¿Que es RPC?

- Invocar un método de manera remota (en espacios de direcciones diferentes)

- Es el mecanismo para procesar de manera remota algún segmento de código (utilizando una CPU, Memoria, Stock, etc., remotos),

necesario localmente, sin salir del procesamiento local.

* ASPECTOS A TENER EN CUENTA -

Hay que tener en cuenta una serie de detalles y problemas, como por ejemplo:

- Marshalling y Unmarshalling (Encapsulado): Estandarizar el orden de los datos / Codificación de los parámetros. JAVA usa SERIALIZACION.

- Inestabilidad de la Red: Como retornar a la programación

- Gestión automática de bloqueos por si hay varias llamadas a un mismo proceso y esta ocupado atendiendo anteriores

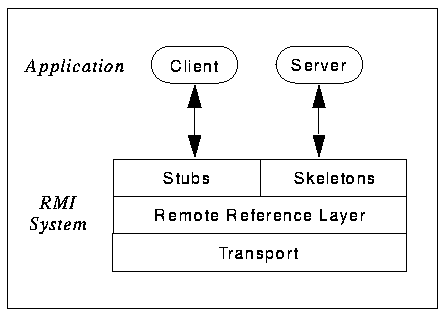
- Seguridad: No cualquiera pueda interrumpir mi flujo de procesamiento.

- Localización de objetos remotos (Rmiregistry)

- Representación de Datos: Por Ejemplo, UTF8 v/s ASCII

- Distribución de Hardware: Distinta zona de memoria, stack, clock de la CPU, etc.

- ESQUEMA RMI –



- Server -

+ Crea objetos remotos

+ Hace accesible referencias a objetos remotos

+ Ejecuta las tareas sobre los métodos definidos

+ Espera que los clientes invoquen los métodos

Los nombres no son transparentes a la ubicación:

- Cliente -

+ Obtiene una referencia a 1 o + objetos remotos

+ Invoca sus métodos.

- STUB/SKELETON -

Es la interfaz entre el nivel de aplicación y el resto del sistema. Es un objeto que encapsula el método que deseamos invocar remotamente. ES UN PROXY.

Este prepara información con la identificación del objeto remoto a invocar, el método a invocar y codificación de los parámetros (Marshalling).

- REFERENCIA REMOTA -  
La capa de referencia remota es responsable de la semántica de invocación. Los niveles superiores (Aplicacion y Stub/Skeleton) no ven estas diferencias. Diferentes protocolos de invocación pueden implementarse en este nivel.

Ejemplo

* Invocación Unicast Punto a Punto.
* Invocación a grupos de objetos replicados.
* Soportar una estrategia de replicación especifica.
* Soportar una referencia persistente al objeto remoto.
* Estrategias de reconexión.

***TRANSPORTE***.   
Es responsable de :

* Establecer conexiones a JVM remotas.
* Administrar las conexiones.
* Escuchar las llamadas entrantes.
* Mantener una tabla de objetos remotos que residen en la JVM.
* Establecer una conexión para una llamada entrante.
* Localizar el despachador del objetivo de la llamada remota y pasar la conexión a este despachador.

Clases y métodos implicados en RMI JAVA

\*\* InterfaceRemota.

Una interfaz en Java es una coleccion de metodos abstractos (nombre y signatura) y propiedades constantes.

En las interfaces se especifica QUE se debe hacer pero no su implementacion (COMO).

Seran las clases que implementen estas interfaces las que describan la logica del comportamiento de los metodos.

En este caso extends Remote. -> Es una interface java con todos los metodos que queramos poder invocar de

forma remota, es decir, los metodos que queremos llamar desde el cliente, pero que se ejecutaran

en el servidor. Todos los parametros y valores devueltos de estos metodos deben ser tipos primitivos

de java o bien clases que implementen la interface Serializable de java

El interfaz debe ser público.

\*\* ObjetoRemoto. Es una clase con la implementacion de los metodos de InterfaceRemota.

A esta clase solo la ve el servidor de rmi.

\*\* Servidor:

un programa java que instancie y registre el objeto remoto.

- Debe instanciar el ObjetoRemoto y registrarlo en el rmiregistry.

Una vez registrado el ObjetoRemoto, el servidor no muere, sino que queda vivo.

