INF 560
Calcul Parallèle et Distribué
Cours 5

Eric Goubault et Sylvie Putot

Ecole Polytechnique

19 janvier 2015

Plan du cours

- Coordination en mémoire partagée en Java: exclusion mutuelle
  - modèle Cuda: synchronisation non bloquante (opérations atomiques) + barrières de synchronisation
  - en Java: synchronisation bloquante avec `synchronized`, et les moniteurs/variables de condition `wait()` et `notify()`
  - sémaphores, interblocage
- Calcul distribué: Remote Method Invocation (Java/RMI)

Synchronisation? Revenons à nos comptes Java...

```
public class Compte {
  private int valeur;

  Compte(int val) {
    valeur = val;
  }
  public int solde() {
    return valeur;
  }
  public void depot(int somme) {
    if (somme > 0)
      valeur+=somme;
  }
  public boolean retirer(int somme) throws InterruptedException {
    if (somme > 0) && (somme <= valeur) {
      try{ Thread.currentThread().sleep(50);
      valeur -= somme;
      Thread.currentThread().sleep(50);
      return true; } catch(InterruptedException e) {}
    }
    return false;
  }
}
```


La banque...

```
public class Banque implements Runnable {
  Compte nom;
  Banque(Compte n) { nom = n; }

  public void Liquide (int montant) throws InterruptedException {
    if (nom.retirer(montant)) {
      Thread.currentThread().sleep(50);
      Donne(montant);
      Thread.currentThread().sleep(50);
    }
      ImprimeRecu();
      Thread.currentThread().sleep(50);
  }
  public void Donne(int montant) {
    System.out.println(Thread.currentThread().
        getName()+": Voici vos " + montant + " euros.");
  }
  public void ImprimeRecu() {
    if (nom.solde() > 0)
      System.out.println(Thread.currentThread().
          getName()+": Il vous reste " + nom.solde() + " euros.");
    else
      System.out.println(Thread.currentThread().
          getName()+": Vous etes fauches!");
}
```

```
  public void run() {
    try {
      for (int i=1;i<10;i++) {
          Liquide(100*i);
          Thread.currentThread().sleep(100+10*i);
      }
    } catch (InterruptedException e) {
      return;
    }
  }

  public static void main(String[] args) {
    Compte Commun = new Compte(1000);
    Runnable Mari = new Banque(Commun);
    Runnable Femme = new Banque(Commun);
    Thread tMari = new Thread(Mari);
    Thread tFemme = new Thread(Femme);
    tMari.setName("Conseiller Mari");
    tFemme.setName("Conseiller Femme");
    tMari.start();
    tFemme.start();
  }
}
```


Une des exécutions possibles

```
> java Banque
Conseiller Mari: Voici vos 100 euros.
Conseiller Femme: Voici vos 100 euros.
Conseiller Mari: Il vous reste 800 euros.
Conseiller Femme: Il vous reste 800 euros.
Conseiller Mari: Voici vos 200 euros.
Conseiller Femme: Voici vos 200 euros.
Conseiller Femme: Il vous reste 400 euros.
Conseiller Mari: Il vous reste 400 euros.
Conseiller Mari: Voici vos 300 euros.
Conseiller Femme: Voici vos 300 euros.
Conseiller Femme: Vous etes fauches!
Conseiller Mari: Vous etes fauches! ...
```

Résultat...

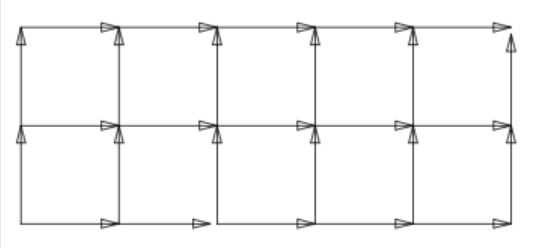
- Le mari a retiré 600 euros du compte commun,
- La femme a retiré 600 euros du compte commun,
- qui ne contenait que 1000 euros au départ!

Explication

("Sémantique")

- L'exécution de plusieurs threads se fait en exécutant une action insécable ("atomique") de l'un des threads, puis d'un autre ou d'éventuellement du même etc.
- Tous les "mélanges" possibles sont permis

Sémantique par entrelacements



Explication et solution

Explication

Si les 2 threads `tMari` et `tFemme` sont exécutés de telle façon que dans `retirer`, chaque étape soit faite en même temps:

- Le test pourra trouvé être satisfait par les deux threads en même temps,
- Qui donc retireront en même temps de l'argent.

