**PLANIFICACION DE TAREA Y TIEMPOS**

-> Hacer el diagrama gant

- Aprendizaje Investigacion

- Extraer las caracteristicas principales que se van a utilizar para los ejercicios

- Hacer los ejercicios

-Documentar los ejercicios

+ En palabras suyas hay que dedicarle un tercio del tiempo a documentar. Y lo demas ha hacer ejercicios

+ PONER planificacion por semanas y cuantas horas a cada tarea

- 20-21 : Planteamiento del problema y poner de acuerdo las cosas

- 24-28

- 1-5

**ENTRADA DE JAVA RMI (usar diapositivas de clase para completar)**

- Siglas

- Entorno en el que se creo

- Funcionamiento

- Estado hoy en dia ( sirve para hacer conexiones y acciones concretas de forma eficiente,.

Muy usado en entornos java, ejemplo (esta detras del funcionamiento de los @Bean en java y SpringBoot)):

**EJERCICIOS DE CONCEPTO**

*SETUP BASICO, STUB Y SKELETON*

Ejemplo en eclipse

* - Ejercicio base de funcionamiento basico [(tutorial de oracle)](https://docs.oracle.com/javase/tutorial/rmi/overview.html)  (como los de clase) (en local) (se puede usar el siguiente para hacer la explicacion, y el ejercicio en local -> [ver](https://www.javatpoint.com/RMI) ) [Una version con registry](https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/hello/hello-world.html) ejecucion en eclipse.

Ejemplo de consola

* Usar Naming.rebind para unir los 2 programas a traves un nombre de red ( en diferente dispositivos) -> [ver](https://www.javatpoint.com/RMI)
* Compilacion de los programas del tutorial de oracle probar ha hacer ejecucion en diferentes ordenadores. Usar java 1.8 o inferior, porque sino no funcionara.

*NAMING REGISTRY Y SERIALIZACION EN RMI*

- Basico de mandar int , char ..... (las anteriores entrarian aqui)

- Aplicar Serializable, y mandar objetos enteros. (usar ejemplo de aita)

- Hacer un server al que accedan varios clientes a la vez (aplicar concurrencia).

*SEGURIDAD*

-> Una vez que consiga hacerlos funcionar en remoto o en diferente ordenador

* -Implementar la seguridad. Aplicacion de la clase RMISecurityManager, en elejemplo anterior

*CARGA DINAMICA*

* - Carga dinamica, usando la clase RMIClassLoader para implementar el Stub manualmente..
* Mirar Compute engine ejemplo de moodle ejemplo clase, modificarlo para aplicarlo en tema real. Es un server mas potente que los clientes, para poder hacer tareas mas importates.
  + La implementacion de la interfaz esta en el cliente y se ejecutaba en el server, se mandaba/cargaba

***EJERCICIO GRANDE***

* Plantearlo desde el punto de vista de la carga dinamica. Tiene que sea ejemplos que aprobechen y justifiquen el uso de Rmi.

BIBLIOGRAFIA

# PLANIFICACIÓN

# ENTRADA

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 debe utilizarse 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 parametros.

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 metodo del metodo del servidor para ejecutarlo hay. Para eso le pasa los parametros serializados. Y se queda esperando su respuesta.
* -El servidor recibe los parametros y ejecuta el metodo 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.

# EJERCICIOS DE CONCEPTO

En este apartado se trabajaran los diferentes conceptos que componen Rmi de una forma ordenada para construir una comprensión practica del tema.

## 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**

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**

La funcion de los comandos 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**

Con el codigo mostrado, se pretende actualizar el primer ejercicio realizado, poniendo un servidor de nombres que 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**

Es una extensión del ejercicio anterior que implementa la misma estructura. Esta pensada para ver que es posible llamar múltiples veces a un objeto registrado, para hacer las diferentes acciones mencionadas.

**Output**



#### 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 creado el objeto Saludador y este se transfiere directamente al servidor para ejecutar el método.

Como el objeto Saludador 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, si es valida, diga que es valida y enseñe 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**

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/)

Ademas el String que se le pasa no hace falta serializarlo, debido a que la clase String ya implementa Serializable de fabrica.

**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 caso es un ejemplo donde se mandan y se reciben objetos. Pero al ser estos objetos ( el List como el Double) de java, ya son elementos Serializables no hace falta implementarlo.

