Computación Distribuida

Tema 3: Objetos Distribuidos

Paso de mensajes frente a Objetos Distribuidos

Paso de Mensajes frente a Objetos Distribuidos

- El paradigma del paso de mensajes es un modelo natural para la computación distribuida, en el sentido de que simula la comunicación entre humanos. Es un paradigma apropiado para servicios de red donde los procesos interactúan entre ellos a través del intercambio de mensajes.
- Sin embargo, la abstracción que proporciona este paradigma no cumple con las necesidades de aplicaciones de red complejas.

Paso de Mensajes frente a Objetos Distribuidos –2

- **El paso de mensajes necesita que los procesos participantes se encuentren fuertemente acoplados: a través de esta interacción, los procesos deben comunicarse directamente entre ellos. Si la comunicación se pierde entre los procesos (debido a fallos en el enlace e comunicación, en el sistema o en uno de los procesos) la colaboración falla.**
- **El paradigma del paso de mensajes está orientado a datos. Cada mensaje contiene datos con un formato mutuamente acordado y se interpreta como una petición o respuesta de acuerdo al protocolo. La recepción de cada mensaje desencadena una acción en el proceso receptor. No es adecuado para aplicaciones complejas que impliquen un gran número de peticiones y respuestas. En dichas aplicaciones, la interpretación de los mensajes se puede convertir en una tarea inabordable.**

El paradigma de objetos distribuidos

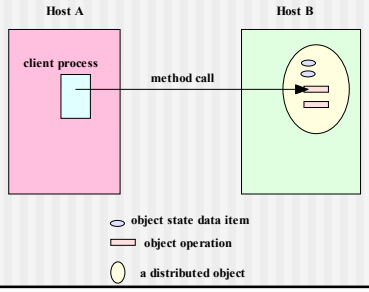
- El paradigma de objetos distribuidos es un paradigma que proporciona mayor abstracción que el modelo de paso de mensajes. Como su nombre indica, este paradigma está basado en objetos existentes en un sistema distribuido.
- En programación orientada a objetos, basada en un lenguaje de programación orientado a objetos, tal como Java, los objetos se utilizan para representar entidades significativas para la aplicación. Cada objeto encapsula:
  - el **estado** o datos de la entidad: en Java, dichos datos se encuentran en las variables de instancia de cada objeto;
  - las **operaciones** de la entidad, a través de las cuales se puede acceder o modificar el estado de la entidad

Objetos Locales frente a Objetos Distribuidos

- Los objetos locales son objetos cuyos métodos sólo se pueden invocar por un **proceso local**, es decir, un proceso que se ejecuta en el mismo computador del objeto.
- Un objeto distribuido es aquel cuyos métodos pueden invocarse por un **proceso remoto**, es decir, un proceso que se ejecuta en un computador conectado a través de una red al computador en el cual se encuentra el objeto.

El paradigma de Objetos Distribuidos

En un paradigma de objetos distribuidos, los recursos de la red se representan como objetos distribuidos. Para solicitar un servicio de un recurso de red, un proceso invoca uno de sus métodos u operaciones, pasándole los datos como parámetros al método. El método se ejecuta en la máquina remota y la respuesta es enviada al proceso solicitante como un valor de salida.

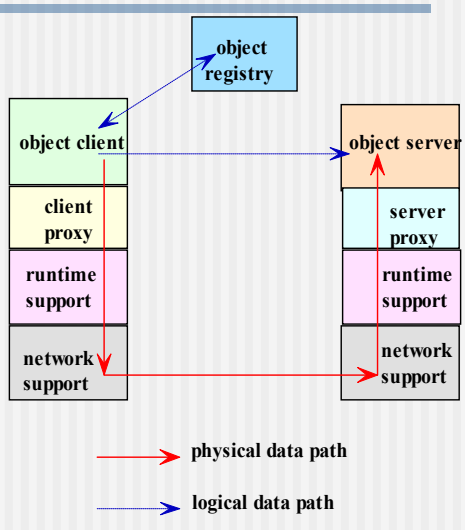


Paso de mensajes frente a Objetos Distribuidos - 3

Comparado con el paradigma de paso de mensajes, que es orientado a datos, el paradigma de objetos distribuidos es **orientado a acciones:** hace hincapié en la invocación de las operaciones, mientras que los datos toman un papel secundario. Aunque es menos intuitivo para los seres humanos, el paradigma de objetos distribuidos es más natural para el desarrollo de software orientado a objetos.