Une solution

- Rendre "atomique" le fait de `retirer` de l'argent,
- Se fait en déclarant `synchronized` la méthode `retirer` de la classe `Compte`:

```
public synchronized boolean retirer(int somme)
```

Synchronized

```
public class Compte {
  private int valeur;

  Compte(int val) {
    valeur = val;
  }
  public int solde() {
    return valeur;
  }
  public void depot(int somme) {
    if (somme > 0)
      valeur+=somme;
  }
  public synchronized boolean retirer(int somme) throws Interrupte
    if (somme > 0) && (somme <= valeur) {
      try{ Thread.currentThread().sleep(50);
      valeur -= somme;
      Thread.currentThread().sleep(50);
      return true; } catch(InterruptedException e) {}
    }
    return false;
  }
}
```


Maintenant...

```
% java Banque
Conseiller Mari: Voici vos 100 euros.
Conseiller Mari: Il vous reste 800 euros.
Conseiller Femme: Voici vos 100 euros.
Conseiller Femme: Il vous reste 800 euros.
Conseiller Mari: Voici vos 200 euros.
Conseiller Mari: Il vous reste 400 euros.
Conseiller Femme: Voici vos 200 euros.
Conseiller Femme: Il vous reste 400 euros.
Conseiller Femme: Il vous reste 100 euros.
Conseiller Mari: Voici vos 300 euros.
Conseiller Mari: Il vous reste 100 euros.
Conseiller Femme: Il vous reste 100 euros.
Conseiller Mari: Il vous reste 100 euros...
```

Résultat...

- Le mari a tiré 600 euros,
- La femme a tiré 300 euros,
- et il reste bien 100 euros dans le compte commun.

Synchronized

- Ici `synchronized` qualifie une méthode (`retirer`), mais est en fait un verrou au niveau de l'objet (`Compte`) sur lequel s'applique la méthode
  - lorsqu'un thread appelle cette méthode, il acquiert le verrou sur l'objet
  - aucun autre thread ne peut acquérir le verrou tant que le thread n'a pas relâché le verrou
  - tout appel d'une autre méthode `synchronized` sur cet objet est bloquée tant que le thread n'a pas relâché le verrou
- Dans le cas d'une méthode `static`, le verrou s'applique à la classe (i.e. toutes ses instances sont verrouillées)
- On peut aussi définir un bloc comme `synchronized` sur un objet `O` par `synchronized(O) { ... }`
- Le corps d'une méthode ou d'un bloc `synchronized` est souvent appelé *section critique*

Moniteurs: `wait()` et `notify()` en JAVA

Considérons:

```
public synchronized void set_var() {
  // on attend qu'une condition passe à vrai
  while (flag == false) {};
  [...] // l'action à réaliser
}
```

- Pas une très bonne façon de faire dans l'absolu...
- ... mais spécialement catastrophique dans une méthode synchronized!
- Le thread a pris le verrou; et tout appel d'une autre méthode synchronized est bloqué: situation d'interblocage!

Solution = mécanismes de mise en attente et de réveil `wait()` et `notify()` sur l'objet

Moniteurs: `wait()` et `notify()` en JAVA

Chaque objet fournit un verrou, mais aussi un mécanisme de mise en attente

- Ces mécanismes doivent être appelés depuis l'intérieur d'une méthode ou d'un bloc `synchronized` (qui prend donc le verrou sur l'objet sur lequel va s'appliquer le mécanisme)
- `void wait()` rend le verrou sur l'objet sur lequel il s'applique et bloque en attendant l'arrivée d'une condition (il y a aussi une version avec timeout)

```
if (flag == false) wait();
  [...]
```

- `void notify()` notifie un thread en attente sur un objet, de l'arrivée d'un évènement.
- `void notifyAll()` même chose mais pour tous les threads en attente sur l'objet.

Plus précisément

- Chaque objet JAVA *o* comporte une liste de threads en attente u verrou sur l'objet
- Un thread y rentre en exécutant `o.wait()`
- Un thread en sort si un `o.notify()` est exécuté par un autre thread, et que ce soit celui-ci dans la liste qui est libéré (en général, c'est le premier en attente qui est libéré)
- On doit absolument protéger l'accès à cette liste d'attente par un `synchronized`

Un exemple (faux...)

```
class buffer1 {
 Object data = null;

 public synchronized void push(Object d) {
  try { if (data != null) wait();
   } catch (Exception e) { System.out.println(e); return; }
  data = d;
  System.out.println("Pushed "+data);
  try { if (data != null) {notify();
          System.out.println("Push-notify"); }
   } catch (Exception e) { System.out.println(e); return; }
 }
 public Object pop() {
  try { if (data == null) wait();
   } catch (Exception e) { System.out.println(e); return null; }
  System.out.println("Read "+data);
  data = null;
  try { if (data == null) {notify();
         System.out.println(from+" Pop-notify");  }
  } catch (Exception e) { System.out.println(e); return null; }
  return o;
  }
}
```


Les threads producteur/consommateur

```
class Prod extends Thread {
 buffer1 buf;

 public Prod(buffer1 b) {
  buf = b; }

 public void run() {
  while (true) {
   buf.push(new Integer(1)); }
 }
}
class Cons extends Thread {
 buffer1 buf;

 public Cons(buffer1 b) {
  buf = b; }

 public void run() {
  while (true) {
   buf.pop(); }
  }
}
```

Le main...

