Aplicații Integrate pentru Întreprinderi

Laborator 5

01.11.2011

Proiectarea de aplicații distribuite folosind mecanismul RMI

Scopul laboratorului îl reprezintă înțelegerea mecanismului RMI şi distribuirea unei aplicații în funcționalități implementate în componente capabile să ruleze în maşini virtuale diferite, având un comportament vizibil „la distanță".

1. Ce este RMI ?
2. Arhitectura mecanismului RMI
3. Noțiunea de clasă/obiect serializabil(ă)
4. Clase şi interfețe „la distanță"
5. Dezvoltarea unui server
6. Dezvoltarea unui client

1. Ce este RMI ?

RMI (*eng.* Remote Method Invocation) este un mecanism care extinde funcționalitatea RPC[1] permițând invocarea unei metode a unui obiect care există într-un alt spațiu de adresă, ce poate fi pe aceeaşi maşină sau pe una diferită. RMI implementează comunicația la distanță dintre programe scrise în Java.

Obiectivul RMI s-a concentrat pe a pune la dispoziția programatorilor mijloacele pentru a dezvolta aplicații distribuite în acelaşi fel în care ar scrie aplicații nedistribuite (folosind o sintaxă şi o structură semantică asemănătoare), încât folosirea de obiecte distribuite să fie similară cu utilizarea de obiecte locale, diferențele fiind sintetizate mai jos:

Concept	Obiect local	Obiect „la distanță"
definirea obiectului	se face într-o clasă Java	definirea comportamentului (exportat) al obiectului „la distanță" se face printr-o interfață care trebuie să extindă interfața `java.rmi.Remote`;
implementarea obiectului	se face în clasa Java corespunzătoare	comportamentul obiectului accesat „la distanță" este realizat printr-o clasă care implementează interfața
crearea obiectului	instanța unui obiect nou se face prin operatorul `new`	o instanță nouă pentru un obiect aflat „la distanță" se face folosind operatorul `new` pe maşina gazdă (clientul nu poate crea direct obiectul „la distanță")
accesul obiectului	se realizează direct printr-o variabilă de tip referință	un obiect „la distanță" este accesat printr-o referință care indică spre un delegat al implementării interfeței

[1] RPC = Remote Procedure Call (apel de metodă la distanță), mecanism din care RMI a fost inspirat, extinzând funcționalitatea pe care acesta o pune la dispoziție. RMI este un mecanism RPC orientat obiect.

referințe	o referință indică direct spre obiectul din heap corespunzător	o referință „la distanță” indică spre un obiect delegat (stub) alocat local în heap; obiectul delegat conține informații ce permit conectarea la obiectul „la distanță” unde este realizată implementarea metodelor
referințe active	un obiect este considerat „activ” dacă există ce puțin o referință către el	într-un mediu distribuit unde maşinile virtuale pot manifesta erori critice sau conexiunea poate fi pierdută, o referință este activă pentru un obiect la distanță dacă accesarea se face într-o anumită perioadă de timp (de închiriere); dacă pentru un obiect s-a renunțat (în mod explicit) la toate referințele sau toate referințele au expirat, obiectul „la distanță” este disponibil pentru colectarea memoriei (*eng.* garbage collection) distribuită.
finalizarea	metoda `finalize()` (în caz că este definită) se apelează înainte ca memoria aferentă obiectului să fie dezalocată (prin garbage collector)	dacă obiectul „la distanța” implementează interfața `Unreferenced`, metoda definită de această interfață este apelată când referințele la obiect sunt distruse
colectarea memoriei disponibile	un obiect spre care nu mai există referințe (locale) devine candidat pentru mecanismul de colectare a memoriei disponibile (pentru dezalocarea memoriei aferente)	există modul de colectare a memoriei distribuit care lucrează cu modulul garbage collector local, apelându-se atunci când nu există referințe realizate „la distanță” şi toate referințele locale pentru obiectul accesat „la distanță” au fost distruse
excepții	un program trebuie să trateze toate excepțiile (runtime sau alt tip)	pentru asigurarea robusteții aplicațiilor distribuite, programele trebuie să trateze excepțiile `RemoteException` care ar putea fi aruncate

2. Arhitectura mecanismului RMI

Proiectarea mecanismului RMI a vizat implementarea unui model de obiecte distribuite în Java care să se integreze în limbajul de programare şi totodată să fie compatibil cu modelul de obiecte locale.

Aplicațiile RMI sunt compuse, de cele mai multe ori, din două programe separate, server şi client.

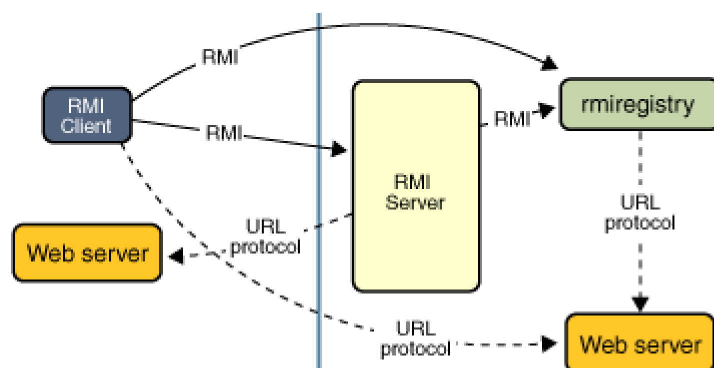
Un ***server*** crează nişte obiecte la distanță, face accesibile referințele spre obiecte şi aşteaptă clienții să invoce metodele pe care le-au definit.

