# UE HAI704I
# Architectures logicielles distribuées

Abdelhak-Djamel Seriai

seriai@lirmm.fr

# RMI : Java Remote Method Invocation

# Java RMI (Remote Method Invocation)

- Le système Java **R**emote **M**ethod **I**nvocation (RMI) permet à un objet s'exécutant dans une machine virtuelle Java d'invoquer des méthodes sur un objet s'exécutant dans une autre machine virtuelle Java.

- RMI permet la communication à distance entre des programmes écrits en Java.

- Invocation de méthodes sur des objets distribués en cachant au programmeur les détails de connexion et de transport.
  - Interagir avec un objet distant comme s'il était local, i.e. dans la même JVM

- Construction d'applications réparties avec Java : Appel de méthode au lieu d'appel de procédure.

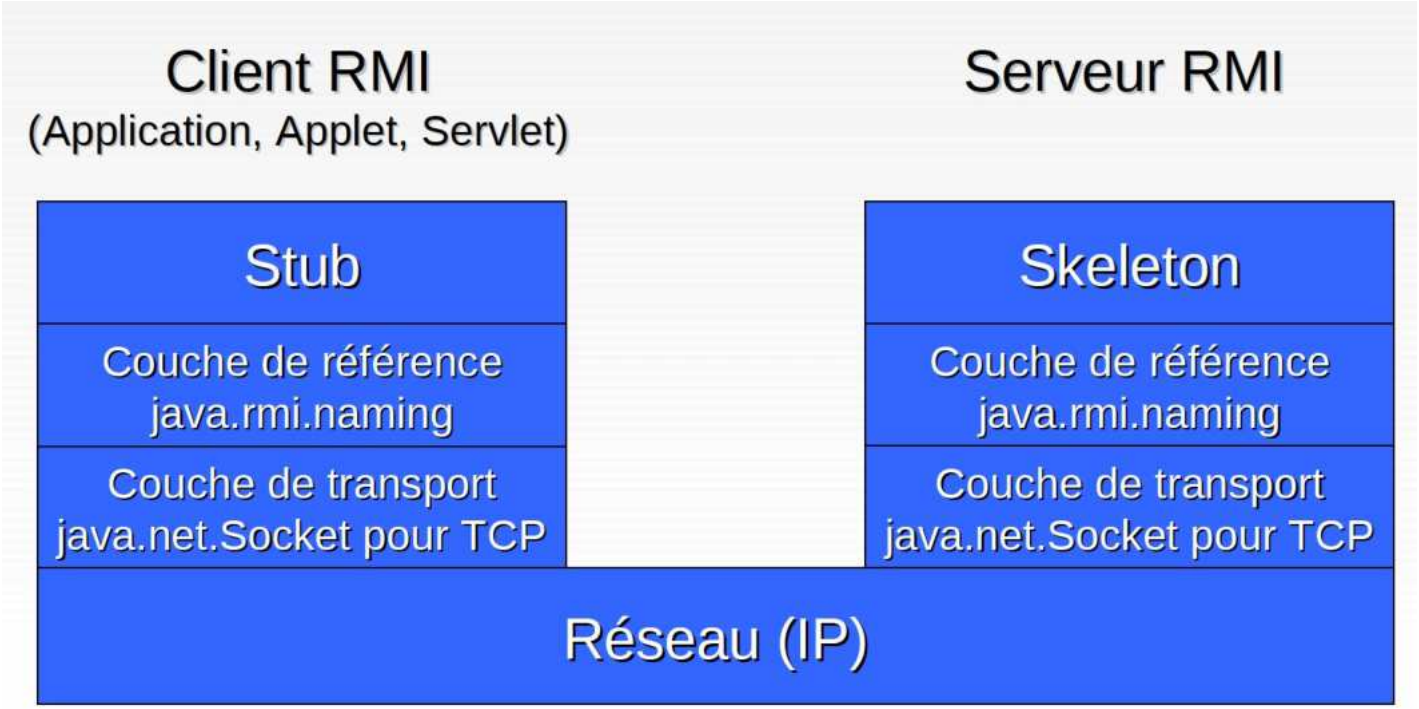
# Caractéristiques

- Le programmeur fournit :
  - Une (ou plusieurs) description(s) d'interface
    - Ici pas d'IDL séparé : Java sert d'IDL
  - Objets réalisant des interfaces fournies (ensemble d’opérations fournies)
  - Le programme de création des objets distants sur le Serveur (programme serveur)
  - Le programme utilisant les objets distant sur le Client (programme client)

