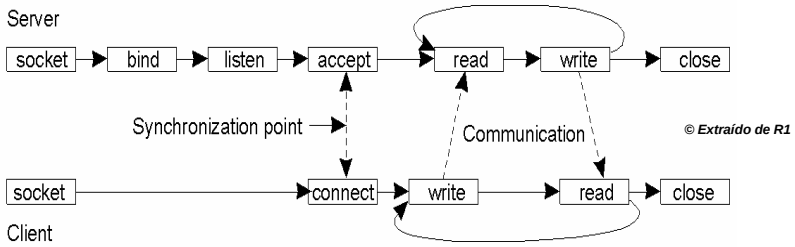
✓Objectos Distribuídos

✓Evolução da Computação Distribuída

Evolução da computação distribuída

- Unix distribuído – Berkeley Sockets
- Sun Remote Procedure Call (RPC)
  - RPC Language / eXternal Data Representation (XDR)
- Distributed Computing Environment DCE
  - Interface Definition Language (IDL)
- CORBA (Common Object Request Broker Architecture)
  - CORBA IDL
- Microsoft COM (Common Object Model) / DCOM (distributed COM)
- JAVA RMI (Remote Method Invocation)
- .NET Platform (Remoting)
- Arquitecturas Orientadas ao Serviço

Comunicação cliente servidor, usando *sockets*



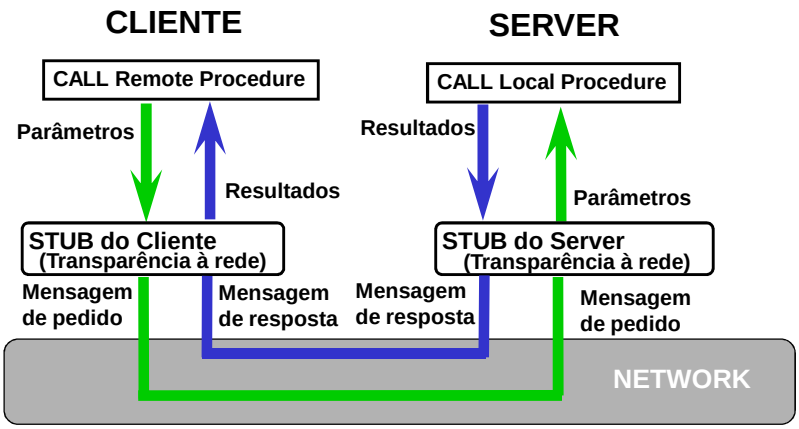
Primitiva	Descrição
Socket	Cria um novo ponto de comunicação (endpoint)
Bind	Associa um socket a um porto
Listen	Anuncia a disponibilidade para aceitar ligações
Accept	Bloqueia o processo até chegar um pedido
Connect	Estabelece uma ligação
Send/write	Envia dados pela ligação estabelecida
Receive/read	Recebe dados pela ligação estabelecida
Close	Termina a ligação

Exemplos cliente/servidor, usando *sockets* em .NET

O ficheiro *ExemplosSockets.zip*, contém exemplos em Visual Studio 2010 com aplicações cliente/servidor, usando *sockets* TCP com o servidor no porto 5000:

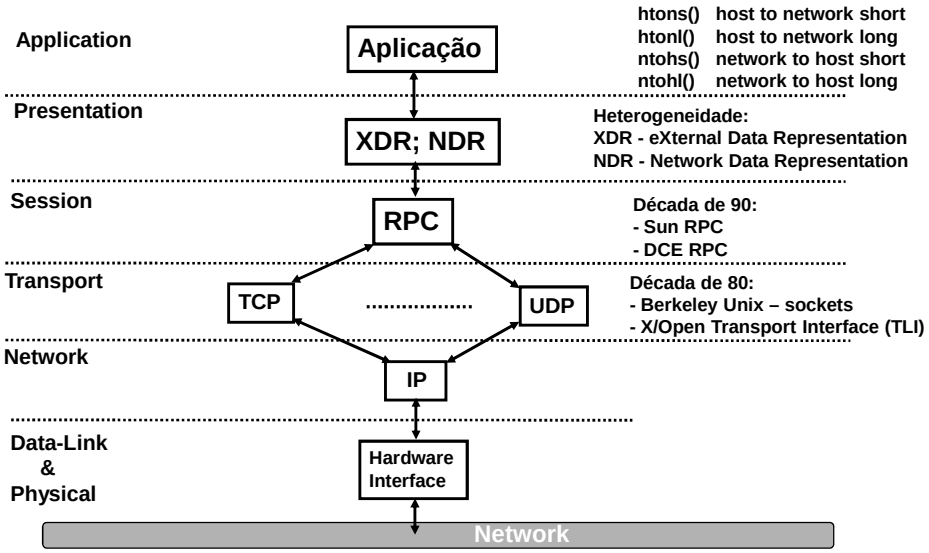
- **No início das aplicações é pedido ao utilizador o nome da máquina onde se encontra a aplicação servidora.**
  - **Exemplo: *Sockets***
    - **O Cliente envia uma mensagem ao servidor**
    - **O Servidor recebe a mensagem, converte-a para maiúsculas e devolve-a ao cliente**
    - **O cliente recebe a mensagem de resposta .**
  - **Exemplo: *SocketsWithObjectSerialization***
    - **Existe uma Classe Produto com atributo *[Serializable]***
    - **O cliente envia um produto para o servidor**
    - **O servidor devolve um novo produto com outro código e designação**

