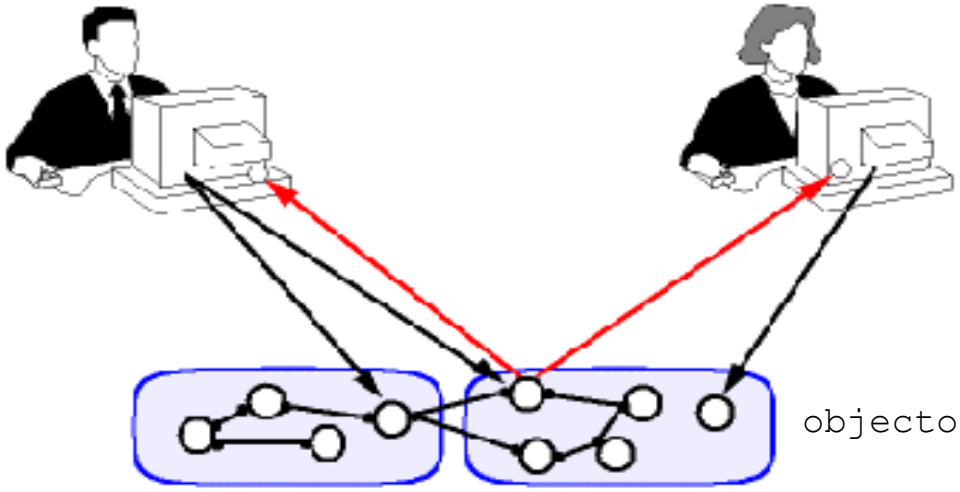
Sistemas distribuídos e Middleware

- Introdução aos sistemas distribuídos e middleware
- Arquitectura CORBA
- Arquitectura DCOM
- Arquitectura Java RMI
- Arquitectura JINI

Introdução aos Sistemas Distribuídos

Sistema Distribuído: Conjunto de computadores ligados em rede, com software que permita a partilha de recursos e a coordenação de actividades, oferecendo idealmente um ambiente integrado.



As aplicações são distribuídas por natureza devido a uma ou mais das seguintes razões:

- Os dados usados pela aplicação são distribuídos
- A computação é distribuída
- Os utilizadores da aplicação estão distribuídos

Caracterização dos Sistemas Distribuídos

- Conjuntos de elementos de processamento independentes, que comunicam unicamente via mensagens.
- Existe a possibilidade de falhas independentes, quer dos elementos de processamento, quer dos canais de comunicação (perda, duplicação ou reordenação de mensagens).

Exemplos de aplicações "sérias":

- Reserva de bilhetes em companhias de aviação.
- Máquinas Multibanco.

Estas aplicações exigem fiabilidade e segurança, mesmo em presença de falhas de comunicação e de computadores, mas existem outro requisitos...

Características desejáveis de um SD

Partilha de recursos

- Hardware: impressoras, discos.
- Dados: ferramentas para trabalho cooperativo, bases de dados.
- Gestores de recursos: oferecem interface para aceder ao recurso.
- Motiva modelo cliente-servidor

Abertura

- Extensibilidade em software e hardware, com possível heterogeneidade.
- Obtida especificando e documentando interfaces.
- Possuindo um mecanismo uniforme de comunicação entre processos.

Características desejáveis de um SD

Concorrência

- Existe e deve ser suportada em SD's por vários motivos:
  - Vários utilizadores podem invocar comandos simultaneamente.
  - Um servidor deve poder responder concorrentemente a vários pedidos que lhe cheguem.
  - Vários servidores correm concorrentemente, possivelmente colaborando na resolução de pedidos.

Escalabilidade

- O software deve ser pensado de modo a funcionar em grandes sistemas sem necessidade de mudança.
- Devem ser evitados algoritmos e estruturas de dados centralizadas.
- Soluções devem tomar em conta a replicação de dados, caching, replicação de serviços, algoritmos distribuídos

Características desejáveis de um SD

Tolerância a faltas

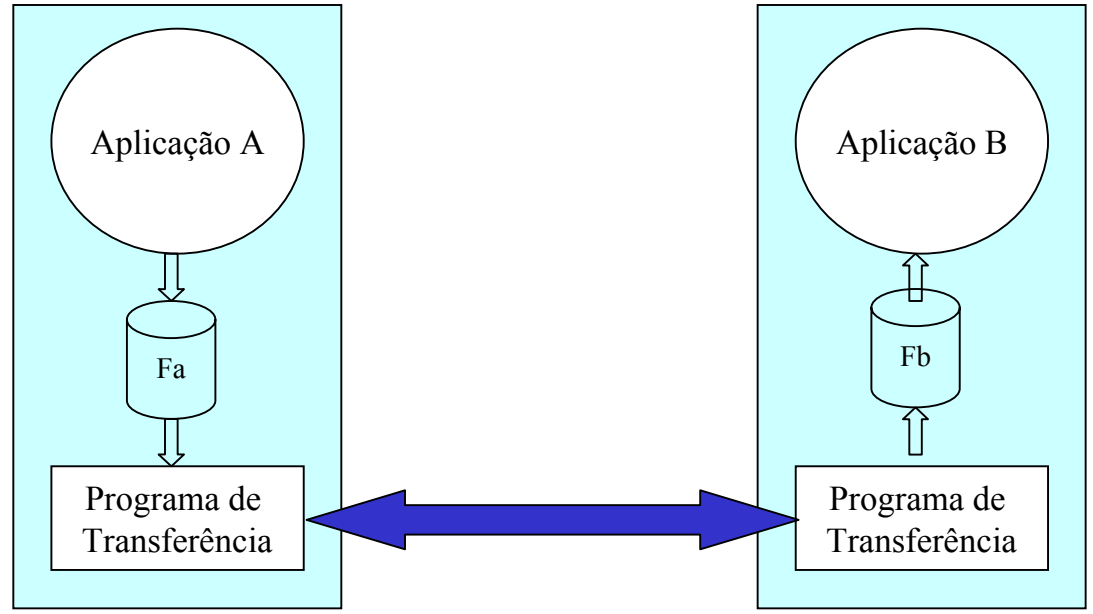
- Faltas em elementos de processamento ou de comunicação.
- Soluções podem passar por redundância de hardware e replicação de servidores/serviços.
- Motiva paradigmas mais avançados do que o cliente-servidor; exemplos: comunicação em grupo, algoritmos de acordo, transacções.
- Em sistemas distribuídos a disponibilidade perante faltas pode ser maior do que em sistemas centralizados, mas exige maior complexidade do software.

Transparência

- O sistema ser visto como um todo e não como uma colecção de componentes distribuídos
- Tipos de transparência: acesso, localização, concorrência, replicação, falhas, migração, desempenho, escalabilidade
- Mais importantes: acesso e localização (juntos referidos como transparência de rede)

Necessidade do Middleware

- Colocam-se, ao nível de software distribuído, três problemas aos gestores do processamento de dados de uma empresa, a saber:
  - A integração de software proveniente de origens destinas;
  - O acesso ao software dentro e fora da empresa;
  - O desenvolvimento rápido de aplicações.

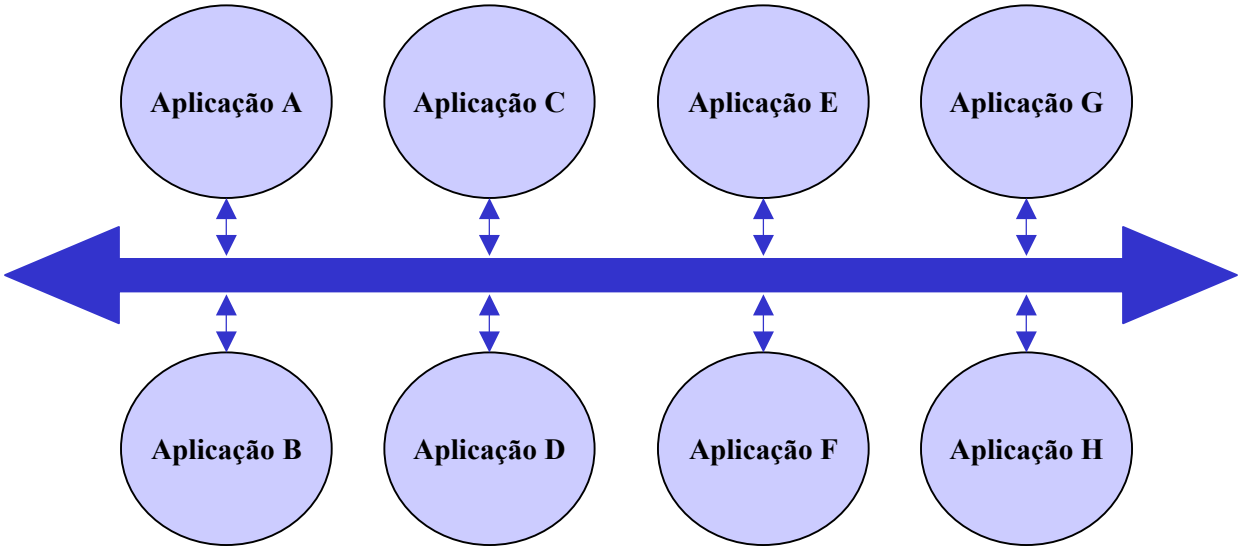


Interface Tradicional

Necessidade do Middleware

O objectivo principal do "Middleware" é o de resolver o problema da integração de software.

- Um sistema de informação constituído por n aplicações pode conduzir, no limite, à criação de n ligações e por sua vez 2n interfaces com aplicações.
- Um processo de resolver este problema é introduzir o conceito de um único "bus" de comunicação designado por "Middleware" ao qual se ligam as aplicações através de uma interface bem definida
- Todas as interligações são efectuadas por um único sistema comum. Nesta arquitectura o "bus" de comunicação tem que oferecer um certo número de funcionalidades capazes de suportar a interligação de duas ou mais aplicações.



Middleware. O que ê?

Termo que se aplica a uma camada de software que proporciona uma abstracção de programação assim como esconde a heterogeneidade da rede, hardware, sistema operativo e linguagens de programação usadas.

