Institut Informatique
Claude Chappe

Master Informatique

Programmation distribuée
M1 / 178UD02

C2 – Java RMI

Thierry Lemeunier
thierry.lemeunier@univ-lemans.fr
www-lium.univ-lemans.fr/~lemeunie

Plan du cours

- Communication par objet distribué :
  - Principes
  - Référenciation des objets distants
  - Passage des paramètres
  - Java *RMI*
    - Principes
    - Implémentation
    - Compléments (code mobile, sécurité, etc.)

Plan du cours

- Communication par objet distribué :
  - Principes
  - Référenciation des objets distants
  - Passage des paramètres
  - Java *RMI*
    - Principes
    - Implémentation
    - Compléments (code mobile, sécurité, etc.)

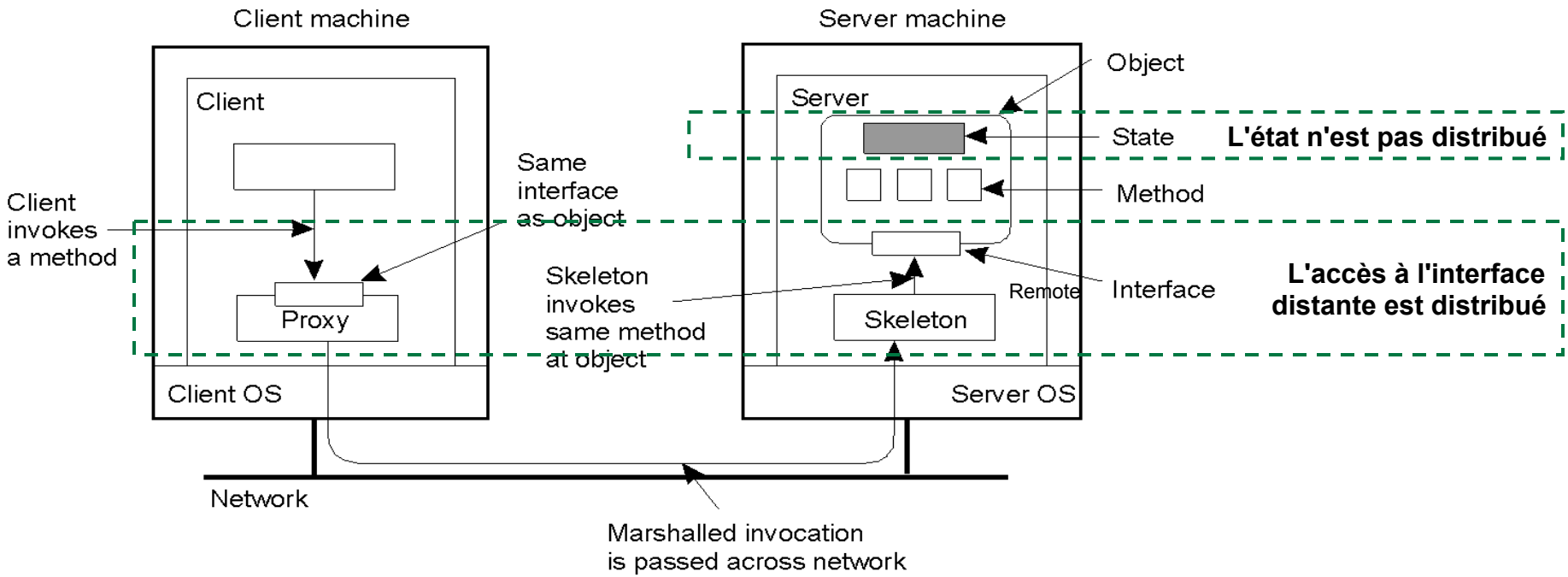
Objet distribué – Principes (1/2)

- Principes :
  - Donner la possibilité à une machine cliente d'invoquer une méthode sur un objet distant se trouvant sur une machine serveur
  - ➔ *RMI* : *Remote Method Invocation*
- Philosophie : faire en sorte qu'un client puisse utiliser un objet distant comme s'il s'agit d'un objet local
  - ➔ Transparence de la distribution des objets et de la communication
- Impact :
  - RMI est le second *middleware* a avoir été défini après RPC (années 1995)
  - Il est très répandu car il est associé à Java

