Programação Concorrente e Distribuída
O Padrão CORBA

Thaís Vasconcelos Batista

2000.2

1 Introdução

Padrões para interoperabilidade entre objetos distribuídos, também chamados de modelos para integração de objetos ou infra-estruturas de componentes de software [1] permitem que objetos escritos em diferentes linguagens e compilados por diferentes compiladores comuniquem-se via protocolos padronizados que oferecem altos níveis de transparência na utilização de objetos heterogêneos distribuídos. A linguagem e o compilador usado para criar objetos servidores são totalmente transparentes para o cliente, bem como a localização do objeto. Como forma de interação, estes padrões utilizam o modo cliente-servidor. Nesta classe de padrões, um dos que tem tido mais destaque é o padrão CORBA (Common Object Request Broker Architecture) [2, 1, 4], por ser uma especificação aberta, independente de linguagem e fabricante. A infra-estrutura oferecida por CORBA gerencia todos os detalhes da interoperabilidade entre objetos assegurando a possibilidade de se desenvolver sistemas baseados em interação que incorporam componentes de múltiplas origens.

CORBA começou a surgir em 1989, quando um grupo de empresas reuniram-se em uma organização, denominada OMG (Object Management Group), com a finalidade de especificar uma arquitetura global e normas para permitir o trabalho conjunto de componentes de diferentes origens. A idéia era aproveitar os benefícios do paradigma de orientação a objetos, principalmente a noção de encapsulamento dos dados e da implementação de um objeto atrás de sua interface. O OMG definiu a arquitetura CORBA, que possui uma camada comum, chamada barramento, através da qual objetos transparentemente trocam mensagens. Os objetos que interagem através deste barramento tem a interface descrita na mesma linguagem - Interface Definition Language (IDL) e implementação escrita em qualquer linguagem que possua um mapeamento para CORBA (*binding*). O OMG publica especificações de interfaces escritas em IDL que são usadas por desenvolvedores de software para a implementação de produtos CORBA. Para a integração entre aplicações, o OMG definiu a OMA (Object Management Architecture). Enquanto CORBA permite a interoperação entre objetos, OMA agrupa o conjunto de objetos CORBA em serviços e facilidades que oferecem suporte para o desenvolvimento de aplicações que usam objetos CORBA.

Em 1990 o OMG publicou a primeira especificação da arquitetura CORBA, cuja revisão foi publicada em 1992. Em 1996 surgiu a especificação CORBA 2.0 introduzindo novas características, dentre as quais um protocolo padrão para comunicação entre ORBs - o GIOP (General Inter-Orb Protocol). Recentemente foi publicada a especificação CORBA 3.0 que, entre outras inovações, especifica um modelo de componentes CORBA. Todas as especificações do OMG estão disponíveis em http://www.omg.org/library/

Esta apostila proporciona uma visão global do padrão CORBA incluindo a descrição da Arquitetura OMA, de cada um dos seus componentes, da linguagem para descrição de interfaces (IDL) e de alguns serviços principais, como os serviços para localização de objetos - Nomes e Trading.

2 Arquitetura Global do OMG

O OMG propôs uma Arquitetura para Gerenciamento de Objetos (OMA) que congrega o barramento CORBA - ORB - e serviços para suporte ao desenvolvimento de aplicações distribuídas. Cada serviço é descrito por interfaces IDL e compostos por vários objetos CORBA. Clientes acessam os serviços através do ORB. A figura 1 ilustra os quatro elementos da Arquitetura que serão comentados a seguir, com exceção do ORB que será comentado na subseção 5.

