Software Distribuït - T6 - Objectes Distribuïts

Eloi Puertas i Prats

Universitat de Barcelona
Grau en Enginyeria Informàtica

3 d'abril de 2014

UNIVERSITAT DE BARCELONA

Paradigma d'Objectes Distribuïts

- Dóna abstracció sobre els sistemes clàssics client-servidor.
- Es basa en Programació Orientada a Objectes.
- Està orientat a funcionalitats i no a l'intercanvi de missatges.
- Simplificació de protocol i gestió de trames.
- Simplificació en el tractament d'errors.

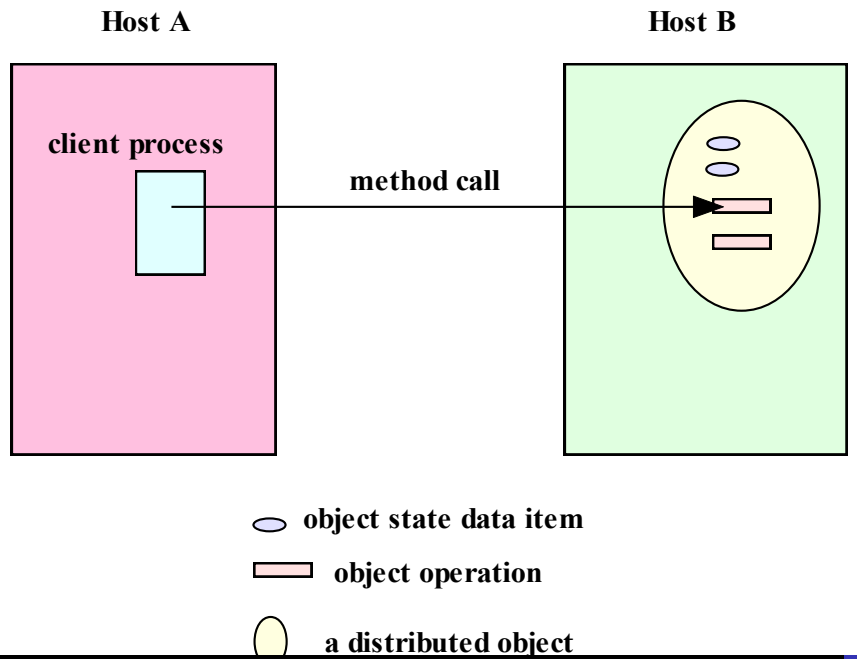
Conceptes fundamentals Objectes Distribuïts

Objecte remot Aquells objectes que algun dels seus mètodes es poden invocar des d'un altre procés corrent en una màquina remota.

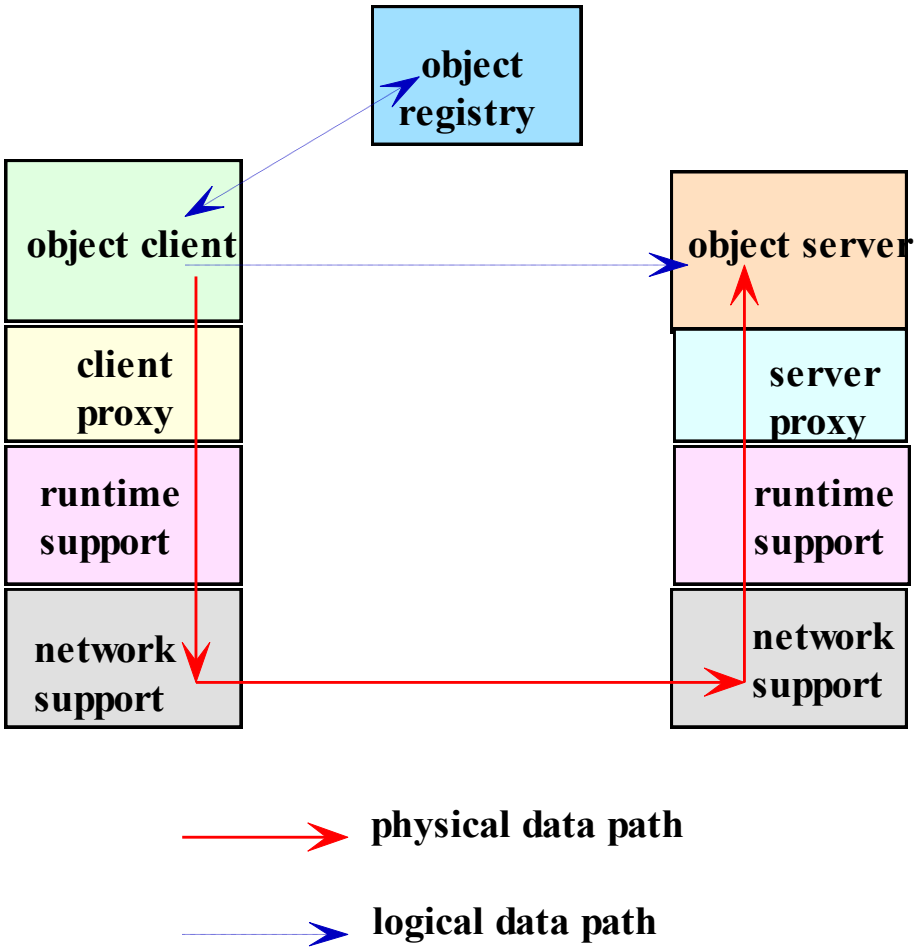
Objecte local Aquells objectes que només poden ser invocats per un procés local en el qual l'objecte existeixi.

Esquema conceptual de funcionament

En un paradigma d'objectes distribuïts, els recursos de xarxa estan representats per objectes distribuïts. Per sol.licitar el servei d'un recurs de xarxa, un procés invoca alguna de les seves operacions o mètodes, passant dades com a paràmetres al mètode. El mètode s'executa a la màquina remota, i la resposta s'envia de nou al procés de sol.licitud com a valor de retorn.



Framework Genèric d'Objectes Distribuïts



Framework Genèric d'Objectes Distribuïts

- Un objecte distribuït és proporcionat, o exportat, per un procés (Servidor d'objectes).
- Un registre d'objectes, ha de ser present en l'arquitectura del sistema per tal de distribuir l'objecte remot.
- Per accedir a un objecte remot, un procés busca una referència remota (handle) en el registre de l'objecte que vol fer servir. Aquesta referència és utilitzada per l'objecte client per realitzar crides als mètodes remots.

Frameworks d'objectes distribuïts

El paradigma d'objectes distribuïts ha estat àmpliament adoptat en les aplicacions distribuïdes, per a això un gran nombre de mecanismes basats en el paradigma estan disponibles. Entre els més coneguts d'aquests mecanismes són:

- Java Remote Method Invocation (RMI).
- Common Object Request Broker Architecture (CORBA).
- Distributed Component Object Model (DCOM)
- Mecanismes que ofereixin el Simple Object Access Protocol (SOAP).