- Rappel :
  - Un objet est constitué d'attributs et de méthodes
  - Les valeurs des attributs forment l'état de l'objet
  - Sérialiser un objet : extraire l'état de l'objet et en faire une suite d'informations
  - Dé-sérialiser un objet : créer un objet à partir de la suite d'informations **et de sa classe**

Objet distribué – Principes (2/2)

- La transparence s'effectue via la distribution
  - de l'interface distante de l'objet distant sur le client = le *proxy*
  - et de l'exportation de l'interface distante sur le serveur = le *skeleton*
- Le *proxy* et le *skeleton* joue le rôle d'emballage et de déballage de l'invocation de méthode en message et gestion de l'envoi de ce message



Plan du cours

- Communication par objet distribué :
  - ✓ Principes
  - ➢ Référenciation des objets distants
  - ❑ Passage des paramètres
  - ❑ Java *RMI*
    - Principes
    - Implémentation
    - Compléments (code mobile, sécurité, etc.)

Objet distribué – Référenciation (1/2)

- Pour utiliser un objet distribué, il faut :
  - Localiser la machine hébergeant le serveur d'objet (adresse + port)
  - Localiser l'objet dans la mémoire du serveur d'objet : connaître son nom

- Le processus de référenciation (rendre accessible un objet distant) :
  - Pour le serveur d'objet, c'est créer un objet distant (**exportation**) et le rendre public aux clients via un serveur de nom (**publication**).
  - Pour le client, c'est accéder à un objet distant c'est-à-dire le trouver puis l'utiliser via son *proxy.*
  - Pour le serveur de nom, c'est maintenir une table de localisation faisant le lien entre le nom d'un objet distant et un proxy sur l'objet distant.

- Le serveur de nom communique avec :
  - Le serveur d'objet qui fait une **requête d'enregistrement** en donnant un nom d'objet et un proxy sérialisé (qui connaît la localisation du *skeleton*).
  - Le client qui fait une **requête de localisation** en donnant un nom d'objet. Le serveur de nom lui envoie le *proxy* sérialisé. Après désérialisation par le client, le client communique uniquement avec le *proxy.*
  - Remarque : le serveur de nom n'a pas besoin de désérialiser le proxy !

Objet distribué – Référenciation (2/2)

- Il y a deux modes de référenciation :
  - Dans les deux modes, le serveur et le client doivent charger les classes nécessaires à l'exécution du *proxy*, de l'objet distant et du *skeleton*
  - Référenciation par chargement local :
    - le serveur charge localement les classes de l'objet distant et du *skeleton*
    - le client charge localement la classe du *proxy*
    - ainsi que toutes les autres classes nécessaires (paramètres ou valeur de retour)
  - Référenciation par téléchargement dynamique :
    - Pour le serveur
      - Le code = tout ce qui est nécessaire pour exécuter l'objet distant et le *skeleton*
      - Le code est téléchargé sur le serveur au moment de la création de l'objet distant ou au moment de l'exécution du code de la méthode distante
    - Pour le client
      - Le code = tout ce qui est nécessaire pour exécuter le *proxy* sur le client
      - Le code est téléchargé sur le client au moment de la première référenciation de l'objet distant (c'est-à-dire à la désérialisation du proxy)

