, a não ser que ela seja caracterizada através de uma implementação escrita em uma linguagem de programação. O desenvolvimento de um objeto CORBA por construções de uma linguagem de programação é análogo à maneira como a memória virtual não existe em um sistema operacional porém é simulada pelo uso de memória física.
- Um objeto alvo, no contexto da emissão de um pedido CORBA, representa o objeto CORBA para quem o pedido está direcionado. O modelo de objetos CORBA é um modelo de submissão único, no qual o objeto alvo para um pedido é determinado exclusivamente através da referência ao objeto utilizada para que se pudesse emitir o pedido.
- Um cliente representa uma entidade que emite um pedido para um objeto CORBA. Um cliente pode existir em um espaço de endereçamento que é completamente separado do objeto CORBA, ou então o cliente e o objeto CORBA podem existir na mesma aplicação. O termo cliente é significativo somente no contexto de um pedido em particular, pois a aplicação que é cliente para um pedido pode também ser o servidor para um outro pedido qualquer.
- Um servidor representa uma aplicação na qual existem um ou mais objetos CORBA. Da mesma forma como nos clientes, o termo servidor só tem sentido dentro do contexto de um determinado pedido.
- Um pedido representa a chamada de uma operação por um cliente em um objeto CORBA. Os pedidos fluem de um cliente para um objeto alvo em um servidor, e o objeto alvo envia os resultados de volta como resposta, caso o pedido exija uma.
- Uma referência a objeto representa uma forma adequada usada para identificar, localizar e encaminhar pedidos para um objeto CORBA. Para os clientes, as referências a objetos são entidades opacas (não transparentes). Os clientes utilizam as referências a objetos para direcionarem pedidos para objetos, porém eles não podem criar referências a objetos a partir das partes que os constituem, nem podem acessar ou modificar o conteúdo de uma referência a objeto. Uma referência a objeto diz respeito somente a um único objeto CORBA.
- Um servente (*servant*) representa uma entidade de linguagem de programação que implementa um ou mais objetos CORBA. Diz-se que os serventes encarnam objetos CORBA porque eles fornecem corpos, ou implementações, para estes objetos. Os serventes existem no contexto de uma aplicação servidora. Em C++, os serventes representam instâncias de objetos de uma determinada classe.

2.2. Arquitetura de um Sistema CORBA

As diversas empresas que exigem uma infra-estrutura de integração para os seus sistemas, a enorme diversidade de possibilidades para executar tal tarefa, e os perigos fornecidos por soluções proprietárias são fortes argumentos para a busca por uma solução não-proprietária. O padrão CORBA tenta atender a essas necessidades quebrando as fronteiras que surgem nas áreas de redes de computadores, linguagens de programação e sistemas operacionais. CORBA pode ser visto como um ambiente para suportar o desenvolvimento de novos sistemas ou como um ambiente para a integração de aplicações nos quais novos sistemas podem ser construídos através da composição de funcionalidades prontas em sistemas (ou sub-sistemas) já existentes.

Uma implementação do padrão é conhecida como um Corretor para Pedidos de Objetos (ORB - *Object Request Broker*) ou seja, um intermediário para permitir que clientes possam encaminhar pedidos aos objetos. Um ORB deve poder fazer pedidos através da rede, entre diferentes sistemas operacionais e entre diferentes linguagens de programação. O padrão é suficiente flexível para permitir diferentes tipos de implementações; por exemplo: aquelas implementações otimizadas para ambientes de tempo real ou aquelas que se integram em um ambiente OLE do Windows, ou implementações embutidas, como em *palmtops* ou telefones celulares, ou ainda, implementações em *mainframes*, ou finalmente aquelas que podem ser carregadas em um navegador da Web.

Apesar de todas essas possibilidades de implementações, dentre outras, cada implementação deve poder ser comunicar com todas as outras através de um protocolo conhecido com IIOP (*Internet Inter-ORB Protocol*). IIOP é definido para rodar sobre TCP/IP. O protocolo IIOP emprega um formato de mensagem chamado GIOP (*General Inter-ORB Protocol*), isto é, IIOP representa a mensagem GIOP enviada através do TCP/IP. O formato de mensagem GIOP pode ser também empilhado sobre outros protocolos de transporte. Protocolos inter-ORB específicos para alguns tipo de ambiente também são permitidos; por exemplo: protocolos especializados estritamente para ambientes de tempo real. Isto significa que qualquer cliente CORBA pode se comunicar com qualquer objeto CORBA (conquanto que este cliente tenha permissão para invocá-lo). Os pacotes com pedidos IIOP contêm a identidade do objeto alvo, o nome da operação a ser chamada, e os parâmetros. Esta informação é usada automaticamente no servidor para que se possa localizar o servidor no objeto alvo, e para que se possa ativar a função correta no mesmo. O protocolo IIOP foi primeiramente adotado em 1996. Existe atualmente uma versão segura do mesmo, a qual fornece mecanismos de autorização e criptografia.

Um sistema CORBA típico é constituído de um conjunto de programas clientes que fazem uso dos objetos distribuídos por toda a rede. Na maioria dos sistemas operacionais, todo o processamento deve ser efetuado dentro de algum processo. Assim, cada objeto deve residir em um processo. Em outros sistemas operacionais, os objetos podem ser executados dentro de *threads* ou dentro de bibliotecas de ligação dinâmica – (DLLs – *dynamic link libraries*). Para evitar dificuldades com a terminologia, CORBA estabelece que os objetos existam dentro de servidores (em CORBA, emprega-se freqüentemente o termo implementação ao invés de servidor). Cada objeto é associado com um servidor único, e se o servidor de um determinado objeto não estiver rodando quando é feita uma chamada para este objeto, o CORBA cuida de ativar automaticamente o servidor. Esta ativação pode signifiar a inicialização de um processo ou de uma thread, ou a carga de uma biblioteca, dependendo do tipo de sistema operacional no qual o servidor estiver registrado. Em UNIX, por exemplo, um servidor pode estar adormecido, e pode não existir nenhum processo associado ao mesmo; ou

então, ele pode ser ativado e, portanto, ter um processo rodando o seu código. Na realidade, alguns servidores podem ter mais do que um único processo associado a eles.

O código para um servidor inclui o código que implementa os objetos que ele contém (na realidade, seus tipos). Inclui também uma função principal (`main()`, em C++) que inicializa o servidor e, normalmente, cria um conjunto inicial de objetos. Como os objetos podem ser pequenos, um servidor pode conter vários objetos CORBA. Esses objetos podem ser todos do mesmo tipo, ou então um servidor pode suportar objetos de diferentes tipos sem qualquer dificuldade e sem a exigência de se empregar código especial. Um servidor pode também suportar objetos não-CORBA; por exemplo, objetos em C++ que só podem ser acessados de dentro do próprio servidor. A Figura 3 ilustra um cliente executando uma chamada para um objeto CORBA em um servidor remoto. Um outro servidor juntamente com seus objetos também é mostrado. Note que um servidor não pode ser invocado pelos clientes, somente seus objetos CORBA podem.

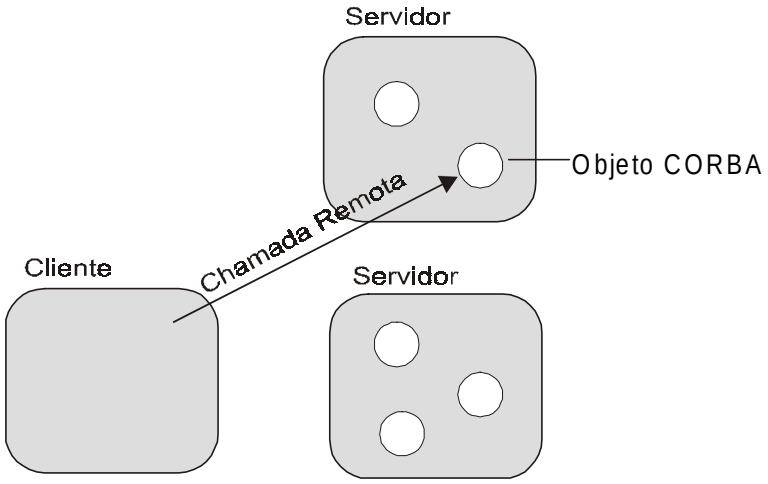

Figura 3. Clientes, Servidores e Objetos

Em CORBA, os objetos em um servidor podem utilizar os objetos em outros servidores. Esta funcionalidade é bastante útil quando se decompõe um sistema em vários componentes, pois esta estratégia permite que um cliente execute uma chamada para um objeto em um determinado servidor, e que este objeto possa executar chamadas para outros objetos de modo a poder atender ao pedido do cliente. Um servidor que executa uma chamada para um objeto remoto (em outro servidor) estará agindo como cliente enquanto durar esta chamada. Note que o termo "remoto" é normalmente empregado caso a chamada seja direcionada para outro servidor, mesmo que este servidor que foi chamado estiver na mesma máquina onde está o servidor chamador. O nível de transparência é preservado pelo emprego da mesma sintaxe empregada nas chamadas a (para) objetos no mesmo espaço de endereçamento, e para objetos em diferentes servidores, estejam eles localizados na máquina local ou em máquinas remotas.

Um conjunto de servidores pode também cooperar para fornecer um serviço geral para um cliente. Existem várias possibilidades de se implementar tal cooperação. Em um dos esquemas, cada um dos servidores do conjunto de servidores pode fornecer a mesma funcionalidade, de modo que um cliente pode usar qualquer um dos servidores. O cliente pode escolher aquele servidor localizado na mesma máquina onde ele se encontra ou, então, se a rede for suficientemente rápida, ele pode escolher um servidor remoto que não esteja muito carregado. Em outro esquema, cada um dos servidores em um conjunto de servidores

podem implementar uma função especializada, de modo que um cliente pode se comunicar com um objeto em um dos servidores, e este objeto pode requerer o uso das funcionalidades dos objetos nos outros servidores.

