INTRODUCCIÓN

JAVA

En los últimos años ha aumentado mucho la presencia de arquitecturas de hardware incompatibles, que soportan distintos sistemas operativos y trabajan con plataformas que operan bajo una o más interfaces gráficas de usuario. Esto, sumando al crecimiento de Internet, y especialmente de la World Wide Web y del comercio electrónico, ha supuesto una creciente complicación para el usuario, que es fuente de numerosos problemas.

Frente a ello, Sun Microsystems ha creado el lenguaje de programación JAVA, que constituye una importante aportación para resolver estos problemas. Presenta un nuevo punto de vista en la evolución de los lenguajes de programación: la creación de un lenguaje sencillo y reducido pero lo suficientemente amplio para afrontar el desarrollo de una gran variedad de software.

JAVA es fácil de usar, y sus principales características pueden resumirse en: ser un lenguaje orientado a objetos; ser un sistema interpretado; sus aplicaciones, portables a distintas plataformas, son sólidas y se adaptan a cambios de entorno porque JAVA puede transferir módulos de código desde cualquier punto de la red. Otro aspecto a destacar es la seguridad de sus aplicaciones, ya que su sistema run−time lleva incorporada una protección contra virus y otras alteraciones.

La primera gran aplicación escrita bajo el entorno JAVA fue HotJAVA, el primer navegador que permitió la transferencia dinámica y la ejecución de fragmentos de código JAVA de modo seguro desde cualquier punto de Internet. Esto es, pudo verse la parte viva de Internet.

Orígenes del proyecto de lenguaje JAVA

JAVA se diseñó para desarrollar aplicaciones en el contexto de los entornos distribuidos con redes y plataformas heterogéneas. Surgió como parte de un proyecto de investigación para conseguir un software avanzado, destinado a una amplia variedad de equipos, dispositivos de red y sistemas integrados. El objetivo era desarrollar un pequeño entorno operativo en tiempo real, fiable, portable y distribuido.

Al comenzar el proyecto se eligió el lenguaje C++, pero pronto las dificultades encontradas forzaron a concluir que era preferible crear un entorno de programación completamente nuevo.

Las decisiones de diseño y arquitectura arrastraron a una variedad de lenguajes como Eiffel, SmallTalk, Objective C,... El resultado fue un entorno de programación que ha demostrado ser excelente para el desarrollo seguro y distribuido de aplicaciones para usuarios finales basadas en red, en entornos tan variados como los dispositivos incorporados en redes, World Wide Web y sistemas locales.

Objetivos de diseño de JAVA

La naturaleza de los entornos informáticos donde debía desplegarse el software determinó los requisitos de diseño de JAVA.

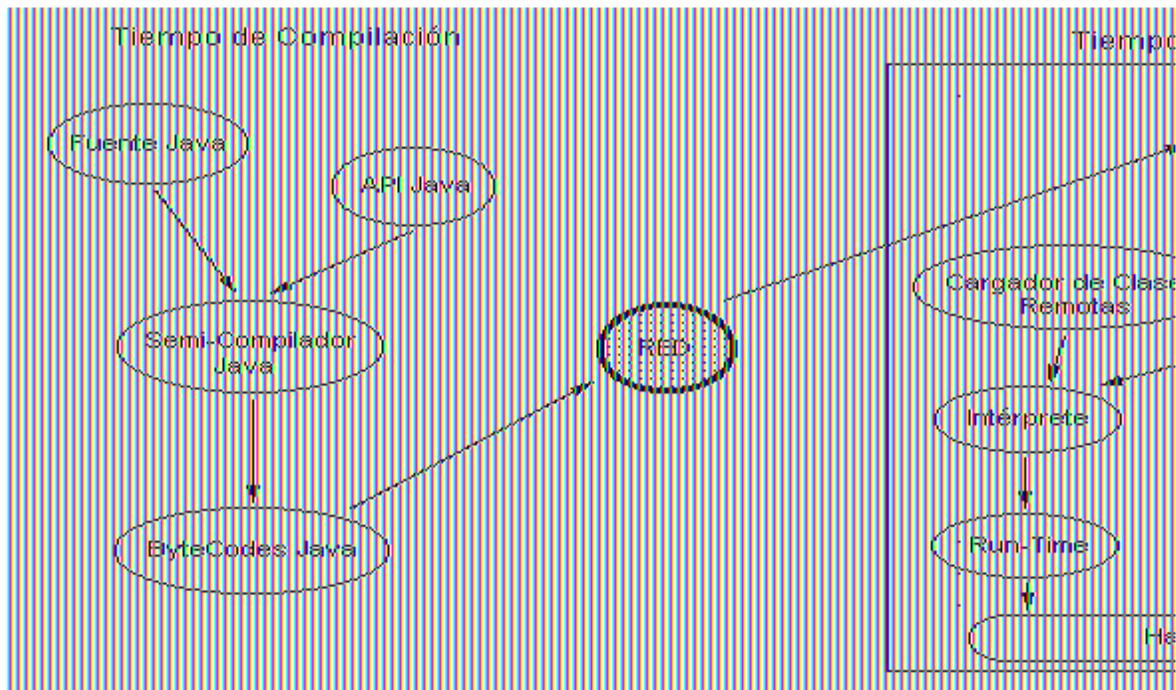
El crecimiento espectacular de Internet y, en especial, de la World Wide Web han implicado una forma completamente nueva de contemplar el desarrollo y distribución de software. Para existir en el mundo de la distribución y comercio electrónicos, JAVA debía permitir el desarrollo de aplicaciones extremadamente robustas, seguras, de alto rendimiento y capaces de funcionar sobre plataformas muy distintas en redes distribuidas y heterogéneas. La operación en múltiples plataformas en redes heterogéneas invalida los

esquemas tradicionales de distribución binaria, nuevas versiones, actualización, corrección de errores, etc. Para sobrevivir en este mundo, JAVA tenía que ser una arquitectura neutra, portable y dinámicamente adaptable.

El sistema que surgió para enfrentarse a estas necesidades es sencillo, de forma que la mayoría de los desarrolladores puedan utilizarlo sin dificultades; resulta familiar, lo cual simplifica el aprendizaje; es orientado a objetos, para aprovechar las ventajas de las modernas tecnologías de desarrollo de software y adaptarse a las aplicaciones cliente−servidor distribuidas; es multitarea, para poder crear aplicaciones de alto rendimiento que puedan realizar muchas actividades simultáneas, como multimedia; y, por último, es interpretado, para obtener una máxima portabilidad y capacidad dinámica.

Características más destacables de JAVA:

- **SIMPLE**: Es muy parecido al C++, pero sin las características menos usadas y más confusas de este lenguaje (aritmética de punteros, registros, necesidad de liberar memoria).
- **ORIENTADO A OBJETOS**: pueden emplearse todas las técnicas de diseño y porgramación OO: jerarquía de clases, herencia, encapsulación y polimorfismo...
- **DISTRIBUIDO**: JAVA soporta ejecución remota y distribuida. Mediante RMI, *Remote Method Invocation* un cliente puede ejecutar acciones en objetos que se encuentren en un servidor, con independencia de la situación del servidor. Aparte de aplicaciones normales, pueden desarrollarse otros programas llamados applets que se ubican en servidoes de páginas Web realizadas con HTML y que se cargan y ejecutan al tiempo que las páginas son requeridas por el usuario con un visualizador con soporte Java.
- **DE ARQUITECTURA NEUTRAL**: un mismo programa JAVA puede ejecutarse en diferentes sistemas y plataformas sin tener que realizar ningún cambio sobre el código: adaptar tipos de datos, resolver ambigüedades en la evaluación de expresiones, recompilar los programas para cada nueva plataforma. Esto es así porque: una vez escrito el código fuente de una aplicación JAVA, se compila a un código intermedio independiente de la arquitectura o JAVA `byte−code', código este que es interpretado por una máquina virtual implementada por software, que emula un croprocesador cuyo código máquina fuera este `byte−code'.
- **PORTABLE**: la neutralidad respecto a la arquitectura es sólo una parte de un sistema realmente portable. Java, además, define estrictamente las especificaciones del lenguaje: tamaño de los tipos de los datos básicos, comportamiento estricto de todos los operadores aritméticos. Los programas se ejecutan sobre la máquina virtual Java, que especifica todas las instrucciones permitidas y su significado. Para poder ejecutar un `byte−code' sobre una nueva plataforma sólo es necesario portar la nueva máquina virtual al nuevo sistema.



- **INTERPRETADO**: los `byte–codes' compilados son ejecutados por una parte de la máquina virtual, el intérprete. Así, las aplicaciones compiladas son ejecutables en cualquier plataforma o sistema hardware que disponga de una emulación software de esta maquina virtual JAVA, sin necesidad de recompilar el código. Al ser interpretado, no existe fase de enlace (*link*) que se sustituye por el proceso de cargar por el `Class Loader' las nuevas clases conforme van siendo necesitadas a través de la red. Aunque, en casos de tiempos de ejecución críticos, existe la posiblidad de compilar, usando compiladores JIT (*Just In Time*), los `bytes–codes' al código nativo de la plataforma sobre la que se va a ejecutar el programa.
- **ROBUSTO Y SEGURO**: Impide que el programador pueda cometer errores que otros lenguajes le permiten y por el contrario le facilita las cosas que hace bien. Se realiza un chequeo en tiempo de compilación donde la comprobación de tipos es muy estricta, pues no se permiten declaraciones implícitas. El modelo de memoria en Java elimina la posibilidad de sobreescribir zonas de memoria por un puntero `perdido', pues no existen punteros. En lugar de aritmética de punteros Java trata los vectores como objetos, esto permite comprobaciones para evitar errores por índices que salen de rango. Por otro lado, su seguridad se basa en que todos los `bytes–codes' pasan un proceso de verificación después de su carga y antes de su ejecución. Se comprueba, por ejemplo, que no hay desbordamiento de operandos en la pila, que los tipos de todos los parámetros son correctos, que no se realizan conversiones ilegales, etc. A la hora de ejecutar las clases se reservan espacios de ejecución separados, uno para las clases locales y otro para las clases importadas.
- **MULTI–THREAD**: En un mismo programa puede haber más de un hilo de ejecución, con lo cual un programa puede hacer muchas cosas a la vez: audio, video, interacciones con el usuario... todo ello sin cortes aparentes. Estos hilos JAVA suelen corresponder con hilos reales del sistema operativo, si el sistema de la máquina soporta estas características. Así se puede aprovechar la potencia de las máquinas multiproceso.

RMI (*Remote Method Invocation*)

RMI, *Invocación de Métodos Remotos* es una funcionalidad que JAVA proporciona para la construcción de aplicaciones distribuidas de la forma más transparente posible para el programador. Es otra implementación del concepto de RPC, *Remote Procedure Call* o Llamada a Procedimientos Remotos, concepto desarrollado desde hace varios años en varias plataformas.

Esencialmente permite que objetos JAVA distribuidos en distintas máquinas puedan comunicarse entre sí, enviarse mensajes, ejecutar código remoto, etc. sin que el programador tenga que preocuparse del proceso de construir paquetes, gestionar problemas de conexión, controlar errores, etc. RMI intenta simular que el programa está trabajando siempre con objetos en la máquina local.

IIOP (*Internet Inter−ORB Protocol*)

IIOP es un protocolo de comunicación para componentes CORBA. Define el modo en que se comunican sobre un medio objetos CORBA clientes y servidores.

La plataforma Java no es una isla independiente que se pueda asociar a cualquier entorno. Su interoperabilidad con el sistema en el que funciona es posible gracias a distintos elementos o mecanismos como JNI (*Java Native Interface*), o Java IDL. Este mecanismo de interoperabilidad es más interesante que los demás pues la comunicación entre Java y el elemento externo se realiza a nivel del objeto. Este objeto puede estar desarrollado igualmente en Java o bien en otros lenguajes como C++, Cobol o Delphi. La vía de transporte es el protocolo IIOP; y para poder emplear un objeto remoto CORBA es preciso disponer de la definición en IDL (*Interface Definition Language*) o Lenguaje de Definición de Interfaces. Se genera así un esqueleto con el código necesario para desde Java acceder y realizar llamadas al objeto remoto. Java IDL es un ORB (*Object Request Broker o Intermediario de Solicitud de Objetos*) Java compatible con las especificaciones CORBA. Se incorpora a los componentes de forma que, ya en ejecución, les permite el uso del protocolo IIOP para comunicación con otros ORB, y facilita el trabajo con el objeto CORBA remoto. Java IDL es un ORB completo por lo que es posible crear tanto clientes como servidores CORBA.

CORBA (*Common Object Request Broker Architecture*) o Arquitectura de Intermediación de Solicitud de Objetos Comunes es una arquitectura estándar de objetos distribuidos diseñada y desarrollada por la asociación OMG (*Object Management Group*) o Grupo de Administración de Objetos. Los rasgos más significativos de esta tecnología son la interoperabilidad entre objetos dentro de entornos corporativos y el desarrollo compartido.

Los interfaces a objetos remotos para definir, implementar y acceder a a objetos CORBA se describen en una plataforma neutral y en un lenguaje de definición de interfaces (IDL).

Entre otras posibilidades implícitas la versión 3.0 de CORBA cargará de forma dinámica tanto los componentes como los diferentes protocolos de mensajería, transacción y transporte de los citados objetos.

Entre otras características adicionales que podrá encontrarse el usuario en esta nueva versión de CORBA pueden destacarse la utilización de herramientas y entornos visuales de aplicaciones para el diseño de los componentes CORBA, el soporte de objetos CORBA para transporte asíncrono de grupos de mensajes en tiempo real y rasgos vinculados directamente al soporte de sistemas heredados.