    → Cela nécessite de paramétrer le client et le serveur

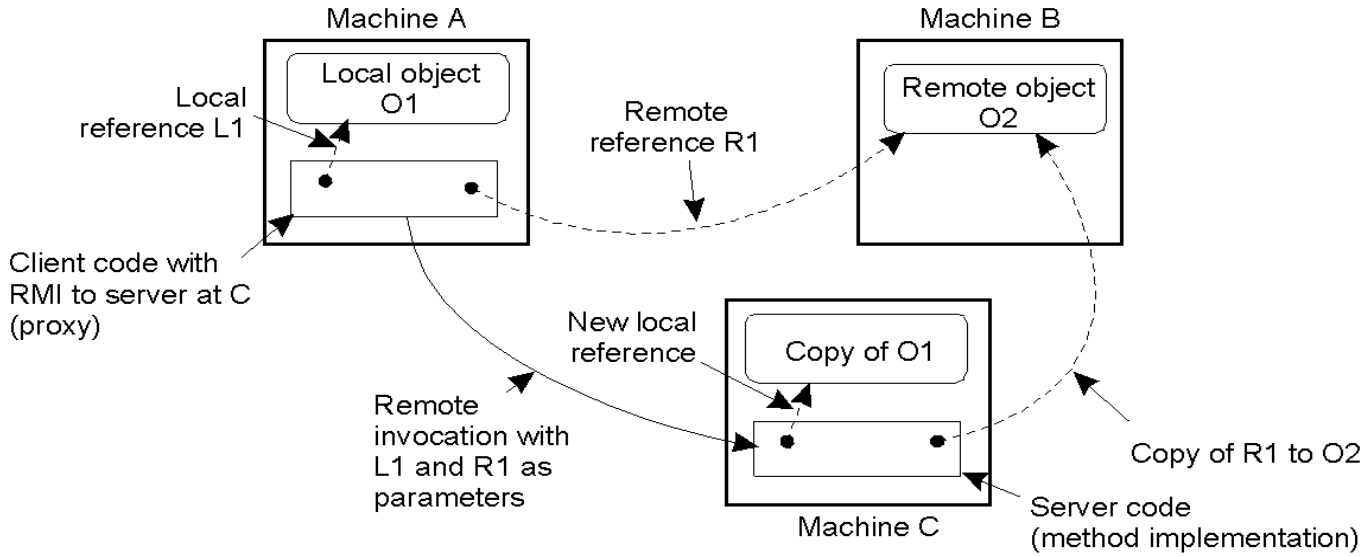
    → Cela nécessite **une politique de chargement et d'exécution** sécurisés (afin de s'assurer que le code chargé n'effectue pas d'opérations interdites !)

Plan du cours

- Communication par objet distribué :
  - ✓ Principes
  - ✓ Référenciation des objets distants
  - ➢ Passage des paramètres
  - ❑ Java *RMI*
    - Principes
    - Implémentation
    - Compléments (code mobile, sécurité, etc.)

Objet distribué – Passage des paramètres

- Les paramètres et les valeurs retournées doivent être sérialisables
- Si la méthode distante a des paramètres ou retourne une valeur :
  - Les types primitifs sont passés/retournés par valeur
  - Les objets locaux sont passés/retournés par valeur
  - Les objets distants non exportés sont passés/retournés par valeur
  - Les objets distants exportés sont passés/retournés « par référence » c'est-à-dire par copie du proxy



Plan du cours

- Communication par objet distribué :
  - ✓ Principes
  - ✓ Référenciation des objets distants
  - ✓ Passage des paramètres
  - Java *RMI*
    - Principes
    - Implémentation
    - Compléments (code mobile, sécurité, etc.)

Objet distribué – Java RMI – Principes (1/4)

- L'offre :
  - Java RMI est un *middleware* RMI gratuit développé par Oracle (anc. Sun)
  - Java RMI existe depuis le JDK1.1 de 1997 (mais a été simplifié depuis)
  - Java RMI marche avec des objets Java (ou d'autres langages : cf. CORBA) !

- Sous le capot :
  - L'API permet de faire interagir des objets créés dans des machines virtuelles différentes lancées sur une même machine ou sur des machines différentes
  - Il utilise les *sockets* (par défaut : TCP/IP non sécurisé)
  - Il utilise un protocole spécifique (*Java Remote Method Protocol*) ou CORBA
  - Il préserve la sécurité inhérente au langage Java avec notamment :
    - *RMISecurityManager*
    - *Distributed Garbage Collector* (*DGC*)

Objet distribué – Java RMI – Principes (2/4)

