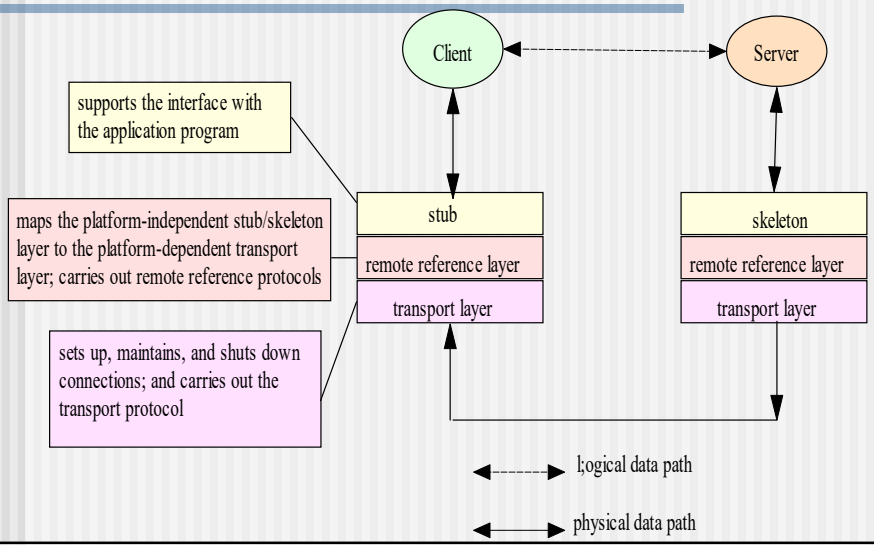
Computación Distribuida

Tema 4: RMI Avanzado

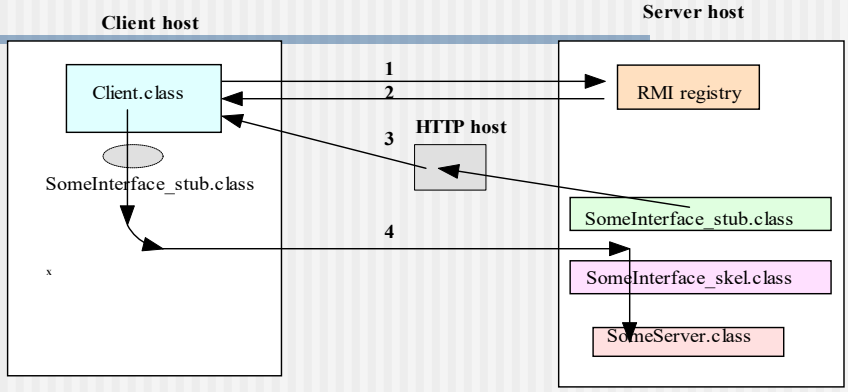
RMI – Cuestiones avanzadas

- El API de Java RMI tiene múltiples funcionalidades.
- Aquí analizaremos algunas de las características avanzadas de RMI más interesantes, a saber:
  - **stub downloading**
  - **security manager**
  - **client callback**.
  - **serialización y envío de objetos**
- Aunque no se trata de características inherentes del paradigma de objetos distribuidos, se trata de mecanismos que pueden ser útiles para los desarrolladores de aplicaciones.

La arquitectura Java RMI



Interacción Cliente Servidor en Java RMI



1. Client looks up the interface object in the RMIregistry on the server host.
2. The RMIRegistry returns a remote reference to the interface object.
3. If the interface object's stub is not on the client host and if it is so arranged by the server, the stub is downloaded from an HTTP server.
4. Via the server stub, the client process interacts with the skeleton of the interface object to access the methods in the server object.

RMI Stub Downloading

- RMI se ha diseñado para permitir que los clientes obtengan dinámicamente el stub. Esto permite realizar cambios en los métodos remotos sin afectar al programa cliente.
- El stub puede ser colocado en un servidor web y descargado usando el protocolo HTTP.
- Es necesario establecer ciertas medidas de seguridad tanto en la parte del cliente como del servidor:
- En concreto, es necesario un fichero que describa la política de seguridad.
- Debe realizarse una instancia de un Java Security Manager tanto en el cliente como en el servidor.

