Cours 8 : Programmation répartie

- Appels distants
- RMI
  - appels distants et concurrents
  - mécanisme de rappel
  - Dgc
  - mécanisme d’activation

Remote Procedure Call

- appel de procédures ou fonctions distantes
- des difficultés :
  - transmission des paramètres :
    - par référence / par copie
    - différentes représentations : XDR
    - typage non sûr
  - localisation des serveurs/services :
    - en dur
    - dynamique : PORTMAPPER
  - génération du code d'encapsulation : RPCGEN
  - panne client ou serveur

portmap et rpcinfo

- enregistrement de services

```
rpcinfo -p hydrogene.pps.jussieu.fr
   program no_version protocole  no_port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100021    1   udp  32768  nlockmgr
    100021    3   udp  32768  nlockmgr
    100021    4   udp  32768  nlockmgr
    100021    1   tcp  32768  nlockmgr
    100021    3   tcp  32768  nlockmgr
    100021    4   tcp  32768  nlockmgr
    100007    2   udp    705  ypbind
    100007    1   udp    705  ypbind
    100007    2   tcp    708  ypbind
    100007    1   tcp    708  ypbind
    100003    2   udp   2049  nfs
    100003    3   udp   2049  nfs
    100003    4   udp   2049  nfs
    100003    2   tcp   2049  nfs
    100003    3   tcp   2049  nfs
```

RPCGEN (1)

- un IDL + générateur des codes souches (clients) / squelettes (serveur)
- add.x :

```
struct intpair {
int a;
int b;
};

program ADDPROG {
version ADDVERS {
int ADDPROC_ADD(intpair) = 1;
int ADDPROC_MULT(intpair) = 2;
} = 1;
} = 55555;
```

RPCGEN (2)

- Génération

```
$ rpcgen -C add.x
$ ls
add.h
add.x
add_svc.c
add_clnt.c
add_xdr.c
```

Exemple : code serveur (1)

```
cc    -c -o add_svc.o add_svc.c
cc    -c -o proc.o proc.c
cc -o server add_svc.o add_xdr.o proc.o
```

```
#include <rpc/rpc.h>
#include "add.h"

int *addproc_add_1_svc(intpair *s, struct svc_req *rqstp)
{
static int r;
r = s -> a + s -> b;
return &r;
}

int *addproc_mult_1_svc(intpair *s, struct svc_req *rqstp)
{
static int r;
r = s -> a * s -> b;
return &r;
}
```

Exemple : code client (2)

```
cc    -c -o client.o client.c
cc    -c -o add_clnt.o add_clnt.c
cc    -c -o add_xdr.o add_xdr.c
cc -o client client.o add_clnt.o add_xdr.o
```

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "add.h"

main(argc,argv)
char **argv;
{
intpair s;
int *r;
CLIENT *cl;

if (argc != 4)
fprintf(stderr,"Use: client host <int> <int>\n"),
exit(1);
...
```

Exemple : code client (3)

```
...
        if ((cl = clnt_create(argv[1],ADDPROG,ADDVERS,"tcp")) == NULL)
                clnt_pcreateerror(argv[1]), exit(1);
s.a = atoi(argv[3]);
s.b = atoi(argv[3]);

if ((r = addproc_add_1(&s,cl)) == NULL)
clnt_perror(cl,argv[1]), exit(1);
printf("From %s : %d + %d = %d\n",argv[1],s.a,s.b,*r);

if ((r = addproc_mult_1(&s,cl)) == NULL)
clnt_perror(cl,argv[1]), exit(1);
printf("From %s : %d * %d = %d\n",argv[1],s.a,s.b,*r);

exit(0);
}
```

Exemple : exécution (4)

```
$ ./server
$ ./client localhost 12 23
From localhost : 12 + 23 = 35
From localhost : 12 * 23 = 276
$ ./client localhost 11 23
From localhost : 11 + 23 = 34
From localhost : 11 * 23 = 253
```

