JAVA RMI

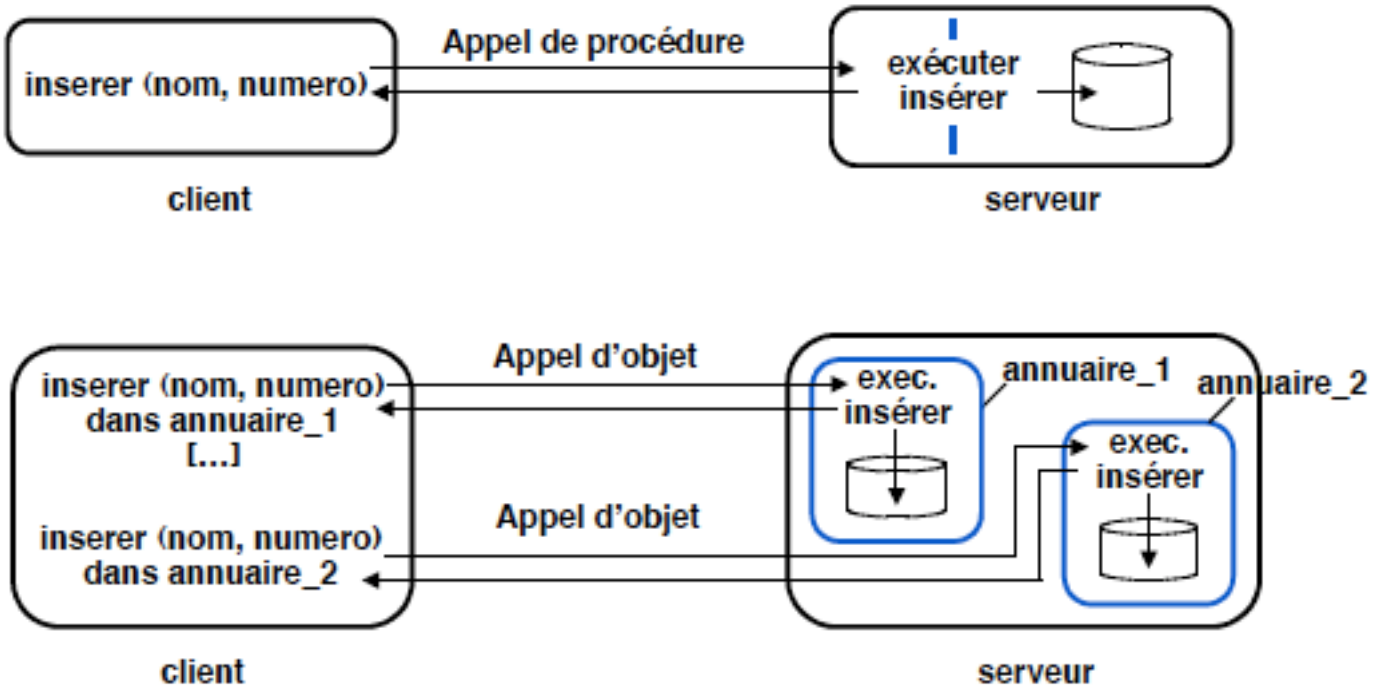
heithem.abbes@gmail.com

Intérêt des objets pour la construction d'applications réparties

- **Encapsulation**
  - L'interface (méthodes + arguments) est la seule voie d'accès à l'état interne, non directement accessible
- **Classes et instances**
  - Mécanismes de génération d'exemplaires conformes à un même modèle
- **Héritage**
  - Mécanisme de spécialisation : facilite la récupération et réutilisation de l'existant
- **Polymorphisme**
  - Mises en œuvre diverses des fonctions d'une interface
  - Facilite l'évolution et l'adaptation des applications