Stub downloading

- **Si el stub va a ser descargado de un servidor remoto, debe moverse la clase stub al directorio apropiado del servidor y asegurarse que tiene los permisos de acceso necesarios.**
- **Cuando activamos el servidor, se debe especificar las siguientes opciones:**

```
java  -Djava.rmi.serve.codebase = <URL>/  \
      -Djava.rmi.server.hostname=<server host name> \
      -Djava.security.policy=<full directory path to java policy file>
```

where <URL> is the URL for the stub class, e.g.,
http://www.csc.calpoly.edu/~mliu/class
<server host name> is the name of the host on which the server runs,
and <full directory path to java policy file> specifies where the security policy file for this application is to be found, e.g., java.security if you have a file by that name in the directory where the server class is.

El fichero java.policy

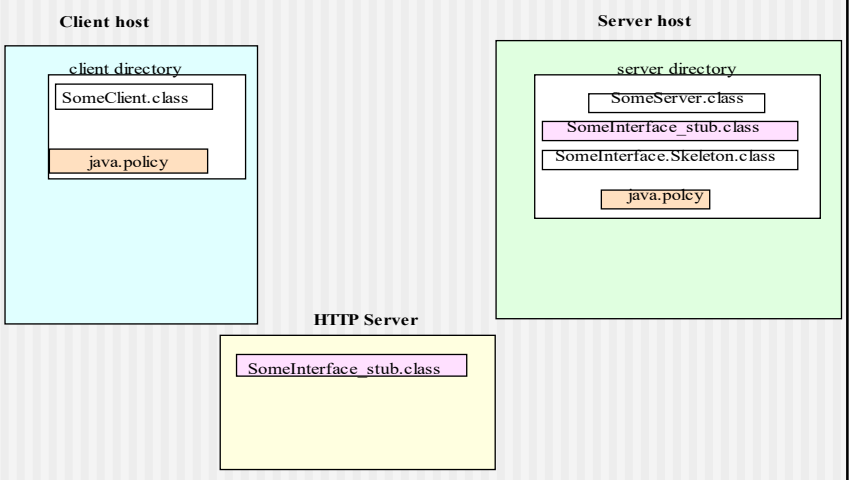
- El gestor de seguridad de RMI no permite un acceso a la red. Las excepciones deben especificarse en un fichero java.policy.

```
grant {
    // permits socket access to all common TCP ports, including the
default
     // RMI registry port (1099) – need for both the client and the server.
       permission java.net.SocketPermission "*:1024-65535",
      "connect,accept,resolve";
     // permits socket access to port 80, the default HTTP port – needed
     // by client to contact an HTTP server for stub downloading
       permission java.net.SocketPermission "*:80", "connect";
};
```

- Este fichero puede colocarse en el mismo directorio del archivo class del servidor.
- Cuando activamos el cliente, debemos especificar también un fichero java.policy:

```
java -Djava.security.policy=java.policy  SomeClient
```

Colocación de los ficheros



RMI Security Manager

- Puesto que RMI involucra el acceso desde/a una máquina remota y, posiblemente la descarga de objetos, es importante que tanto el servidor como el cliente se protejan ante accesos inadecuados o no permitidos.
- **RMISecurityManager** es una clase de Java que puede ser instanciada tanto en el cliente como en el servidor para limitar los privilegios de acceso.
- Es posible escribir nuestro propio gestor de seguridad, si lo deseamos.

```
try {
        System.setSecurityManager(new
           RMISecurityManager( ));
}
catch { …}
```

Algoritmo para construir una aplicación RMI

Lado del servidor:

1. Crear un directorio donde se almacenen todos los ficheros generados por la aplicación.
2. Especificar la interfaz remota y compilarla para generar el archivo .class de la interfaz.
3. Construir el servidor remoto implementando la interfaz y compilarlo hasta que no haya ningún error.
4. Usar rmic para procesar la clase del servidor y generar un fichero .class de stub y un fichero .class de skeleton: *rmic SomeServer*
5. Si se desea stub downloading, copiar el fichero stub al directorio apropiado del servidor HTTP.
6. Activar el RMIRegistry en el caso de que no haya sido activado previamente.
7. Construir un fichero de políticas de seguridad para la aplicación llamado java.policy.
8. Activar el servidor especificando (i) el campo codebase si se utiliza stub downloading, (ii) el nombre del servidor y (iii) el fichero de políticas de seguridad.

Algoritmo para construir una aplicación RMI

Lado del cliente:

1. Crear un directorio donde se almacenen todos los ficheros generados por la aplicación.
2. Implementar el programa cliente o applet y compilarlo para generar la clase cliente.
3. Si no se puede usar stub downloading, copiar el fichero class de stub a mano.
4. Especificar el fichero de políticas de seguridad java.policy.
5. Activar el cliente especificando: (i) el nombre del servidor y (ii) el fichero con las políticas de seguridad.

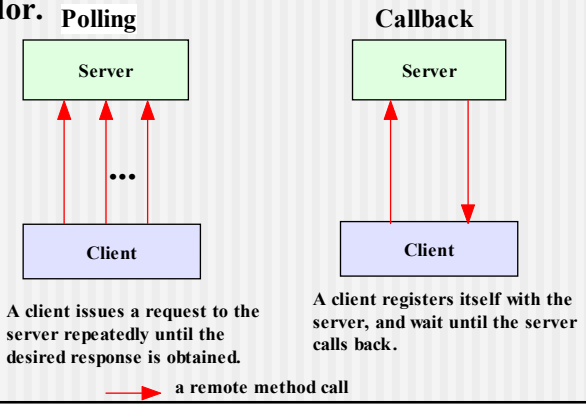
RMI Callbacks

Introducción

- **En el modelo cliente servidor, el servidor es pasivo: la comunicación IPC es iniciada por el cliente; el servidor espera por la llegada de las peticiones y proporciona las respuestas.**
- **Algunas aplicaciones necesitan que el servidor inicie la comunicación ante la ocurrencia de determinados eventos. Ejemplos de este tipo de aplicaciones las tenemos en:**
  - **monitorización**
  - **juegos**
  - **subastas**
  - **trabajo colaborativo**
  - **…**

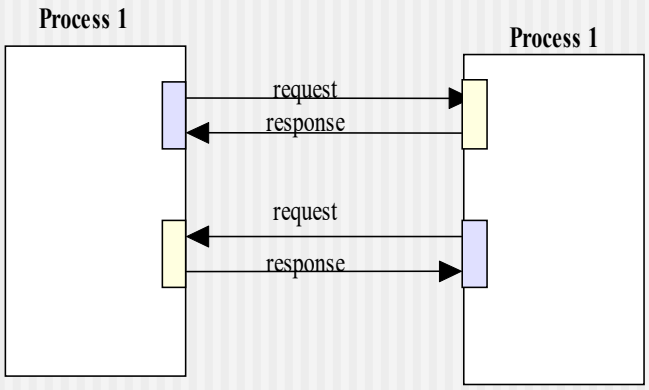
Polling vs. Callback

Si no disponemos de callback, un cliente tendrá que realizar un sondeo (polling) a un servidor pasivo repetidas veces si necesita ser notificado de que un evento ha ocurrido en el servidor.



