
UFC

DEPARTAMENTO DE COMPUTAÇÃO

MDCC

GREat

# Sistemas Distribuídos

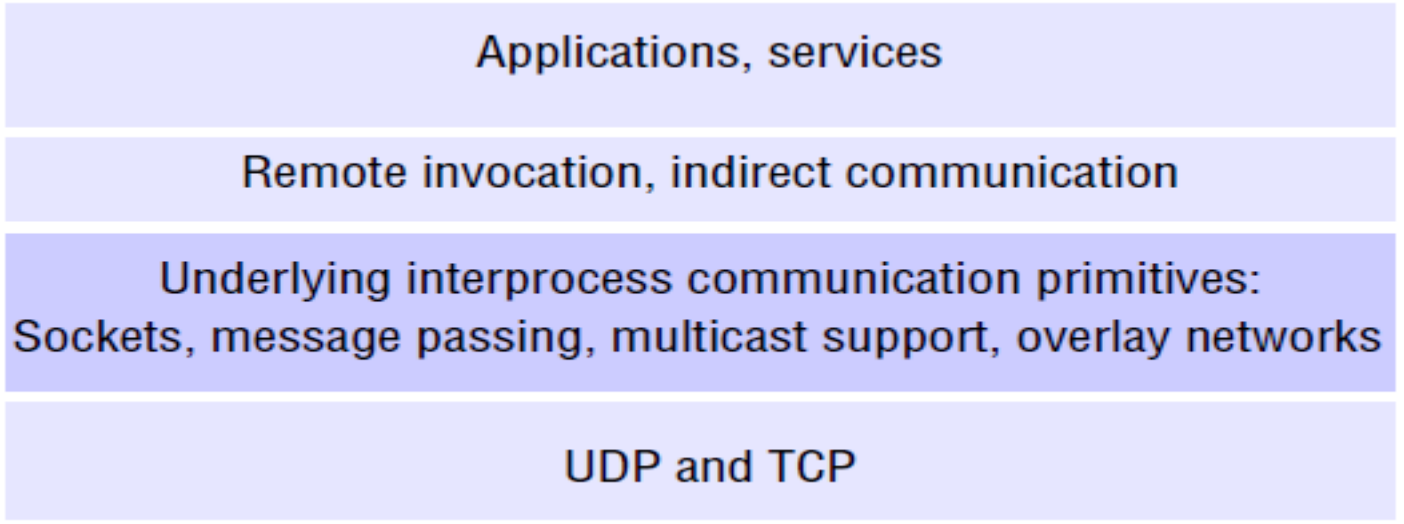
## Invocação Remota de Métodos

Fernando Trinta

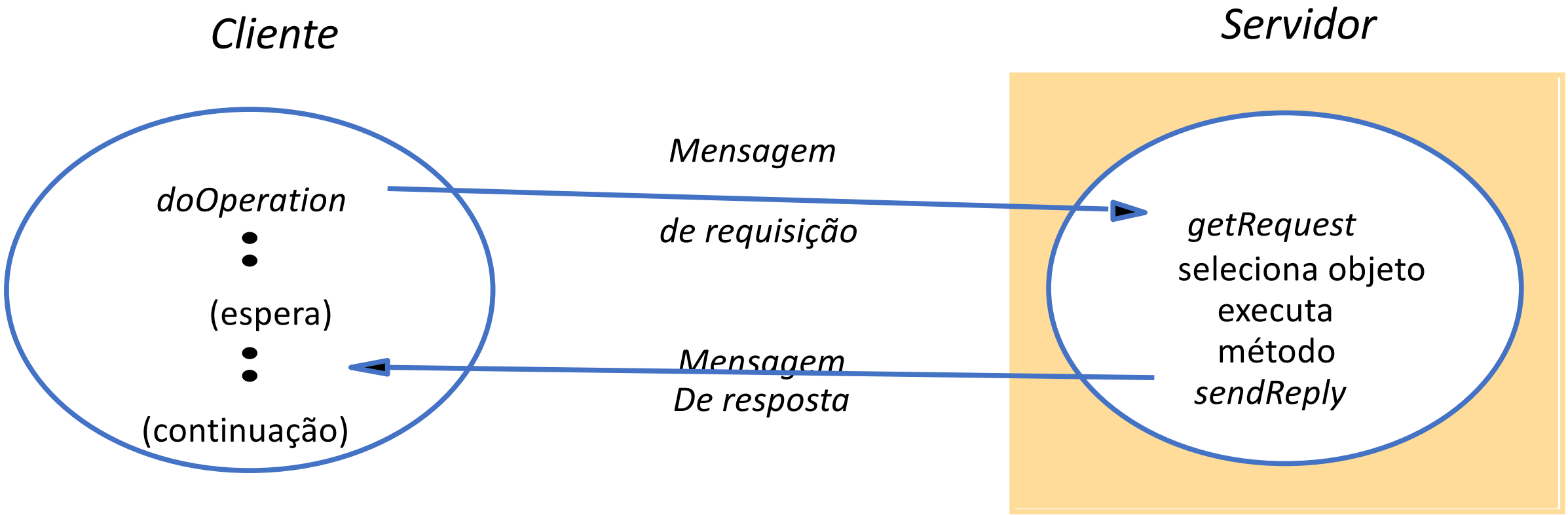
Windson Viana

# Paradigmas de Comunicação

- Invocação Remota
  - Sockets, RPC, RMI
  - Passagem de mensagens
  - Request-Reply
- Comunicação Indireta
  - Eventos (Publish-Subscriber)
  - Memória Compartilhada –Espaço de Tuplas
  - Multicasting



# Comunicação entre processos




Modelo de comunicação RPC/RMI

# TCP e UDP - Representação Externa de Dados

- Passagem de mensagens
  - Recebimento bloquante
- UDP – Não tem confirmação de erro de recebimento e nem garante ordem dos pacotes
  - Menor latência
- TCP – Garante ordem e controle de erro
  - Maior latência
- Exigência de uma representação externa e de transformação dos dados a serem transmitidos
  - Marshalling e Unmarshalling
  - XML, Serialização de Objetos, Protocolos de Middleware

# Exemplo de Serialização em Java

```
public class Nodo implements Serializable {
        Vector filhos;   Nodo pai;   String nome;
        public Nodo(String s) {
                filhos = new Vector(5);      nome = s;
        }
        public void addFilho(Nodo n) {
                filhos.addElement(n);      n.pai = this;
        }
}
```

# Exemplo de Serialização em Java

```
cliente = new Socket(host, port);
Nodo top = new Nodo("topo");
top.addFilho(new Nodo("filho1"));
top.addFilho(new Nodo("filho2"));
ObjectOutput out = new ObjectOutputStream(cliente.getOutputStream());
out.writeObject(top);
out.flush();
```