1 producteur et 2 consommateurs qui se partagent un buffer:

```
public class essaimon0 {
 public static void main(String[] args) {
  buffer1 b = new buffer1();
  new Prod(b).start();
  new Cons(b).start();
  new Cons(b).start(); } }
```

Première erreur...

```
sylvie$ javac essaimon0.java
sylvie$ java essaimon0 | more
java essaimon0
Pushed 1
Push-notify
Thread[Thread-1,5,main] Read 1
java.lang.IllegalMonitorStateExceptionfrom pop-notify
java.lang.IllegalMonitorStateExceptionfrom pop-wait
java.lang.IllegalMonitorStateExceptionfrom pop-wait
...
```

Pas de `synchronized` dans la méthode `pop()`: non repéré à la compilation, mais à l'exécution!

Correction...

On rajoute `synchronized` dans la méthode `pop()`, puis:

```
sylvie$ javac essaimon0.java
sylvie$ java essaimon0 | more
Pushed 1
Push-notify
Thread[Thread-1,5,main] Read 1
Thread[Thread-1,5,main] Pop-notify
Pushed 1
Push-notify
Thread[Thread-1,5,main] Read 1
Thread[Thread-1,5,main] Pop-notify
Thread[Thread-2,5,main] Read null
Thread[Thread-2,5,main] Pop-notify
...
```

Quel est le problème?

- `wait()` et `notify()` s'appliquent ici sur `this` qui est toujours le même et unique `buffer`
- Supposons que `data=null` (pas de `push` par exemple, ou un pop effectué)
- Quand le premier consommateur exécute `pop()`, il voit `data=null` et fait `wait()` : **il suspend sa prise de verrou sur le** `buffer`
- Le deuxième consommateur peut donc s'exécuter, voit `data=null` et fait `wait()` : **il suspend sa prise de verrou sur le** `buffer`
- Le producteur voit `data=null` et fait `data=1` puis `notify`, débloquant ainsi l'un des deux consommateurs, disons le premier
- Le premier consommateur lit alors `1` puis fait `data=null` et `notify()` et termine, libérant son verrou sur le `buffer`
- Le deuxième consommateur est alors débloqué et fait `read null`...

Une solution possible...

Utiliser deux objets dans le buffer, un qui signale "vide" et un autre qui signale "plein" (variables de conditions...):

```
Object full  = new Object();
Object empty = new Object();
Object data = null;

public void push(Object d) {
 synchronized(full) {
  try { if (data != null) full.wait();
   } catch (Exception e) { System.out.println(e); return; } }
 data = d;
 System.out.println("Pushed "+data);
 synchronized(empty) {
  try { if (data != null) empty.notify();
   } catch (Exception e) { System.out.println(e); return; }
}
```

```
  public Object pop() {
   synchronized(empty) {
    try { if (data == null) empty.wait();
     } catch (Exception e) { System.out.println(e); return null; }
   Object o = data;
   System.out.println("Read "+o);
   data = null;
   synchronized(full) {
    try { if (data == null) full.notify();
    } catch (Exception e) { System.out.println(e); return null; } }
   return o; }
}
```

Exécution:

```
sylvie$ javac essaimon1.java
sylvie$ java essaimon1 | more
Pushed 1
Read 1
Pushed 1
Read 1
Pushed 1
Read 1
...
```

(correct!)

Sémaphores en Java

Sémaphores

- Des fonctions explicites pour prendre/rendre un verrou
- Pas réellement nécessaires en Java: les mécanismes de synchronisation suffisent
- Mais intéressant de voir qu'on peut très facilement coder un sémaphore avec ces mécanismes
- Disponibles avec d'autres primitives de synchronisation haut-niveau dans `java.util.concurrent`

Sémaphores en Java

```
public class Semaphore {
  int n; // compteur du sémaphore
  String name;

  public Semaphore(int max, String S) {
    n = max; name = S;
  }

  public synchronized void P() { // acquérir un jeton
    while (n == 0) {
      try { wait(); } catch(InterruptedException ex) {};
    }
    n--;
    System.out.println("P("+name+")");
  }

  public synchronized void V() { // relâcher
    n++; System.out.println("V("+name+")");
    notify();
  }
}
```

Bien sûr ici, `this.wait` et `this.notify` sont ce que l'on souhaite...