Un ***client*** obține o referință spre unul sau mai multe astfel de obiecte la distanță care se găsesc pe server şi apoi invocă metodele pe ele.

RMI oferă un mecanism prin care serverul şi clientul comunică şi transferă informația în ambele sensuri. O astfel de aplicație este denumită cel mai frecvent **aplicație (cu obiecte) distribuite**.

Aplicațiile distribuite trebuie să realizeze următoarele operații: localizarea obiectelor la distanță, comunicarea cu obiectele la distanță, încărcarea definițiilor de clase pentru obiectele care sunt obținute.

1. *localizarea obiectelor la distanță* se poate face prin înregistrarea obiectelor „la distanță” la serviciul de numire RMI (rmiregistry)[2];
2. *comunicarea cu obiectele la distanță* este realizată transparent de mecanismul RMI; pentru programator, comunicarea la distanță se face similar cu invocarea metodelor locale;
3. *încărcarea definițiilor de clase pentru obiectele care sunt obținute* este posibilă concomitent cu transmiterea datelor obiectelor, având în vedere faptul că obiectele pot fi comunicate bidirecțional între client şi server.



Un exemplu de aplicație RMI

În exemplul de mai sus, serverul foloseşte serviciul de nume (rmiregistry) pentru a înregistra un obiect pe care apoi clientul îl găseşte invocând acelaşi serviciu. Ulterior, clientul apelează o metodă a obiectului „la distanță”. Definițiile de clase sunt obținute pentru obiect de pe un server web (existent), în ambele direcții (de la server spre client şi de la client spre server).

Unul dintre avantajele principale[3] puse la dispoziție de mecanismul RMI este obținerea definiției clasei unui obiect care nu este definit în maşina virtuală Java a clientului[4]. Transmiterea obiectului se face prin clasa sa, astfel încât comportamentul nu se modifică atunci când acesta este transmis către altă maşină virtuală. Astfel, noi atribute şi comportamente sunt introduse într-o maşină virtuală aflată la distanță, îmbogățind astfel comportamentul aplicației.

Ca orice aplicație Java, o aplicație distribuită folosind mecanismul RMI conține interfețe (pentru a declara metode) şi clase (pentru a implementa metodele definite în interfețe şi, posibil, pentru a implementa metode noi). Unele implementări se pot găsi doar pe unele maşini virtuale. Obiectele invocate între maşini virtuale diferite sunt denumite obiecte „la distanță” (*eng.* remote objects).

Un obiect „la distanță” implementează o interfață (vizibilă „la distanță”) având următoarele caracteristici:

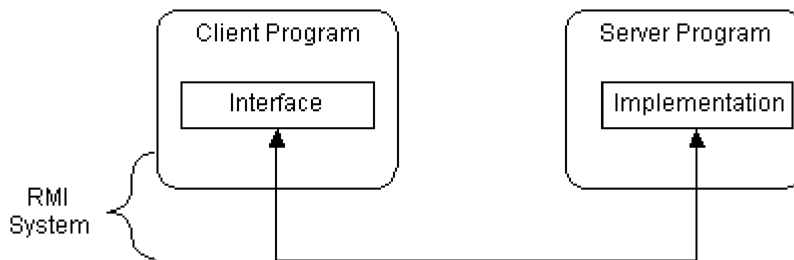
- extinde interfața `java.rmi.Remote`;
- fiecare metodă definită în interfață aruncă excepția `java.rmi.RemoteException`, în plus față de alte excepții;

[2] Alternativ, o aplicație poate transfera obiecte „la distanță” prin intermediul unor invocări ale unor metode.
[3] Este în acelaşi timp şi una dintre funcționalitățile unice ale mecanismului RMI.
[4] Toate atributele şi metodele unui obiect, disponibile până acum într-o singură maşină virtuală, pot fi transmise către o alta, posibil la distanță.

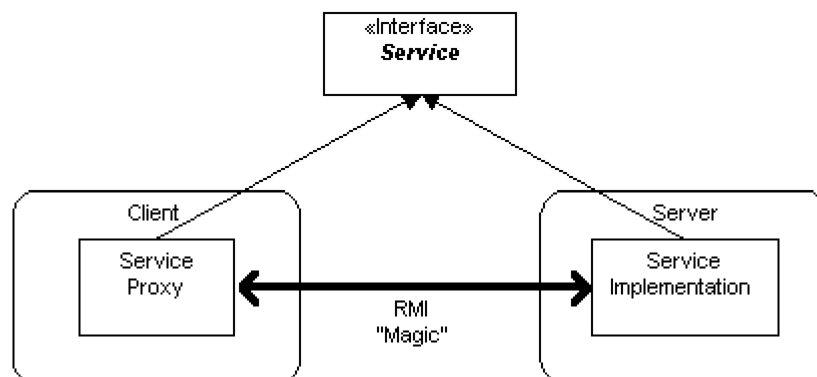
Obiectele „la distanță” sunt tratate diferit (față de obiectele locale) prin mecanismul RMI atunci când sunt transmise de la o maşină virtuală la alta.

În loc să se facă o copiere a implementării obiectului în maşina virtuală (client), mecanismul RMI transmite pentru obiectul la distanță un ciot (*eng.* stub) care are rolul de delegat (*eng.* proxy), reprezentant local al obiectului şi reprezintă referința pentru acesta pe maşina virtuală (client). Astfel, se va invoca metoda pe obiectul ciot local ce este responsabil pentru execuția metodei invocate în obiectul la distanță.