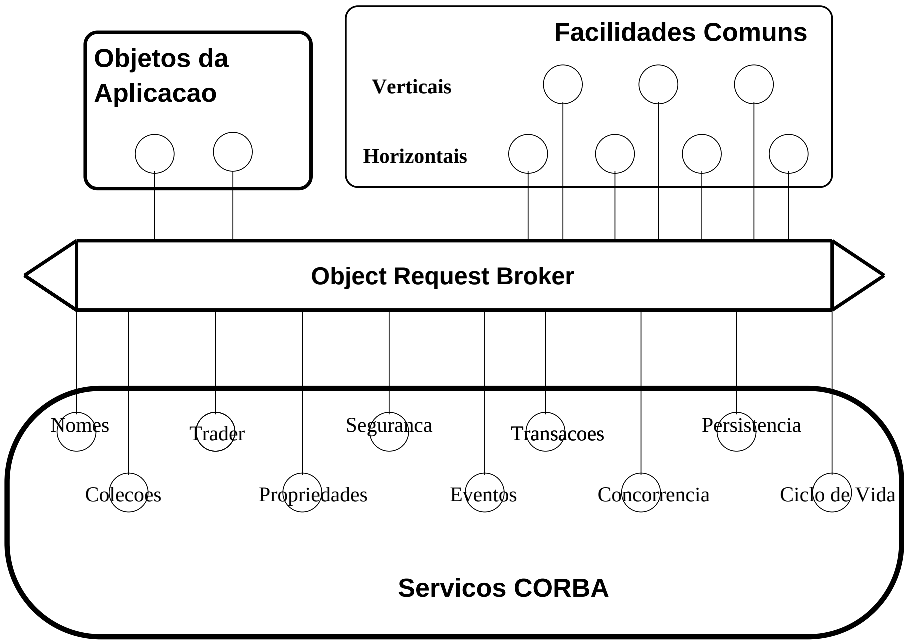

Figura 1: Visão Geral da Arquitetura OMA

Serviços CORBA

Na camada *serviços CORBA* estão os serviços básicos e fundamentais para o desenvolvimento de aplicações, como serviços de Nomes, Trader, Eventos, Ciclo de Vida e outros. Todos os serviços são oferecidos por interfaces IDL. Alguns serviços serão comentados mais adiante. Maiores detalhes sobre cada serviço podem ser encontrados em [2].

Facilidades Comuns CORBA

A camada *facilidades Comuns CORBA* oferece serviços intermediários para aplicações, como, por exemplo, serviços financeiros, facilidades para gerenciamento de sistemas, para definição de interfaces com usuário, etc. As facilidades estão divididas em duas categorias: Facilidades Horizontais e Verticais. As facilidades horizontais incluem funções necessárias à maioria dos sistemas, independentemente do propósito a que eles se destinam. As facilidades verticais incluem funções adequadas à aplicações de domínios específicos, como sistemas bancários, comércio eletrônico, CAD, etc.

Objetos da Aplicação

Esta camada compreende os objetos desenvolvidos especificamente para cada aplicação. Não é padronizada pelo OMG.

3 Referência do Objeto

Todo objeto CORBA possui uma referência, chamada *referência do objeto*, que é atribuída pelo ORB na criação do objeto. A referência do objeto é a identificação de um objeto CORBA. Para utilizar um objeto, o cliente deve obter a referência do objeto pois em uma invocação de um método sobre o objeto, o ORB o identifica através da sua referência. A referência de um objeto pode ser obtida de várias formas, através do Repositório de Interfaces, do serviço de Nomes ou do serviço de Trading.

4 Linguagem para Definição de Interfaces - IDL

Todos os objetos e serviços CORBA são definidos por uma interface escrita na *Linguagem para Definição de Interfaces (IDL)*. A IDL é uma linguagem declarativa. Para publicar um objeto CORBA, programadores escrevem o código do objeto usando uma linguagem específica e usa a IDL apenas para expressar os seus atributos, as operações que o objeto pode realizar, os seus parâmetros de entrada e saída, os eventos que o objeto pode emitir e as exceções. Os métodos especificados em IDL podem ser chamados de qualquer linguagem que ofereça amarrações CORBA (*CORBA bindings*), um mapeamento entre a linguagem e a IDL que especifica como tipos IDL são convertidos em tipos e chamadas de funções da linguagem. Através da IDL são declaradas as operações que um objeto oferece e como elas devem ser chamadas pelos clientes. As especificações em IDL são fortemente tipadas pois definem o tipo de retorno das operações, o tipo dos parâmetros e dos atributos. A IDL permite se declarar herança de interfaces.

A figura 2 exibe os elementos que constituem a estrutura da interface IDL.

```
module <identificador>
{
<declaracoes_de_tipo>;
<declaracoes_de_constantes>;
<declaracoes_de_excecoes>;

   interface <identificador> [:<lista_interfaces_base>]
   {
       <declaracoes_de_tipo>;
       <declaracoes_de_constantes>;
       <declaracoes_de_atributos>;
       <declaracoes_de_excecoes>;

      [<tipo_operacao>]<identificador>(<parametros>)
      [raises excecao] [contexto];
              .
              .
      [<tipo_operacao>]<identificador>(<parametros>)
      [raises excecao] [contexto];
              .
              .
    }
}
```

Figura 2: Estrutura da Interface IDL

Módulos

Módulos são identificados pela palavra-chave *module* e servem para agrupar um conjunto de interfaces, oferecendo uma forma de estruturar nomes a fim de evitar conflitos de nomes entre várias descrições.

