# INTRODUCCIÓN

Este documento es una guía completa sobre RMI diseñada para proporcionar una referencia de los aspectos más importantes que la componen. Después de una pequeña entrada explicando de que se trata RMI y su funcionamiento, se explicaran diferentes aspectos que tiene RMI. Cada tema explicado comenzará mostrando los conceptos más relevantes del tema en cuestión, seguido de los ejercicios para que el estudiante pueda absorber de forma práctica las ideas explicadas previamente. Después de explicar todos los temas de los que se compone RMI, se han planteado 2 ejercicios más complejos para afianzar todo lo aprendido en los temas principales. Antes del final se tratará el tema de la conexión asíncronas y se enseñaran diferente reflexiones sobre el tema, además de las diferentes soluciones que ha encontrado la comunidad. Como colofón se hará una recopilación de los puntos relevantes extraídos del desarrollo de este proyecto.

## PLANIFICACIÓN

## OBJETIVOS DEL PROYECTO

Los objetivos principales de este proyecto son los siguientes:

* Organizar la información recopilada en temas claros y estructurados, que permitan a los estudiantes seguir un progreso lógico y comprender la secuencia de conceptos.
* Diseñar y plantear ejercicios prácticos que permitan a los estudiantes aplicar los conocimientos teóricos adquiridos en situaciones reales y fortalecer su comprensión práctica de RMI.
* Proporcionar a los estudiantes una base sólida en RMI, fomentar su participación activa, promover habilidades de investigación y análisis
* Crear una documentación útil y accesible para futuros usuarios.

# EJERCICIOS DE CONCEPTO

En este apartado se trabajarán los diferentes conceptos que componen RMI de una forma ordenada para construir una comprensión práctica del tema.

## ENTRADA: ¿QUE ES RMI?

RMI son las siglas de Remote Method Invocation (Invocación Remota de Métodos). Es un mecanismo que permite a un objeto que reside en un sistema (JVM) acceder/invocar a un objeto que se ejecuta en otra JVM. RMI se utiliza para construir aplicaciones distribuidas; proporciona comunicación remota entre programas Java. Se proporciona en el paquete **java.rmi.**

### Objetivos de RMI

Los objetivos de RMI son los siguientes

* Minimizar la complejidad de la aplicación.
* Preservar la seguridad de tipos.
* Recolección de basura distribuida.
* Minimizar la diferencia entre trabajar con objetos locales y remotos.

### Funcionamiento

RMI (Java Remote Method Invocation) es un mecanismo ofrecido por Java para invocar un método de manera remota. Forma parte del entorno estándar de ejecución de Java y proporciona un mecanismo simple para la comunicación de servidores en aplicaciones distribuidas basadas exclusivamente en Java. Si se requiere comunicación entre otras tecnologías, se debe emplear CORBA o SOAP en lugar de RMI.

RMI se caracteriza por la facilidad de su uso en la programación por estar específicamente diseñado para Java; proporciona paso de objetos por referencia y paso de tipos arbitrarios en los parámetros.

Toda aplicación RMI normalmente se descompone en 2 partes:

* Un servidor, que crea algunos objetos remotos, crea referencias para hacerlos accesibles, y espera a que el cliente los invoque.
* Un cliente, que obtiene una referencia a objetos remotos en el servidor, y los invoca.

A través de RMI, un programa Java puede exportar un objeto, con lo que dicho objeto estará accesible a través de la red y el programa permanece a la espera de peticiones en un puerto TCP. A partir de ese momento, un cliente puede conectarse e invocar los métodos proporcionados por el objeto.

La invocación se compone de los siguientes pasos:



* Encapsulado (marshalling) de los parámetros (utilizando la funcionalidad de serialización de Java).
* El cliente invoca el método del servidor para ejecutarlo hay. Para eso le pasa los parámetros serializados. Y se queda esperando su respuesta.
* El servidor recibe los parámetros y ejecuta el método concreto que le ha dicho el cliente. Al terminar la ejecución, el servidor serializa el valor de retorno (si lo hay) y lo envía al cliente.
* El código cliente recibe la respuesta y continúa como si la invocación hubiera sido local.

## SETUP BASICO, STUB Y SKELETON

El RMI (Remote Method Invocation) es una API que proporciona un mecanismo para crear aplicaciones distribuidas en java. Esta permite a un objeto invocar métodos en un objeto que se ejecuta en otra JVM, a esto se le denomina comunicación remota.