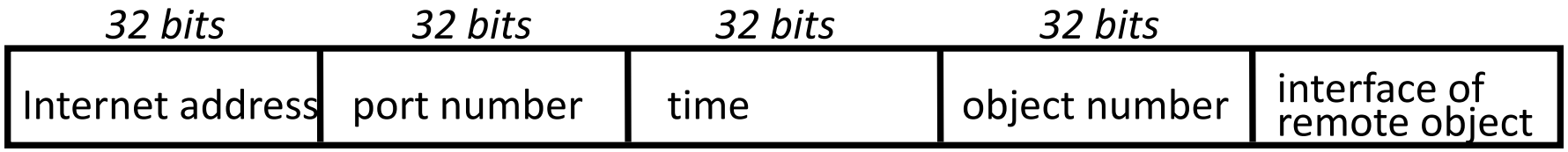
# Exemplo de Serialização em Java

```
cliente = new Socket(host, port);
Nodo top = new Nodo("topo");
top.addFilho(new Nodo("filho1"));
top.addFilho(new Nodo("filho2"));
ObjectOutput out =  new ObjectOutputStream(cliente.getOutputStream());
out.writeObject(top);
out.flush();
```

# Exemplo de Serialização em Java

```
socket = server.accept();
out = new ObjectOutputStream(socket.getOutputStream());
in = new ObjectInputStream(socket.getInputStream());
Nodo n = (Nodo)in.readObject();
n.addFilho(new Nodo("servidor"));
out.writeObject(n);
out.flush();
```

# Como endereçar objetos remotos?



# Detalhamento da chamada

*public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)*

- Envia uma mensagem de requisição para um objeto remoto e retorna resposta. Os argumentos especificam o objeto remoto, o método a ser invocado e os argumentos desse método.

*public byte[] getRequest ();*

- Lê uma requisição de cliente por meio da porta de servidor.

*public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);*

- Envia a mensagem de resposta para o cliente em seu endereço IP e porta.

# Detalhamento do Método

| | |
|---|---|
| messageType | *int (0=Request, 1= Reply)* |
| requestId | *int* |
| remoteReference | *RemoteRef* |
| operationId | *int or Operation* |
| arguments | *array of bytes* |

# O que muda em relação à uma chamada local?

- Identificadores de mensagem;
- Modelo de falhas do protocolo requisição-resposta;
- Timeouts;
- Descartando mensagens de requisição duplicadas;
- Perda de mensagens de resposta;
- Histórico.

