Sistemas Distribuidos
Grado en Ingeniería Informática en Sistemas de Información

Enseñanzas de Prácticas y Desarrollo
EPD Entregable: Remote Method Invocation

Objetivos

1. Paso de objetos como argumentos
2. Seguridad en aplicaciones Java RMI

Conceptos

1. Introducción

En esta práctica desarrollaremos una simple, pero poderosa aplicación distribuida. Básicamente la aplicación consistirá en un servidor que acepta tareas desde unos clientes, ejecuta las tareas, y devuelve los resultados. Llamaremos al servidor *compute engine*. Las tareas se ejecutan en la máquina donde reside el servidor. Este tipo de aplicación distribuida proporciona la posibilidad de que varios clientes puedan utilizar una máquina más potente o una máquina con un hardware especializado para ejecutar determinadas tareas.

La novedad con respecto a la *EPD 09 – Introducción a RMI*, es que **las tareas ejecutadas por compute engine no tienen que ser definidas cuando la aplicación se desarrolla o arranca. Se pueden crear nuevas tareas en cualquier momento y luego pasarlas a compute engine para su ejecución.** El único requisito de una tarea es que su clase implemente una interfaz específica. RMI se encargará de descargar el código necesario para ejecutar la tarea en el servidor. Luego compute engine ejecutará la tarea, utilizando los recursos de la máquina en la cual se está ejecutando.

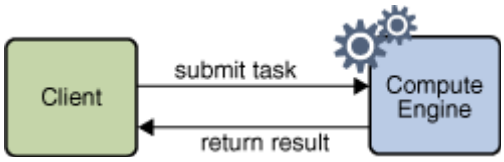
La habilidad de ejecutar tareas arbitrarias es proporcionada por la naturaleza dinámica de la plataforma Java, extendida en la red por RMI. RMI carga dinámicamente el código de la tarea en la maquina virtual de Java de compute engine y ejecuta la tarea sin ningún conocimiento previo de la clase que implementa la tarea. Una aplicación que tiene la capacidad de bajar código dinámicamente es llamada a menudo behavior-based application.

2. El servidor

El servidor compute engine acepta las tareas de los clientes, las ejecuta y devuelve los resultados. Recordamos que el servidor consta de una interfaz y de una clase. La interfaz define la vista que un cliente tiene del objeto remoto, mientras que la clase proporciona la implementación.

2.1 Definir la interfaz remota

El núcleo de compute engine es un protocolo que permite al cliente enviar tareas a compute engine, a compute engine ejecutar esas tareas, y devolver los resultados de las tareas al cliente. Este protocolo se expresa por las interfaces soportadas por nuestra aplicación. La comunicación se ilustra en la siguiente figura:



Cada interfaz contiene un sólo método. La interfaz remota de compute engine, que llamaremos `Compute`, hace que sea posible enviar tareas a compute engine. La interfaz del cliente, que llamaremos `Task`, define como compute engine ejecuta una tarea. El código de la interfaz `Compute` es el siguiente:

```
package compute;
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface Compute extends Remote {
    <T> T executeTask(Task<T> t) throws RemoteException;
}
```

Los métodos genéricos son métodos que introducen sus tipos de parámetros. El ámbito de este tipo es limitado al método. El método `executeTask` utiliza un tipo genérico `T` [3]. Esto es parecido a utilizar `Object`, pero tiene la ventaja de que el compilador se encargará de verificar los tipos, además no hay que hacer ningún casting. Veremos que también es posible declarar tipos genéricos.

Al especificar que extiende `java.rmi.Remote`, la interfaz `Compute` se identifica como una interfaz remota, lo cual quiere decir que se puede invocar desde otra maquina virtual de Java. Cualquier objeto que implemente esta interfaz puede ser un objeto remoto. Ya que `executeTask` es un método remoto, debe ser definido como capaz de lanzar la excepción `java.rmi.RemoteException`.

La segunda interfaz necesaria para el compute engine es la interfaz `Task`, que, como hemos visto, es el tipo de parámetro del método `executeTask` de la interfaz `Compute`. `Task` define la interfaz entre el compute engine y las tareas que tiene que ejecutar, proporcionando una forma de arrancar el trabajo. El código de la interfaz es el siguiente:

```
package compute;

public interface Task<T> {
    T execute();
}
```

La interfaz `Task` define un sólo método, `execute`, que no tiene parámetros ni lanza excepciones. Esta interfaz no extiende `Remote`, así que el método no tiene que lanzar `java.rmi.RemoteException` . Esta interfaz tiene un parámetro de tipo, `T`, que representa el resultado de una tarea. Esto es un tipo genérico [3], y es parecido a devolver un `Object`, solo que el compilador puede establecer en tiempo de compilación si el código es correcto, y no habrá que hacer ningún casting.