Section Critique

```
public class essaiPV extends Thread {
    static int x = 3;
    Semaphore u;

    public essaiPV(Semaphore s) {
        u = s;
    }

    public void run() {
        int y;
        // u.P();
```

```
            try {
                Thread.currentThread().sleep(100);
                y = x;
                Thread.currentThread().sleep(100);
                y = y+1;
                Thread.currentThread().sleep(100);
                x = y;
                Thread.currentThread().sleep(100);
            } catch (InterruptedException e) {};
            System.out.println(Thread.currentThread().
                                getName()+": x="+x);
            // u.V();
        }
        public static void main(String[] args) {
            Semaphore X = new Semaphore(1,"X");
            new essaiPV(X).start();
            new essaiPV(X).start();
        }
}
```


Résultats

Sans P, V

```
% java essaiPV
Thread-2: x=4
Thread-3: x=4
```

Avec P, V

```
% java essaiPV
P(X)
Thread-2: x=4
V(X)
P(X)
Thread-3: x=5
V(X)
```


Ressources partagées et priorités

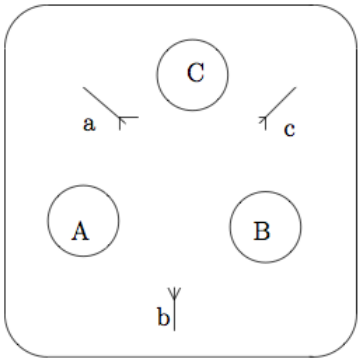
Attention, il est possible de tomber dans le problème d'inversion de priorité:

- Soit T_1 une tâche prioritaire par rapport à T_2, a et b deux objets partagés par T_1 et T_2 (protégés par exemple par deux sémaphores de même nom)
- Supposons l'exécution suivante:

T_1	T_2
Pa	-
(bloqué)	Pb
(obtient le verrou a)	(obtient le verrou b)
Pb	-
(bloqué)	-

- Ainsi, c'est T_2 qui s'exécute alors qu'il est de priorité moindre...

Points morts et philosophes qui dinent




```
public class Phil extends Thread {
    Semaphore LeftFork;
    Semaphore RightFork;

    public Phil(Semaphore l, Semaphore r) {
        LeftFork = l;
        RightFork = r;
    }
```

```
        public void run() {
            try {
                Thread.currentThread().sleep(100);
                LeftFork.P();
                Thread.currentThread().sleep(100);
                RightFork.P();
                Thread.currentThread().sleep(100);
                LeftFork.V();
                Thread.currentThread().sleep(100);
                RightFork.V();
                Thread.currentThread().sleep(100);
            } catch (InterruptedException e) {};
        }
    }
    public class Dining {
        public static void main(String[] args) {
            Semaphore a = new Semaphore(1,"a");
            Semaphore b = new Semaphore(1,"b");
            Semaphore c = new Semaphore(1,"c");
            Phil Phil1 = new Phil(a,b);
            Phil Phil2 = new Phil(b,c);
            Phil Phil3 = new Phil(c,a);
            Phil1.setName("Kant");
            Phil2.setName("Heidegger");
            Phil3.setName("Spinoza");
            Phil1.start();
            Phil2.start();
```


Résultat

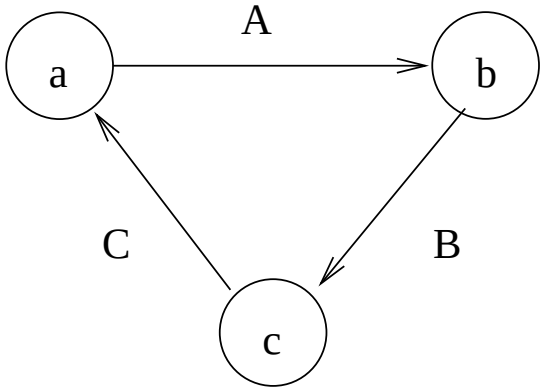
```
% java Dining
Kant: P(a)
Heidegger: P(b)
Spinoza: P(c)
^C
```

Interblocage

- Une exécution possible n'atteint jamais l'état terminal: quand les trois philosophes prennent en même temps leur fourchette gauche.
- On pouvait s'en apercevoir sur le "request graph" ou le "progress graph" (voir poly)
- Plusieurs solutions possibles: philosophes "droitiers/gauchers", sémaphore à compteur égal à (nombre de philosophes - 1), etc

Request graph

`Pa.Pb.Vb.Va|Pb.Pa.Va.Vb|Pc.Pa.Va.Vc`:



Remote Method Invocation

- Permet d'invoquer des méthodes d'un objet distant, c'est à dire appartenant à une autre JVM, sur une autre machine
- Architecture de type client/serveur, l'information est distribuée; similaire aux "Remote Procedure Calls" POSIX
- Se rapproche de plus en plus de CORBA (norme pour la communication d'applications en environnement hétérogène)

Références: JAVA, Network Programming and Distributed Computing, D. Reilly et M. Reilly, Addison-Wesley.

