Sesión 6:
Java RMI. Primeros pasos

SISTEMAS DISTRIBUIDOS

GRADO EN INGENIERÍA INFORMÁTICA

UNED

PRESENTACIÓN

- Proyecto básico. Chat abierto.
- Servicios que ofrece el servidor:
  - Escribir mensaje: escribir(mensaje) → nada
  - Leer mensajes: leer(indice) → mensajes
- Veremos:
  - Arquitectura RMI:
    - Interfaz Remota
    - Servidor que implementa la interfaz
    - Cliente que usa la interfaz
  - Ejecución

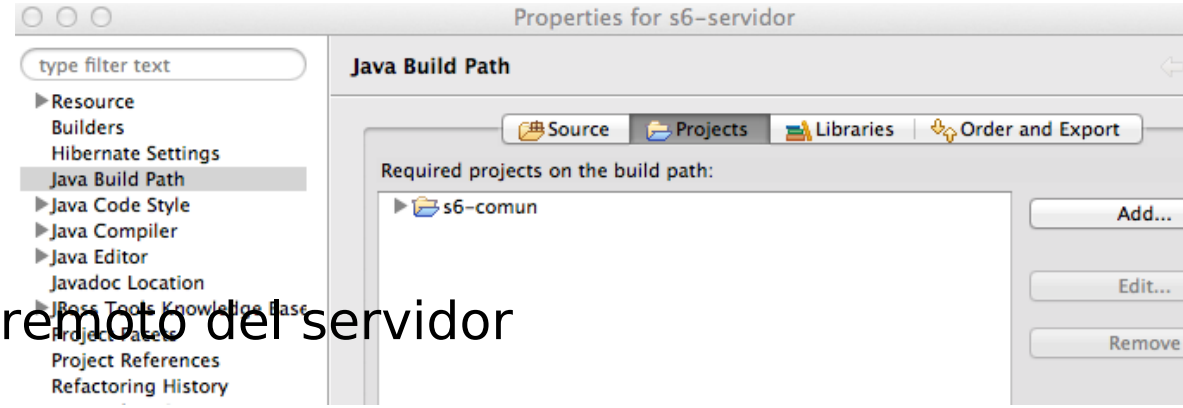
Arquitectura RMI de nuestra app

- Planteamos 2 proyectos Java:
- Servidor de Chat
  - Interfaz para acceso remoto del servidor
  - Implementación de la interfaz remota
  - Clase mensaje
  - Clase inicio servidor
- Cliente de Chat
  - Interfaz para acceso remoto del servidor
  - Clase mensaje
  - Clase inicio cliente

Arquitectura RMI de nuestra app

Las clases comunes las incluiremos en un proyecto que se incluirá como dependencia tanto al cliente como al servidor. No necesitamos skeletons (desde Java 2) ni stubs explícitos (desde Java 5) .

- Servidor
  - Implementación de la interfaz remota
  - Clase inicio servidor
- Cliente
  - Clase inicio cliente
- Común
  - Interfaz para acceso remoto del servidor
  - Clase mensaje
  - Clases auxiliares



Clases comunes: Interfaz Remota

- Interfaz Remota IServidor.java

```java
package ssdd.comun;

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.List;

public interface IServidor extends Remote {

    void escribir(Mensaje mensaje) throws RemoteException;
    List<Mensaje> leer(int indice) throws RemoteException;

}
```

- Interfaz que hereda o extiende de Remote
- Los métodos deben lanzar RemoteException

- Clase Mensaje.java
  - Serializable.
  - serialVersionUID

```java
package ssdd.comun;

import java.io.Serializable;

public class Mensaje implements Serializable {

    private static final long serialVersionUID = 643146247177534068L;
    private String emisor;
    private String texto;

    public String getEmisor() {
        return emisor;
    }
    public void setEmisor(String emisor) {
        this.emisor = emisor;
    }
    public String getTexto() {
        return texto;
    }
    public void setTexto(String texto) {
        this.texto = texto;
    }

}
```