Características do Middleware

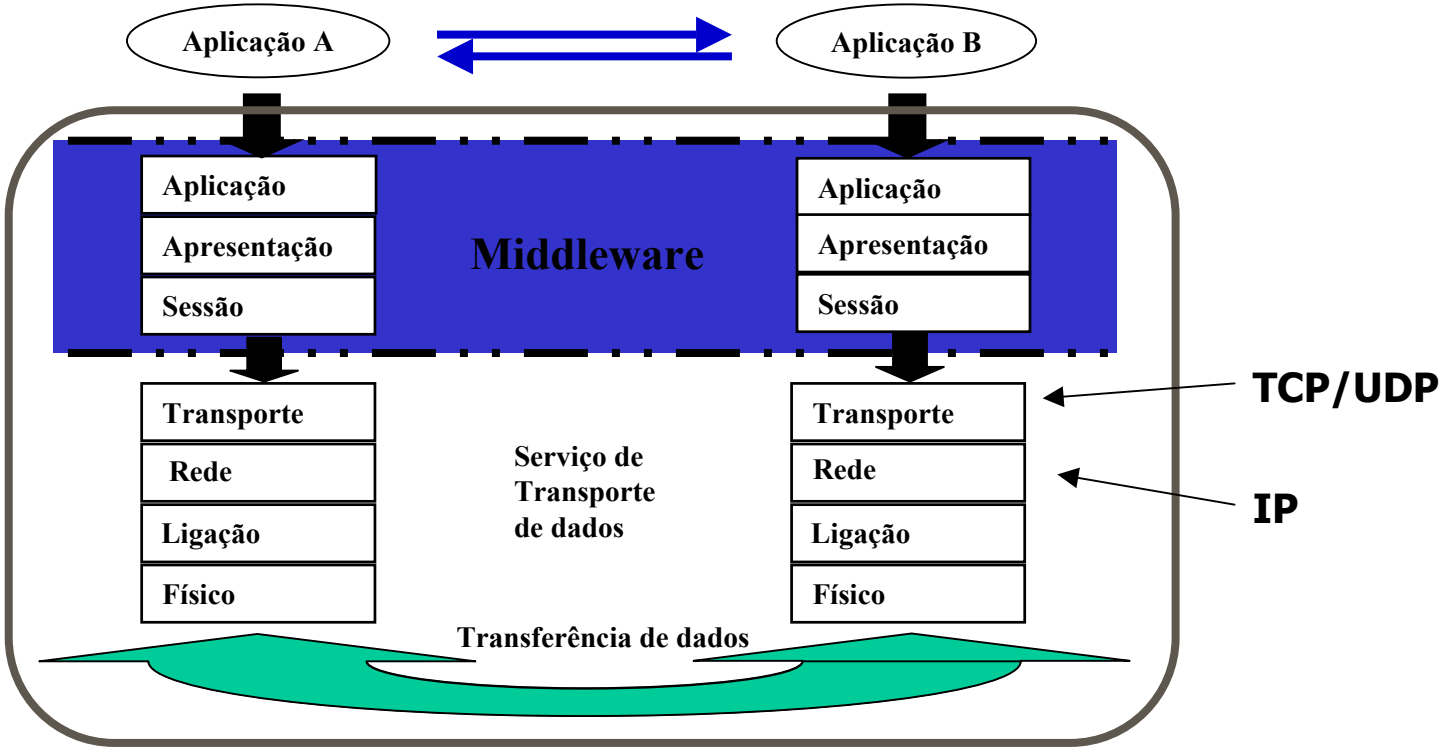
- O "Middleware" tem que estar disponível em diversas máquinas
- As transferências tem que ser fiáveis.
  - Tem que existir a garantia de que quando uma aplicação entrega uma mensagem ao "Middleware" que o destino recebe a mensagem uma única vez. Isto tem que acontecer mesmo que um computador ou a rede falhe.
- Adaptação ao tráfego.
  - A largura de banda do "bus" tem que suportar um aumento de tráfego resultante do aumento do número de aplicações.
  - Esta é uma característica de base muito importante já que o "Middleware" é o esqueleto de qualquer aplicação.

Características do Middleware

- A diversidade das estruturas de comunicação.
  - Uma aplicação pode comunicar com uma outra aplicação ou enviar uma única mensagem para n destinos.
    - Neste último caso é desejável que o emissor entregue apenas uma única cópia da mensagem ao "bus" e fique a cargo deste a responsabilidade de enviar para todos os destinos.
- A utilização de um nome.
  - A aplicação que envia uma mensagem refere-se ao destino através de um nome e não de um endereço físico.
  - Esta característica permite a transferencia de uma aplicação de um computador para outro sem haver implicações nas aplicações que com ela comunicam.
- O conceito de transacção.
  - Este conceito especifica que várias entidades, por exemplo aplicações, pertencentes a uma única transacção ou podem todas executar as suas tarefas ou nenhuma o faz.

Localização do "Middleware" no modelo OSI

- O "Middleware" forma uma estrutura de comunicação independente dos sistemas operativos e da natureza da rede de comunicação usadas.
- O "Middleware" situa-se logo abaixo das aplicações. No modelo de camadas OSI corresponderá as camadas superiores definindo o protocolo a usar entre as aplicações
- O "Middleware será suportado pelos protocolos das camadas inferiores (TCP/IP, DECnet, SNA) e por mecanismos dos próprios sistemas operativos das máquinas onde são executadas as aplicações.



Soluções de Middleware

Existem três grandes soluções para o "Middleware" conforme a unidade elementar a distribuir :

- O message-based Middleware
  - Unidade elementar a distribuir é a própria aplicação.
  - implementa o conceito de "bus" de comunicação comum a várias aplicações permitindo a troca de mensagens entre elas.
  - integração de aplicações já existentes
- O Remote-Procedure-Call Middleware.
  - o procedimento é a unidade elementar a distribuir
- "Middleware" orientada por objectos
  - O objecto é a unidade elementar distribuída
  - permite distribuir logo na fase de modelação.
  - implementações CORBA e COM.

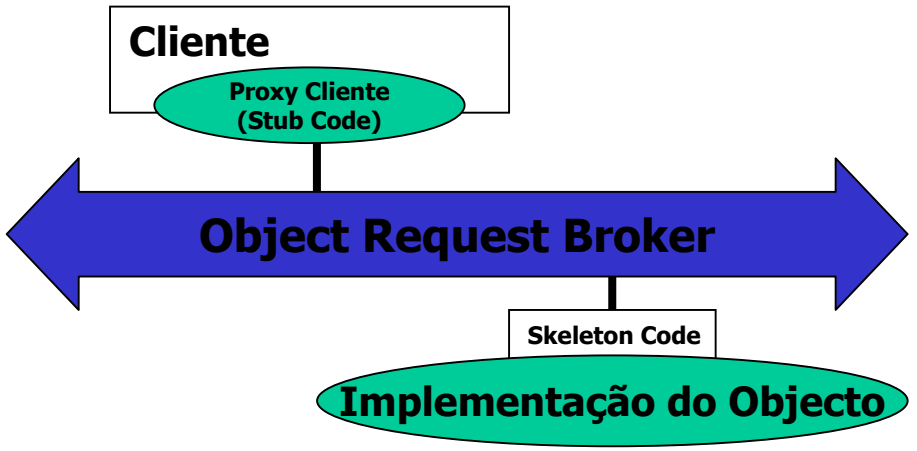
Sistemas distribuídos e Middleware

- Introdução aos sistemas distribuídos e middleware
- Arquitectura CORBA
  - Introdução. Arquitecturas
  - O IDL – Interface Definition Language
  - Interfaces ORB
  - Objectos factory e callback.
  - Interfaces POA
  - Serviços de nomes
  - Serviços de eventos
  - Modelo de concorrência
  - Activação dinâmica de servants e Implementation Repository
  - Interface Repository, DII e DSI
- Arquitectura DCOM
- Arquitectura Java RMI
- Arquitectura JINI

CORBA - Common Object Request Broker Architecture

- arquitectura normalizada para a concepção de aplicações distribuídas orientadas por objectos.
- foi criada pelo consórcio OMG – "*Object Management Group*",
  - mais de 700 membros, incluindo empresas de desenvolvimento de plataformas, de bases de dados, de aplicações e outro software diverso.
  - definir orientações e especificações para o desenvolvimento de aplicações orientadas por objectos, nomeadamente arquitecturas standard para ambientes distribuídos caracterizados pela interoperabilidade entre redes e sistemas operativos heterogéneos.

A arquitectura CORBA é **independente da plataforma** e da **linguagem** orientada por objectos utilizadas.



CORBA e os sistemas distribuídos

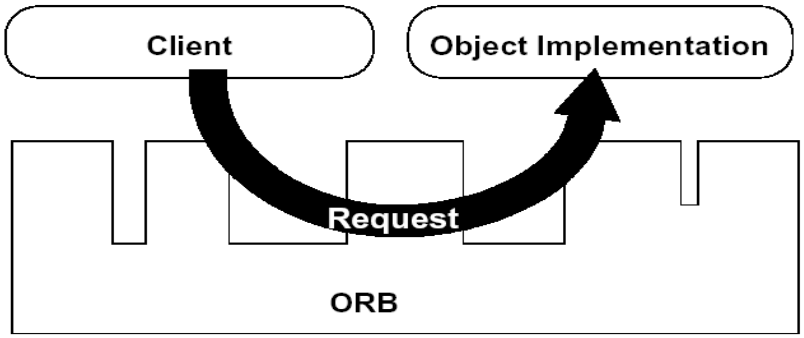
- Os objectos distribuídos CORBA são caracterizados pela sua capacidade de interacção numa mesma máquina, em rede local, e numa rede alargada
- A arquitectura CORBA permite
  - separação de vários níveis de uma aplicação distribuída - dados, tratamento de dados e interface gráfico.
  - A interacção entre estes diferentes níveis é efectuada através do interface dos *objectos distribuídos CORBA*.
- A lista de interfaces dos objectos de um SI representam os serviços oferecidos por esse sistema.
  - É possível em qualquer instante consultar essa base de dados para saber as capacidades do sistema. Adicionar novo software implica adicionar novas interfaces descrevendo os novos serviços oferecidos.
- O CORBA descreve uma arquitectura que especifica como processar objectos distribuídos numa rede.

ORB – Object Request Broker

Nesta arquitectura existe um agente intermediário designado por:

Object Request Broker

- O ORB está entre o cliente e a concretização de um objecto
- Permite que o cliente evoque operações no objecto
- Responsável por mecanismos como: encontrar a concretização do objecto, tratar da execução do código servidor;
- O cliente (que pode ou não ser um objecto) vê uma interface independente da linguagem de concretização do objecto (que pode não ser OO)