Em muitos casos, cada nova aplicação adicionada ao sistema distribuído irá requerer a adição de um novo cliente que possa fornecer uma interface de usuário adequada. Pode ocorrer também que alguns servidores precisem ser adicionados, ou pode ocorrer ainda que alguns dos servidores já existentes precisem ser melhorados.

Em algumas implementações do CORBA, o código do cliente pode também conter objetos CORBA. Esta funcionalidade é bastante útil, pois significa que os servidores podem executar chamadas a estes objetos, talvez com o objetivo de informar aos mesmos da ocorrência de algum evento ou, simplesmente, com o objetivo de mantê-los informados sobre o valor corrente de algum dado. A maneira corriqueira de um servidor localizar um objeto em um cliente faz-se por meio do cliente passando uma referência ao objeto (do cliente) para o servidor na forma de um parâmetro alguma chamada anteriormente feita do cliente para o servidor em questão. Referências a objetos podem ser facilmente encaminhadas desta maneira, mesmo entre códigos que tenham sido escritos em diferentes linguagens de programação.

Em grandes redes, a velocidade dos servidores pode afetar a velocidade do sistema como um todo. Enquanto a grande maioria dos usuários do sistema pode ter suas próprias estações e uma cópia do código cliente, os servidores precisam ser compartilhados, seja por todos os clientes ou por um grupo específico de clientes. Melhorando-se a eficiência de um servidor irá influenciar no número de clientes que este servidor poderá suportar e, portanto, pode-se reduzir os custos para o hardware e para gerência do sistema.

Por *default*, as chamadas efetuadas por um cliente são do tipo *blocking* (bloqueantes), isto é, o cliente permanece bloqueado até que: a chamada tenha sido transmitida para o objeto alvo; o código do objeto alvo tenha sido executado; e uma resposta tenha sido enviada de volta para o cliente. Esta é uma escolha natural, pois ela combina com a semântica normal das chamadas a funções em linguagens de programação, como em C++ e Java. A opção do cliente executar chamadas *non-blocking* (não bloqueantes) também está disponível e, neste caso, permite-se que o chamador rode em paralelo com o pedido emitido, podendo receber a resposta em uma oportunidade posterior. Outras opções são: chamadas do tipo *store-and-forward* (armazena e encaminha), nas quais o pedido é armazenado em um repositório não-volátil (persistente) antes de ser passado para o objeto alvo; chamada do tipo *publish-and-subscribe* (publica e faz assinatura), na qual uma mensagem é enviada tendo como destino um determinado tópico, de modo que qualquer objeto que estiver interessado neste tópico pode receber mensagens (desde que ele tenha permissão para tal)

3. Principais Funcionalidades do CORBA

As principais funcionalidades do CORBA são:

- OMG IDL (*Interface Definition Language*);
- Mapeamento para linguagens;
- Chamadas de operações e facilidades de submissão (*dispatch*) estáticas e dinâmicas;
- Adaptadores de objetos; e
- Protocolo IIOP (*Internet Inter-ORB Protocol*).

A Figura 4 ilustra os relacionamentos entre essas funcionalidades CORBA.

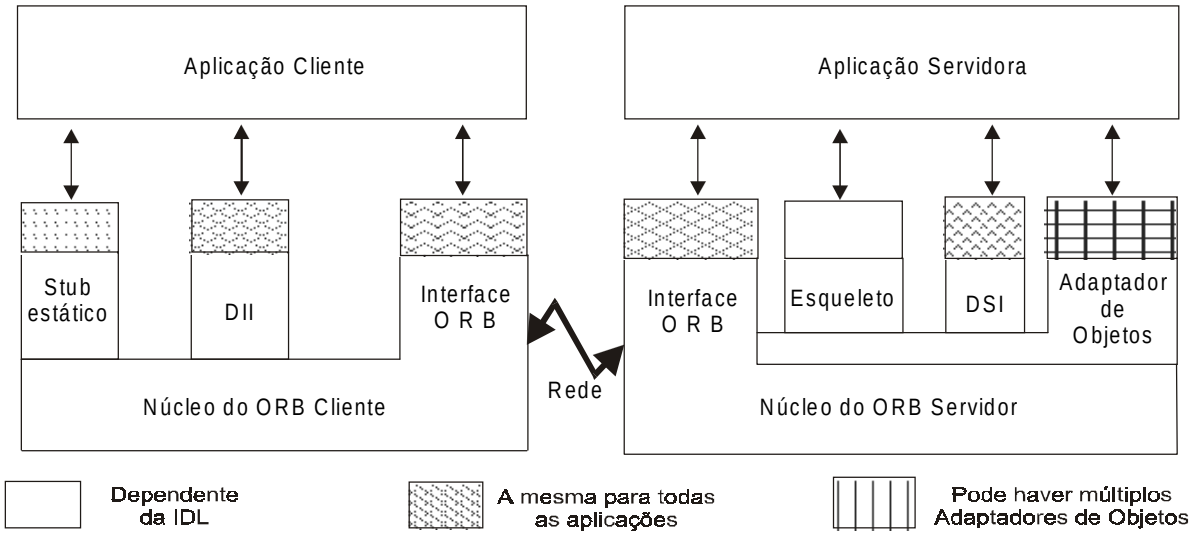

Figura 4. Arquitetura CORBA

3.1. O Fluxo Geral dos Pedidos

Na Figura 4, a aplicação cliente envia pedidos que são recebidos pela aplicação servidora e processados de acordo. Os pedidos fluem da aplicação cliente, através do ORB, até chegarem à aplicação servidora, obedecendo aos seguintes passos:

1. O cliente pode escolher se deseja fazer seus pedidos usando *stubs* estáticos já compilados em C++ a partir de uma definição de interface do objeto ou, então, usando uma interface de chamada dinâmica DII (*Dynamic Invocation Interface*). De qualquer modo, o cliente direciona os pedidos para o núcleo do ORB, o qual se encontra ligado (linkeditado) dentro do seu processo (do cliente).
2. O núcleo do ORB do cliente transmite o pedido para o núcleo do ORB que foi ligado com a aplicação servidora.
3. O núcleo do ORB do servidor submete o pedido para o adaptador de objetos, o qual foi o responsável pela criação do objeto alvo.
4. O adaptador de objetos submete por sua vez o pedido para o servente que estiver implementando o objeto alvo. Da mesma forma que no cliente, o servidor pode escolher dentre os mecanismos de submissão estático ou dinâmico para seus serventes. O servidor pode basear-se em esqueletos estáticos compilados em C++ a partir da definição da interface do objeto ou, então, seus serventes podem utilizar a interface DSI (*Dynamic Skeleton Interface*).
5. Depois que o servente atende ao pedido, ele retorna a resposta para a aplicação cliente.

No item 2.2 anterior, apresentamos os tipos de chamadas que podem ser executadas por uma aplicação cliente. Adicionalmente, o CORBA suporta diversos estilos de pedidos:

- quando um cliente emite um pedido de forma síncrona, ele fica bloqueado esperando por uma resposta. Estes tipos de pedidos são similares a RPCs (*Remote Procedure Calls*);

- um cliente que emite um pedido síncrono deferido, envia p pedido, continua o seu processamento e, então, mais tarde, procura receber uma resposta. Atualmente, este tipo de pedido só pode ser feito através do uso da interface DII;
- o CORBA também fornece pedidos de mão única (*one-way*) que representam pedidos do tipo melhor esforço (*best effort*), os quais não precisam ser realmente submetidos para o objeto alvo e não podem receber respostas. Os ORBs podem descartar de forma silenciosa os pedidos de mão única caso ocorra algum problema na rede, como o seu congestionamento ou outros tipos de limitações no uso de recursos, o que pode causar o bloqueio do cliente desde o momento que o pedido tiver sido encaminhado; e
- espera-se que em uma versão futura do CORBA, possivelmente a versão 3.0, seja possível o suporte a pedidos assíncronos, os quais podem permitir que servidores e clientes conectados ocasionalmente possam intercomunicar-se. A versão irá adicionar, também, suporte para a emissão de chamadas síncronas deferidas usando *stubs* estáticos ou então a DII.

4. OMG IDL (*Interface Definition Language*)

Descremos a seguir os componentes CORBA exigidos para que se possa emitir pedidos e receber respostas.

Para que se possa emitir chamadas para operações em um objeto distribuído, o cliente precisa conhecer a interface que é oferecida pelo objeto. A interface de um objeto é composta pelas operações que ele suporta e pelos tipos de dados que podem ser passados de/para essas operações. Exige-se dos clientes o conhecimento necessário sobre o propósito e a semântica das operações que eles desejam chamar.

Em CORBA, as interfaces para os objetos são definidas na OMG IDL (*Interface Definition Language*). Diferentemente de C++ ou de Java, a IDL não é uma linguagem de programação, de modo que, evidentemente, os objetos e as aplicações não podem ser implementadas em IDL. O único propósito da IDL é permitir que as interfaces de objetos sejam definidas de forma completamente independente de qualquer linguagem de programação em particular. Tal arranjo permite que aplicações implementadas em diferentes linguagens de programação possam interoperar. A independência de linguagem da IDL é fundamental para o principal objetivo do CORBA, o qual consiste em viabilizar sistemas heterogêneos e em integrar aplicações desenvolvidas separadamente.

A OMG IDL suporta tipos simples na forma de *builtins* como, por exemplo, tipo inteiro com ou sem sinal, assim como tipos construídos como, por exemplo, tipos enumerados, estruturas, uniões discriminadas, seqüências (ou vetores unidimensionais) e exceções. Esses tipos são usados para definir os tipos de parâmetros e os tipos de retorno das operações, as quais, por sua vez, são definidas dentro das interfaces. A IDL fornece também uma construção módulo usada para propósito de definição da abrangência de nomes. O exemplo a seguir ilustra uma definição simples em IDL:

```
interface Empregado {
            long numero( );
};
```

Esse exemplo define uma interface chamada `Empregado` que contém uma operação chamada `numero`. A operação numero não recebe argumentos e retorna um argumento do

tipo `long`. Um objeto CORBA que suporte a interface `Empregado` implementa a operação `numero` para retornar o número de matrícula do empregado representado por aquele objeto.