RMI – IIOP

Con RMI pueden escribirse programas distribuidos en JAVA pues es muy fácil de usar y no es necesario aprender un lenguaje de definición de interfaces, Interfaces Definition Language (IDL). Además se optimizan recursos consiguiendo lo que se denomina escribir una vez, ejecutar en cualquier parte.

Clientes, servidores e interfaces remotos se escriben integramente en JAVA. Para comunicar objetos JAVA remotos, RMI usa *JAVA Remote Method Protocol* (JRMP), protocolo de comunicación no−estándar, pero así no puede establecer comunicación con objetos CORBA.

En otros tiempos para soluciones distribuidas debía elegirse entre RMI y CORBA−IIOP. Actualmente, pueden desarrollarse JAVA Remote Method Invocation (RMI) que accedan a objetos remotos usando IIOP, *Internet*

Inter−ORB Protocol. Cumpliendo un pequeño conjunto de restricciones, los objetos RMI pueden usar el protocolo IIOP y así comunicarse con objetos CORBA. Y esta solución es lo que se conoce como RMI−IIOP. Proporciona facilidades de uso unidas a la interoperabilidad de distintos lenguajes con objetos CORBA. Además, pueden desarrollarse nuevos programas RMI−IIOP o convertir programas ya existentes desarrollados inicialmente como RMI.

RMI – IIOP

ARQUITECTURA DE APLICACIONES DISTRIBUIDAS

ENFOQUES AL DISEÑO DE APLICACIONES DISTRIBUIDAS

Una aplicación distribuida es una aplicación cuyo procedimiento se distribuye por múltiples computadoras de una red.

Las aplicaciones distribuidas se implementan típicamente como sistemas cliente/servidor organizados en conformidad con la interfaz del usuario, el procesamiento de la información y las capas de procesamiento de la información.

Capa de interfaz de usuario viene implementada por una aplicación cliente.

Ejemplo: los programas de correo electrónico y navegadores Web.

Capa de procesamiento de la información la implementa la aplicación del cliente, una aplicación servidor o una aplicación con soporte de servidor.

Ejemplo

Una aplicación puede utilizar un cliente de bases de datos para convertir las selecciones del usuario en instrucciones SQL; un servidor con acceso a base de datos puede ser utilizado para admitir la comunicación entre el cliente y un servidor de bases de datos, y el servidor de bases de datos puede usar SW de información para procesar la información solicitada por el cliente.

Capa de almacenamiento de la información la implementan servidores de bases de datos, servidores Web, servidores FTP, servidores de archivo y cualquier otro servidor, cuya finalidad sea almacenar y recuperar información.

LAS TRES CAPAS DE RMI DE JAVA

Aparte de las interfaces remotas, el modelo utiliza clases pertenecientes al fragmento adaptador y al esqueleto de forma muy parecida a como lo hace CORBA. Las *clases de fragmento adaptador* actúan como proxies remotos. Ambas clases implementan la interfaz remota del objeto servidor. La interfaz cliente invoca a los métodos del objeto fragmento adaptador local. El fragmento adaptador local comunica estas invocaciones de métodos al esqueleto remoto, mientras que este último invoca a los métodos del objeto servidor. El objeto servidor devuelve un valor al objeto adaptador, mientras que este último devuelve el valor al cliente.

Las clases de fragmento adaptador y esqueleto las genera automáticamente el compilador rmic a partir del objeto servidor. Estas clases so clases verdaderas de Java y no dependen de un IDL externo. No se necesita un ORB, ya que la RMI de Java es una solución de Java puro. El objeto cliente y el fragmento adaptador se comunican por medio de las invocaciones normales de métodos de Java, al igual que lo hacen el esqueleto y el objeto servidor. El fragmento adaptador y el esqueleto se comunican a través de una **capa de referencia remota**.

Esta capa de referencia remota admite la comunicación entre el fragmento adaptador y el esqueleto. Si el fragmento adaptador se comunica con más de una instancia de esqueleto, el objeto fragmento adaptador se comunicará con esqueletos múltiples de un modo multidifusión. La capa de referencia remota también puede ser utilizada para activar los objetos servidor cuando se invoquen remotamente.

La capa de referencia remota del host local se comunica con la capa de referencia remota del host remoto a través de la **capa de transporte** de la RMI. La capa de transporte configura y administra las conexiones que se dan entre los espacios destinados a direcciones de los host remoto y local, controla los objetos a los que se puede acceder remotamente y determina cuándo las conexiones están en compás de espera y se vuelven inoperativas. La capa de transporte utiliza por defecto sockets TCP para comunicarse entre los hosts local y remoto. No obstante se pueden usar también otros protocolos de capa de transporte como SSL y UDP.

Esta figura ilustra las tres capas que se usan para implementar la RMI de Java. En esta vista ampliada del modelo, el objeto cliente invoca a los métodos del fragmento adaptador local del servidor. El fragmento adaptador local utiliza la capa de referencia remota para comunicarse con el esqueleto del servidor. La capa de referencia remota utiliza la capa de transporte para establecer una conexión entre los espacios de direcciones locales y remotas y para obtener una referencia del objeto esqueleto.

Con el fin de que se pueda acceder remotamente a un objeto servidor, éste se debe registrar a sí mismo con el registro remoto. Esto lo hace asociando su instancia de objeto a un nombre. El registro remoto es un proceso que se ejecuta en el host del servidor y se crea ejecutando el programa rmiregistry, otra herramienta del JDK.

El registro remoto mantiene una base de datos de objetos servidor y los nombres en virtud de los cuales se puede hacer referencia a esos objetos. Cuando un cliente crea una instancia de la interfaz de un objeto servidor, (su fragmento adaptador local), la capa de transporte del host local se comunica con la capa de transporte del host remoto para determinar si el objeto al que se ha hecho referencia existe y para ver el tipo de interfaz que implementa el objeto al que se ha hecho referencia. La capa de transporte del lado del servidor utiliza el registro remoto para acceder a esta información. Un proceso separada, al que se llama Demonio de Sistema de Activación de la RMI de Java, da soporte a la activación de objetos remotos. Este proceso se ejecuta por medio del programa rmid del JDK en el sistema remoto.

LOS OBJETOS Y LA INVOCACIÓN REMOTA DE METODOS

Los requisitos mínimos para que se pueda acceder a los objetos remotamente:

- La clase del objeto debe implementar una interfaz que amplíe la interfaz Remote. Esta interfaz debe definir los métodos que el objeto va a permitir que se invoquen remotamente. Estos métodos deben arrojar la excepción RemoteExcepción.
- La clase del objeto debe ampliar la clase RemoteServer. Esto se hace normalmente ampliando la subclase UnicastRemoteObject de RemoteServer.
- Las clases de fragmento adaptador y el esqueleto de la clase objeto deben ser creados por medio del compilador rmic. El fragmento adaptador debe ser distribuido al host del cliente.
- La clase, interfaz y clase del esqueleto objeto remoto deben estar en la CLASSPATH del host remoto.
- El demonio de activación remota y el registro remoto deben estar activados.
- Se debe crear una instancia de objetos remotos, y se debe registrar en el registro remoto. Los métodos bind() y rebind() de la clase Naming se utilizan para registrar un objeto con su nombre asociado. El objeto remoto debe instalar una administrador de seguridad para permitir la carga de las clases de la RMI.

LA RMI Y LAS APLICACIONES DISTRIBUIDAS

Las *aplicaciones distribuidas* son aquellas que se ejecutan por sistemas host múltiples. Los objetos que se

están ejecutando en un host invocan a los métodos de los objetos de otros host para ayudarles a realizar su procesamiento. Estos métodos se ejecutan en host remotos, de ahí el nombre *invocación remota de métodos.* Los objetos invocados remotamente realizan cálculos y pueden devolver valores que son utilizados por los objetos locales.

Se ha desarrollado una serie de enfoques para la implentación de sistemas distribuidos. Internet y el Web constituyen ejemplos de sistemas distribuidos que han sido desarrollados por medio del enfoque TCP/IP cliente/servidor. Los clientes y servidores se comunican a través de sockets TPC y UDP. Mientras Internet y el Web tienen mucho éxito, el uso de sockets requiere que haya protocolos a nivel de la aplicación para que se establezca una comunicación entre cliente y servidor. La infraestructura viene asociada a estos protocolos prohibe el funcionamiento paralelo completo que se da en otros enfoques.

DESARROLLO DE OBJETOS RMI

Como ejemplo puede tomarse una aplicación financiera que se esté ejecutando de un PC puede invocar métodos de objetos que estén ejecutando en otro PC que pertenezca a la intranet de su empresa. Puede ser que estos objetos busquen en las bases de datos de empresas la información que su aplicación financiera está utilizando, que procesen esta información de acuerdo con las reglas comerciales de su empresa y que la faciliten a su aplicación financiera. Los resultados que haya calculado su aplicación financiera pueden ser automáticamente reenviados a un objeto de distribución de información, el cual pondrá a los resultados a disposición de otros empleados de su empresa, además de a los agentes y clientes seleccionados.

RMI permite que se ejecuten objetos Java en un host para invocar a los métodos de objetos que se ejecuten en host remotos. Los objetos invocados remotamente realizan servicios y pueden devolver valores que los objetos locales utilizan.

La ventaja principal de RMI consiste en que está completamente integrada con el modelo de objetos de Java, es altamente intuitiva y muy fácil de usar.

RMI entraña que se pueden construir aplicaciones distribuidas por medio de objetos remotos de Java, se pueden adquirir o ser desarrollado por otros programadores. También puede desarrollar objetos distribuidos que otros pueden a su vez utilizar.

DESARROLLO DE OBJETOS CORBA

Las grandes aplicaciones de empresa tienden a ser heterogéneas y a veces Java por sí mismo no es suficiente. Para estos casos una aproximación del desarrollo de objetos distribuidos de lenguaje mixto, como puede ser el que admite la ARQUITECTURA DE INTERMEDIACIÓN DE SOLICITOS DE OBJETOS COMUNES (CORBA), constituye una solución para el desarrollo de aplicaciones.

El punto fuerte de CORBA es que admite un modelo de lenguaje independiente. Ofrece una arquitectura estándar para el desarrollo de objetos distribuidos orientados al objeto, esta arquitectura especifica cómo un objeto del cliente que está escrito en un lenguaje puede invocar los métodos de un objeto remoto que se ha desarrollado en un lenguaje diferente.

Corba hace uso de los objetos accesibles a través de los Intermediarios de Solicitud de Objetos (ORB). Los ORB se utilizan para conectar los objetos entre si en una red. Un objeto que está en una computadora (el objeto cliente) invoca los métodos de un objeto que está en otra computadora (el objeto servidor) a través de una ORB.

La interfaz del cliente ORB es un fragmento adaptador que está escrito en el Lenguaje de Definición de Interfaz (IDL). El fragmento adaptador es un proxy local de un objeto remoto. El IDL proporciona un

mecanismo independiente de lenguaje de programación para describir los métodos de un objeto.

La interfaz del ORB con el servidor se hace a través de un esqueleto IDL. El esqueleto dota al ORB con un mecanismo independiente del lenguaje para acceder al objeto.

Invocación remota de métodos bajo CORBA

El objeto del cliente invoca los métodos del fragmento adaptador del IDL que corresponde al objeto remoto. El fragmento adaptador del IDL comunica al ORB las invocaciones de métodos. El ORB invoca a los métodos correspondientes del esqueleto IDL. El esqueleto IDL invoca a los métodos de la implementación de objetos del servidor remoto. El objeto del servidor devuelve el resultado de la invocación del método a través del esqueleto IDL, el cual devuelve el resultado al ORB. Este último devuelve el resultado al fragmento adaptador del IDL, quien a su vez lo devuelve al objeto cliente.

TERMINOLOGIA RMI

RMI se construye en base a la noción fundamental de los objetos locales y remotos.

- **Objetos locales:** son objetos que se ejecutan en un host determinado.
- **Objetos remotos:** objetos que se ejecutan en todos los demás host.

Los objetos que están en hosts remotos se **exportan** para que puedan ser invocados remotamente.

Un objeto que se exporta a sí mismo registrándose con un **servidor de registro remoto.** Este servidor ayuda a los objetos de lo que otros hosts a acceder remotamente a sus objetos registrados. Lo hace manteniendo una base de datos de nombres y de los objetos que están asociados a estos nombres.

Los objetos que se exportan para acceso remoto deben implementar la interfaz Remote. Esta interfaz identifica el objeto como susceptible de ser accedido remotamente. Cualquier método que se vaya a invocar remotamente debe arrojar la excepción RemoteException. Esta excepción se utiliza para indicar los errores que puedan producirse durante una RMI.

El enfoque RMI de Java se organiza en una estructura cliente/servidor. Un objeto local que invoca un método del objeto remoto recibe el nombre de **objeto cliente** o **cliente.** Un objeto remoto cuyos métodos se invocan por un objeto local recibe el nombre de **objeto servidor,** o **servidor.**

El enfoque RMI de Java utiliza fragmentos adaptadores y esqueletos. Un **fragmento adaptador** es un objeto que actúa como proxy local del objeto remoto. El fragmento adaptador ofrece los mismos métodos que el objeto remoto. Los objetos locales invocan a los métodos del fragmento adaptador como si fueran los métodos del objeto remoto. El fragmento adaptador comunica estos métodos al objeto remoto a través del esqueleto que se implementa en el host remoto. El esqueleto es un proxy para el objeto que esta situado en el mismo host que el objeto remoto. El esqueleto se comunica con el fragmento adaptador local y propaga invocaciones de métodos al fragmento adaptador del objeto remoto activo. Luego recibe el valor que devuelve la invocación remota de métodos (sí lo hubiera) y pasa este valor de nuevo al fragmento adaptador. El fragmento, envía el valor de retorno del objeto local que iniciara la invocación remota de métodos.