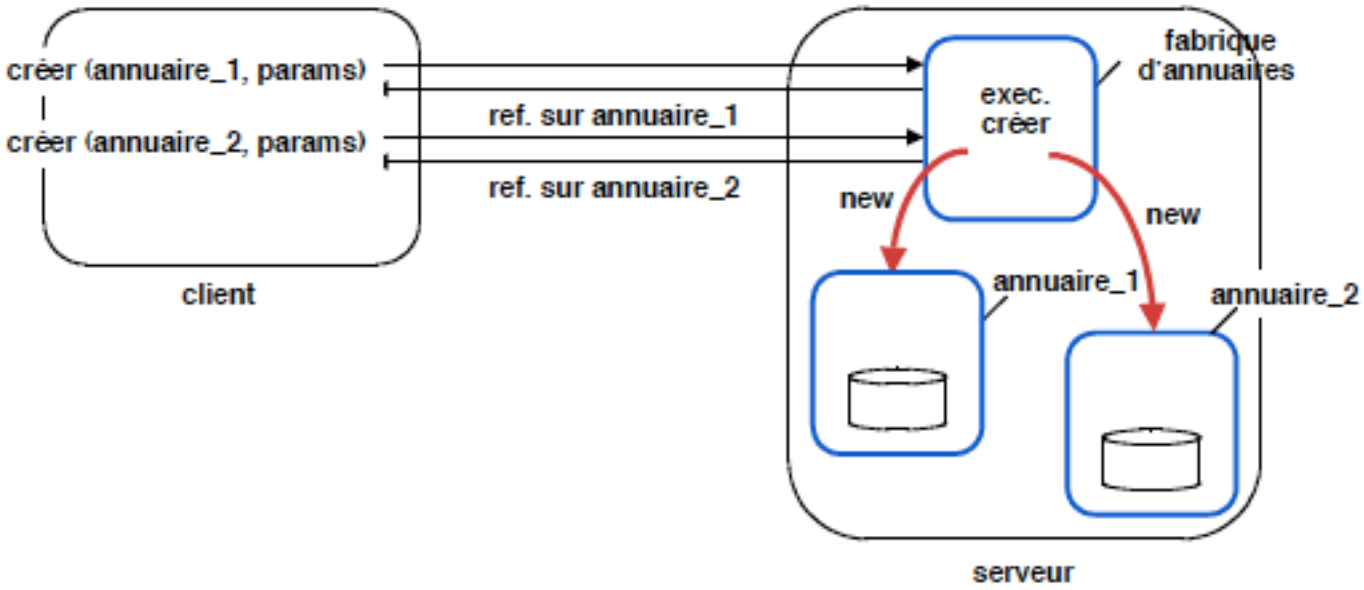
Extension du RPC aux objets (1)

- Appel de procédure vs appel de méthode sur un objet
  - Exemple : insérer une entrée dans un annuaire



Extension du RPC aux objets (2)

- Phase **préalable** : création d'instances d'une classe d'objets
  - Notion de fabrique (factory)



Notion de **référence d'objet** :

Contient tout ce qui est nécessaire pour atteindre l'objet distant (Localisation, protocole d'accès)

Java RMI (*Remote Method Invocation*)

- **Motivation** : construction d'applications réparties avec Java
  - **Appel de méthode** au lieu d'appel de procédure
- **Principe** : même schéma que RPC
  - Le programmeur fournit
    - Une (ou plusieurs) description(s) d'interface
      - Ici pas d'IDL séparé : **Java sert d'IDL**
    - Le programme du serveur
      - Objets réalisant les interfaces
      - Serveur d'objets
    - Le programme du client
- L'environnement Java fournit
  - Un générateur de talons : **rmic**
  - Un service de noms : **Object Registry**

voir **http://java.sun.com/docs/books/tutorial/rmi/**

Java RMI : règles d'usage (1/2)

- **Interface**
  - L'interface d'un objet distant (***Remote***) est celle d'un objet Java, avec quelques règles d'usage :
  - L'interface distante doit être **publique**
  - L'interface distante doit **étendre** l'interface **java.rmi.Remote**
  - Chaque méthode doit déclarer au moins l'exception **java.rmi.RemoteException**
- **Passage d'objets en paramètre**
  - Les objets **locaux** sont passés **par valeur** (copie) et doivent être sérialisables (**étendent** l'interface **java.io.Serializable**)
  - Les objets **distants** sont passés **par référence** et sont désignés par leur interface

