RMI : Remote Method Invocation
Appel de méthodes à distance

Patrice Torguet

torguet@irit.fr

Université Paul Sabatier

Plan du cours

- Les RPC
- Architecture et fonctionnement de RMI
- Etapes de développement et de mise en place
- Paramètres des méthodes
- Objets distants temporaires et ramasse miettes distribué
- Interopérabilité et pare-feux
- Objet Activable
- Travaux pratiques

RPC : Remote Procedure Call

- Modèle client/serveur
- Appel de procédures à distance entre un client et un serveur
  - Le client appelle une procédure locale (souche – stub – ou proxy)
  - La procédure locale utilise une connexion socket pour envoyer un identifiant de procédure et les paramètres au serveur
  - Le serveur reçoit la requête grâce à un socket, extrait l'id de procédure et les paramètres (ceci est fait par un squelette – skeletton)
  - Le squelette exécute la procédure et renvoie le résultat via le socket
- Outil rpcgen
  - Génère la souche et le squelette à partir d'un fichier présentant l'interface des méthodes dans un format indépendant du langage (RPCL : RPC Language)

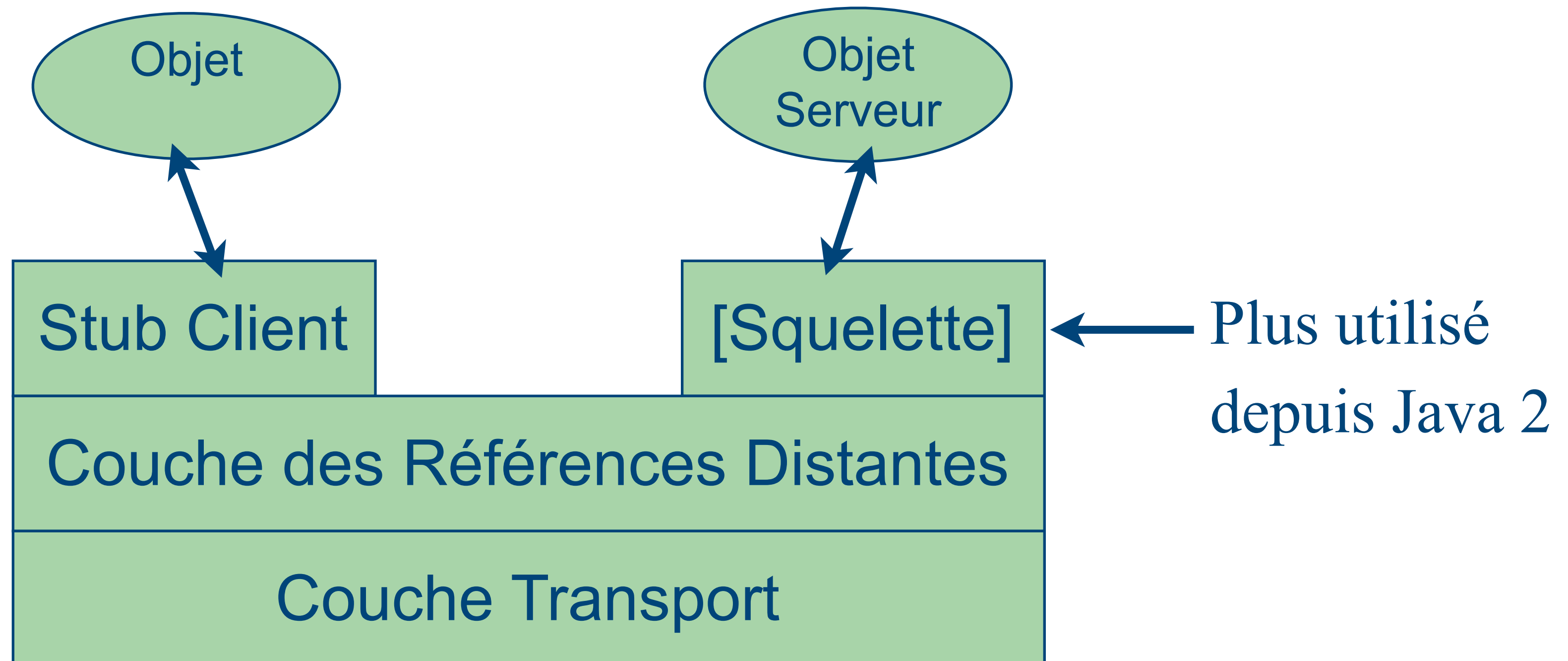
RPC : Remote Procedure Call

- Couche de présentation XDR (eXternal Data Representation)
  - Format pivot de représentation des données de types primitifs et structurés (tableaux, tableaux de taille variable, structures…)
  - Indépendant de
    - L'architecture (little endian/big endian)
    - Le langage (ordre ligne/colonne dans les tableaux C et Fortran)
    - Le système (ASCII, EBCDIC)
- Limitation :
  - Pas de gestion des concepts objets (encapsulation, héritage, polymorphisme)
  - Pas de services évolués : nommage …
- Successeurs :
  - RMI : mono langage, multi plateforme
  - CORBA : multi langage, multi plateforme
  - COM : multi langage, mono plateforme (multi pour DCOM)
  - SOAP / .NET / web services : multi langage, multi plateforme