El paradigma de Objetos Distribuidos

Una arquitectura típica de objetos distribuidos



Sistemas de Objetos Distribuidos

- Al objeto distribuido proporcionado o exportado por un proceso se le denomina objeto servidor. Otra utilidad, denominada registro de objetos debe existir en la arquitectura para registrar objetos distribuidos.
- Para acceder a un objeto distribuido, un proceso –el **objeto cliente** – busca en el registro de objetos para encontrar una **referencia**[1] al objeto. El objeto cliente utiliza esta referencia para realizar llamadas a los métodos del objeto remoto.

[1] Una referencia es un "manejador" de un objeto; es la representación a través de la cual se puede localizar dicho objeto en el computador donde reside.

Sistemas de Objetos Distribuidos - 2

- A nivel lógico, el objeto cliente realiza una llamada directamente al método remoto.
- Realmente, un componente software se encarga de gestionar esta llamada. Este componente se denomina ***proxy de cliente*** y se encarga de interactuar con el software en la máquina cliente con el fin de proporcionar soporte en tiempo de ejecución para el sistema de objetos distribuidos.
- El soporte en tiempo de ejecución es responsable de la comunicación entre procesos necesaria para transmitir la llamada a la máquina remota, incluyendo el empaquetamiento de los argumentos que se van a transmitir al objeto remoto.

Sistemas de Objetos Distribuidos - 3

- Una arquitectura similar es necesaria en la parte del servidor, donde el soporte en tiempo de ejecución para el sistema de objetos distribuidos gestiona la recepción de los mensajes y el desempaquetado de los datos, enviando la llamada a un componente software denominado **proxy de servidor**.
- El proxy de servidor invoca la llamada al método local en el objeto distribuido, pasándole los datos desempaquetados como argumentos.
- La llamada al método activa la realización de determinadas tareas en el servidor. El resultado de la ejecución del método es empaquetado y enviado por el proxy de servidor al proxy de cliente, a través del **soporte en tiempo de ejecución** y del **soporte de red** de ambas partes de la arquitectura.

Sistemas/Protocolos de Objetos Distribuidos

El paradigma de objetos distribuidos se ha adoptado de forma extendida en las aplicaciones distribuidas, para las cuales existe un gran número de herramientas disponibles basadas en este paradigma. Entre las herramientas más conocidas se encuentran:

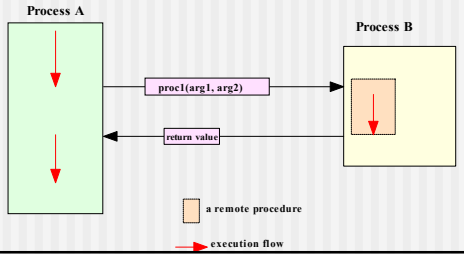
- ~ Java Remote Method Invocation (RMI),
- ~ los sistemas Common Object Request Broker Architecture (CORBA),
- ~ los Distributed Component Object Model (DCOM),
- ~ herramientas y APIs para el Simple Object Access Protocol (SOAP).

De todas estas herramientas, la más sencilla es Java RMI

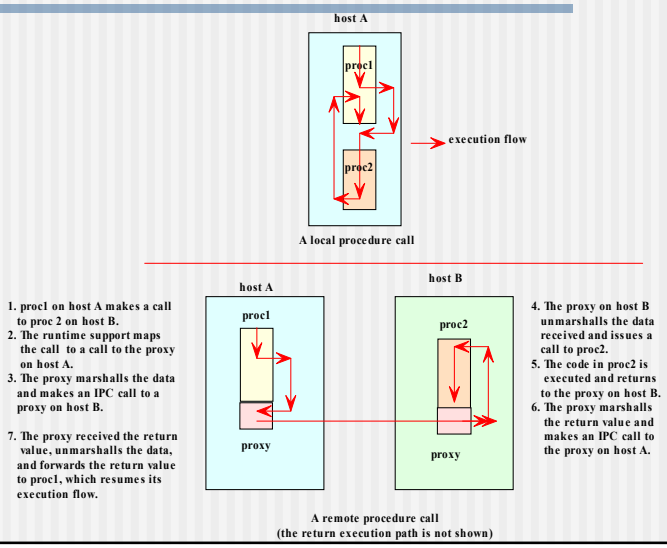
Remote Procedure Call frente a Remote Method Invocation