Java RMI : règles d'usage (2/2)

- Réalisation des classes distantes (Remote)
  - Une *classe distante* doit implémenter une interface elle-même distante (Remote)
  - Une *classe distante* doit étendre la classe java.rmi.server.UnicastRemoteObject (d'autres possibilités existent)
  - Une *classe distante* peut aussi avoir des méthodes appelables seulement localement (ne font pas partie de son interface Remote)

Java RMI : règles d'écriture du serveur

- Un serveur est une classe qui implémente l'interface de l'objet distant
  - Spécifier les objets distants qui doivent être implémentées
  - Définir le constructeur de l'objet distant
  - Fournir la réalisation des méthodes appelables à distance
  - Créer et installer le gestionnaire de sécurité
  - Créer au moins une instance de la classe serveur
  - Enregistrer au moins une instance dans le serveur de noms

Le compilateur IDL : **rmic**

- Le compilateur **rmic** génère les amorces (**stub**/ skeleton)
  - assurent le rôle d'adaptateurs pour le transport des appels distants.
  - réalisent les appels sur la couche réseau.
  - réalisent l'assemblage/désassemblage des paramètres (sérialisation ou *marshalling*/*unmarshalling*).
- Une référence d'objets distant correspond à une référence d'amorce (**stub**).

Stub/Skeleton

- **Stub** (talon client)
  - Représentant local de l'objet distant.
  - Initie une connexion avec la JVM distante
  - Assemble les paramètres pour leur transfert à la JVM distante.
  - Attend les résultats de l'invocation distante.
  - Désassemble la valeur ou l'exception renvoyée.
  - Renvoie la valeur au client (l'appelant).

- **Skeleton** (talon serveur)
  - Désassemble les paramètres pour la méthode distante.
  - Fait appel à la méthode demandée.
  - Assemble le résultat (valeur renvoyée ou exception) à destination de l'appelant.

