RMI
[Remote Method Invocation]

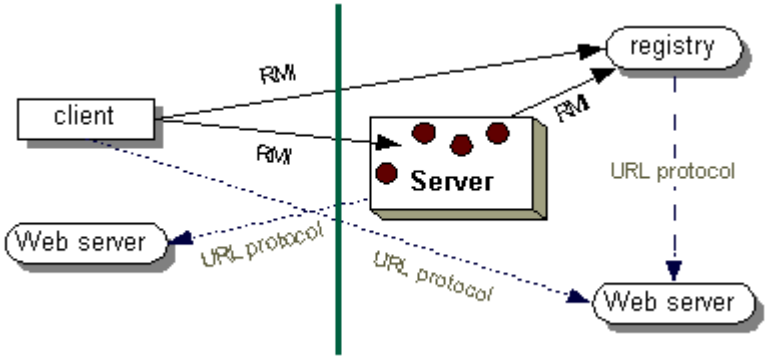
Cuando utilizamos sockets, hemos de preocuparnos de cómo se transmiten físicamente los datos entre los extremos de una conexión (a nivel de bytes, ya que usamos los “streams” estándar)

RMI permite olvidarnos de los detalles de la transmisión de datos y centrarnos en el diseño de la lógica de nuestra aplicación, puesto que nos permite acceder a un objeto remoto como si de un objeto local se tratase.

- Internamente, RMI utiliza **serialización** de objetos para encargarse de la transmisión de datos a través de la red (de cara al programador, el acceso al objeto remoto es como una llamada a un método local).

- Para localizar un objeto al que se desee acceder, RMI proporciona un **registro** que se usa a modo de páginas amarillas.

- Como respuesta de las llamadas a métodos de un objeto remoto, RMI devuelve objetos y se encarga de obtener los **bytecodes** que sean necesarios (cuando se obtiene una referencia a un objeto cuyos bytecodes no están disponibles en la máquina virtual del receptor).



En RMI:

- El servidor crea algunos objetos y los hace accesibles a través del registro. A continuación, se queda esperando a recibir peticiones.

- El cliente obtiene una referencia a un objeto remoto (que está alojado en el servidor) y la utiliza para invocar métodos del objeto de forma remota.

Objetos e interfaces remotos

Para que un objeto sea accesible de forma remota, ha de implementar un interfaz remoto (derivado de `java.rmi.Remote`)

Cuando una llamada a un método realizada desde el cliente devuelve un objeto remoto, en vez de obtener una copia del objeto, se obtiene una referencia al objeto remoto (un ***stub*** que hace de *proxy* y se comporta igual que el objeto remoto).

Desarrollo de aplicaciones distribuidas con RMI

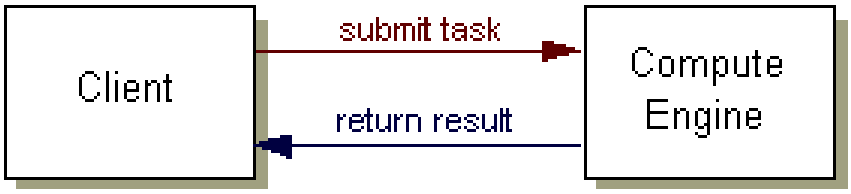
1. Crear los distintos componentes de la aplicación, teniendo en cuenta que, para que a un objeto se pueda acceder de forma remota con RMI, es necesario que implemente una interfaz derivada de `java.rmi.Remote`.

2. Compilar con `javac` y generar los stubs con `rmic`.

3. Hacer accesibles los objetos remotos (usualmente, dejando los bytecodes correspondientes a los interfaces remotos y a los stubs en algún servidor web).

4. Arrancar la aplicación, que incluye el registro RMI, el servidor y el cliente.

Ejemplo: Plataforma distribuida de cómputo



El servidor RMI

1. Interfaz remota

Para los objetos a los que se accederá de forma remota:

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

public interface Compute extends Remote
{
  Object executeTask(Task t)
         throws RemoteException;
}
```

2. Objetos serializables

Aquéllos que se transmitirán a través de la red:

```
import java.io.Serializable;

public interface Task extends Serializable {
    Object execute();
}
```

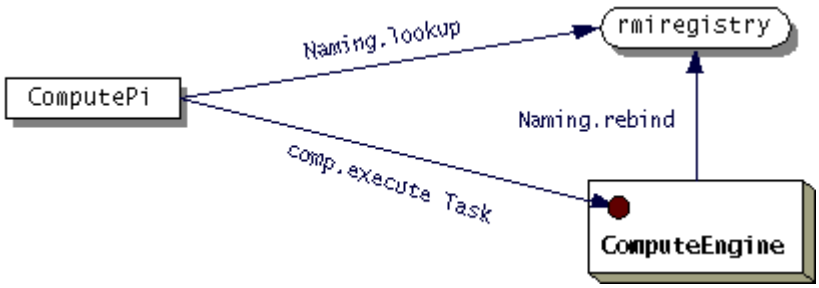
3. Implementación del servidor

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

public class ComputeEngine
       extends UnicastRemoteObject  // Objeto remoto
       implements Compute           // Interfaz remota
{
  public ComputeEngine()
         throws RemoteException
  {
    super();
  }

  public Object executeTask (Task t)
  {
    return t.execute();
  }

  // Programa principal

  public static void main(String[] args)
         throws Exception
  {
    System.setSecurityManager
           (new RMISecurityManager());
    String name = "//elvex.ugr.es/Compute";
    Compute engine = new ComputeEngine();
    Naming.rebind(name, engine);
  }
}
```

- El `SecurityManager` se encarga de controlar las acciones realizadas por el código que se descarga a través de la red (si no creamos uno, no se podrá descargar código remoto).
- `Naming.rebind()` registra un objeto para que se pueda acceder a él de forma remota a través de una URL.
- Mientras que exista alguna referencia a ese objeto (aunque sea la del registro), el servidor no finalizará su ejecución.

El cliente RMI

```
import java.rmi.*;
import java.math.*;

public class ComputePi {

  public static void main(String args[])
  {
    System.setSecurityManager
           (new RMISecurityManager());

    String name = "//" + args[0] + "/Compute";
    Compute comp = (Compute) Naming.lookup(name);

    …
    tarea = new XTask();
    resultado = (XResult) comp.executeTask(task);

    System.out.println(resultado);
  }
}
```



NOTA

La característica más destacable de RMI es que el servidor no tiene por qué disponer de antemano de los bytecodes asociados a la tarea que ha de ejecutar.

Cuando recibe la petición del cliente, carga los bytecodes en su máquina virtual, ejecuta la tarea y devuelve el resultado.

Extensiones de RMI: Jini

Jini está montado sobre RMI y permite descubrir dinámicamente qué dispositivos y servicios están disponibles en la red, sin tener que saber de antemano su localización exacta (“plug & play”).

Alternativas a RMI: CORBA y .NET Remoting

CORBA

RMI se puede interpretar como una versión simplificada de CORBA (un estándar complejo que permite la implementación de sistemas distribuidos basados en objetos, independientemente del lenguaje de programación que se utilice para implementar las distintas partes del sistema).

NOTA: RMI sólo puede utilizarse en Java, por lo que no sirve para conectar una aplicación Java a un sistema no escrito en Java.

.NET Remoting

La plataforma .NET (la alternativa de Microsoft a Java), incluye un mecanismo similar a RMI que se denomina .NET Remoting.

Más información en
http://elvex.ugr.es/decsai/csharp/distributed/remoting.xml