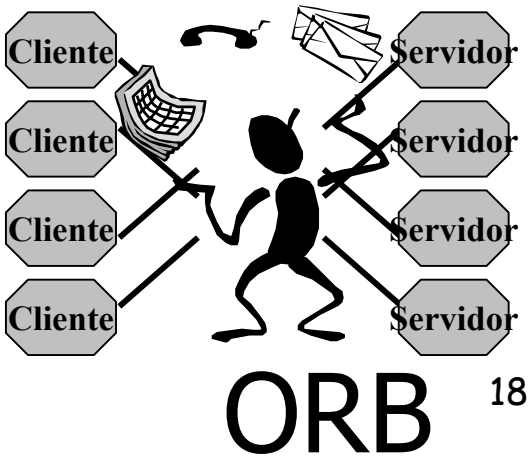


ORB – Object Request Broker

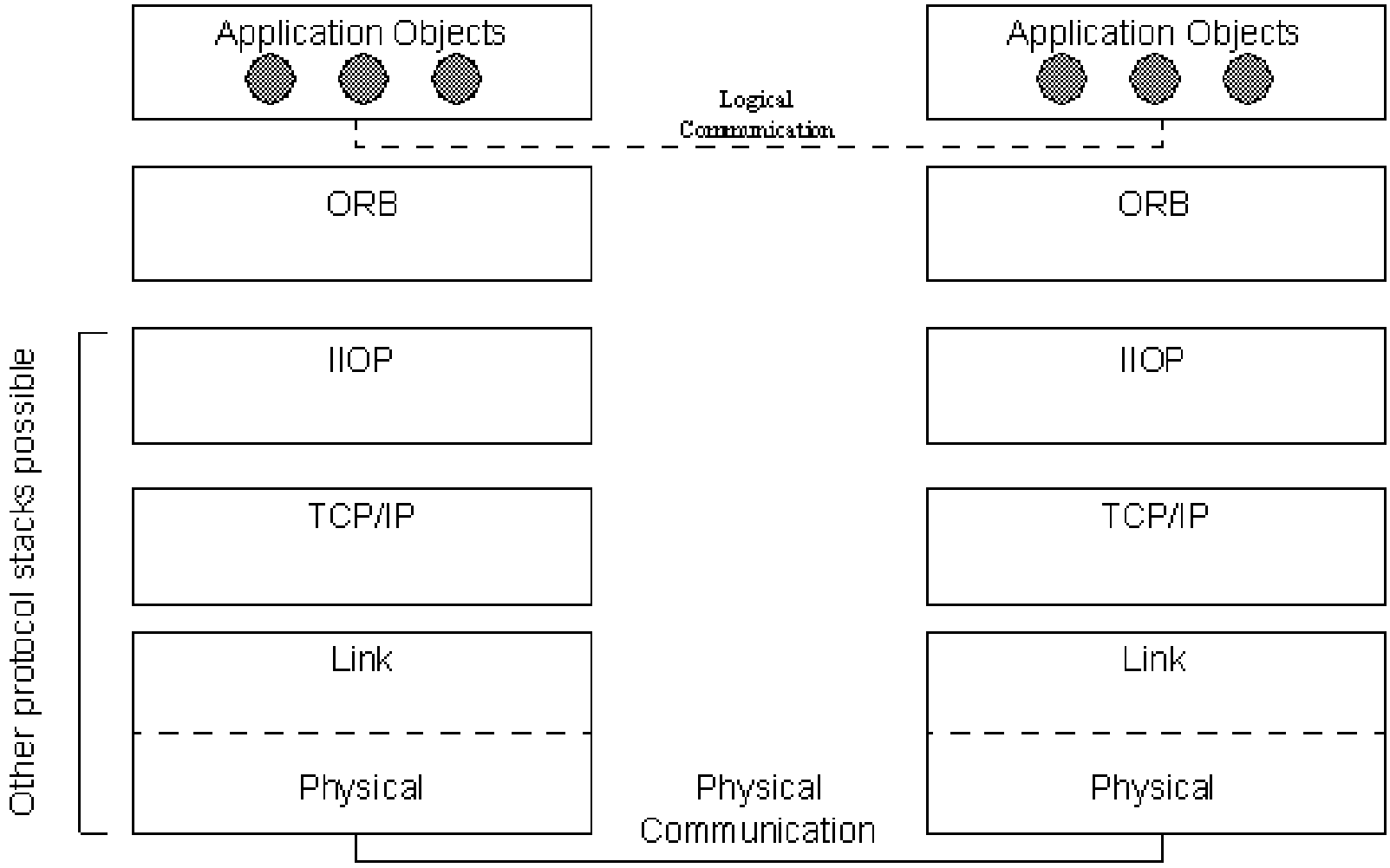
- O papel do "Broker" é de identificar um servidor capaz de executar o serviço pretendido e transmitir-lhe o pedido.
- Neste modelo só o "Broker" conhece os endereços do cliente e do servidor, podendo cliente e servidor correr na mesma máquina ou em máquinas distintas interligadas por uma rede local ou uma WAN.
- Um pedido enviado por um cliente é recebido pelo "Broker" que o envia para o servidor.
- A existência do "Broker" permite isolar os clientes dos servidores.
  - Uma aplicação, cliente, interage com outra aplicação, servidor, sem conhecer o seu endereço nem como ela executa a sua tarefa.

O ORB não é um único componente fisicamente num único local, mas definido pelas interfaces que fornece

Existem diferentes concretizações de ORB, que podem usar mecanismos diferentes para efectar evocações



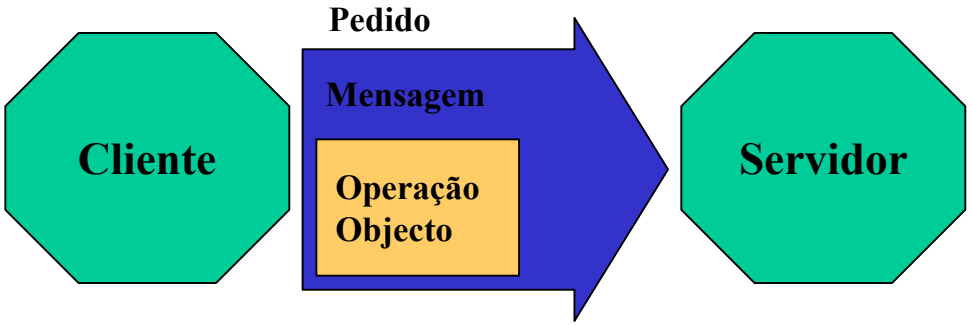
Arquitectura protocolar do CORBA



O modelo CORBA

- O CORBA separa o modo como algo é pedido da forma de como é executado
  - Um cliente CORBA apenas sabe como pedir para efectuar uma dada acção e o servidor sabe apenas como executar a tarefa pedida
  - Isto permite que se modifique como o servidor executa um pedido sem alterar o modo como o cliente o pede.
  - Uma mensagem (resultado do marshalling de uma evocação) contém o nome de uma operação e o nome do objecto sobre o qual irá ser executada a operação. Uma interacção CORBA é baseada no envio de um pedido numa mensagem.

Para isso...



O modelo CORBA

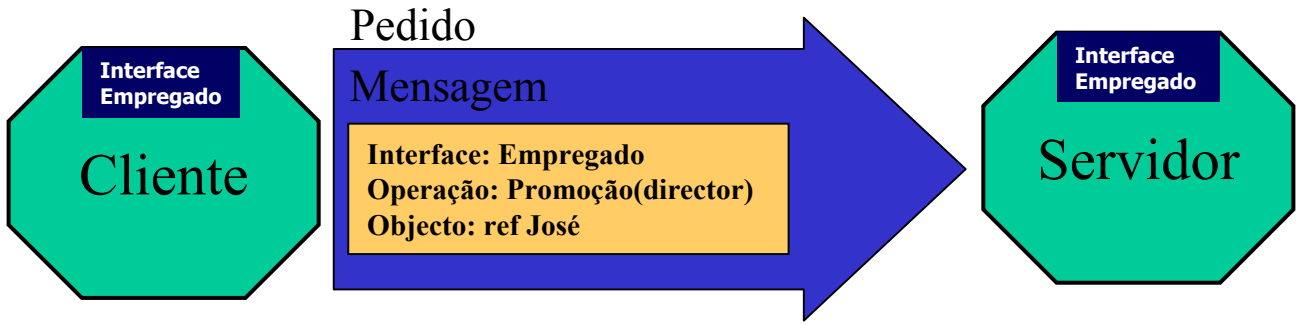
Existe um interface:

- A interface descreve os possíveis pedidos a enviar por um cliente a um servidor.
- A interface representa um contracto entre um cliente e um servidor.
  - Este contrato define as operações que um cliente pode pedir a um servidor para executar sobre um objecto
  - a interface é orientada ao Objecto porque descreve as operações que podem ser efectuadas sobre o objecto
  - A interface é escrita em IDL (Interface Description Language) [OMG IDL].

Interface – o contracto

Como a interface representa um contracto entre cliente e servidor ambos recebem uma cópia deste para justificarem as suas acções.

Neste exemplo, o cliente possui uma cópia da interface empregado e pode pedir para o promover ou demitir numa instancia do objecto tipo empregado. O servidor que oferece a interface empregado sabe como executar as operações promover ou demitir sobre qualquer instancia do tipo empregado...



Uma instancia de um objecto é uma implementação em tempo de execução dum elemento de uma dada classe. José e João podem ser duas instancias da classe empregado. Uma instancia objecto em CORBA é definida por uma referencia. Se um cliente pedir a promoção da instancia José terá que especificar a referencia para José, o nome da interface a usar e o nome da operação a executar conjuntamente com os parâmetros.

O Interface - IDL (Interface Definition Language)

- O IDL descreve a interface de um sistema de objectos distribuídos
- Um sistema de objectos surge como um conjunto de objectos isolando o cliente do objecto remoto.
- Permite ter independência da linguagem
- Permite
  - gerar stubs e esqueletos através de um compilador
  - ser colocada num repositório de interfaces

```
Interface Empregado {
  void promocao(
    in char novo_posto
    );
  void demitir(
    in Codigodemissao razao,
    in string descricao
    );
  };
```

exemplo duma interface escrita em IDL, dois métodos são definidos, promoção e demissão.

O propósito de uma interface é o de definir
- Acções que são possíveis executar sobre um objecto.
- Atributos.
- Etc

Dois tipos de operação estão automaticamente associados com cada atributo:
- *1- operação de leitura*
- *2- operação de escrita, caso não seja declarado "read_only".*

Diferentes formas de evocação

O Cliente:

- Inicia um pedido de uma das seguintes formas:
  1. Invocando um código num Stub IDL, apropriado para a interface do objecto
  2. Construindo o pedido dinamicamente, através da Dynamic Invocation Interface, que pode ser feita em run-time, sem conhecimento prévio da interface
- A concretização do objecto desconhece qual destas formas foi usada para efectuar o pedido

O Servidor:

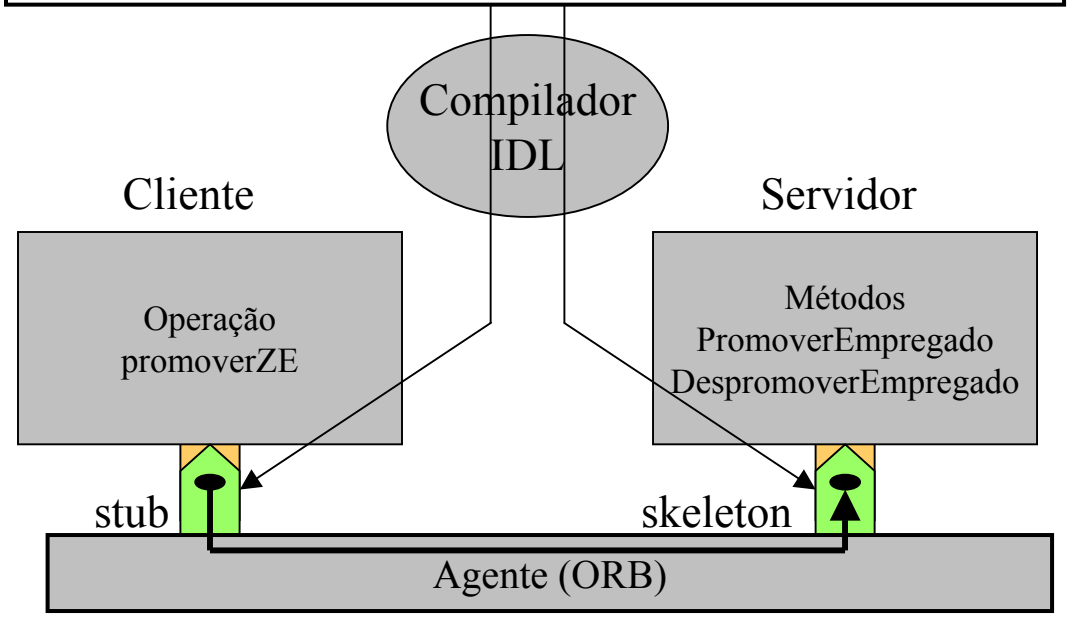
- Aqui o ORB transfere o controlo para o código de concretização do objecto de uma das duas formas:
  1. Skeleton IDL, apropriado para a interface do objecto
  2. Skeleton dinâmico, quando não se conhece a interface a quando do desenvolvimento
- A concretização do objecto pode usar serviços do ORB através do Object Adapter

Evocação estática

- Assume a existência de stubs ao nível do cliente e de skeletons ao nível do servidor.
- O stub do cliente estabelece a ligação entre o cliente e o ORB e o skeleton estabelece a ligação entre o ORB e o objecto.
  - Desta forma um pedido de um cliente passa pelo stub para chegar ao ORB e o ORB entrega-o ao skeleton para chegar ao objecto.
- Assume-se que a comunicação entre cliente e servidor está pré-definida, isto é conhecida quando são construídos cliente e servidor.
  - O código do cliente é ligado ao seu Stub obtendo um único executável.
  - Da mesma forma o código do servidor que implementa o método é ligado ao seu Skeleton obtendo também um único executável.
  - Daqui resulta que se um servidor receber pedidos de uma forma estática de vários clientes ele terá que ter outros tantos skeletons. E se um cliente tiver que contactar vários servidores ele terá que ter tantos stubs quantos os servidores a contactar.