Remote Procedure Calls (RPC)

- Remote Method Invocation (RMI) tiene su origen en el paradigma denominado Llamada a Procedimientos Remotos
- En el modelo de la llamada a procedimientos remotos, un proceso realiza una llamada a procedimiento de otro proceso, pasándole los datos a través de argumentos. Cuando un proceso recibe una llamada, se ejecuta la acción codificada en el procedimiento. A continuación, se notifica la finalización de la llamada al proceso que invoca la llamada y, si existe un valor de retorno, se le envía a este último proceso desde el proceso invocado.



Llamada Local a Procedimiento y Llamada Remota a Procedimiento



Remote Procedure Calls (RPC) - 2

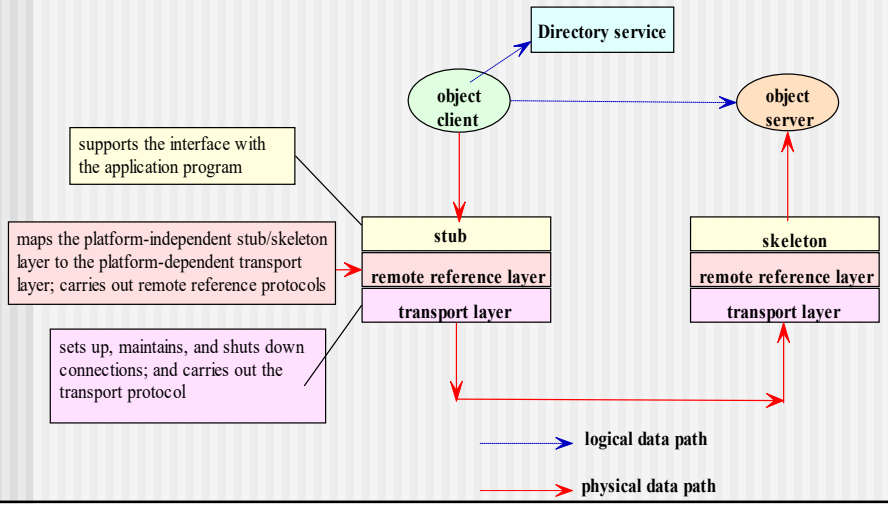
- Desde su introducción a principios de los años 80, el modelo RPC se ha utilizado ampliamente en las aplicaciones de red.
- Existen dos APIs que prevalecen en este paradigma.
  - El API *Open Network Computing Remote Procedure Call* es una evolución del API de RPC que desarrolló orignalmente Sun Microsystems a principios de los años 80.
  - La otra API popular es *Open Group Distributed Computing Environment* (DCE) RPC.
- Ambas interfaces proporcionan una herramienta, *rpcgen*, para transformar las llamadas a procedimientos remotos en llamadas a procedimientos locales (stub).

Java Remote Method Invocation

Remote Method Invocation

- Remote Method Invocation (RMI) es una implementación orientada a objetos del modelo RPC. Se trata de una API exclusiva para programas Java.
- En RMI, un *servidor de objetos* exporta un *objeto remoto* y lo registra en un servicio de directorio. El objeto proporciona métodos remotos, que pueden invocar los programas cliente.
- Sintácticamente:
  - Un objeto remoto se declara como una *interfaz remota.*
  - El objeto servidor implementa la interfaz remota.
  - Un *objeto cliente* accede al objeto mediante la invocación remota de sus métodos utilizando una sintaxis similar a las invocaciones de los métodos locales.