As referências a objetos são expressas em IDL pelo uso do nome de uma interface na forma de um tipo de dado. Por exemplo:

```
interface Registro_do_Empregado {
     Empregado lookup (in long no_emp);
};
```

A operação `lookup` na interface `Registro_do_Empregado` toma como argumento de entrada a matrícula do empregado e retorna uma referência a objeto do tipo `Empregado` que se refere ao objeto empregado identificado pelo argumento `no_emp`. Uma aplicação poderia utilizar essa operação para recuperar um objeto `Empregado` e, a seguir, utilizar o valor de referência ao objeto que for retornado para emitir chamadas e operações de `Empregado`.

Os argumentos das operações em IDL dever ter suas direções declaradas, para que o ORB possa identificar se seus valores devem ser enviados: do cliente para o objeto alvo; ou do objeto alvo para o cliente; ou em ambas direções. Na definição da operação `lookup`, a palavra-chave `in` significa que o argumento número do empregado (`no_emp`) deve ser passado do cliente para o objeto alvo.

Os argumentos podem também ser declarados como `out`, o que indica, como no caso de valores de retorno, que eles devem ser passados do objeto alvo de volta para o cliente. A palavra-chave `in-out` indica um argumento que é inicializado pelo cliente e é, a seguir, enviado do cliente para o objeto alvo; neste caso, o objeto pode modificar o valor do argumento e retornar o valor modificado de volta para o cliente.

Um aspecto marcante das interfaces IDL é que elas podem herdar características de outras interfaces. Tal possibilidade permite que novas interfaces sejam definidas em termos de interfaces já existentes, de modo que objetos que implementem uma dessas novas interfaces podem substituir os objetos que suportam as interfaces básicas já existentes. Por exemplo, considere as seguintes interfaces `Printer`:

```
interface Printer {
     void print ( );
};
interface ColorPrinter: Printer {
     enum ColorMode {BackAndWhite, FullColor};
           void set-color (in ColorMode mode);
};
```

A interface `ColorPrinter` foi derivada da interface `Printer`. Se uma aplicação cliente for escrita para lidar com objetos do tipo `Printer`, essa aplicação pode utilizar também um objeto que suporta a interface `ColorPrinter`, pois tais objetos suportam também a interface `Printer` por completo.

A IDL fornece um caso especial de herança: todas as interfaces IDL herdam implicitamente a partir da interface Object que é definida no módulo CORBA. Esta interface básica especial fornece as operações comuns a todos os objetos CORBA.

4.1. A OMG IDL e as Linguagens de Programação Suportadas

A noção de interação entre objetos é essencial para o CORBA. Outra noção essencial é aquela que determina que cada objeto tenha uma interface definida em uma linguagem que é especializada exatamente na definição dessas interfaces. Esta linguagem é chamada IDL (*Interface Definition Language*), usada exclusivamente para definir interfaces; ela não apresenta construções como por exemplo, variáveis, declarações, nem laços `if` e `while` que deveriam ser empregados na implementação de uma interface. Sua sintaxe é similar à de linguagens como C++ e Java, porém, a linguagem IDL é bem mais simples porque ela só precisa se preocupar com uma pequena parte dos componentes de uma linguagem de programação.

A principal construção em IDL é a interface, que é equivalente a uma classe em C++, ou a uma interface em Java. Cada interface define as operações que podem ser chamadas pelos clientes, porém como é óbvio, não se pode escrever código em IDL voltado para a implementação de tais operações.

Uma interface definida em IDL pode ser facilmente mapeada para uma definição em qualquer linguagem de programação, como por exemplo, C++, Java, Smallltalk, Visual Basic, C, dentre outras. Isto significa que, dada uma interface IDL para um objeto, ela pode ser traduzida para Java, por exemplo, para que um usuário Java desta interface possa utilizá-la, ou pode ser traduzida para C++, de modo que a interface possa ser implementada em um servidor. Esta é a chave para a importância da IDL: permitir que uma interface possa ser implementada em uma determinada linguagem e possa, em seguida, ser invocada a partir de uma outra linguagem qualquer. Adicionalmente, o módulo de execução (*runtime*) do ORB deve poder transmitir pedidos de um cliente através da rede para um objeto alvo que esteja rodando remotamente. A Figura 5 ilustra este conceito.

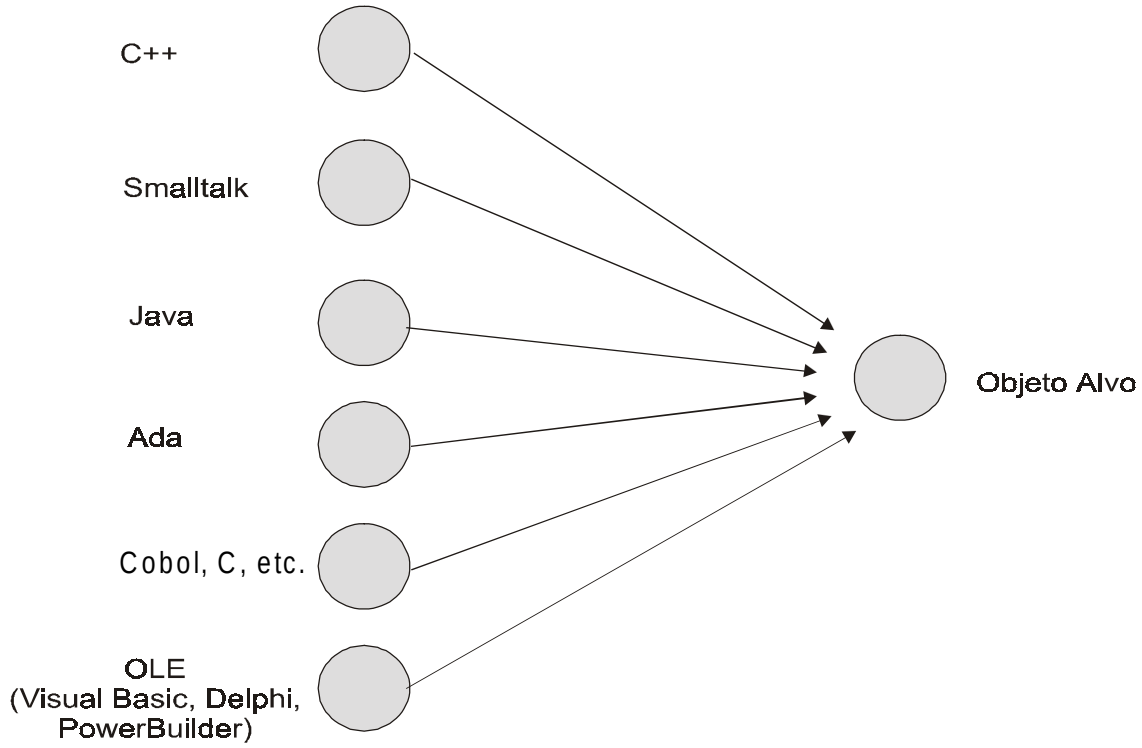

Figura 5. Facilidades de *internetworking* via IDL

A IDL é uma linguagem livre de construções complexas como por exemplo, operadores, funções virtuais, parâmetros *default*, etc. Esta característica torna a IDL uma linguagem de fácil aprendizado, e fácil de ser mapeada para linguagens de programação convencionais. Como a IDL define somente interfaces, ela não presia de características como amigabilidade, redefinição e herança privada, que são relacionadas com a implementação de uma interface (apesar de tais características aparecerem em definições de classes em C++).

A IDL inclui caraterísticas como: herança de interfaces, exceções e tipos de dados básicos e compostos. Sua simplicidade resulta em interfaces mais bem elaboradas, pois o projetista de interfaces só precisa se concentrar nos aspectos mais importantes do nível de aplicação.

Observe que o uso da IDL apresenta vantagens e desvantagens quando comparado a interfaces escritas em uma linguagem de programação convencional. Uma vantagem da IDL é que ela é independente da linguagem de programação empregada para implementar uma interface, o que significa dizer que ela pode ser usada para viabilizar a interoperabilidade entre diferentes linguagens sem exigir difíceis traduções entre essas linguagens de programação; A IDL é mutio simples, o que implica uma traduçào também simples para outras linguagens, de modo que estas outras linguagens podem ser usadas para implementar objetos CORBA, ou para emitir chamadas para eles. A IDL separa, dessa maneira, a especificação da implementação, de modo que nem mesmo a linguagem de programação usada para implementar um determinado objeto fica visível para os usuários deste objeto. Outra vantagem oferecida pela simplicidade da IDL é que existe uma tendência para a utilização exagerada de características complexas de uma linguagem, o que acaba causando distrações sobre o objetivo prático de uma interface. Uma pequena desvantagem apresentada no uso de uma linguagem como a IDL surge da necessidade do aprendizado de uma nova sintaxe (a da própria IDL). Porém, a curva de aprendizado para a mesma é muito curta devido à sua simplicidade.

5. Os Mapeamentos de Linguagens

A IDL é uma linguagem puramente declarativa, como visto anteriormente, e ela não pode ser usada para escrever aplicações reais. Ela não fornece estruturas de controle nem variáveis e, portanto, não pode ser compilada ou interpretada na forma de um programa executável. Ela é adequada somente para declarar interfaces para objetos e para definir os tipos de dados usados na comunicação com os objetos.