Evocação estática

```
Interface Empregado {
  void promover(in char novo_posto);
  void despromover( in despromoCod razao,
    in String descricao);
  };
```



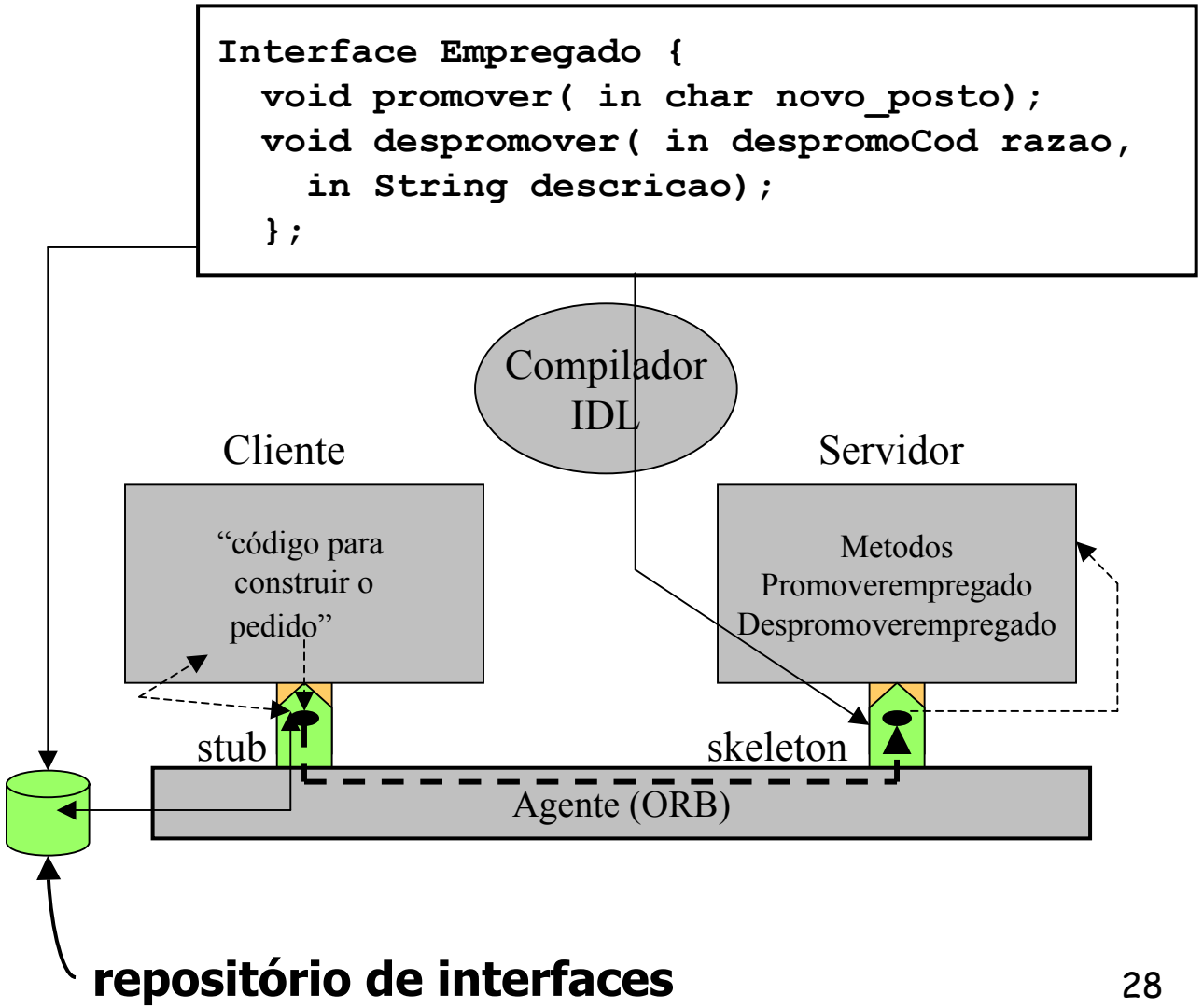
Evocação dinâmica

- Neste tipo de evocação o cliente não tem um stub tendo que construir dinamicamente o seu pedido.
- O servidor pode ter também um esqueleto dinâmico construído da mesma forma.
- Para evocar dinamicamente o cliente usa a interface de invocação dinâmica do Broker (DII).
  - Esta interface fornece o acesso a uma base de dados contendo a descrição das interfaces de todos os servidores que estão disponíveis no sistema. Desta forma o cliente descobre as operações possíveis nos objectos.
  - A construção do pedido fica a seu cargo tendo o código do cliente um troço dedicado exclusivamente a este fim.
- A base de dados contendo a descrição das interfaces é carregada a partir das interfaces em IDL.
  - O seu principal objectivo é o de permitir aos clientes em tempo de execução determinar a existência de interfaces não disponíveis na altura em que o código foi escrito.
  - Desta forma é possível adicionar novos servidores que podem ser usados por clientes já existentes sem necessidade de alterar os seus códigos desde que o tenham sido programados para usar evocação dinâmica.

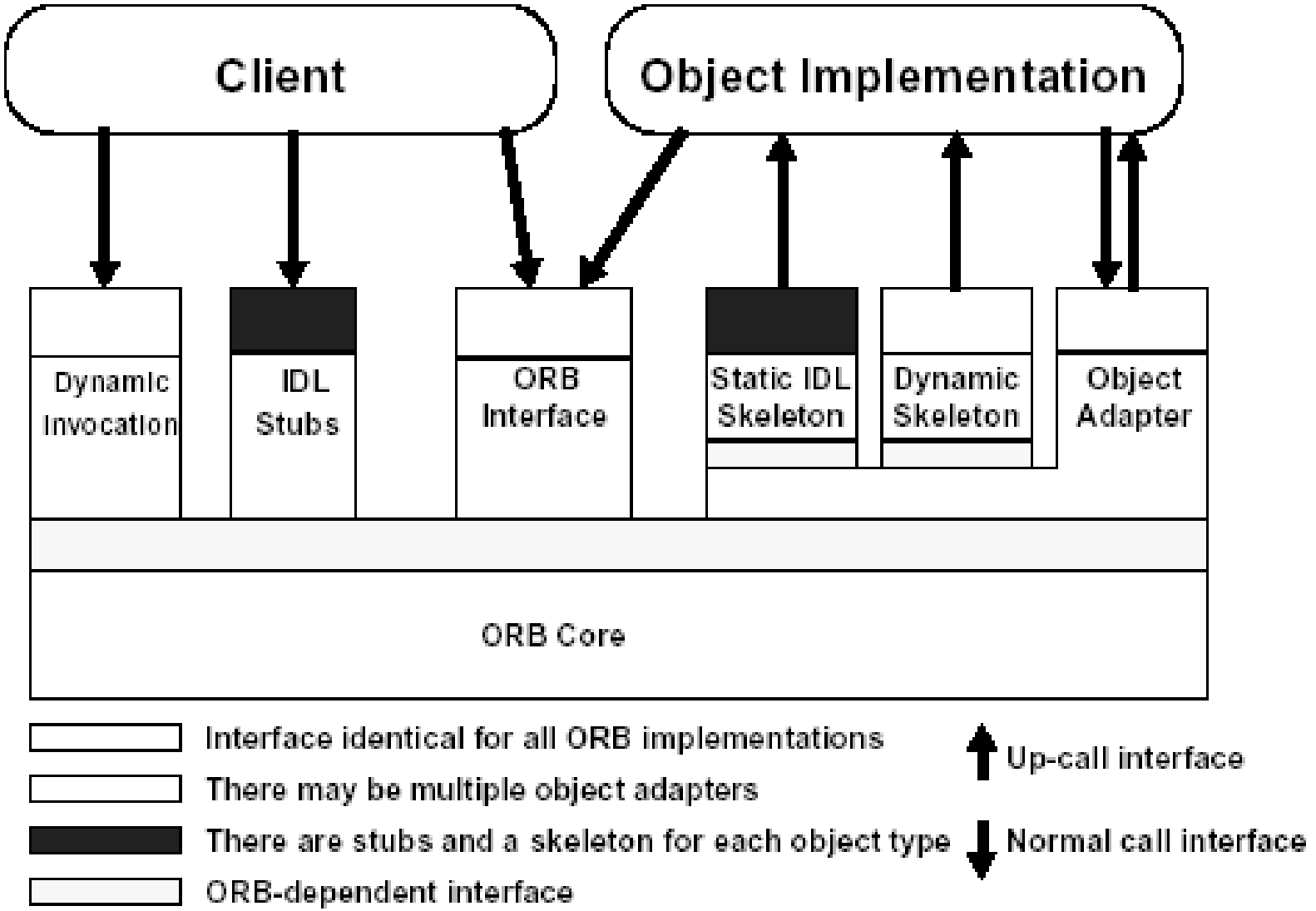
Evocação dinâmica

- um servidor não se apercebe se o cliente usou evocação estática ou dinâmica já que a semântica do pedido é a mesma.

- é da responsabilidade do ORB verificar se o pedido está correctamente formulado.