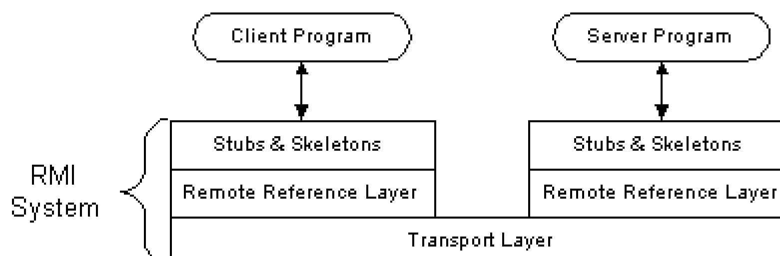
Un ciot pentru un obiect la distanță implementează acelaşi set de interfețe pe care le implementează şi obiectul la distanță, proprietate ce permite ciotului să poată fi convertit la oricare din interfețele obiectului la distanță. Totuşi, doar metodele definite în interfață sunt disponibile spre a fi invocate de client.



Arhitectura RMI (1): Clientul cunoaşte interfața obiectului „la distanță”, implementată pe server



Arhitectura RMI (2): Clientul dispune de un obiect delegat (eng. proxy) care implementează aceeaşi interfață ca obiectul „la distanță”, astfel încât pot fi apelate toate metodele obiectului



Arhitectura RMI (3): Obiectul delegat este format dintr-un ciot care conține metodele propriu-zise ale obiectului la distanță şi o referință catre acesta

Etapele dezvoltării unei **aplicații distribuite** folosind mecanismul RMI constau în:

1. proiectarea şi implementarea componentelor aplicației distribuite; Proiectarea presupune stabilirea componentelor locale şi ale componentelor accesibile la distanță.
   a. *definirea interfețelor la distanță*, prin specificarea metodelor care pot fi apelate de client (nu şi a implementării lor !); trebuie precizate tipurile obiectelor care vor fi folosite ca parametri şi valori întoarse pentru metodele folosite[5];
   b. *implementarea obiectelor „la distanță”*; obiectele la distanță implementează una sau mai multe intefețe, unele fiind accesibile doar local; trebuie implementate şi clasele (locale) care sunt folosite ca parametri sau valori întoarse pentru oricare dintre aceste metode;
   c. *implementarea clientului* poate fi realizată oricând după definirea interfețelor la distanță.
2. compilarea surselor se face apelând compilatorul `javac` atât pentru interfețele la distanță, implementarea lor, alte clase pe server, cât şi pentru clasele client[6];
3. „publicarea” claselor pentru a fi accesibile prin rețea (este vorba despre interfețele la distanță şi tipurile asociate precum şi definițiile claselor care trebuie descărcate pe client sau pe server), de regulă, prin intermediul unui server web;
4. pornirea aplicației în ordinea: serviciul de nume RMI (`rmiregistry`), serverul şi clientul.

Aplicațiile distribuite bazate pe mecanismul RMI sunt de regulă aplicații bazate pe comportamente căci sarcinile sunt încărcate dinamic de catre client fără a exista cunoştințe în prealabil despre clasele care implementează sarcinile respective.

3. Noțiunea de clasă/obiect serializabil(ă)

Tehnologia RMI foloseşte mecanismul de serializare a obiectelor din Java pentru a transmite obiecte prin valoare între maşini virtuale diferite. Pentru ca un obiect să fie considerat serializabil, clasa sa trebuie să implementeze interfața `java.io.Serializable`.

Există două tipuri de clase care pot fi folosite cu mecanismul RMI:

1. clase `Remote` ale căror instanțe pot fi folosite la distanță, obiectele instanță ale acestei clase putând fi accesate în două moduri:
   a. în spațiul de adrese în care a fost creat, obiectul fiind utilizat ca orice alt obiect (local);
   b. în alte spații de adrese, obiectul va fi utilizat prin intermediul unui delegat, care impune unele limitări în accesarea sa comparativ cu metoda anterioară, dar care în linii generale permite folosirea obiectului ca şi acum ar fi accesat local;
2. clase `Serializable` ale căror instanțe pot fi copiate între spații de adrese

[5] Dacă nu se folosesc tipuri de bază ci tipuri ale unor obiecte definite de utilizator, acestea trebuie specificate de asemenea în acest moment.
[6] Crearea claselor ciot prin apelarea compilatorului `rmic`, necesară până la versiunile mai mici de Java 5, nu mai este obligatorie.

Dacă un obiect serializabil este dat ca parametru (sau valoare întoarsă) pentru un apel de metodă la distanță, valoarea obiectului va fi copiat dintr-un spațiu de adrese în altul. De asemenea, dacă un obiect la distanță este transmis ca parametru (sau valoare întoarsă), delegatul acestuia va fi copiat dintr-un spațiu de adrese în altul.

Majoritatea claselor standard sunt serializabile, deci o subclasă ale oricărora dintre acestea este de asemenea serializabilă. Orice informație dintr-o clasă serializabilă ar trebui să fie serializabilă.

Folosirea unui obiect serializabil în apelul unei metode la distanță este foarte simplă. Se specifică obiectul dat ca parametru sau valoare întoarsă, tipul acestuia trebuind să fie o clasă serializabilă, fiind necesar ca atât clientul cât şi serverul să aibă acces la definiția clasei serializabile care este utilizat, descărcarea definițiilor claselor serializabile de pe o maşină virtuală pe alta trebuind să fie specificată prin politici de securitate.