RPC - Remote Procedure Call

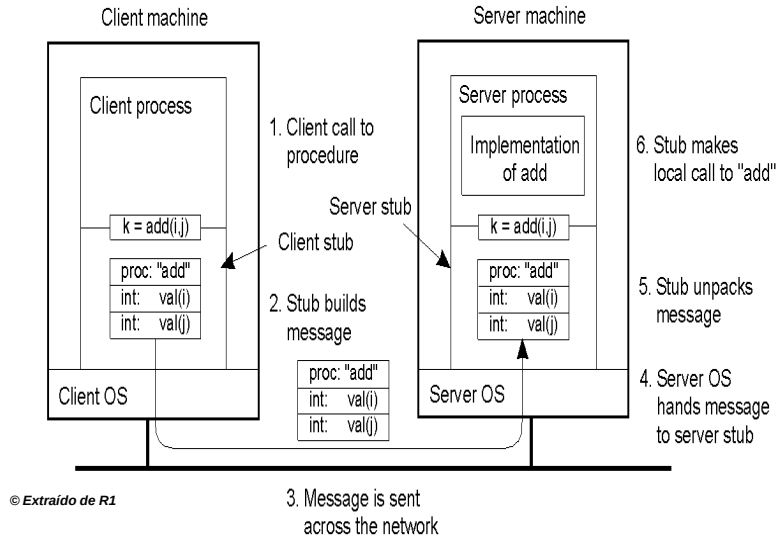


- Os processos CLIENTE e SERVIDOR podem executar-se em máquinas diferentes e comunicam através de STUBs;
- Os dois STUBs comunicam, através da rede, usando mensagens.

Enquadramento no modelo OSI dos RPC

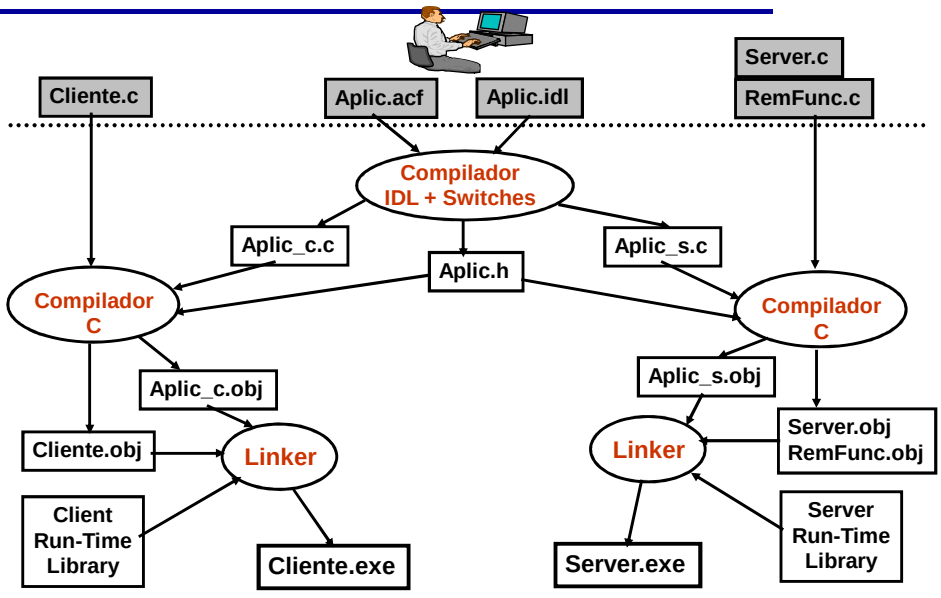


Passos envolvidos numa chamada remota - RPC



© Extraído de R1

Desenvolver aplicações Cliente Servidor com DCE-RPC



RPC – Exemplo: Definição da Interface IDL

PrintStr.idl

```
[
uuid(e1658181-8c6a-4e4f-a1d6-1c1a0a0adebb),
version(1.0)
]
interface INTERFACENAME
{

}
```