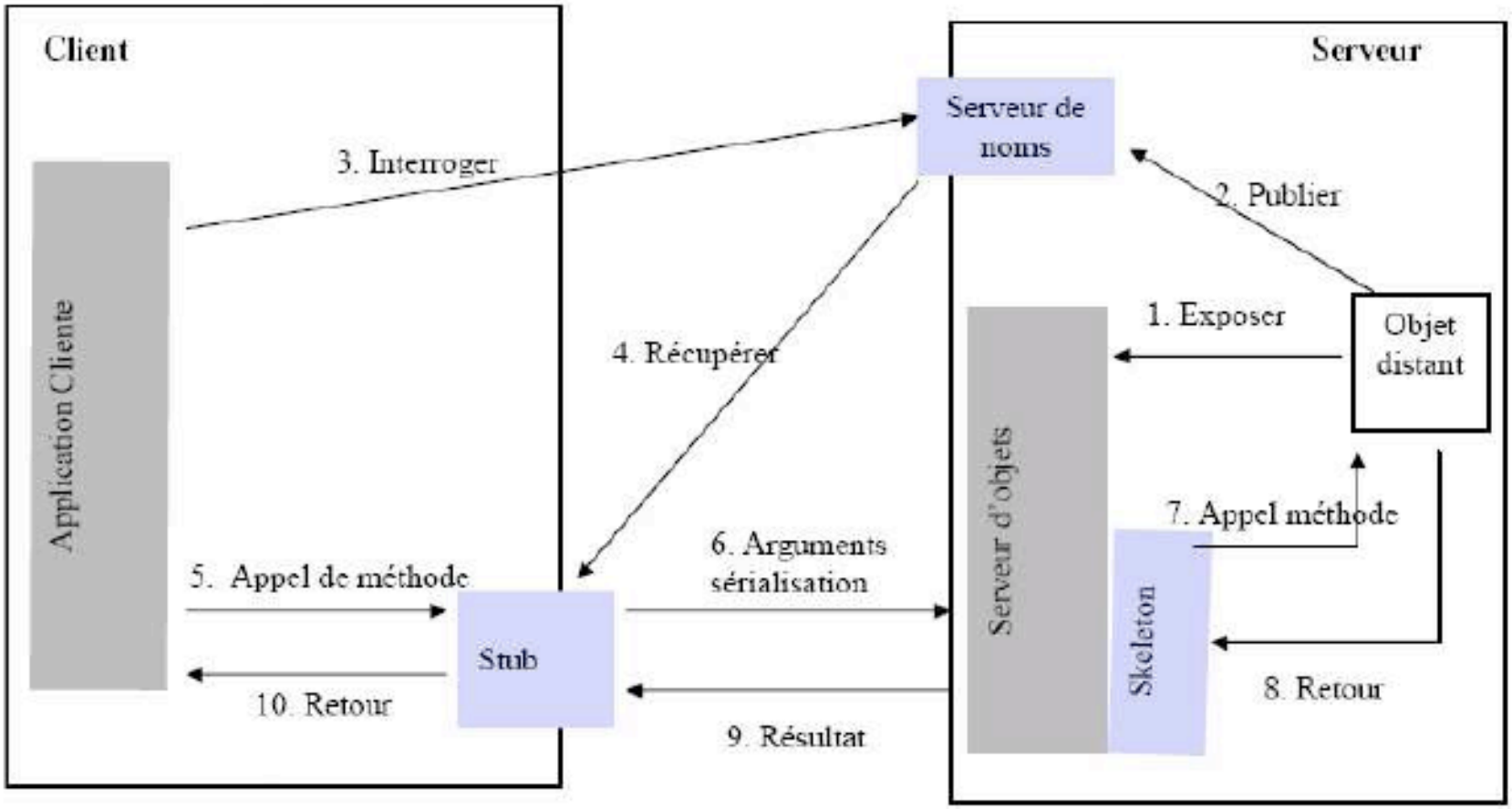
Serveur de noms : **rmiregistry**

- Permet d'obtenir une référence d'objet distant à partir de la référence locale au stub.
- S'exécute sur chaque machine hébergeant des objets distants.
- Utilise par défaut le port **1099**.
- Accepte comme opérations l'enregistrement, l'effacement et la consultation (méthodes de la classe java.rmi.Naming)
  - void Naming.bind(String nom, Remote obj) : associe l'objet au nom spécifié;
  - void Naming.rebind(String nom, Remote obj) : réassocie le nom au nouvel objet;
  - void Naming.unbind(String nom) : supprime l'enregistrement correspondant;
  - Remote Naming.lookup(String nom) : renvoie la référence (stub) de l'objet enregistré sous le nom donné;
  - String[] Naming.list(String nom) : renvoie la liste des noms enregistrés.

Protocole RMI-IIOP

- Elle réalise les connexions réseaux basées sur les flux entre les JVM.
- Elle emploie un protocole de communication propriétaire (**JRMP** : Java Remote Method Protocol) basé sur TCP/IP.

Étapes d'un appel de méthode distante



Développer une application avec RMI

1. Définir une interface distante (**App.java**).
2. Créer une classe implémentant cette interface (**AppImpl.java**).
3. Compiler cette classe (**javac AppImpl.java**).
4. Créer une application serveur (**AppServer.java**).
5. Compiler l'application serveur.
6. Créer les classes stub et skeleton à l'aide de **rmic:**
   AppImpl_Stub.java et **AppImpl_Skel.java**
7. Démarrage du serveur de noms (registre) avec **rmiregistry**.
8. Lancer le serveur **AppServer** pour la création d'objets et leur enregistrement dans rmiregistry.
9. Créer une classe cliente qui appelle des méthodes distantes de l'objet distant (**AppClient.java**).
10. Compiler cette classe et la lancer.