Exemplu

În cele ce urmează, implementarea unei aplicații distribuite folosind mecanismul RMI va fi ilustrată pe cazul particular al unui program cu un comportament asemănător cu Doodle (www.doodle.com).

Doodle®

Doodle™ este o aplicație care permite planificarea unor activități în funcție de necesități specificate în mod particular.

În cadrul laboratorului, vom implementa o aplicație care simulează această funcționalitate, având ca scop alocarea de intervale de timp pentru studenții care vor să prezinte temele la *„Aplicații Integrate pentru Întreprinderi”*.

Aplicația va avea arhitectura client-server şi are definite semnăturile pentru metodele pe care va trebui să le implementați.

Se specifică nişte date calendaristice cu intervale orare aferente în care este posibilă prezentarea temelor, informații reținute de către aplicația server (`ProgramareServer.java`).

Fiecare student (aplicația client – `ProgramareClient.java`) face o cerere pentru o perioadă de timp preferată (având orice durată). Dacă perioada de timp se încadrează în orarul specificat şi intervalul dorit este liber (nu a fost alocat deja altui student), atunci cererea este aprobată şi intervalul marcat ca ocupat. Este reținut totodată şi identificatorul studentului care a realizat programarea, astfel încât tot acesta să aibă posibilitatea de a elibera intervalul ocupat, dacă situația o impune.

Informațiile cu privire la orar sunt specificate într-un fişier text pe server (`orar.txt`) şi încărcate în momentul lansării în execuție a acestuia. Nu se cere menținerea persistenței datelor între rulări diferite ale serverului. Cererile sunt specificate de către studenți prin introducerea de la tastatură având acelaşi format cu care au fost precizate în fişier: `ZZ LL AAAA O1 M1 O2 M2`.

4. Clase şi interfeţe „la distanţă”

Protocolul care permite ca serverul să fie interogat cu cereri de către client care să fie rulate apoi de server iar rezultatul să fie întors la client este realizat prin intermediul unei interfețe care descrie funcționalitatea care poate fi accesată la distanță.

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.*;

public interface Programare extends Remote
{
      public ArrayList<Interval> obtineOrar() throws RemoteException;
      public ArrayList<Interval> obtineIntervaleOcupate()
                                 throws RemoteException;
      public ArrayList<Interval> obtineIntervaleLibere()
                                 throws RemoteException;
      public boolean ocupaInterval (int student, Interval interval)
                     throws RemoteException;
      public boolean elibereazaInterval (int student, Interval interval)
                     throws RemoteException;
}
```

Prin extinderea interfeței `java.rmi.Remote` se specifică faptul că metodele declarate de interfața `Programare` vor putea fi apelate din altă maşină virtuală Java. Orice obiect care implementează interfața `Programare` va putea fi considerat un obiect „la distanță”.

Metodele definite în cadrul acestei interfețe vor putea fi apelate din cadrul altei maşini virtuale („la distanță”), prin urmare este obligatoriu să genereze excepții de tip `java.rmi.RemoteException`[7].

Se observă că metodele `ocupaInterval()` şi `elibereazaInterval()` primesc şi un parametru de tip `Interval`, întoarcând un rezultat de tip `boolean`. Pentru că `Interval` nu reprezintă un tip de bază, el trebuie definit tot acum.

```
import java.io.Serializable;
import java.util.GregorianCalendar;

public class Interval implements Serializable
{

      private static final long serialVersionUID = 1024L;

      GregorianCalendar inceput;
      GregorianCalendar sfarsit;

      public Interval()
      {
            inceput = new GregorianCalendar();
            sfarsit = new GregorianCalendar();
      }
```

[7] O astfel de excepție este aruncată de sistemul RMI dintr-un apel de metodă la distanță pentru indicarea situației în care a intervenit o eroare de comunicare sau de protocol. Codul sursă care apelează metode „la distanță” trebuie să trateze o astfel de excepție (prin prinderea ei sau transmiterea ei mai departe).

```
      public Interval(GregorianCalendar inceput, GregorianCalendar sfarsit)
      {     this.inceput = inceput; this.sfarsit = sfarsit;    }
      public void stabilesteInceput(GregorianCalendar inceput)
      {     this.inceput = inceput;    }
      public GregorianCalendar obtineInceput()
      {     return inceput;    }
      public void stabilesteSfarsit(GregorianCalendar sfarsit)
      {     this.sfarsit = sfarsit;    }
      public GregorianCalendar obtineSfarsit()
      {     return sfarsit;    }
}
```

Câmpul `serialVersionUID` e folosit în procesul de deserializare al obiectelor încât dacă atributul obținut nu corespunde cu cel specificat în definiția clasei, este aruncată o excepție de tipul `InvalidClassException`.

Obiectele de tip `Interval` vor fi transmise între client şi server astfel încât ele trebuie să fie serializabile (asigurând astfel consistența informațiilor reținute în obiect indiferent de formatul de reprezentare din maşinile virtuale). De aceea, clasa `Interval` implementează interfața `Serializable` (din pachetul `java.io`) şi pune la dispoziție un constructor de copiere şi metode de tip setter/getter[8].

Compilarea claselor se face în mod obişnuit:

```
> javac Programare.java Interval.java
```

Deoarece vom avea nevoie de cele două clase atât pe client cât si pe server este bine să le includem într-o arhivă Java (de tip jar) adăugată în classpath atât la compilare cât şi la rulare.