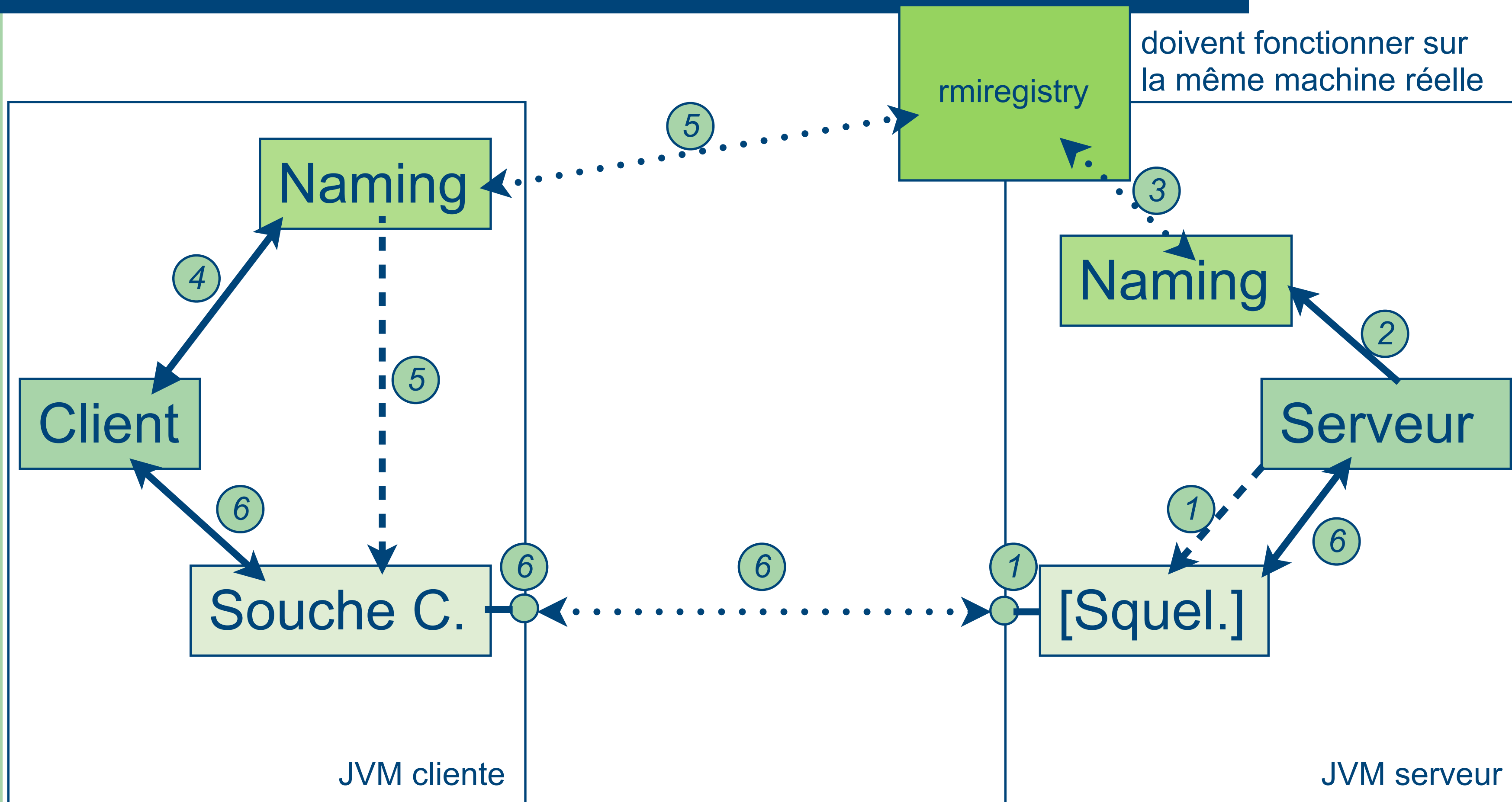
Architecture et fonctionnement de RMI

- RPC en Java
  - Invocation de méthodes sur des objets distribués
  - Très lié au langage => simple à mettre en œuvre
- Outils
  - Génération des souches/squelettes
  - Service de nommage simple (précise la machine d'exécution)
  - Activation
- Mono langage et multi plateforme : de JVM à JVM
- Orienté objet : utilisation de la sérialisation
- Dynamique : téléchargement des classes des souches et des paramètres via HTTP (http://) ou NFS/SMB (file:/)
- Sécurisé : SecurityManager et fichier de sécurité .policy

Architecture et fonctionnement de RMI



Architecture et fonctionnement de RMI



Architecture et fonctionnement de RMI

- Fonctionnement côté serveur :
  - 1) L'objet serveur est créé, si nécessaire il crée l’objet squelette (avant Java 2), puis le port de communication est ouvert
  - 2) L'objet serveur s'enregistre auprès du service de noms RMI via la classe Naming de sa JVM (méthode bind ou rebind)
  - 3) Le Naming enregistre le nom de l’objet serveur et une souche client (contenant l’@IP et le port de comm.) dans le rmiregistry

Architecture et fonctionnement de RMI

- Fonctionnement côté client :
  - 4) L'objet client fait appel au Naming de sa JVM pour localiser l'objet serveur (méthode lookup)
  - 5) Le Naming récupère la souche client de l'objet serveur, …
  - 6) Le client appelle des méthodes de l'objet serveur au travers de la souche, qui crée un port de communication, et du squelette éventuel.

Etapes de développement

1- Définition de l'interface de l'objet distant :

- interface héritant de java.rmi.Remote
- méthodes : "throws java.rmi.RemoteException"
- paramètres de type simple, objets Sérialisables (implements Serializable) ou Distants (implements Remote)

2- Ecrire une implémentation :

- classe héritant de java.rmi.server.UnicastRemoteObject et implémentant l'interface précédente.
