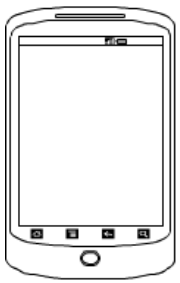
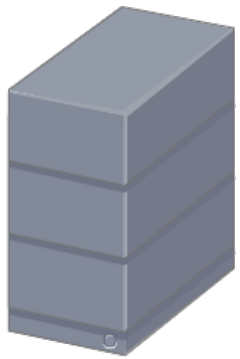
Programmation Répartie : Programmation client/serveur Remote Method Invocation

Application sur un terminal léger ?

- Terminal disposant de ressource limitée
- Serveur disposant de ressource suffisante
- Application s'exécutant sur le terminal malgré les ressources limitées. Ce que l'on souhaite :
  - Les calculs lourds sont faits sur le serveur et seul le résultat est retourné sur le terminal
  - Utilisation de la programmation objet habituelle





Utilisation de socket ?

- Communiquer par socket
- Nécessite de transférer des données (les données doivent être sérialisables)
- Le serveur doit savoir quoi faire lorsqu'il reçoit ces données
- Change la manière de programmer
- Difficile de travailler sur un objet modifiable

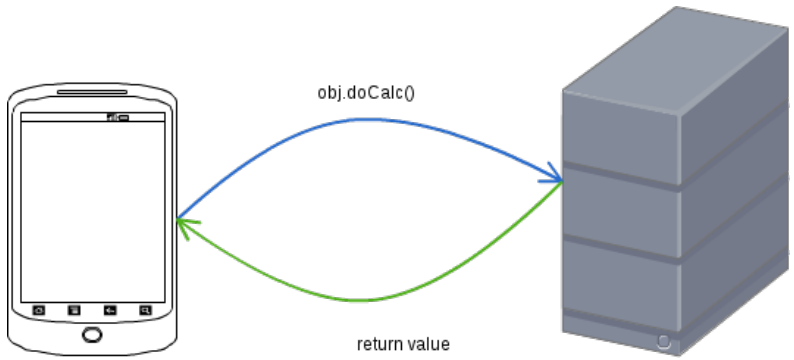
Utilisation d'objet distant

- Récupérer un **objet distant**
- Une fois récupéré, l'objet se comporte comme un objet classique $\Rightarrow$ facilité d'utilisation, ne change pas la manière de programmer
- Les **méthodes** appelées sur cet objet sont exécutées à **distance**

Remote Method Invocation

RMI

- Mécanisme de Java
- Permettre l'utilisation d'objet à distance : les objets sont créés sur le **serveur**, et les clients peuvent utiliser les méthodes sur ces objets
- **RMI** fournit les **mécanismes pour la communication** entre le client et le serveur
- **Application distribuée** : une partie de l'application s'exécute sur le serveur, l'autre sur le client



3 entités

- **Serveur** :
  - Créer les objets
  - Les rend accessibles, en les enregistrant dans le registre
- **Registre** : Permet de retrouver les objets distants
- **Client** :
  - Localiser les objets distants via le registre
  - Appel des méthodes sur l'objet distant

Serveur : Interface des objets distants

Chaque objet distant doit implanter une interface qui définit les méthodes appelables à distance

Interface des objets distants

- Définition des **signatures** des méthodes appelables à **distance**
- L'interface doit hériter de l'interface **java.rmi.Remote** pour pouvoir être utilisée à distance
- Chaque méthode peut échouer à cause d'une erreur réseau, du serveur... Chaque méthode doit donc déclarer qu'elle peut lever une **exception** `java.rmi.RemoteException`. Ainsi le client peut être informé de ce type d'erreur

Cette interface servira aux clients pour connaître les méthodes distantes

Interface

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

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

Définition de l'objet implantant cette interface

Objet distant

- L'objet doit implanter l'interface
- Définition du corps des méthodes

Type de données manipulable à distance

- Les types primitifs
- Les objets sérialisables

Objet appelable à distance

```
public class Server implements Hello {

    public Server() {}

    public String sayHello() {
        return "Hello, world!";
    }
}
```

Registre

Comment les objets distants peuvent-ils être découvert ?

Registre : rôle

- Fait le lien entre un nom et un objet
- Le serveur enregistre son objet avec un nom (il doit être unique)
- Le client découvre cet objet avec ce nom

Registre : utilisation

- Interface `Registry`
- Méthodes
  - bind(name,obj) : associe un nom à un objet
  - lookup(name) : recherche l'objet au nom spécifié en paramètre
- La classe `LocateRegistry` permet de récupérer ou de créer des registres.
  Méthodes statiques :
  - `getRegistry` : récupère le registre en spécifiant le nom de l'hôte (par défaut recherche en local)
  - `createRegistry` : créer un registre

Registre et Serveur

Création et Exportation de l'objet

- L'objet est **créé** de manière classique
- L'objet est **transformé** par la méthode `exportObject` de la classe `UnicastRemoteObjet`, afin qu'il puisse recevoir des informations des clients distants
  - La méthode prend deux paramètres :
    - une instance de l'objet distant
    - le numéro de port TCP utilisé pour communiquer. Si le numéro de port est 0, le numéro de port de RMI par défaut est utilisé (1099)
  - Une autre solution est de faire hériter l'objet de `UnicastRemoteObjet`, mais on perd la capacité de faire hériter d'autres classes

```
Server obj = new Server();
Hello stub =(Hello) UnicastRemoteObject.exportObject(obj, 0);
```