La interfaz `Compute`, a su vez, devuelve el resultado de la ejecución de la instancia de `Task` que recibió. Así que el método `executeTask` tiene su propio parámetro de tipo, `T`, que asocia su propio tipo a devolver con el tipo del resultado de la instancia `Task` que se le ha pasado.

Para que un objeto pueda ser pasado por valor entre máquinas virtuales de java tiene que se serializable. Esto quiere decir que la clase del objeto tiene que implementar la interfaz `java.io.Serializable`. Las clases que implementen `Task` deben entonces implementar también `Serializable`.

Un objeto `Compute` puede ejecutar distintos tipos de tareas, siempre que las tareas sean implementaciones de la interfaz `Task`. Las clases que implementan esta interfaz pueden contener cualquier dato y métodos necesarios para la ejecución de la tarea, y siempre tienen que contener una implementación del método `execute`.

La forma en que RMI hace que sea posible escribir esta sencilla aplicación es la siguiente. Implementaciones de objetos `Task` que antes eran desconocidas por el compute engine pueden ser descargadas por RMI en la maquina virtual de Java de compute engine cuando haga falta. Esto es así gracias al hecho de que RMI asume que los objetos `Task` estén escritos en Java. Esta capacidad permite a los clientes de compute engine definir nuevas tareas que deberán ser ejecutadas en la maquina del servidor, sin necesidad de instalar explícitamente el código en esa maquina.

El servidor, compute engine, implementa la interfaz `Compute`. Los clientes pueden enviar tareas utilizando el método `executeTask`. Estas tareas son ejecutadas utilizando la implementación del método `execute`, y los resultados son devueltos al cliente.

2.2 Implementación de la Interfaz Remota

Recordamos que una implementación de una interfaz remota debe hacer por lo menos lo siguiente:

- Declarar la interfaz remota que implementa
- Definir un constructor para cada objeto remoto
- Proporcionar una implementación de cada método remoto declarado en las interfaces remotas.

Recordamos también desde la práctica anterior que un servidor RMI tiene que crear los objetos remotos y exportarlos, para que sean accesibles a los clientes. Esto se puede hacer en la misma implementación del objeto remoto o en otra clase distinta.

En general, el servidor debe llevar a cabo estos pasos:

- Crear y instalar un gestor de seguridad (security manager) [1,5]
- Crear y exportar uno o más objetos remotos
- Registrar por lo menos un objeto remoto en el registro RMI.

Nuestro servidor será implementado por la clase `ComputeEngine`:

```
package engine;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import compute.Compute;
import compute.Task;

public class ComputeEngine implements Compute {

    public ComputeEngine() {
        super();
    }

    public <T> T executeTask(Task<T> t) {
        return t.execute();
    }

    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String name = "Compute";
            Compute engine = new ComputeEngine();
            Compute stub =
                (Compute) UnicastRemoteObject.exportObject(engine, 0);
            Registry registry = LocateRegistry.getRegistry();
            registry.rebind(name, stub);
            System.out.println("ComputeEngine bound");
        } catch (Exception e) {
            System.err.println("ComputeEngine exception:");
            e.printStackTrace();
        }
    }
}
```

A continuación se verán los varios aspectos de la implementación de compute engine.

Implementación de Método Remotos

La clase de un objeto remoto proporciona la implementación de cada método remoto especificado en la interfaz remota. En nuestro caso, solo hay un método remoto, `executeTask`, que es implementado de la siguiente forma:

```
public <T> T executeTask(Task<T> t) {
    return t.execute();
}
```

Este método implementa el protocolo entre el objeto remoto y los clientes. Cada cliente proporciona un objeto `Task` a `ComputeEngine`. Cada objeto `Task` tiene su propia implementación del método `execute`. Esté método determina el trabajo a desarrollar por el servidor. El servidor ejecuta cada tarea y devuelve el resultado de la ejecución del método `execute` directamente al cliente.

Paso de Objetos en RMI

Los parámetros de un método remoto o los valores devueltos pueden ser de casi cualquier tipo, incluyendo objetos locales, objetos remotos, y datos de tipo primitivo. Más específicamente, cualquier entidad de cualquier tipo puede ser pasada o devuelta a/por un método remoto, siempre que la entidad sea una instancia de un tipo primitivo, de un objeto remoto o de un objeto serializable, lo cual significa que implementa la interfaz `java.io.Serializable`.