Geração do UUID e ficheiro IDL

```
C:\> uuidgen -i –o PrintStr.idl
```

PrintStr.idl

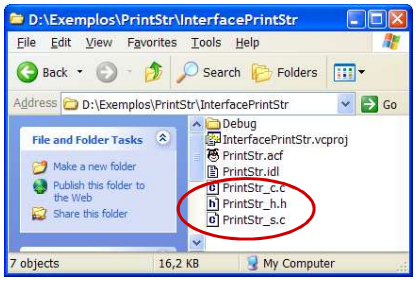
```
[
uuid(e1658181-8c6a-4e4f-a1d6-1c1a0a0adebb),
version(1.0)
]
interface PrintStr
{
  void printChar( [in] char *pch );
  void printStr([in, string] char * pszString);
  void shutdown(void);
}
```

Inclui a possibilidade de o cliente poder ordenar a paragem da aplicação servidora, através do serviço "Shutdown".

RPC – Exemplo: Geração dos *Stubs* Cliente e Servidor

```
C:\> midl PrintStr.idl
```
(ou Build do Visual Studio)

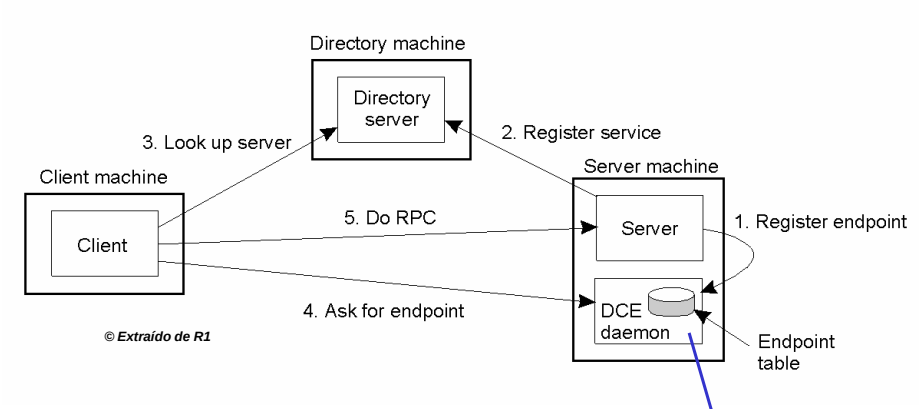
São gerados os ficheiros:
PrintStr_h.h
PrintStr_s.c
PrintStr_c.c



PrintStr_h.h

```
...
#ifdef __cplusplus
extern "C"{
#endif
void * __RPC_USER MIDL_user_allocate(size_t);
void   __RPC_USER MIDL_user_free( void * );

#ifndef __PrintStr_INTERFACE_DEFINED__
#define __PrintStr_INTERFACE_DEFINED__
/* interface PrintStr */
/* [implicit_handle][version][uuid] */
void printChar(
    /* [in] */ unsigned char *pch);

void printStr(
    /* [string][in] */ unsigned char *pszString);

void shutDown( void);

extern handle_t PrintStr_IfHandle;

extern RPC_IF_HANDLE PrintStr_v1_0_c_ifspec;
extern RPC_IF_HANDLE PrintStr_v1_0_s_ifspec;
...
#endif
```

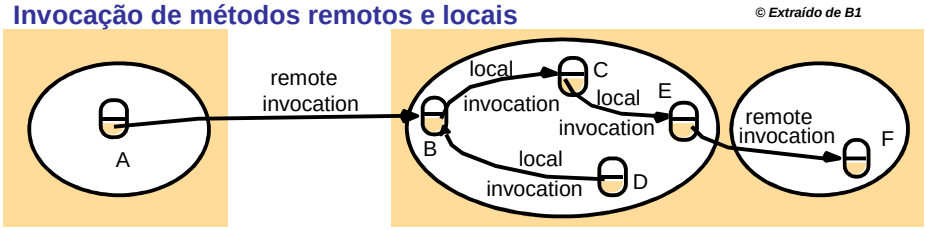
Binding entre Cliente e servidor em DCE.



