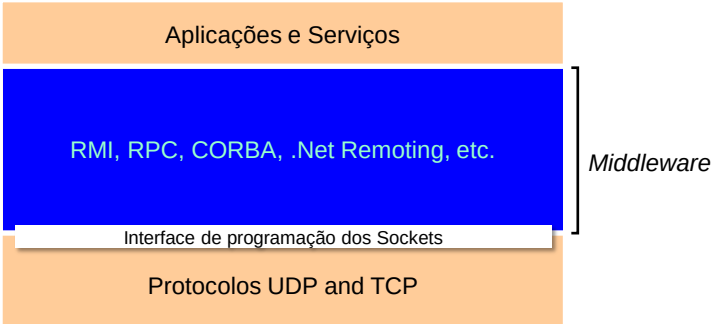
# Invocação Remota de Métodos



## Introdução

- Soluções de *middleware* para sistemas distribuídos
  - Abstração dos desenvolvedores relativamente a vários aspetos
    - Troca de mensagens/comunicação
    - Protocolos de comunicação
    - Sistemas operativos
    - *Hardware*
  - Transparência de localização
  - Algumas soluções suportam várias linguagens de programação

# Introdução

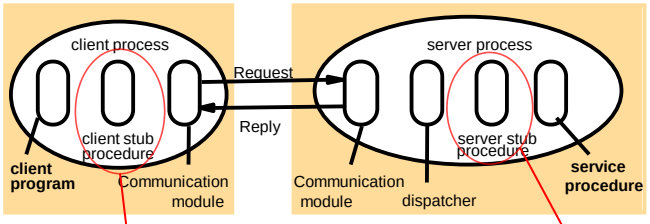





# RPC

- *Remote Procedure Call* (RPC)
- Precursor dos *middleware* para sistemas distribuídos
- O RPC da Sun Microsystems (RFC 1057) é a solução mais conhecida
  - Recorre ao UDP por omissão, mas pode ser realizado sobre qualquer protocolo do nível de transporte
  - Porto do *Port Mapper*: 111 (sunrpc)
  - Os dados trocados, quer pertençam a tipos simples ou estruturados, seguem o formato de representação externa XDR (*e**X**ternal **D**ata **R**epresentation*) da Sun
  - São usados identificadores numéricos em vez de nomes
  - Suporta várias linguagens de programação

# RPC




Um para cada procedimento exportado.
Atua como proxy, i.e., oferece a interface de programação pretendida (procedimentos), mas não realiza a tarefa correspondente.

Invoca o procedimento e devolve o resultado



# RPC

Ficheiro **add.x**

```
struct intpair { int a; int b; };
program ADD_PROG {
        version ADD_VERS {
                int ADD(intpair) = 1;
        } = 1;
} = 0x23451111;
```

*Identificador do procedimento*

*Versão*

*Identificador do servidor remoto*

**rpcgen –C –a add.x**

*Rotinas XDR: conversão dos dados declarados do formato de representação local para XDR e vice-versa*

| *Header file* | *Server stub (skeleton)* | *Client stub (proxy)* | |
|---|---|---|---|
| **add.h** | **add_svc.c** | **add_clnt.c** | **add_xdr.c** |
| **add_client.c** | **add_server.c** | **makefile.add** | |
| *Código exemplo de um cliente* | *Código exemplo de um servidor* | | |

# RPC

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "add.h"

void add_prog_1(char *host) {
    CLIENT *clnt;
    int  *result_1;
    intpair  add_1_arg;
    int r1, r2;

#ifndef  DEBUG
    clnt = clnt_create (host, ADD_PROG, ADD_VERS, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif   /* DEBUG */

        printf("Valor de a: ");
        r1 = scanf("%d", &(add_1_arg.a));
        printf("Valor de b: ");
        r2 = scanf("%d", &(add_1_arg.b));
```

*Em bold: código acrescentado ao que foi gerado pelo rpcgen no ficheiro "add_client.c".*



# RPC

```
        if(r1==1 && r2==1){
                result_1 = add_1(&add_1_arg, clnt);
                if (result_1 == (int *) NULL) {
                        clnt_perror (clnt, "call failed");
                }
                printf("Resultado: %d\n", *result_1);
        }else{
                printf("Deve apenas indicar valores do tipo inteiro!\n");
        }