La arquitectura Java RMI



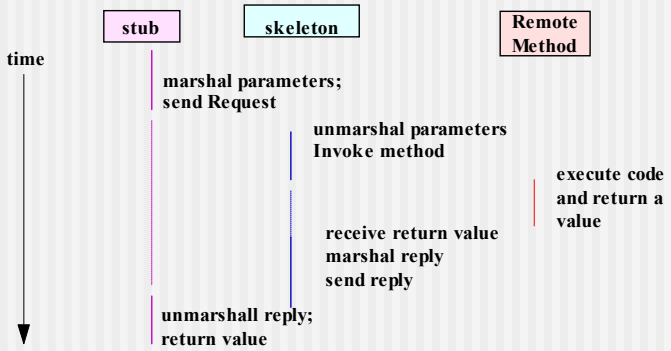
Registro de los objetos

- El API de RMI hace posible el uso de diferentes servicios de directorios para registrar un objeto distribuido[1].
- Usaremos un servicio de directorio llamado registro RMI, ***rmiregistry***, proporcionado por el Java Software Development Kit (SDK). El registro RMI es un servicio cuyo servidor, cuando está activo, se ejecuta en la **máquina del servidor del objetos.** Por convención, utiliza el puerto TCP 1099 por defecto.

[1] Uno de estos servicios de directorios es la **interfaz de nombrado y directorios de Java** (JNDI), que es más general que el registro RMI, en el sentido de que lo pueden utilizar aplicaciones que no usan el API RMI.

Interacción entre el stub y el skeleton

Un diagrama de eventos temporal describiendo la interacción entre el stub y el skeleton:



(based on http://java.sun.com.marketing/collateral/javarim.html)

El API de Java RMI

- Interfaz Remota
- El software en el servidor
  - Implementación de la Interfaz Remota
  - Generación del Stub y Skeleton
  - El Servidor de Objetos
- El software en el cliente

La interfaz remota

- Una interfaz Java es una clase que se utiliza como plantilla para otras clases: contiene las declaraciones de los métodos (denominados métodos abstractos) que deben implementar las clases que utilizan dicha interfaz.
- Una interfaz remota Java es una interfaz que hereda de la clase Java ***Remote,*** que permite implementar la interfaz utilizando sintaxis RMI. Aparte de la extensión que se hace de la clase Remote y de que todas las declaraciones de los métodos deben especificar la excepción RemoteException, una interfaz remota utiliza la misma sintaxis que una interfaz Java local.

Un ejemplo de interfaz remota Java

```
// file: SomeInterface.java
// to be implemented by a Java RMI server class.
import java.rmi.*
public interface SomeInterface extends Remote {
   // signature of first remote method
    public String someMethod1( )
         throws java.rmi.RemoteException;
   // signature of second remote method
    public int someMethod2( float ) throws
                    java.rmi.RemoteException;
   // signature of other remote methods may follow

} // end interface
```

Un ejemplo de interfaz remota Java - 2

- Cada declaración de un método debe especificar la excepción **java.rmi.Remote** en la sentencia *throws*.
- Cuando ocurre un error durante el procesamiento de la invocación del método remoto, se lanza una excepción de este tipo, que debe ser gestionada en el programa del método que lo invoca.
- Las causas que origina este tipo de excepción incluyen los errores que pueden ocurrir durante la comunicación entre los procesos, tal como fallos de acceso y conexión, así como problemas asociados exclusivamente a la invocación de métodos remotos, como por ejemplo no encontrar el objeto, el stub o el skeleton.

Software de la parte servidora

Un objeto servidor es un objeto que proporciona los métodos y la interfaz de un objeto distribuido. Cada objeto servidor debe

- implementar cada uno de los métodos remotos especificados en la interfaz,
- registrar en un servicio de directorio un objeto que contiene la implementación.