# Exemplo de Histórico

| *Nome* | *Mensagens enviadas pelo* | | |
|---|---|---|---|
| | *Cliente* | *Servidor* | *Cliente* |
| R | *Request* | | |
| RR | *Request* | *Reply* | |
| RRA | *Request* | *Reply* | *Acknowledge reply* |

# Requisição-Resposta (Request-Reply)

Request-reply communication

Request-reply message structure

| | |
|---|---|
| messageType | *int (0=Request, 1= Reply)* |
| requestId | *int* |
| remoteReference | *RemoteRef* |
| operationId | *int or Operation* |
| arguments | *// array of bytes* |

doOperation bloqueia a execução até o recebimento do reply
getRequest é método invocado pelo servidor para descobrir e executar a operação
sendReply é método invocado para enviar o resultado da operação

# HTTP

**Figure 5.6** HTTP *Request* message

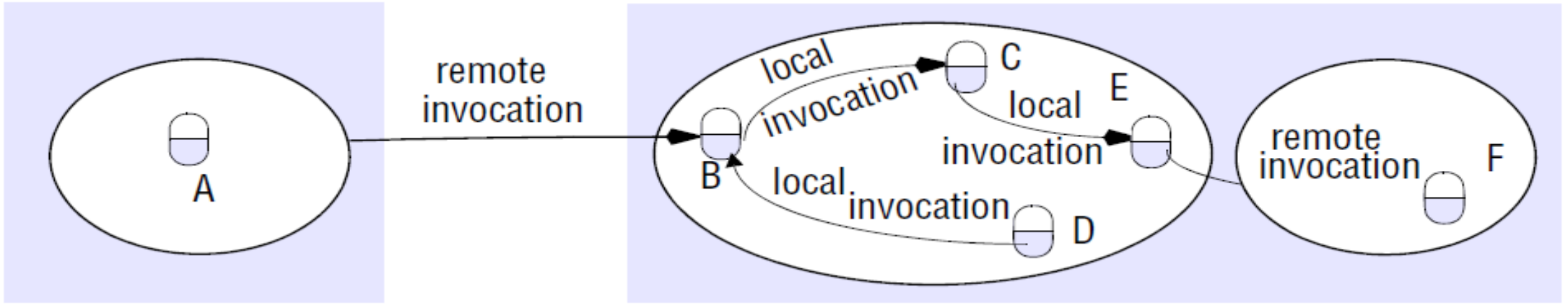
| *method* | *URL or pathname* | *HTTP version* | *headers* | *message body* |
|---|---|---|---|---|
| GET | http://www.dcs.qmul.ac.uk/index.html | HTTP/ 1.1 | | |

HTTP *Reply* message

| *HTTP version* | *status code* | *reason* | *headers* | *message body* |
|---|---|---|---|---|
| HTTP/1.1 | 200 | OK | | resource data |

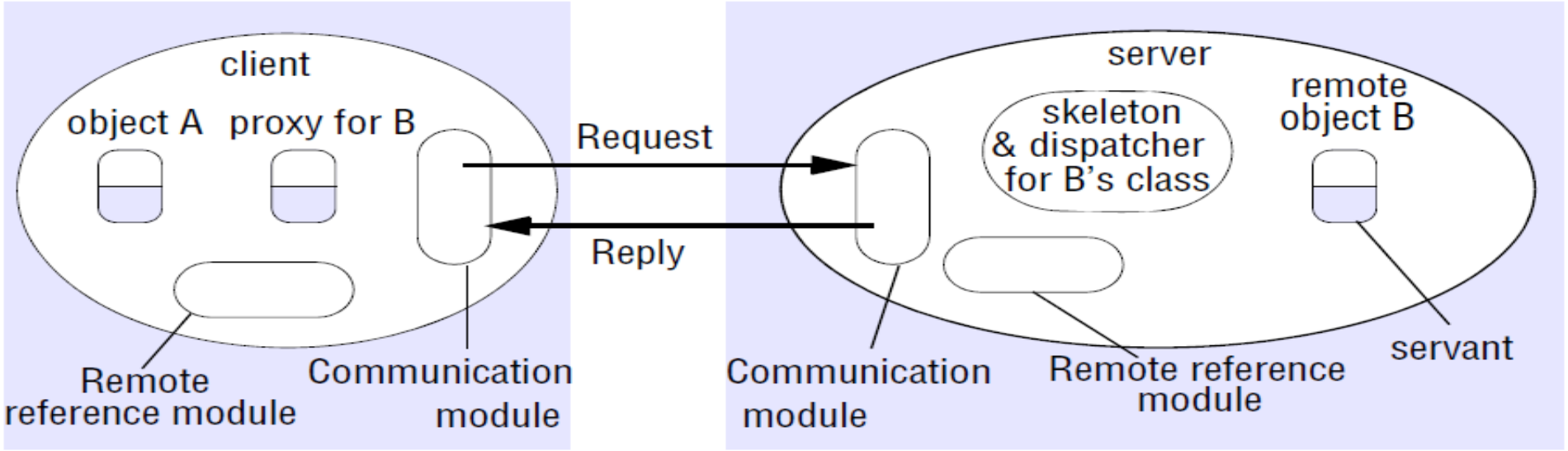
# RMI - Remote Method Invocation

- Procedimento passível de invocação remota
  - Uso de uma IDL (Interface Description Language) para descrever a interface
- Referências via stubs
  - doOperation, getRequest and sendReply



# RMI - Remote Method Invocation

- Procedimento passível de invocação remota
  - Uso de uma IDL (Interface Description Language) para descrever a interface
- Referências via stubs
  - doOperation, getRequest and sendReply



# Interfaces em SD

- Os mecanismos de passagem de parâmetros (por exemplo, chamada por valor e chamada por referência) não são convenientes para processos diferentes.