### Interfaces de objetos remotos

Dado que RMI es un esquema de objetos distribuidos sólo en Java, todas las interfaces de objetos están escritas en Java. Los stubs de cliente y los esqueletos de servidor se generan a partir de esta interfaz, pero utilizando un proceso ligeramente diferente al de CORBA. En primer lugar, la interfaz para el objeto remoto tiene que ser escrita como una extensión de la interfaz java.rmi.Remote. La interfaz Remote no introduce ningún método en la interfaz del objeto; sólo sirve para marcar objetos remotos para el sistema RMI. Además, todos los métodos de la interfaz deben ser declarados como lanzadores de la java.rmi.RemoteException . La RemoteException es la clase base para muchas de las excepciones que RMI define para las operaciones remotas, y los ingenieros de RMI decidieron exponer el modelo de excepción en las interfaces de todos los objetos remotos RMI. Este es uno de los inconvenientes de RMI: requiere alterar una interfaz existente para aplicarla a un entorno distribuido.

Esta comunicación remota entre las aplicaciones se hace utilizando dos objetos, el stub en la parte del cliente y el skeleton en la parte del servidor, haciendo posible la comunicación con el objeto remoto.

### Stub

El stub es un objeto que actúa como puerta de enlace para el lado del cliente. Todas las peticiones salientes se enrutan a través de él. Reside en el lado del cliente y representa el objeto remoto. Cuando el llamador invoca un método en el objeto stub, éste realiza las siguientes tareas:

* Inicia una conexión con la máquina virtual remota (JVM)
* Escribe y transmite (marshals) los parámetros a la Máquina Virtual remota (JVM),
* Espera el resultado
* Lee (unmarshals) el valor de retorno o excepción.
* Por último, devuelve el valor a la persona que llama.

### Esqueleton

El esqueleto es un objeto que actúa como puerta de enlace para el objeto del lado del servidor. Todas las peticiones entrantes se enrutan a través de él. Cuando el esqueleto recibe la petición entrante, realiza las siguientes tareas:

* Lee los parámetros del método remoto
* Invoca el método en el objeto remoto real, y
* Escribe y transmite (marshals) el resultado a la persona que llama.
* En el SDK de Java 2, se introdujo un protocolo stub que elimina la necesidad de esqueletos.



### Estado actual de la arquitectura Skeleton Stub



La arquitectura java RMI, se actualizo con la version de [java 1.5](https://nick-lab.gs.washington.edu/java/jdk1.5b/guide/rmi/relnotes.html) , la cual introdujo los Stubs dinamicos eliminando la necesidad de utilizar Skeleton. Quitando totalmente la necesidad de compilar con la herramienta rmic.

### EJERCICIOS

#### 001- Hello world con naming (modulo antiguo)

**Enunciado**

Ejecuta el programa 001- Hello world con naming, y entiende su funcionamiento.

**Explicacion**

Este ejercicio es para ilustrar, que rmi ademas del codigo, necesita levantar el servidor de forma separada, el proceso para levantar el servicio se explicara en el siguiente ejercicio.

En referencia a lo que pasa en el ejercicio, como el codigo se intenta ejecutar sin haber realizado la compilacion del codigo ni el levantamiento del servicio Rmi, eclipse no sera capaz de ejecutar el codigo, y fallara, ya que no sera capaz de establecer la conexion entre el cliente y el servidor.

**Output**



#### 002 - Compilacion y ejecucion (modulo antiguo)

**Enunciado**

Haz funcionar el ejemplo anterior haciendo todo el proceso de compilacion. Las instrucciones son las siguientes:

1. Habre CMD ,colocate en la carpeta src del proyecto
2. Ejecuta el comando javac <nombre del archivo java> , y compila todos los archivo .java del src
3. Crea los objetos Stub y skeleton con rmic AdderRemote
4. Ejecuta el comando rmiregistry 5000 , para ejecutar el servidor rmi en el puerto 5000
5. Sin cerrar el terminal anterior abre un nuevo terminal, en el src y ejecuta java MyServer  , para empezar el servidor.
6. En otra terminal nueva con java MyClient  empieza el cliente.

**Explicacion**

En este ejercicio se intenta enseñar el proceso de levantar el servicio de RMI.

La funcion de los comandos del enunciados es la siguiente:

* Javac <archivo .java> -> Compila el codigo fuente.
* rmic AdderRemote   -> Crea los objetos stub y skeleton utilizando la herramienta rmic .
* rmiregistry 5000  -> Inicia el servicio rmi en el puerto indicado (por defecto es 5000),
* java MyServer  -> Inicia la ejecucion del servidor
* java MyClient  -> Inicia la ejecucion del cliente

Como curiosidad, si se levanta este servicio rmi en la linea de comandos, la ejecucion del servidor y el cliente se pueden hacer por ejecucion normal en eclipse. Debido a que el servicio permite la interconectividad entre servidor y cliente a traves del puerto 5000.