```
> jar cvf programare.jar Programare.class Interval.class
```

5. Dezvoltarea unui server

Serverul trebuie să implementeze interfața „la distanță” vizibilă în rețea astfel încât comportamentul obiectelor care vor fi accesate de client să fie definit (obiectele „la distanță” sunt obținute prin instanțierea clasei Server). Pe server pot fi definite şi alte metode (în afara celei impuse de interfață) care vor putea fi accesate local, între care constructorul clasei şi metoda `main`.

Argumentele sau valorile întoarse din metodele „la distanță” (definite în interfață) pot fi de orice tip, inclusiv obiecte locale, obiecte „la distanță” sau chiar tipuri primitive de date. Astfel, orice entitate având orice tip poate fi transmisă către sau de la o metodă „la distanță” cât timp aceasta este o instanță a unui tip care este fie primitiv, fie obiect „la distanță” sau obiect local serializabil (aşa cum s-a precizat şi mai sus). Alte tipuri de obiecte, cum ar fi fire de execuție sau descriptori de fişier, având sens doar în spațiul de adrese în care rulează, nu pot să fie utilizați ca argumente sau valori întoarse în metode „la distanță”. Totuşi, majoritatea claselor din pachetele `java.lang` şi `java.util` implementează interfața `java.io.Serializable`.

[8] Metodele nu mai trebuie să arunce excepția `java.rmi.RemoteException` pentru că nu este extinsă interfața `Remote`.

Obiectele „la distanță" sunt transmise prin referință (un ciot care este delegatul la nivelul clientului şi care implementează setul complet de interfețe accesibile prin rețea specificate de obiect).

Sunt vizibile doar metodele definite în interfața „la distanță" pentru obiectul respectiv. Metodele definite pentru obiect care nu sunt definite în interfață nu vor putea fi accesate la distanță.

Orice modificare realizată asupra stării obiectului prin apelurile la distanță sunt reflectate în obiectul original (de pe server).

Obiectele locale sunt transmise prin valoare (este realizată o copie folosind mecanismul de serializare). Sunt copiate toate câmpurile care nu au precizat atributul `static` sau `transient`[9]. Orice modificare realizată în starea obiectului sunt reflectate doar în copie, dar nu şi în instanța originală. Similar, schimbările asupra stării obiectului în instanța originală nu vor fi reflectate în copie.

```
import java.rmi.RemoteException;
import java.rmi.registry.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.*;

public class ProgramareServer implements Programare
{
    public ProgramareServer() {    }

    public ArrayList<Interval> obtineOrar() throws RemoteException
    {    ...    }

    public ArrayList<Interval> obtineIntervaleOcupate()
                               throws RemoteException {    ...    }

    public ArrayList<Interval> obtineIntervaleLibere()
                               throws RemoteException {    ...    }

    public boolean ocupaInterval (int student, Interval interval)
                   throws RemoteException {    ...    }

    public boolean elibereazaInterval (int student, Interval interval)
                   throws RemoteException {    ...    }

    public static void main (String[] args)
    {
        if (System.getSecurityManager() == null)
            System.setSecurityManager(new SecurityManager());