Architecture type client/serveur

- Un serveur fournit un service RMI: des méthodes que l'on pourra appeler à distance
  - ses instances sont des objets ordinaires dans l'espace d'adressage de leur JVM, sur lequels des pointeurs peuvent être envoyés aux autres espaces d'adressage
  - il doit implémenter une interface qui étend l'interface d'objets distants java.rmi.Remote
  - ses méthodes doivent lever des RemoteExceptions (probabilité plus forte de panne sur un système distribué)
  - les paramètres et valeurs retournées des méthodes doivent être des objets `Serializable` ou des objets qui implémentent l'interface `Remote`
  - il doit s'enregistrer auprès du rmiregistry (annuaire de services existants) pour que les clients le trouvent
- Un client invoque des méthodes de ce service
  - il va chercher les services dans le registre d'objets distants
  - il récupère une référence sur un objet (serveur) distant
  - il peut alors appeler les méthodes définies dans l'interface de l'objet, de façon presque transparente

Serializable

- Objets instances peuvent être transcrits en "stream", c'est-à-dire en flots d'octets.
- `writeObject(ObjectOutputStream aOutputStream)`
  `readObject(ObjectInputStream aInputStream)`

  responsables respectivement de décrire un objet sous forme de flot d'octets et de reconstituer l'état d'un objet à partir d'un flot d'octets.
- La plupart des classes (et de leurs sous-classes) de base `String`, `HashTable`, `Vector`, `HashSet`, `ArrayList`...sont `Serializable`.
- Dans le cas où on passe une classe `Serializable`, il faut que la définition de cette classe soit connue des clients et du serveur
- Il peut y avoir à gérer la politique de sécurité (sauf pour les objets "simples", comme `String` etc.).

Stubs (coté client) & squelettes (côté serveur)

Un stub est une sorte de proxy pour le client (cache la sérialization et les communications bas niveau sur le réseau):

- c'est lui qui initie la connection avec la JVM distante contenant l'objet distant
- envoie les arguments à l'objet distant ("marshals")
- attend et récupère les résultats ("unmarshals")

Un squelette est responsable (sur la demande du stub correspondant) d'appeler la méthode sur le serveur:

- lit les paramètres ("unmarshals")
- appelle la méthode de l'objet serveur correspondant
- écrit les paramètres sur le réseau ("marshals")

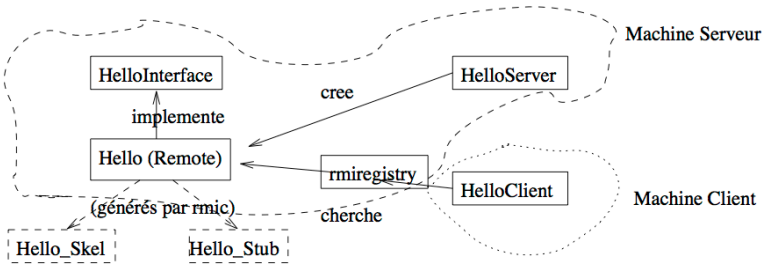
Les différentes classes et interfaces

Implémentation du serveur:

- Son interface doit étendre l'interface Remote
- Il doit étendre la classe abstraite RemoteServer du paquetage java.rmi
- UnicastRemoteObject est une classe concrète qui étend RemoteServer et gére la communication et les stubs

Exemple - Hello World distribué

On va construire les classes:



Pour en savoir plus: packages RMI

- `java.rmi` définit l'interface `RemoteInterface`, et les exceptions,
- `java.rmi.activation` (depuis JAVA2): permet l'activation à distance des objets,
- `java.rmi.dgc`: s'occupe du ramassage de miettes dans un environnement distribué,
- `java.rmi.registry` fournit l'interface permettant de représenter un `rmiregistry`, d'en créer un, ou d'en trouver un,
- `java.rmi.server` fournit les classes et interfaces pour les serveurs RMI.