Le modèle d'objet distribué de Java RMI

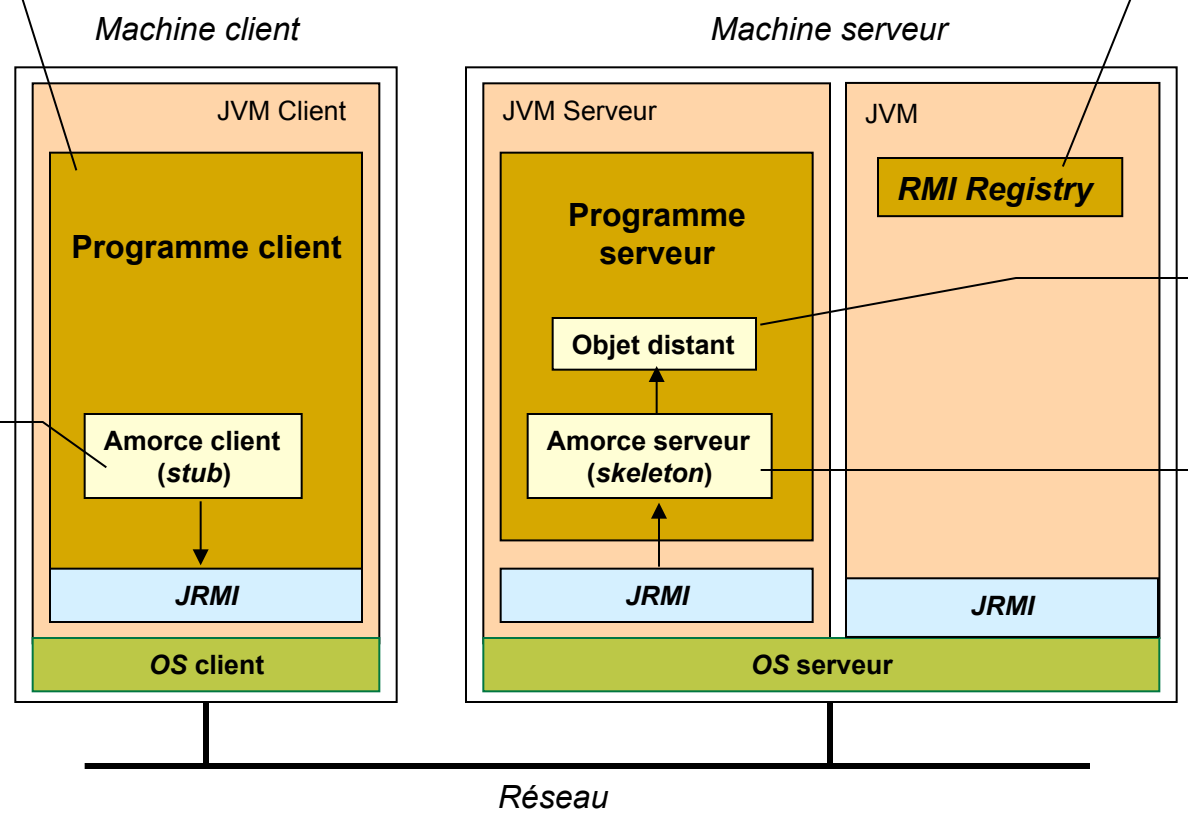
Une machine virtuelle Java exécute le programme client.

Le programme client connaît un représentant de l'objet distant : le *stub* client.

Il communique uniquement avec le *stub* client.

Le *stub* client masque le mécanisme d'appel distant pris en charge par *JRMI*.

Le *stub* et *JRMI* prennent en charge :
- l'invocation dynamique de la méthode distante
- l'emballage des paramètres
- le déballage de la valeur de retour



Serveur de nom par défaut du *middleware JRMI*

Pour des raisons de sécurité, il doit être lancé sur la même machine que celle du programme serveur

L'objet distant exporté par le programme serveur

Le *skeleton* et JRMI prennent en charge
- Le déballage des paramètres
- La sélection de la méthode appelée
- L'invocation sur l'objet distant.
- L'emballage de la valeur de retour

Objet distribué – Java RMI – Principes (3/4)

Fonctionnement côté serveur : démarrage du programme serveur

1. Exportation de l'objet distant
   - Un objet distant doit être exporté pour que les clients puissent invoquer des méthodes sur lui. L'exportation consiste à :
     - créer un *stub* de l'objet distant (qui sera publié via le service de nom) + un *skeleton*
     - et ouvrir un port d'écoute sur la machine hôte pour recevoir le flux d'invocation envoyé par le client une fois la référenciation établie
2. Publication de l'objet distant
   - La publication rend accessible l'objet distant en utilisant un service de nom
   - Un service de nom est un processus serveur qui maintien à jour une table de liaison entre le nom et le *stub* d'un objet distant
   - En pratique on ne donne donc pas accès à l'objet distant mais à son *stub*
   - Les services de nom utilisables sont :
     - Le service rmiregistry de *JRMI* : service de nom simple (sans persistance) de la machine serveur commun à tous les processus serveurs de la machine hôte
     - Un registre propre au processus : registre privé inaccessible aux autres processus serveurs de la machine hôte
     - JNDI (*Java Naming and Directory Interface*) : API fournissant une interface unique d'accès à différents services de nommage ou d'annuaires (LDAP, RMI Registry, DNS, NIS, COS Naming...)