A estrutura do ORB



A estrutura do ORB

O Stub Cliente:

- Proxy local de objectos que delegam a evocação de métodos na implementação remota dos objectos
- Permite o fazer o marshalling de argumentos
- O stub faz evocações ao resto do ORB através de interfaces privadas do ORB

Marshalling – acção de transformar evocações métodos em mensagens a ser transmitidas e vice-versa

A estrutura do ORB

DII Dynamic Invocation Interface (Interface de Invocação Dinâmica): Permite a construção dinâmica de evocações

- O cliente especifica qual o objecto, qual a operação e quais os argumentos da operação, através de uma sequência de chamadas à interface de invocação dinâmica.
- A informação que caracteriza a operação (nº e tipo de parâmetros) é obtida de repositório de interfaces
- É permitida a invocação de operações em objectos cuja interface não era conhecida aquando o cliente foi desenvolvido
- Solução complexa, tipos não são verificados e menor eficiência

A estrutura do ORB

Skeleton

- Com um dado mapeamento, existe uma interface para os métodos que concretizam a interface do objecto
- É um interface de up-call
  - A concretização do objecto consiste em código que obedece a uma interface
  - O ORB invoca este código através do skeleton
  - Permite o unmarshalling
- A sua existência, que não é obrigatória, também não implica a existência de um stub (pode ser efectuada por DII)

A estrutura do ORB

DSI Dynamic Skeleton Interface (interface dinâmica de esqueleto)

- Análogo ao DII do lado do servidor
- Permite a uma concretização ter acesso ao nome da operação e aos seus parâmetros tal com com o DII
- Permite que um extracto de código sirva de concretização para muitos objectos com interfaces diferentes
- Possibilita a utilização de código num servidor que foi implementado sem recurso a uma interface IDL
- Interface de up-call
- Com DSI, a evocação tanto pode vir de stubs ou por DII (transparente)

A estrutura do ORB

Object Adapters

- É o meio principal que uma concretização de um objecto tem para aceder a serviços fornecidos pelo ORB
- Alguns destes serviços são
  - Criação de interpretação de referências para objectos
  - Invocação de métodos
  - Activação/desactivação de objectos e concretizações
  - Mapear referências em concretizações
  - Registar concretizações

A estrutura do ORB

POA - Portable Object Adapter

- Podem existir vários Objects Adapters, fazendo com que as concretizações dos objectos dependam disso.
- O POA permite independência do Object Adapter. Pode ser usado para a maioria dos objectos com concretizações convencionais
- Permite obter concretizações portáveis entre ORBs
- Ter objectos com identidades persistentes
- Activação transparente de objectos e concretizações
- Partilha de concretizações por objectos
- Ter várias instâncias de POA num servidor

A estrutura do ORB

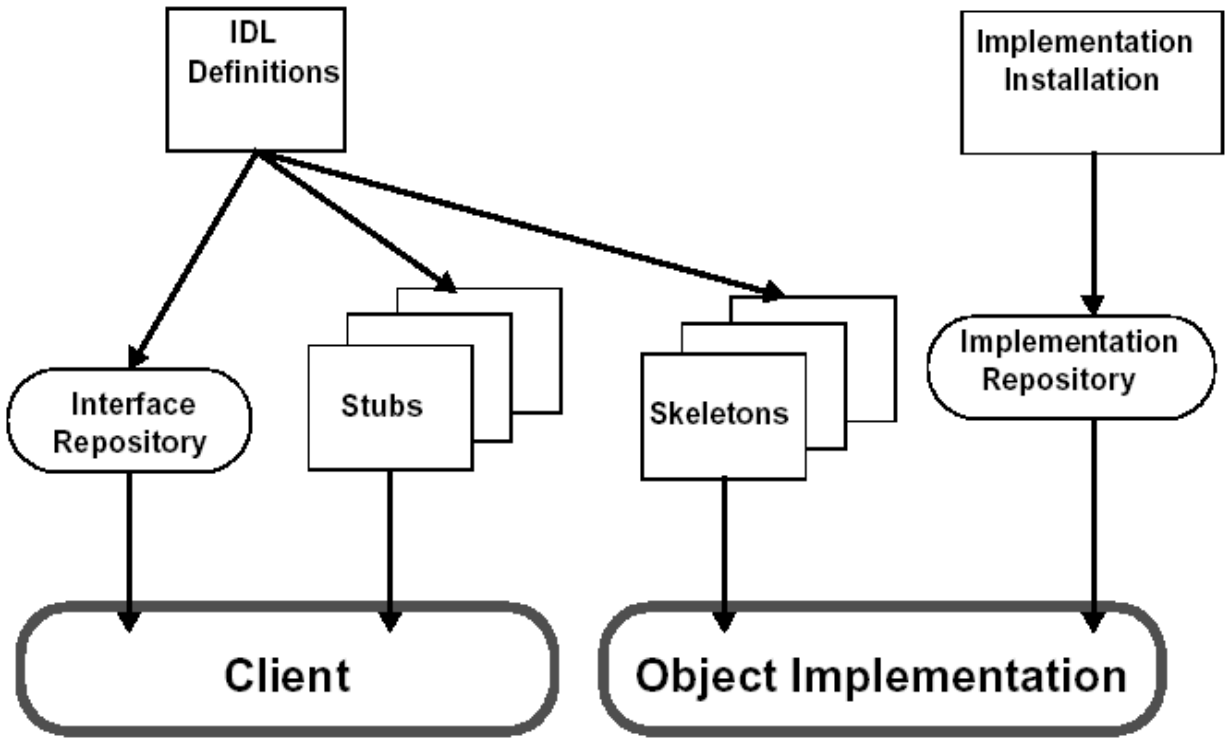
Interface Repository (Repositório de Interfaces)

- As interfaces podem ser adicionadas a um repositório de interfaces
- Esta guarda persistentemente a informação das interfaces como objectos, permitindo a consulta em tempo de execução
- Permite um programa ter informação sobre um objecto cuja interface não era conhecida quando o programa foi compilado
- Sabendo que operações são válidas, é possível proceder a invocações via DII

A estrutura do ORB

Implementation Repository (Repositório de concretizações)

- Contem informação que permite ao ORB localizar e activar concretizações de objectos



Sistemas distribuídos e Middleware

- Introdução aos sistemas distribuídos e middleware
- Arquitectura CORBA
  - Introdução. Arquitecturas
  - O IDL – Interface Definition Language
  - Interfaces ORB
  - Objectos factory e callback.
  - Interfaces POA
  - Serviços de nomes
  - Serviços de eventos
  - Modelo de concorrência
  - Activação dinâmica de servants e Implementation Repository
  - Interface Repository, DII e DSI
- Arquitectura DCOM
- Arquitectura Java RMI
- Arquitectura JINI

O IDL

- O IDL é uma linguagem declarativa, que não implementa código

- A sua sintaxe foi inspirada no C++, mas no entanto é distinta e com palavras chave próprias

Diferentes tipos de construtores suportados no IDL:

- Módulos – para separação de espaços de nomes
- Constantes – ajudam na declaração de tipos
- Declaração de tipos de dados – para atribuir tipos a parâmetros
- Interfaces – que agrupam declarações de tipos de dados, atributos e operações
- Atributos – para ler e escrever um valor de um dado tipo
- Operações – que têm parâmetros e valores de devolução

IDL - Análise lexical

- Identificadores
  - Sequência alfanumérica, ou underscore, sendo o primeiro caracter alfabético
  - Case-sensitive, mas identificadores que apenas difiram no 'case' não podem coexistir
    - Ex: `short DisplayTerminal` ≠ `interface displayTerminal`, mas não podem coexistir
- Pré-processamento
  - Como em C++ podem ser `#include, #define, #ifdef`, etc.
- Palavras chave
  - Todas em minúsculas, e não podem existir identificadores que apenas difiram no 'case'
- Comentários:
  - Como em C++, são admitidos dois tipos de comentários
    - O '/*' que abre um comentário e o '*/' que o fecha. Não é possível o aninhamento
    - O '//' no inicio de uma linha
- Pontuação
  - '{' começa um scope de nomes diferente.
  - Depois de um '}' existe sempre um ';'
  - Declarações são sempre seguidas de ';'
  - Parâmetros são limitados por '(' e ')' e separados por ','

IDL - Módulos

- O módulo é usado como um espaço de nomes, para evitar conflitos

  `module <identificador> { <definição> }`

- Esta pode conter qualquer IDL incluindo declarações de constantes, excepções e interfaces
- Pode ainda existir o aninhamento de módulos, e é permitida a sua reabertura

```
module exterior {
  module interior {
    interface dentro{};
  };
  interface fora {
    interior::dentro le_dentro();
  };
};
```

aninhamento de módulos

nomes relativos

IDL - Tipos

- Pode ser usado o typedef, como em C, para associar nomes a tipos

Existem:

- tipos básicos

  boolean, char, octet, string, short, unsigned short, long, unsigned long, long long, unsigned long, long, float, double

- tipos agregados

  struct, union, enum

- tipos template (requerem o uso de typedef)

  sequence, string wstring, fixed

- arrays

  multidimensionais, de tamanho fixo, com tamanho explicito para cada dimensão

IDL - Exemplo de tipos

```
modulo tipos {

  enum cor_vidro {v_branco, v_verde, v_castanho}; // enumerados

  struct carc_janela {  // estrutura
    cor_vidro cor;
    float altura;
    float largura;
  };

  //array dinâmico que pode ser limitado, mas não necessariamente
  typedef sequence <long, 10> carc_janela_limitada;
  typedef sequence <long> carc_janela;

  typedef carc_janela janela[10]; //array

  interface processaString {
     typedef octstring string <8>; //string limitada em tamanho
     ocstring oito_central(in string str); //string não limitada
  };
};
```

IDL -Interfaces

- Aqui se definem os interfaces para os objectos distribuídos: operações, etc...

```
interface <identificador> { <definição> }
```

Os interfaces podem conter declarações de tipos, constantes, excepções, atributos e operações

São possíveis interfaces vazias sem declarações

É possível uma declaração em avanço, sem definição, para permitir referências mutuas

```
module x {
  interface A; //declaração em avanço de A
}; //modulo x fechado
module y {
  interface B { //B pode usar x:A como um nome de tipo
    x::A le_um_A();
  };
};
module x { // reabertura do modulo para definir A
  interface C { //C pode usar A sem um qualificador
    A le_um_A();
  };
  interface A { //A pode usar y:B como um nome de tipo
    y::B le_um_B();
  };
};
```

IDL - Herança de interfaces

- Herança simples:

```
module heranca {
  interface A {
    typedef unsigned short ushort;
    ushort op1();
  };
  interface B:A {
    boolean op2(ushort num);
  };
};
```

- Pode haver herança de múltiplos interfaces

```
Interface Sub : SuperA, SuperB {...}
```

- Uma interface pode redefinir nomes de tipos, constantes e excepções herdadas

IDL - Operações e Atributos

```
[oneway] <tipo_resultado> <identificador>
   (<params>) <raises> <contexto>
```

- **oneway** significa que a operação é best-effort, at-most-once. Estas não devolvem, não bloqueiam e não levantam excepções
- Se a operação não devolver resultado este deverá ser **void**
- Os parametros podem ser de entrada, de saida ou de entrada e saida: **in**, **out** e **inout** respectivamente
- A operação pode especificar que excepções levanta, com **raises** (Excp1, Excep2, ...)

```
Interface Loja_Janelas {
  readonly attribute long max_pedidos;
  void PedeJanela (
    out long ord_ID,
    out float preco,
    in long quantidade,
    in carc_janela tipo)
   raises (JanelaGrande, CorErrada)
   context("MOEDA_LOCAL");
  };
```

IDL – Operações e Atributos

- A operação pode especificar com **context**(String1, String2, ...) que strings de contexto terão os seus valores passados ao objecto

- Uma interface pode conter atributos, declarados com **attribute**
- Um atributo é equivalente a um par de funções, para ler e modificar o valor
- Um atributo pode ser marcado **readonly**, indicando que é apenas de leitura

```
Interface Loja_Janelas {
  readonly attribute long max_pedidos;
  void PedeJanela (
    out long ord_ID,
    out float preco,
    in long quantidade,
    in carc_janela tipo)
   raises (JanelaGrande, CorErrada)
   context("MOEDA_LOCAL");
  };
```

Passagens em parâmetros

- Passagem por valor em parâmetros (o seu conteúdo é copiado)
  - Tipos básicos, estruturas, uniões e enumerações (tipos de dados que não são interfaces de objectos)
- Passagem por referência (partilha do mesmo objecto no cliente e no servidor)
  - Objectos regulares

- Não existe passagem por valor para tipos regulares de objectos, no entanto o CORBA define um tipo de dados híbrido – valuetype – cujos membros de estado podem ser passado por valor.