- Variáveis do tipo ponteiros de um processo não são válidas em outro processo remoto.

# Interfaces usadas no RPC e RMI

- Interfaces de serviço: no modelo cliente-servidor, cada servidor fornece um conjunto de procedimentos que estão disponíveis para uso pelos clientes.

- Interfaces remotas: no modelo de objeto distribuído, uma interface remota especifica os métodos de um objeto que estão disponíveis para invocação por parte dos objetos de outros processos.

# Linguagem de Definição de Interface

- As linguagens de definição de interface (IDLs) são projetadas para permitir que objetos implementados em linguagens diferentes invoquem uns aos outros.

- Uma IDL fornece uma notação para definir interfaces, na qual cada um dos parâmetros de um método pode ser descrito como sendo de entrada ou saída, além de ter seu tipo especificado.

# Exemplo

```
// Arquivo de entrada Person.idl
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();
};
```

# Interfaces

```
package com.marakana;

// Declare the interface.
interface IAdditionService {
//You can pass values in, out, or inout.
// Primitive datatypes (such as int, boolean, etc.) can only be passed in.
int add(in int value1, in int value2);
}


//Arquivo AIDL
```

# Exemplo - WSDL

```
▼<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tm="http://microsoft.
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:mime="http://schemas.xmlsoap.org
 xmlns:s="http://www.w3.org/2001/XMLSchema" xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" targetNamespace="http://www.webserviceX.NET">
  ▼<wsdl:types>
    ▼<s:schema elementFormDefault="qualified" targetNamespace="http://www.webserviceX.NET">
      ▼<s:element name="GetWeather">
        ▼<s:complexType>
          ▼<s:sequence>
              <s:element minOccurs="0" maxOccurs="1" name="CityName" type="s:string"/>
              <s:element minOccurs="0" maxOccurs="1" name="CountryName" type="s:string"/>
            </s:sequence>
          </s:complexType>
        </s:element>
      ▼<s:element name="GetWeatherResponse">
        ▼<s:complexType>
          ▼<s:sequence>
              <s:element minOccurs="0" maxOccurs="1" name="GetWeatherResult" type="s:string"/>
            </s:sequence>
          </s:complexType>
        </s:element>
      ▼<s:element name="GetCitiesByCountry">
        ▼<s:complexType>
          ▼<s:sequence>
              <s:element minOccurs="0" maxOccurs="1" name="CountryName" type="s:string"/>
            </s:sequence>
          </s:complexType>
        </s:element>
```

# Comunicação entre objetos distribuídos

# Conceitos Comuns com Orientação a Objetos

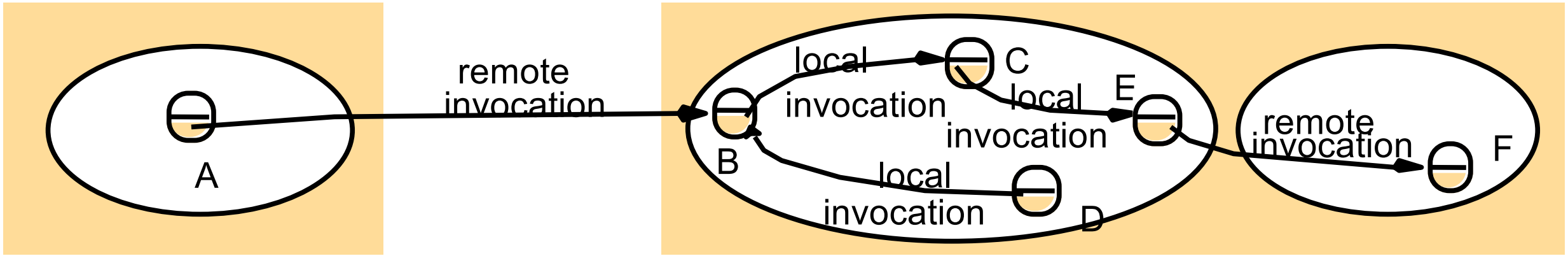
- Referências a objetos
- Interfaces
- Ações executadas por métodos
- Exceções
- Garbage collector
  - O que deveria mudar ou ser estendido em um SD?

# Comunicação entre objetos distribuídos

- Os sistemas de objetos distribuídos podem adotar a arquitetura cliente-servidor. Nesse caso, os objetos são gerenciados pelos servidores e seus clientes invocam seus métodos usando invocação e método remoto, RMI.