Mapeamentos de linguagens especificam como a IDL pode ser traduzida para diferentes linguagens de programação. Para cada construção IDL, um mapeamento da linguagem define que facilidades da linguagem de programação precisam ser empregadas para que se possa viabilizar a construção IDL para as aplicações. Por exemplo, em C++, as interfaces IDL são mapeadas para classes e as operações são mapeadas para funções membro dessas classes. De maneira similar, em Java, as interfaces IDL são mapeadas para interfaces públicas Java. Referências a objetos em C++ são mapeadas para construções que suportem a função `operator->` (isto é, ou um apontador para uma classe ou, então, um objeto de uma classe com uma função membro `operator->` sobrecarregada). Referências a objetos em C, por sua vez, são mapeadas para apontadores opacos (do tipo `void *`), e as operações são mapeadas para funções C que requerem um referência a um objeto opaco como sendo o seu primeiro parâmetro. O mapeamento da linguagem também especifica como as aplicações devem utilizar as facilidades ORB, e ainda como as aplicações servidoras podem implementar os seus serventes.

Os mapeamentos de linguagens da OMG IDL que são padronizados existem para diversas linguagens de programação. Existem mapeamentos de linguagens para C, C++, Smalltalk, Cobol, Ada e Java. Outros mapeamentos de linguagens também estão disponíveis, só que definidos de forma *independente* para linguagens como: Eiffel, Modula3, Perl, Tcl, Objective-C e Python.

Os mapeamentos de linguagnes são essenciais para o desenvolvimento de aplicações em CORBA. Eles viabilizam os meios concretos para os conceitos abstratos e para os modelos fornecidos pelo CORBA. Um mapeamento de linguagem completo e intuitivo facilita o desenvolvimento de aplicações CORBA em uma linguagem específica; por outro lado, um mapeamento de linguagem pobre, incompleto ou ineficiente, pode comprometer seriamente o desenvolvimento de uma aplicação CORBA. No caso dos mapeamentos de linguagens oficiais da OMG, ocorre uma revisão periódica, objetivando-se garantir a sua efetividade.

A existência de múltiplos mapeamentos de linguagens IDL implica que os desenvolvedores podem implementar diferentes partes de um sistema distribuído utilizando diferentes linguagens. Por exemplo, um desenvolvedor pode escrever uma aplicação servidora de alto desempenho em C++ visando a eficiência do sistema, e pode escrever os clientes desta aplicação na forma de *applets* Java, de modo que eles possam ser baixados da Web. A independ6encia de linguagem representa a principal funcionalidade do CORBA quando visto como uma tecnologia de integração voltada para sistemas heterogêneos.

6. A Chamada de Operações e as Facilidades de Submissão

As aplicações CORBA funcionam através da recebimento/emissão de pedidos de/para objetos CORBA. Existem dois enfoques para a emissão de pedidos em CORBA: Chamada e Submissão Estática; e Chamada e Submissão Dinâmica. Falamos um pouco sobre cada uma delas a seguir:

- Chamada e Submissão Estática

  Neste enfoque, a IDL é traduzida para *stubs* e esqueletos específicos para uma determinada linguagem, os quais serão compilados para dentro de uma aplicação. A compilação de *stubs* e de esqueletos para uma aplicação fornece à mesma um conhecimento estático a respeito dos tipos e funções das linguagens de programação mapeadas a partir de descrições IDL para os objetos remotos. Um *stub* representa uma função do lado do cliente que possibilita que a emissão de um pedido seja feita através de uma chamada de função local convencional. Em C++, um *stub* CORBA representa uma função membro de uma determinada classe. O objeto C++ local que suporta funções stub é geralmente chamado de *proxy* porque ele representa o objeto alvo remoto para a aplicação local. De maneira similar, um esqueleto representa uma função do lado do servidor que possibilita que uma emissão de pedido recebida por um servidor possa ser encaminhada para o servente apropriado.

- Chamada e Submissão Dinâmica

  Neste enfoque considera-se a construção e a submissão de pedidos CORBA em tempo de execução e não em tempo de compilação (como ocorre no enfoque estático). Devido ao fato de não haver informações vindas da compilação, a criação e a interpretação dos pedidos na hora da execução implica a necessidade de acesso a serviços que possam

fornecer informações a respeito de interfaces e de tipos. Uma aplicação pode obter este tipo de informação através da consulta a um operador humano via uma GUI *(graphical user interface*), ou pode obter tal informação a partir de um serviço chamado Repositório de Interfaces (*Interface Repository)* , o qual fornece acesso em tempo de execução às definições IDL.

Os desenvolvedores que optarem pelo desenvolvimento de aplicações utilizando linguagens com tipos estáticos, como C++ por exemplo, geralmente, preferem utilizar o enfoque de chamadas estático, porque ele fornece um modelo de programação mais natural. O enfoque dinâmico pode ser útil para alguns tipos de aplicações, como por exemplo, em *gateways* e em pontes, os quais devem receber e encaminhar pedidos sem necessitarem de informações do nível de compilação, como tipos de dados e interfaces envolvidas.

7. Os Adaptadores de Objetos

Em Corba, os adaptadores de objetos são usados para unir os serventes com o ORB. O padrão de projeto do adaptador (*Adapter Design Pattern*), que é independente do CORBA, descreve um adaptador de objetos como sendo um objeto que adapta a interface de um objeto para uma outra interface, a qual é necessária para um determinado chamador. Em outras palavras, um adaptador de objetos representa um objeto interposto que representa (ou age por procuração) um chamador quando ele emite pedidos para um objeto sem conhecer a real interface deste objeto.

Os adaptadores de objeto CORBA a tendem a três exigências básicas:

1. criam refer6encias a objetos, o que permite que clientes possam localizar objetos;
2. asseguram que cada objeto alvo seja encarnado por um servente;
3. interceptam os pedidos submetidos por um ORB a partir de um servidor e os direcionam para os serventes apropriados que encarnam cada um dos objetos alvo.

Sem os adaptadores de objetos, o ORB teria que fornecer ele mesmo tais características, adicionalmente a todas as suas outras atribuições. Desta forma, ele precisaria Ter uma interface muito complexa, o que tornaria bastante difícil o seu gerenciamento pelo OMG, e ademais, a quantidade de possíveis implementações de estilos de serventes seria limitada.

Em C++, os serventes representam instâncias de objetos C++. Eles são tipicamente definidos através da derivação a partir de classes “esqueleto” produzidas pela compilação das definições de interfaces IDL. Para implementar operações, sobrepõem-se funções virtuais das classes esqueleto básicas. Pode-se então registrar os serventes C++ no adaptador de objetos para permitir que o mesmo possa submeter pedidos para os serventes quando os clientes fazem pedidos aos objetos encarnados por estes serventes.

Até à versão 2.1, o CORBA continha especificações somente para o BOA (*Basic Object Adapter*). O BOA era o adaptador original de objetos do CORBA, e os seus projetistas consideravam que ele seria suficiente para a maior parte das aplicações, existindo outros adaptadores de objetos preenchendo somente funções direcionadas para certos nichos de mercado. Entretanto, o CORBA não se desenvolveu como era esperado devido aos seguintes problemas com a especificação BOA:

- A especificação BOA não considera o fato de que, apesar de sua necessidade de dar suporte aos serventes, os adaptadores de objetos tendiam a ser específicos para uma determinada linguagem. Como o CORBA, originalmente, fornecia mapeamento somente para a linguagem C, o BOA foi escrito para suportar somente serventes em C. Mais tarde tentou-se fazer alterações para que ele pudesse suportar também

serventes em C++, porém, tal esforço mostrou-se extremamente difícil. Em geral, um adaptador de objetos que fornece um suporte efetivo para os serventes em uma determinada linguagem de programação não irá, muito possivelmente, fornecer um nível de suporte que seja adequado para os serventes escritos em outras linguagens, devido às diferenças em estilos de implementação e na forma de uso destes serventes;

- Adicionalmente, várias características essenciais ficaram faltando na especificação do BOA. Certas interfaces não estavam definidas e não havia operações para o registro de serventes. Mesmo aquelas operações que foram especificadas continham várias ambigüidades. Assim, fornecedores de ORBs passaram a desenvolver suas próprias soluções proprietárias para preencher os espaços em branco, o que resultou em um nível de portabilidade de aplicações servidoras entre diferentes implementações ORB extremamente deficientes.

Em 1995, o OMG emitiu uma RFP (*Request for Proposals*) de número 27, chamada *Portability Enhancement* (melhoria da portabilidade) para tentar resolver os problemas citados com relação à especificação do BOA.

O CORBA versão 2.2 introduziu o adaptador POA (*Portable Object Adapter*) como um substituto para o BOA. Como o POA atende a uma gama completa de interações entre os objetos CORBA e os serventes de linguagens de programação, enquanto preserva a portabilidade das aplicações, a qualidade da especificação POA é muito superior à do BOA. Assim, a especificação BOA acabou sendo retirada do CORBA.

8. Protocolos Inter-ORB

Antes do CORBA 2.0, uma das principais reclamações com relação ao CORBA referia-se à sua falta de especificações para protocolos de redes padronizados. Para viabilizar a comunicação de aplicações remotas entre ORBs, cada fornecedor de ORB tinha que desenvolver o seu próprio protocolo de rede, ou então, tinha que usar algum emprestado de alguma tecnologia de sistemas distribuídos, o que resultou nas chamadas “ilhas de aplicações ORB”. Cada aplicação era construída tendo por base um ORB de um fornecedor específico, de modo que elas acabaram sem poder se comunicar umas com as outras.

O CORBA 22.o introduziu uma arquitetura de interoperabilidade geral para o ORB chamada de GIOP (*General Inter-ORB Protocol*). O GIOP representa um protocolo abstrato que especifica uma sintaxe de transferência e um conjunto padrão de formatos de mensagens voltados para permitir que ORBs desenvolvidos independentemente pudessem intercomunicar-se através de qualquer protocolo de transporte orientado à conexão. O IIOP (*Internet Inter-ORB Protocol*) especifica como o GIOP deve ser implementado sobre TCP/IP. Todos os ORBs que se dizem compatíveis com o CORBA 2.0 devem implementar GIOP e IIOP. A maioria dos ORBs o faz.