Exemple : Inversion d'une chaîne de caractères

- Invocation distante de la méthode **reverseString()** d'un objet distant qui inverse une chaîne de caractères fournie par l'appelant.
- Classes nécessaires:
  - **ReverseInterface.java** : interface qui décrit l'objet distant
  - **Reverse.java** : qui implémente l'objet distant
  - **ReverseServer.java** : le serveur d'objets RMI
  - **ReverseClient.java** : le client qui utilise l'objet distant

Fichiers nécessaires

Côté Client

- l'interface :

  ReverseInterface.
- le client :

  ReverseClient.

Côté Serveur

- l'interface :

  ReverseInterface.
- l'objet :

  Reverse.
- le serveur d'objets :

  ReverseServer.

Interface

- Interface de l'objet
  - Permet de décrire les différentes méthodes applicables à l'objet distant
  - Partagée par le client et le serveur.
  - Doit étendre l'interface **Remote** définie dans **java.rmi**.
  - Toutes ses méthodes doivent déclarer l'exception **RemoteException**.
- RemoteException est déclenchée :
  - si connexion refusée à l'hôte distant
  - ou bien si l'objet n'existe plus,
  - ou encore s'il y a un problème lors de l'assemblage ou le désassemblage.

Interface


```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface ReverseInterface extends Remote
{
    String reverseString(String chaine) throws RemoteException;
}
```

Implémentation de l'objet distant

- Une classe qui implémente l'interface définie
- Permettre de rendre un objet accessible à distance par des clients.
- Doit permettre l'**exposition** de l'objet distant:
  - La solution (la plus simple) consiste à étendre la classe **UnicastRemoteObject** de **java.rmi.server**.

Implémentation de l'objet distant


```
import java.rmi.*;
import java.rmi.server.*;
public class Reverse extends UnicastRemoteObject implements ReverseInterface {
      public Reverse() throws RemoteException
       {
          super();
       }
       public String reverseString (String ChaineOrigine) throws RemoteException
       {
           int longueur=ChaineOrigine.length();
           StringBuffer temp=new StringBuffer(longueur);
           for (int i=longueur; i>0; i--)
          {
                  temp.append(ChaineOrigine.substring(i-1, i));
           }
           return temp.toString();
      }
}
```

Serveur d'objets

Les différentes opérations réalisées par le serveur:

- Définir un gestionnaire de sécurité
  - Objet de la classe RMISecurityManager qui autorisera le chargement depuis une autre application de classes sérialisables:
  - System.setSecurityManager (new RMISecurityManager());

- Créer les objets distants par instanciation de classes Remote:
  - MonObjetDistant Objet = new MonObjetDistant;

Serveur d'objets

- Exposer les objets distants
  - Rendre accessibles dans le serveur
  - Si la classe de l'objet a été définie comme une extension de UnicastRemoteObject, cette exposition est automatique.
  - Sinon, l'objet doit être exposé explicitement par appel de la méthode exportObject de la classe UnicastRemoteObject:

```
UnicastRemoteObject.exportObject(unObjet);
```

- Faire connaitre l'existence des différents objets au serveur de noms (**rmiregistry**) de la machine sur laquelle ce serveur s'exécutera

```
Naming.rebind("nomUnObjet", unObjet);
```

Serveur d'objets


```
import java.rmi.*;
public class ReverseServer {
    public static void main(String[] args) {
    System.setSecurityManager(new RMISecurityManager());
     try {   System.out.println( "Serveur : Construction de l'implémentation ");
            Reverse rev= new Reverse();
            System.out.println("Objet Reverse lié dans le RMIregistry");
            Naming.rebind("rmi://localhost:1099/MyReverse", rev);
            System.out.println("Attente des invocations des clients ...");
           }
           catch (Exception e) {
            System.out.println("Erreur de liaison de l'objet Reverse");
            System.out.println(e.toString()) }
}}
```

Le client (1/4)

- Le client doit s'adresser au serveur de noms auprès duquel l'objet auquel il souhaite accéder a été enregistré afin de récupérer le stub de l'objet.
- Cette opération est réalisée en invoquant la méthode **lookup** de la classe **Naming**.
- Pour cela, il doit nommer complètement l'objet concerné au travers d'une URL de la forme générale suivante :

  rmi://machine:port/nom

- Dans une telle URL,
  - si le nom de machine est omis, la machine locale est considérée;
  - si le numéro de port est omis, le numéro par défaut (1099) est considéré.