Em Sun RPC este *daemon* designava-se por PORTMAPPER e podia ser acedido através de uma interface RPC no porto 111

Objectos Distribuídos

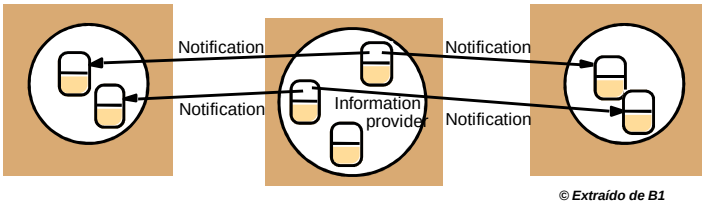
Invocação de métodos remotos e locais



- **Semântica idêntica à invocação de métodos num objecto local com algumas diferenças que não podem ser eliminadas.**
- **Os dados dos objectos (atributos) devem ser acedidos unicamente através de métodos;**
- **As referências para objectos podem ser atribuídas a variáveis, passadas como argumento e serem retornados como resultado de um método;**
- **Deverá ser suportado um mecanismos de excepções que permita que um objecto remoto possa propagar (*throw*) excepções para o objecto cliente.**

Eventos em objectos distribuídos

Notificação de eventos remotos



© Extraído de B1

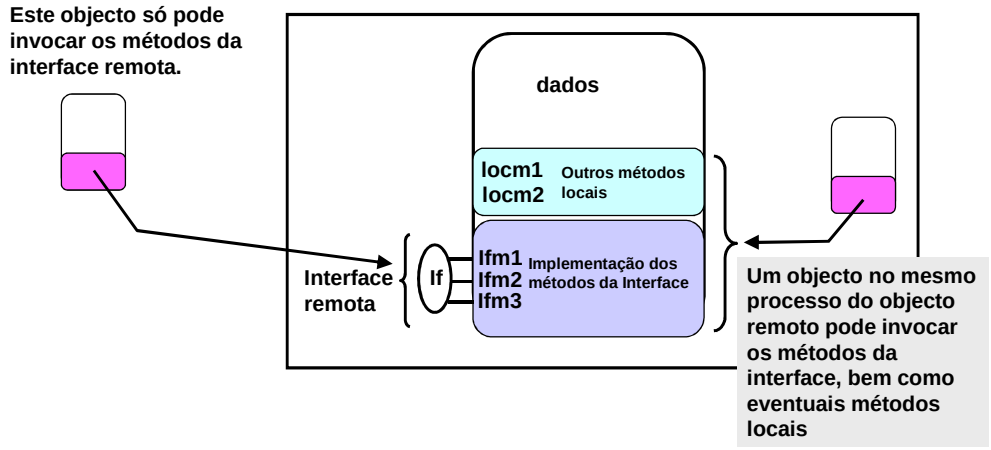
- **Permite que objectos subscrevam o seu interesse por eventos que possam ocorrer em objectos remotos e assim receberem notificações quando os eventos ocorram.**
- **Os eventos permitem que os objectos distribuídos comuniquem entre si de forma assíncrona.**
- **Suporte para *CallBacks***

Interfaces

- **Especificam um contrato entre objectos, isto é, definem a assinatura de um conjunto de métodos (nome do método, tipos dos argumentos, tipo de retorno) sem especificar a implementação;**
- **Um objecto disponibiliza uma interface se a sua classe contém código que implementa os métodos da interface**
- **Em CORBA IDL, Java RMI e em .NET uma classe pode implementar várias interfaces (múltipla herança de interfaces *);**
- **Uma interface pode também definir um tipo que pode ser usado para declarar variáveis, argumentos ou valores de retorno de métodos;**

* **Note que tanto em Java como em C# (.NET) não existe herança múltipla, isto é, uma classe só pode derivar de uma classe base. No entanto uma classe pode implementar múltiplas interfaces e uma interface pode derivar de outras interfaces.**