Algunos tipos no satisfacen estos criterios, por lo que no se pueden pasar a un método remoto ni devolver por un método remoto. La mayoría de estos objetos, como identificadores de thread o de archivos, contienen información que solo tiene sentido en un espacio de direcciones específico. Los tipos definidos en los paquetes `java.lang` y `java.util`, implementan la interfaz `Serializable`.

Las normas que rigen cómo se pasan o se devuelven los argumentos son las siguientes:

- Los objetos remotos se pasan por referencia. Una referencia a un objeto remoto es un stub, que implementa el conjunto de interfaces remotas implementadas por el objeto remoto.
- Los objetos locales se pasan por copia, utilizando la serialización de objetos. Por defecto, todos los campos se copian a excepción de los campos marcados como `static` o `transient`.

Pasar un objeto remoto por referencia significa que cualquier cambio aportado por una invocación remota de un método será reflejado en el objeto remoto original. Cuando se pasa un objeto remoto, solo las interfaces remotas son visibles para el cliente. Cualquier método definido en la implementación de la clase, o definido en una interfaz no remota implementada por la clase, no será visible al cliente.

Por ejemplo, si pasamos una instancia de la clase `ComputeEngine`, el cliente solo tendría acceso al método `executeTask`. Como se ha especificado anteriormente, los objetos locales pasados a un método remoto o devueltos son pasados por valor. En la maquina virtual de Java que recibe el objeto se crea entonces una copia. Cualquier cambio aportado a este objeto será visible solo en la copia de quien recibe el objeto, mientras que el objeto original no se verá afectado por los cambios.

Implementar el método main del Servidor

El método más complejo de la implementación de `ComputeEngine` es el main. El main se utiliza para arrancar el `ComputeEngine` y por esto tendrá que ejecutar las inicializaciones oportunas para que el servidor pueda aceptar peticiones de los clientes. El main no es un método remoto y por lo tanto no se puede invocar desde una máquina virtual de Java diferente. Además como el main es estático, no se asocia a ningún objeto, sino a la clase `ComputeEngine`.

Crear e instalar un gestor de seguridad

La primera tarea del main es crear y instalar un gestor de seguridad (security manager), que proteja el acceso a los recursos del sistema por parte de código descargado en la máquina virtual de Java. Un security manager determina si un código descargado tiene acceso al sistema de archivos local o si puede ejecutar operaciones privilegiadas.

Si un programa RMI no usa un security manager, RMI no descargará las clases (a menos que ya no se encuentren en la misma máquina virtual) de los objetos recibidos como argumentos o enviados como valores de retorno. Esta restricción asegura que las operaciones llevadas a cabo por el código descargado sean sujetas a una política de seguridad.

El código que crea y instala el security manager es el siguiente:

```
if (System.getSecurityManager() == null) {
    System.setSecurityManager(new SecurityManager());
}
```

Además, tendremos que especificar un archivo de política de seguridad, para evitar el lanzamiento de excepciones. Este archivo permitirá a la aplicación de enlazarse a un puerto local (si es un servidor), y de conectarse con hosts remotos (si es un cliente). A la hora de ejecutar un cliente/servidor, tendremos que añadir una opción para que la aplicación utilice un archivo de política de seguridad específico:

```
java -Djava.security.policy=rmi.policy  yourserver
```

Tendremos además que crear el archivo "`rmi.policy`". En esta práctica utilizaremos archivos similares al siguiente:

```
grant codeBase "file:/CarpetaAplicacion/" {
    permission java.security.AllPermission;
};
```

Esta política proporciona todos los permisos a las clases en la carpeta especificada, pero no se da ningún permiso a un código descargado desde otras rutas. Para más información consulte [1,6,7].

2. El Cliente

Compute engine es un programa relativamente simple: ejecuta tareas que se le envían. Los clientes de compute engine son más complejos. Un cliente tiene que llamar a compute engine, pero también tiene que definir la tarea que será ejecutada por compute engine.

En este ejemplo, el cliente está formado por dos clases separadas. La primera clase, llamada ComputePi, busca y invoca un objeto Compute. La segunda clase, llamada `Pi`, implementa la interfaz `Task` y define la tarea que deberá ser ejecutada en el servidor. Lo que la clase `Pi` pretende hacer es calcular el valor de π hasta una cierta precisión.

La interfaz no remota Task es la siguiente:

```
package compute;

public interface Task<T> {
    T execute();
}
```

El código que llama a los métodos del objeto `Compute` tienen que obtener primero una referencia al objeto, crear un objeto `Task` y luego solicitar que la tarea sea ejecutada. Mostraremos la definición de la clase `Pi` más adelante. Un objeto `Pi` se construye con un sólo parámetro, que define la precisión deseada. El resultado de la ejecución de esta tarea es un `java.math.BigDecimal` que representa a π calculado con la precisión especificada.