**Output**

Terminal 1 el de la compilacion y levantamiento del servicio



-> El metodo rmiregistry que se estas utilizando para levantar el servicio esta deprecated y el sistema lo notifica.



Terminal 2: el server



Terminal 3: el cliente



-> Ejecucion alterna en eclipse si el servicio rmi esta levantado



## NAMING REGISTRY

El servicio de nombres RMI (RMI registry) es un servidor que permite a una aplicación buscar objetos que están siendo exportados para su uso mediante llamadas a métodos remotos.

### Funcionamiento

Una vez que el objeto ha sido localizado, ya se puede utilizar utilizando la misma sintaxis que una llamada a un método local. Para encontrar los objetos, RMI utiliza un servicio que mantiene una tabla de direcciones de objetos remotos que están siendo exportados por sus aplicaciones de la siguiente forma.



A todos los objetos se les asigna nombres únicos que se utilizan para identificarlos. Algunos métodos pueden llamarse desde la interfaz rmi.registry.Registry, o desde la clase rmi.Naming, que permite añadir, eliminar y acceder a objetos remotos en la tabla de registro de objetos. El servidor del servicio de nombres registra los objetos mediante llamadas a bind() o rebind() sobre una instancia de un registro del objeto que está siendo exportado. De forma alternativa, rebind() reemplaza un objeto antiguo con un nombre dado, con un nuevo objeto.

Despues Para invocar un objeto remoto, el cliente necesita una referencia de ese objeto. En ese momento, el cliente obtiene el objeto del registro utilizando su nombre bind (mediante el método lookup()).



Hay dos formas de iniciar el servicio de nombres: una es usando la aplicación que proporciona java (rmiregistry), y otra es escribir nuestro propio servicio de nombres usando las clases e interfaces de java.rmi.\*

Normalmente la aplicación de servicio de nombres se inicia como una aplicación en background. Por defecto, se ejecutará sobre el puerto 1099, pero se puede seleccionar otro puerto cuando se lanza la aplicación.

### Serializacion en Rmi

RMI facilita el olvidarse de los detalles de la transmisión de datos y centrarse en el diseño de la lógica de la aplicación. Puesto que permite acceder a un objeto remoto como si de un objeto local se tratase.

Esta facilidad de RMI es dada por la serialización de objetos que se hace usando la interfaz Serializable que el paquete java.io incluye. Que pueden convertir un objeto en un flujo de bytes y volver a ensamblar los bytes en una copia idéntica del objeto original.

Utilizando estas clases, un objeto en un proceso puede ser serializado y transmitido a través de una conexión de red a otro proceso en un host remoto. El objeto (o al menos una copia del mismo) puede entonces ser reensamblado en el host remoto para su posterior uso.

### EJERCICIOS

#### 003-Helloworld con Registry

**Enunciado**

Basandote en el ejercicio 001- Hello world con naming , actualizalo sustituyendo Naming.lookUp, por:

Cliente:

Registry registry = LocateRegistry.*getRegistry*(1099);

Hello stub = (Hello) registry.lookup("Hello");

Server:

registry.bind("Hello", obj);

Nota: Recuerda que se tiene que poner la misma clave para que se comuniquen.

**Explicacion**

La finalidad del ejercicio, es demostrar la diferencia entre la forma de conexion usando Naming, y usando NamingRegistry. Y para marcar esa diferencia se pretende actualizar el primer ejercicio realizado con Naming, poniendo un Naming registry en su lugar. Esto ademas de simplicar el proceso quita la necesidad de compilacion y de utilizar la linea de comandos, ademas de permitir idenfificar cada servicio facilmente con una clave unica, dando la opcion de poder utilizar varios servidores en caso de que sea necesario.

**Output**



#### 004-Calculadora

**Enunciado**

Se quiere desarrollar la capacidad de calculo del programa anterior, haciendo que sea posible realizar diferentes calculos en funcion de lo necesario. Modifica el programa 003-HelloWorld con Registry, para que pueda realizar las siguientes acciones:

1. Sumar
2. Restar
3. Multiplicar
4. Dividir
5. Terminar el programa.

**Explicación**

Este ejercicio una extensión del ejercicio anterior que implementa la misma estructura que 003-HelloWorld con Registry . Esta pensada para ilustrar que es posible hacer múltiples llamadas a un objeto registrado. Y a los diferentes metodos que implementa.