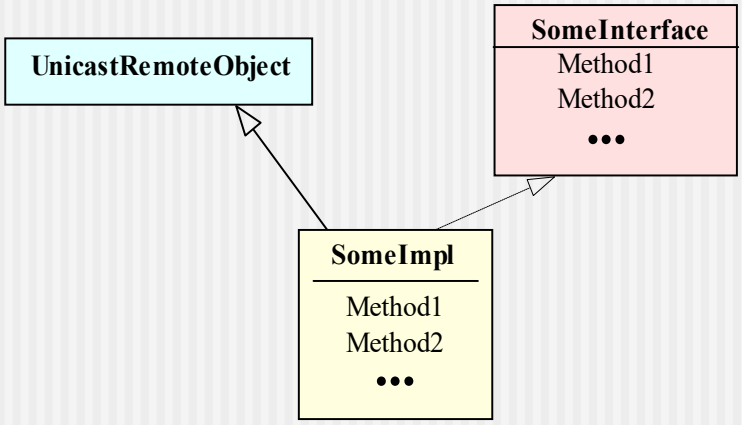
Se recomienda que las dos partes se realicen en clases separadas (la segunda parte la realizaría el servidor de objetos).

La implementación de la interfaz remota

Se debe crear una clase que implemente la interfaz remota. La sintaxis es similar a una clase que implementa la interfaz local.

```
import java.rmi.*;
import java.rmi.server.*;
/**
 * This class implements the remote interface SomeInterface.
 */
public class SomeImpl extends UnicastRemoteObject
     implements SomeInterface {
   public SomeImpl() throws RemoteException {
      super( );
   }
   public String someMethod1( ) throws RemoteException {
      // code to be supplied
   }
   public int someMethod2( ) throws RemoteException {
      // code to be supplied
   }
} // end class
```

Diagrama UML para la clase SomeImpl



UMLDiagram for SomeImpl

Generación del Stub y el Skeleton

En RMI, un objeto distribuido requiere un proxy por cada uno de los servidores y clientes de objeto, conocidos como skeleton y stub respectivamente. Estos proxies se generan a partir de la implementación de una interfaz remota utilizando una herramienta del SDK de Java: el compilador RMI *rmic*.

`rmic <nombre de la clase de la implementación de la interfaz remota>`

Por ejemplo:

```
rmic SomeImpl
```

Como resultado de la compilación se generan dos ficheros proxy, cada uno de ellos con el prefijo correspondiente al nombre de la clase de la implementación:

SomeImpl_skel.class
SomeImpl_stub.class.

El fichero de stub para el objeto

- El fichero de stub para el objeto, así como el fichero de la interfaz remota deben compartirse con cada cliente de objeto – estos ficheros son imprescindibles para que el programa cliente pueda compilar correctamente.
- Una copia de cada fichero debe colocarse manualmente en la parte del cliente. Adicionalmente, Java RMI dispone de una característica denominada "stub downloading" que consiste en que el cliente obtiene de forma dinámica el stub.

El servidor de objetos

La clase del servidor de objetos instancia y exporta un objeto que implementa la interfaz remota. El siguiente código muestra una plantilla para la clase del servidor de objetos.

```
import java.rmi.*;
……
public class SomeServer  {
   public static void main(String args[]) {
      try{
       // code for port number value to be supplied
          SomeImpl exportedObj = new SomeImpl();
          startRegistry(RMIPortNum);
          // register the object under the name "some"
          registryURL = "rmi://localhost:" + portNum + "/some";
          Naming.rebind(registryURL, exportedObj);
          System.out.println("Some Server ready.");
       }// end try
} // end main
```

El servidor de objetos - 2

```
// This method starts a RMI registry on the local host, if it
// does not already exists at the specified port number.
private static void startRegistry(int RMIPortNum)
      throws RemoteException{
  try {
    Registry registry= LocateRegistry.getRegistry(RMIPortNum);
    registry.list( );
          // The above call will throw an exception
          // if the registry does not already exist
  }
  catch (RemoteException ex) {
    // No valid registry at that port.
    System.out.println(
        "RMI registry cannot be located at port " + RMIPortNum);
    Registry registry= LocateRegistry.createRegistry(RMIPortNum);
    System.out.println(
       "RMI registry created at port " + RMIPortNum);
  }
} // end startRegistry
```