# Comunicação entre objetos distribuídos

- Cada processo contém um conjunto de objetos, alguns dos quais podem receber invocações locais e remotas.

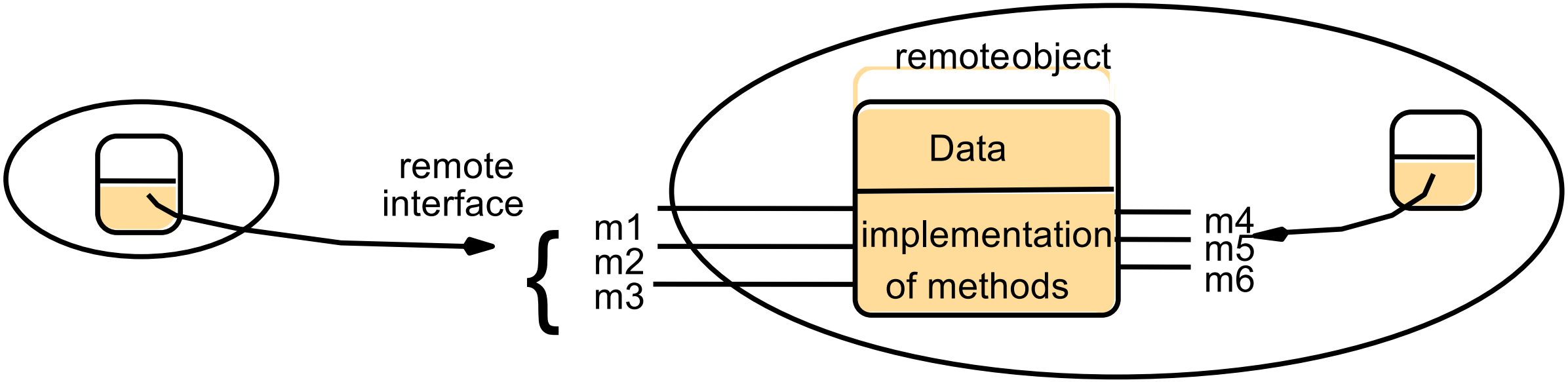


# Comunicação entre objetos distribuídos

- Referências de objeto remoto:
  - A noção de referência de objeto é estendida para permitir que qualquer objeto que possa receber uma RMI tenha uma referência de objeto remoto

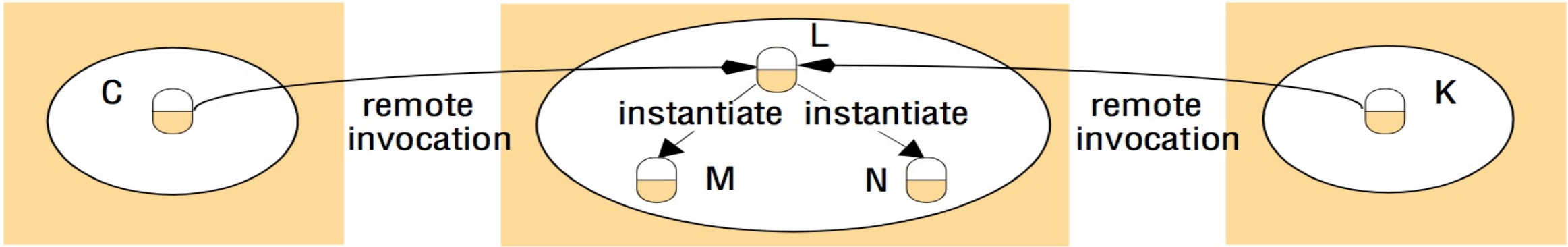
- Interfaces remotas:
  - A classe de um objeto remoto implementa os métodos de sua interface remota, por exemplo, como métodos

# Comunicação entre objetos distribuídos



# Comunicação entre objetos distribuídos

- Ações em um sistema de objeto distribuído:
  - Uma ação é iniciada por uma invocação a  método, a qual pode resultar em mais invocações a métodos em outros objetos e inclusive a criação de novos objetos.



# Comunicação entre objetos distribuídos

- Coleta de lixo em um sistema de objeto distribuído:
  - Se uma linguagem, por exemplo Java, suporta coleta de lixo, então qualquer sistema RMI associado deve permitir a coleta de lixo de objeto remotos.

- Exceções:
  - A invocação a método remoto deve ser capaz de lançar exceções, como timeouts, causados pelo envio de mensagens.