Servidor

- Servidor:
  - Implementa la interfaz remota.
  - Los métodos lanzan RemoteException.
  - Almacenar Mensajes.
  - Recibir Mensaje.
  - Devolver Mensajes Pendientes.
  - No registra el objeto

```
package ssdd.servidor;

import java.rmi.RemoteException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import ssdd.comun.IServidor;
import ssdd.comun.Mensaje;

public class Servidor implements IServidor {

    private Map<Integer, Mensaje> mensajes = new HashMap<Integer, Mensaje>();

    public void escribir(Mensaje mensaje) throws RemoteException {

        if (mensajes == null) {
            mensajes = new HashMap<Integer, Mensaje>();
        }

        mensajes.put(mensajes.size(), mensaje);

        System.out.println(mensaje.getEmisor() + " envia un mensaje: " +
                           mensaje.getTexto());
    }

    public List<Mensaje> leer(int indice) throws RemoteException {

        if (mensajes == null || mensajes.size()==0)
            return null;

        if (indice>mensajes.size())
            throw new RuntimeException("√çndice de mensajes incorrecto (" + indice + ")");

        List<Mensaje> pendientes = new ArrayList<Mensaje>();

        for (int i=indice; i<mensajes.size(); i++) {
            pendientes.add(mensajes.get(i));
        }

        return pendientes;
    }
}
```

```
Servidor extends UnicastRemoteObject implements IServidor
```

Clase de Inicio de InicioServidor.java

- Creación Objeto Remoto.
- Registro de Objeto Remoto (bind).
- Heredar en Servidor o Exportar de UnicastRemoteObjet.
- Al finalizar unbind y unexport el objeto remoto.
- No realizamos carga dinámica de clases (RMISecurityManager y codebase)

```
package ssdd.servidor;

import java.rmi.registry.LocateRegistry;

public class InicioServidor {
    public static void main(String[] args) throws Exception {
        Servidor servidor = new Servidor();
        IServidor remote = (IServidor)UnicastRemoteObject.exportObject(servidor, 8888);

        Registry registry = LocateRegistry.getRegistry();
        registry.rebind("chat", remote);

        System.out.println("Chat Preparado, presione enter para terminar");
        System.in.read();
        registry.unbind("chat");
        UnicastRemoteObject.unexportObject(servidor, true);
        System.out.println("Chat cerrado");
    }
}
```

```
final Registry registry = LocateRegistry.getRegistry(8888);
registry.rebind("chat", servidor);
```

Clase Cliente. Cliente.java (1/2)

- Buscamos (lookup) el objeto en el enlazador local (devuelve Remote).
- Variable de instancia IServidor.
- Dependencia de proyecto común por IServidor y Mensaje.
- Método leerConsola para interfaz de usuario.
- Metodos enviar y recibir permiten:
  - enviar: escribir mensaje en chat
  - recibir: leer mensajes pendientes y mostrarlos en consola.
- No realizamos carga dinámica de clases (RMISecurityManager y codebase)
- LocateRegistry.getRegistry(puerto) en nuestro caso 8888.

```
servidor = (IServidor)Naming.lookup("rmi://localhost:8888/chat");
```

```
package ssdd.cliente;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.List;
import ssdd.comun.IServidor;
import ssdd.comun.Mensaje;

public class Cliente {

    private static IServidor servidor;
    private static int indice;
    private static String nombre;
    private static Console console = System.console();
    private static BufferedReader reader = new BufferedReader(
                                           new InputStreamReader(System.in));
    public static void main(String[] args) throws Exception {
            Registry registry = LocateRegistry.getRegistry();
            servidor = (IServidor)registry.lookup("chat");
            indice = 0;
            System.out.println("Introduzca su nombre:");
            nombre = leerConcola();
            String opt = "";
            do {
                System.out.println("Elija la operación:");
                System.out.println("1 - Escribir");
                System.out.println("2 - Leer");
                System.out.println("3 - Salir");
                opt = leerConcola();
                switch (opt) {
                    case "1": enviar(); break;
                    case "2": recibir();  break;
                }
            }
            while (!opt.equals("3"));
    }
```

Clase Cliente. Cliente.java (2/2)

