Utilizarea RMI în realizarea aplicațiilor distribuite Java

Ing. Augustin MAN
Centrul de Informatică Aplicată Cluj

Într-un articol precedent (Nr.4/1997, pag.49) se explicau avantajele utilizării mediului de dezvoltare Java pentru crearea aplicațiilor complexe, destinate integrării în peisajul informaticii economice moderne.
Articolul de față, cu un caracter mai tehnic, se ocupă de partea destinată realizării prelucrărilor distribuite din cadrul Java. Astfel, se prezintă principiile care stau la baza RMI, avantajele alegerii acestei soluții comparativ cu altele și exemplificări concrete de abordare. Menționăm că în partea a doua a materialului se explică tehnici avansate de programare RMI, unele cu caracter original. Articolul se adresează în principal programatorilor cu experiență în domeniul Java, dar poate fi util pentru oricine este interesat de studiul aplicațiilor distribuite în acest context.
Cuvinte cheie: *RMI, Java, client, server, aplicație distribuită, thread, metodă, interfață, transmisie, apel la distanță.*

1. Soluția RMI

RMI (abrevierea Remote Method Invocation, adică apelarea metodelor la distanță) reprezintă soluția integrală Java propusă de către Sun la problema prelucrării distribuite. O astfel de soluție destinată unui mediu unic are desigur chiar dezavantajul acestei unicități. Pe de altă parte, argumentele în favoarea RMI sunt destul de puternice ca să determine investitia de dezvoltare a unei firme de anvergura Sun. În primul rând, mediul suport nu este unul clasic, ci o platformă modernă, având calitatea intrinsecă a portabilității. În al doilea rând, RMI se impune față de variantele heterogene ale prelucrării distribuite prin simplitate: dacă în toate punctele de prelucrare din cadrul aplicației este implementată Java, nu mai este necesar nici un fel de software auxiliar. Comparând acest mod de lucru cu CORBA (Common Object request Broker Architecture) sau DCE (Distributed Computer Environment) concluzia este evident în favoarea RMI. De aceea credem că alegerea RMI pentru realizarea distributivității de prelucrare din cadrul unor aplicații economice complexe este corectă și de viitor.
Principala caracteristică a RMI este aceea de a permite apelul metodelor unor obiecte îndepărtate ca și cum acestea ar fi instațiate local. Apelul se aliniază standardului Java: *object.method(args)* degrevîndu-l pe programator de efortul gestiunii transmisiei (programare de socket, temporizare, reluări etc.)
O comunicare caracteristică RMI se stabilește între un obiect local, numit client, care apelează o metodă și unul îndepărtat numit server, care implementează metoda apelată de către client. Pentru cele ce urmează stabilim două convenții:
- întrucît RMI se referă exclusiv la obiecte, nu vom mai menționa cuvântul obiect, ci doar clienți și/sau servere;
- server-ul va fi întotdeauna obiectul îndepărtat, chiar cînd se vor trata probleme specifice acestuia.

2. Comunicarea RMI

O comunicare RMI are loc astfel:
- server-ul atașează un nume instanței sale și permite ca numele respectiv și cel al metodelor care pot fi apelate de către clienți să fie cunoscut pretutindeni;
- clienții accesează informațiile de mai sus și pe baza lor apelează metodele server-ului.
Întrucît orice aplicație distribuită se desfășoară între mai multe spații de adrese (minim două), regula de trasmitere a argumentelor unei metode și a valorii de răspuns exclusiv prin referință, așa cum se face într-un spațiu local Java, nu mai corespunde. Se aplică următoarele reguli:

- obiectele îndepărtate sunt accesate prin referință nominală; obiectul ca atare nu părăsește niciodată site-ul propriu;
- argumentele și valorile de răspuns ale metodelor sunt serializate, adică copiate între site-ul local și cel îndepărtat.

În plus, server-ul trebuie să implementeze (sub forma unei extensii a clasei Remote) o interfață cuprinzînd semnăturile metodelor apelabile de către clienți. Un utilitar specializat (rmic - RMI Compiler) creează din aceasta o structură (template) numită stub. Este necesar ca stub-ul să se afle pe site-ul local, întrucît clientul se adresează exclusiv stub-ului. Structura echivalentă stub-ului de la site-ul îndepărtat se numește skeleton. La un apel RMI, informația circulă astfel: argumentele metodei apelate trec de la client la stub și apoi, prin intermediul nivelului transport al protocolului specific (rmi//:), la skeleton și de aici la server. Valoarea de răspuns revine pe același drum parcurs în sens invers.