El servidor de objetos - 3

- En la plantilla de nuestro servidor de objetos, el código para exportar un objeto se realiza del siguiente modo:

```
// register the object under the name "some"
registryURL = "rmi://localhost:" + portNum +
              "/some";
Naming.rebind(registryURL, exportedObj);
```

- La clase ***Naming*** proporciona métodos para almacenar y obtener referencias del registro. En particular, el método rebind permite almacenar en el registro una referencia a un objeto con una URL de la forma

```
rmi://<host name>:<port number>/<reference name>
```

- El método ***rebind*** sobreescribe cualquier referencia en el registro asociada al nombre de la referencia. Si no se desea sobreescribir, existe un método denominado ***bind***.
- El nombre de la máquina debe corresponder con el nombre del servidor, o simplemente se puede utilizar "localhost". El nombre de la referencia es un nombre elegido por el programador y debe ser único en el registro.

El registro RMI

- **Un servidor exporta un objeto registrándolo con un nombre simbólico en un servidor denominado registro RMI.**

```
// Create an object of the Interface
 SomeInterface obj = new SomeInterface("Server1");
// Register the object; rebind will overwirte existing
//  registration by same name – bind( ) will not.
Naming.rebind("Server1", obj);
```

- **Se necesita un servidor, llamado registro RMI ejecutándose en la máquina del servidor que exporta el objeto remoto.**
- **El RMIRegistry es un servidor localizado en el puerto 1099 por defecto**
- **Puede ser invocado dinámicamente por la clase del servidor:**

```
import java.rmi.registry.LocateRegistry;
…
LocateRegistry.createRegistry ( 1099 );
…
```

El registro RMI - 2

- Alternativamente, se puede activar un registro RMI manualmente utilizando la utilidad *rmiregistry*, que se encuentra en el SDK, a través de la ejecución del siguiente mandato en el intérprete de comandos:

  rmiregistry <número puerto>

  donde el número de puerto es un número de puerto TCP. Si no se especifica ningún puerto, se utiliza el puerto por defecto 1099.
- El registro se ejecutará de forma continua hasta que se solicite su terminación (a través de CTRL-C, por ejemplo)

El servidor de objetos - 4

- Cuando se ejecuta un servidor de objetos, la exportación de los objetos distribuidos provoca que el proceso servidor comience a escuchar por el puerto y espere a que los clientes se conecten y soliciten el servicio del objeto.
- Un objeto servidor RMI es un servidor concurrente: cada solicitud de un objeto cliente se procesa a través de un hilo independiente del servidor. Dado que las invocaciones de los métodos remotos se pueden ejecutar de forma concurrente, es importante que la implementación de un objeto remoto sea *thread-safe.*

El software de la parte cliente

La clase cliente es como cualquier otra clase Java.

La sintaxis necesaria para hacer uso de RMI supone

- localizar el registro RMI en el nodo servidor,

  y

- buscar la referencia remota para el servidor de objeto; a continuación se realizará un *cast* de la referencia a la clase de la interfaz remota y se invocarán los métodos remotos.

El software de la parte cliente - 2

```
import java.rmi.*;
….
public class SomeClient {
   public static void main(String args[]) {
     try {
        String registryURL =
         "rmi://localhost:" + portNum + "/some";
        SomeInterface h =
         (SomeInterface)Naming.lookup(registryURL);
        // invoke the remote method(s)
        String message = h.method1();
        System.out.println(message);
        // method2 can be invoked similarly
     } // end try
     catch (Exception e) {
         System.out.println("Exception in SomeClient: " + e);
     }
   } //end main
   // Definition for other methods of the class, if any.
}//end class
```

Búsqueda del objeto remoto

El método *lookup* de la clase *Naming* se utiliza para obtener la referencia del objeto, si existe, que previamente ha almacenado en el registro el servidor de objetos. Obsérvese que se debe hacer un *cast* de la referencia obtenida a la clase de la interfaz remota (no a su implementación).

```
String registryURL =
   "rmi://localhost:" + portNum + "/some";
SomeInterface h =
  (SomeInterface)Naming.lookup(registryURL);
```

Invocación del método remoto

Se utiliza la referencia a la interfaz remota para invocar cualquiera de los métodos de dicha interfaz, como en el ejemplo:

```
String message = h.method1();
System.out.println(message);
```

- Obsérvese que la sintaxis utilizada para la invocación de los métodos remotos es igual que la utilizada para invocar métodos locales.
- Es un error común el hacer un cast del objeto obtenido del registro a la clase que implementa la interfaz o a la clase del servidor de objetos. El cast debe realizarse a la clase de la interfaz.