**Output**



#### 005-SimulacionCoche

**Enunciado**



Construye la siguiente arquitectura cumpliendo las siguientes especificaciones:

* La Cpu esta conectada a todos los elementos.
* Las ruedas delanteras R1 y R2, pueden girar derecha o izquierda.
* Las ruedas traseras R3 Y R4 pueden ir adelante o atras.
* Todas las ruedas pueden frenar.
* La bateria se consume un 5% cada vez que las ruedas van adelante o atras.
* Las acciones tambien se tiene que reflejar en los componentes (servidores)

Los opciones de control en la CPU son los siguientes:

1.- Adelante

2.- Atras

3.- Girar

4.- Parar

5.- Comprobar Bateria

0.- Salir

**Explicación**

Este es un ejercicio se intenta trabajar la conexion de un cliente a multiples Stubs (servidores) con diferentes implementaciones. Esto nos sirve para hacer acciones coordinadas que permiten controlar el coche. Tambien sirve para trabajar correctamente la conexiones de cliente y servidor.

Como referencia la tabla de conexiones usada en el ejercicio es la siguiente.

|  |  |  |
| --- | --- | --- |
| ELEMENTO | PUERTO | NOMBRE REGISTRO |
| R1 | 1091 | direccion1 |
| R2 | 1092 | direccion2 |
| R3 | 1093 | motor3 |
| R4 | 1094 | motor4 |
| Bateria | 1095 | bateria |

Notas adicionales:

cada puerto solo puede ser usado por 1 proceso a la vez.

**Output**

**Opciones 1,2**

#### 

**Opcion 3**



**Opcion 4**



**Opcion 5**

#### 

#### 006- Saludador Serializado

**Enunciado**

Crea un programa de hello world, pero en vez de hacerlo normalmente, hazlo pasando el objeto saludador del cliente al Servidor.

**Explicación**

En este ejercicio se ha puesto como ejemplo de como pasar objetos propios del cliente al servidor, para ello se ha creado el objeto Saludador y como es de nuestra creación, es necesario hacer que implemente la interfaz Serializable para que se pueda transmitir. Ya que de no aplicarlo saltara la excepcion:



**Output**



#### 007- Comprobador de Urls

**Enunciado**

Crea otro programa, que cuando le pase una Url, di si es valida o no. Si es valida tiene que enseñar el html de la pagina web a la que hace referencia en pantalla.

Y si no es valida, que diga que no es valida.

**Explicación**

Este el primero de los ejercicios para pasar objetos nativos de java desde el cliente al servidor, ya que estos objetos implementan Serializable de fabrica. En este caso se ha pasado un objeto String, para el url.

Como nota, para este ejercicio se ha usado el objeto URL de la biblioteca de java. Este objeto al inicializarse, en caso de que el url usado para la inicializacion no sea valido saltara con un UrlMalformedException, que es lo que se usara para decir si es valido o no.

Mas informacion del objeto Url, en el [link.](https://lineadecodigo.com/java/leer-una-url-con-java/)

**Output**





#### 008- Meter notas del curso y calcula la media

**Enunciado**

Realiza un programa en JAVA en el que le pidas al usuario las notas de las 5 asignaturas del semestre y las guarde en un fichero. Posteriormente leerá el fichero y despues de mandar los objetos asignaturas al servidor te calculará la nota media del curso.

**Explicación**

En este ejercicio continuando con el anterior, tambien tiene la finalidad de mostrar la serializacion de los objetos de java. En este caso para los objetos List y Double.

**Output**



#### 009- Transmisión de archivos entre cliente y server.

**Enunciado**

Crea un programa que lea un archivo en el cliente y despues tranfiera el archivo al servidor. Tiene que confirmar si la transferencia se ha realizado correctamente o no.

**Explicación**

En este ejercicio a diferencia de los anteriores, se intenta ilustrar que mediante el tipo byte no es necesario usar la interfaz Serializable, ya que byte es el tipo de datos que se usa en la transferencia, por lo que la transferencia se hace directamente.

Ademas es necesario mencionar que al servidor se le pasa unicamente el contenido . Esto es debido a que el objeto File solo es una representacion del pathName del archivo de la maquina de origen, no de su contenido. Y en un servidor remoto no es posible manipular el archivo del origen desde el servidor, ya a que al cambiar de maquina se pierde el acceso a el fichero original.

Por eso es mas adecuado y fiable pasar solo el contenido.