Activação dos métodos de uma Interface remota

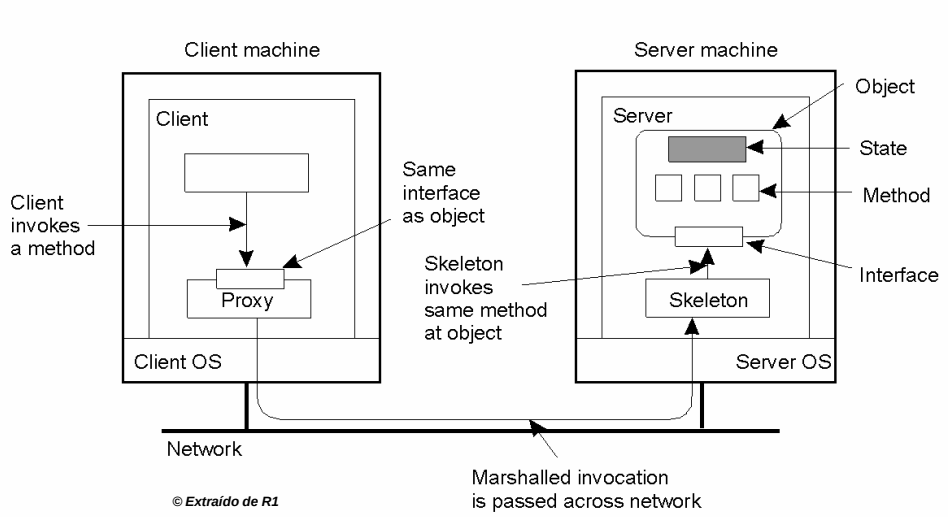


Exemplo CORBA IDL

```
// In file Person.idl                    © Extraído de B1
struct Person {
        string name;
        string place;
        long year;
} ;
interface PersonList {
        readonly attribute string listname;
        void addPerson(in Person p) ;
        void getPerson(in string name, out Person p);
        long number();
};
```

As classes dos objectos remotos podem ser implementados em qualquer linguagem (C++, Java etc.) para a qual exista um compilador de IDL.

Conceito de *proxy* e *skeleton* na invocação remota de métodos



Cliente (*Proxy*) / Servidor (*Dispacher e Skeleton*)

Proxy

- **Torna transparente a chamada de métodos no lado do cliente, comportando-se como um objecto local em representação do objecto remoto;**
- **Quando recebe uma invocação redirecciona-a através de uma mensagem para o objecto remoto, fazendo *marshal* do método e respectivos parâmetros**

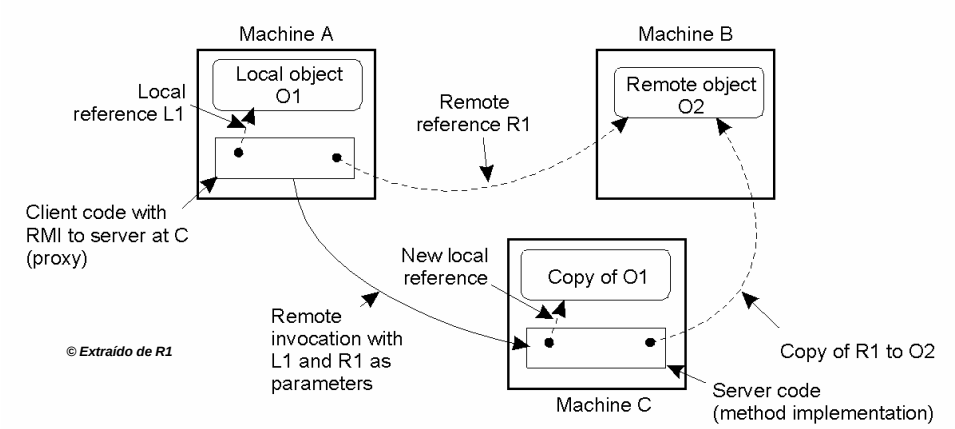
Dispacher

- **Recebe a mensagem do canal de comunicação, entregando-a ao *skeleton* da classe remota**

Skeleton

- **Cada classe de um objecto remoto tem um *skeleton* (estrutura) que implementa os métodos da interface por forma a que:**
  - **Faz *unmarshall* da mensagem (método e argumentos) com o pedido (enviada pelo proxy);**
  - **Chama o método respectivo no objecto remoto;**
  - **Espera que a chamada termine;**
  - **Faz *marshal* dos resultados, enviando uma mensagem de resposta ao *proxy* com os resultados ou eventuais excepções;**

Passagem de objectos como parâmetro por referência e por valor



- **O cliente na máquina A passa ao servidor na máquina C o objecto O1 por valor e a referência R1 para o objecto O2 que reside na máquina B.**
- **Salienta-se os desafios na implementação da passagem de referências para objectos (R1) entre máquinas;**