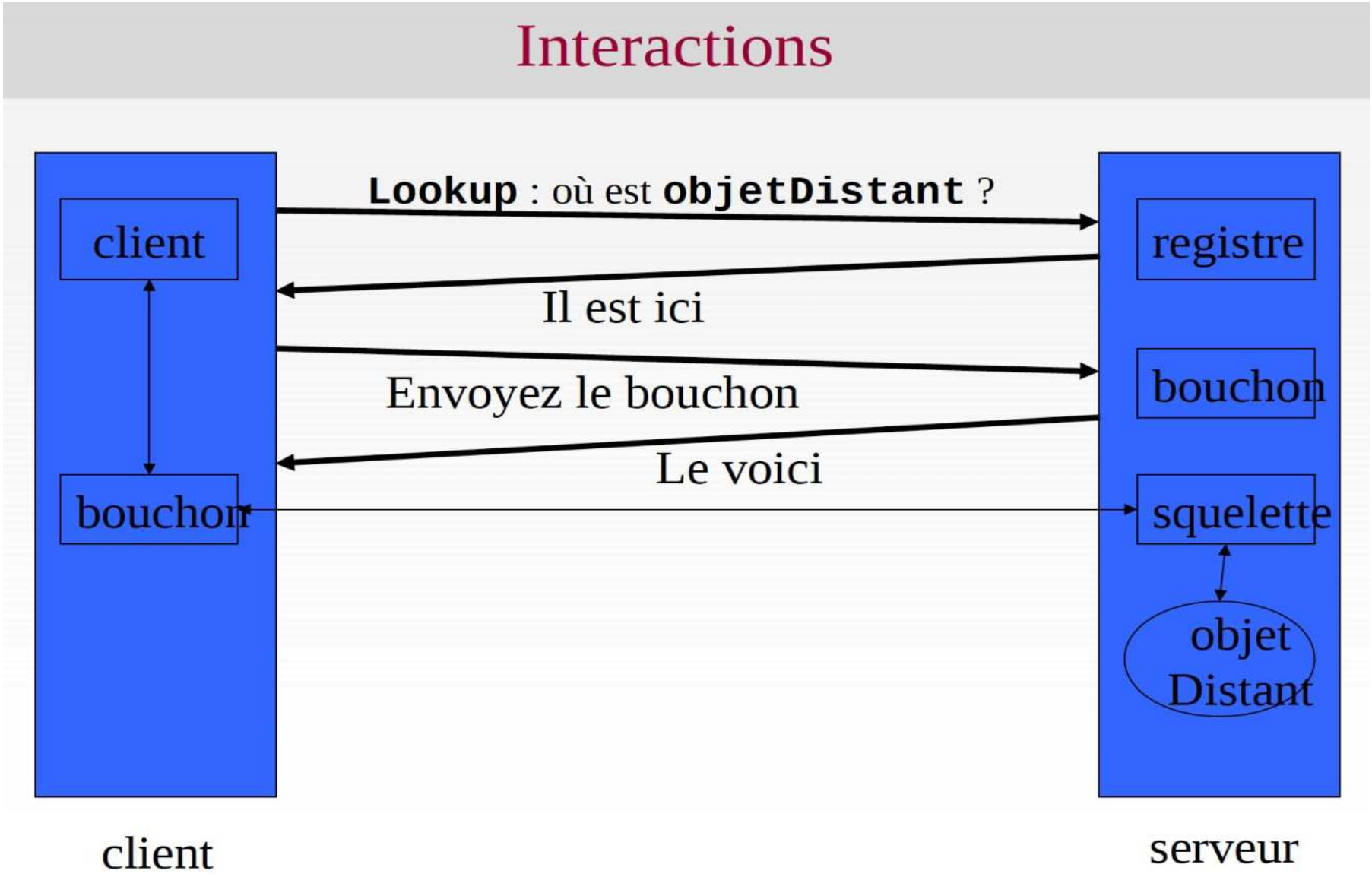
- L'environnement Java RMI fournit :
  - Un générateur de talons (rmic)
    - Utilisation de stubs (talons/bouchons) et de squelettes pour masquer le codage et dés-encodage des données.
  - Un service de noms (Object Registry)
    - Transmission d’objets **par copie** ou **par référence**