El código de `ComputePi` es el siguiente:

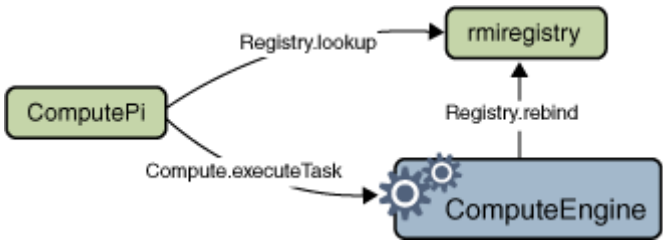
```
package client;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.math.BigDecimal;
import compute.Compute;

public class ComputePi {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String name = "Compute";
            Registry registry = LocateRegistry.getRegistry(args[0]);
            Compute comp = (Compute) registry.lookup(name);
            Pi task = new Pi(Integer.parseInt(args[1]));
```

```
            BigDecimal pi = comp.executeTask(task);
            System.out.println(pi);
        } catch (Exception e) {
            System.err.println("ComputePi exception:");
            e.printStackTrace();
        }
    }
}
```

Al igual que el servidor, el cliente empieza instalando un security manager. Este paso es necesario ya que, en general, el proceso de recibir un stub de un objeto remoto podría requerir descargar definiciones de clases desde el servidor. Para que RMI descargue clases, es necesario utilizar un security manager.

Después de instalar el security manager, el cliente busca al objeto remoto `Compute` utilizando el mismo nombre utilizado en el servidor. `args[0]`, es el primer argumento que se pasa por linea de comando y representa el nombre del host remoto donde se está ejecutando el servidor.

Después de eso el cliente crea un nuevo objeto `Pi`, pasando al constructor de `Pi` el valor del segundo argumento pasado por línea de comando, `args[1]`, después de haberlo convertido a entero. Este argumento indica el número de decimales que se utilizará en el cálculo de π. Finalmente, el cliente invoca el método `executeTask` del objeto remoto `Compute`. El objeto pasado a la invocación de `executeTask` devuelve un objeto de tipo `BigDecimal`, que el programa almacena en la variable `pi`. La siguiente figura muestra el flujo de mensajes entre el cliente, el registro de rmi y el servidor.



La clase `Pi` implementa la interfaz `Task` y calcula el valor de π con el número de decimales especificado. Para este ejemplo, el algoritmo en particular no es importante. Lo que importa es que el algoritmo es costoso desde el punto de vista computacional, así que es deseable poder ejecutar el algoritmo en una maquina potente.

Este es el código de `Pi`:

```
package client;

import compute.Task;
import java.io.Serializable;
import java.math.BigDecimal;

public class Pi implements Task<BigDecimal>, Serializable {

    private static final long serialVersionUID = 227L;

    /** constants used in pi computation */
    private static final BigDecimal FOUR =
        BigDecimal.valueOf(4);

    /** rounding mode to use during pi computation */
    private static final int roundingMode =
        BigDecimal.ROUND_HALF_EVEN;

    /** digits of precision after the decimal point */
    private final int digits;
```

```
/**
 * Construct a task to calculate pi to the specified
 * precision.
 */
public Pi(int digits) {
    this.digits = digits;
}

/**
 * Calculate pi.
 */
public BigDecimal execute() {
    return computePi(digits);
}

/**
 * Compute the value of pi to the specified number of
 * digits after the decimal point.  The value is
 * computed using Machin's formula:
 *
 *          pi/4 = 4*arctan(1/5) - arctan(1/239)
 *
 * and a power series expansion of arctan(x) to
 * sufficient precision.
 */
public static BigDecimal computePi(int digits) {
    int scale = digits + 5;
    BigDecimal arctan1_5 = arctan(5, scale);
    BigDecimal arctan1_239 = arctan(239, scale);
    BigDecimal pi = arctan1_5.multiply(FOUR).subtract(
                                arctan1_239).multiply(FOUR);
    return pi.setScale(digits,
                       BigDecimal.ROUND_HALF_UP);
}
/**
 * Compute the value, in radians, of the arctangent of
 * the inverse of the supplied integer to the specified
 * number of digits after the decimal point.  The value
 * is computed using the power series expansion for the
 * arc tangent:
 *
 * arctan(x) = x - (x^3)/3 + (x^5)/5 - (x^7)/7 +
 *     (x^9)/9 ...
 */
public static BigDecimal arctan(int inverseX,
                                int scale)
{
    BigDecimal result, numer, term;
    BigDecimal invX = BigDecimal.valueOf(inverseX);
    BigDecimal invX2 =
        BigDecimal.valueOf(inverseX * inverseX);

    numer = BigDecimal.ONE.divide(invX,
                                  scale, roundingMode);

    result = numer;
    int i = 1;
    do {
        numer =
```

```
                numer.divide(invX2, scale, roundingMode);
            int denom = 2 * i + 1;
            term =
                numer.divide(BigDecimal.valueOf(denom),
                             scale, roundingMode);
            if ((i % 2) != 0) {
                result = result.subtract(term);
            } else {
                result = result.add(term);
            }
            i++;
        } while (term.compareTo(BigDecimal.ZERO) != 0);
        return result;
    }
}
```