- écrire un main permettant l'enregistrement auprès du Naming

3- Ecriture d'un client

- utilisation du Naming pour trouver l'objet distant
- appel(s) de méthodes

Etapes de développement

- Interface : Hello.java

```
public interface Hello extends java.rmi.Remote {
    String sayHello() throws java.rmi.RemoteException;
}
```

Etapes de développement

- Implémentation de l'objet serveur : HelloImpl.java

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

public class HelloImpl extends UnicastRemoteObject implements Hello
{
    private String name;

    public HelloImpl(String s) throws RemoteException {
      super();
      name = s;
    }
    …
```

Etapes de développement

- Implémentation de l'objet serveur : HelloImpl.java

```
// Implémentation de la méthode distante
public String sayHello() throws RemoteException {
    return "Hello from " + name;
}
```

…

Etapes de développement

- Implémentation de l'objet serveur : HelloImpl.java

```
public static void main(String args[]) {
// Crée et installe un gestionnaire de sécurité
 // inutile si on ne télécharge pas les classes des stubs et parametres
// System.setSecurityManager(new RMISecurityManager());
try {
    HelloImpl obj = new HelloImpl("HelloServer");
    Naming.rebind("HelloServer", obj);
    System.out.println("HelloServer déclaré auprès du serveur de noms");
} catch (Exception e) {
    System.out.println("HelloImpl erreur : " + e.getMessage());
    e.printStackTrace();
}
}
}
```

Etapes de développement

- Implémentation de l'objet serveur : HelloImpl.java
  - **Note :** si on ne veut/peut pas faire dériver la classe de l'objet serveur de UnicastRemoteObject on peut exporter l'objet avec la méthode **UnicastRemoteObject.exportObject(objServeur, 0); // le second paramètre est le numéro de port utilisé par le ServerSocket - ici on laisse le système choisir un port libre.**

Etapes de développement

- Client : HelloClnt.java