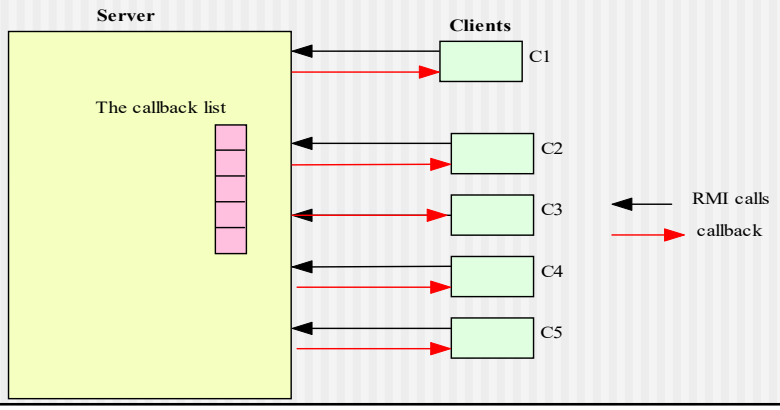
Comunicaciones en ambos sentidos

- **Algunas aplicaciones necesitan que ambos lados puedan iniciar una comunicación IPC.**
- **Si usamos sockets, se puede conseguir una comunicación duplex si utilizamos dos sockets en cada lado.**
- **Con sockets orientados a conexión, cada lado actúa tanto como cliente como servidor.**

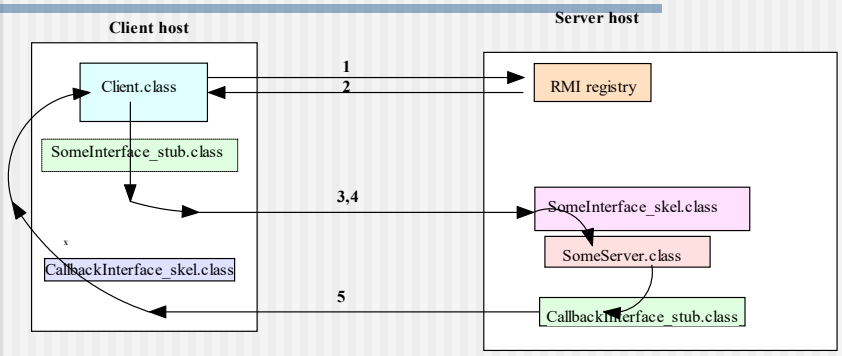


RMI Callbacks

- **Un cliente de un callback se registra en un servidor RMI.**
- **El servidor realiza la callback a cada cliente registrado ante la ocurrencia de un determinado evento.**