3. Registry

Orice sistem care implementează servere RMI trebuie să posede un proces background care să țină evidența acestora în conformitate cu cele precizate în paragraful anterior. Acest proces se numește *rmiregistry*. Portul de ascultare implicit este 1099, dar se poate utiliza oricare port liber. Pentru un port precizat, pe sistem va rula un singur proces rmiregistry, care trebuie să aibă acces la elementele de comunicare (stubs și skeletons) ale obiectelor a căror evidență o controlează. Principalele acțiuni implicînd rmiregistry sunt:
- crearea la site-ul îndepărtat unui proces rmiregistry (dacă nu există) prin LocateRegistry.createRegistry();
- înregistrarea la site-ul îndepărtat a unui obiect (server) prin Naming.bind() sau Naming.rebind();
- la site-ul local, căutarea unui server în rmiregistry, prin Naming.lookup().

4. Interacțiuni peer-to-peer

În aplicații reale de o complexitate mai mare acest mod de comunicare nu este suficient: se pune problema realizării unor servere care să poată apela la rândul lor metode ale clienților, precum și ale altor servere. În continuare vom analiza succint astfel de situații.

4.a. Servere cu caracteristici de client

Singura complicație care intervine în acest caz este aceea că server-ul apelat de către server-ul client poate să nu fie activ. În această situație server-ul client nu poate pur și simplu să termine prelucrarea, pentru că în calitate de server, el deservește deja niște clienți. Soluția este reluarea încercării de conectare, prin tratarea excepțiilor respective, de felul:

```
while (true)
 {
  try
  {
   intf = (Interface)Naming.lookup(name);
         break;
  }
  catch (NotBoundException e)
  {
         continue;
  }
 catch (ConnectException e)
  {
         continue;
  }
 catch (Exception e)
   {
System.err.println("Server WAIT error: " + e);
         break;
    }
}
```

Aceasta secvență nu este viabilă pentru o aplicație reală întrucît duce la "gâtuirea" altor prelucrări: în mod normal, se va crea un thread separat, în cadrul căruia se va relua încercarea de conectare din timp în timp (de exemplu o secundă).

4.b. Clienți cu caracteristici de server

Această situație este puțin mai complicată: esența constă în necesitatea clientului de a-și implementa o interfață de acces (template) pe care s-o trimită server-ului. Evident că server-ul trebuie să posede o astfel de metodă apelabilă de la client, care să conțină

ca argument interfața clientului. După primirea referinței la interfața client, server-ul poate apela toate metodele pe care clientul dorește să i le pună la dispoziție prin interfața respectivă. În plus, clientul trebuie să fie declarat ca un obiect îndepărtat (tip server), printr-o operație de export asupra propriei instanțe. Detalierea următoare va demonstra simplitatea implementării în pofida complexității aparente a exprimării descriptive:
- clientul creează o instanță a clasei sale:
Client cl = new Client();
- se autoexportă, devenind un obiect tip server:
UnicastRemoteObject.exportObject(cl);
- obține interfața server-ului:
Interface intf = (Interface)Naming.lookup (name);
- apelează metoda server-ului pentru trimiterea interfeței:
intf.callClient(cl);
Clientul și-a declarat metoda apelabilă de către server într-o interfață, așa cum s-a explicat în paragrafele anterioare:

```
public interface ClientInterface
        extends Remote
{
public void display(String s)
        throws RemoteException
}
```

și

```
public class Client
        implements ClientInterface
{...}
```

Server-ul și-a declarat metoda astfel:

```
public interface Interface
        extends Remote
{
public callClient(ClientInterface ci)
        throws RemoteException
}
```

În fine, server-ul apelează metoda clientului:

```
public void callClient(ClientInterface ci)
        throws RemoteException
{
ci.display("Client is being called by Server");
}
```

În finalul acestui paragraf să mai observăm că deși argumentul metodei *callClient()* este interfața client, întrucât o interfață nu poate fi instanțiată, se poate apela metoda furnizându-i ca argument o instanță a obiectului care implementează interfața respectivă.

5. Aplicații reale

Într-o aplicație distribuită, performanțele serverelor trebuie să fie superioare celor ale clienților pentru că impactul acestora din urmă asupra întregii aplicații nu este crucial. Pe de altă parte, server-ele, datorită importanței lor, trebuie să-și optimizeze prelucrările. De aceea, un server care apelează metodele altor servere sau ale unor clienți trebuie să execute aceste acțiuni pe cît posibil în fluxuri (thread-uri) independente astfel încât:
- o eroare să nu oprească thread-ul principal, blocînd întregul server;
- o întîrziere a răspunsului unui alt nod să nu irosească resurse sistem.
Un server poate activa thread-uri în două moduri:
- un thread separat în mod implicit: o soluție comodă, dar care partajează prelucrarea doar în două categorii: thread-ul pricipal pentru prelucrări locale și cel nou creat pentru toate celelalte prelucrări;
- oricîte thread-uri dorește, prin subclasarea clasei Thread. Este soluția recomandabilă în toate situațiile complexe.
Să analizăm cîteva scenarii de lucru.