Los fragmentos adaptadores y esqueletos se comunican a través de una capa de referencia remota. Esta capa proporciona a los fragmentos adaptadores la posibilidad de comunicarse con los esqueletos a través de un protocolo de transporte. La RMI actualmente usa TCP para el transporte de la información, si bien es lo suficientemente flexible como para utilizar otros protocolos.

Objetos distribuidos con Java RMI

Un método simple para crear aplicaciones distribuidas con Java

Java dispone de su propia técnica, conocida como RMI, que facilita la creación y uso de objetos distribuidos.

Mediante la computación distribuida es posible realizar un mejor aprovechamiento de los recursos informáticos de los que disponga una empresa o institución, dividiendo las distintas funciones a ejecutar y entregándolas a los equipos más apropiados para ello. La estructura centralizada en la que existe un host u ordenador central con terminales tontos colgados de él, la tan difundida estructura cliente/servidor o la más actual conocida como multi−tier son implementaciones, a un nivel u otro, de computación distribuida. La mayoría de las técnicas actuales están construidas con algún mecanismo basado o derivado de RPC (Remote Procedure Call). Mediante RPC es posible el envío de mensajes entre varias aplicaciones que se ejecutan en dos o más equipos independientes, usando para ello un protocolo de red. Dichos mensajes, que serían las llamadas a procedimientos remotos, van en muchas ocasiones acompañados de parámetros, lo cual supone un problema añadido. Los parámetros son datos que pertenecen a un determinado tipo, generalmente dependiente del lenguaje que se esté usando. Es en este punto donde aparece DCE (Distributed Computing Environment) como un estándar que establece cuál será la representación de los tipos de datos generales en operaciones RPC, dando lugar a lo que se conoce como DCE RPC. Esa representación, a veces llamada NDR (Network Data Representation), es independiente del lenguaje y el sistema, lo cual garantiza la correcta gestión de los parámetros. Para adaptarse a DCE RPC, cualquier aplicación que quiera realizar una llamada remota tiene que efectuar una preparación previa de los parámetros, proceso conocido como marshalling. La aplicación que recibe el mensaje efectúa la operación complementaria, el unmarshalling, recuperando los parámetros originales sobre los que tiene que operar. En realidad, de estos procesos generalmente no se ocupa el programador de la aplicación, sino los mecanismos propios del modelo RPC que esté usando. Implementaciones RPC Actualmente existen diversas implementaciones RPC para los sistemas operativos más usados, como Windows o Unix. Los contendientes más representativos, por su difusión y respaldo, son DCOM y CORBA. Frente a ellos aparece RMI como una tercera alternativa, íntimamente ligada al lenguaje Java. DCOM (Distributed COM) está basado en el modelo de componentes COM (Component Object Model) de Microsoft. Podríamos decir que DCOM es un COM que utiliza mecanismos DCE RPC para permitir el uso remoto de interfaces de objetos. El mayor inconveniente de DCOM es que está fundamentalmente unido a Windows, no existiendo demasiadas implementaciones para otros sistemas operativos. Esto, en un universo distribuido como el que abre Internet, es un serio inconveniente, ya que nos obliga a tener Windows en todos los puestos de trabajo. CORBA (Common Object Request Broker Architecture) es una completa y compleja implementación RPC que facilita la comunicación entre objetos distribuidos con independencia del lenguaje y la plataforma, realmente un sueño hecho realidad. No obstante, siempre tiene que haber un pero, CORBA tiene fama de ser difícil tanto a la hora de crear objetos remotos como a la hora de usar dichos objetos desde otras aplicaciones. Se podrían escribir cientos de páginas tanto hablando de DCOM como de CORBA, pero el tema que nos interesa en este artículo es RMI (Remote Method Invocation). En contraposición a DCOM, se trata de una implementación independiente de la plataforma, lo que permite que tanto los objetos remotos como las aplicaciones cliente residan en sistemas heterogéneos. No contamos, sin embargo, con una independencia del lenguaje. RMI, como se ha apuntado anteriormente, está ligado estrechamente al lenguaje Java. A pesar de todas las ventajas y desventajas que podamos deducir de estos modelos, DCOM, CORBA y RMI, seguramente el punto que inclina la balanza a favor de éste último es su facilidad. Crear un objeto remoto RMI no es mucho más complejo que crear cualquier objeto Java. Usar un objeto remoto con RMI es casi como utilizar un objeto local cualquiera.

¿Cómo funciona RMI?

Tras esta breve introducción, y antes de adentrarnos en los detalles de la creación de servidores y clientes RMI, veamos cuáles son los puntos fundamentales de este mecanismo.

La finalidad es conseguir tener una imagen global para, posteriormente, ir analizando los diversos elementos.

Comencemos imaginando que disponemos de dos ordenadores conectados a una misma red, independientemente del tipo de ésta. Uno de ellos cuenta con un servicio que el otro necesita utilizar. Dicho servicio podría ser desde el acceso a una base de datos hasta las reglas de negocio de una aplicación. Sin embargo, tomemos un ejemplo mucho más sencillo y quizá más accesible: el primer ordenador, al que nos referiremos como servidor, dispone de una batería de lectores de CD−ROM con enormes cantidades de información, como podría ser una biblioteca digital. El otro ordenador, al que llamaremos cliente, no dispone de dicha información pero la precisa. En lugar de facilitar a cada usuario de la empresa que lo necesite toda la batería de CD−ROM con la biblioteca, algo que resultaría bastante caro, es mucho más lógico que el servidor permita realizar consultas remotas, de tal forma que cualquier cliente pueda usar la biblioteca como si dispusiese de ella localmente.

Mediante RMI, el servidor dispondrá de un objeto al que podría accederse remotamente para hacer las consultas. Para el cliente, dicho objeto aparecería como local, obteniéndose los resultados deseados. La configuración hardware con que contamos podría ser la esbozada en la figura siguiente:

Gráfico 1

En este caso son dos clientes conectados al servidor mediante una red, que podría ser local o la propia Internet. El servidor dispone de la biblioteca a la que han de hacerse las consultas. Está claro que un objeto creado en uno de los clientes no puede consultar directamente la biblioteca, sino que tendrá que ejecutar un método del servidor que se encargue de este trabajo. No obstante, tampoco se puede realizar una llamada directa de un ordenador a un método de un objeto que está en otro ordenador, ya que por medio está la red. Mediante RMI el servidor puede crear y registrar un objeto dejándolo preparado para su uso remoto. Un registro RMI se encarga de controlar las peticiones de objetos por parte de los sistemas remotos. El cliente, al crear el objeto remoto, sirviéndose de una interfaz genérica, lo que obtiene es una clase llamada stub. Las llamadas del cliente a métodos del objeto remoto son recibidas por este stub, que se encarga de preparar los parámetros que puedan existir en la llamada, enviando ésta al ordenador destinatario. Cuando el registro RMI recibe una solicitud remota busca un esqueleto asociado al objeto servidor. Dicho esqueleto se encarga de descodificar los parámetros que acompañan a la llamada y de invocar realmente al método del objeto destinatario. Éste efectúa el proceso que le corresponda y posiblemente genera un valor de retorno, que será preparado por parte del esqueleto y devuelto por la red al stub del cliente, que a su vez lo descodificará y hará llegar al código que inicialmente realizó la llamada. Esta es, a grandes rasgos, la idea general del funcionamiento de RMI, representada gráficamente en la figura siguiente:

Gráfico 2

En los puntos siguientes se analizará con detalle todo el proceso necesario para la creación de un servidor, cliente, stub, esqueleto y puesta en marcha del registro RMI.

a) Creación de un servidor.

La interfaz genérica del objeto servidor

El primer paso que habremos de dar en la construcción de nuestro objeto remoto es la definición de una interfaz que especifique qué servicios ofrecerá. Esta interfaz será algo así como el contrato existente entre servidor y cliente, de tal forma que el primero se compromete a dar servicio a las llamadas documentadas en la interfaz, mientras que el segundo sabe que no podrá pedir nada más. Como todos los contratos, éste se entrega tanto al cliente como al servidor. Dicho de otra forma, la interfaz deberá estar disponible tanto para el servidor como para el cliente. Continuando con el ejemplo anterior, en el que el servidor permitirá a los clientes realizar consultas, podríamos definir una interfaz tan simple como la siguiente:

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

public interface InterfazServidor

extends Remote

{

public String Consulta(String Crit) throws RemoteException;

}

Lo único destacable en este fragmento es el hecho de que la interfaz definida deriva de Remote y que el único método existente puede generar la excepción RemoteException. Esto es una condición indispensable. Dicha interfaz no define ningún método que nosotros debamos implementar, simplemente marca aquellas clases o interfaces como utilizables de forma remota. Los métodos remotos pueden provocar la excepción citada por múltiples causas, por ejemplo, la imposibilidad de conexión por un fallo de red. Ambos elementos, la interfaz Remote y la excepción RemoteException, se encuentran en el paquete java.rmi. En este caso nuestra interfaz cuenta con un sólo método que será el encargado de recibir el criterio, realizar la consulta y devolver el resultado. Realmente los clientes tan sólo necesitan saber que los objetos remotos que se ajustan a la interfaz InterfazServidor cuentan con el método Consulta, todo lo demás son detalles de implementación propios del servidor. Implementación de una clase remota Disponiendo de la interfaz, el paso siguiente será definir la verdadera clase que permitirá crear los objetos a utilizar de forma remota. Dicha clase habrá de extender necesariamente la clase UnicastRemoteObject o bien otra que derive de RemoteServer. Al tiempo, también deberá implementar la interfaz que hemos definido anteriormente, por lo que de partida tendríamos una cabecera similar a la siguiente:

public class Servidor

extends UnicastRemoteObject implements InterfazServidor

La clase UnicastRemoteObject no precisa que nosotros implementemos método alguno, por lo que en este caso tan sólo deberíamos codificar el método Consulta de la interfaz InterfazServidor. En realidad también deberemos de implementar el constructor de la nueva clase. En él se llamará al constructor base, tras lo cual se procederá a registrar el objeto en el registro RMI, dándolo así a conocer y permitiendo que los clientes lo usen. Al igual que cualquier otro método, el constructor también puede generar la excepción RemoteException. Para establecer el nombre en el registro RMI del objeto recién creado hay que utilizar el método rebind del objeto Naming, que también forma parte del paquete java.rmi. Dicho método toma como primer parámetro el nombre a asignar y, como segundo, una referencia al objeto. Esta operación puede generar una excepción en caso de que el registro RMI no esté funcionando, excepción que podemos interceptar o dejar que se propague. En el siguiente fragmento puede ver cómo en el constructor de la clase Servidor se dan los pasos comentados.

public Servidor()

throws RemoteException {

super(); // constructor base try {

Naming.rebind(ServidorRMI Simple, this);

}catch(Exception x) {

System.out.println(Se ha producido un error);

```
}

}
```

En este caso el objeto de la clase Servidor se incluye automáticamente en el registro RMI en el momento de la construcción, usando el nombre ServidorRMI Simple como identificador teóricamente único. El objeto estará accesible para los clientes en forma de URL con la forma rmi://servidor/objeto, donde servidor es el nombre o dirección IP del sistema y objeto el nombre de la clase remota.

Creación del objeto remoto.

Los clientes sólo pueden comunicarse con el servidor realizando llamadas a métodos de un objeto existente y reconocido en el registro RMI. Es decir, el objeto Servidor deberá estar creado y a la espera en el servidor, de lo contrario los clientes no podrán entrar en contacto con él. Podemos crear un objeto Servidor desde otra clase de nuestro proyecto o bien añadiendo la función main a la propia clase. Esto es precisamente lo que se hace en el código del listado nº 1, correspondiente al archivo Servidor.java. Observe que antes de proceder a la creación del objeto se establece un gestor de seguridad, concretamente uno de clase RMISecurityManager. Fíjese también en que el método Consulta se limita a sacar un mensaje por la consola y devolver una cadena, sin efectuar ningún proceso extraordinario. En la práctica este código se vería sustituido por el necesario para realizar la consulta real. En el listado nº 2 tiene el código correspondiente al archivo Interfazservidor.java, en el que se define la interfaz que implementa la clase Servidor.

Listado nº 2: definición de la interfaz del servidor

```
/*

InterfazServidor.java

Definimos la interfaz que permitirá a los clientes conocer los servicios que ofrece el

objeto remoto

*/

package RMISimple;

import java.rmi.*;

// Interfaz del objeto remoto

public interface InterfazServidor extends Remote

{

// un método para realizar la consulta

public String Consulta(String Criterio) throws RemoteException;

}
```

Ambos archivos forman parte de un mismo paquete que deberá alojarse en un directorio llamado RMISimple, justo en el nivel siguiente al indicado por la variable de entorno CLASSPATH. Lógicamente puede crear la

citada carpeta en cualquier punto y después modificar apropiadamente esa variable. Si en este momento se compilan los dos archivos de que disponemos, primero InterfazServidor.java y después Servidor.java, y se intenta ejecutar Servidor verá que obtiene un error. Si observa el código del listado nº 1, podrá deducir que el error se produce en el momento en que se intenta llamar al método rebind del objeto Naming. Lo que ocurre es que el registro RMI no está en funcionamiento, por lo que cualquier intento de acceder a él provoca una excepción. Para ponerlo en marcha introduzca la sentencia start rmiregistry y pulse Intro. De esta forma el registro se iniciará en una ventana independiente, permitiéndole ejecutar a continuación la clase Servidor. Verá que ahora obtiene un mensaje positivo, el objeto ha sido creado y está a la espera de solicitudes por parte de los clientes. Objetos remotos y seguridad Java es un lenguaje diseñado teniendo muy en cuenta la seguridad e integridad del código que se ejecuta, evitando actuaciones peligrosas.