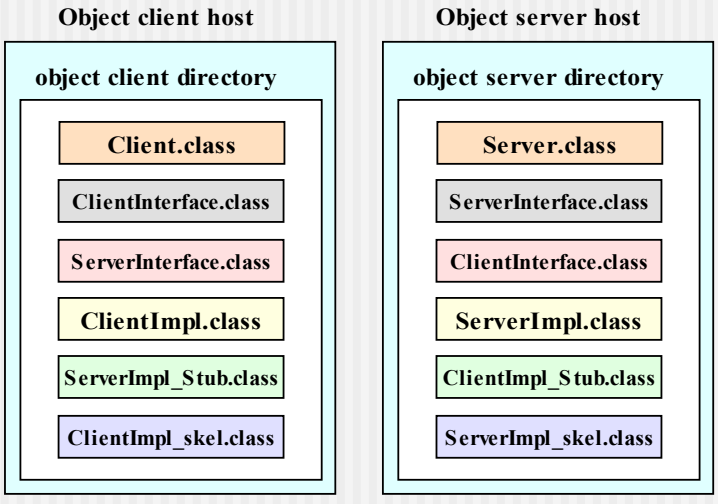


Interacciones Cliente-Servidor con Callback

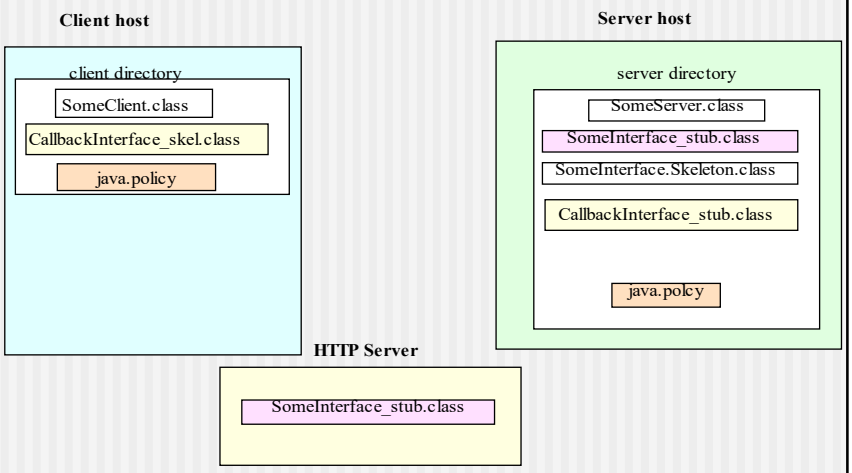


1. Client looks up the interface object in the RMIregistry on the server host.
2. The RMIRegistry returns a remote reference to the interface object.
3. Via the server stub, the client process invokes a remote method to register itself for callback, passing a remote reference to itself to the server. The server saves the reference in its callback list.
4. Via the server stub, the client process interacts with the skeleton of the interface object to access the methods in the interface object.
5. When the anticipated event takes place, the server makes a callback to each registered client via the callback interface stub on the server side and the callback interface skeleton on the client side.

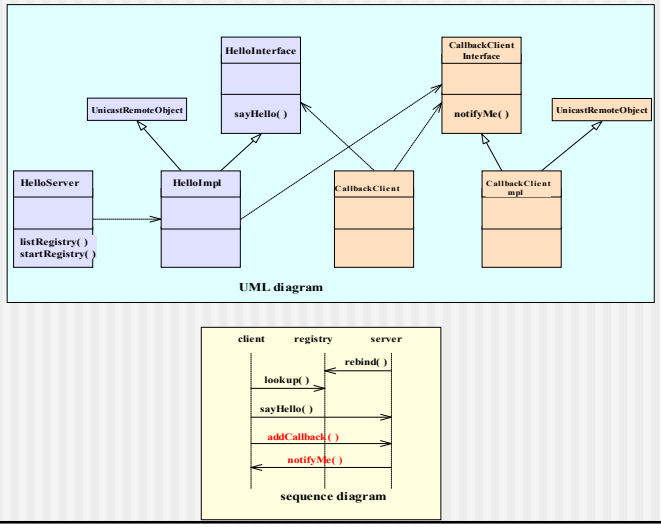
Ficheros de una aplicación Callback



Colocación de los ficheros en una RMI Callback



La aplicación Hello con Callback



Interfaz RMI de Callback

- El servidor proporciona un método remoto que permite al cliente registrarse para recibir callbacks.
- Se necesita una interfaz remota para la callback, además del interfaz de servidor.
- La interfaz especifica un método para aceptar llamadas de un servidor.
- El programa cliente es una subclase de RemoteObject e implementa la interfaz de callback, incluyendo el método de callback.
- El cliente se registra para el callback en su método main.
- El servidor invoca el método remoto del cliente ante la ocurrencia de un determinado evento.

Interfaz remoto para el servidor

```
public interface HelloInterface extends Remote {
  // remote method
  public String sayHello() throws java.rmi.RemoteException;
  // method to be invoked by a client to add itself to the
    callback list
  public void addCallback(
    HelloCallbackInterface CallbackObject)
    throws java.rmi.RemoteException;
}
```

Interfaz remoto para el cliente de callback

```
// an interface specifying a callback method
public interface HelloCallbackInterface extends java.rmi.Remote
{
  // method to be called by the server on callback
  public void callMe (
      String message
   ) throws java.rmi.RemoteException;
}
```