#ifndef  DEBUG
    clnt_destroy (clnt);
#endif    /* DEBUG */
}

int main (int argc, char *argv[]) {
    char *host;

    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    add_prog_1 (host);
    exit (0);
}
```

*Em bold: código acrescentado ao que foi gerado pelo rpcgen no ficheiro "add_client.c".*

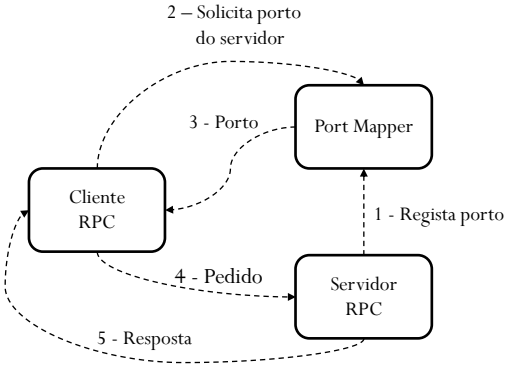
# RPC

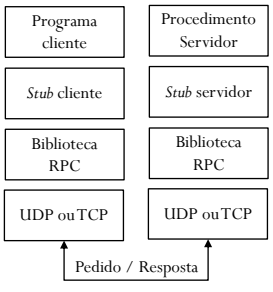
```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "add.h"

int * add_1_svc(intpair *argp, struct svc_req *rqstp) {
    static int  result;

    /*
     * insert server code here
     */
    result = argp->a + argp->b;

    printf("Funcao add invocada: %d + %d = %d\n", argp->a, argp->b, result);
    fflush(stdout);