# Semântica de Invocação RMI

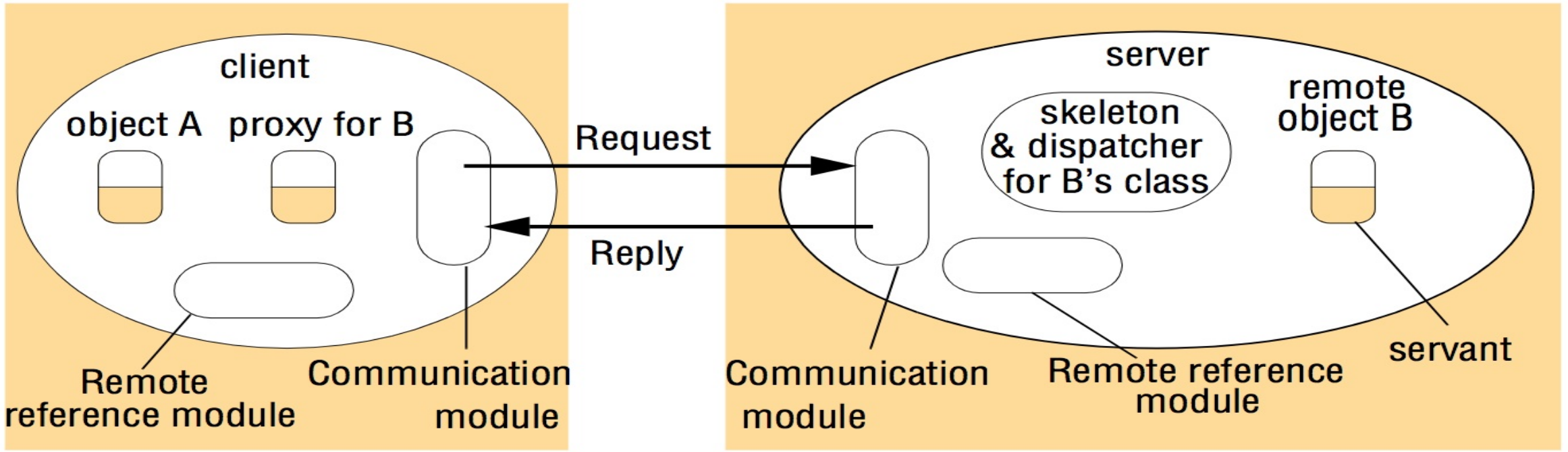
- Semântica de invocação talvez
  - O método remoto pode ser executado uma vez ou não ser executado.
- Semântica de invocação pelo menos uma vez
  - o invocador recebe um resultado quando o método foi executado pelo menos uma vez, ou recebe uma exceção.
- Semântica de invocação no máximo uma vez
  - Ideal para operações que não são idempotentes

# Semântica de Invocação RMI

- Entretanto, as invocações remotas são mais vulneráveis à falhas do que as locais, pois envolvem a rede, outro computador e outro processo. Qualquer que seja a semântica de invocação escolhida, há sempre a chance de que nenhum resultado seja recebido.

# Implementação em RMI

- Vários objetos e módulos separados estão envolvidos na realização de uma invocação a método remoto.



# Implementação de RMI

- Proxy: a função de um proxy é tornar a invocação a método remoto transparente para os clientes, comportando-se como um objeto local para o invocador.

- Despachante: um servidor tem um despachante e um esqueleto (Skeleton) para cada classe que representa um objeto remoto.

- Servant: o objeto real compartilhado

# Algumas implementações RMI

- RMI Java
  - http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html
- DRB – Distributed System for Ruby
  - http://ruby-doc.org/stdlib-1.9.3/libdoc/drb/rdoc/DRb.html
- .NET Framework
  - https://msdn.microsoft.com/en-us/library/kwdt6w2k(VS.85).aspx
- PYRO (PYthon Remote Objects)
  - https://pythonhosted.org/Pyro/1-intro.html
- RMI.js Javascript RMI
  - https://github.com/mmarcon/rmi.js

# Java RMI

# Alguns conceitos

- Conceitos e Exemplo de Código
- Interface Remote
- RMIRegistry
- SecurityManager
- Naming and Binding
- Callback

# Naming & Binding

- Naming:
  - Forma pela qual os objetos remotos associam nomes (identificadores) a si mesmos, fazendo que clientes possam procurá-los através destes nomes

- Binding:
  - Maneira pela qual clientes conseguem as referências de objetos remotos e podem, então, chamar seus métodos

# Metódo convencional

- Uso de um servidor (serviço) de nomes
  - Objetos servidores registram suas referências no SN de modo que objetos clientes possam encontrá-los e utilizarem seus serviços
  - Objetos clientes acessam este serviço e buscam as referências de seus servidores através de um identificador (nome)
  - Após encontrar as referências, começa a interação entre cliente e servidor

# Metódo convencional

- Uso de um serviço de nomes
  - rmiregistry
- Objetos servidores registram-se neste servidor
  - Extender a classe UnicastRemoteObject
  - Naming.rebind(String, Objeto)
- Objetos clientes buscam referências de servidores no SN;
  - Naming.lookup(String)