```
import java.rmi.*;
public class HelloClnt {
    public static void main(String args[]) {
      try {
                // Récupération d'un proxy sur l'objet
                Hello obj =
                        (Hello) Naming.lookup("//machine/HelloServer");
                // Appel d'une méthode sur l'objet distant
                String message = obj.sayHello();
                System.out.println(message);
      } catch (Exception e) {
                e.printStackTrace();
      }
    } }
```

Mise en place

- Compilation des classes
- Génération de la souche et du squelette (inutile pour une utilisation simple en Java 2)
  - Compilateur rmic : **rmic HelloImpl**
  - Génération de HelloImpl_Stub.class (Java 2) et HelloImpl_Skel.class (avant Java 2)
- Lancement du serveur de nom (rmiregistry)
- Note : on peut aussi appeler dans le main serveur
  - LocateRegistry.createRegistry(1099);

Mise en place

- Mise en place de l'objet d'implémentation (simple)

  Java 2 : java HelloImpl

  Avant : java -Djava.security.policy=java.policy HelloImpl

- Contenu de java.policy

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

Mise en place

- Mise en place de l'objet d'implémentation avec téléchargement de classe de paramètre

```
java -Djava.security.policy=java.policy
   -Djava.rmi.server.codebase=http://hostwww/myclasses/
   HelloImpl
```

- Contenu de java.policy

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

Mise en place

- Lancement du client
  - Simple : java HelloClnt
  - Avec téléchargement de classes :

    java HelloClnt -Djava.security.policy=./client.policy -Djava.rmi.server.codebase=http://hostwww/myclasses/ HelloClient
- Contenu de client.policy

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

Paramètres des méthodes

- Les paramètres et résultats des méthodes distantes peuvent être :
  - Des valeurs de types primitifs
  - Des objets de classes implémentant l'interface Serializable
    - L'objet est sérialisé par la souche et envoyé au squelette pour désérialisation
    - Le passage de paramètre est donc réalisé par copie !
  - Des objets de classes implémentant l'interface Remote
    - C'est la souche qui est sérialisée et envoyée
- Sinon exception : not serializable exception

Paramètres des méthodes

- Si la classe d’un paramètre est inconnue du serveur (classe dérivée de la classe d’un paramètre) :
- Lors de la désérialisation, la JVM va télécharger automatiquement le .class depuis le java.rmi.server.codebase

Objets distants temporaires et DGC

- Passage d'une souche en paramètre :
  - La souche correspond à un objet d'implémentation enregistré auprès du rmiregistry (choix d'un nom)
- On utilise un objet distant temporaire
  - L'objet d'implémentation n'est pas nommé et il n'est utilisé que pour la durée d'une ou plusieurs invocations distantes
  - L'objet peut implémenter l'interface java.rmi.server.Unreferenced pour être prévenu lors de sa récupération par le DGC

Objets distants temporaires et DGC

- Interface Personne

```
public interface Personne extends Remote {
    public String getNom() throws RemoteException;
    public String getPrénom() throws RemoteException;
}
```

Objets distants temporaires et DGC

- Classe d'implémentation

```
public class PersonneImpl extends UnicastRemoteObject
                                    implements Personne, Unreferenced {
    private String nom; private String prénom;
    public PersonneImpl(String p, String n) throws RemoteException {
      super(); prénom = p; nom = n; }
    public String getNom() throws RemoteException {
      return nom; }
    public String getPrénom() throws RemoteException {
      return prénom; }
    public void unreferenced() {
      // utilisé pour libérer des ressources (threads, fichiers…)
    }
}
```

Objets distants temporaires et DGC

- Retour sur Hello :