Pasos para construir una aplicación RMI

Algoritmo para desarrollar el software en la parte del servidor

1. Crear un directorio donde se almacenen todos los ficheros generados por la aplicación.
2. Especificar la interfaz remota del servidor en ***SomeInterface.java***. Compilarla y revisarla hasta que no exista ningún error de sintaxis.
3. Implementar la interfaz en ***SomeImpl.java*** Compilarla y revisarla hasta que no exista ningún error de sintaxis.
4. Utilizar el compilador de RMI ***rmic*** para procesar la clase de la implementación y generar los ficheros stub y skeleton para el objeto remoto:

   ```
   rmic  SomeImpl
   ```

   Los ficheros generados se encontrarán en el directorio como ***SomeImpl_Skel.class*** y ***SomeImpl_Stub.class.***

   Se deben repetir los pasos 3 y 4 cada vez que se realice un cambio a la implementación de la interfaz.
5. Crear el programa del servidor de objetos ***SomeServer.java***. Compilarlo y revisarlo hasta que no exista ningún error de sintaxis.
6. Activar el servidor de objetos

   ```
   java   SomeServer
   ```

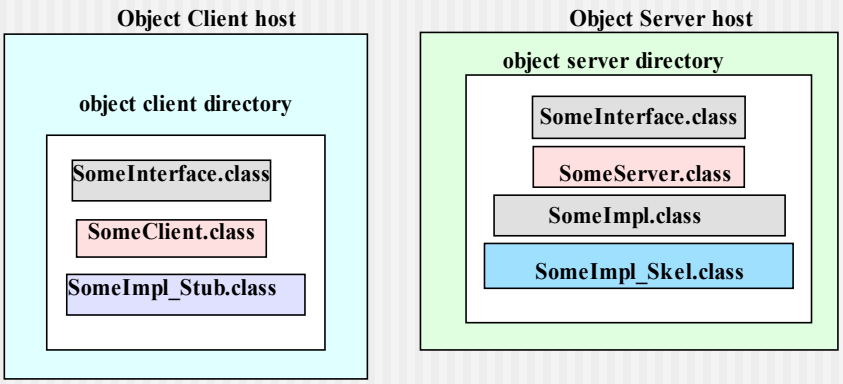
Algoritmo para desarrollar el software en la parte del cliente

1. Crear un directorio donde se almacenen todos los ficheros generados por la aplicación.
2. Obtener una copia del fichero *class* de la interfaz remota. Alternativamente, obtener una copia del fichero fuente de la interfaz remota y compilarlo utilizando ***javac*** para generar el fichero *class* de la interfaz.
3. Obtener una copia del fichero stub para la implementación de la interfaz:

   SomeImpl_Stub.class.
4. Desarrollar el programa cliente ***SomeClient.java***. Compilarlo y revisarlo hasta que no exista ningún error de sintaxis.
5. Activar el cliente.

```
java SomeClient
```

Colocación de los ficheros en una aplicación RMI



Pruebas y depuración de una aplicación RMI

1. Construir una plantilla para un programa RMI básico. Empezar con una interfaz remota que sólo contenga la declaración de un método, su implementación utilizando un stub, un programa servidor que exporte el objeto y un programa cliente con código que sólo invoque al método remoto. Probar la plantilla en una máquina hasta que se pueda ejecutar correctamente el método remoto.
2. Añadir una declaración cada vez a la interfaz. Con cada adición, modificar el programa cliente para que invoque al método que se ha añadido.
3. Rellenar la definición de cada método remoto uno a uno. Probar y depurar de forma detallada cada método añadido antes de incluir el siguiente.
4. Después de que todos los métodos remotos se han probado detalladamente, crear la aplicación cliente utilizando una técnica incremental. Con cada incremento, probar y depurar los programas.
5. Distribuir los programas en máquinas separadas. Probarlos y depurarlos.