Como todas las clases serializables, `Pi` tiene que declarar un atributo `serialVersionUID` de tipo `private static final` para garantizar la compatibilidad de serialización entre versiones de la clase.

La cosa más interesante de este ejemplo es que la implementación de `Compute` no necesita la definición de la clase `Pi` hasta que un objeto `Pi` sea pasado como argumento al método `executeTask`. En ese momento, el código de la clase será cargado por parte de RMI en la maquina virtual de Java del objeto `Compute`, el método `executeTask` es invocado, y el código de la tarea es ejecutado. El resultado, que en el caso de `Pi` es un objeto `BigDecimal`, se devuelve al cliente, que lo utiliza para imprimir a pantalla el resultado del cálculo.
El hecho de que el objeto `Task` proporcionado calcule el valor de `Pi` es irrelevante para el objeto `ComputeEngine`. Se podría pasar, por ejemplo, otra tarea que, por ejemplo, genera un número primo utilizando un algoritmo probabilístico. Esa tarea también sería costosa desde el punto de vista computacional, y sería entonces una buena candidata para ejecutarse en un servidor, pero necesitaría un código muy distinto. Este código también se descarga cuando el objeto `Task` se pasa al objeto `Compute`. Como para el algoritmo para calcular π se descarga cuando es necesario, el código que genera el número primo también se descargaría solo cuando haga falta. El objeto `Compute` sólo sabe que cada objeto que recibe implementa el método `execute`, pero no sabe nada, y no hace falta que lo sepa, sobre qué hace la implementación específica.

3. Compilar la Aplicación

En aplicaciones reales, en las cuales se despliega un servicio como el compute engine, un desarrollador generalmente genera un archivo Java (JAR) que contenga las interfaces `Compute` y `Task`. La primera interfaz será implementada por el servidor, y la segunda será utilizada por los clientes. A continuación un desarrollador del servidor podría escribir una implementación de la interfaz `Compute` y desplegar el servicio en una máquina accesible a los clientes. Las aplicaciones clientes podrán utilizar las interfaces `Compute` y `Task`, contenidas en el archivo JAR, y desarrollar de forma independiente tareas y clientes que utilizan el servicio `Compute`.
Veremos que la clase cliente `Pi` será descargada por el servidor en tiempo de ejecución, al igual que las interfaces `Compute` y `Task` serán descargadas desde el servidor al registro RMI en tiempo de ejecución.
El código de este ejemplo es separado en tres paquetes:

- `compute`, que contiene las interfaces `Compute` y `Task`
- `engine`, que contiene la implementación de la clase `ComputeEngine`
- `client`, que contiene el código del cliente `ComputePi` y la implementación de la tarea `Pi`

Primero crearemos el archivo JAR que contiene las interfaces, para proporcionarlo luego tanto al servidor como al cliente.

Crear un Archivo JAR de Interfaces

Para poder crear el archivo JAR, primero tendremos que compilar el código fuente de las interfaces en el paquete `compute`. Para esto utilizaremos los siguientes comandos:

Windows

```
javac compute\Compute.java compute\Task.java
jar cvf compute.jar compute\*.class
```

Linux

```
javac compute/Compute.java compute/Task.java
jar cvf compute.jar compute/*.class
```

Ahora podemos distribuir el archivo `compute.jar` para su uso por parte del servidor y del cliente.

Después de haber compilado las clases del servidor y del cliente con `javac`, si algunas de esas clases necesitan ser descargadas dinámicamente por otras máquinas virtuales de Java, tenemos que asegurarnos que sus archivos de clase se encuentren en un sitio accesible desde la red. En este ejemplo, en Linux, utilizaremos la ruta `/home/user/public_html/classes` ya que muchos servidores web permiten acceder al directorio `public_html` a través de un URL HTTP construido como `http://host/~user/`. Si el servidor web no soporta esta convención, podemos utilizar un URL de archivo. Este URL tendrá la forma de `file:/home/user/public_html/classes/` en Linux y de `file:/c:/home/user/public_html/classes/` en Windows. También podemos seleccionar otro tipo más apropiado de URL, según el caso.