Atributos

Atributos descrevem propriedades de um objeto e por *default* podem ser consultados e modificados. Atributos que não podem ser modificados devem ser declarados como *readonly*.

Interfaces

Interfaces definem um conjunto de métodos (*operações* na terminologia OMG) e contém declarações de tipos, atributos e exceções. Uma interface pode ter várias exceções para indicar quando uma operação não teve sucesso. IDL permite herança múltipla, o que significa que uma interface pode ser derivada de uma ou mais interfaces.

Operações

As operações, também chamadas de métodos, representam os serviços que o objeto oferece. A descrição de uma operação inclui o seu nome, seu modo de execução, os tipos de seus prâmetros e o retorno. Os parâmetros possuem um *modo* que indica se o valor é passado do cliente para o servidor (*in*), do servidor para o cliente (*out*) ou de ambas as partes (*inout*). Os parâmetros possuem tipos, para restringir os valores que eles podem assumir. A assinatura do método opcionalmente define as exceções que um método exibe quando detecta um problema na execução do método.

Tipos de Dados

Os tipos de dados são usados para descrever os valores aceitáveis para parâmetros, atributos, exceções e valores de retorno. CORBA classifica os tipos de dados em duas categorias: Básicos e Construídos. Os tipos básicos são `short`, `long`, `unsigned long`, `unsigned short`, `float`, `double`, `char`, `boolean` e `octet`. Os tipos construídos são: `enum`, `string`, `struct`, `array`, `union`, `sequence` e `any`. O tipo `any` é utilizado em situações dinâmicas pois pode representar qualquer tipo IDL. Os *bindings* das linguagens mapeiam tipos IDL para tipos próprios da linguagem.

A figura 3 mostra um exemplo da IDL. São definidas três interfaces, *pessoa, estudante e professor* e um módulo *pessoas_da_escola* que contém as

```
interface pessoa
{
   string nome;
   short identidade;
   long data_nascimento;

   void add_pessoa(in string nome, in short id, in long data)
        raises (JaCadastrado);
}

module pessoas_da_escola
{
  interface estudante:pessoa
  {
     short matricula;
     string curso;

     void add_estudante(in string nome, in short id, in long datanasc,
          in short mat, in string cur)
          raises (JaCadastrado);
     string consulta_curso(in short mat);
  }
  interface professor:pessoa
  {
     string disciplina;

     void add_prof(in string nome, in short id, in long datanasc,
          in string disc)
          raises (JaCadastrado)
  }
}
```

Figura 3: Exemplo de Interface IDL

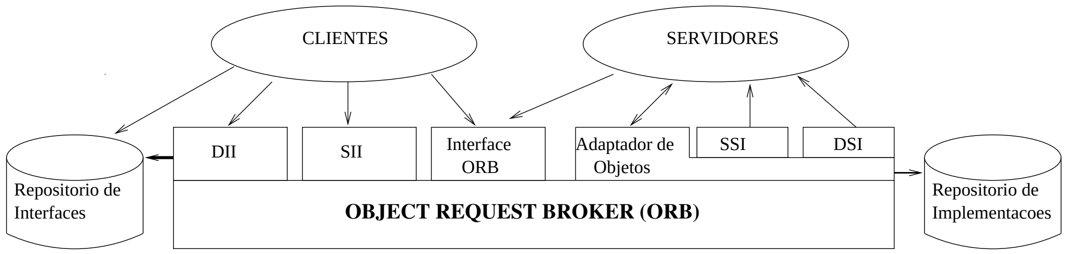

Figura 4: Arquitetura CORBA

interfaces *estudante* e *professor*. Estas interfaces são derivadas da interface *pessoa* e, internamente, chamam o método `add_pessoa` definido na interface *pessoa*, que retorna uma exceção quando já existe o registro da pessoa cujo cadastramento está sendo solicitado. Além dos atributos herdados da interface *pessoa*, as interfaces *estudante* e *professor* possuem seus próprios atributos.

A partir da IDL um precompilador gera arquivos cabecalhos (*header*) do cliente e esqueletos de implementação para o servidor.

5 ORB - Object Request Broker

A arquitetura CORBA, ilustrada na figura 4, compreende um conjunto de blocos funcionais que usam o suporte de comunicação do ORB.