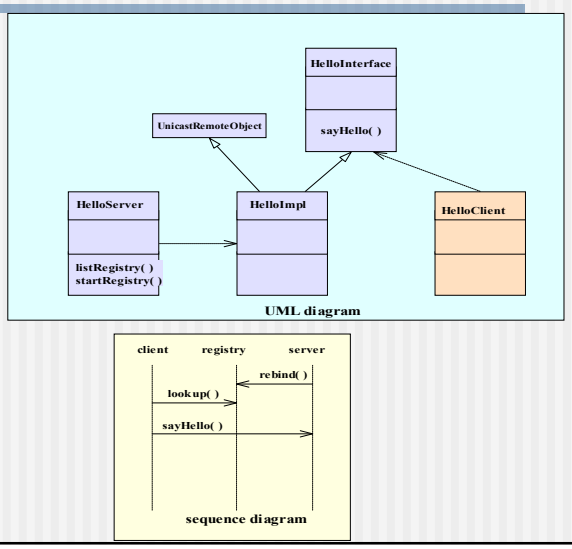
Comparación entre RMI y el API de sockets

El API de RMI es una herramienta eficiente para construir aplicaciones de red. Puede utilizarse en lugar del API de sockets para construir una aplicación de red rápidamente. Sin embargo, esta opción tiene pros y contras:

- El API de sockets está más cercano al sistema operativo, por lo que tiene menos sobrecarga de ejecución. Para aplicaciones que requieran un alto rendimiento, el API de sockets puede ser la única solución viable.
- El API de RMI proporciona la abstracción necesaria para facilitar el desarrollo de software. Los programas desarrollados con un nivel más alto de abstracción son más comprensibles y, por lo tanto, más sencillos de depurar.

El ejemplo HelloWorld

Diagramas para la aplicación Hello

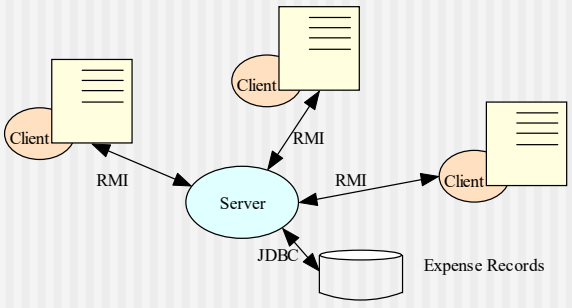


Fuentes para la aplicación Hello

- HelloInterface.java
- HelloImpl.java
- HelloClient.java

Una aplicación empresarial

En la aplicación que ilustramos, el servidor de objeto proporciona métodos remotos que permiten a clientes buscar o actualizar los datos en una base de datos.



An Expense Reporting System, from http://java.sun.com

Resumen - 1

- El paradigma de objetos distribuidos tiene un mayor nivel de abstracción que el paradigma del paso de mensajes.
- Usando el paradigma, un proceso invoca métodos de un objeto remoto, pasando datos como argumentos y recibiendo un valor de respuesta con cada llamada, usando una sintaxis similar a la llamada de métodos locales.
- En un sistema de objetos distribuidos, un objeto servidor proporciona un objeto distribuido cuyos métodos pueden ser invocados por un objeto cliente.

Resumen - 2

- Cada lado necesita un proxy que interactúa con soporte en tiempo real del sistema para realizar las operaciones IPC necesarias.
- Debe estar disponible un registro de objetos que permita a los objetos distribuidos registrarse y buscarse.
- Entre los sistemas más conocidos de objetos distribuidos tenemos RMI, DCOM, CORBA y SOAP.

Resumen - 3

- Java RMI es un ejemplo de sistema de objetos distribuido.
  - La arquitectura de Java RMI incluye tres capas de abstracción tanto en la parte del cliente como del servidor.
  - El software para una aplicación RMI incluye un interfaz remoto, software en la parte del servidor y software en la parte del cliente.