D'aquestes, la més simple és el RMI de Java

UNIVERSITAT DE BARCELONA

JAVA RMI

- L'objectiu principal de RMI és que els programadors desenvolupin aplicacions distribuïdes amb Java usant la mateixa sintaxi i semàntica que les aplicacions Java tradicionals.
- L'aplicació distribuïda s'executarà en vàries màquines virtuals simultàniament.
- L'arquitectura RMI defineix:
  - com es comporten els objectes,
  - quan i com poden ocórrer excepcions,
  - com es gestiona la memòria
  - com es passen els paràmetres a mètodes remots i com es reben els resultats

En quins aspectes ens ajuda JAVA RMI

En la majoria d'aplicacions distribuïdes apareixen 5 aspectes a codificar:

1. Lògica d'aplicació
2. Interfície d'usuari
3. Preparació de dades per a l'enviament a altres processos
4. Codi que llança l'aplicació (permet establir la connexió entre client i servidor)
5. Codi que intenta fer una aplicació distribuïda més robusta i escalable: caches en el client, serveis de noms, balanceig de càrrega, ...

L'RMI ens ajuda amb la generació automàtica de part del codi dels punts 3 i 4.

Conceptes fundamentals RMI

Objecte remot Aquells objectes que algun dels seus mètodes es poden invocar des d'un altre procés corrent en una màquina remota.

Objecte local Aquells objectes que només poden ser invocats per un procés local en el qual l'objecte existeixi.

Servidor d'objectes La màquina virtual que té instanciats els objectes remots

Interfície remota Interfície Java que declara els mètodes d'un objecte remot

Invocació de mètode remot Acció d'invocar un mètode d'una interfície remota en un objecte remot.

RMIregistry Servidor de noms o registre d'objectes remots. Permet localitzar objectes remots a partir del seu nom

Utilitzant objectes remots

Per utilitzar un objecte remot, hem de saber com:

- Obtenir una referència remota.
- Crear un objecte remot i deixar-lo accessible.
- Invocar un mètode remot.

Com obtenir una referència remota o aixecar-se tibant d'una llengüeta

Quan una màquina virtual s'inicia no té cap referència remota al seu abast. Per poder comunicar-se amb l'exterior ha de buscar la primera referència remota utilitzant un servei de directori per localitzar un objecte remot a partir del seu nom. Un cop haguem localitzat el primer objecte remot és possible que aquest ens envïi noves referències remotes com valors de retorn de la crida d'algun dels seus mètodes remots (*Callback*).

El servei de directori: rmiregistry

L'API de RMI ens permet fer servir diferents serveis de directori per registrar un objecte distribüit Utilitzarem un servei de directori simple anomenat rmiregistry, que es proporciona amb el programari de Java Development Kit (SDK). El RMIRegistry és un servidor que s'ha d'executar a la màquina del servidor d'objectes, per convenció i per defecte en el port TCP 1099.

>:/$**rmiregistry**

S'ha d'engegar abans d'intentar utilitzar-lo en el mateix path que el servidor d'objectes, o l'intent d'utilitzar-lo fracassarà.

El registre estarà actiu fins que es mati el procés via CTRL-C, per exemple.

Accés al registre de noms des d'un programa Java

El rmiregistry també es pot engegar de forma dinàmica des de la classe servidor d'objectes:

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

La classe que dóna accés al registre és java.rmi.Naming. Llegir javadoc

Mètodes de rmi.Naming

El servei de noms és, doncs, fonamantalment un directori tipus *guia de telèfons* en el que es pot:

- Afegir i associar un nou objecte remot a un nom (mètode **bind**)
- Eliminar un objecte remot ja existent (mètode **unbind**)
- Modificar l'associació d'un nom existent a un altre objecte remot (mètode **rebind**)
- Cercar un objecte remot per nom (mètode **lookup**)
- Obtenir totes les associacions (mètode **list**)

L'API del RMI JAVA

Un cop ja tenim solucionat com obtenir referències remotes, hem de saber com:

a) crear objectes remots i deixar-los accessibles

b) Invocar un mètode remot

Necessitarem els següents elements:

- Interfície remota tant al costat del servidor com del client.
- L'aplicació del costat del servidor
  - Implementació de la interfície remota
  - Instancia Objecte Remot i el deixa disponible al servei de directori.
- L'aplicació del costat del client, invoca mètodes remots.

UNIVERSITAT DE BARCELONA

La interfície remota

Una interfície remota Java és una interfície que hereta de la interfície marcador Remote de JAVA (no implementa cap mètode en concret).
Especifica els mètodes remots que un objecte remot oferirà.
L'ús d'interfíces facilita la crida de mètodes d'objectes remots, ja que el client només necessita de la classe interfície remota i no pas de tota la implementació.
Tot mètode remot ha llançar una excpeció `java.rmi.RemoteException` ja que poden haver-hi causes de posible error com ara:

- Error en les comunicacions
- Error al empaquetar / desempaquetar paràmetres
- Error de protocol.

Exemple Interfície remota

```
package es.ub.gei.sd.dateserver;

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Date;

public interface DateServer extends Remote {
  public Date getDate() throws RemoteException;
}
```

L'aplicació del costat del servidor

Un servidor d'objecte distribuït és un objecte que proporciona els mètodes i la interfície d'un objecte distribuït. Cada servidor d'objecte:

- ha d'implementar cada un dels mètodes remots especificats en la interfície
- registrar l'objecte que conté la implementació dins del servei de directori.

Es recomana que les dues parts s'ofereix com classes separades.

La implementació de la interfície remota

S'ha de codificar una classe que implementi la interfície remota. Aquesta classe ha d'extendre d' **UnicastRemoteObject**, o si no s'haurà d'exportar amb el mètode `UnicastRemoteObject.exportObject(Remote obj)`

```
package es.ub.gei.sd.dateserver;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;
public class DateServerImpl extends UnicastRemoteObject
  implements DateServer {
  public DateServerImpl() throws RemoteException {}
  public Date getDate() throws RemoteException {
    return new Date();
  }
}
```

El servidor d'objecte remot

La classe del servidor d'objecte remot és una classe que té el codi que crea i exporta un objecte remot.

```
package es.ub.gei.sd.dateserver;
import java.rmi.Naming;
public class Server {
  public static void main(String[] args) throws Exception {
    int portNum=1099;
    DateServerImpl dateServer = new DateServerImpl();
    // This method starts a RMI registry on the local host, if it
    // does not already exists at the specified port number.
    startRegistry(portNum);
    // register the object under the name DateServer in localhost and portNum
     registryURL = "rmi://localhost:" + portNum + "/DateServer";
     Naming.rebind(registryURL, dateServer);
     System.out.println("Some Server ready.");
     }
}
```

El servidor d'objecte remot

- Quan un servidor d'objectes s'executa, l'exportació d'objecte distribuït fa que el procés del servidor comenci a escoltar i esperi que els clients es connectin i sol·licitin el servei de l'objecte.
- Un servidor d'objectes RMI és un servidor concurrent: cada sol·licitud d'un client a un dels mètodes de l'objecte es pot fer en un fil separat. Tingueu en compte doncs que si un procés client invoca diverses crides a mètodes remots, aquestes crides es poden executar simultàneament. Igualment, en la banda del servidor cal tenir en compte que diferents clients poden estar executant el mateix servei. Cal posar sistemes de sincronització allà on calgui.