- recibir(): Hacemos RMI al servidor y obtenemos una lista con los mensajes pendientes,
  y actualizamos el índice donde controlamos los mensajes recibidos.
- Dependencia de proyecto común por IServidor y Mensaje.
- Método leerConsola para interfaz de usuario.
- Métodos enviar y recibir permiten:
  - enviar: escribir mensaje en chat. RMI al método escribir.
  - recibir: leer mensajes pendientes y mostrarlos en consola. RMI al método leer del servidor.

```
    private static void recibir() throws RemoteException {
        System.out.println("-- Mensajes recibidos --");
        List<Mensaje> mensajes = servidor.leer(indice);
        System.out.println("Recibidos " + mensajes.size() + " mensajes nuevos.");
        for (Mensaje mensaje : mensajes)
            System.out.println(mensaje.getEmisor() + " : " + mensaje.getTexto());
        indice += mensajes.size();
        System.out.println();
    }

    private static void enviar() throws RemoteException {
        System.out.println("Introduzca el texto del mensaje:");
        String texto = leerConcola();
        Mensaje mensaje = new Mensaje();
        mensaje.setEmisor(nombre);
        mensaje.setTexto(texto);
        servidor.escribir(mensaje);
    }

    private static String leerConcola() {
        if (console != null) return console.readLine();
        try {
            return reader.readLine();
        }
        catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Ejecución (1/2)

- Ejecutar el enlazador RmiRegistry de Java.
  - Desde eclipse con RMI plugin www.genady.net
  - Desde consola. Abrimos consola y ejecutamos rmiregistry. Hoy optaremos por esta opción.

    RmiRegistry debe poder acceder a las clases de nuestro servidor, para ello podemos:
    - Incluir el directorio raíz de nuestras clases en la variable CLASSPATH
    - Ejecutar rmiregistry desde el directorio raíz. Este directorio es donde estén las clases compiladas, normalmente el directorio 'bin' de nuestro proyecto. En nuestro caso del proyecto común, que es donde esta la interfaz remota.
- Ejecutar la clase IniciarServidor desde Eclipse.
  - Si no es correcta esta configuración tendréis errores del tipo:

```
Exception in thread "main" java.rmi.ServerException: RemoteException occurred in server thread; nested exception is:
        java.rmi.UnmarshalException: error unmarshalling arguments; nested exception is:
        java.lang.ClassNotFoundException: ssdd.comun.IServidor
        at sun.rmi.server.UnicastServerRef.oldDispatch(UnicastServerRef.java:419)
        at sun.rmi.server.UnicastServerRef.dispatch(UnicastServerRef.java:267)
```

  - Si olvidáis que tenéis la clase corriendo previamente obtendréis una excepción de este tipo:

```
Exception in thread "main" java.rmi.server.ExportException: Port already in use: 8888; nested exception is:
        java.net.BindException: Address already in use
        at sun.rmi.transport.tcp.TCPTransport.listen(TCPTransport.java:328)
        at sun.rmi.transport.tcp.TCPTransport.exportObject(TCPTransport.java:236)
        at sun.rmi.transport.tcp.TCPEndpoint.exportObject(TCPEndpoint.java:411)
```

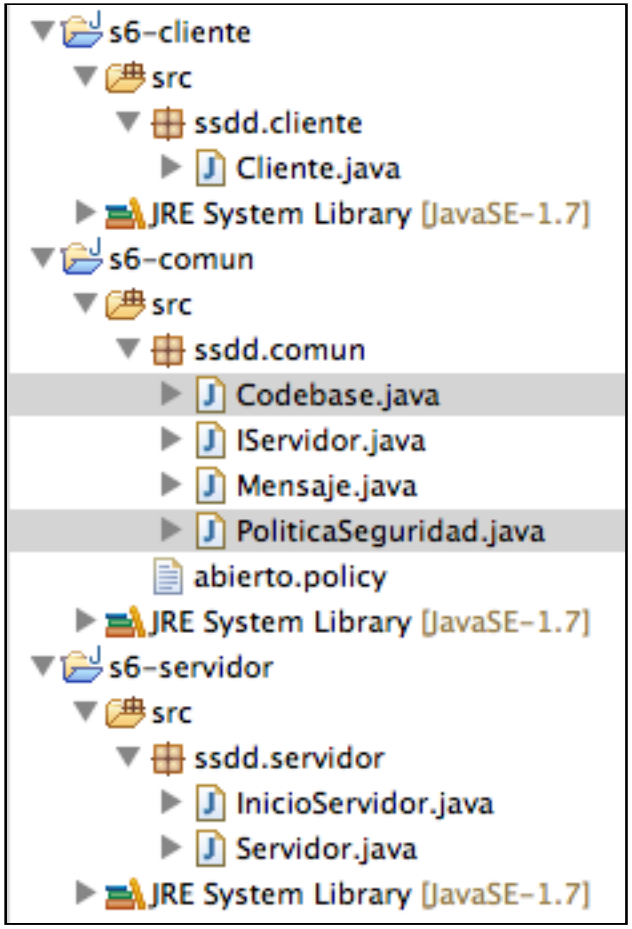
Ejecución (2/2)

- Ejecutamos varias instancias del Cliente y en cada una:
  - Introducimos nuestro nombre.
  - Podremos escribir y leer mensajes.
- Es posible que queramos utilizar la descarga dinámica de código de clases del servidor, para ello debemos incorporar:
  - Un gestor de seguridad RMISecurityManager
  - Establecer la propiedad java.rmi.server.codebase
- Para facilitar esta tarea incluiremos dos clases auxiliares en nuestro proyecto común, y modificaremos el código de las clases InicioServidor y Cliente.