El hecho de que los archivos de clases (los archivos con extensión class) sean accesibles en la red, permite la descarga de código por parte RMI cuando sea necesario. RMI no utiliza un protocolo propio para descargar código, sino que utiliza protocolos URL soportados por las plataformas Java (por ejemplo HTTP).

Crear las Clases del Servidor

El paquete `engine` contiene sólo una implementación de servidor, `ComputeEngine`, que implementa la interfaz remota `Compute`. Supongamos que un usuario, `ann`, desarrollador de la clase `ComputeEngine`, haya colocado el archivo `ComputeEngine.java` en el directorio `c:\home\ann\src\engine` en Windows o en el directorio `/home/ann/src/engine` en Linux. Los archivos de clase serán desplegados en un subdirectorio de `public_html`, por ejemplo `c:\home\ann\public_html\classes` en Windows o `/home/ann/public_html/classes` en Linux. Esta ubicación es accesible a través de algún servidor web utilizando un URL del tipo `http://host:port/~ann/classes/`.

La clase `ComputeEngine` depende de las interfaces `Compute` y `Task`, que son contenidas en el archivo JAR `compute.jar`. Como consecuencia, necesitamos que el archivo `compute.jar` esté presente en la ruta de clase (class path) cuando se cree la clase servidor. Supongamos que el archivo `compute.jar` esté ubicado en el directorio `c:\home\ann\public_html\classes` en Windows o en `/home/ann/public_html/classes` en Linux. Entonces podemos utilizar los siguientes comandos para construir la clase servidor:

Windows:

```
cd c:\home\ann\src
javac -cp c:\home\ann\public_html\classes\compute.jar engine\ComputeEngine.java
```

Linux:

```
cd /home/ann/src
javac -cp /home/ann/public_html/classes/compute.jar engine/ComputeEngine.java
```

El stub de `ComputeEngine` implementa la interfaz `Compute`, que hace referencia a la interfaz `Task`. Así que las definiciones de clase para esas dos interfaces tienen que ser accesibles en la red, de tal forma que el stub las pueda recibir en otra máquina virtual de Java. La máquina virtual de Java del cliente ya tiene esas dos interfaces en su ruta de clase, así que no necesita descargar sus definiciones. El archivo `compute.jar` en el directorio `public_html` puede ser utilizado a tal fin.
Ahora el compute engine está listo para su despliegue.

Crear las Clases Cliente

El paquete `client` contiene dos clases, el programa principal `ComputePi` y `Pi`, la implementación de la interfaz `Task`. Supongamos que el usuario `jones`, desarrollador de las clases clientes, haya colocado `ComputePi.java` y `Pi.java` en el directorio `c:\home\jones\src\client` en Windows o en `/home/jones/src/client` en Linux.

Los archivos de clase serán desplegados en un subdirectorio de su directorio `public_html`, `c:\home\jones\public_html\classes` en Windows o `/home/jones/public_html/classes` en Linux. Esta ubicación es accesible a través de algunos servidores web con el URL `http://host:port/~jones/classes/`. Las clases clientes dependen de las interfaces `Compute` y `Task`, contenidas en el archivo JAR `compute.jar`. Es necesario tener el archivo en la ruta de clase en el momento de compilar las clases cliente. Supongamos que el archivo `compute.jar` está en el directorio `c:\home\jones\public_html\classes` en Windows o en `/home/jones/public_html/classes` en Linux. Podemos entonces utilizar los siguientes comandos para crear las clases cliente:

Windows:

```
cd c:\home\jones\src
javac -cp c:\home\jones\public_html\classes\compute.jar client\ComputePi.java
client\Pi.java
mkdir c:\home\jones\public_html\classes\client
cp client\Pi.class c:\home\jones\public_html\classes\client
```

Linux:

```
cd /home/jones/src
javac -cp /home/jones/public_html/classes/compute.jar client/ComputePi.java
client/Pi.java
mkdir /home/jones/public_html/classes/client
cp client/Pi.class /home/jones/public_html/classes/client
```

Solo la clase `Pi` tiene que ser colocada en el directorio `public_html\classes\client` ya que sólo la clase `Pi` tiene que estar disponible para ser descargada por la máquina virtual de Java de compute engine.
Ahora podemos ejecutar el servidor y el cliente.