- ValueType
  - Mistura de interface e estrutura, declarados com a palavra chave **`valuetype`**
  - Tipo de dados especial em IDL para permitir a passagem por valor de "objectos" não distribuídos. Permite também a semântica do "null" para valores IDL, como nas estruturas e nas sequências.

ValueTypes

- Os ValueTypes podem ser:
  - Regulares ou Statefull
    - Pode conter atributos, operações e declaração de tipos, com a vantagem de ter membros de estado, em que o seu estado pode ser passado por valor (enviado) para o receptor, e de inicializadores
    - E.g.:

```
valuetype Node {
    private long value;
    long getValue();
    void setValue(in long val);
    };
```

  - Boxed
    - Versão simplificada do valuetype regular em que não existe operações ou herança mas apenas um membro de estado
  - Abstract
    - Não podem ser instanciados. Não têm inicializadores ou membros de estado. Conjunto de operações com implementações locais

IDL - Excepções e constantes

- A excepções são declaradas da mesma forma que as estruturas

```
exception JanelaGrande {
  float max_altura;
  float altura_submetida;
  };
```

- As constantes podem ser declaradas dentro de um espaço de nomes ou então globalmente. Pode-se usar expressões

```
const max_casas = 45;
const max_janelas = max_casas * 5;
```

Tipo de de dados **Any**

- Any é um tipo básico do IDL
  - Designa um container que pode conter qualquer tipo IDL e identifica o tipo do conteúdo.
  - O interface TypeCode é usado para identificar o tipo de Any
  - É da responsabilidade de cada linguagem de fornecer métodos para inserção e extracção de tipos básicos, bem como de Helper Classes para tipos definidos em IDL que gerem Anys

Mapeamento IDL para Java - conceitos

- A correspondência entre os conceitos constituintes de IDL e de JAVA é a seguinte:

IDL	JAVA
módulo	package
interface	interface
operação	método
excepção	excepção
constante	`public static final`
string	`java.lang.String`
`enum, struct`	classe
atributo de leitura	método para aceder ao valor de uma variável membro
atributo de leitura e de escrita	método para aceder e modificar o valor de uma variável membro

Mapeamento IDL para Java - tipos

- Também os tipos de dados básicos de IDL têm correspondência nos tipos de dados primitivos e de classe de JAVA:

IDL	JAVA
boolean	boolean
char	char
octet	byte
short	short
long	int
long long	long
float	float
double	double
string	String

- O mapeamento de tipos inteiros sem sinal (unsigned) em IDL não é feito de uma forma adequada para o Java que só admite tipos inteiros com sinal

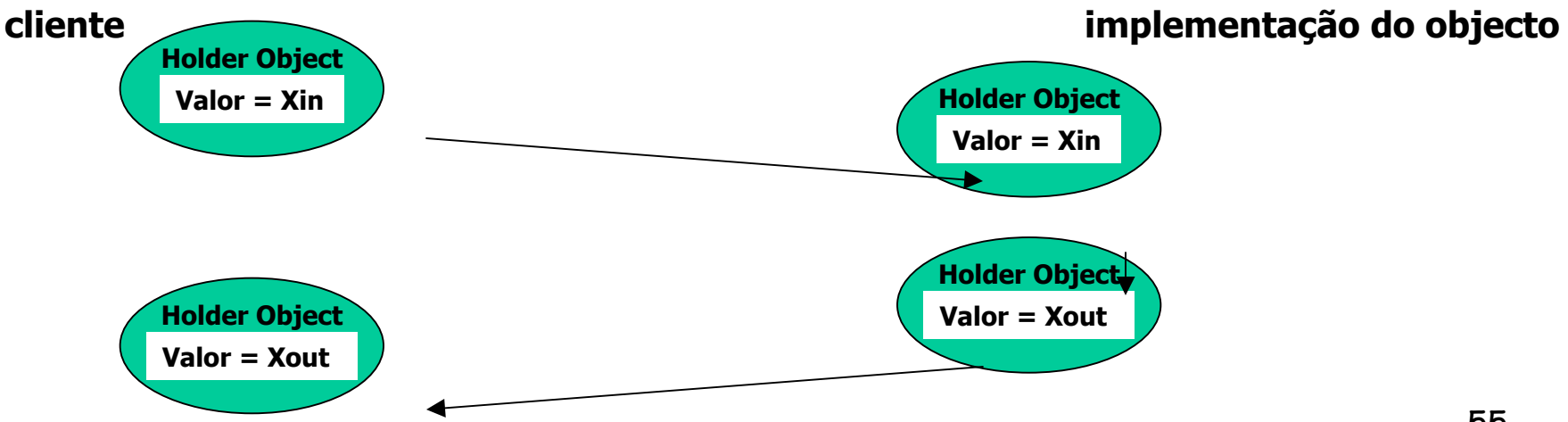
Mapeamento IDL para Java - operações

- As operações em IDL são mapeadas para métodos em Java
- O problema surge nos parâmetros, pois eles podem ter um acesso diferente. Ou seja os parâmetros:
  - in – passagem por valor
  - out – passagem por resultado
  - inout – passagem por referência
- O problema é que o Java apenas possui passagem por valor!

Solução:...

Mapeamento IDL para Java - operações

- Assim existe o conceito de HolderClass para tratar os casos out e inout
  - Uma HolderClasses passada por valor em Java, permite guardar este tipo de parâmetros como atributos dessa classe
  - Assim a passagem por valor/resultado é feita adequadamente, visto os atributos poderem ser modificados, pois é permitido que o estado de um objecto seja modificado.



Mapeamento IDL para Java - Excepções

- Excepções em IDL e em JAVA são semelhantes.

- Em CORBA existem dois tipos de excepções

System Exceptions, correspondem a problemas de sistema:

- excepções de sistema no servidor;
- excepções motivadas por deficiências de comunicação;
- excepções de sistema no cliente.
  - Apesar de todas as operações declaradas num interface IDL poderem levantar excepções, o interface IDL não necessita de prever o seu tratamento, que é implícito. O programador da aplicação cliente deve prever o seu lançamento dentro de um bloco try-catch de forma a não terminar a execução da aplicação inadvertidamente.

User Exceptions.

- As excepções declaradas no interface IDL dão origem a classes de excepções em JAVA, o que permite ao programador da aplicação implementar as suas próprias excepções de acordo com o desenho da aplicação.

Value Type Regulares

- Os valuetype regulares são traduzidos para classes abstractas em Java
- Esta classe tem um campo para cada membro estado
- Membro public transforma-se em public e private em protected.
- Os métodos têm que ser implementados tanto no lado do cliente como no lado do servidor (não há migração de código)

Exemplo IDL:

```
valuetype Node {
    private long value;
    long getValue();
    void setValue(in long val);
    factory createNode(in long val);
    };
```

Value Type Boxed

- Modo simples de passar valores nulos, onde antes era necessário que tipos não-objecto passassem valores vazios
- Exemplo: `valuetype sequence <Observer> ObserverSeq;`

- Existe duas forma de mapear em Java:
  - Para tipos primitivos que não são tranformados em classes, o valuetype é transformado numa classe em java

Ex: O IDL

```
valuetype SharedInteger long;
```

Fica

```
public classe SharedInteger
  implementas org.omg.CORBA.portable.ValueBase {
    public int value;
    public SharedInteger (int value) {
      this.value = value;
      }
    }
```

Value Type Boxed

- Para tipos primitivos que são transformados em classes, o valuetype não gera uma classe em java para si, mas é usada a classe original, mas o marshaling é feito segundo a semântica do value type. Para isso, tanto neste caso como no anterior, a Classe Helper tem os métodos adicionais read_value() e write_value()

Ex do caso anterior:

```
public final classe SharedIntegerHelper
  implementas org.omg.CORBA.portable.BoxedValueHelper {
    ...
    public java.io.Serializable read_value(
  org.omg.CORBA.portable.InputStream in ) {...}
    public void write_value(org.omg.CORBA.portable.OutputStream
  out, java.io.Serializable value) {...}
    ...
}
```

Tipo de de dados any: exemplo

```
public abstract class Any implements org.omg.CORBA.portable.IDLEntity {
        public abstract boolean equal(org.omg.CORBA.Any a);
        public abstract org.omg.CORBA.TypeCode type();
        public abstract void type(org.omg.CORBA.TypeCode t);
        public abstract void
      read_value(org.omg.CORBA.portable.InputStream is,
      org.omg.CORBA.TypeCode t)
            throws org.omg.CORBA.MARSHAL;
        public abstract void
      write_value(org.omg.CORBA.portable.OutputStream os);
        public abstract org.omg.CORBA.portable.OutputStream
      create_output_stream();
        public abstract org.omg.CORBA.portable.InputStream
      create_input_stream();
        public abstract short extract_short()
      throws org.omg.CORBA.BAD_OPERATION;
        public abstract void insert_short(short s);
        public abstract int extract_long()
      throws org.omg.CORBA.BAD_OPERATION;
        public abstract void insert_long(int i);
        public abstract long extract_longlong()
      throws org.omg.CORBA.BAD_OPERATION;
        //etc...
```

Sistemas distribuídos e Middleware

- Introdução aos sistemas distribuídos e middleware
- Arquitectura CORBA
  - Introdução. Arquitecturas
  - O IDL – Interface Definition Language
  - Interfaces ORB
  - Objectos factory e callback.
  - Interfaces POA
  - Serviços de nomes
  - Serviços de eventos
  - Modelo de concorrência
  - Activação dinâmica de servants e Implementation Repository
  - Interface Repository, DII e DSI
- Arquitectura DCOM
- Arquitectura Java RMI
- Arquitectura JINI

O Interface ORB

- A interface encontra-se definida no módulo IDL CORBA
- A Interface do ORB contém operações que não dependem do Object Adapter usado e que são iguais para todos os ORB e concretizações de objectos
- Podem ser invocadas por clientes ou concretizações de objectos
- Não são operações sobre os objectos

O Interface ORB

- O ORB é um pseudo-objecto com interface IDL
  - Serve por exemplo para criar representação em strings de referências de objectos Corba

```
Module CORBA {
  interface ORB {
    string object_to_string(in Object obj);
    Object string_to_object (in string str);
    ...
```

  - Pode servir também para obter informação sobre serviços disponibilizados

```
    boolean get_service_information (in ServiceType service_type,
      out ServiceInformation service_information);
    ...
```

O Interface ORB

```
module CORBA {
  interface ORB {
    ...
```

- O interface ORB pode servir para por exemplo obter referências iniciais. Serve para localizar objectos essenciais. Isto funciona como um serviço de nomes. E.g.: NameService, TradingService, SecurityCurrent, InterfaceRepository, etc...

```
    typedef string ObjectId;
    typedef sequence <ObjectId> ObjectIdList;
    exception InvalideName {};
    ObjectIdList list_initial_referencies();
    Object resolve_initial_references (in ObjectId identifier)
      raises (InvalidName);
    ...
  };
```