```
$ rpcinfo -p
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
     44444    1   udp  51133
     44444    1   tcp  53879
```

Pannes

- demande : réémission sur expiration de temporisation
- réponse : réémission sur expiration de temporisation, avec réexécution de la requête

Objets distribués

Possibilités:

clients/serveurs + persistance $\Rightarrow$ transport d'objets par copie (et création de nouvelles instances)

Références distantes:

références de plusieurs endroits du réseau au même objet pour invoquer ses méthodes et/ou modifier ses variables d'instances.

Difficultés de mise en œuvre

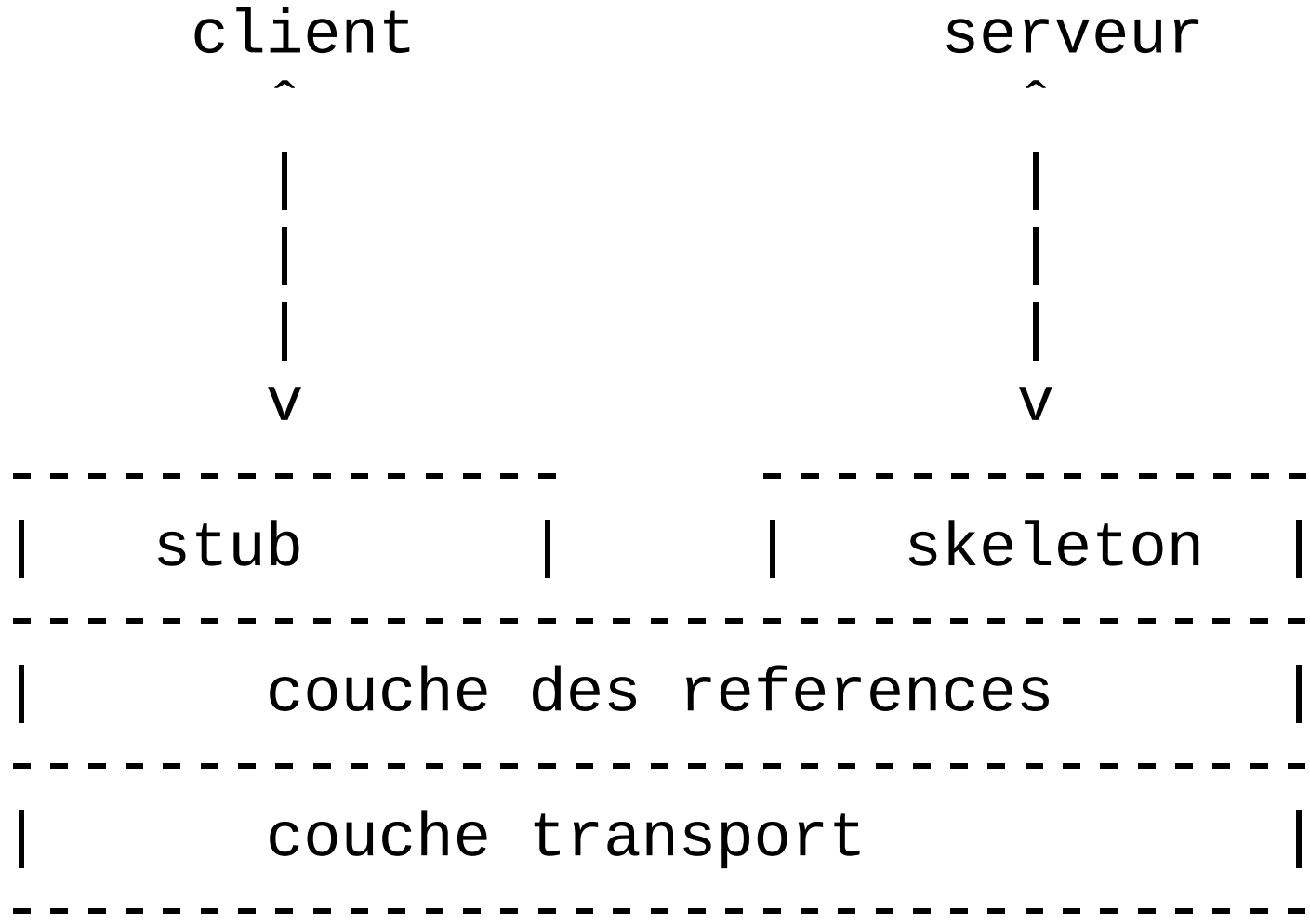
- transparence référentielle
- Garbage Collector
- typage des objets copiés
- exceptions distantes

le jdk 1.1 offre un mécanisme simple nommé RMI (*Remote Method Invokation*) permettant de manipuler des objets distants.

RMI

- objet distant : sur une autre machine virtuelle Java
- garde le même modèle objet
- utilisation d'interface étendant `Remote`
- passage de paramètre et résultat par copie (`Serializable`)

Structure générale



Serveur

Le serveur s’en trouve compliqué sur les services suivants :

- GC des objets distribués
- réplication d’objets distants
- activation d’objets persistants

Paquetages

Les paquetages utiles sont :

- `java.rmi` : pour le coté client
- `java.rmi.server` : pour le coté serveur
- `java.rmi.registry` : pour l'utilisation d'un service d'enregistrement et de référentiel des objets serveurs,
- `java.rmi.dgc` : pour le GC distribué.

L'interface d'objets distants étend l'interface `Remote` en ajoutant les méthodes désirées qui peuvent déclencher des `RemoteException`.

Exemple : Points distants

Interface:

```
import java.rmi.*;
public interface PointRMI extends Remote {

    void moveto (int a, int b)    throws RemoteException;

    void rmoveto (int dx, int dy) throws RemoteException;

    void affiche()    throws RemoteException;

    double distance() throws RemoteException;
}
```

Implantation d'interfaces distantes

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class PointD extends UnicastRemoteObject implements PointRMI {

    int x,y;
    PointD(int a, int b) throws RemoteException {x=a;y=b;}
    PointD() throws RemoteException {x=0;y=0;}

    public void moveto (int a, int b) throws RemoteException
    { x=a; y=b;}

    public void rmoveto (int dx, int dy) throws RemoteException
    { x = x + dx; y = y + dy;}

    public void affiche()   throws RemoteException
    { System.out.println("(" + x + "," + y + ")");}

    public double distance()  throws RemoteException
    { return Math.sqrt(x*x+y*y);}
}
```

Compilation

- interface et implantation par : `javac`
- *stubs* et *skeletons* par : `rmic` de la manière suivante :

`rmic PointD` qui créera les fichiers `PointD_Stub.class` et `PointD_Skel.class`.

création et enregistrements

Le serveur de points lui va créer des instances de `PointD` et les enregistrer (`rebind`) auprès du démon (`rmiregistery`) qui gère le protocole rmi.

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class Creation {
    public static void main (String args[]) {

    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
     }
     try {
    PointD p0 = new PointD();
    PointD p1 = new PointD(3,4);
    Naming.rebind("//localhost/point0",p0);
    Naming.rebind("//localhost/point1",p1);
    System.out.println("Objets distribues 'p0' " +
       "et 'p1' sont enregistres");
     }
     catch (Exception e) { e.printStackTrace(); }
  }
```

Commentaires

- le “security manager” garantit qu’il n’y aura pas d’opérations illégales.
- l’enregistrement nécessite de nommer chaque instance

La classe `Naming` permet de nommer des objets pour leur enregistrement sur le serveur (`bind`, `rebind`, `unbind`, `list`, `lookup`).

Compilation classique avec `javac`.

un client

```
import java.rmi.*;
public class Client {
  public static void main( String argv[]) {
    String machine = argv[0];
    String port = argv[1];
    String url0="rmi://"+machine+":"+port+"/point0";
    String url1="rmi://"+machine+":"+port+"/point1";
     try {
       PointRMI p0 = (PointRMI)Naming.lookup(url0);
       PointRMI p1 = (PointRMI)Naming.lookup(url1);
       p0.affiche(); p1.affiche();
       p0.rmoveto(7,12);
       p1.rmoveto(5,6);
       p0.affiche();    p1.affiche();
       if (p0.distance() == p1.distance())
         System.out.println("c'est le hasard");
       else System.out.println("on pouvait parier");
    }
    catch (Exception e) {
      System.err.println("exception : " +  e.getMessage());
      e.printStackTrace(); } } }
```

Lancement

- Service :
  `rmiregistry &`
- Serveur :
  `java -Djava.security.policy=java.policy Creation`
- Client :
  `java -Djava.security.policy=java.policy Client chrome.pps.jussieu.fr 1099`

Le fichier `java.policy` indique les opérateions autorisées :

```
grant {
    permission java.net.SocketPermission "*:1024-65535",
        "connect,accept";
    permission java.net.SocketPermission "*:80", "connect";
};
```

Exécution

1ère exécution:

```
(0,0)
(3,4)
(7,12)
(8,10)
on pouvait parier                  // client
```

2ème exécution:

```
(7,12)
(8,10)
(14,24)
(13,16)
on pouvait parier                // client
```

Exceptions

Si le démon ne fonctionne plus au moment de l'utilisation d'un objet distant, on récupère l'exception dans le `catch` et on affiche les messages suivants :

```
exception : Connection refused to host;
...
```

et si l'objet demandé n'est pas enregistré, la suivante :

```
exception : point0
java.rmi.NotBoundException: point0
```

port de communication

Par défaut, le port du service rmi est le 1099. Il est possible de changer de numéro. Pour cela le serveur d'enregistrement doit être lancé avec le paramètre du port :