Parte de esta seguridad se apoya en las clases derivadas de SecurityManager, encargadas de limitar el acceso al sistema de archivos, la posibilidad de imprimir, manipular hilos de ejecución, abrir nuevas conexiones, etc. Los applets Java cuentan con un gestor de seguridad muy restrictivo, mientras que las aplicaciones estándar Java no tienen gestor de seguridad, por lo cual tienen acceso a todos los recursos disponibles. Un objeto que va a ejecutarse de forma remota ha de contar con una seguridad que garantice la imposibilidad de realizar operaciones peligrosas, gestor que se define en la clase RMISecurityManager.

Listado nº 1: IMPLEMENTACIÓN DE LA INTERFAZ DEL SERVIDOR

```
/*

Servidor.java

El servidor implementa la interfaz que actúa como contrato, aunque puede añadir

otros métodos.

*/

package RMISimple;

import java.rmi.*;

import java.rmi.server.*;

// Toda clase remota RMI ha de extender UnicastRemoteObject o

// una clase derivada, implementando la interfaz remota

public class Servidor extends UnicastRemoteObject

implements InterfazServidor

{

public Servidor() throws RemoteException {

super(); //constructor base

try {
```

```
// Establecemos el nombre del servicio asociándolo con
// el objeto recién creado
Naming.rebind(ServidorRMI Simple, this);
// e indicamos que todo ha ido bien
System.out.println(Servidor RMI simple esperando solicitudes);
}catch(Exception x) { // si hay algún error
// lo indicamos igualmente
System.out.println(Se ha producido un error en Servidor RMI simple);
}
}
// Este es el método remoto que será llamado por los clientes
public String Consulta(String Criterio) throws RemoteException {
System.out.println(Atendiendo consulta < + Criterio + >);
return Consulta < + Criterio + > procesada; }
// en la función main() ponemos en marcha el servidor
public static void main(String args[]) {
// estableciendo un gestor de seguridad RMI
System.setSecurityManager(new RMISecurityManager());
try
{
// y creando un objeto Servidor que quedará a la espera
Servidor MiServidor = new Servidor();
}
catch(Exception x) {
System.out.println(Se produce un error al crear el objeto Servidor);
}
```

}

}–

En el listado nº 1 puede ver cómo se crea un objeto de esa clase y se usa como parámetro en la llamada al método setSecurityManager del objeto System. De esta forma creamos un gestor de seguridad y lo instalamos en el sistema, facilitando el normal funcionamiento de RMI.

Si no damos el paso anterior no sólo estaremos pasando por alto la seguridad, sino que además impediremos la carga de otras clases, ya sean éstas locales o remotas. En caso de que el objeto RMISecurityManager no se ajustase a nuestras necesidades, siempre podemos crear una nueva clase y redefinir los métodos que nos convengan, por ejemplo, abriendo el número de operaciones posibles. El gestor de seguridad ha de utilizarse tanto en el servidor como en el cliente, a no ser que éste último sea un applet Java, en cuyo caso ya cuenta con su propio gestor por defecto con limitaciones mayores a las de RMISecurityManager.

b) Diseño de un cliente.

Llegados a este punto contamos con un registro RMI en marcha y un objeto esperando en nuestro servidor a ser usado remotamente. Tendremos, por lo tanto, que crear un programa cliente que acceda al objeto remoto y use el único método disponible. Si todo va bien obtendremos una cadena como resultado, cualquier fallo, ya sea de red o del propio mecanismo RMI, generará una excepción. Un cliente RMI que quiere usar un objeto remoto tan sólo cuenta con dos puntos que le diferencian de cualquier otro programa: la instalación de un gestor de seguridad y la forma en que obtiene la referencia al objeto que desea utilizar. Sobre la primera no hay mucho que decir, ya que usaremos exactamente el mismo método visto antes en el servidor. La segunda es que el objeto en lugar de crearlo dinámicamente, como se haría en un programa normal, tiene que usar el registro RMI para localizarlo y obtener una referencia. Tendremos que usar, por lo tanto, el mismo objeto Naming utilizado en el servidor para realizar el registro. En esta ocasión lo que queremos es localizar un objeto y obtener una referencia a él, función que desempeña el método lookup. Éste toma como único parámetro el URL del objeto al que se quiere acceder, devolviendo una referencia que deberá ser apropiadamente convertida. En caso de no localizarse el objeto, se genera una excepción.

El listado nº 3 muestra el código del archivo Cliente.java, parte del paquete RMISimple en el que se encuentran las demás clases de ejemplo.

Listado nº 3: implementación del cliente.

/*

Cliente.java

Este programa usa el objeto remoto ServidorRMI Simple llamando al método

Consulta tantas veces comoparámetros se hayan pasado en la línea de comandos.

*/

package RMISimple;

import java.rmi.*;

public class Cliente {

```
public static void main(String[] args) {

try {

//Establecemos el gestor de seguridad

System.setSecurityManager(new RMISecurityManager());

// Obtenemos la referencia al objeto

InterfazServidor Objeto = (InterfazServidor )

// especificando su URL

Naming.lookup(rmi://ntserver/ServidorRMI Simple);

// Recorremos los parámetros entregados

for(int Ind=0; Ind <args.lenght: ind++)

// y los usamos para llamar al método consulta

System.out.println(Objeto.Consulta(args[ind]));

catch(Exception x) // en caso de fallo

System.out.println(Se produce un error en el cliente);

}

}

}
```

Como puede ver, tras intentar obtener una referencia al objeto que está ejecutándose en el servidor, mediante el método lookup mencionado antes, se llama al método Consulta en el interior de un bucle, pasando como parámetros los dados en la línea de comandos. Fíjese en que se ha indicado como nombre de servidor ntserver, nombre que tiene el servidor usado para desarrollar este ejemplo. Deberá sustituir ese parámetro estableciendo el nombre de su propio sistema. Si compila e intenta ejecutar el cliente verá que obtiene un mensaje de error, no tomándose los parámetros de la línea de comandos para realizar una consulta al servidor.

c) Generación del stub y el esqueleto.

Lo único que nos falta para que nuestro cliente funcione, accediendo al objeto del servidor y ejecutando sus métodos, es la generación de las clases que actuarán como stub, en el cliente, y esqueleto, en el servidor. Recuerde que la finalidad de estas clases es realizar la preparación de los parámetros en ambos sentidos, mediante los procesos de marshalling y unmarshalling comentados al principio del artículo. No tenemos que codificar estas clases a mano, aunque sería posible hacerlo. Vamos a dejar todo el trabajo en una utilidad llamada Rmic.exe, que se encuentra entre los ejecutables del JDK al igual que el programa Rmiregistry.exe que hemos usado antes para poner en marcha el registro RMI. Al llamar a rmic hay que pasar como parámetro el nombre de la clase remota, en este caso Servidor, incluyendo, como es habitual, el nombre del paquete.

Tras ejecutar el programa Rmic en el directorio tenemos dos nuevas clases, ya compiladas, que tienen el mismo nombre de la clase original con los sufijos _Stub y _Skel. En este caso tanto cliente como servidor están ejecutándose en un mismo sistema, aunque en máquinas virtuales Java diferentes. Bastaría con copiar a cualquier ordenador conectado a la misma red los archivos InterfazServidor.class, Cliente.class y Servidor_Stub.class para tener el servidor en una máquina y el cliente en otra. Tanto el cliente como el servidor pueden usarse en cualquier sistema operativo que disponga de una máquina virtual Java versión 1.1 o posterior. El cliente podría estar en Windows y el servidor en Linux, por poner un ejemplo. Los distintos ordenadores habrán de tener instalada una pila TCP/IP, algo habitual en todos los sistemas que cuentan con conexión a Internet. Uso de un entorno de desarrollo Java Hasta ahora se ha visto cómo crear y poner en marcha un cliente y un servidor RMI, incluyendo la generación de los archivos _Stub y _Skel, usando sólo las herramientas que se facilitan con el JDK. Cuando las necesidades van más allá de la simple realización de algunas pruebas, sin embargo, el uso de múltiples herramientas a nivel de línea de comandos no constituye el método más cómodo ni rápido. Lo habitual es usar algún entorno de desarrollo que facilite el trabajo, permitiendo la edición, compilación y depuración, ocupándose de tareas comunes, etc.

Puesto en marcha el servidor, se puede ejecutar la aplicación cliente, bien desde el mismo equipo, en una máquina virtual distinta, o bien desde otro equipo conectado a la red. Como se ha apuntado antes, lo único preciso es una máquina virtual Java compatible con la versión 1.1 del JDK y servicios de red TCP/IP. Pase de parámetros entre objetos Facilitar parámetros al llamar a una función o método de un objeto es algo muy habitual, así como devolver un valor de retorno. Cuando el código que realiza la llamada y el llamado forman parte de una misma aplicación, el pase de parámetros no representa ninguna complicación y es una tarea muy rápida en la que, normalmente, se utilizan los registros del procesador y la pila. Si los parámetros han de pasar de una aplicación a otra y volver, dentro de la misma máquina, el tema se complica más o menos dependiendo del sistema operativo. En Windows, por ejemplo, la complejidad y velocidad dependen de si el código al que se llama forma parte de una librería de enlace dinámico, caso en el cual se ejecuta en el mismo proceso que el programa que la usa, o si es una aplicación independiente. El caso más complejo se presenta cuando los parámetros no sólo han de pasar de una aplicación a otra, sino que además han de transportarse entre dos máquinas que, presumiblemente, pueden estar ejecutando sistemas operativos distintos. Para posibilitar esta comunicación los mecanismos RPC tradicionales lo que hacen es buscar un mínimo común denominador, de tal forma que sólo permiten pasar parámetros de tipos intrínsecos, tipos que dicho mecanismo sabe cómo convertir entre los distintos sistemas y cómo transportar por la red. RMI está hecho en Java para ser usado en Java, indistintamente del sistema operativo en el que resida la máquina virtual. Esto significa que no hay necesidad de buscar ese mínimo común denominador, permitiéndose el pase de cualquier tipo de parámetro, tanto básicos, como puede ser un número entero, como complejos, como pueden ser los objetos.

Cuando el tipo de dato a pasar como parámetro es de tipo intrínseco, un número entero es el caso más claro, RMI usa el mismo sistema que los demás mecanismos RPC. Es decir, se pasa una copia del valor al método llamado. Éste puede usar el valor como le interese y puede modificarlo, sabiendo siempre que se trata de una copia local cuyo valor no volverá al cliente que ha realizado la llamada. Si el tipo de un parámetro no es intrínseco sino un objeto, como puede ser un String, Java RMI usa sistemas mucho más elaborados para el transporte del dato. Éstos mecanismos son posibles porque se trabaja con la seguridad de que tanto cliente como servidor están funcionando con una máquina virtual Java compatible con RMI y las capacidades de publicación. Parámetros que son objetos no remotos Comencemos analizando el caso más habitual y sencillo: el cliente pasa al servidor un parámetro que es un objeto no remoto, o bien, el servidor devuelve al cliente un objeto no remoto. El tipo String, citado antes y usado en el ejemplo propuesto previamente, es un buen ejemplo de objeto no remoto. Este tipo no es intrínseco de Java, sino que está implementado como una clase. De igual forma podría servir cualquier tipo definido por nosotros mismos. En una aplicación Java estándar al llamar a un método pasando como parámetro un objeto, lo que recibe dicho método es una referencia, o dicho de otro modo, un puntero que indica la localización en memoria de ese objeto. Si el método al que se llama pertenece a un objeto remoto, que está ejecutándose en otra máquina, una referencia no tiene sentido alguno, ya que la hará a algo que existe sólo en la máquina local. Lo que necesita el método al que se llama, por lo tanto, es una copia del objeto. Pasar un parámetro por valor es algo que no representa problemas cuando el

tipo de dato es simple, pero que puede ser realmente muy complejo en el caso de los objetos. Java RMI afronta este tema usando el mecanismo de publicación (serialization) propio de Java, de tal forma que cuando un objeto va a ser pasado como parámetro se crea un flujo de bytes en el cuál se escribe el objeto. Por suerte el mecanismo de publicación de Java es totalmente automático, de tal forma que no necesitamos hacer nada en especial para que un objeto pueda ser pasado como parámetro a un método remoto. El flujo de datos que se crea describe por completo al objeto que se pasa como parámetro, usando un formato que es independiente de la red y el sistema en el que se ejecutan las JVM. No es importante, por ejemp o, el orden de los bytes en una palabra. Parámetros que son objetos remotos Cualquier objeto que extienda la clase Remote u otra derivada es un objeto remoto, preparado para ejecutarse localmente en la máquina en que reside, a petición de llamadas externas recibidas de clientes remotos. Dicho de otra forma, esta clase de objetos no están hechos para ser transportados por la red y ejecutarse localmente en el cliente, como es el caso de los que se han descrito en el punto anterior. Los objetos remotos suelen ser devueltos como resultado por parte del servidor al cliente, previa llamada de éste último a un método remoto de inicialización o mecanismo similar. Está claro que el valor devuelto por el servidor no puede ser una referencia, ya que ésta no tendría sentido en el cliente. También es lógico pensar que el objeto devuelto no se transporta mediante el mecanismo de publicación Java, ya que en ese caso lo que obtendría el cliente sería una copia del objeto original, copia que se ejecutaría localmente. Según el esquema esbozado al principio, la invocación remota de métodos se basa en el uso de un objeto stub en el cliente y un esqueleto en el servidor. Para que el cliente pueda llamar al objeto remoto que recibe como resultado, por lo tanto, necesitará disponer del objeto stub que se encargará de la gestión de los parámetros y todo el trabajo de comunicación. Lógicamente el cliente, en el momento en que llama a un método asignando el resultado a una variable de cierta clase, debe contar con una interfaz, de lo contrario el código no podría ni compilarse. El stub, sin embargo, no tiene porqué estar disponible durante el desarrollo del cliente, ya que eso es lo que el servidor devolverá como resultado en el caso que se expone.