O *ORB* é a parte do modelo CORBA que estabelece o relacionamento cliente/servidor entre os componentes, realizando as tarefas de comunicação e coordenação para que os clientes não se envolvam com as complexidades da comunicação com objetos remotos, nem com aspectos de implementação ou informações sobre a execução, como por exemplo os detalhes da máquina na qual determinado objeto será executado. O ORB intercepta as chamadas dos clientes direcionando-as para o servidor apropriado. A finalidade do CORBA é manter o lado do cliente o mais simples possível. Para isto, aspectos relativos ao gerenciamento de objetos são de responsabilidade do ORB. A visão do cliente é de instâncias de objetos ativos sempre prontos para receber solicitações. Para garantir esta transparência para o cliente, o CORBA nem mesmo oferece operações para o cliente ativar uma instância de objeto (há uma operação para criar uma instância chamando um objeto *factory*), o cliente apenas envia uma solicitação e o ORB realiza as demais tarefas para atender a solicitação.

6 Repositório de Interfaces

O *Repositório de Interfaces (RI)* armazena todas as definições IDL dos objetos CORBA disponíveis para clientes. CORBA especifica como as informações são organizadas e como são recuperadas do RI. Um conjunto de interfaces (*ModuleDef, InterfaceDef, OperationDef, AttibutDef, ExceptionDef, ParameterDef, TypeDef, ConstantDef*) oferecem acesso aos objetos armazenados.Para cada estrutura IDL existe uma interface que oferece informação sobre a estrutura. Com o suporte deste serviço, é possível verificar assinaturas dos métodos e obter a descrição dos objetos CORBA. Estas características tornam possível a invocação dinâmica de métodos de objetos cuja interface estão publicadas no RI. O *RepositoryID* identifica globalmente os elementos do RI (módulos, as interfaces, constantes, etc). Por exemplo, o RepositoryID da interface *professor* no módulo *pessoas_da_escola* poderia ser: "IDL:Escola/pessoas_da_escola/professor/:1.0"

Os seguintes métodos são usados para leitura e navegação no RI:

- *Describe*: retorna uma estrutura que descreve o objeto
- *Lookup*: retorna ponteiros para os objetos
- *Lookup_Name*: localiza um objeto por nome
- *Contents*: retorna uma lista de objetos que fazem parte de uma hierarquia.
- *Lookup_id*: recebe a identificação e retorna o objeto *InterfaceDef*
- *Describe_interface*: método do objeto *InterfaceDef* que retorna a estrutura que descreve a interface (nome, RepositoryID, número de versão, operações, atributos e todas as interfaces pai).
- *Is_a*: método do objeto *InterfaceDef* que retorna TRUE se a interface é idêntica à especificada no parâmetro deste método.

A chamada `get_interface`, do ORB, recebe uma referência de objeto e retorna o objeto *InterfaceDef*.

7 Interface de Invocação Estática e Interface de Invocação Dinâmica

Solicitações à métodos em objetos servidores podem ser elaboradas de duas maneiras com igual poder expressivo: estaticamente usando a *Interface de Invocação Estática (SII)* ou dinamicamente via a *Interface de Invocação Dinâmica (DII)*. Em relação à chamada estática, as especificações escritas em IDL são compiladas por um compilador IDL que gera stubs do cliente (simplesmente stubs) e stubs do servidor (esqueletos de programas). Stubs correspondem as interfaces estáticas através das quais um cliente chama métodos remotos da mesma maneira que faz chamadas locais. O stub age como uma interface para o ORB que irá rotear a chamada para o servidor destino. Para que isto seja possível é necessário que cada cliente conheça o stub IDL para cada interface usada. O problema desta abordagem é que cria uma ligação estática entre programas clientes e as interfaces IDL dos objetos servidores, com isto, quando as interfaces evoluem, é necessário rever todos os programas clientes e seus servidores, incluir as modificações e proceder novamente as etapas de compilação.