L'aplicació del costat del client

L'aplicació client és com qualsevol altra classe de Java. La sintaxi de RMI necessària és la que fa referència a:

- Localitzar el registre RMI en l'equip servidor,
- Buscar la referència remota de l'objecte remot,
- Convertir a la classe d'interfície remota i invocar els seus mètodes remots.

L'aplicació del costat del client

```
package es.ub.gei.sd.dateclient;
import java.rmi.Naming;
import java.util.Date;
import es.ub.etis.ppx.dateserver.DateServer;
public class DateClient {
  public static void main(String[] args) throws Exception {
    if (args.length != 1)
      throw new IllegalArgumentException("Syntax: DateClient <hostname:port>");
    DateServer dateServer = (DateServer) Naming.lookup("rmi://"+args[0]+"/DateServer");
    Date when = dateServer.getDate();
    System.out.println(when);
  }
}
```

L'aplicació del costat del client

- La referència de la interfície remota s'ha d'utilitzar per invocar qualsevol dels mètodes remots.
- Noteu que la sintaxi de la invocació dels mètodes remots és la mateixa que per als mètodes locals.
- ALERTA És un error comú convertir la referència obtinguda des del registre a la classe d'implementació de la interfície en comptes de la classe d'interfície.

Passos per crear una aplicació RMI client/servidor

En l'aplicació del client

- Interficies d'Objectes Remots.
- Codi Aplicació Client
  - Main
  - GUI
  - Crides a mètodes remots

En l'aplicació del servidor

- Interficies d'Objectes Remots
- Implementacions d'Objectes Remots
- Codi Aplicació Servidor
  - Main
  - Instanciació d'Objectes Remots
  - *(Engegar RMIRegistry des de l'aplicació)
  - Publicitar Objectes Remots al servidor de directori.

UNIVERSITAT DE BARCELONA

Passos per arrencar una aplicació RMI client/servidor

En la màquina Servidor

- *(Engegar rmiregistry)
- Executar l'aplicació del servidor

En les màquines Clients

- Executar l'aplicació del client.

ALERTA Cal vigilar que el rmiregistry estigui executant-se en el mateix path que el servidor, en cas contrari cal tenir-ho en compte quan especifiqueu la URL `rmi://...`

ALERTA Les interfícies dels objectes remots a client i a servidor han de ser exactament iguals, vigileu si feu servir packages.

Diferències entre invocació local i remota

- Els clients d'objectes remots sempre interactuen amb interfícies remotes, mai amb les classes d'implementació d'aquestes interfícies.
- Els arguments no-remots, així com els resultats retornats per la invocació d'un mètode remot, són passats per còpia en lloc de per referència. És necessari que implementin `Serializable`.
- Els objectes remots es passen sempre per referència (mitjançant Interfície Remota), no copiant les implementacions remotes.
- El significat d'alguns dels mètodes definits en java.lang.Object s'especialitzen en el cas d'objectes remots.
- Atès que una invocació remota pot fallar per raons addicionals a les que ho fa una invocació local, en treballar amb objectes remots s'han d'atendre excepcions addicionals

Diferències entre invocació local i remota

- **Invocació local**

```
public class Cliente {...}
public class IdentificadorCliente {...}
public Cliente obtenerCliente(IdentificadorCliente id);
```

S'envia una referència local i es rep una referència local

Diferències entre invocació local i remota

- **Invocació remota amb paràmetre local i retorn remot**

```
public Interficie ICliente extends Remote{...}
public class Cliente implements ICliente {...}
public class IdentificadorCliente implements Serializable {...
public ICliente obtenerCliente(IdentificadorCliente id)
throws RemoteException;
```

*S'envia una còpia de l'objecte **id** i es rep una referència remota*

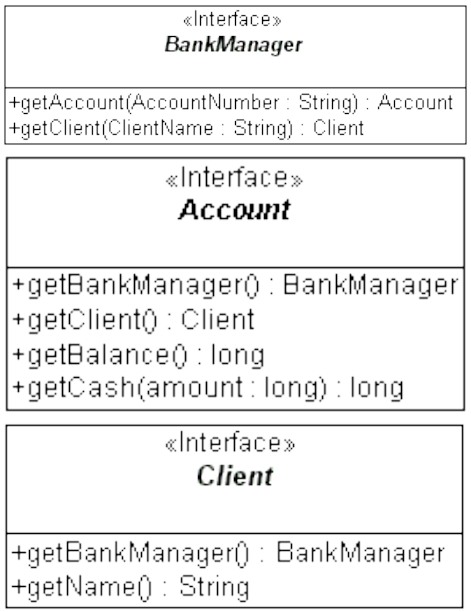
Diferències entre invocació local i remota

- **Invocació remota amb paràmetre remot i retorn remot**

```
public Interficie ICliente extends Remote{...}
public Interficie IIdentificadorCliente extends Remote{...}
public class Cliente implements ICliente {...}
public class IdentificadorCliente implements
IIdentificadorCliente{...}
public ICliente obtenerCliente(IIdentificadorCliente id)
throws RemoteException;
```

*S'envia una referència remota **id** i es rep una referència remota d' **ICliente***

Exemple Interfícies de SimpleBankSystem



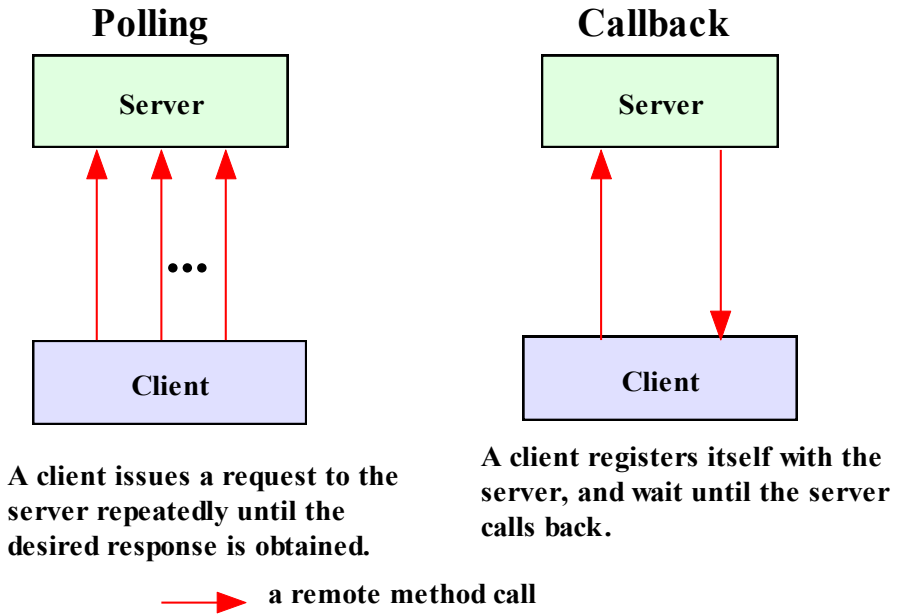
RMI Avançat. Més enllà del C/S

En el model client-servidor, el servidor és passiu: el client inicia l'IPC, el servidor espera l'arribada de les sol·licituds i proporciona respostes. Però algunes aplicacions requereixen que el servidor iniciï la comunicació en determinades incidències *Callback*. Exemples d'aplicacions són:

- Monitorització.
- Jocs.
- Subhastes.
- Votacions / Enquestes
- Sales de xats
- Tauler d'anuncis/ Servei de missatges.
- Treball en grup

Polling vs. Callback

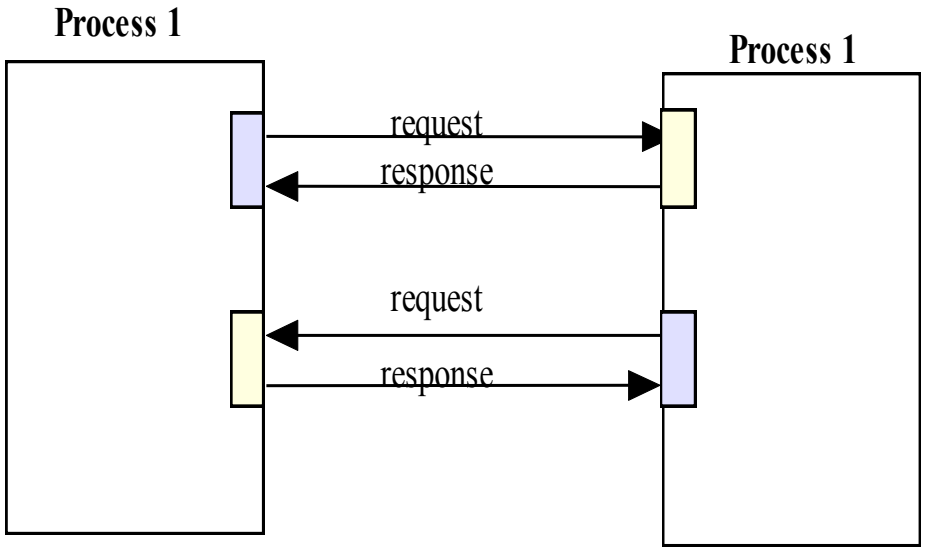
Si no hi ha callback, en cas que el client hagi de ser notificat sobre un esdeveniment asíncron que ha ocorregut en el costat del servidor, el client l'hauria de sondejar (*polling*) diverses vegades.



Comunicació en ambdos sentits

Algunes aplicacions requereixen que les dues parts puguin iniciar l'IPC.

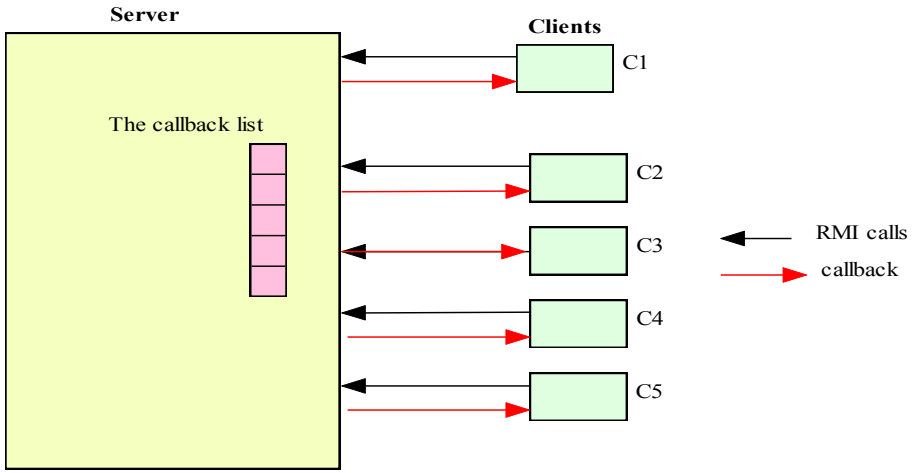
Utilitzant sockets, una comunicació dúplex es pot aconseguir mitjançant l'ús de dos sockets en cada costat. Amb sockets orientats a la connexió, cada part actua com a client i servidor.