Resumiendo: cuando el cliente llama a un método del servidor que devuelve como resultado un objeto remoto, lo que se obtiene es el objeto stub que permitirá realizar llamadas a dicho objeto y ejecutar sus métodos remotamente. Para transportar el objeto stub se usa el sistema de publicación Java, la única diferencia es el objeto que se convierte en un flujo de bytes para ser transportado.

RMI y applets Java El ejemplo desarrollado en los listados 1 a 3 y todos los planteamientos previos hacen mención siempre a una aplicación cliente y otra servidor. Nada nos impide, sin embargo, que el cliente sea un applet Java en lugar de una aplicación estándar. De esta forma podemos usar RMI para facilitar la comunicación de funciones entre un servidor y un cliente que puede ejecutarse con cualquier navegador Web que disponga de una JVM apropiada. Como se anotó en el punto dedicado a la seguridad, los applets Java cuentan con su propio gestor, por lo que no es preciso crear y establecer uno como haríamos en una aplicación estándar. El gestor de seguridad usado por los applets Java impone muchas limitaciones, entre ellas la imposibilidad de conectar con un servidor que no sea aquel del que se ha descargado el propio applet. Esto, como vamos a ver, tiene algunas implicaciones para nosotros. Además del propio archivo de clase en que se aloja el applet, el cliente que acceda y descargue la página Web en la que se hace referencia a él también deberá tener acceso a la interfaz del servidor y el objeto stub. Estos elementos, por lo tanto, deberán poder descargarse de la red al igual que el applet, para lo cual deberán encontrarse en el mismo servidor y directorio que éste. Al igual que cualquier otro cliente, el applet usará el objeto Naming para encontrar el objeto remoto disponible en el servidor y poder ejecutar los métodos que le interesen. Al llamar al método lookup, con el fin de encontrar el objeto, es preciso facilitar el nombre del servidor, que podemos obtener mediante el método getHost() del objeto devuelto por getCodeBase(). Puesto que el applet no puede establecer conexión alguna nada más que con el servidor del que se ha descargado, es fácil deducir que el servidor ha de encontrarse necesariamente en el mismo servidor, aunque no necesariamente en el mismo directorio. Teniendo en cuenta estos puntos, desarrollar un applet Java que se comunique con el servidor de ejemplo que hemos desarrollado no es demasiado complejo.

Ventajas de RMI

A la vista de los ejemplos expuestos seguramente estará preguntándose qué le ofrece RMI que no pueda conseguir usando sistemas alternativos más habituales, como una comunicación de datos bidireccional entre cliente y servidor apoyada en un socket o similar. Si de lo que se trata es de consultar una base de datos, opciones como JDBC parecen más lógicas. No obstante, RMI ofrece posibilidades que serían muy difícil de implementar usando métodos como JDBC o una simple comunicación de datos. El mecanismo de publicación que permite que un objeto viaje por la red desde el servidor al cliente, o viceversa puesto que los papeles pueden invertirse en un determinado momento, pone a nuestro alcance el desarrollo de aplicaciones distribuidas que funcionan sin grandes inversiones de trabajo en administración de clientes. JDBC es un mecanismo ideal cuando lo que se quiere es centralizar los datos que van a ser usados por los clientes.

De forma análoga, podríamos representar a RMI como el mecanismo que permite hacer lo mismo con las reglas de negocio, que se gestionan en el servidor y viajan a los clientes en forma de objetos cada vez que es necesario. Imagine que necesita desarrollar una aplicación distribuida de tal forma que los clientes disponen de una interfaz y unas reglas que le permiten, por ejemplo, realizar determinadas operaciones con una cuenta propia. Su negocio podría ser desde un banco hasta una sencilla tienda preparada para el comercio electrónico, en cualquier caso, la parte de la aplicación que ejecutarían los clientes debería realizar ciertas comprobaciones antes de efectuar un movimiento o enviar un pedido, por poner dos ejemplos. A esto es a lo que llamamos reglas de negocio. Si éstas están centralizadas en el servidor, de tal forma que la información viaja desde el cliente al servidor con el único fin de realizar comprobaciones, lo que obtenemos es un aumento claro del tráfico en la red, lo que se traducirá en mayor lentitud de funcionamiento. En muchos casos los datos se transportarán tan sólo para que el servidor los compruebe y rechace, reiniciando el ciclo. En caso de que decidamos almacenar las reglas de negocio en la aplicación cliente solucionamos el problema anterior, el tráfico de red será mucho menor y el funcionamiento global se agilizará. El inconveniente es que cada vez que dichas reglas cambien, lo cual suele ocurrir con mayor frecuencia de la que uno se imagina al principio, será preciso actualizar la aplicación de todos los clientes que tengamos, lo cual supone una gran inversión de trabajo en la administración de esos clientes. RMI supone una solución clara a todos estos problemas. Las reglas de negocio pueden implementarse como un objeto que el cliente puede descargar del servidor, usando el mecanismo de publicación Java. El objeto se ejecuta localmente en el cliente, minimizando el tráfico de red y agilizando el funcionamiento. Si las reglas de negocio cambian tan sólo será preciso actualizar el objeto en el servidor. Los clientes lo descargarán automáticamente mediante RMI en cuanto se produzca la modificación, lo cual tiene como resultado una administración cero de los clientes. Limitaciones de RMI Como puede suponer, no todo en RMI son ventajas. La mayor limitación de este mecanismo es que está íntimamente ligado al lenguaje Java, aunque esto no parece ser una limitación dado que en los últimos tiempos pocas son las empresas o desarrolladores que no quieren usar Java. En caso de que las aplicaciones cliente vayan a distribuirse en forma de applets hay que tener en cuenta que los visualizadores Web actuales no se llevan muy bien con RMI, siendo preferible el uso de la utilidad appletviewer o, si lo creemos más cómodo, la instalación del módulo (plug−in) conocido como Java Activator que permite sustituir la máquina virtual propia de nuestro navegador por la JVM 1.1 de Sun Java es un lenguaje relativamente nuevo, pero su uso no requiere que prescindamos de todos los desarrollos heredados para comenzar desde cero. Podemos usar Java RMI para facilitar la creación de una aplicación distribuida al tiempo que aprovechamos desarrollos heredados no escritos en Java. Entre ambos elementos será preciso usar JNI (Java Native Interface), mecanismo que permite la comunicación entre código Java y código escrito en otro lenguaje.

IIOP, *Internet Inter−ORB Protocol*

GIOP (*General Inter−ORB Protocol*) especifica una sintaxis de transferencia estándar (representación de bajo nivel de datos) y un conjunto de formatos de mensajes para comunicaciones entre ORBs. GIOP está construido especialmente para las interacciones entre ORBs y se diseño para trabajar directamente sobre cualquier protocolo de transporte no orientado a conexión. El protocolo es simple, tan simple como es posible, escalable y relativamente fácil de implementar. Está diseñado para permitir implementaciones portables y con unas mínimas dependencias del software bajo la capa de transporte.

El Protocolo Internet Inter−ORB (IIOP) especifica cómo los mensajes GIOP son intercambiados usando conexiones TCP/IP. IIOP especifica un protocolo estandarizado para la interoperabilidad en Internet, permitiendo interoperar con otros ORBs compatibles basados en los productos más populares.

El protocolo está diseñado para que su uso sea indicado y apropiado en cualquier ORB interoperando en dominios IP bajo algún protocolo alternativo si es necesario por un diseño específico de ORB. En este sentido representa el protocolo básico inter−ORB para direcciones TCP/IP, la capa de transporte reinante.

La relación entre IIOP y GIOP es que IIOP es una realización concreta de las definiciones abstractas de GIOP.

RMI – IIOP: Ejemplos

Cómo hacer que los programas RMI usen IIOP

Los pasos siguientes son una breve guía para convertir una aplicación de RMI a RMI−IIOP.

- **Si se está usando el registro de RMI** para nombrar los servicios, se necesita cambiar a JNDI con el plugin CosNaming. Es necesario hacer lo siguiente:
- En su cliente y código del servidor, usted necesita crear un InitialContext para JNDI que usa el código siguiente:

```
import javax.naming. *;

...

El initialNamingContext del contexto = nuevo InitialContext ();
```

- Modificar todos los usos del registro de RMI el lookup () y el lazo () para usar JNDI en cambio. Por ejemplo, usar en lugar de su servidor RMI:

```
import java.rmi. *;

...

Naming.rebind ("MyObject", myObj);
```

usar:

```
import javax.naming. *;

...

initialNamingContext.rebind ("MyObject", myObj);
```

- Si el cliente es un applet, el applet del cliente necesita pasar esto al JNDI plugin CosNaming. Reemplazar el código anterior con lo siguiente:

```
import java.util.*;

import javax.naming.*;

...
```

```
Hashtable env = new Hashtable();

env.put("java.naming.applet", this);

Context ic = new InitialContext(env);
```

- **Si no se está usando el registro de RMI** para servicios de nombres, existe otra manera de arrancar la referencia inicial al objeto remoto. Por ejemplo, el código de servidor puede usar serialización Java para escribir una referencia a un objeto RMI con un ObjectOutputStream y pasando este al código del cliente para deserializar en un stub RMI.
- En el lado del servidor, usar la llamada PortableRemoteObject.toStub() Para obtener un stub, entonces usar writeObject() para serializar este stub a un ObjectOutputStream. El código para hacer esto es similar a lo siguiente:

```
org.omg.CORBA.ORB myORB = org.omg.CORBA.ORB.init(new String[0], null);

Wombat myWombat = new WombatImpl();

javax.rmi.CORBA.Stub myStub = (javax.rmi.CORBA.Stub)PortableRemoteObject.toStub(myWombat;

myStub.connect(myORB);

// myWombat esta ahora conectado con myORB. Para conectar //otros objetos al mismo ORB usar
PortableRemoteObject.connect(nextWombat, myWombat);

FileOutputStream myFile = new FileOutputStream("t.tmp");

ObjectOutputStream myStream = new ObjectOutputStream(myFile);

myStream.writeObject(myStub);
```

En el cliente usar readObject() para deserializar una referencia remota al objeto desde un ObjectInputStream, con un código como:

```
FileInputStream myFile = new FileInputStream("t.tmp");

ObjectInputStream myStream = new ObjectInputStream(myFile);

Wombat myWombat = (Wombat)myStream.readObject();

org.omg.CORBA.ORB myORB = org.omg.CORBA.ORB.init(new String[0], null);

((javax.rmi.CORBA.Stub)myWombat).connect(myORB);

// myWombat esta ahora conectado con myORB. Para conectar //otros objetos al mismo ORB usar
PortableRemoteObject.connect(nextWombat, myWombat);
```

- Cualquier cambio de la implementación de clases remotas esta en javax.rmi.PortableRemoteObject, puede exportarse explícitamente después de la creación mediante una llamada a PortableRemoteObject.exportObject().
- Cambiar todas las localizaciones en su código donde hay una interface remota por javax.rmi.PortableRemoteObject.narrow().

- No depende de la recolección de basura distribuida o usar una facilidad de RMI DGC. Usar para no exportar objetos que no serán usados en mucho tiempo, PortableRemoteObject.unexportObject(). Esto no tiene efectos con objetos exportados con JRMP 1.1.6.
- Regenerar las clases talón y lazo RMI usando el comando rmic con la opción –iiop. Esto producirá archivos de talón y lazo con los siguientes nombres:

  _<implementionName_Tie.class

  _<interfaceName_Stub.class

- Antes de arrancar el servidor, arrancar el proceso de servidor CosNaming usando el siguiente comando:

  tnameserv

Esto usa el puerto por defecto 900.Si desea usar un puerto diferente (por ejemplo el puerto 1050), se usará el modificador ORBInitialPort:

  tnameserv –ORBInitialPort 1050

- Cuando arrancan las aplicaciones cliente y servidor especificar las siguientes propiedades de los sistemas:

  java –Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory

    –Djava.naming.provider.url=iiop://<hostname:900

    <appl_class

Este ejemplo usa el puerto por defecto 900. Si especifica un puerto distinto en el paso 7 se necesita usar el mismo número en el proveedor URL aquí. El <hostname en el proveedor URL es el host name usado para arrancar el servidor en paso 7 CosNaming.

- Si el cliente es un applet, especificar las siguientes propiedades en la etiqueta del applet:

  java.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory

  java.naming.provider.url=iiop://<hostname:900

Este ejemplo usa el puerto por defecto 900. Si especifica un puerto distinto en el paso 7 necesitará usar el mismo número en el proveedor URL aquí. El <hostname en el proveedor URL es el host name usado para arrancar el servidor en paso 7 CosNaming.

Restricciones al ejecutar programas RMI sobre IIOP

Para hacer que programas RMI ya existentes trabajen sobre IIOP, se necesitan seguir las siguientes restricciones.

- Asegúrese que todas las definiciones constantes en las interfaces remotas son de tipos primitivos o Strings y evaluados en tiempo de compilación.
- No use nombres Java en conflicto con IDL se generaran nombres truncados al seguir las reglas de mapeado de Java a IDL. Ver la sección 28.3.2 de *Java Language to IDL Mapping* specification for the Java to IDL name mapping rules.
- No hacer nombres iguales de métodos inherentes a una interface remota.