5.a. Un thread pentru toți clienții apelați de către server

În metoda callClient() (§ 4.b.) se face:
ClientThread ct = new ClientThread(ci)
Corect? Greșit! Pentru că secvența funcționează pentru exact un client. În momentul conectării unui alt client care-și pune metodele la dispoziția server-ului, thread-ul creat este pur și simplu abandonat, soluție sigură pentru un dezastru. Soluția corectă este:

```
private ClientThread ct = new ClientThread();
```

și apoi:

```
public void callClient(ClientInterface ci)
        throws RemoteException
{
 ct.update(ci);
 if (!ct.isAlive()) ct.start();
 if (!ct.isInterrupted()) ct.resume();
}
```

5.b. Cîte un thread pentru fiecare client apelat de către server

Serverul gestionează un vector de threads. Când un client se deconectează, thread-ul aferent lui se opreşte şi ulterior, la conectarea unui alt client, se elimină. Server-ul stabileşte modul de lucru pentru fiecare client în parte. Secvenţa devine:

```
//   - the vector of threads (one/client)
private Vector threads = new Vector();
//   - the time to sleep, different from
          thread to thread
private static long slept = 3000;
```

iar metoda este:

```
public void callClient(ClientInterface ci)
          throws RemoteException {
synchronized (threads) {
for(int i=0; i<threads.size(); i++) {
if(!(((ClientThread)threads.elementAt(i)).isAlive()))
  {
threads.removeElementAt(i);
System.out.println("Thread " + i + " removed");
  } }
threads.addElement(new ClientThread(ci, slept));
((ClientThread)threads.lastElement()).start();
}
slept+=1000;                    // modify for next client
}
```

5.c. Diferite thread-uri în raport cu cererile clienţilor

Printr-un parametru adresat server-ului, clienţii solicită includerea într-un anumit thread. Server-ul poate accepta sau ignora cererea. Secvenţa se modifică astfel:

```
//   - the type1 thread for these clients
private ClientThreadType1 ct1 = new
          ClientThreadType1();
//   - the type2 thread for these clients (default type)
private ClientThreadType2 ct2 = new
          ClientThreadType2();
```

iar metoda este:

```
public void callClient(ClientInterface ci, int type)
          throws RemoteException
{
if(type == 1)
{
ct1.update(ci);
if (!ct1.isAlive()) ct1.start();
if (!ct1.isInterrupted()) ct1.resume();
}
else
{
ct2.update(ci);
if (!ct2.isAlive()) ct2.start();
if (!ct2.isInterrupted()) ct2.resume();
}
}
```

În fine, cazurile prezentate în paragraful precedent şi în cel de faţă se pot combina, considerând vectori de threads în raport cu condiţiile impuse de către server pentru fiecare tip de prelucrare solicitată de către clienţi, deşi un asemenea nivel de complexitate nu este necesar în aplicaţiile reale. Se demonstrează însă puterea şi eleganţa RMI.

6. Perspective de viitor

Noua versiune a Java (JDK 1.2) aflată la versiunea Beta 4, are un număr însemnat de îmbunătăţiri, care pot fi consultate în [D1].
Se constată că practic toate dezavantajele RMI faţă de CORBA, din primele versiuni (analizate extrem de pertinent în [D2]), sunt rezolvate, astfel că se poate afirma cu toată certitudinea că RMI constituie o alternativă serioasă procedeelor clasice de realizare a aplicaţiilor distribuite.

Bibliografie

A1. Cliff Berg - "How do I Browse and Dynamically Invoke Remote Objects?", (Dr. Dobb's Journal (USA) - December 1997, p.121 - on-line la adresa: http://www.ddj.com)
A2. Govin Seshadri - "How Do I Implement Callbacks with Java's RMI?", (Dr. Dobb's Journal (USA) - March 1998, p.123 - on-line la adresa: http://www.ddj.com)
A3. Piroz Mohseni - "Exploit distributed Java computing with RMI", (Part One - http://www.ncworldmag.com/ncworld/ncw-01- 1998/ncw-01-rmi.html; Part Two - http://www.nc world mag.com/ncworld/ncw -02- 1998/ncw-02-rmi2.html)
A4. Glen McCluskey - "Remote Method Invocation: Creating Distributed Java-to-Java Applications", (http://developer.java soft.com/javaInDepth/mccluskey_rmi.html)
Documentaţie despre RMI
D1. http://java.sun.com/products/jdk1.2/docs/guide/rmi/index.html
D2. Christophe Warland - "Evaluation of the CORBA technology in a context of Internet distributed financial applications written in Java", (http://www.montefiore.ulg.ac.be/~warland/tfe.html