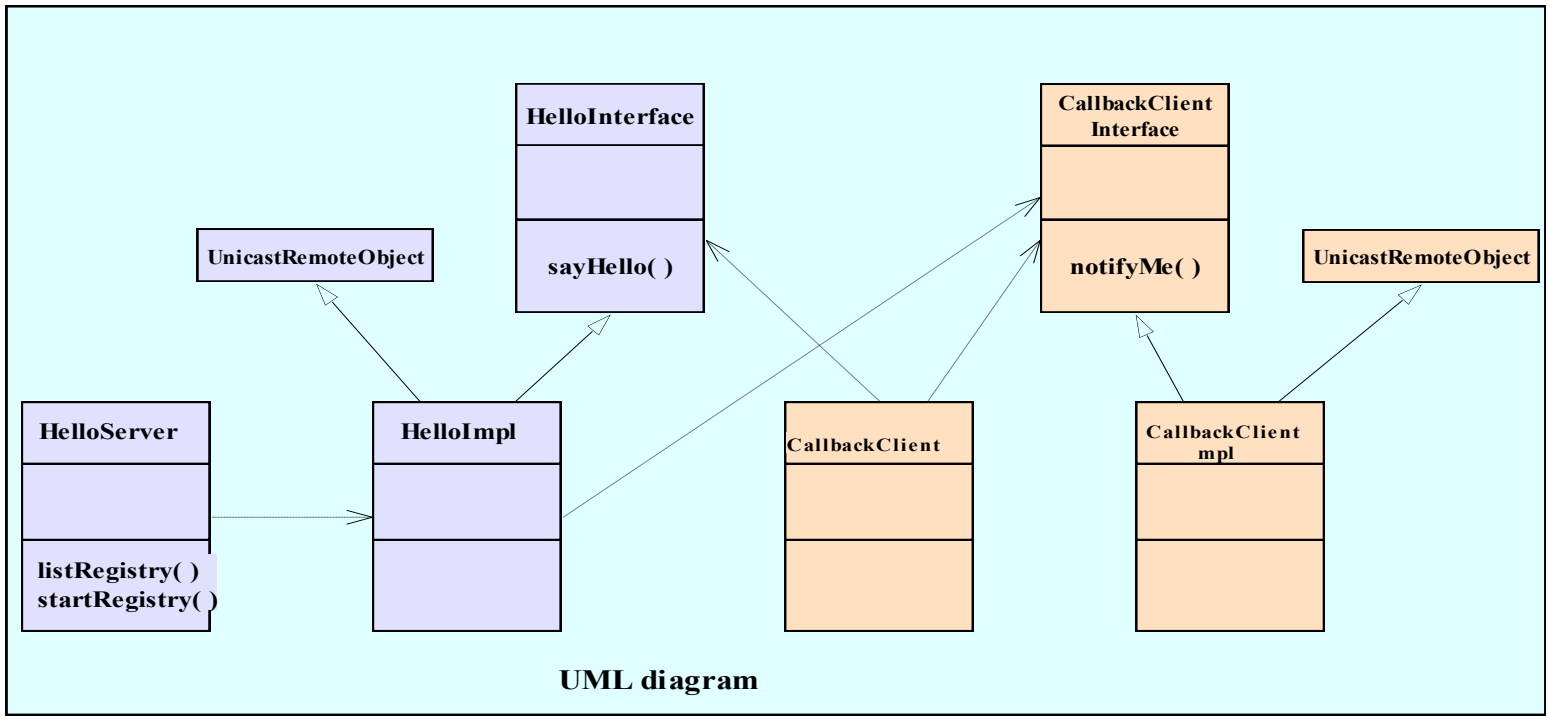
RMI Callbacks

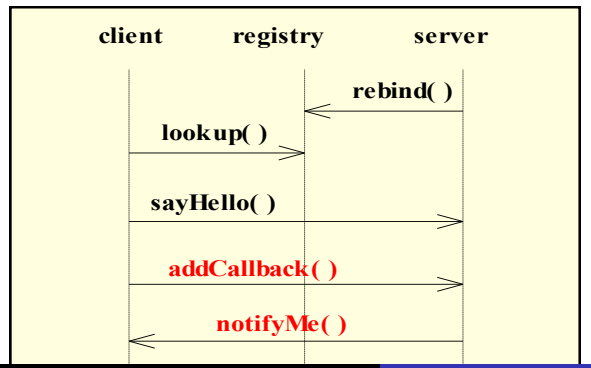
Elements necessaris per realitzar el callback:

- Un client amb Callback es registra a un servidor amb RMI.
- El servidor fa callbacks a cada client registrat davant l'ocurrència d'un esdeveniment determinat.



UML RMI Callbacks





Interfícies RMI Callbacks

- El servidor proporciona un mètode remot que permet a un client que es registri per a fer-li el callback.
- Es necessita una interfície remota pel callback a la banda de client, a més de la interfície de l'objecte remot del costat del servidor.
- Aquesta interfície especifica un mètode per rebre un callback des del servidor.
- El programa client és una subclasse de RemoteObject i implementa la interfície de callback, incloent-hi el mètode per rebre'l.
- El client, abans de res, s'ha registrar al servidor per rebre el callback.
- El servidor invoca el mètode remot del client en cas de produir-se l'esdeveniment esperat.

Estructura aplicació distribuïda RMI amb Callbacks

Banda Client	**Banda Servidor**
• Interfície Callback Client • Implementació Interfície Callback Client • Aplicació Client	• Interfície Callback Servidor • Implementació Interfície Callback Servidor • Aplicació servidor • RMIRegistry

Interfície Callback Servidor

```
import java.rmi.*;
public interface CallbackServerInterface extends Remote {
    throws java.rmi.RemoteException;
// This remote method allows an object client to
// register for callback
// @param callbackClientObject is a reference to the
//        object of the client; to be used by the server
//        to make its callbacks.
  public void registerForCallback(
    CallbackClientInterface callbackClientObject
    ) throws java.rmi.RemoteException;
// This remote method allows an object client to
// cancel its registration for callback
  public void unregisterForCallback(
    CallbackClientInterface callbackClientObject)
    throws java.rmi.RemoteException;
}
```

Implementació Interfície Callback Servidor

```
public synchronized void registerForCallback(
    CallbackClientInterface callbackClientObject)
    throws java.rmi.RemoteException{
      // store the callback object into the vector
      if (!(clientList.contains(callbackClientObject))) {
          clientList.addElement(callbackClientObject);
      System.out.println("Registered new client ");
      }
public synchronized void unregisterForCallback(
    CallbackClientInterface callbackClientObject)
    throws java.rmi.RemoteException{
    if (clientList.removeElement(callbackClientObject)) {
      System.out.println("Unregistered client ");
    } else {
       System.out.println(
         "unregister: clientwasn't registered.");
    }
```

Aplicació Callback Servidor

```
  public static void main(String args[]) {
...
      startRegistry(RMIPortNum);
      CallbackServerImpl exportedObj =
        new CallbackServerImpl();
      registryURL =
        "rmi://localhost:" + portNum + "/callback";
      Naming.rebind(registryURL, exportedObj);
      System.out.println("Callback Server ready.");
...
 }
```

Interfície Callback Client

```
import java.rmi.*;
public interface CallbackClientInterface
  extends java.rmi.Remote{
  // This remote method is invoked by a callback
  // server to make a callback to an client which
  // implements this interface.
  // @param message - a string containing information for the
  //                  client to process upon being called back
    public String notifyMe(String message)
      throws java.rmi.RemoteException;
} // end interface
```

Implementació Interfície Callback Client

```
import java.rmi.*;
import java.rmi.server.*;
public class CallbackClientImpl extends UnicastRemoteObject
     implements CallbackClientInterface {
   public CallbackClientImpl() throws RemoteException {
      super( );
   }
   public String notifyMe(String message){
      String returnMessage = "Call back received: " + message;
      System.out.println(returnMessage);
      return returnMessage;
   }
}// end CallbackClientImpl class
```

Aplicació Callback Client

```
public static void main(String args[]) {
...
      String registryURL =
      "rmi://localhost:" + portNum + "/callback";
    // find the remote object and cast it to an
    //   interface object
    CallbackServerInterface h =
      (CallbackServerInterface)Naming.lookup(registryURL);
    System.out.println("Lookup completed " );
    System.out.println("Server said " + h.sayHello());
    CallbackClientInterface callbackObj =
      new CallbackClientImpl();
    // register for callback
    h.registerForCallback(callbackObj);
    System.out.println("Registered for callback.");
...
}
```

Objectes Remots amb múltiples interfícies

Un mateix Objecte Remot pot donar serveis a diferents tipus de clients. Si totes les funcionalitats que dóna es troben en única Interfície Remota, qualsevol client pot invocar mètodes que no li pertocarien

Exemple: Aplicació Distribuïda gestió de notes per alumnes i professors.

Problema: Com fem que professors puguin modificar notes i no els alumnes?

Solució:

Objectes Remots amb múltiples interfícies

Un mateix Objecte Remot pot donar serveis a diferents tipus de clients. Si totes les funcionalitats que dóna es troben en única Interfície Remota, qualsevol client pot invocar mètodes que no li pertocarien

Exemple: Aplicació Distribuïda gestió de notes per alumnes i professors.

Problema: Com fem que professors puguin modificar notes i no els alumnes?

Solució: Objecte Remot que implementa dues Interfícies: Una per l'alumne i una altre pel professor. Cada client disposa només de la Interfície que li pertoca amb els seus mètodes.