## Porém, existem outras formas.

- rmiregistry utiliza um classe definida na API java
  - Classe java.rmi.registry.Registry
  - Interface que implementa as funções de bind(), unbind(), lookup() e list();
- Com isso, é possível dentro de um código Java, lançar o Serviço de Nomes
  - Uso da classe java.rmi.registry.LocateRegistry
  - Classe que serve para criar ou obter a referência a um SN, um Registry

# LocateRegistry

- Para criar o Registry
  - LocateRegistry.createRegistry(int Porta);

- Para obter o Registry (caso este já tenha sido criado)
  - LocateRegistry.getRegistry(1099);

# Usando a Classe LocateRegistry

```
Registry reg=null;
try {
   System.out.println("Creating registry...");
   reg = LocateRegistry.createRegistry(1099);
 } catch(Exception e){
   try {
     reg = LocateRegistry.getRegistry(1099);
          } catch(Exception e){  System.exit(0); }
 }
```

# Classe UnicastRemoteObject

- Classe que implementa um objeto servidor não replicado
  - Faz a implementação de vários métodos (hashCode, equals, toString) para Objetos Remotos
- Porém, não obrigatoriamente é necessário estender tal classe
  - Importante é chamar o método estático exportObject(), geralmente no construtor da classe;

**RemoteStub rstub = UnicastRemoteObject.exportObject(objetoRemoto)**

  - O parâmetro deve direta ou indiretamente implementar a interface java.rmi.Remote
- O Objeto é colocado numa porta anônima (acima de 1023)
  - Caso queira-se colocar o objeto em uma porta específica, deve-se usar outra versão do método exportObject()
    - exportObject(ObjetoRemoto, Porta)
    - Geração automática do STUB

# Ainda sobre UnicastRemoteObject

- o objeto não está colocando uma referência sua para acesso no SN
- O método exportObject():

  - Coloca o skeleton associado ao Objeto em um porta para ser usado por clientes;
  - O stub retornado pelo método pode ser utilizando para registrar o Objeto no serviço de nomes

# Stub

- O "stub" funciona semelhante a um proxy para o objeto remoto. Quando um objeto local invoca um método num objeto remoto, o "stub" fica responsável por enviar a chamada ao método para o objeto remoto.
- Passos do "stub" quando é invocado
  - Iniciar conexão com a "Virtual Machine" que contém o objeto remoto.
  - Escrever e transmitir os parâmetros para a "Virtual Machine" remota.
  - Esperar pelos resultados da invocação do método.
  - Ler os resultados retornados.
  - Retornar os valores ao objeto que executou a chamada .

# Skeletons

- Na "Virtual Machine" remota, cada objeto deve ter um "skeleton" correspondente ao "stub". O "skeleton" é responsável por enviar a chamada ao objeto remoto.
- Passos do "skeleton" quando recebe uma chamada:
  - Ler os parâmetros enviados pelo "stub"
  - Invocar o método no objeto remoto
  - Escrever e transmitir o resultado ao objeto que executou a chamada

# Classe Naming

- Classe estática responsável pela ligação entre objetos clientes e servidores, através do Serviço de Nomes
- Principais métodos
  - bind(String name, Remote obj)
  - list(String name)
  - lookup(String name)
  - rebind(String name, Remote obj)
  - unbind(String name)

# Classe Naming

```
public static void rebind(String url, Remote obj) {
        Registry reg=LocateRegistry.getRegistry();
        Registry reg=LocateRegistry.getRegistry(url,port);
        reg.rebind(url,obj);
}
```

# Unindo as partes

- //Classe compartilhada pelo cliente e pelo servidor

```
public interface Hello extends Remote {
   String sayHello() throws RemoteException;
}

//Objeto remoto real
public class Server implements Hello {
   public Server() {}
   public String sayHello() {
  return "Hello, best students are in SMD!";
   }
```

# Unindo as partes - Servidor

```
Server obj = new Server();
Hello stub = (example.hello.Hello) UnicastRemoteObject.exportObject(obj,0);
Registry reg=null;
try {
      System.out.println("Creating registry...");
      reg = LocateRegistry.createRegistry(1099);
} catch(Exception e){
      try {
            reg = LocateRegistry.getRegistry(1099);
      } catch(Exception ee){  System.exit(0); }
}
reg.rebind("HelloService", stub);
```