Objet distribué – Java RMI – Principes (4/4)

Fonctionnement côté client : accès et utilisation de l'objet distant

1. Localisation de l'objet distant
   - Le client doit trouver l'objet distant pour l'utiliser
   - Il fait une requête au service de nom en indiquant le nom de l'objet distant recherché (il doit connaître la machine serveur de nom)
   - Le service de nom lui répond en lui envoyant une copie du *stub* de l'objet distant

2. Utilisation de l'objet distant
   - Une fois que le client possède un *stub* sur l'objet distant, il peut invoquer des méthodes sur ce *stub* qui implémente l'interface distante
   - Le *stub* n'exécute pas la méthode lui-même (il ne la possède pas) mais fait une invocation dynamique sur l'objet distant qu'il représente
   - Le *stub* communique avec le *skeleton*

Plan du cours

- Communication par objet distribué :
  - ✓ Principes
  - ✓ Référenciation des objets distants
  - ✓ Passage des paramètres
  - Java *RMI*
    - ✓ Principes
    - ➢ Implémentation
    - Compléments (code mobile, sécurité, etc.)

Objet distribué – Java RMI – Implémentation (1/6)

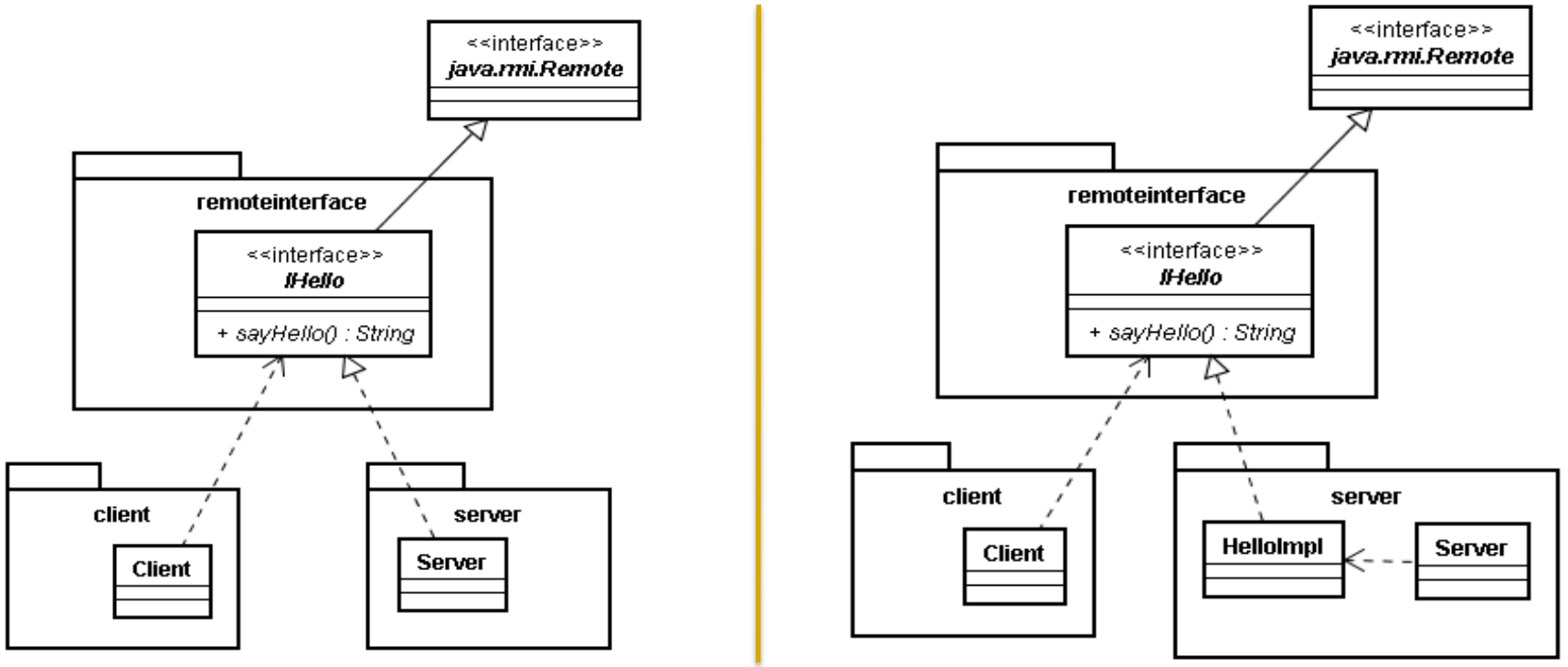
- Démarche d'implémentation
  1. Définir l'interface distante (= une classe interface Java)
  2. Ecrire une classe d'implémentation de l'interface distante pour instancier l'objet distant
  3. Implémenter un serveur qui crée, exporte et publie l'objet distant
  4. Implémenter un client qui localise et utilise l'objet distant
  5. Compiler les fichiers .java