Semântica de invocação de métodos remotos versus falhas

- ***Exactly once*** **– Numa chamada é garantido que o método é executado uma única vez. A chamada de métodos locais tem esta semântica;**

Porém:

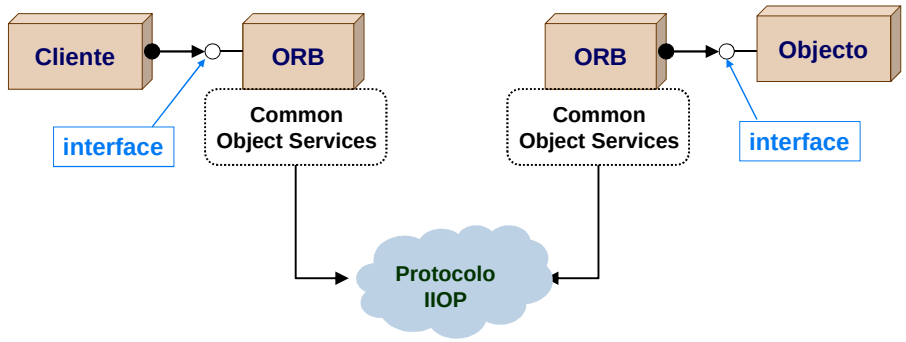
- **Em ambiente distribuído podem ocorrer falhas que podem ser mascaradas das seguintes formas:**
  - **Retransmitir a mensagem até receber uma resposta ou o objecto remoto assumir que houve uma falha (por exemplo *throw* de uma excepção);**
  - **Detectar/filtrar mensagens duplicadas do lado do objecto remoto;**
  - **Retransmitir resultados sem executar de novo o método no objecto remoto**

- ***Maybe*** **- Numa chamada não é possível saber se o método foi executado ou não.**
  - **Pode acontecer quando não são usados mecanismos de tolerância a falhas: perda de mensagens; falha do processo ou máquina que contém o objecto remoto.**

- ***At-least-once*** **– Numa chamada sabe-se que o método foi executado pelo menos uma vez, recebendo sempre um resultado ou uma excepção.**
  - **No caso de existir retransmissão de mensagens o método no objecto remoto pode ser executado mais que uma vez, podendo causar resultados inconsistentes;**
  - **Na presença desta semântica os métodos devem ser *idempotentes*, isto é, podem executar-se repetidamente causando o mesmo efeito.**

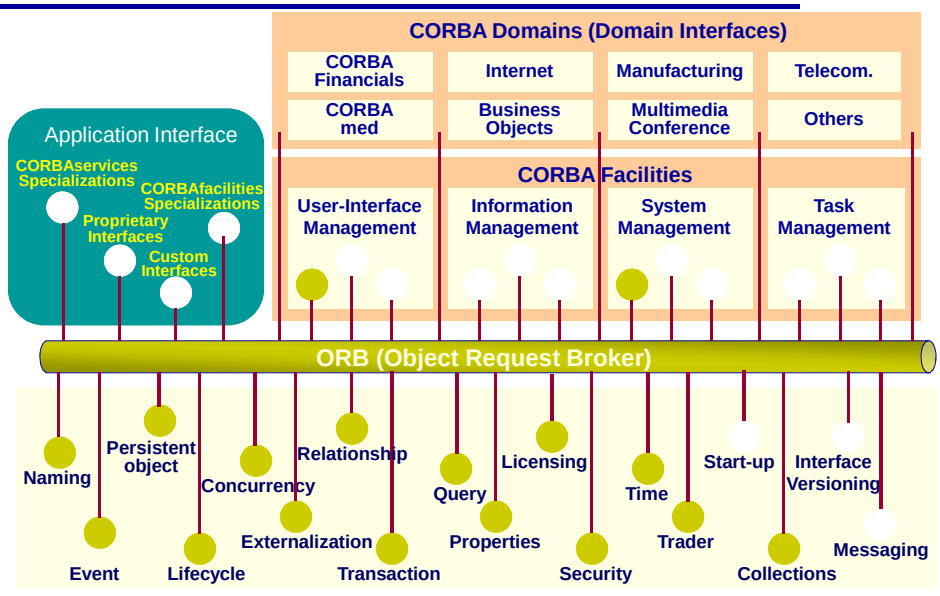
- ***At-most-once*** **– Numa chamada recebe-se sempre um resultado, sabendo-se que o método foi chamado uma só vez, ou foi recebida uma excepção.**
  - **Esta semântica obriga a usar formas de tratar falhas e suporte para a existência de mecanismos de *timeout*.**