# Architecture d'une système distribué en Java RMI

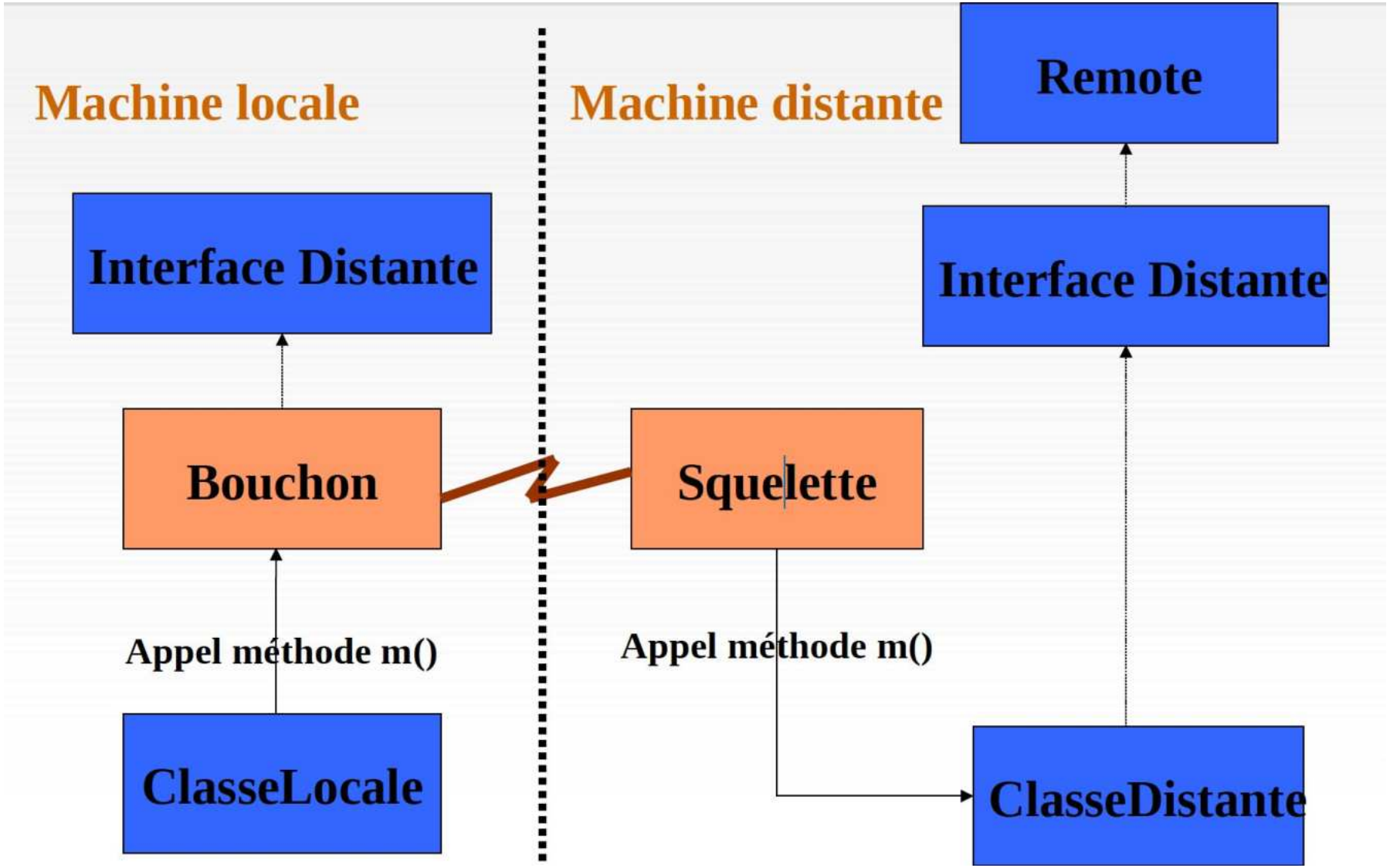
- **Le bouchon (stub/proxy)**
  - Implémente une interface identique à celle de l'objet, mais ne contient pas de logique métier.
  - Implémente les opérations réseau à effectuer pour transmettre la requête à l'objet.
- **Le squelette (skeleton)**
  - Analyse les messages reçus en provenance d'un bouchon.
  - Invoque la méthode métier correspondante sur l'objet.
- **Le registre**
  - Annuaire des objets distribués.