    return &result;
}
```

*Em bold: código acrescentado ao que foi gerado pelo rpcgen no ficheiro "add_server.c".*

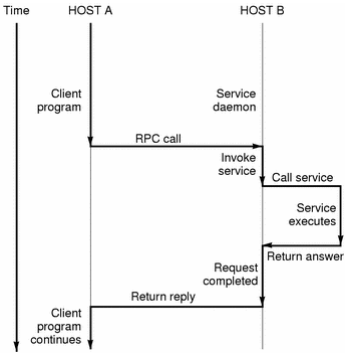


# RPC



# RPC




https://docs.oracle.com/cd/E19455-01/805-7224/6j6q44qgg/index.html



# RPC

- Estrutura das mensagens de pedido e resposta RPC (sobre UDP)

| Campo | # Bytes |
|---|---|
| Cabeçalho IP | 20 |
| Cabeçalho UDP | 8 |
| Identificador de transação | 4 |
| … | |
| Identificador do programa | 4 |
| Identificador da versão | 4 |
| Identificador do procedimento | 4 |
| … | |
| Parâmetros | N |

## Introdução ao Java RMI

- **Java RMI** (*Remote Method Invocation*): permite que uma máquina virtual java invoque métodos em objectos situados em outras máquinas virtuais
- Tecnologia de ***middleware*** que permite invocar métodos em objectos remotos tão facilmente quanto em objetos locais
- Os métodos invocáveis remotamente são descritos através de interfaces
- As interfaces são usadas no desenvolvimento das aplicações que pretendem aceder aos objectos remotos



## Introdução ao Java RMI

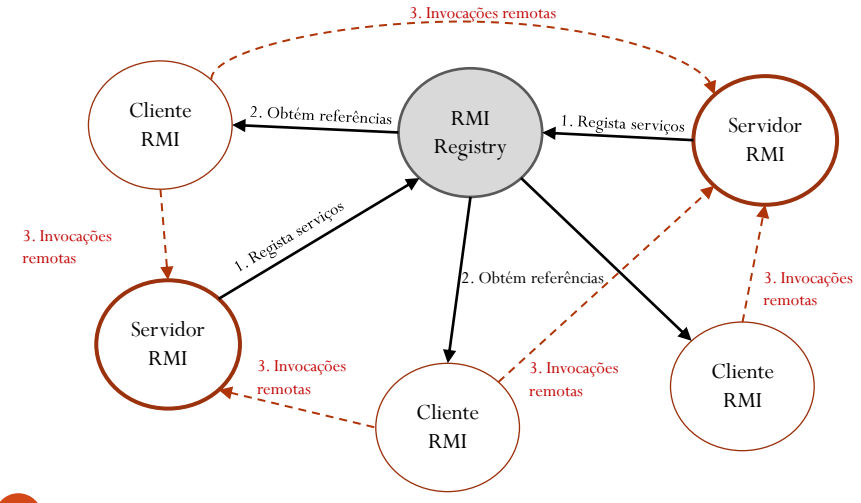
- Podem existir várias realizações práticas para uma mesma interface
- O Java RMI apenas permite desenvolver aplicações distribuídas em Java
- Outros sistemas, tais como o CORBA (*Common Object Request Broker Architecture*), suportam várias linguagens de programação

# Arquitetura

- **Servidores**: alojam os serviços RMI (i.e., objectos remotos)
- **Clientes**: invocam os serviços RMI
- O servidores podem registar-se num **serviço de diretório** que, posteriormente, é acedido pelos clientes para obter referências para os serviços/objectos pretendidos
- **RMI Registry**: aplicação que, depois de posta a correr, permite o registo de servidores RMI e a pesquisa de serviços RMI através dos respetivos nomes → **serviço de diretório**
- A identificação dos objetos remotos através de nomes permite incrementar a tolerância a falhas do sistema (i.e., transparência de localização)



# Arquitetura

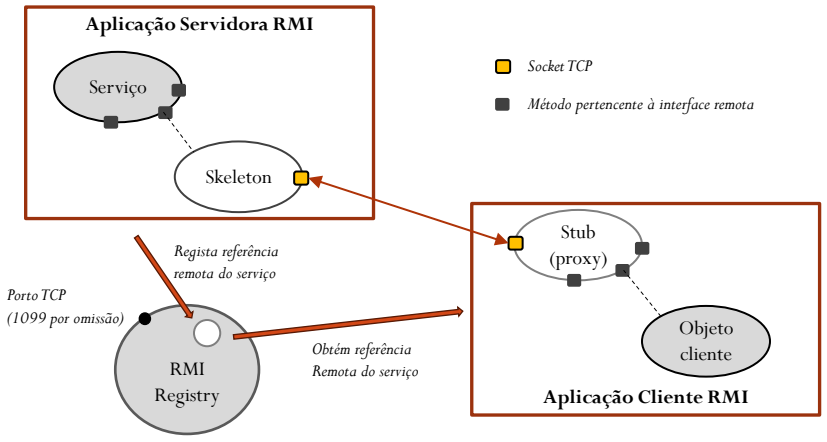


# Arquitetura

- Depois de obter a referência para um serviço/objeto, um cliente pode acedê-lo
- Para obter a referência de um objeto remoto (porto por omissão do serviço de diretório: 1099):
  *rmi://registryname:port/servicename*
- Os pormenores de rede/comunicação envolvidos no acesso aos objectos remotos é transparente para os programadores das aplicações RMI
- A transparência é conseguida à custa de dois objetos
  - ***Stub*** (integra o cliente e implementa a interface remota)
  - ***Skeleton*** (integra a aplicação servidora)

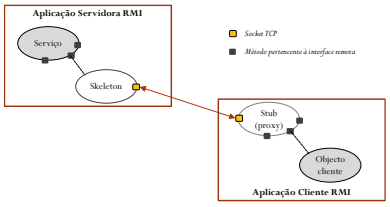


# Arquitetura



# Arquitetura

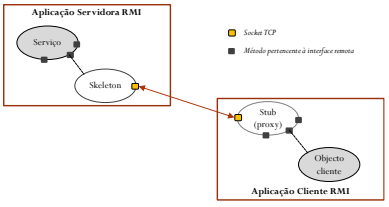
- ***Stub***
  - Objeto que integra a aplicação cliente e actua como um proxy
  - Implementa a interface remota associada ao serviço pretendido
  - É invocada pelo cliente à semelhança de qualquer objeto local
  - Não executa os pedidos directamente
  - Envia uma mensagem ao serviço remoto pretendido, aguarda pela resposta e devolve-a ao cliente






# Arquitetura

- ***Skeleton***
  - Objeto que aguarda por pedidos na aplicação servidora
  - Invoca o método pretendido no serviço/objeto alvo (uma instância de uma implementação concreta da interface)
  - Devolve o resultado ao *Stub*



## Argumentos e Resultados

- Qualquer tipo primitivo ou objeto que seja *serializável* pode ser passado como argumento ou devolvido como resposta numa invocação remota
- Antes de serem enviados através da ligação TCP, os dados (argumentos e resultados) são serializados (*data marshalling*)
- São usadas subclasses de *ObjectOutputStream* e *ObjectInputStream*



## Passos Principais...

1. Definir e compilar a **interface remota** pretendida
2. Definir e compilar uma classe concreta/**serviço** que implemente a interface
3. [~~Produzir as classes ***Stub*** e ***Skeleton*** associados ao serviço~~]
4. Definir e compilar a classe **servidor**
5. Definir e compilar a classe **cliente**
6. [~~Copiar as classes necessárias para os sistemas de ficheiros locais aos clientes e servidor (verificar a *classpath*)~~]
7. Lançar o **RMI Registry**
8. Lançar o servidor
9. Correr os clientes

## Interfaces Remotas

- Primeiras componentes a serem definidas
- Definem
  - Os métodos que podem ser invocados remotamente
  - Os parâmetros
  - Os tipos devolvidos
  - As excepções que podem ser lançadas
- Estendem a interface ***java.rmi.Remote***
- Os métodos devem ser públicos e capazes de lançar excepções do tipo j***ava.rmi.RemoteException***



## Interfaces Remotas