- Tener cuidado con el uso de nombres que solo se diferencian en el proceso. El uso de nombres de Tipo y variables de ese Tipo que solo se diferencian en el proceso esta soportado. Otras combinaciones de nombres que solo difieren en el proceso no se soportan.
- No depender en tiempo de ejecución compartido de otras referencias a objetos para preservar exactamente cuando transmite referencias a objetos a través de IIOP. El tiempo de ejecución compartido con otros objetos se preserva correctamente.
- No usar las siguientes herramientas:
    - RMISocketFactory
    - UnicastRemoteObject
    - Unreferenced
    - The DGC interfaces

Convertir el programa RMI "Hello World" a RMI–IIOP

Aquí están las partes del RMI "Hello World":

- HelloImpl.java es el servidor RMI.
- Hello.java es la interface remota implementada por HelloImpl.
- HelloApplet.java es el cliente RMI.
- Hello.html carga el HelloApplet.

El ejemplo Hello World RMI usa un directorio de desarrollo como $HOME/jdk1.1/mysrc/example/hello y el directorio de despliegue $HOME/public_html/codebase, donde $HOME es su directorio raíz. El ejemplo asume que usará esos directorios.

Si está listo para ir a través del ejemplo RMI Hello World, Complete los pasos siguientes.

Adaptar la Implementation Class (Servidor) a RMI–IIOP:

- Importar javax.rmi.server.PortableRemoteObject antes que esta javax.rmi.server.UnicastRemoteObject:

```
//Goodbye

//import java.rmi.server.UnicastRemoteObject;

//Hello

import javax.rmi.PortableRemoteObject;
```

- Importar JNDI:

```
import javax.naming.*;
```

- Hacer que HelloImpl amplíe PortableRemoteObject antes de unicastremoteobject:

```
public class HelloImpl

        extends PortableRemoteObject

        ...
```

- Usar el registro JNDI, antes del registro RMI, añadiendo el siguiente código:

```
Context initialNamingContext = new InitialContext();
```

Este paso nos da un contexto inicial de nombres JNDI (también es necesitado por el cliente).

- Usar JNDI rebind(), antes de la versión RMI:

Anterior código:

```
HelloImpl obj = new HelloImpl("HelloServer");
Naming.rebind("HelloServer", obj);
```

Nuevo código:

```
HelloImpl obj = new HelloImpl("HelloServer"); //unchanged
initialNamingContext.rebind("HelloServer",obj);
```

Aquí están los cambios que hay que realizar en helloapplet.java:

- Importar el paquete portableremoteobject:

```
import javax.rmi.PortableRemoteObject;
```

```
<!-- ORIGINAL STEP TWO
<li><b>(Current Release)</b> Create a JNDI (<code>javax.naming</code>)
initial naming context, and start the ORB explicitly:
<pre>
import java.util.*;
import javax.naming.*;
import org.omg.CORBA.ORB;
...
Hashtable env = new Hashtable();
ORB orb = ORB.init(this, null);
env.put("java.naming.corba.orb", orb);
//The next two values will be specifiable
//By applet tag params in future releases
```

```
env.put("java.naming.factory.initial",
"com.sun.jndi.cosnaming.CNCtxFactory");
env.put("java.naming.provider.url", "iiop://&lt;hostname&gt;:900");
Context initialNamingContext = new InitialContext(env);
</pre>
<b>(Future Releases)</b> Create a JNDI initial naming context,
and pass <code>this</code> to the <code>CosNaming</code>
plugin:
<pre>
import java.util.*;
import javax.naming.*;
...
Hashtable env = new Hashtable();
env.put("java.naming.corba.applet", this);
Context ic = new InitialContext(env);
</pre>
--><!-- NEW STEP TWO -->
```

- Crear un contexto inicial de nombres JNDI, y pasárselo al CosNaming plugin:

```
import java.util.*;
import javax.naming.*;
...
Hashtable env = new Hashtable();
env.put("java.naming.corba.applet", this);
//los dos valores siguientes serán especificados
//por la etiqueta de parámetros para futuras versiones
env.put("java.naming.factory.initial",
```

```
        "com.sun.jndi.cosnaming.CNCtxFactory");

    env.put("java.naming.provider.url", "iiop://<hostname:900");

    Context ic = new InitialContext(env);
```

- Usar JNDI lookup(), antes que la versión RMI, y cambiar la interface remota Java por una llamada a javax.rmi.PortableRemoteObject.narrow():

Viejo código:

```
    import java.rmi.*;

    ...

    Hello obj = (Hello)Naming.lookup("//" +

                getCodeBase().getHost() + "/HelloServer");
```

Nuevo código:

```
    import javax.naming.*;

    ...

    Hello obj =

     (Hello)PortableRemoteObject.narrow(

                    initialNamingContext.lookup("HelloServer"),

                    Hello.class);
```

Especificar las propiedades de nombres en la etiqueta del applet

Añadir las siguientes propiedades a Hello.html:

```
<param name="java.naming.factory.initial" value="com.sun.jndi.cosnaming.CNCtxFactory"

<param name="java.naming.provider.url" value="iiop://<hostname:900"
```

Compilar los ficheros fuente Java

```
javac –d $HOME/public_html/codebase Hello.java HelloImpl.java HelloApplet.java
```

Generar las clases de lazo y cola

Asegúrese de que el camino de búsqueda encuentra el comando rmic en el directorio $RMI_IIOP_HOME/bin.

```
rmic –iiop –d $HOME/public_html/codebase examples.hello.HelloImpl
```

Esto generará los archivos Hello_Stub.class (en el lado proxy del cliente) y HelloImpl_Tie.class (el lado

proxy del servidor) en el directorio $HOME/public_html/codebase/examples/hello .

Arrancar el servidor de nombres JNDI

tnameserv

Esto arranca el JNDI name server con el puerto por defecto 900. Si quiere usar un puerto diferente (usuarios de Solaris pueden preferir el 1024), use una linea de comando como:

tnameserv –ORBInitialPort 1050

Arrancar el servidor Hello

```
java –Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory

   –Djava.naming.provider.url=iiop://<hostname:900

   examples.hello.HelloImpl
```

Arrancar el cliente Hello

Use el appletviewer para cargar Hello.html.

appletviewer Hello.html

Crear Hello.html parecido a lo siguiente:

```
<html>

<title>Hello World</title>

<center><h1>Hello World</h1></center>

El mensage del HelloServer es:

<p>

<applet>

     code="examples.hello.HelloApplet"

     width=500 height=120

</applet>

</HTML>
```

Si todo sale conforme al plan, el appletviewer mostrará el mensaje del HelloServer.

Convirtiendo el Applet Cliente a una Aplicación

Aquí está como cambiar el applet cliente a una aplicación:

Adaptar Aplicación Cliente a RMI–IIOP:

- Convertir el HelloApplet en una aplicación:

    - Copiar HelloApplet.java (la versión original RMI) a HelloApp.java.
    - Cambiar el nombre de clase(por ejemplo, a HelloApp).
    - Eliminar la cláusula extends Applet.
    - Cambiar init() por main().
    - Mover el String message ="";.
    - Eliminar el método paint().

- Usar el registro JNDI , antes que el registro RMI:

```
import javax.naming.*;

...

Context initialNamingContext = new InitialContext();
```

- Usar JNDI lookup() , antes que la versión RMI , y cambiar el interface remoto Java por una llamada a javax.rmi.PortableRemoteObject.narrow():

Viejo código:

```
import java.rmi.*;

...

Hello obj = (Hello)Naming.lookup("//" +

                getCodeBase().getHost() + "/HelloServer");
```

Nuevo código:

```
import javax.naming.*;

...

Hello obj = (Hello)PortableRemoteObject.narrow(

                    initialNamingContext.lookup("HelloServer"),

                    Hello.class);
```

El host y el puerto son designados cuando arranca el servidor.

Compilar el fuente HelloApp

```
javac –d $HOME/public_html/codebase HelloApp.java
```

No necesita regenerar el lazo y el talón.

Arrancar el servidor de nombres y el Hello Server

Arrancar del mismo modo que en el ejemplo del applet.

Arrancar la aplicación cliente Hello

Aquí está:

java –Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory

–Djava.naming.provider.url=iiop://<hostname:900

examples.hello.HelloApp

Puede ver el mensaje del servidor en la consola del cliente.

APLICACIONES REALES CON JAVA RMI

Java RMI permite una fácil realización de productos de informática distribuida basados en plataformas Java. Especificamente Java RMI permite a los diseñadores de aplicaciones distribuidas Java tratar los objetos y métodos remotos como si fueran locales. Java RMI trae un nuevo nivel de funcionalidad a las aplicaciones distribuidas, gestión automática de los objetos y paso de objetos de máquina a máquina sobre la red. Java RMI es una parte importante de la plataforma Java y esta siendo enviado como parte del JDK 1.1.

Aquí se muestran experiencias reales que usan Java RMI.

Experiencias con Java RMI:

- Avitek
- CEAS Consulting
- IBM
- Swiss Federal Supreme Court

Avitek

Perfil de la compañía

Avitek es una compañía de rápido crecimiento centrada en la creación de soluciones Java de Boulder, Colorado. Fundamos la compañiía en 1995 y éramos una de las compañías pioneras del mundo en desarrollo real en Java. Desde el principio, el 85% de rentas son resultado del trabajo de desarrollo en Java y esto ha continuado hasta ser nuestro núcleo y área de competencia. Actualmente, Avitek consta de cuatro arquitectos senior cada uno capaz de llevar cualquier solución avanzada basada en Java, así como varios especialistas Java entre el personal.

Nuestro rango de soluciones Java van desde Internet, applets, y sistemas de punto de venta al servicio de intranet y sistemas reservados escritos completamente en Java.. Proyectos actuales incluyen un sistema de ventas on–line con una configuración muy compleja del producto y un sistema de entrenamiento y aprendizaje para soportar el funcionamiento corporativo y ayuda en el trabajo sobre la intranet.

La primera aclamación de la crítica para Avitekt fueron dos premios para nuestro primer applet llamado WebContact. Gamalan y JARS lo colocaron como uno de los mejores applets en la red. Como consecuencia, nosotros hemos desarrollado varios productos a medida y soluciones en Java. Nuestra mayor fuerza esta en

nuestra habilidad en hacer correctamente aplicaciones y reunir requerimientos específicos técnicos y de negocios. Nosotros tenemos una profunda experiencia que incluye:

1. Java front ends para Bases de Datos en el lado del servidor,

2. Soluciones cliente y servidor en JDBC implementadas con Oracle, Sybase, SQL Server, o Access,

3. LDAP basados en preguntas y aplicaciones de administración,

4. Aplicaciones basadas en RMI que soportan integración con Bases de Datos propietarias,

5. Servidores Java usando Java Servlet API y el servidor Netscape Java API, y

6. La organización Avitek desarrolló y construyó todos nuestros applets.

http://www.avitek.com

Aplicación

Nuestro proyecto RMI fue diseñado para construir unos estándares robustos y extensibles basados en aplicaciones software que nuestro cliente pudiera actualizar fácilmente, modificar, y ampliar en una intranet funcionando bajo Windows 95 o NT. La aplicación está siendo desarrollada como una reescritura de un completo sistema reservado de contabilidad con 10 personas/año en su desarrollo inicial. El objetivo de este proyecto era desarrollar y soportar la creación de una prueba completa del concepto de front end Java y un sistema de Bases de Datos. Esto fue logrado con Java en el cliente y en el servidor usando RMI y una API personalizada implementada en una DLL. La primera fase del proyecto se completó satisfactoriamente en Enero y la fase dos está actualmente bajo desarrollo con una fecha de finalización de Junio de 1997.

CEAS Consulting, Workgroup Solutions, Inc. Company

Perfil de la compañía

CEAS Consulting provee re–ingeniería y servicios de automatización de procesos para organizaciones industriales. CEAS Consulting esta especializado en la industria y la metodología de implementación, ha conseguido un refuerzo de la productividad, reduciendo el tiempo de respuesta al mercado, mejorando la calidad del producto, y reduciendo los costos de producción globales.

http://www.ceas.com

Aplicación

CMSRemote es una utilidad de clase que forma la base de applets distribuidos, los cuales operan con el Product Data Management (PDM) producto CMS de Workgroup Technology Corporation (WTC). Nosotros usamos nuestra utilidad de clase CMSRemote para construir aplicaciones de propósito especial basadas en CMS para nuestros clientes. Una descripción de CMS está en http://www.workgroup.com/cms.html.

Aparte de las ventajas obvias del proceso distribuido, esta biblioteca tiene los rasgos siguientes:

Solo una licencia es necesaria para dar servicio a muchos usuarios. Típicamente, es requerida una licencia por usuario, pero la API CMS permite multiples usuarios por licencia.

El tiempo de arranque es cero, siempre esta corriendo una licencia sencilla en el servidor. En otro caso se

tardan cinco segundos aproximadamente.

CMS puede acceder a plataformas no soportadas si la plataforma soporta un navegador Java con capacidades RMI.

CMS puede ser accedido por una computadora de la red. Operaciones de archivo como chequeo, copia, están registradas en el sistema servidor

Desde un applet usando RMI basado en la clase CMSRemote puede escribir y leer archivos para transferirlos desde y al servidor, el applet puede ser cargado desde el sistema local del cliente si son requeridas manipulaciones de archivos locales. Como están disponibles muchas aplicaciones Java, la manipulación y almacenamiento local son menos importantes.

La Figura 1 da un ejemplo de las maneras en las cuales CMS Product Data System puede ser accedido por una variedad de clientes, pequeños y grandes. Los métodos de la API CMS son ejecutados vía llamadas RMI desde los applets que corren en los clientes.

Figura 1. Un ejemplo de una variedad de clientes que sirven datos

La estación de trabajo UNIX y la estación SPARC están ejecutando aplicaciones CAD/CAM que ya existen. Requieren la capacidad de comprobar los dibujos en el proceso de desarrollo, almacenado los archivos en el cliente para procesarlos.