RMI & Threads

- Les crides a mètodes remots poden esdevenir en diferents threads en la banda de l'objecte remot.
- cal assegurar-se que tot el codi dins dels mètodes sigui thread-safe
- Cal sincronitzar només les zones d'exclusió mútua (accés a zones de dades compartides).
- Cal anar amb compte amb les colections de java, ja que no són thread-safe. Utilitzeu les col·leccions que hi ha a `java.utils.concurrent`

Broadcast en RMI

Enviar un missatge a tots els usuaris registrats:

```
for( IClient client : registredClients)
    client.notifyResult("OK");
```

Problemes:

- No és just (s'envia de forma seqüencial)
- No és robust (si un client cau o no respon, els demés es veuen afectats)

Solució: Aproximació MultiThread

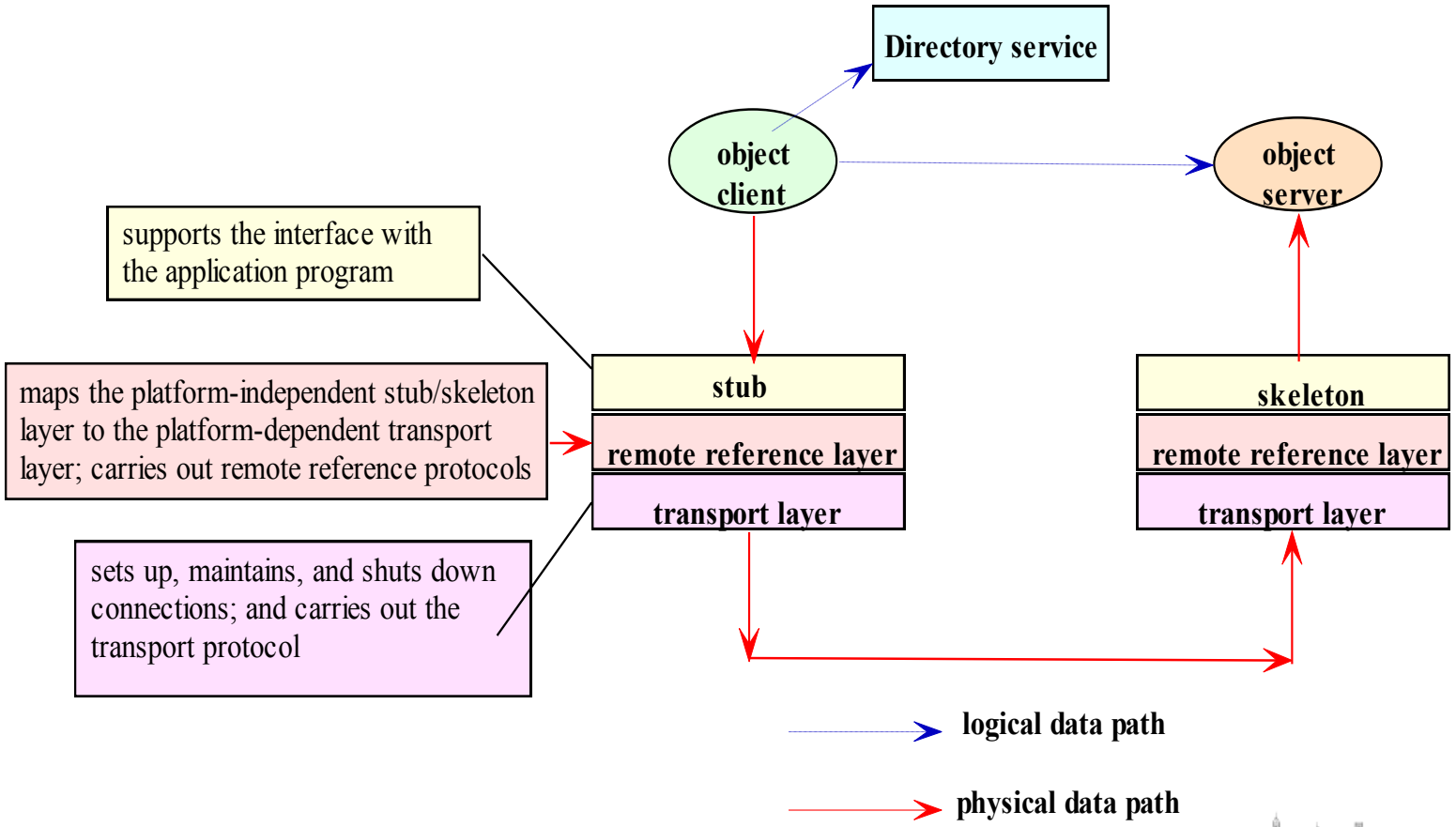
```
for( IClient client : registredClients):
    new ClientNotifyThread(client,"OK").start()
```

Peer2Peer fent servir RMI

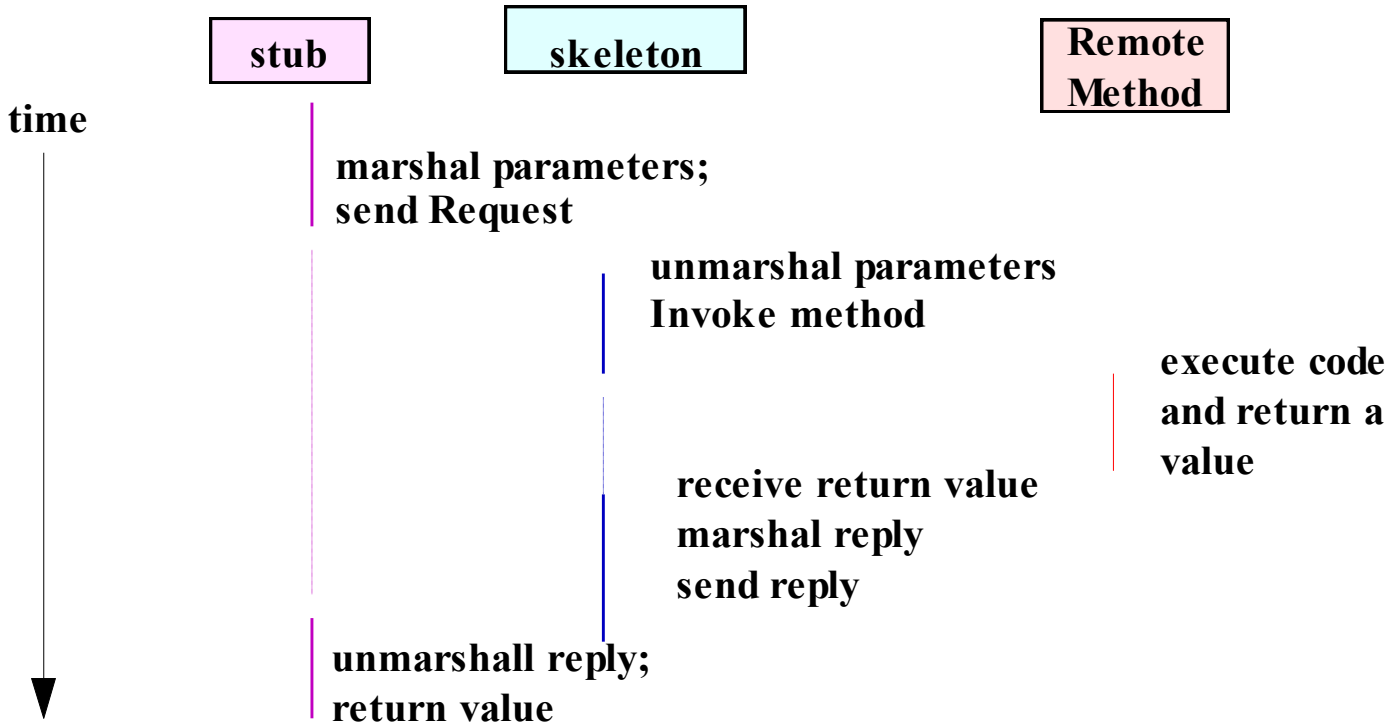
No existeix diferència entre rols, cada peer és igual que els demés.