```
public interface RMILightBulb extends java.rmi.Remote
{
    public void on() throws java.rmi.RemoteException;
    public void off() throws java.rmi.RemoteException;
    public boolean isOn() throws java.rmi.RemoteException;
}
```

## Classes Concretas/Serviços

- Depois de definir uma interface, é necessário implementá-la
- A classe concreta resultante deve estender a classe *java.rmi.server.UnicastRemoteObject*
- Todos os métodos da interface remota devem ser implementados
- Não é requerido qualquer tipo de código adicional
  - Muito simples na perspetiva do programador



## Classes Concretas/Serviços

```
public class RMILightBulbImpl extends java.rmi.server.UnicastRemoteObject
    implements RMILightBulb
{
    private boolean lightOn;

    // A CONSTRUCTOR MUST BE PROVIDED FOR THE REMOTE OBJECT
    public RMILightBulbImpl() throws java.rmi.RemoteException {
        setBulb(false);
    }

    public void setBulb(boolean value) {
        lightOn = value;
    }

    public boolean getBulb() {
        return lightOn
    }
```

# Classes Concretas/Serviços

```
   // REMOTELY ACCESSIBLE METHODS

    public void on() throws java.rmi.RemoteException {
        setBulb(true);
    }

     public void off() throws java.rmi.RemoteException {
        setBulb (false);
    }

    public boolean isOn() throws java.rmi.RemoteException {
        return getBulb();
    }

}
```



# Classes *Stub* e *Skeleton*

- Classes (*byte code*) produzidas pela ferramenta ***rmic*** a partir da classe concreta (i.e., da implementação do serviço) e da interface

```
rmic  -v1.1 RMILightBulbImpl
```

```
RMILightBulbImpl_Stub.class
RMILightBulbImpl_Skeleton.class
```

- Nota:
  - Por omissão, a ferramenta *rmic* usa a versão 1.2 do protocolo *Stub* que não recorre a classes *Skeleton* → não são produzidas classes *Skeleton*
  - A partir da versão 1.6 da linguagem Java, a ferramenta *rmic* deixou de ser usada

# Servidor RMI

- O servidor é responsável pela criação de uma instância do serviço
- A sua funcionalidade pode ser incluída num método estático *main* na classe concreta