HelloServer con callback

```
public class HelloServer extends UnicastRemoteObject implements
   HelloInterface {
   static int RMIPort;
   // vector for store list of callback objects
   private static Vector callbackObjects;

   public HelloServer() throws RemoteException {
    super();
    // instantiate a Vector object for storing callback objects
    callbackObjects = new Vector();
   }
   // method for client to call to add itself to its callback
   public void addCallback(  HelloCallbackInterface CallbackObject) {
     // store the callback object into the vector
      System.out.println("Server got an 'addCallback' call.");
     callbackObjects.addElement (CallbackObject);
   }
```

HelloServer con callback - 2

```
public static void main(String args[]) {
    …
  registry = LocateRegistry.createRegistry(RMIPort);
    …
  callback( );
    …
} // end main
private static void callback( ) {
    …
  for (int i = 0; i < callbackObjects.size(); i++)  {
    System.out.println("Now performing the "+ i +"th callback\n");
    // convert the vector object to a callback object
    HelloCallbackInterface client =
      (HelloCallbackInterface) callbackObjects.elementAt(i);
     …
    client.callMe ( "Server calling back to client " + i);
     …
```

Algoritmo para construir una aplicación RMI con callback

Lado del servidor:

1. Crear un directorio donde se almacenen todos los ficheros generados por la aplicación.
2. Especificar la interfaz remota de servidor y compilarla para generar el fichero .class de la interfaz.
3. Construir la clase remota del servidor implementando el interfaz y compilarla hasta que no exista ningún error de sintaxis.
4. Utiliar rmic para procesar la clase del servidor y generar un fichero .class de stub y otro fichero .class de skeleton
5. Si se requiere stub downloading, copiar el fichero stub al directorio apropiado del servidor HTTP.
6. Activar el registro de RMI, si no estaba previamente activo.
7. Establecer la política de seguridad en el archivo java.policy.
8. Activar el servidor especificando (i) el codebase si se requiere stub downloading, (ii) el nombre del servidor y (iii) el fichero con la política de seguridad.
9. Obtener el CallbackInterface. Compilarlo con *javac* y usar *rmic* para generar el fichero de stub para la callback.

Algoritmo para construir una aplicación RMI con callback

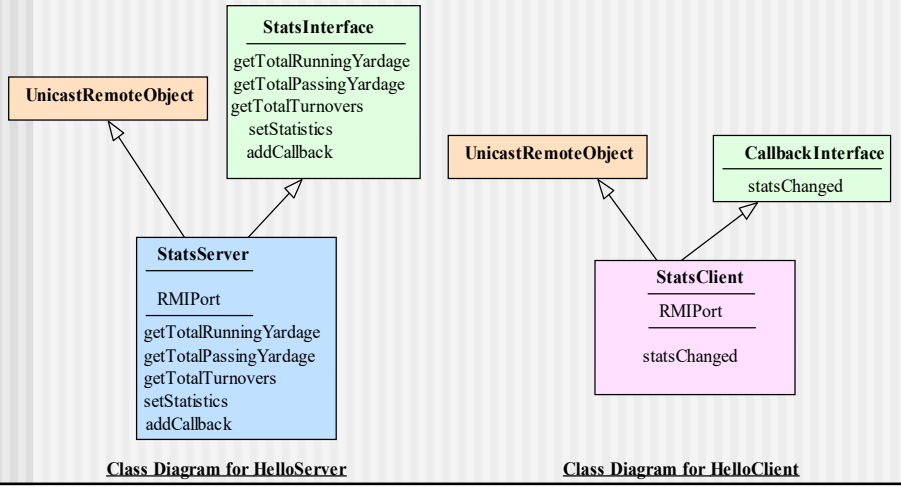
Lado del cliente:

1. **Crear un directorio donde se almacenen todos los ficheros generados por la aplicación.**
2. **Implementar el programa cliente o applet y compilarlo para generar la clase cliente.**
3. **Si no está activo el stub downloading, copiar el fichero .class del stub correspondiente al interfaz del servidor a mano.**
4. **Implementar la interfaz de callback. Compilarla usando *javac*, y usando *rmic* generar los ficheros .class correspondientes al stub y el skeleton.**
5. **Establecer la política de seguridad en el fichero java.policy.**
6. **Activar el cliente especificando (i) el nombre del servidor y (ii) el fichero con la política de seguridad.**