`http://docs.oracle.com/javase/6/docs/technotes/guides/rmi/index.html`

Objet distant Hello

Interface `HelloInterface`:

```
import java.rmi.*;

public interface HelloInterface extends Remote {
    public String say() throws RemoteException;
}
```

Implémentation de l'objet distant `Hello`:

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

public class Hello extends UnicastRemoteObject
  implements HelloInterface {
  private String message;

  public Hello(String msg) throws RemoteException {
    message = msg; }

  public String say() throws RemoteException {
    return message;
  }
}
```


Client

```
import java.rmi.*;
public class HelloClient {
  public static void main(String[] argv) {
    try {
      HelloInterface hello =
      (HelloInterface) Naming.lookup
              ("//cher.polytechnique.fr/Service");
      System.out.println(hello.say());
    } catch(Exception e) {
        System.out.println("HelloClient exception: "+e);
    }
  }
}
```

(le serveur est supposé toujours être sur `cher`, voir plus loin pour d'autres méthodes)

Serveur

```
import java.rmi.*;

public class HelloServer {
  public static void main(String[] argv) {
    try {
      Naming.rebind("Service",new Hello("Hello, world!"));
      System.out.println("Hello Server is ready.");
    } catch(Exception e) {
        System.out.println("Hello Server failed: "+e);
    }
  }
}
```


Compilation et exécution du serveur

```
[cher Hello]$ javac Hello*.java
```

Démarrer le serveur de noms:

```
[cher Hello]$ rmiregistry &
```

(attendre un minimum). Puis lancer le serveur sur `cher`:

```
[cher Hello]$ java HelloServer
Hello Server is ready.
```

Mais si j'ai oublié de lancer rmiregistry, par exemple:

```
[cher Hello]$ killall rmiregistry
[cher Hello]$ java HelloServer
Hello Server failed: java.rmi.ConnectException:
Connection refused to host: 129.104.252.80; nested exception is:
java.net.ConnectException: Connexion refusée
```

Enfin sur le (ou les) client(s), ici sur `allemagne`:

```
[allemagne Hello]$ java HelloClient
Hello, world!
```


Installation locale aux salles de TD

(récupérer les programmes java sur la page web)

- `rmiregistry` doit être démarré avec un numéro de port distinct pour plusieurs utilisateurs sur une même machine (numéros à partir de 1099), voir répartition sur fiche TD. Exemple sur machine serveur:

  ```
  rmiregistry 1100 &
  ```

- Dans ce cas, le serveur devra s'enregistrer par

  ```
  Naming.rebind("rmi://localhost:1100/Service");
  ```

  Et le client devra chercher sur le même port:

  ```
  Naming.lookup("rmi://cher.polytechnique.fr:1100/Service");
  ```


Autre méthode pour le serveur

Créer le démon `rmiregistry` dans le code du serveur:

```
import java.rmi.*;
import java.rmi.registry.*;
public class HelloServer {
  public static void main(String[] argv) {
    try {
      // créer le démon rmiregistry ecoutant sur le port 1100
      LocateRegistry.createRegistry(1100);
      // reference sur ce demon pour avoir un objet manipulable
      Registry r = LocateRegistry.getRegistry("oncidium",1100);
      // ou "localhost" = la machine ou on execute HelloServer:
      // Registry r = LocateRegistry.getRegistry("localhost",1100)
      // on enregistre le service "Service":
      Naming.rebind("//oncidium:1100/Service",new Hello("Hi!"));
     // ou on utilise explicitement le demon pour enregistrer:
     // r.rebind("Service",new Hello("Hi!"));
      // on peut par exemple parcourir les services connus:
      String[] name = r.list();
      for (int i=0; i<name.length;i++)
        System.out.println(name[i]); // ici: Service
      System.out.println("Hello Server is ready.");
    } catch(Exception e) {
      System.out.println("Hello Server failed: "+e); }
  }
```

Et pour le client: (accès au rmiregistry)

```
import java.rmi.*;
import java.rmi.registry.*;