```
import java.rmi.*;
import java.rmi.server.*;

public class LightBulbServer
{
   public static void main(String args[])
   {
      System.out.println ("Loading RMI service");

      try{
         // LOAD THE SERVICE
         RMILightBulbImpl bulbService = new RMILightBulbImpl();
```

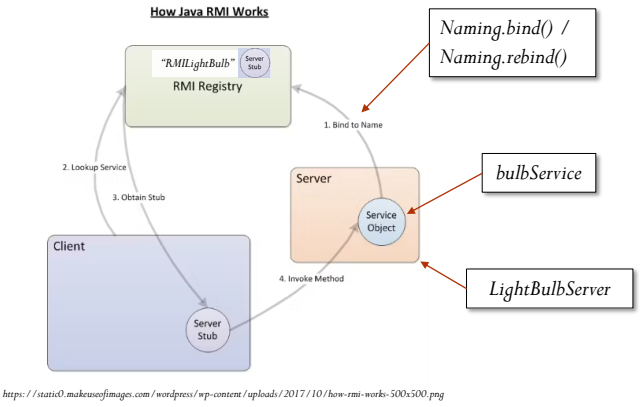


# Servidor RMI

```
         // EXAMINE THE SERVICE TO SEE WHERE IT IS STORED
         RemoteRef location = bulbService.getRef();
         System.out.println (location.remoteToString());

         // CHECK TO SEE IF A REGISTRY WAS SPECIFIED
         String registry = "localhost";
         if (args.length >= 1){ registry = args[0]; }

         // REGISTRATION FORMAT: //REGISTRY_HOSTNAME[:PORT]/SERVICE_NAME
         String registration = "rmi://" + registry + "/RMILightBulb";

         // REGISTER WITH SERVICE SO THAT CLIENTS CAN FIND US
         // AN ALREADY REGISTERED SERVICE WILL BE REPLACED
         Naming.rebind( registration, bulbService );

      }catch (RemoteException e){
         System.err.println ("Remote Error - " + e);
      }catch (Exception e){
         System.err.println ("Error - " + e);
      }
   }
}
```

*Naming.**bind()*** gera uma excepção caso já se encontre registado um serviço com o mesmo nome.

# Servidor RMI




https://static0.makeuseofimages.com/wordpress/wp-content/uploads/2017/10/how-rmi-works-500x500.png



# Cliente RMI

- Apenas deve obter uma referência para o objeto remoto, via RMI Registry
- Os aspectos da comunicação (ligações TCP, mensagens, etc.) são completamente transparentes para o programador

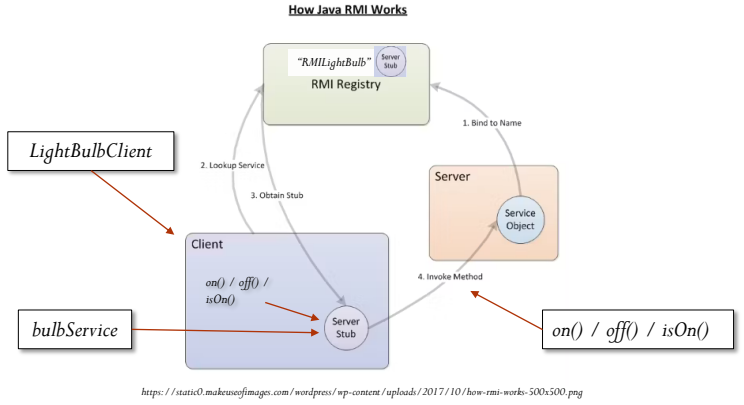
```
import java.rmi.*;

public class LightBulbClient
{
   public static void main(String args[])
   {
      System.out.println ("Looking for light bulb service");

      try{
         String registry = "localhost";
         if (args.length >=1){ registry = args[0]; }

         String registration = "rmi://" + registry + "/RMILightBulb";
```

# Cliente RMI

```
            Remote remoteService = Naming.lookup ( registration );

            // CAST TO A RMILIGHTBULB INTERFACE
            RMILightBulb bulbService = (RMILightBulb) remoteService;

            bulbService.on();
            System.out.println ("Bulb state : " + bulbService.isOn() );

            bulbService.off();
            System.out.println ("Bulb state : " + bulbService.isOn() );