```
Introduzca su nombre:
Carlos
Elija la operación:
1 - Escribir
2 - Leer
3 - Salir
1
Introduzca el texto del mensaje:
Mensaje de Prueba
Elija la operación:
1 - Escribir
2 - Leer
3 - Salir
2
-- Mensajes recibidos --
Recibidos 1 mensajes nuevos.
Carlos : Mensaje de Prueba
```

Nuevas clases PoliticaSeguridad y Codebase

- La estructura de nuestros proyectos quedará de esta forma:.



Policita de Seguridad

```java
package ssdd.comun;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStream;

public class PoliticaSeguridad {

    public static final String POLICY_FILE_NAME = "/abierto.policy";

    public static String getLocationOfPolicyFile() {
        try {
            File tempFile = File.createTempFile("s6-comun", ".policy");
            InputStream is = PoliticaSeguridad.class.getResourceAsStream(POLICY_FILE_NAME);
            BufferedWriter writer = new BufferedWriter(new FileWriter(tempFile));
            int read = 0;
            while((read = is.read()) != -1) {
                writer.write(read);
            }
            writer.close();
            tempFile.deleteOnExit();
            return tempFile.getAbsolutePath();
        }
        catch(IOException e) {
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args) {
        System.out.println(new File(getLocationOfPolicyFile()).exists());
    }

}
```

```
grant {
    permission java.security.AllPermission;
};
```

Codebase

- Esta clase nos permitirá definir de forma dinámica donde se ubican los ficheros .class de las clases Serializables y de los stubs de las clases Remote, y asignarlo a la propiedad del sistema java.rmi.server.codebase
- A partir del objeto remoto obtenemos el path.

```
package ssdd.comun;

public class Codebase {

    public static final String CODEBASE = "java.rmi.server.codebase";


    public static void setCodeBase(Class<?> c) {
        String ruta = c.getProtectionDomain().getCodeSource()
                       .getLocation().toString();

        String path = System.getProperty(CODEBASE);

        if (path != null && !path.isEmpty()) {
            ruta = path + " " + ruta;
        }

        System.setProperty(CODEBASE, ruta);
    }
}
```

Cambios en InicioServidor y Cliente

- InicioServidor: Añadimos Gestor de Seguridad y carga de propiedad codebase.

```
public class InicioServidor {
    public static void main(String[] args) throws Exception {
        // Gestor de seguridad
        System.setProperty("java.security.policy",
                PoliticaSeguridad.getLocationOfPolicyFile());
        if(System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        // java.rmi.server.codebase
        Codebase.setCodeBase(IServidor.class);

        Servidor servidor = new Servidor();
        IServidor remote = (IServidor)UnicastRemoteObject.exportObject(servidor, 8888);
```

- Cliente: Añadimos gestor de seguridad.

```
                                        new InputStreamReader(System.in));
    public static void main(String[] args) throws Exception {
        // Gestor de seguridad
        System.setProperty("java.security.policy", PoliticaSeguridad.getLocationOfPolicyFile());
        if(System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
```