public class HelloClient {
    public static void main(String[] argv) {
try {
  Registry r = LocateRegistry.getRegistry("oncidium",1100);
  HelloInterface hello = (HelloInterface) r.lookup("Service");
  System.out.println(hello.say());
} catch(Exception e) {
    System.out.println("HelloClient exception: "+e); }
    }
}
```


Callback

L'idée est la suivante (programmation "événementielle", typique d'interfaces graphique par exemple Swing):

- les "clients" vont s'enregistrer auprès d'un serveur,
- le "serveur" va les "rappeler" uniquement lorsque certains événements se produisent,
- le client n'a pas ainsi à faire de "l'active polling" (c'est à dire à demander des nouvelles continuellement au serveur) pour être mis au fait des événements.

Principe du "rappel"

Comment notifier un objet ou client (distant) de l'apparition d'un événement?

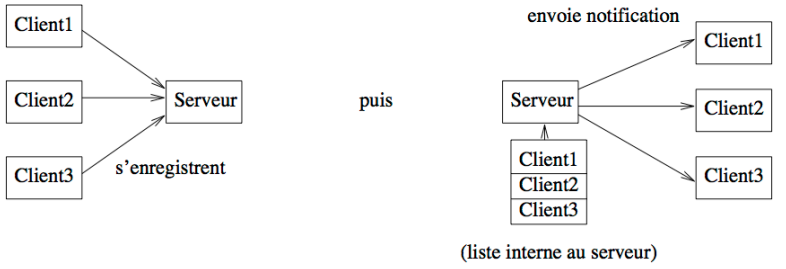
- On passe la référence du client/objet à rappeler, au serveur chargé de suivre (ou source des) les événements
- A l'apparition de l'événement, le serveur va invoquer la méthode de notification du client.

Ainsi,

- Pour chaque type d'événement, on crée une interface spécifique
  - que le client voulant être notifié devra implémenter
  - contenant une méthode de notification du client
- Le serveur implémente une interface qui permet aux clients potentiels à notifier de s'enregistrer.

Cela implique que "clients" et "serveurs" sont tous à leur tour "serveurs" et "clients".

C'est à dire...



Exemple (ci-après)

- un serveur TemperatureServerSensor qui définit et modifie une température
- des clients TemperatureMonitor qui veulent être notifiés de changements de température

Exemple - l'interface du serveur d'événements

... doit au moins pouvoir permettre l'inscription et la désinscription de clients voulant être notifié:

```
interface TemperatureSensor extends java.rmi.Remote {
    public double getTemperature() throws
        java.rmi.RemoteException;
    public void addTemperatureListener
        (TemperatureListener listener)
        throws java.rmi.RemoteException;
    public void removeTemperatureListener
        (TemperatureListener listener)
        throws java.rmi.RemoteException; }
```


Exemple - interface associée à un évévenement

... ici, changement de température:

```
interface TemperatureListener extends java.rmi.Remote {
    public void temperatureChanged(double temperature)
        throws java.rmi.RemoteException;
}
```

C'est la méthode de notification de tout client intéressé par cet événement. Forcément un objet `Remote`.

L'implémentation du serveur: TemperatureSensorServer

- Doit être une sous-classe de `UnicastRemoteObject` (pour être un serveur...).
- Doit implémenter l'interface `TemperatureSensor` pour que les clients puissent s'enregistrer et demander la température,
- implémente également `Runnable` ici pour avoir un thread indépendant qui simule les changements de température.

```
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;
public class TemperatureSensorServer extends UnicastRemoteObject
                    implements TemperatureSensor, Runnable {
  private volatile double temp;
  private Vector<TemperatureListener> list =
            new Vector<TemperatureListener>();
  static final long serialVersionUID = 42L;
```

(le vecteur `list` contiendra la liste des clients)

Implémentation TemperatureSensorServer

Constructeur (température initiale) et méthode de récupération de la température:

```
    public TemperatureSensorServer()
      throws java.rmi.RemoteException { temp = 98.0; }

    public double getTemperature()
      throws java.rmi.RemoteException {  return temp; }
```

Méthodes d'ajout et de retrait de clients:

```
public void addTemperatureListener
            (TemperatureListener listener)
  throws java.rmi.RemoteException {
  System.out.println("adding listener "+listener);
  list.add(listener); }

public void removeTemperatureListener
            (TemperatureListener listener)
  throws java.rmi.RemoteException {
  System.out.println("removing listener "+listener);
  list.remove(listener); }
```

Implémentation TemperatureSensorServer

Thread responsable du changement aléatoire de la température:

```
public void run()
{ Random r = new Random();
  for (;;)
    { try {
        int duration = r.nextInt() % 10000 +2000;
        if (duration < 0) duration = duration*(-1);
        Thread.sleep(duration); }
      catch(InterruptedException ie) {}
      int num = r.nextInt();
      if (num < 0)
        temp += .5;
      else
        temp -= .5;
      notifyListeners(); } }
```

(`notifyListeners()` est la méthode suivante, chargée de broadcaster le changement d'événements à tous les clients enregistrés)

Implémentation TemperatureSensorServer

```
private void notifyListeners() {
  for (Enumeration e = list.elements(); e.hasMoreElements();)
  { TemperatureListener listener =
        (TemperatureListener) e.nextElement();
    try {
      listener.temperatureChanged(temp);
    } catch(RemoteException re) {
        System.out.println("removing listener -"+listener);
        list.remove(listener); } } }
```

(on fait simplement appel, pour chaque client, à la méthode de notification `temperatureChanged`)

Implémentation TemperatureSensorServer

Enregistrement du service auprès du `rmiregistry` (nom de la machine éventuellement fourni à la ligne de commande):