Acesso a objectos remotos via CORBA

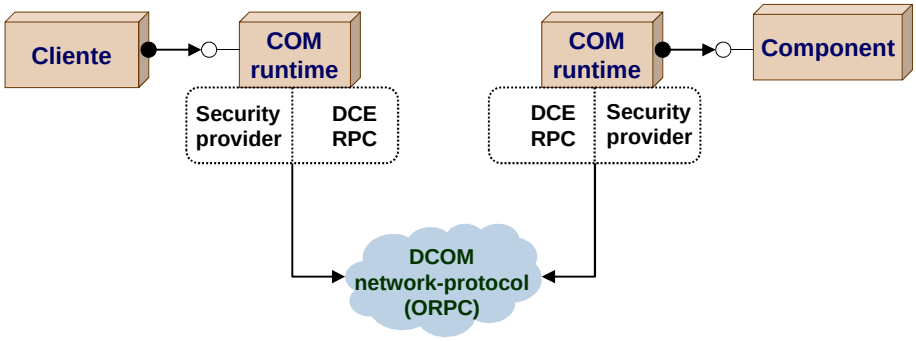


CORBA – Common Object Request Broker Architecture (inicio da década de 90)
OMG – Object Management Group
ORB – Object Request Broker (interoperabilidade entre objectos)
IDL – Interface Definition Language
GIOP – General Inter-ORB Protocol (interoperabilidade entre ORBs; Common Data representation (CDR); Formatos de mensagens e Gestão de conexões;
IIOP – Internet Inter-ORB Protocol

CORBA da *Object Management Architecture (OMA)*



Componentes (objectos!) remotos via DCOM

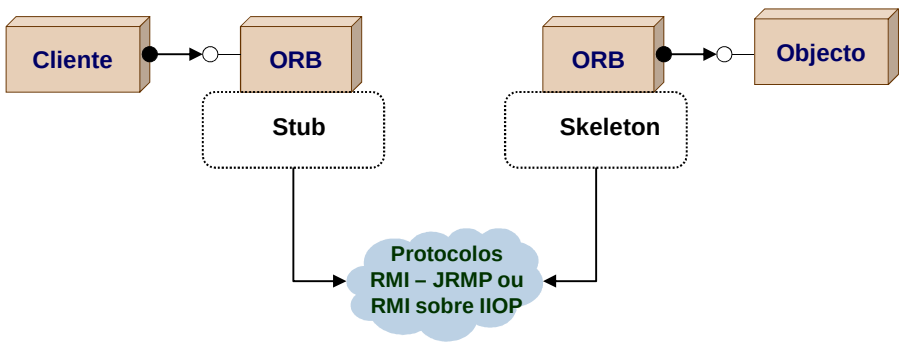


Protocolo DCOM, conhecido como *Object* RPC (ORPC), corresponde a um conjunto de definições que estendem o protocolo standard DCE RPC

Vulnerabilidades de CORBA/IIOP e COM/DCOM

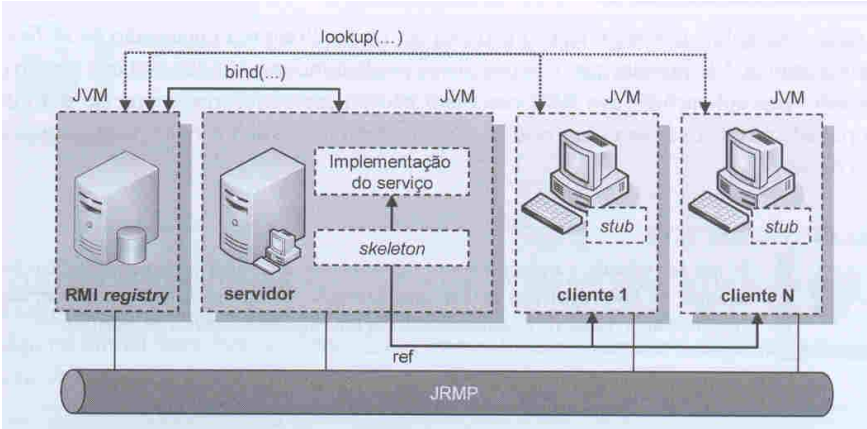
- Comunicação na base de formatos binários
  - Dificuldade em ultrapassar *Firewalls* mesmo que determinados portos possam ser configurados.
- Expansibilidade difícil
  - O desempenho depende da "qualidade ao nível da programação" principalmente quando está envolvido um elevado número de componentes/objectos - COM/DCOM ou CORBA/IIOP
- Dependência de linguagens específicas
  - COM/DCOM do sistema operativo Windows e linguagem C++/VB e o CORBA/IIOP da linguagem Java, embora existam ORBs para C++