HelloClient con callback

```
HelloClient() { // constructor
    System.setSecurityManager(new RMISecurityManager());
    // export this object as a remote object
      UnicastRemoteObject.exportObject(this);
        // …
      Registry registry = LocateRegistry.getRegistry("localhost",  RMIPort);
      h = (HelloInterface) registry.lookup("helloLiu");
      h.addCallback(this);  //  …
} // end constructor
// call back method - this displays the message sent by the server
public void callMe (String message) {
  System.out.println( "Call back received: " + message );
}
public static void main(String args[]) {  // …
     HelloClient client   = new HelloClient();  // …
} // end main
} // end HelloClient class
```

StatsServer, StatsClient



Serialización y envío de objetos

Serialización de objetos

- A veces resulta necesario el pasar como argumento a un método de un objeto remoto tipos de datos complejos como, por ejemplo, objetos que hayamos creado nosotros.
- En Java con RMI es posible gracias al concepto de serialización, que consiste en encapsular el contenido de un objeto (código + datos) en una cadena de caracteres susceptible de poder ser enviada a través de la red.
- La máquina virtual Java nos garantiza que la reconstrucción del objeto recibido en la parte remota será correcta y que el objeto funcionará sin problemas.

Un ejemplo complejo: integración numérica

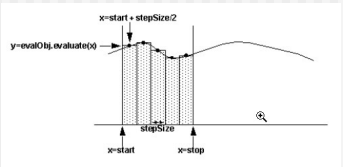
- **Programa iterativo para calcular la suma:**

$$\sum_{x=start}^{stop} f(x)$$

- **Se usa para aproximar numéricamente integrales de la forma:**

$$\int_{start}^{stop} f(x)dx$$

- **Regla del punto medio:**



- Motivación para usar RMI
  - Cuanto más pequeños sean los rectángulos mejor será la aproximación, pero esto puede demandar elevados recursos de computación
  - RMI puede permitir la ejecución remota de las partes con una mayor demanda de computación

Integración numérica - 2

```
public class Integral {
/** Devuelve la suma de f(x) desde x=start hasta x=stop, donde la funcion f
 * se define por la evaluacion del metodo del objeto Evaluatable
 */
public static double sum (double start, double stop,
                    double stepSize,
                    Evaluatable evalObj) {
    double sum = 0.0, current = start;
    while(current <= stop) {
          sum += evalObj.evaluate(current);
          current += stepSize;
    }
    return(sum);
}

public static double integrate(double start, double stop,
                    int numSteps,
                    Evaluatable evalObj) {
    double stepSize = (stop - start) / (double)numSteps;
    start = start + stepSize / 2.0;
    return(stepSize * sum(start, stop, stepSize, evalObj));
    }
}
```

Integración numérica - 3

```
/** Un interfaz para evaluar funciones y=f(x) en un valor
 * especifico. Tanto x como y son numeros en punto flotante de doble
 * precision
 */

public interface Evaluatable {
   public double evaluate( double value);
}
```

Integración numérica - 4

- La interfaz remota compartida tanto por el cliente como por el servidor

```
import java.rmi.*;

public interface RemoteIntegral extends Remote {

   public double sum(double start, double stop,
   double stepSize,
             Evaluatable evalObj)
       throws RemoteException;

   public double integrate(double start, double
   stop,
             int numSteps, Evaluatable evalObj)
       throws RemoteException;
}
```