**Output**



## SEGURIDAD

Java realiza un importante control de seguridad en los programas, que impide que programas escritos en dicho lenguaje puedan dañar nuestra información o accedan a información privada.   Para este fin se utilizan las siguientes medidas de seguridad:



**Cargador de clases**. Este elemento se encarga de separar las clases que carga para evitar ataques.El orden de búsqueda de las clases es: sistema, extensión y usuario. El cargador de clases es una clase Java y puede ser extendida para definir cargadores de clases especiales, pero sólo por las aplicaciones; si un applet pudiera definir su propio cargador podría modificar el cargador del sistema y potencialmente tomar la máquina en la que se ejecuta el navegador.

**Verificador de archivos de clases**. La función de este componente es validar los bytecodes. El sistema distingue entre 2 tipos de código

* De confíanza: Estas son generalmente las clases del sistema y las validadas por el usuario y no se validan.
* No confíables: Estan se validan.

**Gestor de seguridad.** Se encarga de comprobar el acceso en tiempo de ejecución. Es una clase del sistema y puede ser extendida por las aplicaciones.

#### Ficheros de políticas de seguridad

Los ficheros de política (policy) establecen la políticas de seguridad (permisos) que se llevarán a cabo en los programas Java que Si no se especifica nada, la norma afectará a cualquier programa del sistema. Podemos restringir el número de programas afectados por la norma especificando:

* **Base de códigos (CodeBase)**: Se refiere a la localización en el sistema (path o Url) de las implementaciones de las interfaces que utiliza el codigo, y le dice de donde se descarga esa implementacion.
* **Firmado por (Signed by):** Aquí podemos especificar quien debe haber firmado el programa para que la norma le sea aplicable.

Para utilizar el fichero de políticas que hayamos definido, podemos optar bien por añadirlo a este fichero de propiedades de seguridad, o bien añadirlo como propiedad del sistema el ejecutar el intérprete Java.De esta manera estamos forzando a que se utilice la política definida en el fichero.

En una aplicación Java por defecto no se instala ningún gestor de seguridad. Si queremos imponer restricciones de seguridad tendremos que definir y implementar un gestor de seguridad.

#### Gestor de seguridad

El gestor de seguridad (SecurityManager) será el objeto encargado de determinar si cierta operación es permitida o no, impidiendo su realización en tal caso. Cuando una aplicación carga un gestor de seguridad todas las acciones que vaya a realizar sujetas a posibles restricciones de seguridad se comprobarán en dicho gestor antes de ser realizadas.

En las aplicaciones independientes no se carga por defecto ningún gestor de seguridad, pero podremos hacer que se cargue bien indicándolo en la línea de comando como hemos visto en el punto anterior, o bien desde el mismo código de nuestra aplicación.

El gestor de seguridad por defecto (clase SecurityManager) será el gestor que carguen los Applets. Este gestor de seguridad seguirá la política de seguridad indicada en los ficheros de políticas que hayamos indicado (tanto en el fichero de propiedades de seguridad como en la línea de comando).

Además de poder utilizar el gestor de seguridad definido por defecto, también podremos definir nuestro propio gestor de seguridad creando una subclase de SecurityManager y sobrescribiendo en ella una serie de métodos para darles el comportamiento que queramos.

Para instalar un gestor de seguridad, de forma que sea utilizado para establecer las operaciones permitidas, utilizaremos el método **setSecurityManager** de la clase System.

#### Gestor de seguridad en Rmi

Uno de los problemas más comunes que uno encuentra con RMI es un fallo debido a restricciones de seguridad de java estandar.

En cambio para java Rmi se utiliza la clase RMISecurityManager que aplica la política de seguridad a las clases que se cargan como stubs para objetos remotos, sobrescribiendo todos los métodos de comprobación de acceso relevantes del SecurityManager estandar de java.

#### Restricciones de seguridad en Applets y aplicaciones Java

Los Applets son programas Java que pueden descargarse de fuentes remotas y ejecutarse de forma automatica.

Por el peligro que representan este tipo de programas, cuentan con una serie de restricciones importantes:

* No pueden acceder a métodos nativos.
* No pueden leer ni escribir en ficheros de la máquina local donde se ejecute. Sólo puede acceder a ficheros si proporciona la URL absoluta del fichero.
* No pueden establecer conexiones de red a ningún host distinto al host desde el que nos hemos descargado el Applet.
* No pueden ejecutar programas en la máquina donde se está ejecutando.
* No pueden leer las propiedades del sistema.

Podremos cambiar la configuración de forma que se eliminen algunas de estas restricciones, estableciendo los permisos necesarios para ello. De la misma forma, aunque las aplicaciones no tengan por defecto estas restricciones de seguridad, podremos hacer que también se vean sujetas a ellas.