- Cada peer pot comunicar-se amb qualsevol peer, sense haver de necessitar un server.
- Cada peer ha de poder localitzar els demés peers.
  - Una taula hash distribuïda és necessària per tenir totalment descentralitzada la localització dels peers.
  - Cada peer pot mantenir una part de la hash actualitzada amb els seus peers ”propers”.
- Ens hem d’assegurar que el peer és realment qui diu ser.
  - En registrar-se al servidor, aquest li ha de proporcionar una clau i el peer presentar-la en cada iteració.
  - En el cas de comunicació entre peers, s’ha de presentar una clau pública tipus RSA i verificar en el servidor que la referència remota és realment del mateix usuari que es va registrar.

Arquitectura interna RMI



Stubs & Skeletons



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

Recull les crides locals de mètodes remots i gestiona el seu enviament a l'objecte remot. Gestiona també la recepció per part de l'objecte remot de la crida al mètode i el retorn d'informació.

Stub

Un stub d'un objecte remot actua com un representant local a la part del client de l'objecte remot. El stub recull la petició localment i l'envia a l'objecte remot. Quan s'invoca un mètode en un stub, aquest:

1. Inicia una connexió amb la JVM remota que conté l'objecte remot.
2. Empaqueta i transmet els paràmetres a la JVM remota.
3. Espera al resultat de la invocació del mètode.
4. Llegeix i desempaqueta el valor de retorn o l'excepció retornada.
5. Retorna el valor a l'objecte local que va fer la invocació.

Skeleton

A la JVM remota, cada objecte remot té un skeleton associat. El skeleton és el responsable d'enviar la crida rebuda des del client a la implementació de l'objecte remot. Quan un skeleton rep l'entrada d'una invocació a un mètode:

1. Pot crear un nou thread per atendre la petició.
2. Llegeix i desempaqueta els paràmetres.
3. Invoca el mètode en l'objecte local.
4. Empaqueta i transmet el resultat de la invocació.

Generació de Stubs i Skeletons

Així doncs, els Stubs i Skeletons són classes Java que inclouen el codi que empaqueta i desempaqueta paràmetres, obre i tanca sockets, realitza les operacions d'E/S, l'accept, etc.

- Històricament es generaven utilitzant l'executable rmic sobre el. class de la classe remota: `$ rmic ImplObjecteRemot.class`. Els Stubs s'havien de distribuir als clients.
- A partir de la versió 1.2 de Java, els Skeletons es generen automàticament en exportar els objectes.

```
$ rmic -keep -v1.1 BankManagerImpl
```

BankManagerImpl_Skel.java
BankManagerImpl_Stub.java

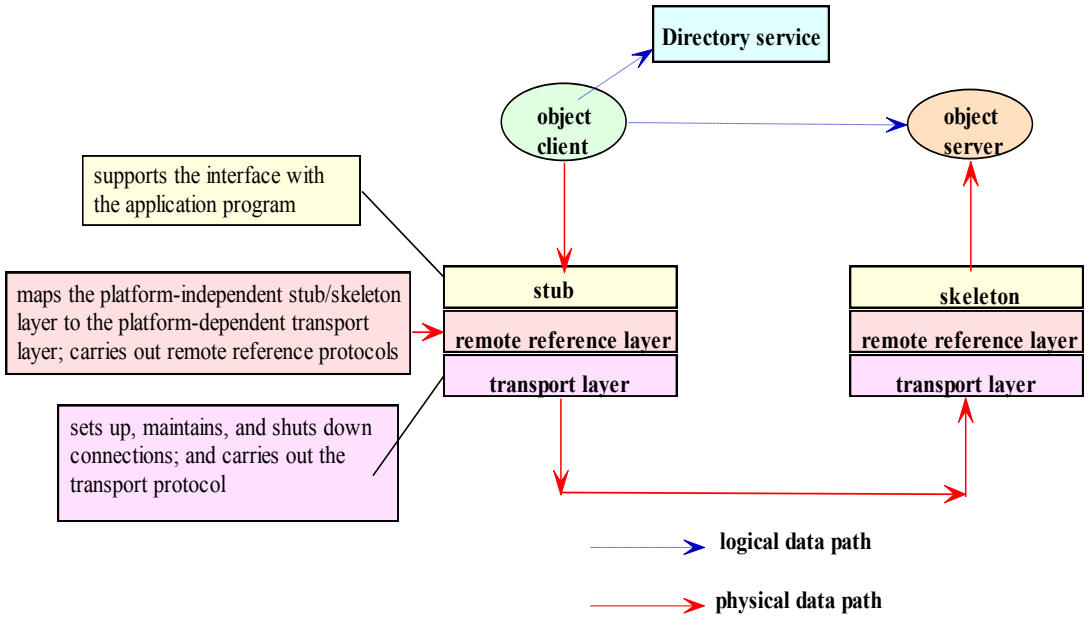
Generació de Stubs i Skeletons

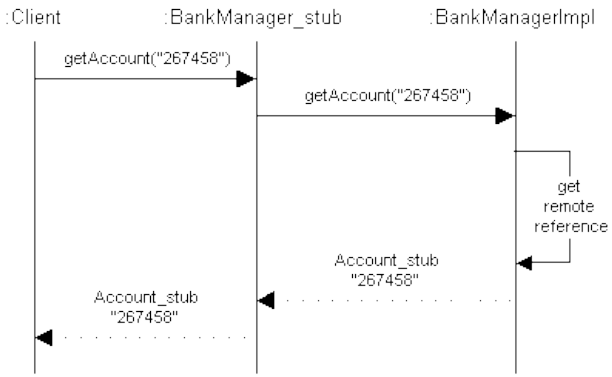
A partir de la versió 1.5 de Java els stubs ja no cal generar-los ni distribuir-los:

- En el servidor: En instanciar-se un objecte `UnicastRemoteObject` una instància de `java.lang.Proxy` és construïda, que implementa totes les interfícies remotes de l'objecte remot i té com a controlador d'invocacions un `java.rmi.server.RemoteObjectInvocationHandler`.
- En el client: A partir de la l'invocació d'un mètode remot i l'instància de `java.lang.reflect.Proxy` la funcionalitat d'invocar el mètode remot del stub es fa de forma automàtica.

Alerta si es fa servir el mètode estàtic `UnicastRemoteObject.exportObject(Remote)` per exportar un Objecte Remot es faran servir els stubs generats per rmic.