A interoperabilidade entre ORBs também exige formatos de referência a objetos padronizados. As referências a objetos são opacas (isto é, são não transparentes) para as aplicações, porém, elas contêm as informações que os ORBs precisam para poderem estabelecer a comunicação entre os clientes e os objetos alvo. O formato da referência a objeto padronizada, chamada IOR (*Interoperable Object Reference*), é bastante flexível para poder guardar informações para praticamente qualquer protocolo inter-ORB imaginável. Um IOR identifica um ou mais de um dos protocolos suportados e, para cada protocolo, contém informações específicas para cada um deles. Este esquema permite que novos protocolos sejam adicionados ao CORBA sem inviabilizar as aplicações existentes. Para o IIOP, um IOR

contém o nome de uma máquina (*host name*), um número de pacote TCP/IP, e uma chave de objeto, que identifica o objeto alvo em uma combinação de *host name* e número de porta.

9. A Emissão de Pedidos

Os clientes manipulam objetos através do envio mensagens. O ORB envia uma mensagem para um objeto sempre que um cliente chama uma operação. Para poder enviar uma mensagem para um objeto, um cliente deve manter uma referência ao objeto de interesse. A referência ao objeto age como um mecanismo que identifica de forma única o objeto alvo e encapsula toda a informação requerida pelo ORB para enviar a mensagem para o destino correto.

Quando um cliente chama uma operação através de uma referência a objeto, o ORB procede da seguinte maneira:

- localiza o objeto alvo;
- ativa a aplicação servidora, caso ela ainda não esteja rodando;
- transmite os argumentos da chamada para o objeto;
- ativa um servente para o objeto, se necessário;
- espera que o pedido se complete;
- retorna os parâmetros do tipo `out` ou `inout`, e o valor de retorno, para o cliente quando a chamada termina com sucesso;
- retorna uma exceção (incluindo os dados contidos nesta exceção) para o cliente quando a chamada falha.

O mecanismo completo de emissão de pedidos é totalmente transparente para o cliente, para o qual a emissão de um pedido para um objeto remoto irá parecer com um método de chamada comum para um objeto local em C++. Em particular, a emissão de pedidos apresenta as seguintes características:

- Transparência de localização

  Para o cliente, não interessa saber se o objeto alvo é local ao seu espaço de endereçamento ou se ele está implementado em um outro processo na mesma máquina, ou ainda, se ele está implementado em um processo em uma outra máquina. Os processos servidores não são obrigados a permanecerem na mesma máquina durante ttodo o tempo; eles podem movimentar-se de uma máquina para outra sem que os clientes precisem tomar conhecimento deste fato.

- Transparência dos Servidores

  O cliente não precisa saber qual servidor implementa que objeto.

- Independência de Linguagem

  O cliente não precisa se preocupar com qual a linguagem que está sendo usada no servidor. Por exemplo, um cliente C++ pode chamar uma implementação Java sem Ter necessariamente conhecimento deste fato. A linguagem de implementação dos objetos pode até ser alterada para objetos já existentes sem que isto venha a afetar os clientes.

- Independência de Implementação

O cliente não precisa saber como a implementação funciona. Por exemplo, o servidor pode implementar os seus objetos como serventes em C++, ou pode implementá-los utilizando técnicas não OO (como, por exemplo, implementando os objetos como agrupamentos de dados. O cliente irá enxergar a mesma semântica consistente de orientação a objetos independentemente de como os objetos são implementados no servidor.

- Independência de Arquitetura

O cliente desconhece detalhes sobre a arquitetura da CPU que é utilizada pelo servidor.

- Independência de Sistema Operacional

O cliente não precisa se importar com o sistema operacional que está sendo usado pelo servidor. O servidor pode até ser implementado sem precisar do suporte de um sistema operacional. Por exemplo, como um programa de tempo real embutido em algum equipamento específico.

- Independência de Protocolo

O cliente não precisa saber qual é o protocolo de comunicação que está sendo usado para a transferência de mensagens. Se houver vários protocolos disponíveis para a comunicação com o servidor, o ORB irá selecionar de forma transparente um dos protocolos que estiverem disponíveis em tempo de execução.

- Independência de transporte

O cliente é completamente ignorante com relação aos aspectos das camadas de enlace de dados e de transporte. Os ORBs podem, de forma transparente, utilizar várias tecnologias de redes, como Ethernet, ATM, TokenRing ou linhas seriais.

10. A Semântica para a Referência a Objetos

As referências a objetos são análogas aos apontadores de instâncias de classes em C++, porém, podem referencia-se a objetos implementados em outros processos (provavelmente localizados em uma máquina remota) ou podem referencia-se a objetos implementados no próprio espaço de endereçamento do cliente. Excetuando-se esta capacidade para endereçamento distribuído, as referências a objetos apresentam uma semântica muito próxima dos apontadores de instâncias de classes em C++, incluindo:

- cada referência a objeto identifica exatamente uma instância de objeto;
- várias referências podem denotar o mesmo objeto;
- as referências podem ser nulas (do tipo *nil*), isto é podem apontar para lugar nenhum;
- as referências podem oscilar (como em apontadores C++ que apontam para instâncias deletadas;

- as referências são opacas (os clientes não têm permissão para enxergar o seu conteúdo);
- as referências são fortemente direcionadas por tipos de dados;
- as referências suportam uma ligação (*binding*) atrasada;
- as referências podem ser persistentes; e
- as referências podem ser interoperáveis.

Esses pontos merecem uma explicação mais aprofundada porque eles são essenciais para o modelo de objetos do CORBA.

- Cada referência a objeto identifica exatamente uma instância de objeto

Da mesma forma que os apontadores de instâncias de classes em C++ identificam exatamente uma instância de objeto, uma referência a objeto denota exatamente um único objeto CORBA, o qual pode ser implementado em um espaço de endereçamento remoto. Um cliente que tem consigo uma referência a objeto deve esperar que essa referência denote sempre o mesmo objeto enquanto ele existir. Só permitido que uma referência a objeto perca o seu sentido quando o seu objeto alvo tiver sido permanentemente destruído. Depois que um objeto é destruído, suas referências tornam-se não funcionais. Isso significa que uma referência para um objeto destruído não pode, acidentalmente, vir a denotar outro objeto mais adiante.

- Um objeto pode ter várias referências

Várias referências diferentes podem denotar o mesmo objeto. Em outras palavras, cada referência indica exatamente um objeto, porém, permite-se que um objeto tenha vários nomes. O mesmo ocorre em C++. Um apontador de instância de classe em C++ denota exatamente um objeto, e o valor do apontador (por exemplo: 0x48bf0) identifica esse objeto. No entanto, a herança múltipla pode fazer com que uma instância C++ única possa ter até 5 (cinco) diferentes valores de apontadores. Essa situação é similar em CORBA. Se duas referências a objetos possuem conteúdos diferentes, isso não significa necessariamente que as duas referências denotam diferentes objetos. Assim, vê-se que uma referência a objeto não representa exatamente a identidade de um objeto. Tal fato afeta profundamente o projeto de sistemas orientados a objetos.

- Referências a objetos podem ser nulas

O CORBA define um valor nulo (*nil*) distinto para as referências aos objetos. Uma referência nula não aponta para lugar nenhum, e é análoga a um apontador nulo em C++. Referências nulas são bastante úteis para implementar semânticas do tipo “não encontrado”. Por exemplo, uma operação pode retornar uma referência nula para indicar que uma consulta vinda de um cliente em busca de um objeto não localizou a instância correspondente para este objeto. Referências nulas podem também ser usadas para implementar parâmetros de referência que sejam opcionais. A passagem de um valor nulo em tempo de execução indica que o parâmetro, de fato, não foi passado.

- Referências podem oscilar

Depois que um servidor passa uma referência a um objeto para um cliente, esta referência fica permanentemente fora do controle do servidor, e pode propagar-se livremente através de meios invisíveis ao ORB (por exemplo, na forma de uma cadeia de caracteres transmitida através de um *e-mail*). Isto significa dizer que o CORBA não possui qualquer mecanismo *builtin* automático que permita ao servidor informar a um cliente quando um objeto que pertença a uma determinada referência for destruído. De maneira similar, não existe uma forma automática *builtin* que permita a um cliente informar a um servidor que ele perdeu o interesse por uma determinada referência a objeto. Isto não quer dizer que não se possa criar tal semântica em uma aplicação que venha a exigi-la; significa somente que o CORBA não fornece tal semântica na forma de *builtins*. Para que se possa descobrir se uma referência a objeto continua a ser válida em um determinado momento (isto é, continua a denotar um objeto existente), o cliente pode chamar a operação `non-existent`, que é suportada por todos os objetos.

- Referências são opacas

As referências a objetos contêm vários componentes padronizados que são exatamente os mesmo em todos os ORBs, e contêm também, informações proprietárias que são específicas para um ORB em particular. Para permitir que exista compatibilidade de código fonte entre diferentes ORBs, clientes e servidores não podem enxergar a representação de uma referência a objeto. Ao contrário, eles devem tratar a referência ao objeto como uma caixa-preta que só pode ser manuseada através de uma interface padronizada. O encapsulamento de referências a objetos representa um aspecto essencial do CORBA. Esta funcionalidade permite que sejam adicionadas novas características de tempos em tempos, como por exemplo, diferentes protocolos de comunicação, sem quem se invalide o código fonte existente. Adicionalmente, os fornecedores podem utilizar a parte proprietária das referências a objetos para fornecerem características de valor adicionado, como um nível de desempenho melhorado, sem comprometer a interoperabilidade entre os ORBs.