El PC−AT es un viejo modelo de PC ejecutando un navegador Java basado en DOS. Es usado para ver dibujos guardados en CMS, en el suelo de la tienda o en un contador de impresión. El navegador que se ejecuta en este sistema tiene plug−ins compatibles con el visor de dibujos.

La JavaStation es usada para actividades de administración de CMS. Estas actividades tampoco requieren de manipulación de archivos. Los applets de administración son cargados de la red.

IBM

Perfil de la compañía

IBM, la mayor compañía de software del mundo, crea, desarrolla y manufactura productos y servicios avanzados de tecnología de la información, incluyendo ordenadores, software, sistemas de red, dispositivos de almacenamiento y microelectrónica. IBM ha sido un líder en el desarrollo de Internet desde el principio de esta tecnología y está dedicada a ayudar a fabricantes y desarrolladores explotando el potencial de Java. Con los recursos de mas de 20 laboratorios de desarrollo en el mundo, IBM rápidamente monta sus propios productos −− incluyendo sistemas operativos −− así como desarrollando herramientas y tecnologías de Internet para soportar Java.

Desarrolladores y fabricantes pueden encontrar más información acerca de los esfuerzos de IBM en Java en:

http://www.ibm.com/java

Aplicación

El tiempo de desarrollo en aplicaciones de negocios es actualmente lo más importante para las empresas actualmente. La necesidad es responder rápidamente a los cambios en los negocios, agregar componentes y desarrollar nuevas aplicaciones. La compañías y sus departamentos de desarrollo necesitan una manera de acelerar este proceso para aumentar su productividad y ser más competitivos.

El Proyecto San Francisco de IBM muestra las necesidades de una pequeña o mediana empresa y provee soluciones independientes para un más rápido desarrollo de aplicaciones. Esta iniciativa de IBM, anunciada en agosto de 1996, proveerá una serie de aplicaciones de alto nivel escritas en Java. Primero, la capa Base provee un conjunto unificado de servicios para construir con objetos distribuidos Java. Segundo, la capa Common Business Object (Objetos de Negocios Comunes) provee un conjunto de objetos de negocios comunes a una gran variedad de aplicaciones de negocios. Tercero, el marco de aplicación provee funciones especificas del dominio del negocio. El Proyecto San Francisco permite a las sociedades enfocarse menos en la tecnología y más en las soluciones únicas, para desarrollar mas fácilmente aplicaciones de negocios que se adapten a los cambios y puedan interoperar con otras aplicaciones de otros desarrollos.

El Proyecto San Francisco Project usa Remote Method Invocation (RMI) de Java como la base de la infraestructura distribuida. Hemos elegido RMI porque es muy cercana a los requerimientos distribuidos de este proyecto. La distribución trae la necesidad de gestionar y conectar todos los servidores que realizaran el trabajo distribuido. Se han realizado extensiones a RMI las cuales hacen fácil gestionar y usar múltiples direcciones de servidores donde múltiples objetos distribuidos residen. Estas extensiones proveen la forma de conectarse de los clientes con los objetos distribuidos y de arrancar y parar los servidores así como otras funciones básicas de gestión de servidores.

Ejemplos de areas de las extensiones son:

Manejo de excepciones

Gestión de transacciones distribuidas

Gestión de los procesos de servidor

IBM es una marca registrada de International Business Machines Corporation

Swiss Federal Supreme Court

Perfil de la compañía

En la práctica la Corte Federal Suprema decide sobre 5500 casos al año. Todas las decisiones se almacenan en el sistema de bases de datos textual BASISplus. Los 30 jueces tienen acceso a esta bse de datos textual.

De las 250 personas que trabajan en la Corte Federal Suprema, 15 trabajan en el departamento de computación. El equipo de desarrollo de software trabaja para desarrollar las aplicaciones necesarias para los juristas, en particular aplicaciones para gestionar las decisiones que estan tratando. Las nuevas versiones de aplicaciones están basadas en intefaces gráficas. Estan escritas con el lenguaje de programación Ada83. La base de datos BASISplus es usada para gestionar las decisiones de la Corte.

En el futuro estamos pensando en cambiar a una plataforma servidor. Pero necesitamos escribir aplicaciones lo mas independientes posible del hardware. Esta es una de las razxones para usar el lenguaje de programación Java.

http://www.admin.ch/TF/E/

Aplicación

Marco para acceder al sistema de bases de datos textual BASISplus desde aplicaciones Java.

Con el crecimiento de Internet, vino la idea de publicar una parte de nuestras decisiones a través de la red.

Como la seguridad es lo que más nos preocupaba, hicimos un pequeño análisis de las tecnologías existentes en ese momento para acceder a bases de datos a través de clientes WWW. El proyecto arranca a fines de 1996, paralelamente con la expansión del lenguaje Java. En este tiempo, RMI no estaba disponible y se usaba la arquitectura CGI.

Pero al principio del año (1997), como teniamos un poco más de tiempo, echamos un vistazo a la tecnología Java. Entonces descubrimos la API RMI y decidimos evaluar si esta tecnología nos permitía acceder a nuestra base de datos BASISplus desde cualquier cliente.

Usando la API JNI, creamos un driver para acceder a la base de datos BASISplus desde el lenguaje Java. Nosotros llamamos a esta clase OpenAPI la cual es un pegamento que permite acceder al DBMS localmente en el servidor. Entonces creamos, con la API RMI una fábrica de objetos en el lado del servidor. Un cliente, el cual necesita acceder a la base de datos se conecta a la fábrica de objetos, la cual crea un objeto del servidor (cliente del DBMS). Una vez creado, el objeto del servidor devuelve su referencia a la fábrica de objetos, y esta lo envía al cliente. El cliente tiene ahora capacidad para dialogar directamente con la base de datos a traves de un objeto dedicado del servidor. Como BASISplus no tiene seguridad multihilo, cada objeto del servidor se ejecuta en su propio subproceso.

Las principales ventajas son las siguientes:

Cada cliente debe autentificarse en la base de datos. Con otras soluciones, la autentificacion se hace normalmente a través de UID & PASSWORD que están almacenados en el servidor HTTP.

Las transacciones no se restringen a un solo documento HTML.

Verdaderas aplicaciones cliente/servidor que pueden ponerse en Internet.

Hemos creado un pequeño prototipo de una parte de la aplicación. Estamos usando el lenguaje Ada83.
Conclusión: Java es fácil, potente y independiente del hardware!

ANEXO A

OTRAS COSAS QUE DEBEN SABERSE

El servidor necesita Threads seguros

La invocación remota de métodos del mismo objeto remoto necesita seguridad en la implementación de los Threads de ejecución al existir la posibilidad de concurrencia.

Tablas Hash con idénticos vectores de claves

Cuando una tabla hash que contiene idénticos vectores de claves se pasa desde una aplicación RMI en JDK 1.1 a una plataforma RMI basada en una aplicación Java 2 (usando IIOP o JRMP), las claves idénticas se "carbonizan" en una sola clave debido a las reglas de deserialización de plataformas Java 2.

Por ejemplo:

- Una aplicación RMI 1.1crea una tabla hash.
- La aplicación pone un valor en la tabla usando un vector de claves A.
- La aplicación pone otro valor en la tabla usando un vector B. El vector B es estructuralmente idéntico al vector A, pero es un objeto diferente.
- La tabla tiene dos entradas con claves A y B.

- La aplicación RMI 1.1 RMI envía la tabla por valor a una aplicación RMI 1.2 RMI.
- La tabla en el lado 1.2 tiene una entrada B. Esto se debe a que el código de deserialización para tablas usa las reglas 1.2 para cargar la tabla, estas reglas comparan los vectores de clavespor el valor instanciado por la identidad del objeto.
- Si la tabla se envía atrás a traves de RMI desde 1.2 a 1.1 tambien tendrá una entrada y corresponderá al vector B.

Interoperabilidad con otros ORBs

RMI−IIOP puede interoperar con otros ORBS que soporten la especificación CORBA 2.3. No puede interoperar con viejos ORBs, porque no pueden manejar las codificaciones IIOP Para Objetos por Valor. Esto en necesario para enviar clases RMI (incluido strings) sobre IIOP. Ahora solo CORBA 2.3 ORBs están disponibles comercialmente.

Problemas conocidos

- JNDI 1.1 no soporta java.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory como un parametro Applet. Debe ser pasado explicitamente como una propiedad al constructor InitialContext. Esta capacidad es soportada por JNDI 1.2.
- El puerto 900 es reservado en Solaris. Debe usar un puerto mayor que 1024 trabajando en Solaris.
- En Solaris, puede experimentar problemas de tipo "out of file descriptors" ejecutando aplicaciones RMI−IIOP. Esto puede ocurrir si usa demasiados descriptores de archivo. Antes de ejecutar aplicaciones establece el límite de descriptores a un número mayor que 64 que es el valor por defecto. Por ejemplo establezca el número límite a 90 o mayor como sigue:

ulimit −n 90

ANEXO B: Seguridad en RMI

Seguridad en RMI: Características

- RMI tiene una aproximación simple para crear la conexión entre el cliente y el servidor. Los objetos se transmitan a través de la red, no hay encriptación, cualquiera en la red puede leer todos los datos durante la transferencia.
- No hay autentificación: el cliente solamente pide un objeto y el servidor lo entrega. La subsiguiente comunicación, se asume, será desde el mismo cliente. Esto niega una de las ventajas de la seguridad de los objetos distribuidos: la capacidad de ocultar el objeto real siempre y sólo permitir el acceso del cliente, a través de métodos bien definidos. La llave para esto es autentificar los clientes antes de manipular los objetos.
- No hay control de acceso a los objetos.
- Los objetos no son persistentes, las referencias sólo son válidas durante la vida del proceso que creó el objeto remoto.
- Los errores del servidor y de red provocan excepciones, los programas deben estar preparados para manejarlas.

La carga de clases es un mecanismo para proveer clases remotas RMI. Cuando el RMIClassLoader es invocado intenta cargar las clases a través de la red.

Java Security Manager

Puede ser definido un manager de seguridad, en otro caso esto puede causar una excepción. Los programadores pueden escribir su propio Security Manager o usar el restrictivo RMISecurityManager. Esto

desactiva todas las funciones excepto definición de clases y acceso. Si se necesita una política de seguridad diferente es preciso crear su propio Security Manager.

Si el cliente y el servidor están conectados a través de uno o varios firewalls, las llamadas de RMI están embebidas en HTTP, asumiendo razonablemente que a ese HTTP puede permitírsele el paso a través de firewall.

ANEXO C: compiladores "rmic" e "idlj"

EL NUEVO COMPILADOR rmic

El software de RMI–IIOP viene con un nuevo compilador rmic, que puede generar IIOP, y emite IDL.

El nuevo rmic se comporta diferente de las versiones anteriores cuando ningún directorio del rendimiento (la opción –d) se especifica. En el JDK, el talón y archivos del lazo son siempre escritos en el directorio activo actual cuando no es especificada la opción –d, sin tener en cuenta el paquete. El nuevo rmic escribe los archivos en los subdirectorios del directorio actual que corresponden a sus paquetes.

Flags

Flag –iiop

Usando rmic con –iiop esta opción genera talón y lazo las clases. Una clase del talón es una apoderada local para un objeto remoto. Las clases del talón se usan por los clientes para enviar las llamadas a un servidor. Cada interface remota requiere una clase del talón con los instrumentos de la interface remota. La referencia del cliente a un objeto remoto es realmente una referencia a un talón. Se usan las clases del lazo en el lado del servidor para procesar las llamadas entrantes, y despacha las llamadas a la clase de aplicación apropiada. Cada clase de aplicación requiere un lazo la clase.

También se generan las clases del talón para las interfaces abstractas. Una interface abstracta es una interface que no extiende java.rmi.Remote, pero cuyos métodos son de java.rmi.RemoteException o una superclase de java.rmi.RemoteException. Las interfaces que no extienden java.rmi.Remote y no tienen ningún método también son interfaces abstractas.

Flag –idl

Usando rmic con el –idl la opción genera OMG IDL para las clases especificadas y cualquier referencia de las clases.

IDL proporciona un completamente declaratorio, programando el lenguaje los medios independientes para especificar la API para un objeto.

El IDL se usa como una especificación para los métodos y datos que pueden escribirse en y pueden invocarse de cualquier lenguaje que soporta CORBA. Esto incluye Java y C++ entre otros.

Nota: El IDL generado que sólo usa a un compilador de IDL apoyado en CORBA 2.3 con extensiones a IDL puede compilarse.

Flag de –noValueMethods

La opción –noValueMethods, cuando se usó con –idl, asegura que los métodos e initializadores son *no* incluidos en el valuetype s emitido durante la Generación de IDL. Éstos son optativos para el valuetype s y se

omiten por otra parte.

EL NUEVO COMPILADOR idlj

El software de RMI–IIOP incluye a un nuevo compilador de IDL a Java. Este compilador soporta el nuevo CORBA Objects para el que se requiere la interoperacion con RMI–IIOP. Esta escrito en Java, para que pueda correr en cualquier plataforma.

ANEXO D

Lecturas adicionales de interés

Algunos URL´s para aprender más sobre esta tecnología:

- RMI–IIOP home page contiene los enlaces a la documentación de RMI–IIOP, código de ejemplo, especificaciones, noticias, otros URL's relacionados, y más cosas.
- En la página RMI–IIOP FAQ page hay respuestas a muchas preguntas básicas de RMI–IIOP.
- Java RMI contiene los enlaces a la documentación de RMI, ejemplos, la especificación, y más cosas.
- RMI trail en la guía didáctica de Java.
- En la página Java IDL para familiarizarse con la implementación CORBA/IIOP de Sun.
- En Java IDL trail en la guía didáctica de Java.
- Java Language to IDL Mapping documento detallado de la especificación técnica de RMI–IIOP.