Acesso a objectos remotos via RMI ou RMI sobre IIOP



JRMP – Java Remote Method Protocol
IIOP – Internet Inter-ORB Protocol

Aplicação Cliente/Servidor em Java RMI



© Extraído de: "Programação de Sistemas Distribuídos em Java", Jorge Cardoso, FCA-Editora de Informática, Lda

Aplicação Cliente/Servidor em Java RMI

Interface a partilhar entre Cliente e Servidor

```
package RMIInterface;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RMIInterface extends Remote {

    String sayHello(String name) throws RemoteException;

}
```

A interface `Remote` identifica os métodos que podem ser invocados remotamente. Qualquer objecto remoto tem directa ou indirectamente de implementar esta interface.

- Criar uma directoria com o nome do package `RMIInterface;`
- Copiar o resultado da compilação - RMIinterface.class – para essa directoria;
- Lançar o serviço de *registry*
  > start rmiregistry
- *Por omissão o registry executa-se no porto 1099. para executar noutro porto , por exemplo 2010 lançar > start rmiregistry 2010*

Seguindo o guião *ComoExecutar.txt,* executar e estudar o exemplo *ExemploRMI.zip*

Aplicação Servidor Java RMI

```
public class RMIServer implements RMIInterface {

  public RMIServer() {}

  public String sayHello(String name) {
        return "Hello, "+ name;
  }

  public static void main(String args[]) {

     try {
        RMIServer obj = new RMIServer(); // cria instância do servidor
        RMIInterface stub=(RMIInterface) UnicastRemoteObject.exportObject(obj, 0);

        // Bind the remote object's stub in the registry
        Registry registry = LocateRegistry.getRegistry();
        registry.bind("Hello Server", stub);  //regista stub com nome lógico

        System.out.println("Server ready");
     } catch (Exception e) {
           System.err.println("Server exception: " + e.toString());
           e.printStackTrace();
      }
  } //main
}
```

```
import RMIInterface.RMIInterface;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
```

Se o serviço *registry* estiver no porto 2010 efectuar :

```
Registry registry = LocateRegistry.getRegistry(2010);
```

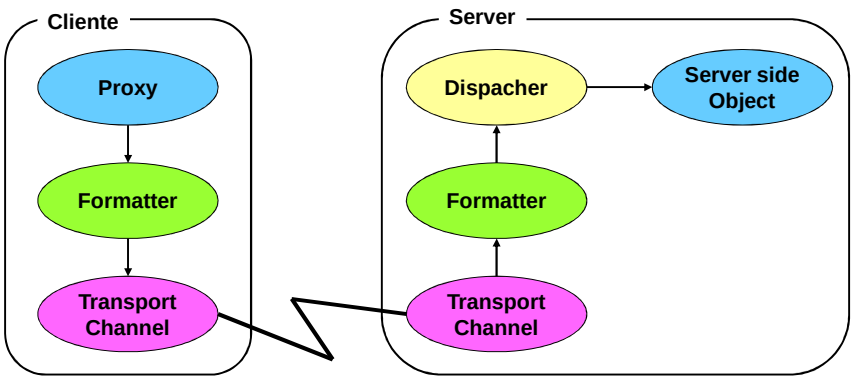

Aplicação Cliente Java RMI

```
import RMIInterface.RMIInterface;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Main {

   public static void main(String[] args) {

        try {
            Registry registry = LocateRegistry.getRegistry("localhost");
            RMIInterface stub = (RMIInterface) registry.lookup("Hello Server");

            String response = stub.sayHello("luis");

            System.out.println("response: " + response);
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

Se o serviço *registry* estiver no porto 2010 efectuar :

```
Registry registry = LocateRegistry.getRegistry("localhost", 2010);
```

Arquitectura simplificada do *.NET Remoting*



- O objecto remoto é representado no cliente por um objecto *proxy* que permite chamar os métodos de forma transparente;
- O *proxy* transforma as chamadas a métodos em mensagens que são passadas ao nível de seriação (*Formatter*) que as converte num formato especifico, por exemplo SOAP;
- A mensagem formatada é entregue ao nível de transporte que a transfere para o processo remoto, via protocolo TCP, HTTP, IPC ou outros.