        try
        {
            String nume = "ServiciuProgramare";
            Programare serviciu = new ProgramareServer();
            Programare delegat =
                       (Programare)
                       UnicastRemoteObject.exportObject(serviciu, 0);
            Registry registry = LocateRegistry.getRegistry();
            registry.rebind(nume, delegat);
        }
        catch (Exception e) {    e.printStackTrace();    }
    }
}
```

[9] Modul în care este realizat comportamentul implicit de serializare poate fi modificat.

Metoda `main` este cea mai complexă, fiind folosită pentru a porni serverul, realizând totodată şi inițializările necesare pentru a pregăti serverul să accepte conexiuni dinspre clienți. Metoda nu poate fi apelată din altă maşină virtuală, pentru că nu extinde interfața `Remote`. Fiind statică, ea este asociată cu clasa şi nu cu obiectul.

1. Este creat şi instalat **un sistem de securitate** care protejează accesul la resursele sistemului de cod sursă (descărcat) care rulează în maşina virtuală Java. Acest sistem de securitate stabileşte dacă codul sursă poate să acceseze sistemul local de fişiere sau poate realiza alte operații privilegiate.

În situația în care aplicația distribuită care foloseşte tehnologia RMI nu dispune de un sistem de securitate, nu vor fi descărcate clase (altele decât din sistemul de fişiere specificat în classpath) pentru obiectele primite ca argumente sau valori întoarse.

Se asigură totodată faptul că operațiile realizate de codul descărcat sunt conforme cu o politică de securitate.

2. Se creează o **instanță** a clasei `ProgramareServer` şi este exportată spre RMI folosindu-se instrucțiunea `UnicastRemoteObject.exportObject`[10] astfel încât obiectul să poată fi apelat de către clienți de la distanță. La întoarcerea din execuția acestei metode, obiectul poate procesa apeluri „la distanță".

Este întors un ciot (*eng.* stub) care are tipul `Programare`, deci tipul interfeței şi nu al clasei (pentru că ciotul implementează doar interfața „la distanță").

3. Înainte ca un client să poată invoca metodele unui obiect „la distanță", trebuie să obțină referința spre acesta, operație ce poate fi realizată în acelaşi fel în care este obținută o referință spre un obiect într-o aplicație (ca valoare întoarsă pentru o metodă sau ca un câmp al unei structuri de date care conține referința).

Mecanismul RMI pune la dispoziție un tip particular de obiect „la distanță" serviciul de nume (RMI registry), pentru găsirea de referințe către alte obiecte de acest tip prin specificarea unui nume. Un astfel de sistem este în mod obişnuit folosit doar pentru găsirea primului obiect solicitat de client, putând ajuta la găsirea altor obiecte.

Interfața `java.rmi.registry.Registry` conține specificațiile pentru **înregistrarea** şi găsirea de obiecte „la distanță" în serviciul de nume. Clasa `java.rmi.registry.LocateRegistry` pune la dispoziție metode statice pentru identificarea unei referințe la distanță într-o rețea anume (specificată prin adresă şi port). Metodele crează referința către obiectul „la distanță" conținând adresa de rețea specificată fără a se realiza comunicare la distanță[11].

După ce obiectul „la distanță" a fost înregistrat de sistemul de nume, clienții de pe orice maşină îl pot căuta după nume, îi pot obține referința şi pot apela metodele „la distanță" de pe el[12].

Metoda `rebind` reprezintă un apel la distanță către serviciul de nume RMI şi poate avea ca rezultat aruncarea unei excepții de tip `RemoteExcption`.

---

[10] Cel de-al doilea argument al metodei `exportObject` reprezintă un port TCP utilizat pentru ascultarea apelurilor metodelor la distanță de către client. Valoarea 0, folosită în exemplul de față specifică utilizarea unui port anonim, ales de RMI sau de sistemul de operare.

[11] Clasa `java.rmi.registry.LocateRegistry` pune la dispoziție metode statice pentru crearea unui serviciu de nume nou în maşina virtuală Java.

[12] Serverele de pe o maşină pot partaja acelaşi serviciu de nume sau fiecare aplicație (de tip server) poate să îşi creeze şi să utilizeze propriul său sistem de nume.

Metoda `LocateRegistry.getRegistry` este apelată fără parametri, obținându-se o referință la serviciul de nume de pe maşina locală şi portul 1099, definit implicit[13].

Când se face apelul la distanță către serviciul de nume, se obține un obiect de tip ciot (în loc de copia obiectului) pentru că obiectele care implementează interfețe de tip `Remote` rămân în cadrul maşinii virtuale Java în care au fost create. Astfel, atunci când un client realizează o căutare în serviciul de nume al serverului, se întoarce o copie a obiectului de tip ciot, obiectele „la distanță” fiind transmise deci prin referință.

Din motive de securitate, o aplicație poate (re)construi sau distruge legătura dintre o referință a unui obiect la distanță şi serviciul de nume care rulează pe aceeaşi maşină, prevenind situația în care o altă maşin ar înlătura sau suprascrie intrările într-un serviciu de nume.

Operația de căutare (`lookup`) poate fi realizată de pe orice maşină, locală sau „la distanță”.

Excepțiile ce pot fi aruncate de metoda `main` sunt de tip `RemoteException`, generate fie de metoda `UnicastRemoteObject.exportObject` sau de apelul `rebind`. Se poate realiza recuperarea din eroare (în caz că o excepție a fost aruncată) prin reapelarea metodei care a generat-o sau prin folosirea unui alt server.

După realizarea acestor operații, metoda `main` se încheie. Nu este necesară definirea unui fir de execuție care să țină serverul în starea de rulare, pentru că atât timp cât există o referință către un obiect de tip `ProgramareServer` într-o maşină virtuală (locală sau la distanță), el nu va fi distrus de garbage collector. Mecanismul RMI menține activ procesul asociat obiectului `ProgramareServer`, pentru că există o legătură către el în serviciul de nume, accesibilă la distanță. Obiectul va fi distrus atunci când legătura dintre el şi serviciul de nume va fi eliberată şi nu vor mai exista clienți la distanță care să aibă referințe către el.

Compilarea claselor se face în mod obişnuit:

```
> javac –cp programare.jar ProgramareServer.java
```

Lansarea în execuție nu poate fi realizată decât după:

- pornirea serviciului de nume, care permite clienților să obțină o referință către un obiect la distanță[14]
  - Windows
    ```
    > start rmiregistry [port]
    ```
  - Unix
    ```
    # rmiregistry [port] &
    ```
- specificarea unei politici de securitate, într-un fişier numit `policy[.txt]`, codul sursă obținând permisiunile de care are nevoie pentru a putea rula:
    ```
    grant codeBase "file:///<cale_absoluta>\\Server" {
        permission java.security.AllPermission;
    };
    ```

[13] Dacă serviciul de nume a fost creat pe un alt port, se foloseşte o metodă supraîncărcată care primeşte un parametru de tip `int` care specifică portul.

[14] Parametrul port este opțional, el poate să nu fie menționat, caz în care serviciul de nume foloseşte portul implicit 1099.

Toate permisiunile sunt acordate claselor din calea locală a aplicației pentru că acesta are un grad de încredere limitat, dar nu se acordă drepturi pentru codul sursă descărcat din alte maşini virtuale.

Lansarea în execuție a serverului se face prin comanda[15]:

```
> java -cp .;programare.jar
       -Djava.rmi.server.codebase=file:///<cale_absoluta>\programare.jar
       -Djava.rmi.server.hostname=localhost
       -Djava.security.policy=policy
       ProgramareServer
```

Proprietatea `java.rmi.server.codebase` specifică locația de unde pot fi descărcate definiții pentru clasele provenind de la server[16].

Proprietatea `java.rmi.hostname` specifică numele maşinii sau adresa care urmează a fi completată în obiectele de tip ciot care vor fi exportate. Aceeaşi valoare va fi folosită de clienți când vor încerca să apeleze metodele la distanță. Implicit, implementarea RMI foloseşte adresa IP indicată de `java.net.InetAddress.getLocalHost`. Totuşi, câteodată o astfel de adresă nu este întotdeauna potrivită pentru toți clienții şi un nume (al maşinii) este mai adecvat Pentru ca numele sa fie vizibil de toți clienții, trebuie configurată proprietatea `java.rmi.server.hostname`.

Proprietatea `java.rmi.policy` specifică politica care conține permisiunile ce vor fi acordate.

6. Dezvoltarea unui client

Codul care apelează metodele unui obiect de tip `Programare` trebuie să obțină o referință către acesta.

Ulterior, va creea un obiect de tip `Interval` (eventual din parametrii specificați în linie de comandă) pe care îl va transmite ca argument metodelor `ocupaInterval` sau `elibereazaInterval`.

1. Ca şi în cazul serverului, inițial se va instala un **sistem de securitate**, necesar pentru că în procesul de obținere a referinței către obiectul ciot corespunzător obiectului la distanță se poate întâmpla să se descarce definiții de clase de la server, lucru care este posibil doar în condițiile existenței unui sistem de securitate.
2. Clientul va **căuta** obiectul „la distanță” specificând un nume (acelaşi folosit de server pentru a înregistra obiectul la serviciul de nume) – în acest sens, e folosită de asemenea metoda `LocateRegistry.getRegistry` pentru a obține referința la sistemul de nume RMI care rulează pe maşina server, parametrul cu care este apelat reprezentând numele maşinii[17] pe care rulează obiectul de tip `Programare`. Apoi este utilizată metoda `lookup` pentru a căuta obiectul la distanță prin nume în sistemul de nume de pe maşina server.

---

[15] Pe Linux, caracterul ; va fi înlocuit cu :.

[16] Dacă în locul unei arhive se specifică o ierarhie de directoare, aceasta trebuie încheiată prin caracterul ’/’.

[17] În mod implicit, se consideră că sistemul de nume RMI ascultă pe portul 1099. În plus, poate fi folosită o metodă supraîncărcată care să specifice şi portul, prin specificarea unui parametru de tip `int`.

3. Se va crea un obiect de tip Interval (eventual din parametrii specificați în linia de comandă) şi se vor **apela metodele la distanță**, afişându-se rezultatul.

```
public class ProgramareClient
{
	...

	public static void main (String[] args)
	{
		if (System.getSecurityManager() == null)
			System.setSecurityManager(new SecurityManager());

		try
		{
			String nume = "ServiciuProgramare";
			Registry registry = LocateRegistry.getRegistry(args[0]);
			Programare programare =
				(Programare)registry.lookup (nume);

			int student = 0;
			Interval interval = new Interval (...);

			if (programare.ocupaInterval(student,interval))
				System.out.println ("succes");
			else
				System.out.println ("insucces");
		}
		catch (Exception e) {	e.printStackTrace();	}
	}
}
```

Compilarea claselor se face în mod obişnuit:

```
> javac -cp programare.jar ObtineProgramare.java
```

Lansarea în execuție nu poate fi realizată decât după:

- pornirea serviciului de nume RMI pe maşina server
- lansarea în execuție a aplicației server ServiciuProgramare
- specificarea unei politici de securitate, într-un fişier numit `policy[.txt]`, codul sursă obținând permisiunile de care are nevoie pentru a putea rula:

```
grant codeBase "file:///<cale_absoluta>\\Client" {
    permission java.security.AllPermission;
};
```

Lansarea în execuție a serverului se face prin comanda:

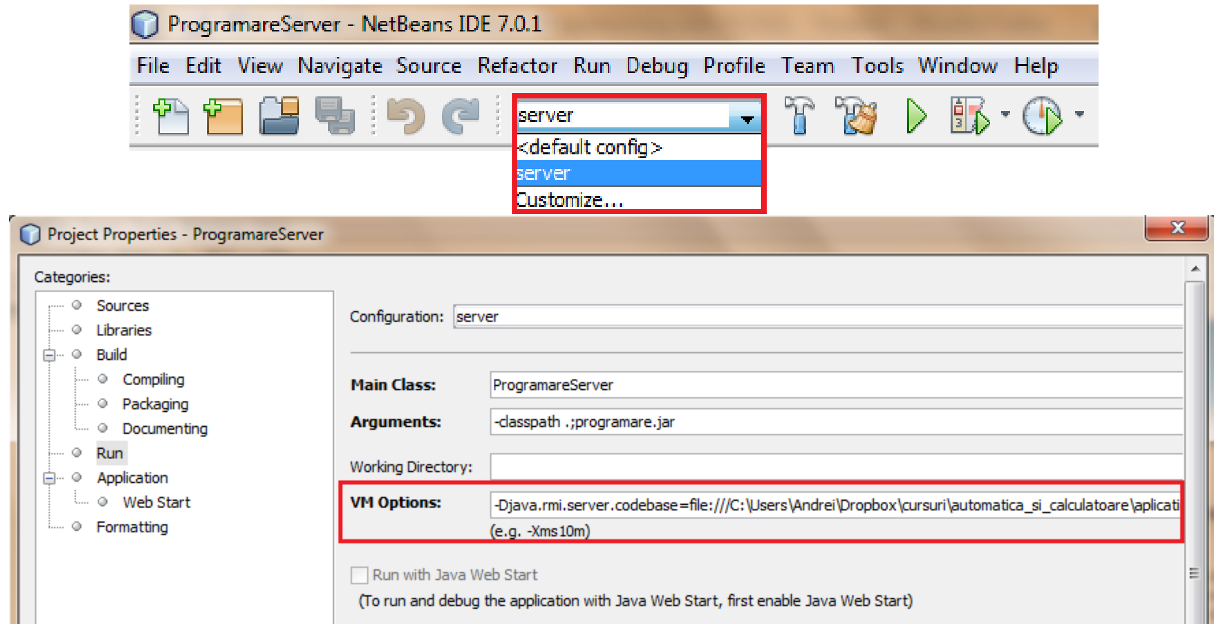
```
> java -cp .;programare.jar
       -Djava.rmi.server.codebase=file:///<cale_absoluta>\
       -Djava.security.policy=policy
       ProgramareClient <adresa>
```

Se observă că prin proprietatea `java.rmi.server.codebase` s-a precizat locația unde clientul îşi defineşte clasele, iar prin proprietatea `java.security.policy` s-a specificat fişierul care conține permisiunile acordate pentru diferite coduri sursă.

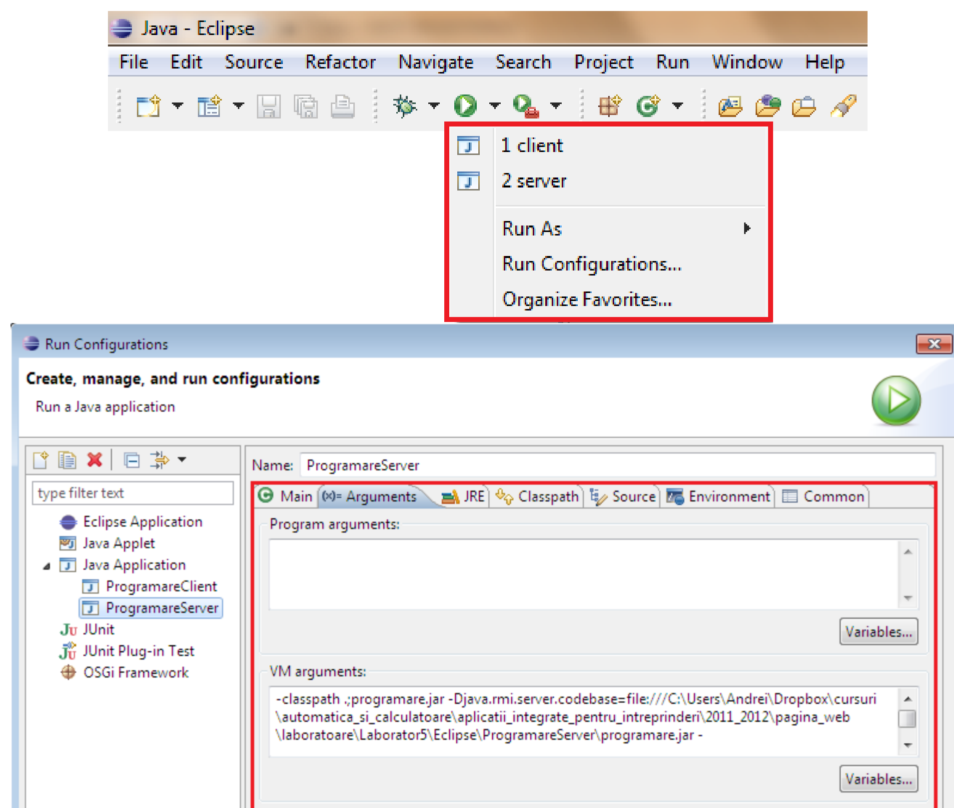
Activitate de Laborator

[0p] 1. Să se pornească registrul de nume `rmiregistry` prin apelarea scripturilor `pornesteregistrudenume [.bat|.sh]`.

[0p] 2. Să se modifice căile absolute din fişierele `policy`, respectiv din configurațiile de rulare (Eclipse / Netbeans) pentru rularea serverului, respectiv a clientului.



Modificarea configurației de rulare în NetBeans



Modificarea configurației de rulare în Eclipse

[1p] 3. Să se încarce orarul în constructorul clasei `ProgramareServer` cu datele conținute de fişierul `orar.txt`.
[1p] 4. Să se implementeze metoda `obtineOrar` a clasei `ProgramareServer`.
[1p] 5. Să se implementeze metoda `obtineIntervaleOcupate` a clasei `ProgramareServer`.
[3p] 6. Să se implementeze metoda `obtineIntervaleLibere` a clasei `ProgramareServer`.
[2p] 7. Să se implementeze metoda `ocupaInterval` a clasei `ProgramareServer`.
[2p] 8. Să se implementeze metoda `elibereazaInterval` a clasei `ProgramareServer`.
[0p] 9. Să se testeze că metodele `ocupaInterval` şi `elibereazaInterval` au fost implementate corect.
BONUS.
[1p] 10. Să se modifice comportamentul metodei `elibereazaInterval` astfel încât să fie permisă eliberarea unei porțiuni din intervalul ocupat inițial.