#### Descarga de codigo/clases remoto

Debido a las restricciones explicadas en el apartado de Restricciones de seguridad en Applets y aplicaciones, la descarga dinamica de las clases de destinos remotos esta totalmente restringido en las politicas por defecto de java .

Para realizar este tipo de operaciones se requiere que haya un gestor de seguridad, que sobrescriba los permisos vigentes. Por ejemplo en RMI se descargará una clase Serializable desde otra máquina sólo si hay un gestor de seguridad y el gestor de seguridad permite la descarga de la clase desde esa máquina.

Esta acción se controla mediante el valor de la propiedad **java.rmi.server.useCodebaseOnly**.

Establecer esta propiedad a false habilita la carga remota de código, lo que incrementa el nivel de riesgo de seguridad del sistema. Aunque su valor por defecto es true.

**Proceso de ejecutar en diferentes maquinas**

-> Primero se tiene que compilar el codigo, esto se puede hacer manualmente, o puedes encontrar el codigo compilado en la carpeta bin, del proyecto.

-> necesitamos las .class y la estructura de carpetas del proyecto

-> Para ejecutar, los archivos se necesita levantar la vm usando el comando **java <classpath>,**



Nos colocamos en la carpeta raíz de donde esta la estructura de carpetas (que ha sido sacada del bin del proyecto, separando la ejecución del cliente y el servidor, en 2 carpetas separadas. Que para la ejecución remota solo es necesario llevarlas a diferentes maquinas y hacer los siguiente:

#

|server

|Server.class



**Y hasi lanzamos el server, lo mismo con el cliente**

**#**

**|client**

**|Client.class**



-> La seguridad tiene sentido en el tiempo de ejecucion, solo para definir de donde se hace la descarga de las implementaciones de las interfaces que conoce el servidor.

Estando esta implementacion en el cliente (ejemplo Compute Engine).

#### Ejercicios

#### 010- Security hello world example

**Enunciado**

Implementa Rmi Security en el ejemplo de hello world. Para eso realiza los siguientes pasos

* Cambia las propiedades de la VM para habilitar la descarga dinamica del codigo en la aplicacion.
* Aplica el RMISecurityManager, y sobreescribe el SecurityManager estandar de java
* Aplica una politica de seguridad, que habilite todos los permisos.

Finalmente sigue los pasos para hacer una ejecucion remota en otra maquina.

**Explicacion**

La propiedad que posibilita la descarga dinamica de codigo va unicamente en el servidor.

Y para la ejecucion remota se recomienda, tomar los .class despues de hacer la ejecucion en eclipse, ya que genera los .class del codigo despues de cada ejecucion de codigo.

Ademas, en mi caso la ejecucion del servidor la he hecho en mi pc, y le he puesto para que busque mi ip al cliente, para que la busqueda sea mas facil.

NOTA: para la ejecucion ten cuidado con el firewall, del ordenador y de la red en la que estas.

**Output**

## CARGA DINAMICA

Una de las características más significativas de la plataforma Java es su capacidad para descargar dinámicamente software Java desde cualquier URL (Uniform Resource Locator) a una máquina virtual Java (JVM) ejecutándose en un proceso diferente, normalmente ubicado en un sistema físico distinto. Java RMI utiliza esta capacidad para descargar y ejecutar clases sobre sistemas en los que dichas clases nunca han sido instaladas. Usando el API de RMI, cualquier máquina virtual, puede descargar cualquier fichero de clases Java, incluyendo las clases stub, y por lo tanto pueden ejecutar llamadas a métodos remotos sobre un servidor remoto utilizando los recursos del sistema servidor.

Un codebase puede definirse como una fuente, o un lugar, desde el que cargar clases en una JVM. Cualquier programa que tenga que enviar un objeto a otro, antes debe fijar la propiedad codebase, para que así el receptor pueda conocer desde donde puede descargar dicho código, si no lo tiene disponible de forma local. El receptor, una vez de-serializado el objeto, obtendrá el codebase y cargará el código desde la localización correspondiente.

#### FUNCIONAMIENTO

Mediante RMI, las aplicaciones pueden crear objetos remotos que aceptan llamadas a métodos desde clientes a otras JVMs. Para que un cliente pueda llamar a métodos de un objeto remoto, el cliente debe tener una forma de comunicarse con el objeto remoto. RMI utiliza clases especiales denominadas stubs que pueden ser descargadas al cliente para que pueda comunicarse con el objeto remoto.