- Referências são fortemente baseadas em tipos de dados ("*strongly typed*")

Cada referência a objeto contém uma indicação a respeito da interface suportada por ela. Tal esquema permite que o módulo de execução (*runtime*) do ORB possa forçar um certo nível de confiabilidade para os tipos de dados. Por exemplo, uma tentativa de envio de uma mensagem `Print` para um objeto `Empregado` (o qual não suporta tal operação) pode ser tratado em tempo de execução. Para linguagens com tipos de dados estáticos, como C++, a confiabilidade de tipos é forçada em tempo de compilação. O mapeamento da linguagem não permite que uma operação seja chamada a menos que o objeto alvo possa garantir a oferta desta operação na sua interface. (Tal afirmação só é verdadeira se forem usados os *stubs* gerados para a chamada de operações. Se for utilizada a interface DII, a confiabilidade de tipos estática será necessariamente perdida).

- Referências suportam ligações atrasadas

Os clientes podem tratar uma referência a objeto derivado como se ela fosse uma referência a objeto básico. Por exemplo, considere que uma interface `Gerente` seja derivada a partir de `Empregado`. Um cliente pode, de fato, manter uma referência para o objeto `Gerente`, porém, pode ficar imaginando que esta referência é do tipo `Empregado`. Do mesmo modo como acontece em C++, um cliente não pode chamar operações de `Gerente` através de uma referência ao objeto `Empregado` (pois isto iria violar a confiabilidade estática de tipos). Entretanto, se um cliente chamar a operação `numero` através da referência a `Empregado`, a mensagem correspondente ainda assim seria enviada para o servente `Gerente` que estiver implementando a interface `Empregado`. Este esquema é exatamente igual às chamadas de funções virtuais em C++: a chamada de um método através de um apontador base faz com que a função virtual seja chamada na instância derivada. Uma das maiores vantagens do CORBA, se comparado com plataformas de RPCs tradicionais, é que o polimorfismo e a ligação atrasada funcionam para os objetos remotos exatamente da mesma maneira que funcionariam para os objetos C++ locais. Isto significa que não existe nenhuma "parede" artificial na arquitetura na qual se exija o mapeamento de um projeto orientado a objetos para um paradigma de RPCs. Ao contrário, o polimorfismo funciona transparentemente ao longo da rede.

- Referências podem ser persistentes

Clientes e servidores podem converter uma referência a objeto para a forma de uma cadeia de caracteres e gravá-la em disco. Mais tarde, esta cadeia de caracteres pode ser convertida de volta na forma de uma referência a objeto que irá denotar o objeto original.

- Referências pode ser interoperáveis

O CORBA especifica um formato padronizado para as referências a objetos. Isto significa dizer que um determinado ORB pode utilizar referências criadas pelo ORB de outro fornecedor, sejam elas passadas como parâmetros ou como cadeias de caracteres. Por esta razão, estas referências a objetos padronizados são também chamadas de IOR (*Interoperable Object References*). Note que além do formato padrão IOR, um ORB pode implementar uma codificação de referência de forma proprietária. Esta facilidade pode ser bastante útil no caso de um ORB ser desenhado sob medida para um dado ambiente, como por exemplo, um banco de dados orientado a objetos. No entanto, referências proprietárias não podem ser intercambiadas entre ORBs de diferentes fornecedores.

10.1. A Aquisição de Referências

Os clientes só conseguem alcançar os objetos alvo através das referências a estes objetos. Um cliente não consegue se comunicar a não ser que ele possua uma referência a um objeto. Como é então que um cliente obtém referências? (se ele precisa de pelo menos uma referência a objeto para poder começar). Na realidade, as referências são publicadas de alguma maneira pelos servidores. Um servidor pode, por exemplo:

- retornar uma referência como o resultado de uma operação (na forma de um valor de retorno de um parâmetro do tipo `inout` ou `out`);
- pode divulgar a referência através de algum serviço bem conhecido, como por exemplo, através dos serviços CORBA *Naming* ou *Trading*;
- pode publicar uma referência a objeto convertendo-a para um formato de cadeia de caracteres, e gravando-a a seguir em um arquivo em disco;
- pode transmitir uma referência a objeto através de algum mecanismo "fora de banda", como por exemplo, enviando-a via *e-mail* ou publicando-a em uma página Web.

A forma mais comum de um cliente poder adquirir referências a objetos é recebendo-as como resposta a uma chamada de operação. Neste caso, as referências a objetos representam valores de parâmetros, e não são muito diferentes de algum outro tipo de valor, como por exemplo, de uma cadeia de caracteres. Os clientes simplesmente contactam um objeto, e o objeto retorna uma ou mais referências a objetos. Assim, os clientes podem navegar em uma "*web* de objetos", como se estivessem seguindo ligações de hipertexto.

Só muito raramente os clientes empregam outros métodos para obterem referências a objetos. Por exemplo, a busca (*lookup*) de uma referência em um servidor Trader (negociador) ou a leitura de uma referência a objeto a partir de um arquivo, ocorrem geralmente somente durante a fase de inicialização do cliente. Depois que o cliente tem em seu poder algumas referências a objetos, ele as utiliza para adquirir mais referências a outros objetos, através de chamada de operações. Independentemente da origem das referências a objetos, elas são sempre criadas pelo módulo de execução *runtime* do ORB, em atendimento ao pedido do cliente. Este enfoique esconde do cliente a representação interna das referências a objetos.

10.2. O Conteúdo de uma Referência a Objeto

Devido às caraterísticas de transparência de localização e de transporte fornecidas pelo CORBA, é necessário que exista uma quantidade mínima de informações encapsuladas em cada IOR. A Figura 6 ilustra uma visão conceitual do conteúdo de um IOR, que contém três partes principais:

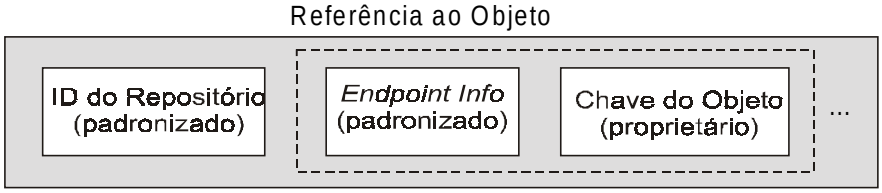

Figura 6. O conteúdo de uma referência a um objeto

- ID do Repositório

A identificação (ID) do Repositório consiste de uma cadeia de caracteres que identifica os tipos de dados mais comumente derivados do IOR no momento em que ele é criado. A ID do Repositório permite que se localize uma descrição detalhada da interface no *Interface Repository* (repositório de interfaces), caso o ORB forneça um.

- Informações sobre o nó final (*endpoint info*)

Este campo contém todas as informações exigidas pelo ORB para que ele possa estabelecer uma conexão física com o servidor que é o responsável pela implementação do objeto. A informação de nó final (*endpoint info*) especifica que protocolo deve ser usado, e contém informações sobre o endereçamento físico apropriado para um determinado protocolo de transporte. Por exemplo, no caso do IIOP, que é suportado por todos os ORBs interoperáveis, o campo *endpoint info* contém um nome de domínio da Internet ou, então, um endereço IP juntamente com um número de porta TCP.

A informação de endereçamento no campo de informações sobre o nó final (*endpoint info*) pode conter diretamente o endereço e o número da porta do servidor que implementa o objeto. Entretanto, na maioria dos casos, ele irá conter na realidade o endereço de um repositório de implementação que poderá ser consultado para que se possa localizar o servidor apropriado. Este nível extra de referência indireta permite que os processos em um determinado servidor possam migrar de uma máquina para outra sem invalidar as referências mantidas pelo cliente.

O CORBA permite também que informações para vários tipos de protocolos e transportes sejam embutidos na referência, o que possibilita que uma única referência possa suportar mais de um protocolo (o ORB irá escolher o protocolo mais apropriado de forma transparente). Uma versão futura do CORBA irá, muito provavelmente, permitir que o cliente possa influenciar a escolha do protocolo pela seleção de políticas ligadas à qualidade de serviço para as referências a objetos.

- A chave para o Objeto (*Object Key*)

O ID do repositório e o campo de informações do nó final são padronizados, enquanto a chave do objeto irá conter informações proprietárias. Como exatamente essa informação será organizada e usada irá depender do ORB. No entanto, todos os ORBs permitem que o servidor guarde dentro da chave do objeto um identificador de objeto que seja específico para uma determinada aplicação. Isto ocorre quando o servidor cria a referência para o objeto. Este identificador de objeto será usado pelo ORB e pelo adaptador de objetos do lado do servidor, para que se possa identificar o objeto alvo no servidor para cada pedido que ele venha a receber.

O módulo de execução (*runtime*) do lado do cliente simplesmente envia a chave na forma de uma gota de dados binários com cada pedido que ele emite. Portanto, o cliente não se preocupa com o fato de que os dados de referência estejam em um formato proprietário. Essa referência jamais será lida por qualquer ORB, exceto por aqueles ORBs que estiverem responsáveis pelo objeto alvo (que é o mesmo ORB que criou a chave do objeto).

A combinação da informação de nó final com as chaves de objetos pode ocorrer várias vezes em um IOR. Vários pares nó-final/chave, conhecidos como perfis multicomponetes (*multicomponent profiles)* permitem que um IOR suporte eficientemente mais do que um protocolo e transporte. Um IOR pode conter, também, perfis múltiplos, cada um contendo informações distintas sobre protocolo e transporte. O módulo de execução do ORB escolhe dinamicamente o protocolo que deve ser usado, dependendo daquele que seja usado tanto pelo cliente quanto pelo servidor.

Desta forma, todos os ingredientes essenciais para que se possa encaminhar pedidos com sucesso são encapsulados em uma referência. O ID do repositório fornece validação de tipos de dados. A informação sobre o nó final é usada pelo ORB do lado do cliente para identificar o espaço de endereçamento alvo correto, e a chave do objeto é usada pelo ORB do lado do servidor para identificar o objeto alvo dentro do seu espaço de endereçamento.