```
public interface Hello extends Remote {
    String sayHello(Personne p) throws RemoteException;
}
public class HelloImpl extends UnicastRemoteObject implements Hello {
    …
    String sayHello(Personne p) throws RemoteException {
      return "Hello "+p.getPrénom()+" "+p.getNom();
    }
    …
}
```

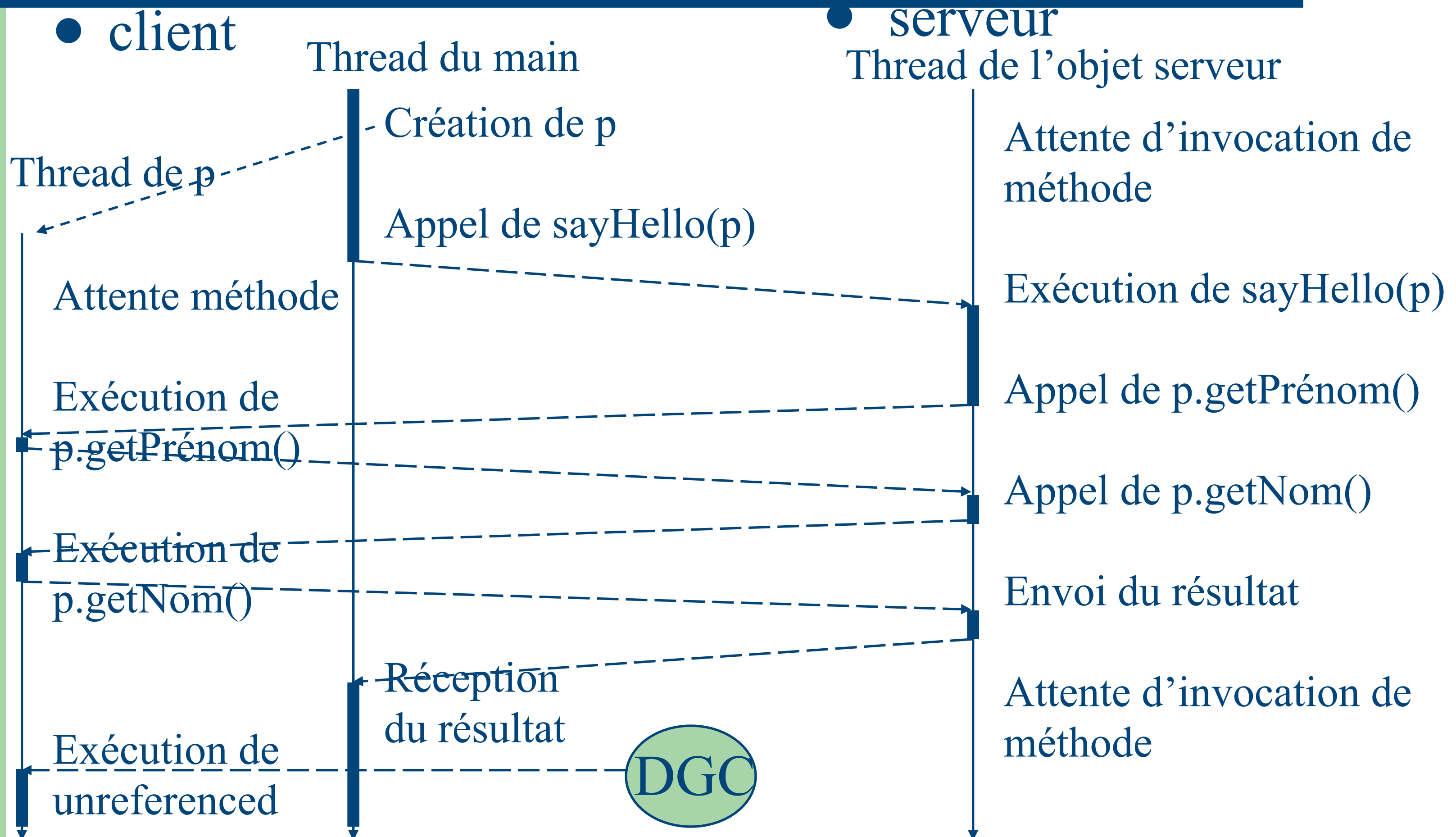
Objets distants temporaires et DGC

- Retour sur Hello :