4. Ejecutar la aplicación

Acerca de la Seguridad

El servidor y el cliente se ejecutan con un security manager instalado. Cuando se ejecutan, estos programas necesitan de un archivo que especifique las políticas de seguridad que se van a adoptar, de tal forma que el código tenga los permisos de seguridad para que se ejecute correctamente. En este ejemplo, podemos utilizar estos simples archivos.

Para el servidor:

```
grant codeBase "file:/home/ann/src/" {
    permission java.security.AllPermission;
};
```

Y para el cliente:

```
grant codeBase "file:/home/jones/src/" {
    permission java.security.AllPermission;
};
```

Los dos archivos proporcionan todos los permisos a las clases en la ruta local del programa, ya que las aplicaciones locales son seguras, pero no se asigna ningún permiso a código descargado desde ubicaciones distintas. De esta forma, el servidor no permite a la tarea que ejecuta (cuyo código es desconocido y podría ser peligroso) ejecutar ninguna operación que necesite permisos de seguridad. En nuestro ejemplo, la tarea `Pi` no necesita ningún permiso para ejecutarse.

Guardaremos las políticas de seguridad del servidor en un archivo llamado `server.policy` y las del cliente en un archivo llamado `client.policy`.

Arrancar el Servidor

Recuerde que antes de arrancar el servidor, hay que arrancar el registro RMI. Antes de esto, tenemos que asegurarnos de que la shell desde la cual se ejecuta `rmiregistry` no tenga definida ninguna variable `CLASSPATH` o bien que la variable `CLASSPATH` no incluya la ruta de ninguna clase que vaya a ser descargada por los clientes. En este ejemplo suponemos que el registro sea ejecutado en el host `mycomputer` (que podría ser `localhost`).

Una vez arrancado el registro, podemos arrancar el servidor. Tenemos que asegurar que tanto el archivo `compute.jar` como la implementación del objeto remoto estén en nuestra ruta de clase. Cuando arrancamos al servidor, tenemos que especificar, utilizando la propiedad `java.rmi.server.codebase`, dónde se puede acceder a las clases del servidor. En este ejemplo, las clases que se necesitará descargar son las interfaces `Compute` y `Task`, contenidas en el archivo `compute.jar`, localizado en el directorio `public_html\classes` del usuario `ann`. El servidor se arranca en el mismo host que el registro RMI.

Windows:

```
java -cp c:\home\ann\src;c:\home\ann\public_html\classes\compute.jar
    -Djava.rmi.server.codebase=file:/c:/home/ann/public_html/classes/compute.jar
    -Djava.rmi.server.hostname=mycomputer.example.com
    -Djava.security.policy=server.policy
        engine.ComputeEngine
```

Linux:

```
java -cp /home/ann/src:/home/ann/public_html/classes/compute.jar
     -Djava.rmi.server.codebase=http://mycomputer/~ann/classes/compute.jar
     -Djava.rmi.server.hostname=mycomputer.example.com
     -Djava.security.policy=server.policy
        engine.ComputeEngine
```

Estos comandos establecen las siguientes propiedades de sistema:

- La propiedad `java.rmi.server.codebase` especifica la ubicación, un URL, desde la cual las definiciones de las clases generadas por el lado servidor pueden ser descargadas. Si esta propiedad especifica un directorio (y no un archivo JAR), entonces hay que terminar el URL con una barra /.
- La propiedad `java.rmi.server.hostname` especifica el nombre o dirección del host que se colocará en los stubs de los objetos remotos exportados en este máquina virtual de Java. Este valor es el nombre o dirección del host utilizado por los clientes cuando ejecutan invocaciones remotas de métodos. Por defecto, la implementación de RMI utiliza la dirección IP del servidor tal como indicado por la API `java.net.InetAddress.getLocalHost`. Pero, no siempre esta dirección es apropiada para todos los clientes, y en estos casos un nombre de host sería más efectivo. Para asegurar que RMI utilice un nombre de host (o dirección IP) hay que utilizar la propiedad `java.rmi.server.hostname`.
- La propiedad `java.security.policy` es utilizada para especificar el fichero que contiene las políticas de seguridad.

Arrancar el Cliente

Una vez que el registro y el servidor se estén ejecutando, podemos arrancar el cliente, especificando lo siguiente:

- La ubicación desde la cual el cliente hace disponibles sus clases (la clase `Pi`), utilizando la propiedad `java.rmi.server.codebase`.
- La política de seguridad, utilizando la propiedad `java.security.policy`.
- El nombre de host del servidor (de tal forma que el cliente sepa donde localizar el objeto remoto) y el número de decimales que se utilizarán en el calculo de π . Esta información se pasa como argumentos por linea de comando.