- Démarche « classique » de déploiement manuel
  1. Générer les classes *stub* et *skeleton* avec l'outil rmic du jdk
  2. Copier le fichier *stub* sur le client et le fichier *skeleton* sur le serveur
  3. Démarrer un service de nom (optionnel quand on utilise un registre privé)
     - Exemple avec rmiregistry : par défaut ce service écoute sur le port 1099 du serveur
       Sous Windows : start rmiregistry [*num_port*]
       Sous Unix : rmiregistry [*num_port*] &
  4. Lancer le serveur
  5. Lancer le client

- Remarques :
  - Le déploiement est automatique par référenciation par téléchargement (cf. fin du cours)
  - A partir du jdk 1.2 : plus besoin de générer le *skeleton* si le serveur est >= 1.2 !
  - A partir du jdk 1.5 : plus besoin de générer le *stub* si le client est >= 1.5 !

Objet distribué – Java RMI – Implémentation (2/6)

- Un exemple simple : le « hello world » distribué
  - Deux implémentations possibles du serveur
    - Soit le serveur implémente lui-même l'interface distante (à gauche)
    - Soit le serveur utilise une classe d'implémentation (à droite)



Objet distribué – Java RMI – Implémentation (3/6)

- Définition de l'interface distante *IHello :*
  - Une interface distante étends l'interface *java.rmi.Remote*
  - Elle déclare le service rendu par le serveur en déclarant les méthodes distantes
  - Les méthodes distantes lèvent une *java.rmi.RemoteException*

```
package remoteinterface;

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

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

Objet distribué – Java RMI – Implémentation (4/6)

- Implémentation de l'interface distante
  - Il y a 2 méthodes possibles d'implémentation
    - Soit l'objet distant étend *java.rmi.server.UnicastRemoteObject*
      - ➔ *L'exportation se fait automatiquement à la création de l'objet distant*
    - Soit l'objet distant n'étend pas *UnicastRemoteObject*
      - ➔ *L'exportation doit se faire manuellement*
        - Soit dans le constructeur de l'objet distant
        - Soit au démarrage du serveur
- Exemple d'implémentation dans une classe séparée avec exportation automatique

```
package server;

import remoteinterface.IHello;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public HelloImpl extends UnicastRemoteObject implements IHello {
        public HelloImpl() throws RemoteException {};
        String sayHello() throws RemoteException { return "Hello World"; }
}
```

Objet distribué – Java RMI – Implémentation (5/6)

- Exemples d'implémentation d'un serveur
  - A gauche : le serveur utilise un objet distant qu'il crée (exportation automatique) puis publie
  - A droite : le serveur implémente l'interface (c'est l'objet distant), s'exporte manuellement et se publie

```
...
public class Server {
  public static void main(String args[]) {
    try {
      IHello stub = (IHello)new HelloImpl();
      Naming.rebind("Hello", stub);
      System.out.println("Server ready");
    } catch (Exception e) {...}
  }
}
```

```
...
public class Server implements IHello {
  public String sayHello() throws RemoteException { return "Hello World"; }
  public static void main(String args[]) {
    try {
      Server obj = new Server();
      IHello stub = (IHello)UnicastRemoteObject.exportObject(obj,0);
      Registry registry = LocateRegistry.getRegistry();
      registry.bind("Hello", stub);
      System.out.println("Server ready");
    } catch (Exception e) {...}
  }
}
```

- Remarques :
  - rmiregistery peut être accédé soit par la classe *java.rmi.registry.LocateRegistry* soit par la classe *java.rmi.Naming* (cf. fin du cours)
  - *JRMI* maintien le serveur actif tant que l'objet distant est référencé (par le client et/ou le registre)
  - Les invocations sur l'objet distant sont « threadées » pas *JRMI* (un thread est créé pour exécuter la méthode invoquée) : l'objet distant doit gérer l'accès parallèle (*thread-safe*)

Objet distribué – Java RMI – Implémentation (6/6)

- Implémentation d'un client