Le client (2/4)

- Comme lors de l'enregistrement d'un objet par un serveur, la méthode **lookup** est susceptible de provoquer une exception (**RemoteException**). Il est donc nécessaire d'en prévoir l'occurrence.
- La récupération d'un stub par un client a ainsi la forme générale suivante:

```
MonInterface stub;
try
{
        stub= (MonInterface) Naming.lookup (adresse_de_l'objet);
}
catch (Exception e){-------- }
```

Le client (3/4)

- Une fois le stub est récupéré, un client peut invoquer sur ce stub des méthodes de l'interface comme il ferait sur un objet local:

  objet=stub.méthode(paramètres);

Le client (4/4)


```
import java.rmi.*;
public class ReverseClient {
    public static void main (String [] args) {
        System.setSecurityManager(new RMISecurityManager());
        try{
            ReverseInterface rev = (ReverseInterface) Naming.lookup
            ("rmi://localhost:1099/MyReverse");
            String result = rev.reverseString (args [0]);
            System.out.println ("L'inverse de "+args[0]+" est "+result);
        }
        catch (Exception e) {
            System.out.println ("Erreur d'accès à l'objet distant.");
            System.out.println (e.toString());
        }
    }
}
```

Gestionnaire de sécurité

- Installer un gestionnaire de sécurité
  - Permet de protéger la machine virtuelle java de tout utilisateur malintentionné qui essaye de faire tourner du code malicieux sur la machine.
- Utiliser le gestionnaire de sécurité fournit par RMI: RMISecurityManager

```
System.setSecurityManager(new RMISecurityManager());
```

Gestionnaire de sécurité

- La politique de sécurité spécifie les actions autorisées, en particulier sur les sockets
- Les opérations possibles sur les sockets:
  - **connect** est utilisé pour établir une connexion avec un serveur.
  - **listen** est utilisé par un serveur pour indiquer qu'il est disposé à accepter des connexions.
  - Après un **listen**, un serveur peut attendre une demande de connexion d'un client par **accept**.
- Exemple de contenu du fichier **java.policy**:

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

Gestionnaire de sécurité

- Exécuter le programme java utilisant un gestionnaire de sécurité avec l'option :