11. Referências a *Proxies*

Quando uma referência é recebida por um cliente, o módulo de execução do lado do cliente gera a instância para um objeto *proxy* (ou simplesmente, *proxy* que significa procurador em inglês) no espaço de endereçamento do cliente. Um *proxy* é uma instância em C++ que fornece ao cliente uma interface para o objeto alvo.

A interface no *proxy* é a mesma interface que existe no objeto remoto; quando um cliente chama uma aplicação no *proxy*, o *proxy* envia uma mensagem correspondente para o servente remoto. Em outras palavras, o *proxy* delega os pedidos para o servente remoto correspondente, e atua como um representante local para o objeto remoto, como ilustrado na Figura 7.

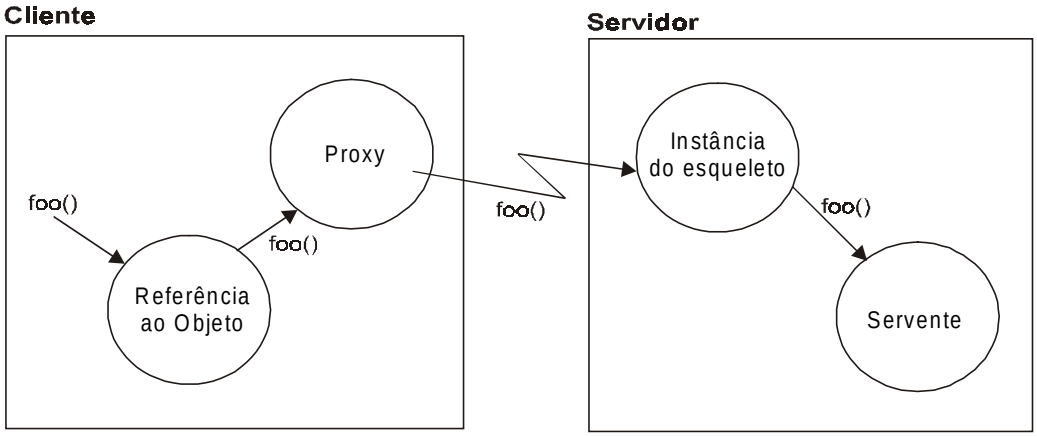

Figura 7. Um *proxy* local representando um objeto remoto

O mapeamento em C++ não se modifica caso o cliente e o servidor estejam localizados no mesmo espaço de endereçamento. De maneira particular, não há necessidade de alterações no código fonte nem do cliente nem do servidor, caso se deseje ligar o servidor juntamente com o cliente, como ilustrado na Figura 8.

Se o cliente e o servidor estiverem localizados no mesmo espaço de endereçamento, um pedido emitido por um cliente continuará sendo encaminhado pelo *proxy* de forma transparente para o servente correto; assim, preserva-se a transparência de localização do CORBA. Alguns ORBs não utilizam um *proxy* quando o cliente e o servidor compartilham o mesmo espaço de endereçamento. Ao invés disso, o servente do objeto atua como um *proxy*. No entanto, tais implementações não estão de acordo com a especificação POA, e não preservam a propriedade de transparência de localização. Pode-se, então, considerar o caso de não se implementar um *proxy* quando cliente e servidor compartilham o mesmo espaço de endereçamento como sendo uma opção deficiente.

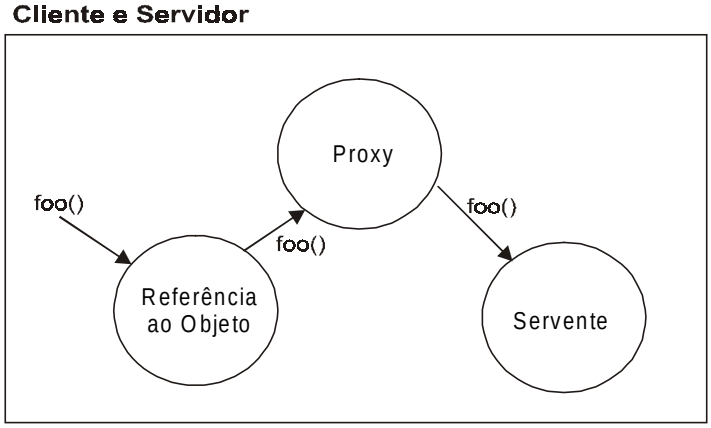

Figura 8. *Proxy* para um objeto no mesmo espaço de endereçamento do cliente e do servidor

Tanto no cenário remoto quanto no cenário onde cliente e servidor compartilham o mesmo espaço de endereçamento, o *proxy* cuida de delegar as chamadas a operações emitidas pelo cliente para o servente. No cenário remoto, o *proxy* envia o pedido através da rede, enquanto no cenário onde cliente e servidor compartilham o mesmo espaço de endereçamento, o pedido é enviado através de chamadas a funções C++.

A interação entre o esqueleto e o servente no caso remoto é geralmente implementada como uma chamada de função virtual em C++.

Observe que a instância *proxy* fornece ao cliente uma interface que é específica para o tipo de objeto que estiver sendo acessado. A classe para o *proxy* é gerada a partir da definição IDL para a interface correspondente, e implementa o *stub* através do qual o cliente submete chamadas. Este enfoque assegura a confiabilidade dos tipos de dados; o cliente não pode invocar uma operação a não ser que ele mantenha um *proxy* para o tipo correto, pois somente um *proxy* do tipo correto terá a função membro correta.

12. O Desenvolvimento de Aplicações em CORBA

Ao longo deste trabalho, foram apresentados, de forma introdutória, todas as partes do CORBA necessárias para o desenvolvimento de aplicações. Nesta seção, são formuladas algumas recomendações de caracter geral requeridos para a construção de sistemas baseados no CORBA. A intenção não fornecer um falso sentimento de simplicidade, mas sim ajudar ao leitor a compreender como todas as partes do CORBA, que foram descritas até agora, relacionam-se umas com as outras no contexto do desenvolvimento de aplicações.

Para que possamos desenvolver uma aplicação CORBA em C++ composta por duas partes, uma cliente e uma servidora, pode-se observar as seguintes recomendações:

1. Determinar que objetos compõem a aplicação e definir as suas interfaces em IDL;
2. Compilar as definições IDL em *stubs* e esqueletos C++;
3. Declarar e implementar classes serventes em C++ que possam encarnar os objetos CORBA;
4. Escrever o programa principal para o servidor;
5. Compilar e ligar os arquivos de implementação juntamente com os *stubs* e esqueletos gerados para que se possa criar os módulos executáveis do servidor; e
6. Escrever, compilar e ligar o código do cliente juntamente com os *stubs* gerados.

A seguir, detalhamos cada uma destas recomendações:

1. Determinar que objetos compõem a aplicação e definir as suas interfaces em IDL

Assim como no desenvolvimento de qualquer programa orientado a objetos, deve-se determinar que objetos compõem a aplicação, definir suas interfaces, e definir como eles relacionam-se uns com os outros. Este processo é geralmente um processo interativo e trabalhoso, de modo que o CORBA não irá tornar esta parte do ciclo de desenvolvimento mais fácil para o programador.

Na realidade, o projeto de uma aplicação CORBA, assim como o projeto de qualquer aplicação distribuída, é geralmente mais difícil que o projeto para um programa normal, pois deve-se considerar uma série de questões ligadas ao aspecto distribuído da aplicação. Apesar do CORBA, juntamente com seus mapeamentos de linguagens, esconderem a maior parte da complexidade e muito dos detalhes de baixo nível associados com a programação de redes, ele não trata de forma mágica os problemas encontrados em sistemas distribuídos como, por exemplo, o atraso de mensagens, o particionamento de redes, e falhas parciais no sistema. Tomar por base um ORB para o desenvolvimento de uma aplicação irá certamente ajudar o programador. Devemos, entretanto, continuarmos a considerar questões como o atraso ou as falhas distribuídas se desejarmos escrever aplicações distribuídas de alta qualidade.

2. Compilar as definições IDL em *stubs* e esqueletos C++

As implementações de ORBs geralmente fornecem compiladores IDL que obedecem às regras de mapeamento de linguagens que compilam a definição IDL em *stubs* de clientes e em esqueletos de servidores. Para C++, os compiladores IDL emitem, tipicamente, arquivos de cabeçalho em C++ que contêm declarações para classes de *proxies*, esqueletos de servidores, e outros tipos que possam ser suportados. Os compiladores IDL geram, também, arquivos de implementações em C++, que implementam as classes e os tipos declarados nos arquivos de cabeçalho.

Traduzindo-se as definições IDL para C++, gera-se um código base que permite que sejam escritos clientes e serventes que acessam e implementam objetos CORBA, respectivamente, os quais suportam as interfaces IDL definidas previamente.

3. Declarar e implementar classes serventes em C++ que possam encarnar os objetos CORBA

Cada um dos objetos CORBA devem ser encarnados por uma instância de uma classe servente em C++ antes que o ORB possa submeter pedidos para eles. Deve-se definir as classes serventes e implementar as suas funções membro (que representam seus métodos em IDL) para que se possa executar os serviços que se espera que os objetos CORBA possam fornecer para os clientes.

4. Escrever o programa principal para o servidor

Como ocorre em todos os programas em C++, a função `main` fornece os pontos de entrada e de saída para todas as aplicações CORBA em C++. Para o servidor, a função `main` deve inicializar o ORB e o POA, criar alguns serventes, cuidar para que os serventes encarnem os objetos CORBA e, finalmente, iniciar o recebimento dos pedidos.

5. Compilar e ligar os arquivos de implementação juntamente com os *stubs* e esqueletos gerados para que se possa criar os módulos executáveis do servidor

Para um servidor C++, deve-se fornecer as implementações do método. Os *stubs* e esqueletos que foram gerados fornecem as implementações dos tipos da IDL e fornecem, também, o código de submissão de pedidos usados para traduzir os pedidos CORBA que estão chegando para chamadas de funções nos serventes.