Enregistrement de l'objet

- Récupération du registre
- **Enregistrement** de l'objet dans le registre de nom RMI avec la méthode `bind` de l'objet de type `Registery`, en spécifiant le nom par lequel il sera trouvable
  - Un unique objet est associé à un nom (une exception est levée si on tente d'associer un objet à un nom déja associé)
  - la méthode `rebind` remplace une association si elle existe (la crée sinon)

```
registry.bind(''Hello'', stub);
```

Méthode principale du serveur : Création et Enregistrement de l'objet

```
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.server.UnicastRemoteObject;

public class MainServer{
public static void main(String args[]) {
 try {
    Server obj = new Server();
    // objet exportable
    Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
    // Lie l'objet au registre
    Registry registry = LocateRegistry.getRegistry();
    registry.bind("Hello", stub);

    System.err.println("Server ready");
 }
 catch (Exception e) {
       System.err.println("Server exception: " + e.toString());
       e.printStackTrace();
 }
}
}
```

Client

Le client doit disposer

- du **nom** avec lequel l'objet distant est enregistré dans le registre
- de l'**interface** définissant les méthodes appelables
- de **l'adresse** ou du **nom** de la machine où s'exécute le registre

Client

Comment récupérer l'objet ?

- Le client **récupère le registre** de nom avec l'adresse (ou le nom) de l'hôte
- Le client **recherche dans le registre** l'objet distant grâce à son nom avec la méthode `lookup` de l'objet de type `Registry` récupéré à l'étape précédente. Cette méthode retourne une référence sur l'objet distant de type `Remote`
- Transformation du type de l'objet (**cast**) avec le type de l'interface de l'objet distant

Les méthodes sur cet objet distant peuvent être appelées comme avec un objet classique

Client (extrait)

```
Registry registry = LocateRegistry.getRegistry(host);
Hello stub = (Hello) registry.lookup("Hello");
```

Client

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {

    public static void main(String[] args) {

    String host = (args.length < 1) ? null : args[0];
    try {
        Registry registry = LocateRegistry.getRegistry(host);
        Hello stub = (Hello) registry.lookup("Hello");
        String response = stub.sayHello();
        System.out.println("response: " + response);
    }
    catch (Exception e) {
      System.err.println("Client exception: " + e.toString());
      e.printStackTrace();
    }
   }
}
```

Compilation

- Compilation des classes du serveur
- Compilation des classes clientes, avec l'interface de l'objet distant, afin que le client puisse connaître les méthodes utilisables sur l'objet distant

Pour le serveur :

```
javac Hello.java Server.java MainServer.java
```

Pour le client :

```
javac Client.java Hello.java
```

Exécution

- **Registre** : Créer et lancer le registre avec la commande `rmiregistry`.

```
rmiregistry
```

- **Serveur** : lancer le serveur en lui spécifiant son adresse avec l'option `Djava.rmi.server.hostname`

```
java -Djava.rmi.server.hostname=adresseServeur MainServ
```

- **Client** : exécution classique

```
java Client
```

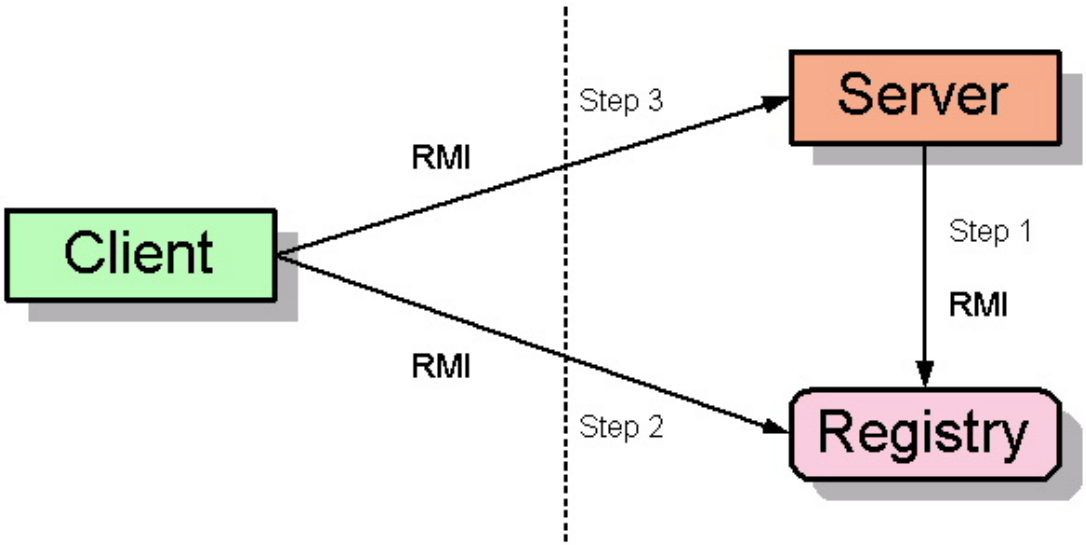
Dans le cas particulier de l'exemple, on spécifie l'adresse du serveur au client à l'exécution `java Client adresseDuServeur`

Résumé

Conception

- Concevoir une interface distante
- Concevoir un objet distant implantant cette interface
- Exporter et enregistrer cet objet dans un registre
- Concevoir le client : Récupération du registre puis de l'objet distant

Résumé



Conclusion

- Permet la création d'application répartie entre plusieurs JVM
  - Déport de calcul lourd
  - Accès à des ressources sensibles
  - Partage d'objets entre JVM
  - ...
- Utilisation comme si l'objet distant est local
- Mise à jour des méthodes transparentes vis à vis des clients
- Le client n'a pas besoin de connaître les adresses des serveurs, uniquement celle du registre

- Services web offrent un service similaire, mais avec interopérabilité

Liens

- http://docs.oracle.com/javase/tutorial/rmi/
- http://docs.oracle.com/javase/6/docs/technotes/guides/rmi/