java -Djava.security.policy=java.policy NomClasse

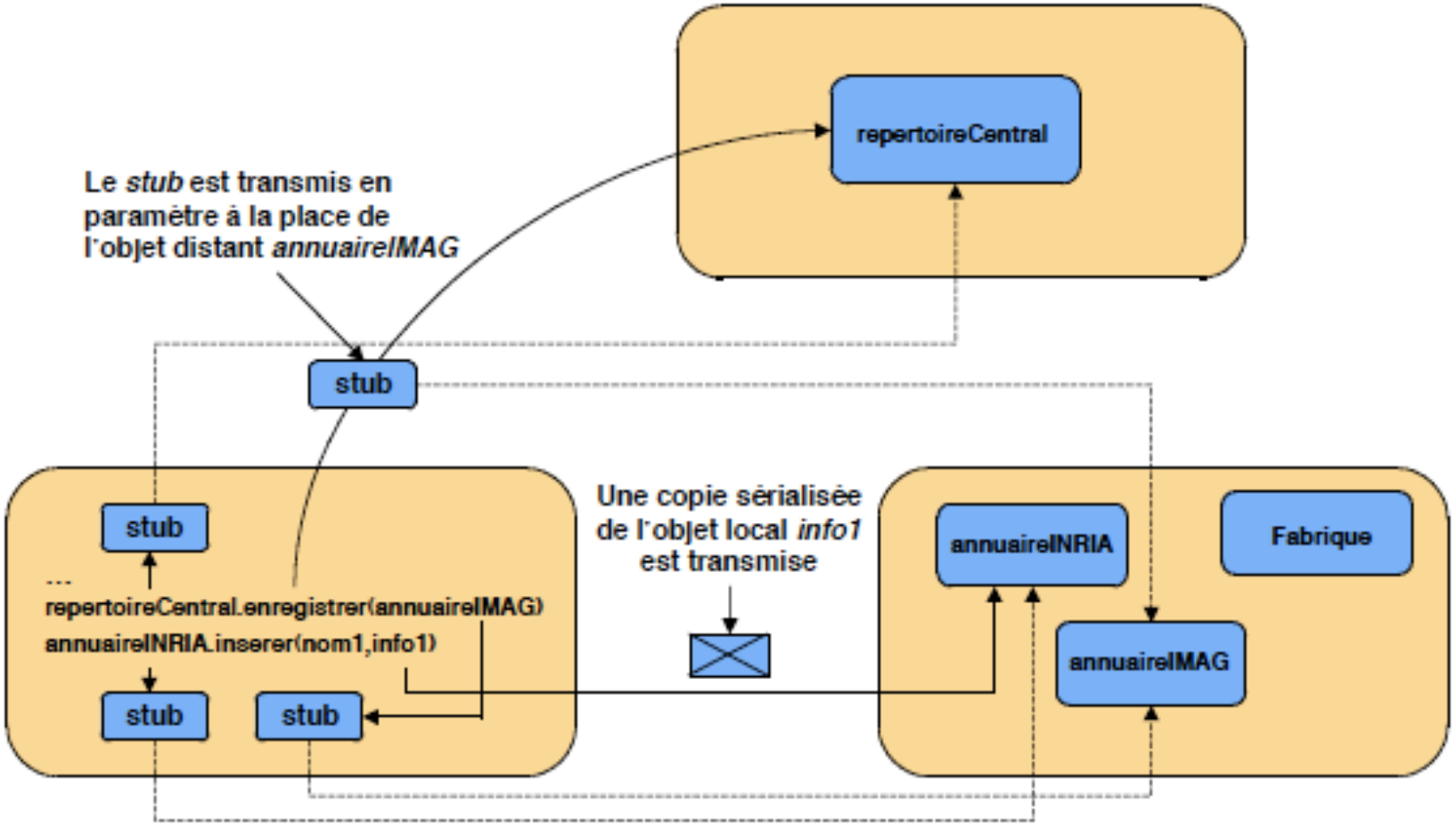
Où java.policy et le fichier des permissions

Passage d'objets en paramètres

Deux cas possibles

- Passage en paramètre d'un **objet local** (sur la JVM de l'objet appelant)
  - **Passage par valeur** : on transmet une copie de l'objet (plus précisément : une copie de l'ensemble ses variables d'état)
  - Pour cela l'objet doit être sérialisable (i.e. implémenter l'interface java.io.Serializable)
  - Exemple de l'annuaire : le client transmet un objet de la classe locale Info
- Passage en paramètre d'un **objet non-local** (hors de la JVM de l'objet appelant, par ex. sur un site distant)
  - **Passage par référence** : on transmet une référence sur l'objet (plus précisément : un stub de l'objet). Le destinataire utilisera ce stub pour appeler les méthodes de l'objet.
  - Exemple de l'annuaire : le client passe un objet de type Annuaire (localisé sur le serveur distant)

Passage d'objets en paramètres



Fabrique d'objets (Factory)