# Le schéma général de la communication distante



# Le schéma général de la communication distante



# Côté client : le Stub

- **Stub pour un objet distant (remote object)**
  - Joue le rôle d'un **proxy** (représentant local) pour l'objet distant.
  - Implémente les mêmes interfaces distantes que celles de l'objet distant.
  - Masque à l'utilisateur la sérialisation des paramètres (ou un passage de stub) et la communication réseau.
- **Quand une méthode est appelée sur un stub, celui ci :**
  - Initie une connexion avec la JVM distante.
  - Lui transmet les paramètres d'appel.
  - Attend le résultat.
  - Récupère la valeur ou l'exception de retour.
  - Envoie le résultat à l'appelant.

# Côté serveur : le squelette

- Responsable de transmettre l'appel à l'implantation du réel objet distant.
- Quand un squelette reçoit une invocation de méthode :
  - Il lit les paramètres.
  - Il invoque la méthode sur l'implémentation de l'objet distant.
  - Il transmet le résultat à l'appelant.

# Couche des références distantes et de transport

- **Couche des références distantes**
  - Permet l'association stub/objet distant.
  - Processus tiers : ***rmiregistry***.
    - **Serveur de liaison**, sur le **port 1099** par défaut.
    - **Expose un objet distant** serveur de liaisons (de noms)
    - Fait la **correspondance entre nom et instance de stub** enregistré par le serveur avec ***Naming.bind().***

- **Couche de transport**
  - Écoute les appels entrants.
  - Gestion des connexions avec les sites distants.
  - Possibilité d'utiliser différentes classes pour le transport.

# Créer une application distribuée en RMI

# Les packages

- ***Java.rmi*** : accès aux objets distants (OD)
- ***Java.rmi.server*** : création d'OD
- ***Java.rmi.registry*** : localisation et nommage d'OD
- ***Java.rmi.dgc*** : ramasse miettes d'OD
- ***Java.rmi.activation*** : activation d'OD

# Les 5 étapes

1. Spécifier et écrire l'interface de l'objet distant.
2. Ecrire l'implémentation de cette interface.
3. Générer les stubs et squelettes (versions <1.5).
4. Ecrire le serveur (instancie l'objet, exporte son stub, attend les requêtes via le squelette).
5. Ecrire le client (réclame l'objet distant, importe le stub, invoque une méthode sur l'objet distant via le stub).

# Etape 1 : Spécifier et écrire l'interface de l'objet distant

- Un objet distant est une instance d'une classe qui implémente une interface distante.

- Une interface distante
  - Étend l'interface ***java.rmi.Remote***
  - Déclare un ensemble de méthodes distantes.
  - Doit être publique.

- Chaque méthode distante doit déclarer ***java.rmi.RemoteException*** (ou une superclasse de *RemoteException*) dans sa clause *throws*, en plus de toute exception spécifique à l'application.

  - Pourquoi ?
    - Les invocations de méthodes distantes peuvent échouer de plusieurs manières supplémentaires par rapport aux invocations de méthodes locales (comme les problèmes de communication liés au réseau et les problèmes de serveur), et les méthodes distantes signaleront ces échecs en lançant une *java.rmi.RemoteException*.

# Etape 1 : Spécifier et écrire l'interface de l'objet distant

- Exemple :

```
package example.hello;
import java.rmi.Remote;
import java.rmi.RemoteException;

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

# Etape 2 : Ecrire l'implémentation de l'Interface Distante

A. Implémente une ou plusieurs interfaces distantes.

B. Peut hériter de ***UnicastRemoteObject***

C. Implémente toutes les méthodes distantes

D. Définit le constructeur d'objets distants

E. Peut définir des méthodes non spécifiées dans l'interface distante

—> mais ces méthodes ne peuvent être invoquées que dans la machine virtuelle exécutant le service et ne peuvent pas être appelées à distance.