```
rmiregistry 2000&
```

et il faut indiquer ce nouveau port aux URL employées :

```
//chrome.pps.jussieu.fr:2000
```

Concurrence des appels

Que se passe-t-il quand il y a plusieurs invocations de méthodes sur un même objet?

- du coté du client : appel bloquant;
- du coté serveur : queue ou thread? : cela dépend si les appels proviennent de la même JVM ou non
  - même JVM : en queue
  - différentes JVM : threads

Mécanisme de rappel

- Objectif : rendre l'appel distant non bloquant
- Problème : comment récupérer le résultat?
- Possibilité : Fournir un objet distant du coté du client qui récupèrera le résultat.
- Difficultés : objet (serveur) distant temporaire + consultation de celui-ci pour accéder au résultat quand il sera là.

Exemple : Points distants

- une interface `RappelRMI` contenant une méthode pouvant stocker la valeur calculée du coté du serveur RMI sur l'objet serveur distant temporaire du client
- une interface `MRPointRMI` des points distants dont la méthode surchargée `distance` n'est plus bloquante
- une classe `Rappel` : pour le serveur temporaire implantant les interfaces `RappelRMI` et `Unreferenced` pour annuler l'objet temporaire.
- une classe `MRPointD` qui hérite de `PointD` et surcharge `distance` en appelant un *thread* instance de `Calcul`.
- un nouveau serveur `CreationN` qui stocke des instances de `MRPointD`
- un nouveau client `ClientN` qui appelle la méthode `distance` non bloquante.

Les interfaces

- RappelRMI :

```
import java.rmi.*;

public interface RappelRMI extends Remote {
   void put(double r) throws RemoteException;
}
```

- MRPointRMI :

```
import java.rmi.*;

public interface MRPointRMI extends PointRMI {
    double distance  (RappelRMI r) throws RemoteException;
}
```

La classe Rappel

```
import java.rmi.*;
import java.rmi.server.*;
public class Rappel  extends UnicastRemoteObject implements RappelRMI, Unr
    private double result = 0.0;
    private boolean f = false;
    public Rappel() throws RemoteException{}
    public void finish () {f=true;}
    public boolean is_finished () {return f;}
    public void put(double r)
      throws RemoteException {
      try{synchronized(this) {wait(2000);}}
      catch (InterruptedException ie){};
      result = r;
      finish();
      System.out.println("rangement du resultat " + result);
      synchronized(this) {
      this.notifyAll(); } }
    public void unreferenced() {
      try {boolean b = UnicastRemoteObject.unexportObject(this,true);}
      catch (NoSuchObjectException nsoe) {};  }
    public double get_result() {return result;}
}
```

La classe MRPointD

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.io.*;

public class MRPointD extends PointD  implements MRPointRMI {

    MRPointD() throws RemoteException {}

    MRPointD(int a, int b) throws RemoteException {super(a,b);}

    public double distance(RappelRMI r)  throws RemoteException {
      new Calcul(this,r).start();
      return 0.0;
    }
}
```

La classe Calcul

```
public class Calcul extends Thread {
    MRPointD  p;
    RappelRMI r;
    Calcul(MRPointD p, RappelRMI r) { this.p=p; this.r=r; }

    public void run(){
      try {
        double z = p.distance();
        r.put(z);
      }
      catch (Exception e){e.printStackTrace();};
    }
}
```

La classe CreationN

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class CreationN {
    public static void main (String args[]) {
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
     }
    try {
      MRPointD p0 = new MRPointD();
      MRPointD p1 = new MRPointD(3,4);

      Naming.rebind("//chrome.pps.jussieu.fr/point0",p0);
      Naming.rebind("//chrome.pps.jussieu.fr/point1",p1);
      System.out.println("Objets distribues 'p0' " +
       "et 'p1' sont enregistres");
    }
    catch (Exception e) { e.printStackTrace();
    }
  }
}
```

La classe ClientN

```
import java.rmi.*;
public class ClientN {
  public static void main( String argv[]) {
    String machine = argv[0]; String port = argv[1];
    String url0="rmi://"+machine+":"+port+"/point0";
    String url1="rmi://"+machine+":"+port+"/point1";
    try {
      MRPointRMI p0 = (MRPointRMI)Naming.lookup(url0);
      MRPointRMI p1 = (MRPointRMI)Naming.lookup(url1);
      p0.affiche(); p1.affiche();
      p0.rmoveto(7,12); p1.rmoveto(5,6);
      p0.affiche();    p1.affiche();