A DII serve como uma interface genérica que não necessita de stubs, ao invés disto, ela permite a invocação de métodos em tempo de execução. Para se construir a invocação dinâmica deve-se seguir tais passos [1]:

- descobrir os novos objetos

  O CORBA especifica duas maneiras pelas quais um objeto pode localizar outro objeto no sistema. O primeiro é o *Serviço de Nomes*. Este serviço é análogo à uma lista telefônica: um objeto procura outro objeto pelo nome com o qual este se registrou no ORB. O serviço retorna a referência do objeto. Este método de busca tem o inconveniente do cliente ter de conhecer o nome exato com o qual o objeto foi registrado. O segundo serviço é o *Serviço de Trading* que é semelhante ao serviço de Páginas Amarelas onde objetos podem perguntar ao serviço quais são os objetos registrados que possuem certas características. O repositório de "trading" retorna as referências para os objetos que atenderam as características especificadas pelo cliente, além das informações apropriadas sobre os serviços do referido objeto.

- descobrir suas interfaces

usando a referência obtida no passo anterior, o cliente chama a operação *get_interface* para obter a descrição da interface do objeto.

- obter a completa definição do método

  a chamada *describe* retorna a estrutura que descreve o método

- criar uma lista de argumentos

  a estrutura de dados *NamedValue list* é usada para definição da estrutura de dados para passagem de parâmetros. Esta lista é criada usando a operação *create_list* e chamadas sucessivas a operação *add_arg* para adicionar cada elemento na lista.

- criar a solicitação

  na solicitação deve-se especificar a referência ao objeto, o nome do método chamado e a lista de argumentos. Cria-se a solicitação usando a chamada *create_request*.

- fazer a chamada

Modos de Invocação

CORBA permite chamar as operações segundo três modos:

- chamada síncrona e bloqueante até o retorno do resultado (método *invoke*)
- chamada assíncrona (método *send-oneway*)
- chamada com espera deferida (método *send-deferred*): o cliente envia a solicitação e retorna o controle para o programa. Para obter a resposta o programa deve chamar o método *get_response* ou *poll_response*

Invocações estáticas são em geral síncronas (a menos que sejam declaradas como *Oneway* na sua definição IDL) enquanto que invocações dinâmicas podem ser realizadas em qualquer modo.

8 Interface de Esqueleto Estático e Interface de Esqueleto Dinâmico

Para o servidor é transparente se a invocação ao método é estática ou dinâmica. Antes de transmitir a solicitação ao objeto destino, o ORB prepara solicitações dinâmicas de tal forma que elas têm exatamente a mesma forma que solicitações estáticas.

Similarmente ao stub do cliente, o servidor possui os esqueletos de programa que denotam a *Interface de Esqueleto Estático (SSI)* para o componente servidor. A SSI desempacota as solicitações para transmiti-las às implementações dos objetos e, ao receber os resultados, empacota-os para envia-los ao cliente. Para tratar componentes sem stubs IDL, o CORBA oferece a *Interface de Esqueleto Dinâmico (DSI)* que liga chamadas recebidas para métodos de tais objetos em tempo de execução. A DSI consulta os valores dos parâmetros que vêm na mensagem para obter o objeto e o método destino da invocação. O ORB usa a DSI para criar um *proxy* de esqueleto dinâmico para passar a chamada para objetos, tipicamente remotos, cujos esqueletos estáticos não são ligados a ele. Com este recurso o ORB pode enviar solicitações para qualquer implementação de objeto. A DSI é implementada via uma *Rotina de Implementação Dinâmica* ou *DIR*. O ORB chama a DIR para cada solicitação DSI que realiza. Assim, para ser acessível via a DSI a implementação do objeto deve utilizar a DIR explicitamente. Devido a maneira como a DSI funciona, todas as operações e interfaces são tratadas por uma única interface ao invés de um conjunto de stubs.

9 Adaptador de Objetos

O *Adaptador de Objetos* é um componente chave para a portabilidade da implementação dos objetos. As suas funções são: registrar componentes no repositório de implementações, criar instâncias de novos objetos em tempo de execução, gerar e gerenciar referências de objetos, mapear referências de objetos em suas respectivas implementações, ativar e desativar implementações de objetos, informar ao ORB os servidores que estão ativos, receber a solicitação dos clientes direcionada pelo ORB e roteá-las para o método apropriado via a DSI ou SSI.

10 Repositório de Implementações

O *Repositório de Implementações* armazena informações sobre a implementação do objeto: as classes que um servidor implementa, os objetos que estão instanciados e as referências dos objetos, disponibilizando um repositório para consulta em tempo de execução sobre uma variedade de serviços.

11 Serviço de Nomes

O serviço de Nomes realiza um mapeamento entre um Nome simbólico e uma referência de objeto. A associação Nome-Objeto é denominada *ligação de Nomes (binding name)* [1]. O serviço de Nomes define interfaces para inserção e remoção de ligações de Nomes, consulta e navegação pelo repositório de nomes.