```
public class HelloClnt {
    public static void main(String args[]) {
        try {
    Hello obj = (Hello) Naming.lookup("//machine/HelloServer");
    for(int i=0; i < 1000; i++) {
            PersonneImpl p = new PersonneImpl("Albert", "Einstein");
            String message = obj.sayHello(p);
            System.out.println(message);
            p = null; // force le GC local et le GC réparti
    }
        } …
```

Ramasse miettes réparti (DGC)

- client
- serveur



Gestion des threads

- Attention : RMI peut utiliser plusieurs threads pour l'invocation des méthodes.
- Il faut donc protéger les attributs et autres ressources partagées avec des synchronized

Traversée des pare-feu (RMI over HTTP)

- RMI sait encapsuler ses requêtes/réponses dans des messages HTTP POST en utilisant un proxy HTTP pour traverser un pare-feu
- 2 solutions
  - Soit le proxy HTTP dialogue directement avec le serveur RMI
  - Soit le proxy HTTP dialogue avec un CGI qui dialogue avec le serveur RMI
- Le fonctionnement est automatique : si un dialogue direct ne marche pas, RMI tente un passage par proxy HTTP

Traversée des pare-feu (RMI over HTTP)

- Si on veut désactiver ce mécanisme on peut lancer un client avec l'option : -Djava.rmi.server.disableHttp=true
- Côté serveur, il faut que RMI puisse connaître le nom complet de la machine serveur. Si le système ne le permet pas on peut le préciser manuellement en utilisant l'option : -Djava.rmi.server.hostname=chatsubo.javasoft.com

Traversée des pare-feu (RMI over HTTP)

- Pour utiliser la seconde solution il faut sur la machine serveur :
  - un serveur web sur le port 80 ;
  - un script CGI mappé ainsi : /cgi-bin/java-rmi.cgi
  - ce script doit :
    - lancer java et une classe interne à RMI qui appelle le serveur RMI
    - mappe les variables CGI dans des variables pour java
  - il y a un exemple de ce script dans le JDK pour Solaris et Windows
- Ce mécanisme est bien entendu beaucoup moins efficace que RMI sur TCP

Interopérabilité avec CORBA (RMI over IIOP)

- Si on veut utiliser RMI avec d'autres langages que Java ou avec des applications compatibles CORBA on peut utiliser RMI over IIOP
- Au lieu de dériver de UnicastRemoteObject on doit dériver de javax.rmi.PortableRemoteObject
- Dans le main on doit remplacer Naming.bind par :

```
Context initialNamingContext = new InitialContext();
initialNamingContext.rebind("HelloServer", obj);
// Context et InitialContext sont dans javax.naming
```

Interopérabilité avec CORBA (RMI over IIOP)

- Côté client on fera :

```
Context ic = new InitialContext();
Object objref = ic.lookup("HelloServer"); // référence distante
Hello hi = (Hello) PortableRemoteObject.narrow(
                objref, Hello.class); // trans-typage CORBA
String message = hi.sayHello();
```

- Pour générer les stubs/squelettes on utilise

```
rmic -iiop HelloImpl
```

Interopérabilité avec CORBA (RMI over IIOP)

- Pour exécuter il faut :
  - Lancer le serveur de noms

    orbd -ORBInitialPort 1050&
  - Lancer le serveur

    java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory -Djava.naming.provider.url=iiop://localhost:1050 -classpath . HelloImpl
  - Lancer le client

    java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory -Djava.naming.provider.url=iiop://localhost:1050 -classpath . HelloClient

Interopérabilité avec CORBA (RMI over IIOP)

- Pour l'interopérabilité on doit générer un fichier IDL (Interface Definition Language) en faisant
  - rmic -idl HelloImpl
- Il est aussi possible d'utiliser un POA (Portable Object Adapter) CORBA
- RMI over IIOP est utilisé dans Java Enterprise Edition pour les EJB (Entreprise Java Beans) avec une interface Remote
- Cf cours de Master 2

Objet Activable

- Principe du démon inetd d'Unix
  - Le démon rmid démarre une JVM qui sert l'objet d'implémentation uniquement au moment de l'invocation d'une méthode (à la demande) ou au démarrage de la machine
  - Package java.rmi.activation
    - La classe d'un objet activable doit dériver de Activatable
    - La classe doit déclarer un constructeur à 2 args :
      - java.rmi.activation.ActivationId,
      - java.rmi.MarshalledObject (paramètre de mise en place)

Objet Activable

- Implémentation de l'objet serveur : HelloImpl.java