        }catch (NotBoundException e){
            System.out.println ("No light bulb service available!");
        }catch (RemoteException e){
            System.out.println ("RMI Error - " + e);
        }catch (Exception e){
            System.out.println ("Error - " + e);
        }
    }
}
```



# Servidor RMI

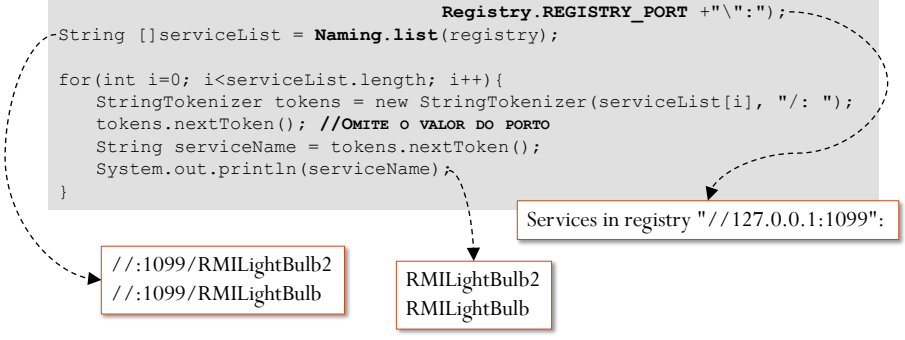



https://static0.makeuseofimages.com/wordpress/wp-content/uploads/2017/10/how-rmi-works-500x500.png

# Cliente RMI

- É possível obter a lista de serviços registados no RMI registry

```
String registry = "127.0.0.1";
System.out.println ("\nServices in registry \"//" + registry + ":" +
                                    Registry.REGISTRY_PORT +"\":");
String []serviceList = Naming.list(registry);

for(int i=0; i<serviceList.length; i++){
   StringTokenizer tokens = new StringTokenizer(serviceList[i], "/: ");
   tokens.nextToken(); //OMITE O VALOR DO PORTO
   String serviceName = tokens.nextToken();
   System.out.println(serviceName);
}
```






# Funcionalidades adicionais

- Anular o registo de um serviço no RMI Registry

```
Naming.unbind( "rmi://127.0.0.1/RMILightBulb");
```

- Terminar um serviço RMI

```
UnicastRemoteObject.unexportObject(bulbService, true);
```

- Definir de forma explícita o endereço incluído na referência remota de um serviço RMI (importante quando existem várias interfaces de rede ativas na máquina onde corre o serviço): ***java.rmi.server.hostname***

```
No código do servidor:
     System.setProperty("java.rmi.server.hostname","10.202.128.22");
ou com a opção "-Djava.rmi.server.hostname" ao lançar o servidor:
     java -Djava.rmi.server.hostname=10.202.128.22 myRmiService
```

# Funcionalidades adicionais

- Localizar e lançar RMI Registries de forma programática: métodos da classe java.rmi.registry.***LocateRegistry***