Interface do ORB

Para se usar um ORB, uma dada aplicação necessita de obter uma referência para um pseudo-objecto com interface ORB – bootstraping; Tal é obtido com a operação ORB_init (que não é executada sobre um objecto)

```
module CORBA {
  typedef string ORBid;
  typedef sequence <string> arg_list;
  ORB ORB_init (inout arg_list argv, in ORBid orb_identifier);
  };
```

Exemplo em java:

```
org.omg.CORBA.ORB orb =
        org.omg.CORBA.ORB.init(args, new java.util.Properties());
```

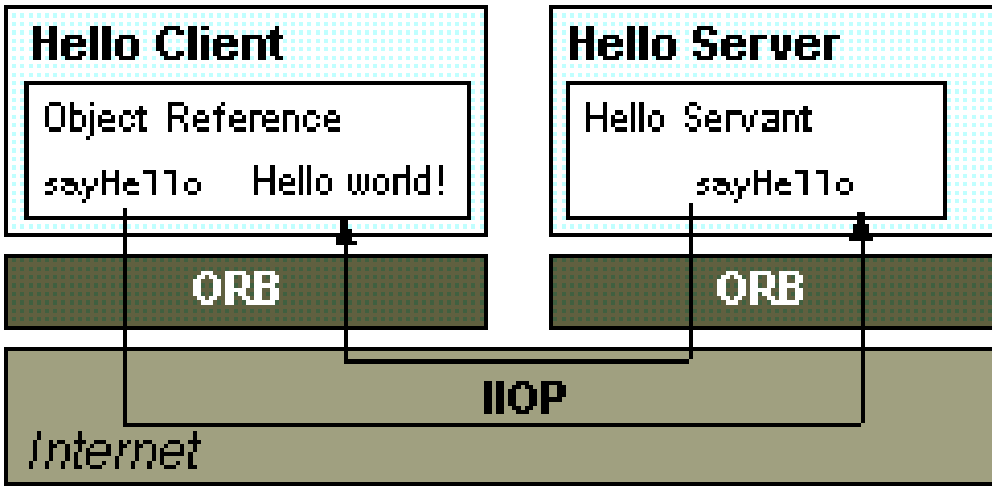
O Interface Object

Outro interface IDL é designado Object e define operações sobre referências de objectos. Estas operações são locais e não se relacionam com a concretização do objecto

```
interface Object {
   InterfaceDef get_interface ();
   Object duplicate ();
   void release();
   ...
};
```

Exemplo do Olá Mundo

Neste exemplo, um cliente, funcionando em modo applet ou em modo aplicação, invoca a operação `sayHello()` *de um objecto distribuído CORBA localizado no servidor. Na sequência do pedido do cliente, o seu ORB contacta o ORB do servidor que está registado como contendo o objecto distribuído CORBA que executa a operação* `sayHello()`*. A operação* `sayHello()` *definida no servidor é executada, retornando a string « Hello World », o que permite ao ORB do cliente obter esta string, que é impressa na consola do cliente.*



Exemplo do Olá Mundo

Exemplo do módulo em IDL para o exemplo « Hello World » (ficheiro Hello.idl):

```
module HelloApp{
   interface Hello {
     string sayHello();
   };
};
```

Este código em IDL, quando compilado e convertido em JAVA, dá origem ao package `HelloApp` *e à classe Hello que contém o método* `String sayHello()`.

Compilação do IDL

- Depois de declarado, um módulo IDL é compilado e implementado na linguagem de programação escolhida para a implementação dos objectos distribuídos CORBA.
- No que respeita ao caso especifico de JAVA, estão disponíveis diversos compiladores que constróem ficheiros preparatórios para a programação de uma aplicação CORBA.
- A plataforma Java (desde a versão SDK 1.2) integra o compilador idltojava, um disponibilizado pela Sun Microsystems, Inc.
- No entanto as potencialidades CORBA no Java Sun não são totalmente exploradas...
- Existem outras soluções CORBA para Java mais completas. O **ORBacus** é um desses exemplos, com o qual é fornecido o compilador para Java: **jidl**

Exemplo de utilização do compilador **jidl**

- A compilação de Hello.idl por idlj gera vários ficheiros, entre os quais Hello.java, posicionado na directoria HelloApp.

Resultado da compilação

Os ficheiros gerados pela compilação de `Hello.idl` *por jidl são os seguintes:*

Hello.java	***O interface IDL é convertido num interface JAVA sem as operações***
HelloOperations.java	***As operações do interface IDL são postas neste interface***
HelloHelper.java	***Classe auxiliar que permite converter as referências para objectos distribuídos CORBA nos tipos que lhes correspondem***
HelloHolder.java	***Ficheiro que contém os parâmetros out e inout declarados no interface***
_HelloStub.java	***Implementação no cliente de um objecto que representa o objecto distribuído CORBA remoto. Ista class faz part do stub***
HelloPOA.java	***Ficheiro que gera a estrutura da classe que permite implementar no servidor as operações declaradas pelo interface***

Hello.java contém o código JAVA seguinte :

```
package HelloApp;
public interface Hello extends HelloOperations,
                               org.omg.CORBA.Object,
                               org.omg.CORBA.portable.IDLEntity { }
```

Como se pode verificar, o interface Hello deriva de org.omg.CORBA.Object, o que é uma característica comum a todos os interfaces IDL.

A Inicialização do ORB

- A criação de um objecto do tipo ORB permite que uma aplicação (ou applet) possa executar as operações a que o ORB dá acesso.

Exemplo: criação de um ORB para uma aplicação :

```
import org.omg.CORBA.ORB;
    public static void main(String args[]) {
      try {
        ORB orb = ORB.init(args, null);
        // …
```

A invocação do método estático init da classe ORB possibilita a criação de um objecto da classe ORB utilizando argumentos obtidos a partir da linha de comando, o que permite definir no momento da instanciação quais as propriedades do ORB. O segundo parâmetro de init (null no exemplo acima) pode ser um objecto do tipo Properties que associa as propriedades do sistema (do cliente ou do servidor) ao objecto ORB.

A Inicialização do ORB

Exemplo: Criação de um ORB para um applet :

Também no caso de um cliente programado em modo applet, a instanciação da classe ORB recebe as propriedades do sistema.

```
import org.omg.CORBA.ORB;
    public void init() {
      try {
        ORB orb = ORB.init(this, null);
       // …
```

- Neste caso, this referencia a instância implícita do applet cujas propriedades são obtidas através de System.getProperties().

Tornar o objecto distribuído invocável

- Um servidor pode prestar diferentes serviços, sendo cada um identificado por um objecto objecto distribuído CORBA.
- Um objecto distribuído CORBA implementa o interface gerado pelo compilador de IDL, isto é, implementa as operações declaradas nesse interface.
- No Java existe um classe servo que implementa esse interface e que forma o esqueleto do objecto:

```
public abstract class InterfaceNamePOA extends
   org.omg.PortableServer.Servantimplements
   implements org.omg.CORBA.portable.InvokeHandler, InterfaceNameOperations {
   ... }
```

- O programador tem que implementar uma classe que estende este esqueleto, adicionando o código necessário aos métodos.

```
   public class InterfaceNameServant extends InterfaceNamePOA {
   ... }
```

Tornar o objecto distribuído invocável

- Depois de criado a classe que implementa o interface tem que se tornar essa implementação acessível via o ORB:
  - Não usar o Object Adapter, usando para isso a operação connect() do ORB. O único modo suportado pelo Java IDL
  - Usar o Object Adapter
  - Usar o Portable Object Adapter

Exemplo do POA:

```
// iniciar o POA responsável pela gestão de objectos no servidor
org.omg.CORBA.Object poaObject = orb.resolve_initial_references("RootPOA")
org.omg.PortableServer.POA rootPOA = org.omg.PortableServer.POAHelper.narrow(poaObject);
org.omg.PortableServer.POAManager manager = rootPOA.the_POAManager();
manager.activate();

// Torna o objecto invocavel em clientes CORBA.
org.omg.CORBA.Object obj = rootPOA.servant_to_reference ( helloRef );
```

Obtenção da referência de um objecto CORBA

- Para invocar um método referente a um objecto distribuído CORBA, um cliente necessita de uma referência para esse objecto. Uma das soluções para obter essa referência é a conversão de uma string criada a partir de uma referência para um objecto distribuído CORBA.

- Para isso, a aplicação servidor deve inicialmente converter a referência para o objecto distribuído CORBA numa string.

Exemplo:

```
// instanciação do objecto ORB
org.omg.CORBA.ORB orb = ...
// instanciação do objecto POA
org.omg.PortableServer.POA rootPOA = ...
// criação da referência para um objecto genérico CORBA
org.omg.CORBA.Object obj = ...
//referência em forma de string
String str = orb.object_to_string(obj);
```

Obtenção da referência de um objecto CORBA

- Seguidamente, o cliente obtém a string (por exemplo, a partir da leitura de um ficheiro) e transforma-a numa referência para uma réplica do objecto distribuído CORBA que corre no servidor.

```
// instanciação de um ORB
org.omg.CORBA.ORB orb = ...
// leitura da string (por exemplo a partir de um ficheiro)
String stringifiedref = ...
// instanciação de um objecto genérico CORBA a partir da string
org.omg.CORBA.Object obj =
  orb.string_to_object(stringifiedref);
```

Aplicação cliente

```
import HelloApp.*;
import org.omg.CORBA.*;
import java.io.*;

public class HelloStringifiedClient{
  public static void main(String args[]) {
    try{
    //propiedades para comecarmos o ORB da ORBacus e nao o da SUN
    java.util.Properties props = System.getProperties();
    System.out.println (props.getProperty("java.version"));
    props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
    props.put("org.omg.CORBA.ORBSingletonClass", "com.ooc.CORBA.ORBSingleton");
    // cria e inicializa o ORB
    ORB orb = ORB.init(args, props);
    // Le do ficheiro de texto que tem a referência stringified.
    String filename = System.getProperty("user.home") +
      System.getProperty("file.separator") +
      "HelloIOR";
    FileInputStream fis = new FileInputStream(filename);
    BufferedReader br = new BufferedReader(new InputStreamReader(fis));
```

Aplicação cliente, continuação

```
    String ior = br.readLine();
    // Obter a referência para o objecto distribuído CORBA através da
    // String que a representa
    org.omg.CORBA.Object obj = orb.string_to_object(ior);
    // Converter o objecto distribuído CORBA no seu tipo
    Hello helloRef = HelloHelper.narrow(obj);
    // chama o objecto servidor Hello e imprime o resultado
    String hello = helloRef.sayHello();
    System.out.println(hello);
    } catch (Exception e) {
      System.out.println("Erro : " + e) ;
      e.printStackTrace(System.out);
    }
  }
}
```

Aplicação servidor

```
import HelloApp.*;
import org.omg.CORBA.*;
import java.io.*;
import HelloServant;
public class HelloStringifiedServer {
  public static void main(String args[]){
    try{
      // propriedades para comecarmos o ORB da ORBacus e nao o da SUN
      java.util.Properties props = System.getProperties();
      System.out.println (props.getProperty("java.version"));
      props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
      props.put("org.omg.CORBA.ORBSingletonClass", "com.ooc.CORBA.ORBSingleton");

      // cria e inicializa o ORB
      ORB orb = ORB.init(args, props);

      // iniciar o POA responsável pela gestão de objectos no servidor
      org.omg.PortableServer.POA rootPOA = org.omg.PortableServer.POAHelper.narrow(
          orb.resolve_initial_references("RootPOA"));
      org.omg.PortableServer.POAManager manager = rootPOA.the_POAManager();
      manager.activate();

      // cria o objecto distribuído CORBA e regista-o no ORB
      HelloServant helloRef = new HelloServant();
```