Cuando un cliente llame a un metodo de ObjetoRemoto, el codigo de ese metodo se ejecutara en este proceso.

- rmiregistry. Este es un programa (Herramienta) que nos proporciona java

Registra un objeto para que puedan ser invocados remotamente y admite peticiones de clientes

para ejecutar metodos de estos objetos.

\*\* Cliente.

Pide al rmiregistry (CLiente) del pc/maquina servidor una referencia remota al ObjetoRemoto.

Una vez que la consigue, puede hacer las llamadas a sus metodos.

Los metodos se ejecutaren en el Servidor, pero Cliente quedara bloqueado en cada llamada hasta

que Servidor termine de ejecutar el metodo.

* Pasos de Ejecución

0) Servidor instancia ObjetoRemoto

1) Servidor crea registroRMI

2) RegistroRMI registra el par: Objeto / Servidor

3) Servidor exporta un objeto (Accesible por la red) a la espera de peticiones TCP

4) El cliente obtiene el RMIRegistry

5) Objeto cliente hace un llamado remoto

6) el STUB del cliente codifica los parámetros

7) el STUB encapsula el método que se va a invocar (llamado igual a local). Prepara id. de obj. remoto, el método y los parámetros.

8) STUB envía a Remote Reference Layer el cual Inicia la conexión con MV remota y envía los datos.

9) el STUB del server (SKELETON) decodifica los parámetros, ubica el objeto y lo llama.

10) Ejecuta en el server el método con los parámetros

11) el STUB SKELETON codifica el resultado y envía a STUB Cliente

12) STUB Cliente pasa a código de programa cliente

13) El código cliente continua como si hubiera sido ejecución local

RPC VS SOCKETS

- Los sockets necesitan mas codigo para identificar que mensaje llega y tratarlo en consecuencia (sentencias switch-case o

tantos if como tipos de mensaje distintos).

- En rmi se hace con llamadas a metodos, por lo que no es necesario ningun tipo de codificacion adicional.

- En rmi se necesitan una serie de herramientas adicionales, compilar los objetos remotos ademas de con javac con rmic y

finalmente tener lanzado el programa rmiregistry.

AMBOS BLOQUEAN. En ambos casos suele ser necesario el uso de hilos si las respuestas pueden tardar.

- En los sockets se envia un mensaje y se debe esperar por la respuesta, que puede llegar o no. Las lecturas de mensajes dejan bloqueado

el hilo hasta que llegan.

- En rmi la llamada queda bloqueada hasta recibir la respuesta.

BANDWITH:

- En sockets, por la red solo se envian los mensajes que nosotros enviamos.

- En rmi, se envia ademas todo el protocolo interno de rmi, que suele ser de un tamanio considerable.

Rmi consume mas ancho de banda de nuestro enlace fisico que los sockets.

Comunicacion con otros lenguajes:

- Rmi es puramente java, asi que solo puede integrarse entre dos programas java.

- Con sockets podemos comunicarnos con cualquier otro programa hecho en cualquier otro lenguaje,

aunque tendremos que enviar los mensajes campo a campo para hacerlos compatibles con el otro lenguaje (NO SERIALIZABLE).

Mensajes en broadcast.

- Los sockets permiten enviar mensajes sin destinatario, de forma que cualquiera puede recogerlo.

- Con rmi la comunicacion siempre es punto a punto.

Cuando usar sockets y cuando usar rmi

SOCKETS:

- Si nuestro enlace fisico es lento (puerto RS232, modem, etc) es mejor usar sockets.

- Si tenemos pocos tipos de mensajes y se usan para transmitir muchos datos (por ejemplo, transferencia de ficheros), es mejor sockets.

RMI:

- Si el servidor ofrece muchas funcionalidades, es mejor rmi.

- Si la aplicacion servidor va a modificarse con cierta frecuencia y no queremos tener que actualizar

todos los clientes uno a uno, la carga dinamica de clases de rmi puede ser una solucion

SOCKETS

-> Desarrollar sus propios mecanismos para manejar de forma eficiente los mensajes intercambiados

-> uno debe preocuparse de como se transmiten los datos entre los extermos de la conexion

RMI

- Mayor grado de abstraccion

- Olvidarse de los datos de transmision y centrarse en el dise�o de la aplicacion