Integración numérica - 5

- El cliente envía un objeto Evaluatable al servidor que representa a la función a integrar

```
public class RemoteIntegralClient{
     public static void main(String[] args) {
          try{
                    String host = (args.length > 0) ? args[0] : "localhost";
                    RemoteIntegral remoteIntegral =
                              (RemoteIntegral)Naming.lookup("rmi://" + host + "/RemoteIntegral");
                    for(int steps=10; steps<=10000; steps*=10) {
                              System.out.println("Approximated with " + step + "steps:" +
                                        "\n Integral from 0 to pi of sin(x)=" +
                                        remoteIntegral.integrate(0.0, Math.PI, steps, new Sin()));
                    }
                    System.out.println("'Correct' answer using Math library:" +
                              "\n Integral from 0 to pi of sin(x)=" +
                              (-Math.cos(Math.PI) - -Math.cos(0.0)));
          } catch(RemoteException re) {
                    System.out.println("RemoteException: " + re);
          } catch(NotBoundException nbe) {
                    System.out.println("NotBoundException: " + nbe);
          } catch(MalformedURLException mfe) {
                    System.out.println("MalformedURLException: " + mfe);
          }
     }
}
```

Integración numérica - 6

- La función evaluatable Sin

```
import java.io.Serializable;

class Sin implements Evaluatable, Serializable {
  public double evaluate(double val) {
      return(Math.sin(val));
  }

  public String toString() {
      return("Sin");
  }
}
```

Integración numérica - 7

- La implementación de la integral remota

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

public class RemoteIntegralImpl extends UnicastRemoteObject
        implements RemoteIntegral {

   public RemoteIntegralImpl() throws RemoteException {}

   public double sum(double start, double stop, double stepSize,
                Evaluatable evalObj) {
        return(Integral.sum(start, stop, stepSize, evalObj));
   }

   public double integrate(double start, double stop, int numSteps,
                Evaluatable evalObj) {
        return(Integral.integrate(start, stop, numSteps, evalObj));
   }
}
```

Integración numérica - 8

- El servidor de objeto

```
import java.rmi.*;
import java.net.*;

public class RemoteIntegralServer {
   public static void main(String[] args) {
        try {
                RemoteIntegralImpl integral = new RemoteIntegralImpl();
                Naming.rebind("rmi:///RemoteIntegral", integral);
        } catch(RemoteException re) {
                System.out.println("RemoteException: " + re);
        } catch(MalformedURLException mfe) {
                System.out.println("MalformedURLException: " + mfe);
        }
   }
}
```

Resumen - 1

- **Client callback:**
  - Client callback es útil para una aplicación donde los clientes desean ser notificados por el servidor de la ocurrencia de un evento.
  - Client callback permite que un objeto servidor haga una invocación remota a un método de un cliente a través de un interfaz remoto del cliente.

Resumen - 2

- **Client callback:**
  - Para proporcionar un callback de cliente el software de cliente debe:
    - proporcionar una interfaz remota,
    - instanciar un objeto que implementa dicha interfaz,
    - pasar una referencia al objeto al servidor a través de una invocación remota de un método del servidor.
  - El objeto servidor debe:
    - recoger esas referencias al cliente en una estructura de datos,
    - cuando ocurra el evento esperado, el objeto servidor invoca el método callback (definido en la interfaz remota del cliente) para pasarle los datos al cliente.
  - Se necesitan dos conjuntos stub-skeleton: uno para el interfaz remoto del servidor y otro para el interfaz remoto del cliente.

Resumen - 3

- El stub downloading permite que se pueda cargar una clase stub por parte del objeto cliente en tiempo de ejecución, permitiendo de esta manera la modificación de la implementación del objeto remoto al regenerarse la clase stub sin que afecte al software del cliente.
- Un gestor de seguridad accederá a las restricciones impuesta por un fichero de **políticas de seguridad**, que pueden ser aplicables a nivel de todo el sistema o simplemente a nivel de la aplicación.
- Por cuestiones de seguridad se recomienda el uso de gestores de seguridad en todas las aplicaciones RMI, independientemente de que exista stub downloading.

Resumen - 4

- La serialización es un mecanismo mediante el cual un objeto puede ser transformado en una cadena de bytes susceptible de ser enviada a través de un canal de comunicación o almacenada en una base de datos.
- Para poder pasar como argumento en una invocación remota un objeto es indispensable que dicho objeto sea serializable.