```
package client;
import remoteinterface.IHello;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {
    private Client() {}
    public static void main(String[] args) {
          String host = (args.length < 1) ? null : args[0];
          try {
              Registry registry = LocateRegistry.getRegistry(host);
              IHello stub = (IHello) registry.lookup("Hello");
              System.out.println("response: " + stub.sayHello());
          } catch (Exception e) {...}
    }
}
```

Plan du cours

- Communication par objet distribué :
  - ✓ Principes
  - ✓ Référenciation des objets distants
  - ✓ Passage des paramètres
  - Java *RMI*
    - ✓ Principes
    - ✓ Implémentation
    - ➢ Compléments (code mobile, sécurité, etc.)

Objet distribué – Java RMI – Compléments (1/3)

- Mobilité de code
  - Java peut charger dynamique du code dans une *JVM* depuis une URL (*file*, *http*, *ftp*)
  - *JRMI* utilise le chargement dynamiquement de tout code nécessaire dans les deux cas suivants :
    - Au moment de la localisation de l'objet distant par le client, le registre envoie une copie du flux d'octets que le client désérialise pour avoir un *stub* local
    - Au moment de l'exécution d'une méthode distante ayant pour argument ou valeur de retour un objet de classe inconnue par le serveur, celui-ci doit accéder à la classe inconnue
  - La propriété *java.rmi.server.codebase* indique l'adresse où trouver le code utile. Elle doit donc être positionnée :
    - du côté serveur pour que JRMI crée des copies du *stub* sur le client
    - et du côté client pour que JRMI transfère tout code utile pour exécuter une méthode sur le serveur
  - Exemples de démarrage d'un serveur et d'un client :

```
java –Djava.rmi.server.codebase=http://lucke.univ-lemans.fr/~lemeunie/public/code.jar server.Serveur
java –Djava.rmi.server.codebase=http://iupmime.univ-lemans.fr/~p0520/public/class/ client.Client lucke.univ-lemans.fr
```

Objet distribué – Java RMI – Compléments (2/3)

- Sécurité et mobilité de code
  - Le chargement dynamique n'est possible qu'avec la mise en place d'un gestionnaire de sécurité (gère les droits accordés au code téléchargé)
  - Il y a deux façons de gérer les restrictions :
    - Utiliser *SecurityManager* et un fichier de police
    - Définir son propre gestionnaire qui étend *SecurityManager*
  - Mise en œuvre sur le client et le serveur s'ils doivent télécharger du code
    - Installer le gestionnaire de sécurité par défaut et positionner la propriété *java.security.policy* au démarrage du programme

      Exemple : java -Djava.security.policy=allpermission.policy client.Client lucke.univ-lemans.fr

```
…
if (System.getSecurityManager() == null))
   System.setSecurityManager(new SecurityManager());
…
```

```
Fichier allpermission.policy
grant codebase "file:/tmp/appclient/" {
  permission java.security.AllPermission;
};
```

    - Ou ajouter au code du programme l'installation d'un gestionnaire particulier

```
…
if (System.getSecurityManager() == null))
   System.setSecurityManager(new MySecurityManager());
…
```

```
// Security Manager qui permet tout
public class MySecurityManager extends SecurityManager {
  public void checkPermission(Permission perm){}
}
```

Objet distribué – Java RMI – Compléments (3/3)

- Bref tour d'horizon de l'*API Java RMI*
  - Package *java.rmi*
    - Surtout utile pour le client
    - Contient les classes :
      - *Naming* : accès à un registre de référenciation via une URL (ressemble à *LocateRegistry*)
      - *RMISecurityManager* : gestion de la police de sécurité pour *JRMI*
      - La hiérarchie des exceptions liées à JRMI (héritent de *RemoteException)*
  - Package *java.rmi.server*
    - Surtout utile pour le serveur
    - Contient les classes :
      - *RemoteObject* : définition d'un objet distant
      - *UnicastRemoteObject* : exportation/dé-exportation d'un objet distant et création de *stub*
  - Package *java.rmi.registry*
    - Utile pour accéder directement à un registre commun ou créer un registre privé
    - Contient l'interface *Registry* et la classe *LocateRegistry*
  - Package *java.rmi.dgc* : classes pour le *DGC*
  - Package *java.rmi.activation* : classes pour l'activation d'objet distant