**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 al servidor se le pasa unicamente el contenido debido a que el objeto File de java pese a ser serializable, no es conveniente hacer uso de el durante la transferencia, debido a que File solo es una representacion del pathName del archivo en cuestion en la maquina de origen, no de su contenido. Y en un servidor remoto no es posible manipular el archivo del origen desde el servidor, debido a que al cambiar de maquina se pierde el acceso a el fichero original. Por eso es mejor pasar su contenido.

Para la transferencia se ha pasado el contenido del mensaje a bytes para poder transportarlo directamente.

**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 los programas afectados por la norma.
* **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.

La aplicación de esta propiedad sera importante en el siguiente tema.

#### Ejercicios

#### 010- Security hello world example

**Enunciado**

Implementa Rmi Security en el ejemplo de hello world.

**Explicacion**

**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 por el servidor del objeto remoto fijando la propiedad java.rmi.server.codebase. El servidor RMI registra un objeto remoto, enlazado a un nombre, en el registro RMI. El codebase fijado en la JVM del servidor se "anota" en la referencia al objeto remoto en el registro RMI.

2. El cliente RMI solicita una referencia a un objeto remoto. Dicha referencia (la instancia del stub del objeto remoto) es la que el cliente usará para realizar llamadas a métodos del objeto remoto.

3. El registro RMI devuelve una referencia (la instancia del stub) a la clase solicitada. Si la definición de la clase para la instancia stub puede encontrarse localmente en el CLASSPATH del cliente, el cual se busca siempre antes que el codebase, el cliente cargará la clase localmente. Sin embargo, si la definición para el stub no se encuentra en el CLASSPATH del cliente, éste intentará recuperar la definición de la clase a partir del codebase del objeto remoto.

4. El cliente solicita la definición de la clase a partir del codebase. El codebase que usa el cliente utiliza el URL que fué asociado con la instancia del stub cuando la clase del stub fué cargada por el registro en el paso 1.

5. La definición de la clase para el stub (y para cualquier otra clase(s) que sea(n) necesaria(s) es descargada en el cliente.

Los pasos 4 y 5 son los mismos que el registro tuvo que realizar para cargar las clases del objeto remoto, cuando dicho objeto fue registrado. Cuando el registró intentó cargar la clase del stub del objeto remoto, solicitó la definción de dicha clase consultando el codebase asociado con dicho objeto remoto.

1. Ahora el cliente tiene toda la información que necesita para invocar métodos sobre el objetor remoto. La instancia del stub actúa como un proxy del objeto remoto que existe en el servidor.

#### Usos adicionales del codebase en RMI

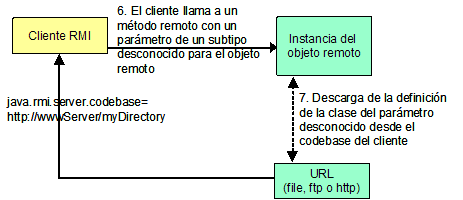
Además de utilizar el codebase para descargar en los clientes los stubs y sus clases asociadas, la propiedad java.rmi.server.codebase puede utilizarse para especificar una ubicación desde la cual cualquier clase puede ser descargada, no solamente los stubs.

Cuando un cliente realiza una llamada a un objeto remoto, la llamada al método puede o no aceptar argumentos. Hay tres casos distintos que pueden darse, según el tipo de datos de los argumentos del método remoto:

Caso 1. Todos los parámetros del método (y valores de retorno) son datos primitivos, de forma que el objeto remoto sabe cómo interpretarlos. Aquí no hay necesidad de comprobar el CLASSPATH ni el codebase.

Caso 2. Al menos un parámetro del método o valor de retorno es un objeto para el que el objeto remoto puede encontrar su definición de clase localmente en su CLASSPATH.

Caso 3. El método remoto recibe una instancia de un objeto que no puede ser encontrado por el objeto remoto de forma local en su CLASSPATH. Este tipo de llamada se ilustra en la Figura 1.8 como el paso 6. La clase del objeto enviado por el cliente será un subtipo del tipo declarado como parámetro. Un subtipo puede ser: (a) una implementación de la interfaz declarada como el tipo de parámetro del método (o valor de retorno); (b) una subclase de la clase declarada como tipo del parámetro del método (o valor de retorno).



.