```
import java.rmi.*;
import java.rmi.activation.*;
public class HelloImpl extends Activatable implements Hello
{
    private String name;
    public HelloImpl(ActivationID id, MarshalledObject o)
      throws RemoteException {
                super(id, 0);
                name = (String)o.get();
    }
    …
```

Objet Activable

- Programme de mise en place
  - Donne l'information nécessaire à l'activation à rmid
  - Enregistre le nom de l'objet auprès du rmiregistry
  - ActivationDesc : description des informations nécessaires à rmid
  - Groupes d'objets activables
    - rmid crée une JVM par groupe
    - Les objets du même groupe partagent la même JVM
      - ActivationGroup : groupe d'objets activables
      - ActivationGroupDesc : description du groupe
      - ActivationGroupId : identifiant du groupe

Objet Activable

```
import java.rmi.*; import java.rmi.activation.*; import java.util.Properties;
public class Setup {
    public static void main(String[] args) throws Exception {
      // fichier .policy qui sera utilisé par la JVM activée
      Properties props = new Properties();
      props.put("java.security.policy", "/home/test/java.policy");
      // Permet de préciser des paramètres à la JVM et de choisir la JVM
      ActivationGroupDesc.CommandEnvironment ace = null;
      // Descripteur du groupe
      ActivationGroupDesc exampleGroup = new ActivationGroupDesc(props,
          ace);
      // Enregistrement du groupe
      ActivationGroupID agi =
          ActivationGroup.getSystem().registerGroup(exampleGroup);
```

Objet Activable

```
// Endroit où se trouve la classe de l'objet activable
String location = "file:/home/test/";
// L'objet passé en paramètre du constructeur
MarshalledObject data = new MarshalledObject(new String("serveur"));
// Création du descripteur de l'objet
ActivationDesc desc = new ActivationDesc (agi, "HelloImpl", location,
    data);
// Enregistrement de l'objet activable ; récupération d'une souche
Hello obj = (Hello)Activatable.register(desc);
// Enregistrement de la souche auprès du rmiregistry
Naming.rebind("//machine/HelloServer", obj);
System.exit(0);
} }
```

Objet Activable

- Lancement :
  - lancer rmiregistry
  - lancer rmid –J-Djava.security.policy=java.policy
  - lancer le programme de mise en place (Setup)

  Java –Djava.security.policy=java.policy
  –Djava.rmi.server.codebase=file:/home/test/
  Setup
  - lancer le client

  => rmid crèe automatiquement la JVM + l'objet serveur

Exercices

- But du TD/TP : application répartie permettant de gérer des comptes bancaires.
- Un serveur gérera tous les comptes bancaires et permettra à des clients de se connecter et d'effectuer les opérations suivantes :
  - void creer_compte(String id, double somme_initiale);
  - void ajouter(String id, double somme);
  - void retirer(String id, double somme);
  - Position position(String id);

Exercices

- Écrire l'interface Banque, avec les méthodes :
  - `void creer_compte(String id, double somme_initiale);`
  - `void ajouter(String id, double somme);`
  - `void retirer(String id, double somme);`
  - `Position position(String id);`
  - Position est la classe suivante (à compléter) :

```
public class Position {
  private double solde;
  private Date derniereOperation;
  public Position(double solde) {
    this.solde = solde;
    this.derniereOperation = new Date();
  }
}
```

Exercices

- Écrire une interface Banque dérivant de Remote qui déclare les méthodes distantes.

- Écrire la classe Compte qui permet de consulter la position d'un compte, d'ajouter et de retirer une somme à un compte.

- Écrire une classe BanqueImpl qui gère la partie serveur de notre application répartie. Les comptes seront stockés dans une Hashtable qui permettra de retrouver un compte à partir de son identification.

Exercices

- Écrire une classe BanqueClient qui gère la partie client de notre application répartie. L'application présentera un petit menu (sous forme textuelle) permettant d'accéder aux diverses méthodes.

- On veut maintenant que le serveur, prévienne le client quand le solde de son compte devient négatif ou inférieur à une valeur choisie par le client.
  - Quel mécanisme, vu en cours, peut être utilisé ?
  - Modifiez l'interface et les classes pour gérer cette situation.