      Rappel ri = new Rappel();
      System.out.println("appel de distance");
      double r = p1.distance(ri);
      System.out.println("retour de distance : r = "+r);
      if (! ri.is_finished()) {
        synchronized(ri) {
          ri.wait(); }}
      System.out.println("le resultat est " +  ri.get_result());
    }
    catch (Exception e) { System.err.println("exception : " + e.getMessag
```

Dgc

Difficultés du GC: en présence de références locales et distantes.

- comment savoir si une référence distante est encore active?
  - implanter un mécanisme de GC réparti : difficile et lent
  - laisser ce travail au programmeur : durée de vie limitée d'une référence distante

Si l'objet exposé est publié dans un serveur de nom, il y a toujours une référence distance sur cet objet.

Période de détention

Un client possede une référence distante pour une certaine période (période de détention).

- passé ce temps : le serveur peut récupérer la mémoire
- le client peut aussi informer le serveur qu'il vient de libérer de son coté la référence (methode `finalize`)
- si le client utilise de nouveau la référence, la période de détention est de nouveau maximum
- sinon le client doit signifier au serveur qu'il désire la garder (méthode `dirty()`)
- durée connu par `leaseValue`.

Trace du Dgc

objet non référencé et récupéré: :

1. implanter `unreferenced` (classe PointD)

```
  implements RemoteConnection, java.rmi.server.Unreferenced

public void unreferenced() {
  System.out.println("RemoteConnectionImpl for client  unreferenced")
}
```

2. et la méthode `finalize` (classe PointD)

```
protected void finalize() {
  System.out.println("RemoteConnectionImpl for client  finalized");
```

Activation d’objets

Pour résoudre : :

- l’exposition (attente de connexions) continuelle d’objets :

  ressources mémoire et réseau monopolisées

- si le serveur s’arrête puis repart :

  le stub d’un client sur l’ancienne exposition n’est plus valable

Démon d’activation

1. un descripteur d’activation d’un objet distant est enregistré auprès du démon d’activation (`rmid`);
2. c’est ce descripteur qui est publié dans le serveur de noms (`rmiregistry`);
3. lors d’une requête sur cet objet, le stub client demande l’activation de cet objet qui est alors exposé dans le serveur d’objet;
4. le client peut alors envoyer les paramètres de l’invocation d’une méthode sur cet objet.

Package et commande

- package `java.rmi.activation`
- commande `rmid` : démon d'activation des objets distants

Exemple : Points distants

```
import java.rmi.*;
// nouvelles importations
// import java.rmi.server.UnicastRemoteObject;
import java.rmi.activation.*;
public class PointD extends Activatable
// herite de Activable
//UnicastRemoteObject
    implements PointRMI {
    int x,y;
//    PointD(int a, int b) throws RemoteException {x=a;y=b;}
//  change le constructeur sans argument
//  PointD() throws RemoteException {x=0;y=0;}
    public PointD(ActivationID id, MarshalledObject data) throws RemoteExc
      {super(id,0); }
    public void moveto (int a, int b)  throws RemoteException  { x=a; y=b;
    public void rmoveto (int dx, int dy) throws RemoteException {
      x = x + dx; y = y + dy;}
    public void affiche()  throws RemoteException
      { System.out.println("(" + x + "," + y + ")");}
    public double distance() throws RemoteException
      { return Math.sqrt(x*x+y*y);}
}
```

CreationA

```
import java.rmi.*;
import java.rmi.activation.*;
public class CreationA {
    public static void main (String args[]) {
      if (System.getSecurityManager() == null) {
        System.setSecurityManager(new RMISecurityManager());
       }
       try { ActivationSystem as = ActivationGroup.getSystem();
         ActivationGroupDesc agd = new ActivationGroupDesc(null,null);
         ActivationGroupID agi = as.registerGroup(agd);
         String oClass = "PointD";
         String oClassLocation = "file:/tmp";
         MarshalledObject oArgs=null;
    ActivationDesc ad = new ActivationDesc(agi, oClass, oClassLocation,
                                           oArgs);
 // PointD p0 = new PointD();  //PointD p1 = new PointD(3,4);
         Remote r1 = Activatable.register(ad);
         Naming.rebind("//127.0.0.1/pointp0",r1);
         Remote r2 = Activatable.register(ad);
         Naming.rebind("//127.0.0.1/pointp1",r2);
   // Naming.rebind("//chrome.pps.jussieu.fr/point0",p0);
       }
       catch (Exception e) { e.printStackTrace();
```

Lancement

```
$ rmiregistry&
$ rmid&
$  java -Djava.security.policy=java.policy Creat
$ java Client
```