6. Escrever, compilar e ligar o código do cliente juntamente com os *stubs* gerados

Finalmente, implementam-se os clientes para que se possa, de forma inicial, obter as referências a objetos para os objetos CORBA. Para poder requisitar serviços, o cliente emite chamadas a operações nos objetos CORBA. O código cliente emite pedidos e recebe respostas, como se estivesse executando chamadas a funções normais em C++. Os *stubs* gerados traduzem estas chamadas a funções para pedidos enviados para objetos no servidor.

Naturalmente, essas recomendações podem variar de alguma maneira, dependendo da natureza da aplicação. Por exemplo, em alguns casos o servidor já existe, e o programador só precisará escrever o cliente. Neste caso, só existe a necessidade que se sigam as recomendações relacionadas com o desenvolvimento de clientes.

12.1 O Relacionamento com a Análise e o Projeto Orientados a Objetos

No projeto de um grande sistema, após a primeira fase de levantamento de necessidades, parte-se para uma análise orientada a objetos e depois para o projeto do sistema que se deseja construir. A fase de análise identifica as classes que capturam as entidades – os objetos de negócios – do sistema e os relacionamentos entre elas. A fase de projeto é usada para decidir a maneira como estas classes devem ser implementadas e também para decidir que outras classes precisam ser incluídas. Todos esses estágios precedem a implementação, na qual escreve-se o código fonte em uma determinada linguagem de programação e testa-se este código. Detalhes das técnicas empregadas diferem para diferentes métodos (por exemplo, técnicas de Modelagem de Objetos, Booch, etc), porém, em todos os casos, os princípios são os mesmos.

A análise e o projeto orientados a objetos são extremamente úteis no desenvolvimento de um sistema em CORBA, pois ajudam a identificar os objetos do sistema e o seus tipos. No entanto, é muito pouco provável que todos os objetos identificados venham a ser implementados como objetos CORBA; isto é, que todos estes objetos identificados possam Ter uma interface IDL. Considere, por exemplo, uma aplicação para compra eletrônica. A análise do sistema identificará, com grande possibilidade, classes como clientes, contas, mercadorias, compras, imagens, etc., e a fase de projeto deve adicionar classes, como por exemplo, aquelas exigidas para ajuste de carga em um servidor. Provavelmente alguma, ou então, todas as classes introduzidas na fase de análise terão que ser definidas em IDL, modelando-se então os objetos de negócios que poderão ser utilizados pelos clientes. Estes objetos tornam-se então disponíveis a partir de clientes remotos que utilizam diferentes sistemas operacionais, e diferentes linguagens de programação.

Por outro lado, é muito pouco provável que várias das classes introduzidas na fase de projeto tornem-se disponíveis para os clientes. Se tais classes não forem independentes da maneira como o sistema será implementado, então elas podem ser indisponibilizadas para os clientes. No entanto, se o sistema precisa ser implementado em diversas camadas, então algumas destas camadas podem ser definidas em IDL, apesar de tais interfaces não precisarem ser disponibilizadas para os sistemas finais.

Transformar alguns objetos em objetos CORBA pode configurar-se, também, como uma oportunidade para se incluir algumas funções adicionais (operações, em termos de CORBA) para determinadas classes. Considere, por exemplo, uma classe introduzida durante a fase de análise, contendo dez atributos que podem ser acessados por outros componentes do sistema. Em um sistema que tivesse sido implementado em um espaço de endereçamento único, faz sentido que se exija que o cliente tenha que chamar uma função para ler cada um dos atributos à medida que se necessita deles, mesmo que se tenha de ler os dez atributos um a um. Em um sistema distribuído, porém, é pouco provável que tal estratégia forneça um bom desempenho. Mesmo que o cliente não se incomode de fazer as dez chamadas, uma a uma, de forma remota, o servidor terá que tratar estas dez chamadas sucessivas para o mesmo cliente, o que irá, certamente, limitar a sua capacidade de atendimento aos outros clientes do sistema. Portanto, é muito melhor que se adicione uma operação única onde seja possível recuperar todos os dez atributos (em uma estrutura de dados), ou parte deles, de uma só vez.

Pode ser também necessário que se alterem algumas operações. Por exemplo, se uma dada operação incrementa um valor, pode ser interessante a inclusão de um parâmetro que retorna o novo valor para o chamador. Tal esquema poderia ser empregado no caso de ocorrer com muita ferqüencia o fato de uma alteração de valor implicar um pedido imediato deste novo valor por parte do cliente.

Portanto, vê-se que é muito importante que os programadores tratem de questões como estas desde a implementação da primeira versão da sua aplicação, evitando perdas de tempo com questões de interconexão de baixo nível. As definições IDL dos objetos CORBA são bem simples de serem alteradas, e pode-se adicionar novas interfaces que melhoram aquelas já existentes.

13. Observações Finais

O CORBA é suportado atualmente em quase todas as combinações de hardware e sistemas operacionais disponíveis, e está disponível a partir de vários fornecedores, existindo inclusive versões *freeware*. Está disponível também para várias linguagens de programação.

CORBA é encontrado como plataforma básica para viabilizar a criação de aplicações críticas em ambientes empresariais como: saúde, previdência privada, telecomunicações, bancos e indústrias. CORBA é voltado para o desenvolvimento de sistemas diversos como: *middleware* para integração de software, sistemas operacionais, e até como *toolkit* para *desktops*. A sua primeira versão publicada em 1991. Em 1994, deu-se a conclusão do processo de padronização do mapeamento de CORBA para C++; tal mapeamento foi publicado com o CORBA 2.0, que representa uma versão estável e portável. A importância da linguagem C++ deve-se ao fato de a mesma ser uma linguagem multi-paradigma, que suporta uma variedade de enfoques para o desenvolvimento de aplicações, incluindo: Programação estruturada; Abstração de dados; Programação OO; e, Programação genérica. C++ é a linguagem de implementação dominante para o CORBA, apesar de Java estar começando a ser usada no desenvolvimento de clientes.

O CORBA fornece as abstrações e serviços necessários ao desenvolvimento de aplicações distribuídas e portáveis, sem a necessidade de que o programador se preocupe com detalhes de baixo nível. Ele fornece suporte para vários modelos do tipo pedido-resposta para a localização e a ativação transparente de objetos, e para a independência de linguagens de programação e de sistemas operacionais, o que propicia uma base sólida tanto para a integração de sistemas legados quanto para o desenvolvimento de novas aplicações.

Os desenvolvedores de aplicações definem as interfaces para os objetos CORBA em IDL, uma linguagem declarativa no estilo do C++. Utiliza-se a IDL para definir tipos de dados como estruturas, seqüências, e arranjos para serem passados para as operações suportadas pelos objetos. Através do uso de técnicas de desenvolvimento orientadas a objetos, pode-se agrupar operações inter-relacionadas na forma de interfaces, da mesma maneira como se definem as funções membro inter-relacionadas em C++ na forma de classes C++.

Para que se possa implementar objetos CORBA em C++, deve-se criar instâncias de objetos C++ chamadas serventes, e registrá-las no POA. O ORB e o POA cooperam para submeter para o servente que encanar este objeto todos os pedidos emitidos para um determinado objeto alvo.

Os clientes emitem pedidos através de referência a objetos, que são entidades opacas que contêm as informações de comunicação utilizadas pelos ORBs para direcionar os pedidos para os seus objetos alvo. IORs fornecem formatos padronizados que permitem que os ORBs que tenham sido desenvolvidos independentemente possam interoperar.

A seguir, enumeramos uma série de requisitos preliminares para o desenvolvedor de aplicações distribuídas em CORBA usando C++:

- Experiência com programação de redes (por exemplo, com a interface *Socket*);
- Experiência com alguma plataforma de RPC (*Remote Procedure Call*);
- Experiência com a linguagem C++ padrão da ISO/IEC: incluindo o domínio de conceitos como herança, funções virtuais, sobrecarga de operadores, *templates*;
- Experiência com *The Standard Templete Library* (STL);
- Experiência com programação com *threads*;
- Experiência com a Análise e o Projeto de sistemas orientados a objetos.
- Experiência com o uso de Interfaces Dinâmicas:
  - Dynamic Invocation Interface (DTI)
  - Dynamic Skeleton Interface (DSI)
  - Repositóro de Interfaces
- Experiência com o uso da facilidade *Object-By-Value* (OBV)
- Conhecimento dos Serviços CORBA:
  - Transaction Service
  - Security Service
  - Naming Service
  - Trading Service
  - Event Service
- Conhecimento sobre o Portable Object Adapter (POA).

A Plataforma de testes adotada por nosso grupo de trabalho consiste do ORB ORBIX da empresa irlandesa IONA Technologies, o qual encontra-se disponível nas estações de trabalho do Laboratório RAVEL.

14. Referências Bibliográficas

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

[2] Baker, S. *CORBA Distributed Objects using ORBIX*. Addison-Wesley/ACM Press, United Kingdom, 1997.

[3] Redlich, J., Suzuki, M. and Weinstein, S. Distributed Object technology for Networking. IEEE Communications Magazine, Vol. 36, No. 10, October 1998.

[4] Chan, M.C. and Lazar, A. A. Designing a CORBA-based High Performance Open Programmable Signaling System for ATM Switching Platforms. IEEE JSAC, Vol. 17, No. 9, September 1999.

[5] Schmiidt, D. C. Evaluating Architectures for Multithreaded Object Request Brokers. Communications of the ACM, Vol. 41, No. 10, October 1998.

[6] Lewandowski, S. M. Frameworks for Component-Based Client/Server Computing. ACM Computing Surveys, Vol. 30, No. 1, March 1998.

[7] Haggerty, P. and Seetharaman, K. The Benefits of CORBA-Based Network Management. Communications of the ACM, Vol. 41, No. 10, October 1998.