Este serviço permite que Nomes sejam estruturados hierarquicamente em *Contextos* - estruturas similiares à diretórios dos sistemas de arquivos.

Nomes são sequências contendo dois atributos: um string *identificador* e um *tipo* que é um string que serve para qualificar o nome, ou seja, é um atributo descritivo para o nome - por exemplo, um tipo de arquivo [1]. O serviço de Nomes não interpreta este atributo, que é usado por aplicações. A figura 5 mostra a descrição do Nome utilizada pelo serviço de Nomes.

```
typedef string Istring
struct NameComponent{
   Istring id;
   Istring kind;
};

typedef sequence <NameComponent> Name;
```

Figura 5: Serviço de Nomes - Descrição do Nome

Duas interfaces, agrupadas no módulo *CosNaming*, especificam o serviço de Nomes: *NamingContext* e *BindingIterator*.

```
interface NamingContext{

void bind(in Name n, in Object obj) // liga um nome a um objeto
  raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
void rebind(in Name n, in Object obj)
  raises(NotFound, CannotProceed, InvalidName);

void bind_context(in Name n, in NamingContext nc) // liga um nome a um contexto
  raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
void rebind_context(in Name n, in Naming Context nc)
  raises(NotFound, CannotProceed, InvalidName);

// Metodos para Encontrar Objetos por Nome
Object resolve(in Name n)
  raises(NotFound, CannotProceed, InvalidName);

// Metodo para Navegacao em um Contexto de Nomes
void list (in unsigned long how_many)
  raises(NotFound, CannotProceed, InvalidName);

// Metodo para Remover Nomes de um Contexto
void unbind(in Name n)
  raises(NotFound, CannotProceed, InvalidName);

// Metodo para Destruir um Contexto vazio
void destroy()
  raises(NotEmpty);
};
```

Figura 6: Interface IDL NamingContext

A interface *NamingContext*, ilustrada na figura 6, modela o contexto de nomes e oferece métodos para fazer associações (*bind*), modificá-las (*rebind*), recuperá-las (*resolve*) e destrui-las (*unbind*). além de métodos para criação de novos contextos (*new_context* e *bind_new_context*)e consulta ao conteúdo de um contexto (*list*). A interface *BindingIterator*, descrita na figura 7, permite a navegação em um contexto de nomes através dos métodos *next_one* e *next_n*.

O serviço de Nomes impõe que o cliente conheça o nome simbólico do objeto desejado. O método `list` juntamente com os métodos oferecidos pela interface *BindingIterator*, introduzem flexibilidade à este serviço permitindo a navegação no contexto de nomes para se descobrir o objeto desejado.

```
enum BindingType {nobject, ncontext};

struct Binding{
   Name        binding_name;
   BindingType binding_type;

typedef sequence <Biding> BindingList;

interface BindingIterator{

boolean next_one(out Binding b); // retorna um binding e TRUE se existem outros                                     binding
boolean next_n(in unsigned long how_many,
               out BindingList bl);
void destroy();
};
```

Figura 7: Interface IDL BindingIterator

12 Serviço de Trading

O serviço de *Trading* é um mecanismo que facilita o anúncio e a descoberta de serviços. Enquanto o serviço de Nomes procura objetos por Nome, o serviço de *Trading* procura objetos por tipo e propriedades. *Trader* é o objeto que