# Unindo as partes - Cliente

```
tring host="localhost";
try {
   Registry registry = LocateRegistry.getRegistry(host);
   Hello stub = (Hello) registry.lookup(“HelloService");
   if (stub!=null){
   String response = stub.sayHello();

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

# Exercício

- Implementem o Cliente e o Servidor dos slides
  - http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html
  - http://www.devmedia.com.br/uma-introducao-ao-rmi-em-java/28681
  - https://docs.oracle.com/javase/tutorial/rmi/overview.html

- Implemente um objeto que captura informações sobre o computador em que executa (IP, Versão do S.O.; ....). Em seguida, construa um cliente RMI para ele
  - InetAddress
    - http://download.java.net/jdk7/archive/b123/docs/api/java/net/InetAddress.html
  - System Properties
    - https://docs.oracle.com/javase/tutorial/essential/environment/sysprop.html

- Crie um cascateamento de objetos
  - A invoca B que invoca C onde B e C são objetos remotos

# Introdução a Callbacks

# Cenário tradicional

- Aplicações convencionais
  - Objeto cliente, outro servidor
- Papéis bem definidos
  - One-way interaction (Interação em único sentido)

# Cenários mais complexos

- Os papéis de cliente ou servidor não são muito claros
- Inversão de papéis
  - Servidores podem necessitar fazer chamadas de métodos em seus clientes
- Exemplos:
  - Relatório de erros ou problemas;
  - Atualização periódica & relatório de progresso;
  - Notificação de eventos
- Mudança para a chamada "2-way interactions"
  - Interação em dois sentidos

# Client-side Callback

- Define-se "Callback", a invocação de um método do objeto cliente por parte do, outrora, objeto servidor

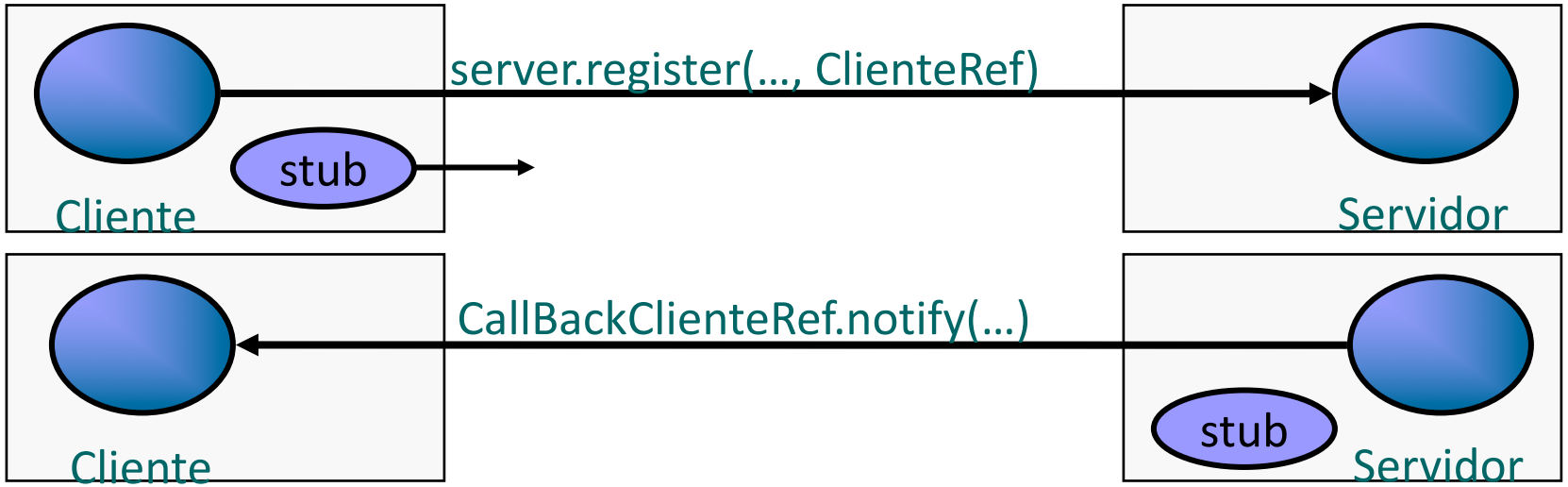
# Suporte para Callback

- Basicamente, são necessários dois passos:
- Passo 1: "cliente" deve transformar-se em "servidor"

  - Definição de interfaces do cliente
  - Devem-se ser criados stubs e skeletons (dinâmico a partir da versão 5.0 do Java)
  - Registrar-se num serviço de nomes para se habilitar a receber chamadas

# Suporte para Callback

- 20. Passo: "cliente" deve fornecer ao servidor sua referência,



# CallBack em Java RMI

- Regras:
  - Cliente deve definir uma interface de seus serviços que implemente a Interface java.rmi.Remote
  - Cliente deve definir sua implementação que implemente a interface anteriormente definida
  - Cliente torna-se disponível como servidor (exportar sua interface como um servidor remoto):
    - Estender a classe UnicastRemoteObject
    - Usar o método UnicastRemoteObject.exportObject(Remote)
  - Passar a referência ao servidor

# Tarefas de casa em RMI

- Uma aplicação com a seguinte arquitetura:
  - Um servidor de mensagens (um chat);
  - Clientes registram-se no servidor para receber mensagens;
  - Clientes cadastram mensagens neste servidor;
  - Servidor envia as mensagens cadastradas pelos clientes para todos os cadastrados;
  - Servidor detecta a queda ou saída dos clientes