Al igual que los applets, las clases necesarias para ejecutar llamadas a métodos remotos pueden descargarse desde URLs de tipo "file:///", pero esto requiere que el cliente y el servidor residan en el mismo host físico, a menos que el sistema de ficheros referenciado por la URL esté accesible utilizando algún otro protocolo, como NFS.

Normalmente, las clases necesarias apra ejecutar llamadas a métodos remotos deberían hacerse accesibles desde un recurso de red, como por ejemplo un servidor HTTP o FTP.

La descarga de los stubs RMI funciona de la siguiente manera:



1. El codebase de los objetos remotos se especifica en el servidor del objeto remoto mediante la propiedad java.rmi.server.codebase. El servidor RMI registra un objeto remoto, asignándole un nombre, en el registro RMI. El codebase establecido en la JVM del servidor se "anota" en la referencia al objeto remoto en el registro RMI.
2. Cuando el cliente RMI solicita una referencia a un objeto remoto, se le devuelve una instancia del stub del objeto remoto. Esta referencia será utilizada por el cliente para realizar llamadas a métodos del objeto remoto.
3. El registro RMI proporciona una referencia (la instancia del stub) a la clase solicitada. Si la definición de la clase para el stub se encuentra localmente en el CLASSPATH del cliente, que se busca antes que el codebase, el cliente cargará la clase localmente. Sin embargo, si la definición del stub no se encuentra en el CLASSPATH del cliente, este intentará recuperar la definición de la clase a partir del codebase del objeto remoto.
4. Luego, el cliente solicita la definición de la clase desde el codebase. El codebase utilizado por el cliente es el URL asociado con la instancia del stub cuando la clase del stub fue cargada por el registro en el paso 1.
5. La definición de la clase del stub y cualquier otra clase necesaria se descargan en el cliente. Los pasos 4 y 5 son los mismos que el registro tuvo que realizar para cargar las clases del objeto remoto durante el registro inicial. Cuando el registro intentó cargar la clase del stub del objeto remoto, solicitó la definición de dicha clase consultando el codebase asociado con el objeto remoto.
6. Ahora el cliente tiene toda la información necesaria para invocar métodos en el objeto remoto. La instancia del stub actúa como un proxy del objeto remoto que existe en el servidor.

#### Ejercicios

#### 011- Calculo dinamicos finanzas

**Enunciado**

Usando el ejemplo 011-Compute Engine Carga Dinamica (base),sin hacer modificaciones en la parte del servidor (ni en las interfaces), haz que los clientes puedan calcular lo siguiente:

Interes compuesto:

El total a devolver de un prestamo:

**Explicacion**

En este ejercicio se intenta trabajar, el uso de multiples clientes que implementan el interfaz Compute,teniendo cada uno su propia implementacion. Y asi ver que sin hacer cambios es posible usar el servidor, para ejecutar diferentes funciones.

**Output**





#### 012- Conversor de monedas

**Enunciado**

Desarrolla una aplicación de conversión de monedas. La aplicación debe permitir convertir una cantidad de EUROS a los siguientes 9 tipos de monedas diferentes:Dolar Estadounidense(USD), Yen japones(JPY),libra esterlina (GBP), dólar australiano (AUD), dólar canadiense (CAD), franco suizo (CHF), renminbi chino (CNY),dólar hongkonés (HKD), dólar neozelandés (NZD).

**Explicacion**

En este ejercicio se intenta trabajar, el uso de los enum como elemento comun entre el cliente y servidor, y tambien hacer un ejemplo simple de transacciones monetarias.

**Output**



#### 013- AccesoEntreClientes

**Enunciado**

Desarrolla una aplicación cliente-servidor con carga dinámica de métodos, donde el cliente 1 lanza un método que necesita datos del cliente 2. El cliente 2 ejecuta un método para proporcionar los datos al servidor, y el servidor ejecuta el método del cliente 1 con los datos del cliente 2 recibidos, enviando el resultado al cliente 1.

**Explicacion**

En este ejercicio se intenta trabajar, el tener la implementacion de los metodos de calculo y gestion, separada de la implementacion de los datos. Haciendo una distribucion de ambos en diferentes clientes, que se comunican a traves de un servidor, que solo ejecuta lo que le mandan sin saber nada de nada.

**Output**

 -> recibimos los datos del Baul en ComputePi

# EJERCICIOS COMPLETOS

## Ejercicio 014- Simulacion repartidores

**Planteamiento**

Desarrolla una aplicación que simule un sistema de gestión de pedidos de hamburguesas. El sistema consta de un cliente que genera un pedido y lo envía a una empresa de gestión de pedidos. La empresa de gestión de pedidos cuenta con tres empresas de reparto subcontratadas: Subcontrata 1, Subcontrata 2 y Subcontrata 3.