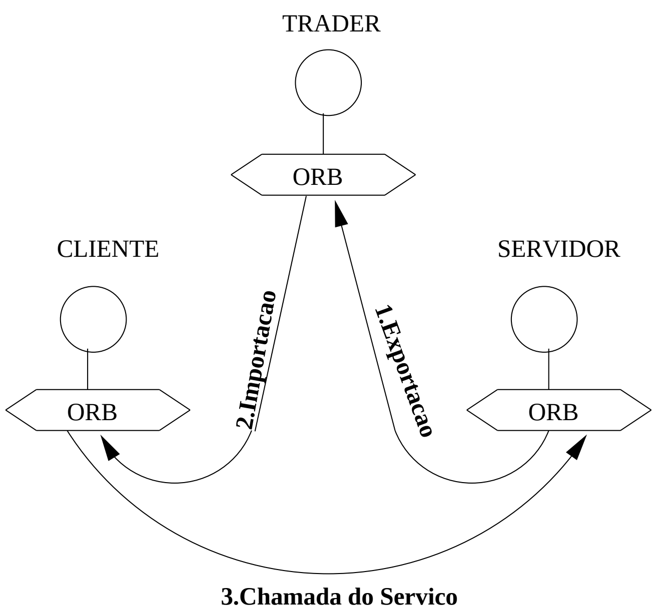

Figura 8: Serviço de Trader

mantém o repositório de referências de objetos e oferece o serviço de *Trading*. O ato de um servidor anunciar seu serviço (serviço oferecido) é chamado *exportação*. Um serviço oferecido é descrito por um *Tipo de Serviço* que consiste de um *nome*, um *tipo de interface* e um conjunto de especificações de *propriedades* do serviço. *Propriedades* são pares (*nome, valor*) que descrevem as capacidades dos serviços. Propriedades podem ser obrigatórias (*mandatory*) ou opcionais. Quando um cliente deseja um serviço, ele envia uma solicitação de serviço, chamada de *importação*, ao objeto *Trader*. Esta solicitação expressa as características. O *Trader* busca em seu banco de dados o serviço especificado pelo cliente. A lista dos serviços encontrados é retornada para o cliente que pode escolher qual servidor utilizar. A figura 8 ilustra a comunicação entre clientes, servidores e o serviço de *Trading*.

Tipos de Serviços são armazenados na interface *ServiceTypeRepository*. A figura 9 mostra os tipos de dados que especificam os serviços oferecidos publicados no *Trader*.

No serviço de *Trading* duas interfaces oferecem funções para importação e exportação de serviços (Figura 10).

- *Lookup*: usada por clientes para importação de serviços. Oferece um

```
typedef sequence <CosTrading::ServiceTypeName> ServiceTypeNameSeq;

enum PropertyMode{
   PROP_NORMAL, PROP_READONLY,
   PROP_MANDATORY, PROP_MANDATORY_READONLY
};

struct PropStruct{
   CosTrading::PropertyName name;
   CORBA::TypeCode value_type;
   PropertyMode mode;
};

typedef sequence <PropStruct> PropStructSeq;

typedef CosTrading::Istring Identifier;  // IR::Identifier
```

Figura 9: Serviço de Trading - Descrição dos Serviços

método `query()` que recebe uma especificação do tipo de serviço, procura no trader serviços publicados que satisfaçam as especificações e retorna uma lista de serviços oferecidos.

- *Register*: usada por servidores para exportar serviços (`export()`), para mudar os valores de propriedades do serviço oferecido (`modify()`), para remover um serviço oferecido (`withdraw()`).

A figura 10 ilustra os métodos `query()` e `export()`. No método `query()`, o parâmetro `desiredprops` especifica se o retorno deve ser o conjunto de propriedades que satisfazem a solicitação ou a referência dos objetos que satisfazem as propriedades estabelecidas nos parâmetros. O tipo `SpecifiedProps` é definido como:

```
enum HowManyProps{none, some, all};
```

```
union SpecifiedProps switch(HowManyProps){
   case some:PropertyNameSeq prop_names;
};
```

Um tipo de serviço pode conter propriedades que não interessam a um cliente. Neste caso o cliente especifica no campo `prop_names` qual propriedade irá retornar.

```
void query(
   in ServiceTypeName type,
   in Constraint constr,
   in Preference pref,
   in PolicySeq policies,
   in SpecifiedProps desired_props,
   in unsigned long how_many,
   out OfferSeq offers,
   out OfferIterator offer_itr,
   out PolicyNameSeq limits_applied
) raises (
   IllegalServiceType,
   UnknownServiceType,
   ...
);

OfferId export(
   in Object reference,
   in ServiceTypeName type,
   in PropertySeq properties
) raises (
   InvalidObjectRef,
   IllegalServiceType,
   ...
);
```

Figura 10: Serviço de Trading - Importação e Exportação de Serviços

13 Serviço de Eventos

Em CORBA o modelo de comunicação entre objetos caracteriza-se por sempre haver um objeto destino que é único e caso este objeto não está disponível, a comunicação falha. Em algumas aplicações é necessário um tipo de comunicação mais desacoplada que não se adequa a este modelo restritivo. Para atendê-las, CORBA oferece o serviço de eventos [4] que permite objetos dinamicamente registrar e desregistrar seu interesse em eventos específicos e receber notificação quando um evento acontece. O serviço de eventos é descrito por um conjunto de interfaces que define operações para manipular eventos. Um *evento* é uma ocorrência em um objeto especificada como de interesse de um ou mais objetos[1]. Uma *notificação* é uma mensagem que um objeto envia para partes interessadas informando a elas que um evento específico ocorreu. O serviço de eventos oferece suporte a estes conceitos criando um canal de comunicação entre objetos que não se conhecem. São definidos dois papéis para os objetos: *Fornecedores* produzem eventos e *Consumidores* processam-os. A comunicação de eventos entre fornecedores e consumidores acontece via um canal de eventos e segue um entre dois modelos: *Push* e *Pull*.