Podemos arrancar el cliente en otro host (por ejemplo en un host llamado `mysecondcomputer`) o en `localhost` de la siguiente forma:

Microsoft Windows:

```
java -cp c:\home\jones\src;c:\home\jones\public_html\classes\compute.jar
     -Djava.rmi.server.codebase=file:/c:/home/jones/public_html/classes/
     -Djava.security.policy=client.policy
        client.ComputePi mycomputer.example.com 45
```

Linux:

```
java -cp /home/jones/src:/home/jones/public_html/classes/compute.jar
     -Djava.rmi.server.codebase=http://mysecondcomputer/~jones/classes/
     -Djava.security.policy=client.policy
        client.ComputePi mycomputer.example.com 45
```

Hay que subrayar que la ruta de clase es establecida en la linea de comando, de tal forma que el intérprete pueda encontrar las clases clientes y el archivo JAR. También notamos que el valor de la propiedad `java.rmi.server.codebase` especifica un directorio y termina en una barra.

Después de arrancar al cliente, se produce la siguiente salida:

```
3.14159265358979323846264338327950288419716939 9
```

Cuando el servidor vincula su referencia al objeto remoto al registro RMI, el registro descarga las interfaces `Compute` y `Task` ya que el stub depende de ellas. Estas clases serán descargadas o desde el servidor web de `ComputeEngine` o desde el sistema de archivos, dependiendo del tipo de URL utilizado al arrancar el servidor.

El cliente cargará las interfaces desde su ruta de clase, ya que tiene acceso a las interfaces `Compute` y `Task` a través su ruta de clase.

Finalmente la clase `Pi` será cargada en la máquina virtual del servidor cuando se pasa el objeto `Pi` al método remoto. La clase `Pi` es cargada por el servidor bien desde el servidor web del cliente o bien desde el sistema de archivos, dependiendo el URL utilizado al arrancar el cliente.

Bibliografia

1. Security Manager http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/SecurityManager.html
2. Java RMI API http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/index.html
3. Java generics http://docs.oracle.com/javase/tutorial/java/generics/index.html
4. http://docs.oracle.com/javase/tutorial/java/generics/bounded.html
5. http://www.eg.bucknell.edu/~cs379/DistributedSystems/rmi_tut.html#secure
6. http://www.cs.swan.ac.uk/~csneal/InternetComputing/RM3.html
7. http://docs.oracle.com/javase/1.5.0/docs/guide/security/PolicyFiles.html

Consideraciones de Desarrollo

Se pide desarrollar la aplicación descrita en el guión. Además se pide modificar el cliente para que pueda ejecutar más tipos de tareas. Ejemplos de posibles tares podrían ser:

- Multiplicación de dos matrices 1500x1500
- Comprobar si un número es primo utilizando un algoritmo probabilístico. (http://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test#Algorithm_and_running_time)
- Generar un número primo utilizando un algoritmo probabilístico (http://www.google.es/url?sa=t&rct=j&q=probabilistic%20algorithm%20for%20generating%20prime%20number&source=web&cd=7&ved=0CEwQFjAG&url=http%3A%2F%2Fece.gmu.edu%2Fcourses%2FECE543%2Fproject%2Freports_1999%2Fdong_report.pdf&ei=GbOjUP75BYX54QS-d7oDoAg&usg=AFQjCNHk3DsQfnYNpl-W7lQUdf5dKYJ_tg&cad=rja) Este tipo de algoritmo se utiliza en la generación de llaves para RSA
- Funciones similares a las propuestas en los problemas y ampliaciones de problemas de las prácticas 1 y 2

NORMAS DE ENTREGA

LEA DETENIDAMENTE LAS NORMAS DE LA PRUEBA

Cualquier incumplimiento de las normas significará una respuesta nula en el examen y por tanto una valoración de 0 puntos.

1. La resolución de los ejercicios propuestos se entregará en un único archivo formato ZIP que contendrá EXCLUSIVAMENTE los archivos que compongan la solución a los problemas planteados.

2. El nombre del fichero tendrá un formato específico dictado por el nombre de cada alumno. Por ejemplo, para un alumno llamado "José María Núñez Pérez" el fichero se nombrará como NunyezPerezJM.zip. Obsérvese que las tildes son ignoradas y las eñes sustituidas.

3. El fichero se subirá utilizando la correspondiente tarea en WebCT

IMPORTANTE: Cualquier envío que no respete el formato de compresión o el nombre adecuado será ignorado y, por tanto, valorado con cero puntos.