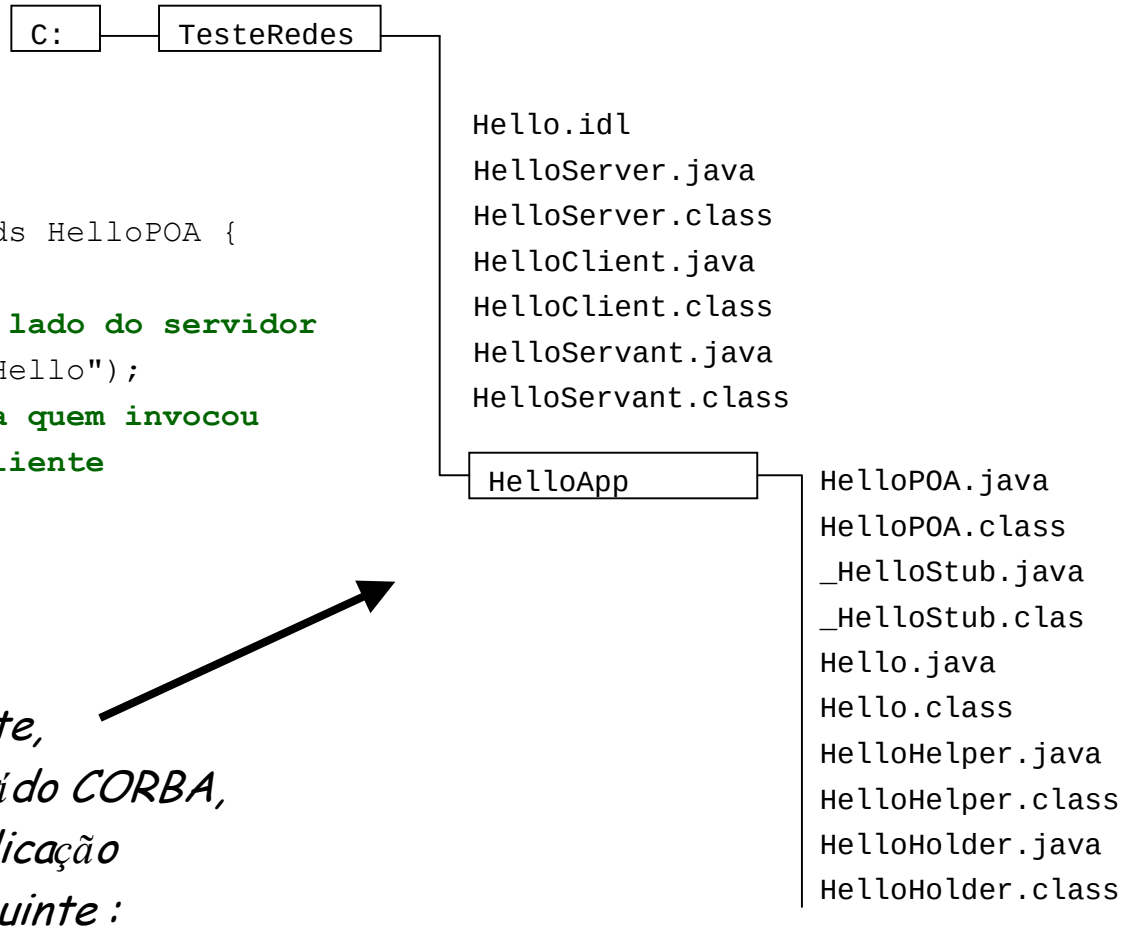
Aplicação servidor, continuação

```
// Torna o objecto invocavel em clientes CORBA.
// Criar uma referencia nao persistente para o nosso servante que se possa usar
org.omg.CORBA.Object obj = rootPOA.servant_to_reference ( helloRef );

//Obter uma referência para um objecto distribuído CORBA
//na forma de um objecto do tipo String, de forma a poder ser gravada em disco
String ior = orb.object_to_string(obj);

//Construir uma path para o ficheiro, através das propriedades do sistema
String filename = System.getProperty("user.home") +
        System.getProperty("file.separator") + "HelloIOR";
//Grave o ficheiro de texto, contendo a String, em disco
FileOutputStream fos = new FileOutputStream(filename);
PrintStream ps = new PrintStream(fos);
ps.print(ior);
ps.close();

// fica à espera do contacto de clientes
orb.run();

 } catch (Exception e) {
  System.err.println("Erro: " + e);
e.printStackTrace(System.out);
 } } }
```

O objecto distribuído

- *No caso do exemplo "Hello World", a classe HelloServer tem como classe auxiliar a classe HelloServant que implementa o servidor "Hello" e que é codificada em JAVA da seguinte forma :*

```
import HelloApp.*;
import java.io.*;
public class HelloServant extends HelloPOA {
  public String sayHello() {
    //escreve «Saying Hello» no lado do servidor
    System.out.println("Saying Hello");
    //devolve «Hello World» para quem invocou
    //este metodo, ou seja, o cliente
    return "\nHello World\n";
  }
}
```



Depois de programados o cliente, o servidor e o objecto distribuído CORBA, a estrutura de ficheiros da aplicação CORBA « Hello World » é a seguinte :

Execução:

Para executar as aplicações servidor e cliente em Java deve:

1. *Compile o ficheiro IDL com o interface do objecto distribuído CORBA*

   ```
   C:\tesredes\jidl -cpp ... hello.idl
   ```

2. *Compile os ficheiros JAVA que foram criados*

   ```
   C:\tesredes\javac HelloApp\*.java
   ```

- *Crie a aplicação cliente usando as classes criadas pelo **idlj** do ORBacus*
- *Compile a aplicação cliente*

  ```
  C:\tesredes\javac HelloStringifiedClient
  ```

- *Crie a classe que concretiza o interface do objecto a distribuir, estendendo a classe esqueleto criada pelo **idltj** do ORBacus*
- *Crie a aplicação servidor usando as classes criadas pelo **idltj** do ORBacus*
- *Compile a aplicação servidor*

  ```
  C:\tesredes\javac HelloStringifiedServer
  ```

1. *Corra a aplicação servidor*

   ```
   C:\tesredes\java ... HelloStringifiedServer
   ```

2. *Corra a aplicação cliente*

   ```
   C:\tesredes\java HelloStringifiedClient
   ```

Sistemas distribuídos e Middleware

- Introdução aos sistemas distribuídos e middleware
- Arquitectura CORBA
  - Introdução. Arquitecturas
  - O IDL – Interface Definition Language
  - Interfaces ORB
  - Objectos factory e callback.
  - Interfaces POA
  - Serviços de nomes
  - Serviços de eventos
  - Modelo de concorrência
  - Activação dinâmica de servants e Implementation Repository
  - Interface Repository, DII e DSI
- Arquitectura DCOM
- Arquitectura Java RMI
- Arquitectura JINI

Factory Objects

- Objectos que proporcionam acesso a um ou mais objectos adicionais
- Ponto focal para os clientes CORBA
  - referências para este objecto pode ser conhecida
  - referências subsequentes são devolvidas pelo factory object
- Aplicabilidade:
  - Segurança
    - o cliente fornece informação de segurança ao factory object para ganhar acesso a outro objectos
  - Load-balacing
    - o factory object gere um conjunto de objectos, e gere a atribuição dos clientes a este segundo um algoritmo de utilização
  - Polimorfismo
    - o factory object pode devolver referencias para objectos diferentes segundo um critério a obter do cliente

Exemplo IDL, em que um novo objecto é criado para cada cliente:

```
// IDL
interface Product {
      void destroy(); //permite o cliente destruir um objecto já não usado
      };
interface Factory {
      Product createProduct(); //devolve a referência de um novo producto
      };
```

Factory Objects - exemplo em Java

Implementação em Java do interface Product:

```
// Java
// implementação do interface Produto (o esqueleto é o objecto ProductPOA)
public class Product_impl extends ProductPOA {
  //desactiva o servant do POA (mantém uma contagem de referencias usadas para este servant)
  //se não existirem outras referências para o servant este pode ser libertado
  public void destroy() {
    byte[] id = _default_POA().servant_to_id(this);
    _default_POA().deactivate_object(id);
  }
}
```

Implementação em Java do do Factory do Produto:

```
// Java
// implementação do interface Produto (o esqueleto é o objecto ProductPOA)
public class Factory_impl extends FactoryPOA {
  public Product createProduct() {
    Product_impl result = new Product_impl(); //activa o servant
    //obtem o POA do servant. Pode-se usar o _default_POA(), sendo obtido o RootPOA se não
    //tivermos feito o override deste metedo para um POA guardado localmente
    org.omg.PortableServer.POA poa = ...
    byte[] id = ... // Escolhe um ID
    poa.activate_object_with_id(id, result); // activa o servant no POA com Id especificado
    return result._this(orb_); //devolve uma referencia de Product para o cliente
  }
}
```

Callback Objects

- Os clientes de uma aplicação distribuída CORBA podem ter necessidade de conhecer periodicamente qual o estado dos objectos distribuídos CORBA.
  - Neste caso, podem contactar regularmente os objectos distribuídos CORBA para actualizar a informação relativa ao seu estado
  - ou podem pedir ao servidor para serem notificados quando um objecto distribuído CORBA muda de estado. Nesta ultima alternativa são utilizados objectos callback

Definição do interface IDL para o objecto distribuído CORBA Hello

```
//IDL
module Loja {
  //interface usado do lado do cliente para receber notificações callbacks
  interface EncomendaCallback {
    oneway void pronta(in string message);
    };
  //interface usado do lado do servidor para receber uma mensagem
  //e uma interface de callback a invocar quando necessário nos cliente
  interface Encomenda {
    string creat(in EncomendaCallback objRef, in string message);
  };
};
```

Callback Objects - exemplo em Java

Classe do objecto distribuído a usar do lado do servidor

```
class Encomenda_impl extends EncomendaPOA {
  //método que recebe uma referencia para um objecto de callback
  public String creat(EncomendaCallback callobj, String msg) {
    //o objecto de callback foi invocado dentro deste método, mas a referência
    //poderia ser guardada para uma invocação posterior
    callobj.pronta("Encomenda do " + msg + " esta pronta");
    return "\nEncomenda creada !!\n";
  }
}
```

Classe do objecto distribuído a usar do lado do cliente e que será usada para callback

```
class EncomendaCallback_impl extends EncomendaCallbackPOA {
  //método para receber os callbacks do servidor. Imprime a notificação recebida
  public void pronta(String notification){
    System.out.println(notification);
  }
}
```

Extracto de código do cliente que irá criar entregar a referência do objecto callback

```
//criação do servant para responder aos pedidos de callback
EncomendaCallback_impl encomenda_cb_impl =
  new HelloCallback_impl ();
//activação implícita do servant no POA de defeito deste servant
//será o RootPOA se outro não foi especificado
EncomendaCallback encomenda_cb_ref = encomenda_cb_impl._this();
// obtenção da referência para o objecto Encomenda de uma dada forma
...
// chamada do servidor Encomenda, em que é enviado o objecto de callback
String estado =
  encomenda_ref.creat(encomenda_cb_ref, "\nproduto x\n");
```