CONCLUSIONES

- RMI es potente, simple y fácil de usar, un mecanismo para ampliar Java al trabajo en red. Hay muchos desarrolladores interesados en usar las características únicas de RMI. Desde pequeños applets en Internet hasta grandes aplicaciones en intranets utilizan RMI, un ejemplo es el proyecto de IBM San Francisco.
- Inicialmente, las características de Java RMI fueron diseñadas para trabajar con un protocolo nativo de transporte, JRMP. Se añadieron ciertas características de RMI para poder trabajar con IIOP. Y más adelante se definió un subconjunto restringido de características de Java RMI para trabajar con IIOP. Los desarrolladores que escriban sus aplicaciones con este subconjunto tienen la posibilidad de usar IIOP como su protocolo de transporte.
- RMI sobre JRMP (protocolo nativo de transporte) ofrece ilimitadas posibilidades en la construcción de sistemas en red puramente en Java. RMI fue desarrollado y diseñado para Internet y permitir acceso a múltiples lenguajes a través de IIOP. La ventaja de RMI–IIOP sobre RMI estriba en que el elemento externo que comunica con Java puede estar escrito en cualquier otro lenguaje como C++,Cobol o Delphi.
- El mayor inconveniente de DCOM (Distributed Component Object Model) es, fundamentalmente, su dependencia de Windows (Microsoft), no existiendo demasiadas implementaciones para otros sistemas operativos. RMI en contraposición a DCOM, se trata de una implementación independiente de la plataforma, lo que permite que tanto los objetos remotos como las aplicaciones cliente residan en sistemas heterogéneos. A pesar de todas las ventajas y desventajas que podamos deducir de estos modelos, DCOM, CORBA y RMI, seguramente el punto que inclina la balanza a favor de éste último es su facilidad.
- La RMI de JAVA posee todas las ventajas de CORBA, pero está especialmente adaptada al modelo de objetos JAVA. Esto hace que la RMI de Java sea mucho más eficaz y facil de usar que CORBA en lo que respecta a aplicaciones de Java puro.
- RMI proporciona una implementación de Java puro del modelo de objetos distribuidos que emplea CORBA. Destaca su sencillez, pero está limitada aplicaciones de Java puro. No así CORBA que es neutral y permite comunicar objetos Java con objetos escritos en otros lenguajes de programación.

- Un cliente RMI puede llamar a un objeto remoto en un servidor, y este servidor, a su vez, puede también ser un cliente de otros objetos remotos.
- La invocación remota de métodos del mismo objeto remoto necesita seguridad en la implementación de los Threads de ejecución al existir la posibilidad de concurrencia.
- No hay autentificación: el cliente solamente pide un objeto y el servidor lo entrega. La subsiguiente comunicación, se asume, será desde el mismo cliente. Esto niega una de las ventajas de la seguridad de los objetos distribuidos: la capacidad de ocultar el objeto real siempre y sólo permitir el acceso del cliente, a través de métodos bien definidos. La llave para esto es autentificar los clientes antes de manipular los objetos.
- RMI implementa unos controles de seguridad muy restrictivos en los accesos a los objetos remotos, si se desea usar una política de seguridad diferente, es necesario definir un Security Manager propio. Los controles de seguridad tambien pueden estar embebidos en los cortafuegos.
- Con RMI pueden conseguirse accesos independientemente del SGBD y además en condiciones de seguridad porque la autentificación la realiza la propia base de datos.
- Java RMI no sustituye a Java IDL. Java RMI provee funcionalidad avanzada para programadores Java y acceso a objetos remotos CORBA. Java IDL provee total interoperabilidad con CORBA. Para nuevos proyectos podrán usar Java RMI cuando no necesiten IIOP. Para proyectos existentes que usan IDL, continuaran soportando Java IDL.
- ***Conclusión final:*** Java RMI es un mecanismo que facilita el desarrollo y uso de objetos distribuidos mediante métodos sencillos para cualquier programador acostumbrado a usar Java. Se refleja lo necesario para la creación de cualquier otro desarrollo más complejo. Por ejemplo, el servidor podría usar JDBC para acceder a una base de datos y JNI para comunicarse con partes de la aplicación desarrolladas en otros lenguajes, mientras que el cliente podría obtener sus reglas de funcionamiento en forma de objetos descargables del servidor. En cualquier caso los pasos a seguir serían iguales y usando los mismos recursos.

RMI – IIOP: Preguntas y respuestas

P: ¿Que es Java RMI?

R: Java RMI provee facilidades de computación distribuida basadas en Java . Específicamente, Java RMI permite a los desarrolladores de aplicaciones distribuidas tratar objetos remotos y sus métodos de un modo más normal como los objetos en Java.

Java RMI ofrece un nuevo nivel de funcionalidad a los programas distribuidos con características como distribución, gestión automática de objetos y paso de objetos de máquina a máquina sobre la red. Java RMI es una parte importante de la plataforma Java y se empaqueta como parte del JDK 1.1.

P: ¿Java RMI no es exactamente un protocolo de transporte?

R: No. Java RMI es un conjunto de APIs y un modelo para objetos remotos que permite a los desarrolladores construir fácilmente aplicaciones distribuidas en Java. Por ejemplo, usa interfaces Java normales para definir objetos remotos en un lenguaje separado como IDL. Normalmente Java RMI usa una combinación de serialización Java y el Java Remote Method Protocol (JRMP) para convertir la apariencia normal de un método en una invocación remota de métodos.

Con Java RMI, JRMP continuará soportando y mejorando el protocolo nativo para Java RMI.

P: ¿Será remplazando JRMP con IIOP?

R: No.

P: ¿Qué es un protocolo de transporte?

R: Un protocolo de transporte define un conjunto de formatos de mensaje que permiten pasar datos a través de una red de un ordenador a otro. Java RMI soporta su propio protocolo de transporte (JRMP) y otros protocolos estándar de la industria incluyendo IIOP.

P: ¿En qué aplicaciones puede usarse Java RMI?

R: Los desarrolladores pueden usarlo cuando quieran construir aplicaciones distribuidas basadas en Java. Podrán seleccionar el protocolo de transporte del entorno donde la aplicación será colocada.

P: ¿Qué es IIOP?

R: IIOP es un protocolo de transporte que forma parte de CORBA para computación distribuida.

P: ¿Pueden trabajar juntos Java RMI e IIOP?

R: Sí. Sun ha añadido acceso a la interoperabilidad IIOP a Java RMI.

P: ¿qué pasa si se precisa usar IDL para definir interfaces de objetos remotos?

¿Puede la plataforma Java soportar directamente IDL?

R: Sí. El soporte lo provee a través de Java IDL.

P: ¿Java RMI es el sustituto de Java IDL?

R: No. Ambas tecnologías tienen su uso y lugar. Java RMI provee funcionalidad avanzada para programadores Java y acceso a objetos remotos CORBA. Java IDL provee total interoperabilidad con CORBA. Para nuevos proyectos podrán usar Java RMI cuando no necesiten IIOP. Para proyectos existentes que usan IDL, continuaran soportando Java IDL.

P: ¿Cómo puedo acceder a objetos basados en CORBA a través de Java RMI?

R: Se tiene esa capacidad a través de IIOP.

P: ¿Cómo pueden trabajar juntos Java RMI e IIOP?

R: Inicialmente, las características de Java RMI fueron diseñadas para trabajar con un protocolo nativo de transporte, JRMP. Se añadieron ciertas características de RMI para poder trabajar con IIOP. Y más adelante se definió un subconjunto restringido de características de Java RMI para trabajar con IIOP. Los desarrolladores que escriban sus aplicaciones con este subconjunto tienen la posibilidad de usar IIOP como su protocolo de transporte.

P: Descripción del proceso de cambios en Java RMI

R: Sun inició una revisión de la computación distribuida. Se ha ido al mercado, hablado con desarrolladores, y preguntado a varias licencias de Java importantes sobre tecnologías de computación distribuidas. Siete de las ocho licencias que fueron invitadas fueron atendidas y también hubo una fuerte representación de la comunidad de desarrolladores. Se creó un comité de Java RMI que decidió añadir interoperabilidad con IIOP incluyendo mejoras para IIOP.

P: ¿Tiene futuro Java RMI? , ¿O se impondrán otras tecnologías para aplicaciones distribuidas en red?

R: Nada más lejos de la realidad. JavaSoft ha tenido claro el papel de RMI, RMI seguirá siendo parte de la plataforma Java. RMI es potente, simple y fácil de usar, un mecanismo para ampliar Java al trabajo en red. Hay muchos desarrolladores interesados en usar las características únicas de RMI. Desde pequeños applets en Internet hasta grandes aplicaciones en intranets utilizan RMI, un ejemplo es el proyecto de IBM San Francisco.

P: ¿Cómo puede RMI interoperar con lenguajes distintos de Java?

R: RMI sobre el JRMP ofrece ilimitadas posibilidades en la construcción de sistemas en red puramente en Java. RMI fue desarrollado y diseñado para Internet y permitir acceso a múltiples lenguajes a través de IIOP.

P: ¿Es RMI seguro?

R: RMI sigue el modelo de seguridad de Java el cual incluye nuevas características de seguridad como autentificación, integridad y privacidad en comunicaciones distribuidas.

GLOSARIO

ORB *Object Request Broker*.

Intermediario de Solicitud de Objetos. Conexión de objetos distribuidos.

CORBA *Common Object Request Broker Architecture*

Arquitectura de Intermediación de Solicitud de Objetos Comunes. Arquitectura estándar para el desarrollo de sistemas distribuidos orientados a objetos. Especifica como un objeto cliente escrito en un determinado lenguaje puede invocar a los métodos de un objeto servidor remoto que ha sido desarrollado en otro lenguaje.

IDL *Interface Definition Language*.

Lenguaje de Definición de Interfaces. Herramienta para la definición de interfaces de objetos.

OMG *Object Management Group*. http://www.omg.org

Grupo de Administración de Objetos, consorcio de empresas y otras organizaciones fundado en 1989 y cuyo objeto es promover el desarrollo de software basado en componentes a través del establecimiento de estándares y directrices de software orientado a objetos.

RMI *Remote Method Invocation.*

IIOP *Internet InterORB Protocol.*

RPC *Remote Procedure Call*.

Llamada a Procedimientos Remotos.

JRMP *JAVA Remote Method Protocol*

RMI usa el Protocolo de Método Remoto de Java (JRMP) para la comunicación de los objetos remotos de Java.

BIBLIOGRAFÍA

LIBROS

- **JAVA 1.2 AL DESCUBIERTO**; Jaworski, Jamie; Ed. Prentice Hall; Madrid, 1999; ISBN 84–8322–061–X
- "**JAVA Network Security**"; Macgregor, R. | Durbin, D. | Owlett, J. | Yeomans, A.; Prentice Hall; 1998; ISBN 0–13–761529–9
- "**Programación en JAVA**"; Cuenca Jiménez, Pedro; Anaya; 1997; ISBN 84–415–0174–2
- **PCWorld**; Revista; Ed. IDG Communications; Números 147, 148, 155 del año 1998; ISBN 0213–1307

INTERNET

- http://java.sun.com/a–z/
- http://www.programacion.net
- http://dir.yahoo.com/Computers_and_Internet/Programming_Languages/Java/
- http://www.iti.upv.es:81/java/
- http://www.idg.es/pcworld
- http://www.omg.org
- http://www.idg.es/iworld
- http://www.gamelan.com
- http://www.javaworld.com
- http://www.geocities.com/SiliconValley/Lakes/2227/
- http://www.ibm.com/developer/java/
- http://forum.java.sun.com

- RMI–IIOP home page
- RMI–IIOP FAQ page
- Java RMI
- RMI trail
- Java IDL trail
- Java Language to IDL Mapping

CRÉDITOS

UNIVERSIDADE DE VIGO

ESCOLA SUPERIOR DE ENXEÑERÍA INFORMÁTICA – OURENSE

PROA, Programación Avanzada 1999–2000

"Para saber bien las cosas, no basta con haberlas aprendido".

Sócrates

PROFESOR:

Dr. JUAN MANUEL CORCHADO RODRÍGUEZ

AUTORES:

CUEVAS DE LA FUENTE, BENIGNO

benigno.cuevas@canal21.com

ENRÍQUEZ TRIGÁS, JOSÉ MANUEL

jmtrigas@canal21.com

QUINTAS RODRÍGUEZ, MARCO ANTONIO

marcoqr@teleline.es

TOUCEDA FUNGUEIRO, JUAN MANUEL

jtouceda@canal21.com

ÍNDICE

INTRODUCCIÓN

JAVA: RMI–IIOP

Nivel de interfaz de usuario

Nivel de procesamiento de información

Nivel de almacenamiento de información

Organización de sistemas distribuidos

Objeto cliente

Fragmento

Objeto servidor

Esqueleto

Protocolos bajo Nivel

Protocolos bajo Nivel

Comunicación Virtual

Comunicación Activa

Objeto Cliente

Objeto Servidor

Fragmento

Esqueleto

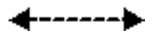
Nivel de Referencia remota

Nivel de Referencia remota

Nivel de transporte

Nivel de transporte

Protocolos de bajo nivel Protocolos de bajo Nivel



Comunicación Virtual

Comunicación activa

HOST B

HOST A

HOST C

HOST D

Objeto

Invocación Remota

Aplicación

financiera

Objeto de normas comerciales

Objeto de búsqueda

Objeto de distribución de información

Nivel Cliente Nivel Funcional Nivel de Almacenamiento

Bases de Datos

Host del cliente

Objeto cliente

Fragmento IDL

ORB

Host del Servidor

Objeto servidor

Esqueleto IDL

HOST LOCAL

Objeto local

HOST REMOTO

Registro Remoto

Objeto Remoto

Regístreme como José

Aquí esta José

Permítame

Acceder a