No modelo Push o fornecedor invoca o método `push` no objeto canal de eventos que por sua vez direciona o evento para os objetos consumidores. Para registrar no canal de eventos seu interesse em algum evento, o consumidor chama o método `add_push_consumer`. Para parar de receber eventos o consumidor chama o método `disconnect_push_consumer`. O fornecedor invoca o método `disconnect_supplier` para se desconectar do canal.

No modelo *Pull* o consumidor invoca o método `pull` no canal de eventos que solicita o evento do fornecedor. Para parar de aceitar solicitações para fornecer eventos, um fornecedor chama o método `disconnect_pull_supplier` no canal de eventos. Para registrar sua referência e oferecer seus serviços o fornecedor chama o método `add_pull_supplier`. O consumidor pode usar dois métodos para verificar o recebimento de um evento: `pull` e `try_pull`. O método `pull` bloqueia até que um evento seja recebido enquanto que o `try_pull` retorna imediatamente um valor booleano indicando se algum evento foi retornado ou não.

A diferença entre os modelos *Push* e *Pull* está no elemento que inicia a requisição dos eventos, o fonecedor no caso do modelo *Push*, e o consumidor no modelo *Pull*. Enquadrando estes modelos na comunicação cliente-servidor,

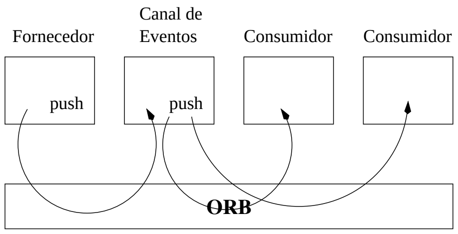

Figura 11: Estilo Push

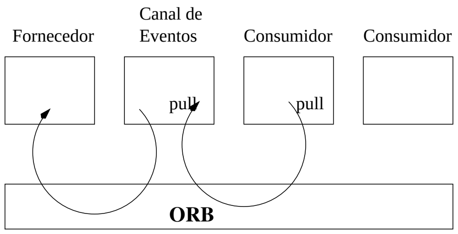

Figura 12: Estilo Pull

no *Push* o fornecedor faz o papel de cliente pois solicita eventos do consumidor que faz o papel do servidor. No modelo *Pull* o consumidor é o cliente pois requisita eventos do fornecedor.

O canal de eventos é um objeto CORBA que viabiliza a interação indireta assíncrona entre fornecedores e consumidores. Como um objeto CORBA, o canal de eventos é beneficiado por todas as propriedades que se aplicam aos demais objetos. Desta forma, a comunicação dos objetos com o canal de eventos é feita usando solicitações comuns CORBA e a localização de canais de eventos pode ser feita da mesma maneira que se localiza os demais objetos. Opcionalmente, a aplicação pode estabelecer políticas em relação a criação dos canais e a sua localização. Por exemplo, referências para os canais podem ser fornecidas como parte de um protocolo específico (pode ser como retorno da operação "open" em um objeto). Canais de eventos podem ser genéricos ou tipados. O canal genérico não conhece o conteúdo dos dados que transmite, os parâmetros são do tipo ANY. Neste caso os fornecedores e consumidores devem concordar sobre a semântica dos eventos. O canal tipado permite aplicações descreverem os conteúdos dos eventos usando a IDL, os parâmetros são de tipos definidos no CORBA.

Referências Bibliográficas

[1] R. Orfali and D. Harkey and J. Edward. The Essential Distributed Objects Survival Guide. John Wiley & Sons. 1996.

[2] J. Siegel. CORBA: Fundamentals and Programming. John Wiley & Sons. 1996.

[3] J. Siegel. OMG Overview: CORBA and the OMA in Enterprise Computing. Communications of the ACM. Vol. 41, Number 10, pp 37-43, October 1998.

[4] OMG. The Common Object Broker Architecture and Specification. Technical Report Revision 2.2, OMG, 1998.