Capa de Referències Remotes





Defineix les diferents semàntiques d'invocació d'un servei remot. Aquesta capa proporciona un objecte de tipus RemoteRef que facilita la implementació dels stubs i skeletons, que faran ús de les referències remotes.

Mètodes de java.rmi.server.RemoteRef

Mètodes de RemoteRef:

- `Object invoke (Remote obj, Method method, Object [] params, long opnum)`

  Permet invocar un mètode en un objecte remot.

- `boolean remoteEquals (RemoteRef obj)`

  Compara dos objectes remots per veure si són iguals.

- `int remoteHashCode ()`

  Retorna un codi hash per un objecte remot. Dues referències remotes que apuntin al mateix objecte retornaran el mateix valor de codi hash.

- `String remoteToString ()`

  Retorna un string que representa l'objecte remot.

UNIVERSITAT DE BARCELONA

Garbage Collector Distribuït

L'RMI implementa un GC distribuït per tal de:

1. Assegurar que si existeix una referència local o remota d'un objecte distribuït,aquest continuarà existint.
2. Tan bon punt com ja no quedi cap referència remota o local sobre l'objecte remot, serà completament esborrat de la memòria del servidor.

El GCD està basat en un comptador de referències igual que el GC normal de JAVA, però les referències remotes en els clients han de ser reportades al servidor de l'objecte mitjançant crides remotes. El GCD i GC local treballen en cooperació.

Distribució automàtica de fitxers

Tal com hem vist la distribució d'aplicacions fins ara, RMI no permet enviar els .class, sinó únicament objectes. Només viatgen les dades. Però els clients i servidors han de compartir tots els .class d'objectes que s'intercanvien com a paràmetres i/o valors de retorn.

- **Problema:** Si en un determinat moment el servidor crea una certa subclasse d'una classe de retorn, el client l'haurà d'incloure també. Es trenca opacitat d'implementació.
- RMI ofereix la possibilitat que les classes es descarreguin d'un o diversos repositoris de codi. És habitual que hi hagi un únic repositori de codi gestionat per qui gestioni el servidor.

Distribució automàtica de fitxers

Quan el client o el servidor troben una classe que no coneixen l'intenten carregar:

1. del CLASSPATH
2. o d'una URL, si la referència remota du internament una propietat amb l'URL d'on es troben els .class necessaris.

Per motius de seguretat, per a que es realitzi aquesta descàrrega de codi s'haurà d'instal·lar un SecurityManager, en cas contrari saltarà una SecurityException.

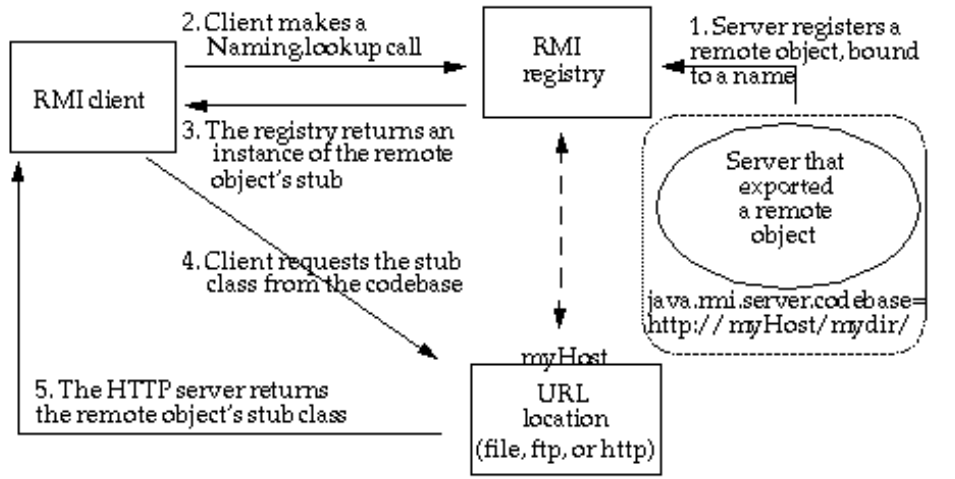
- **Client:** `java -Djava.security.policy=java.policy SomeClient`
- **Servidor:** `java -Djava.rmi.codebase=<URL> -D-Djava.security.policy=java.policy Server`

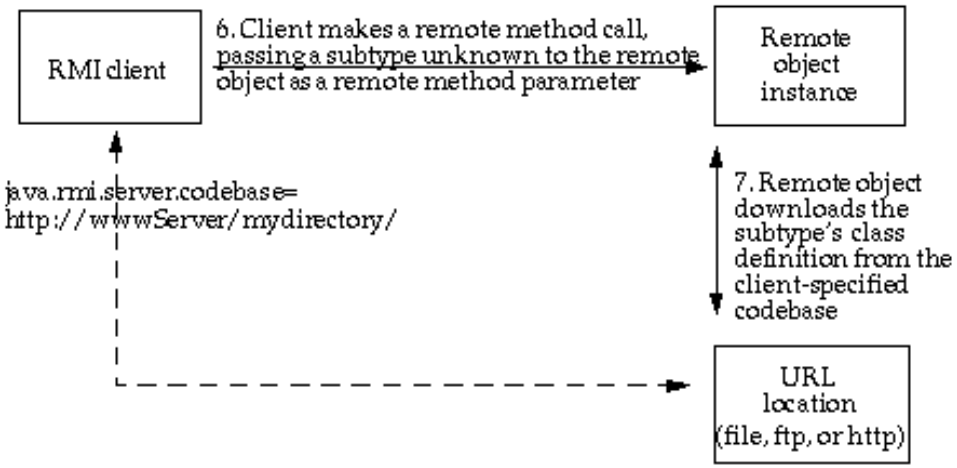
UNIVERSITAT DE BARCELONA

Fitxer java.policy

```
grant {
    // Allows RMI clients to make socket connections to the
    // public ports on any host.
    // If you start the RMIregistry on a port in this range,
    // you will not incur access violation.
    permission java.net.SocketPermission "*:1024-65535",
        "connect, accept, resolve";
    // Permits socket access to port 80, the default HTTP port
    // - needed by client to contact an HTTP server for stub
    // downloading.
    permission java.net.SocketPermission "*:80", "connect";
};
```

Arquitectura RMI amb distribució de classes





Objectes Activables

- Un objecte és actiu quan està disponible per rebre invocacions en un procés en marxa.
- Un objecte és passiu quan no es troba en estat actiu però es pot activar. Consisteix en
  - La implementació dels seus mètodes.
  - L'estat en que es troba emmagatzemat.
- Quan un objecte passiu necessita d'activar-se degut a una invocació remota, es crea una nova intstància de l'objecte a partir de l'estat emmagatzemat.

  Implementació en RMI:
- Necessitem que el nostre objecte remot extengui de la classe `Activatable` en comptes de `UnicastRemoteObject`.
- Necessitem activar el servei dimoni rmid en el servidor on es trobi l'objecte remot per a que pugui gestionar l'activació
- Tutorial sobre `Activatable`