El objetivo del sistema es garantizar que el pedido siempre sea entregado por la subcontrata con el número más bajo. Y que el gestor de pedidos enseñe la cantidad total facturada hasta el momento.

Ademas el sistema debe ser capaz de manejar situaciones en las que el servidor de una subcontrata esté caído. En ese caso, se debe delegar el pedido a la siguiente subcontrata con el número más bajo que esté operativa. El objetivo es garantizar que el pedido se entregue lo más rápido posible. En caso de que no haya ninguna subcontrata disponible el pedido se rechazara.

La aplicación debe ser capaz de mostrar el estado de las subcontratas (encendidas o apagadas) y notificar al cliente de la situación de su pedido. Finalmente se deben implementar manejo de errores para garantizar la integridad y confiabilidad del sistema.

SE HA QUITADO LA PARTE DE SEGURIDAD PORQUE SINO NOS QUEDAMOS SIN NADA PARA EL TOCHO DE CARGA DINAMICA

-> REVISA EL EJERCICIO AL REPASAR LA DOCUMENTACION DE LOS EJERCICIOS GRANDES

**Explicacion**

La llamada de funciones de un servidor a otro es totalmente procedural, y funciona de la siguiente manera.



Los datos de conexion de los diferentes servidores es la siguiente.

|  |  |  |
| --- | --- | --- |
|  | Puerto | Nombre |
| Servidor Gestion | 1090 | ServerGestion |
| Subcontrata1 | 1091 | Subcontrata1 |
| Subcontrata2 | 1092 | Subcontrata2 |
| Subcontrata3 | 1093 | Subcontrata3 |

El cliente unicamente se conecta al servidor Gestion.

Para hacer la comprobación de los servidores, solo se llama a todos los servidores Subcontratas, y si falla se recoge en una excepcion, y si conecta se le manda el pedido y se corta el bucle.

**Output**

## Ejercicio 015- Simulacion banco

**Planteamiento**

Desarrolla una aplicación utilizando RMI que simule un sistema bancario. El sistema consta de un cliente y dos servidores: un servidor de banco y un servidor de bóveda.

El cliente debe ser capaz de realizar las siguientes transacciones:

1. Consultar saldo.
2. Realizar depósito.
3. Realizar retiro.
4. Transferir fondos.
5. Solicitar un préstamo a 6 meses.

El servidor de banco recibe las instrucciones de transacciones bancarias por parte de los clientes y las envía al servidor de bóveda para que este manipule los datos de todas las cuentas. Además, registra un registro de todas las transacciones que pasan por él.

El servidor de bóveda se encarga de gestionar la bóveda y realizar todas las operaciones necesarias en las cuentas bancarias. Debe tener en cuenta todos los posibles casos que pueden surgir durante las transacciones.

Toda la comunicación entre el cliente, el servidor de banco y el servidor de bóveda debe realizarse mediante carga dinámica, utilizando RMI para la transmisión de objetos y llamadas a métodos remotos. Además, se deben implementar mecanismos de seguridad para proteger la comunicación y manejar errores y excepciones de manera adecuada, a fin de proporcionar una experiencia de usuario robusta y confiable.

**Explicación**

Se ha utilizado la comunicación mediante cadenas de texto (String) para permitir el envío de diferentes comandos utilizando el mismo tipo de variable. Esta elección se debe a su facilidad de manejo y flexibilidad.

En los mensajes, se ha empleado el carácter "\_" como divisor en lugar del "." para evitar problemas en comandos que contengan espacios. Esto se debe a que el uso del "." presentaba inconvenientes al procesar cantidades en formato decimal, ya que al dividir la cadena se perdían los decimales.

Al desarrollar los programas, se han considerado los siguientes casos de uso:

* No se puede retirar más dinero del disponible en la cuenta en ninguna de las transacciones.
* Es necesario verificar la existencia de los IDs de los clientes involucrados en la transacción, para realizar la operacion.
* En el caso de los préstamos, se requiere realizar una llamada al banco para verificar el estado y otra llamada para realizar o devolver el préstamo.
* Si no existe un préstamo, se debe ofrecer la opción de solicitarlo. Si el cliente acepta, se procede a realizar el préstamo.
* Al devolver el préstamo en su totalidad, el estado del préstamo debe cambiarse a "false". En caso de devolver una cantidad mayor que el monto restante del préstamo, solo se cobrará el importe del préstamo de la cuenta.

**Output**

# RMI ASINCRONO

# CONCLUSIONES

# BIBLIOGRAFIA