```
public static void main(String args[]) {
  System.out.println("Loading temperature service");
  try {
    TemperatureSensorServer sensor =
                        new TemperatureSensorServer();
    String registry = "localhost";
    if (args.length >= 1) registry = args[0];
    String reg = "rmi://"+registry+"/TemperatureSensor";
    Naming.rebind(reg,sensor);
```

Démarrage du thread en charge de changer aléatoirement la température, et gestion des exceptions:

```
    Thread thread = new Thread(sensor);
    thread.start(); }
  catch (RemoteException re) {
    System.err.println("Remote Error - "+re); }
  catch (Exception e) {
    System.err.println("Error - "+e); } } }
```


Implémentation clients: TemperatureMonitor

Etend `UnicastRemoteObject` car serveur également! De même implémente `TemperatureListener`)

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

public class TemperatureMonitor extends UnicastRemoteObject
    implements TemperatureListener {
    public TemperatureMonitor() throws RemoteException {}
```

Implémentation de la méthode de rappel:

```
public void temperatureChanged(double temperature)
  throws java.rmi.RemoteException {
  System.out.println("Temperature change event : "
                     +temperature);
}
```

Exemple - implémentation clients

Recherche du service serveur d'événements:

```
public static void main(String args[]) {
  System.out.println("Looking for temperature sensor");
  try {
    String registry = "localhost";
    if (args.length >= 1)
    registry = args[0];
    String registration = "rmi://"+registry+
                          "/TemperatureSensor";
    Remote remoteService = Naming.lookup(registration);
    TemperatureSensor sensor = (TemperatureSensor)
                               remoteService;
```

Création d'un moniteur et enregistrement auprès du serveur d'événements:

```
    double reading = sensor.getTemperature();
    System.out.println("Original temp : "+reading);
    TemperatureMonitor monitor = new TemperatureMonitor();
    sensor.addTemperatureListener(monitor);
```


Exemple - implémentation clients

Gestion des exceptions:

```
} catch(NotBoundException nbe) {
    System.out.println("No sensors available"); }
  catch (RemoteException re) {
    System.out.println("RMI Error - "+re); }
  catch (Exception e) {
    System.out.println("Error - "+e); } }
```


Compilation

```
[cher Temperature]$ javac Temperature*.java
```

```
[cher Temperature]$ rmiregistry &
[cher Temperature]$ java TemperatureSensorServer
Loading temperature service
```

Premier client (sur loire):

```
[loire Temperature]$ rmiregistry &
[loire Temperature]$ java TemperatureMonitor cher
Looking for temperature sensor
Original temp : 97.5
Temperature change event : 97.0
Temperature change event : 96.5
Temperature change event : 97.0
Temperature change event : 96.5
Temperature change event : 96.0
Temperature change event : 96.5
Temperature change event : 97.0
[...]
```


Exécution

On voit alors sur la console de cher:

```
adding listener -Proxy[TemperatureListener,
RemoteObjectInvocationHandler[UnicastRef [liveRef:
[endpoint:[129.104.254.47:51609](remote),
objID:[20bbb3df:14afdebe4f4:-7ffe, 676724842184262975]]]]]
```

Rajoutons un moniteur sur allemagne:

```
[allemagne Temperature]$ rmiregistry &
[allemagne Temperature]$ java TemperatureMonitor cher
Looking for temperature sensor
Original temp : 97.0
Temperature change event : 96.5
Temperature change event : 97.0
Temperature change event : 97.5
Temperature change event : 98.0
Temperature change event : 98.5
Temperature change event : 98.0
...
```

Exécution

Ce qui produit sur cher:

```
adding listener -Proxy[TemperatureListener,
RemoteObjectInvocationHandler[UnicastRef [liveRef:
[endpoint:[129.104.254.97:44615](remote),
objID:[6fc82051:14afdeee0b0:-7ffe, -6123071618817864736]]]]
```

Les températures et événements sont synchronisés avec l'autre client sur loire.

Si on interrompt sur allemagne par Ctrl C, sur cher

```
removing listener -Proxy[TemperatureListener,
RemoteObjectInvocationHandler[UnicastRef [liveRef:
[endpoint:[129.104.254.97:44615](remote),
objID:[6fc82051:14afdeee0b0:-7ffe, -6123071618817864736]]]]
```

L'exécution continue sur loire.

La suite

- TD à 10h15: RMI
- La semaine prochaine: algorithmique sur anneau
- Choix des projets pour la semaine prochaine