```
Registry r1, r2, r3, r4, r5;
r1=r2=r3=r4=r5=null;
try{
    r1 = LocateRegistry.createRegistry(Registry.REGISTRY_PORT); //HOST LOCAL
    r2 = LocateRegistry.getRegistry(); //HOST LOCAL, PORTO WELL-KNOWN 1099
    r3 = LocateRegistry.getRegistry(1099); //HOST LOCAL
    r4 = LocateRegistry.getRegistry("localhost"); //PORTO WELL-KNOWN 1099
    r5 = LocateRegistry.getRegistry("localhost", 1099);
}catch(RemoteException e){
    //...
}
//...
r1.rebind("RMILightBulb", bulbService);
```



# *Callbacks RMI*

- *Callback*: uma técnica habitual no paradigma de programação *event-driven*
- Permite a notificação assíncrona de eventos
- Evita a verificação periódica do estado de um objeto por outro (*polling*)
- Basicamente, recorre-se a objectos *listener* que se registam no objeto fonte do evento
- Quando um evento ocorre, o objeto fonte notifica todos os *listeners* registados

## Callbacks RMI

- Alguns considerações gerais sobre *callbacks* locais
  - Um *listener* deve implementar uma determinada interface com métodos destinados a notificá-lo (e.g., *void* ***propertyChange****(PropertyChangeEvent evt)* da interface *java.beans.* ***PropertyChangeListener***)
  - Quando um *listener* regista-se no objeto fonte do evento (e.g., o método ***addPropertyChangeListener****(PropertyChangeListener listener)* de *java.beans.****PropertyChangeSupport***), este último guarda a sua referência numa lista



## Callbacks RMI

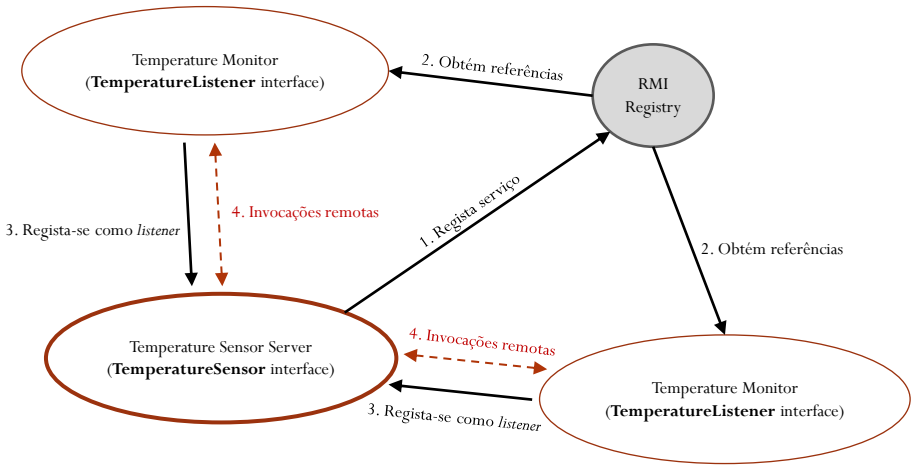
- Quando o evento ocorre, o objeto fonte invoca o método adequado em todos os *listeners* que se encontram na lista (e.g., *public void* ***firePropertyChange****(String propertyName, boolean oldValue, boolean newValue)* de *java.beans.****PropertyChangeSupport***)

# Callbacks RMI

- *Callbacks* remotos
  - Os *listeners* devem ser realizados sob a forma de serviços RMI
  - As fontes dos eventos devem ser realizados sob a forma de serviços RMI
  - *Listeners* e fontes de eventos são reciprocamente clientes e servidores RMI
- Exemplo
  - Um serviço de medição de temperatura que notifica os *listeners* registados sempre que ocorre uma variação



# Callbacks RMI



# Callbacks RMI

- A interface remota associada aos *listeners* define apenas o método *temperatureChanged()*

```
interface TemperatureListener extends Remote
{
   public void temperatureChanged(double temperature) throws RemoteException;
}
```

- Interface associada ao serviço de medição de temperatura (i.e., fonte do evento mudança de temperatura)

```
interface TemperatureSensor extends java.rmi.Remote
{
   public double getTemperature() throws java.rmi.RemoteException;
   public void  addTemperatureListener (TemperatureListener listener)
                                          throws java.rmi.RemoteException;
   public void  removeTemperatureListener (TemperatureListener listener)
                                          throws java.rmi.RemoteException;
}
```



# Callbacks RMI

- Serviço + Servidor sensor de temperatura

```
import java.util.*; import java.rmi.*;
import java.rmi.server.*;

public class TemperatureSensorServer extends UnicastRemoteObject implements
TemperatureSensor, Runnable
{
   private volatile double  temp; //THE VALUE OF THIS VARIABLE WILL NEVER BE
                                  //CACHED THREAD-LOCALLY
   private List<TemperatureListener> list;

   public TemperatureSensorServer() throws java.rmi.RemoteException
   {
      temp = 20.0; list = new ArrayList<TemperatureListener>();
   }

   public synchronized double getTemperature() throws java.rmi.RemoteException
   {
      return temp;
   }
```

## Callbacks RMI

```
public synchronized void addTemperatureListener ( TemperatureListener
                                listener ) throws RemoteException
{
    System.out.println ("adding listener -" + listener);
    list.add (listener);
}

public synchronized void removeTemperatureListener ( TemperatureListener
                        listener ) throws java.rmi.RemoteException
{
    System.out.println ("removing listener -" + listener);
    list.remove (listener);
}

public synchronized void changeTemp(Random r) {
    int num = r.nextInt();
    if (num < 0) temp += 0.5;
    else temp -= 0.5;
    // NOTIFY REGISTERED LISTENERS
    notifyListeners();
}
```



## Callbacks RMI