- **Motivation**
  - Permettre au client de construire des instances multiples d'une classe **C** sur le site serveur
  - Le ***new*** n'est pas utilisable tel quel (car il ne gère que la mémoire locale, celle du client)
  - Solution : appel d'un objet **FabriqueC**, qui crée localement (sur le serveur) les instances de **C** (en utilisant ***new*** **C**)
- Exemple

```
/*Interface de l'objet distant*/
public interface Annuaire extends Remote{
        public boolean inserer(String nom, int id) throws RemoteException;
        public boolean supprimer(String nom, int id) throws RemoteException;
        public Info rechercher(String nom) throws RemoteException;
}
```

```
/*Interface de fabrique d'objets*/
public interface FabAnnuaire extends Remote{
        public Annuaire newAnnuaire(String titre) throws RemoteException ;
}
```

Fabrique d'objets (Factory)


```
/*L'implementation de l'objet distant
public class AnnuaireImpl implements Annuaire extends UnicastRemoteObject{
        private String titre;
        public Annuaire(String titre) {this.titre=titre};
        public boolean inserer(String nom, Info info) throws RemoteException{…};
        public boolean supprimer(String nom) throws RemoteException{…};
        public Info rechercher(String nom) throws RemoteException{…};
}
```

```
/*L'implémentation de la fabrique d'objets
public class FabAnnuaireImpl implements FabAnnuaire extends UnicastRemoteObject{
        public FabAnnuaireImpl{};
        public Annuaire newAnnuaire(String titre) throws RemoteException {
                return new AnnuaireImpl(titre)};
}
```

Fabrique d'objets (Factory)


```
/*Serveur d'objets*/
import java.rmi.*;
public class Server {
        public static void main (String [ ] argv) {
                /* lancer SecurityManager */
                System.setSecurityManager (new RMISecurityManager ()) ;
                try {
                        Naming.rebind ("Fabrique",new (FabAnnuaireImpl)) ;
                        System.out.println ("Serveur prêt.") ;
                }
                catch (Exception e) {
                        System.out.println("Erreur serveur : " + e) ;
                }
        }
}
```

Fabrique d'objets (Factory)


```
/*Code Client*/
import java.rmi.*;
public class HelloClient {
        public static void main (String [ ] argv) {
        /* lancer SecurityManager */
        System.setSecurityManager (new RMISecurityManager ()) ;
        try {
                /* trouver une référence vers la fabrique */
        FabAbannuaire fabrique =  (FabAbannuaire ) Naming.lookup ("rmi://goedel.imag.fr/Fabrique") ;
        /* créer un annuaire */
        annuaireIMAG = fabrique.newAnnuaire("IMAG");
        /* créer un autre annuaire */
        annuaireINRIA= fabrique.newAnnuaire("INRIA");
        /* utiliser les annuaires */
        annuaireIMAG.inserer(…, …);
        annuaireINRIA.inserer(…, …);
        ….
        } catch (Exception e) {
                System.out.println
        ("Erreur client : " + e) ;
}}}
```

Conclusion

- **Extension du RPC aux objets**
  - Permet l'accès à des objets distants
  - Pas de langage séparé de description d'interfaces (IDL fourni par Java)
- **Limitations**
  - Environnement restreint à un langage unique (Java)
  - Mais passerelles possibles, en particulier RMI/IIOP
- **Services réduits au minimum**
  - Service élémentaire de noms (sans attributs)
  - Pas de services additionnels
    - Duplication d'objets
    - Transactions

Ce qui reste à voir

- **Parallélisme sur le serveur**
  - Un serveur RMI peut servir plusieurs clients. Dans ce cas, un thread séparé est créé pour chaque client sur le serveur. C'est au développeur du programme serveur d'assurer leur bonne synchronisation (exemple : méthodes synchronized) pour garantir la cohérence des données)
- **Activation automatique de serveurs**
  - Un veilleur (demon) spécialisé, rmid, peut assurer le lancement de plusieurs classes serveur sur un site distant (cette opération est invisible au client)
- **Téléchargement de code**
  - Un chargeur spécial (ClassLoader) peut être développé pour assurer le téléchargement des classes nécessaires