```
public void run()
{
    Random r = new Random();

    while(true){
        try{
            // SLEEP FOR A RANDOM AMOUNT OF TIME
            int duration = r.nextInt() % 10000 + 2000;
            if (duration < 0) duration = -1 * duration;
            Thread.sleep(duration);
        }catch (InterruptedException e) { }

        // COMPUTE NEW VALUE FOR TEMP
        changeTemp(r);
    }
} //WHILE
```

# Callbacks RMI

```
    private synchronized void notifyListeners()
    {
        for(int i=0; i<list.size(); i++){
            try{
                list.get(i).temperatureChanged(temp);
            }catch (RemoteException e){ //UNABLE TO CONTACT LISTENER
                System.out.println ("removing listener -" + list.get(i));
                list.remove( i-- );
            }
        }
    }

    public static void main(String args[])
    {
        System.out.println ("Loading temperature service");

        // ONLY REQUIRED FOR DYNAMIC CLASS LOADING
        //System.setSecurityManager ( new RMISecurityManager() );
```



# Callbacks RMI

```
        try{
            // LOAD THE SERVICE
            TemperatureSensorServer sensor = new TemperatureSensorServer();

             // REGISTER WITH SERVICE SO THAT CLIENTS CAN FIND US
            String registry = "localhost";
            if (args.length >=1){registry = args[0];}

            String registration = "rmi://" + registry + "/TemperatureSensor";
            Naming.rebind( registration, sensor );

            // CREATE A THREAD TO TRIGGER REGULAR TEMPERATURE CHANGES
            Thread thread = new Thread (sensor);
            thread.start();
        }catch (RemoteException re){
            System.err.println ("Remote Error - " + re);
        }catch (Exception e){
            System.err.println ("Error - " + e);
        }
    } //MAIN
}
```

# Callbacks RMI

- Monitor de mudança de temperatura (*listener*)

```
import java.rmi.*;
import java.rmi.server.*;

public class TemperatureMonitor extends UnicastRemoteObject implements
TemperatureListener
{
   public TemperatureMonitor() throws RemoteException { // NO CODE REQUIRED }

   public static void main(String args[])
   {
      System.out.println ("Looking for temperature sensor");

      // ONLY REQUIRED FOR DYNAMIC CLASS LOADING
      //SYSTEM.SETSECURITYMANAGER ( NEW RMISECURITYMANAGER() );

      try{
         String registry = "localhost";
         if (args.length >=1){registry = args[0];}
```



# Callbacks RMI

```
         // LOOKUP THE SERVICE IN THE REGISTRY, AND OBTAIN A REMOTE SERVICE
         String registration = "rmi://" + registry + "/TemperatureSensor";
         Remote remoteService = Naming.lookup( registration );
         TemperatureSensor sensor = (TemperatureSensor) remoteService;

         // GET AND DISPLAY CURRENT TEMPERATURE
         double reading = sensor.getTemperature();
         System.out.println ("Original temp : " + reading);

         // CREATE A NEW MONITOR AND REGISTER IT AS A LISTENER WITH REMOTE SENSOR
         TemperatureMonitor monitor = new TemperatureMonitor();
         sensor.addTemperatureListener(monitor);
      }catch (NotBoundException e){
         System.out.println ("No sensors available");
      }catch (RemoteException e)  {
         System.out.println ("RMI Error - " + e);
      }catch (Exception e){
         System.out.println ("Error - " + e);
      }
   } //MAIN
```

## Callbacks RMI

```
    public void temperatureChanged(double temperature)
        throws java.rmi.RemoteException
    {
        System.out.println ("Temperature change event : " + temperature);
    }
}
```



## Bibliografia


- REILLY, David; REILLY, Michael - *Java Network Programming & Distributed Computing* - Addison-Wesley
- GROSSO, William - *Java RMI* - O'Reilly Media
- COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim - *Distributed Systems – Concepts and Design*, Addison-Wesley
- http://download.oracle.com/javase/tutorial/essential/