```
package example.hello;

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

public class Server implements Hello {
    public Server() {}

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

# Etape 3 : Génération des stubs et squelettes

- Appel de l'outil ***rmic*** (versions <1.5)
  - génère la classe stub
  - génère la classe squelette à partir de l'implémentation (le .class)
- Génération dynamique (>=1.5)
  - Quand un OD est enregistré
  - Incompatible avec clients <1.5
- Squelettes pas requis dès Java 2
  - Code générique utilisé à la place
  - Pas générés par défaut depuis java 1.5

- A partir de la version J2SE 5.0, **les classes de stub pour les objets distants n'ont plus besoin d'être pré-générées à l'aide du compilateur de stub *rmic*,** sauf si l'objet distant doit prendre en charge les clients exécutés dans les machines virtuelles antérieures à la version 5.0.

# Etape 4 : Ecrire le serveur d'objets distants

- Définition de la classe "Serveur"
  - Une classe « *serveur* » est la classe qui :

    1. Définit une méthode qui crée une instance de l'objet distant.
    2. Exporte l'objet distant.
    3. Enregistre l'objet distant dans le registre d'objets distants RMI.
    4. Crée et installe un gestionnaire de sécurité.

- Remarque : La classe qui contient cette méthode principale peut être la classe d'implémentation de l'objet distant elle-même ou une autre classe.

# Etape 4 : Ecrire le serveur d'objets distants

## A) Créer un objet distant

- La méthode ***main*** de la classe serveur doit :
  - Créer l'objet distant qui fournit le service.

## B) Exporter l'objet distant

- La méthode ***main*** de la classe serveur doit :
  - Créer l'objet distant qui fournit le service.
  - Exporter l'objet distant vers l'environnement d'exécution Java RMI afin qu'il puisse recevoir les appels distants entrants.

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

- La méthode statique *UnicastRemoteObject.exportObject()* exporte l'objet distant qui :
  - Reçoit les appels distants de méthodes, entrants sur un port TCP anonyme
  - Renvoie le stub pour l'objet distant à transmettre aux clients.
- À la suite de l'appel *exportObject()*, le runtime peut commencer à écouter sur un nouveau socket serveur ou peut utiliser un socket serveur partagé pour accepter les appels distants entrants pour l'objet distant.
- Le stub renvoyé implémente le même ensemble d'interfaces distantes que la classe de l'objet distant et contient le nom d'hôte et le port sur lesquels l'objet distant peut être contacté.

# Etape 4 :Ecrire le serveur d'objets distants

## C) Enregistrez l'objet distant avec un registre Java RMI

- Un registre Java RMI est un service de noms simplifié qui permet aux clients d'obtenir une référence (un stub) à un objet distant.
  - Le code suivant sur le serveur :
    - obtient un registre sur l'hôte local et le port de registre par défaut (1099),
    - puis utilise le stub de registre pour lier le nom « Hello » au stub de l'objet distant dans ce registre

```
Registry registry = LocateRegistry.getRegistry();
registry.bind("Hello", stub);
```

→ La méthode statique *LocateRegistry.getRegistry*

- Ne prend aucun argument
- Renvoie un stub qui implémente l'interface distante java.rmi.registry.Registry.
- Envoie des appels au registre sur l'hôte local du serveur sur le port de registre par défaut (1099)

→ La méthode de liaison est ensuite invoquée sur le stub du registre afin de lier le stub de l'objet distant au nom "Hello" dans le registre.

- Remarque : L'appel à *LocateRegistry.getRegistry* renvoie simplement un stub approprié pour un registre.
  - L'appel ne vérifie pas si un registre est réellement en cours d'exécution.
  - Si aucun registre n'est exécuté sur le port TCP 1099 de l'hôte local lorsque la méthode de liaison est invoquée, le serveur échouera avec une *RemoteException*.

# Etape 4 :Ecrire le serveur d'objets distants

- Exemple :

```
package example.hello;

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

public class Server implements Hello {

    public Server() {}

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

```
public static void main(String args[]) {
        try {
            Server obj = new Server();
            Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);

// Bind the remote object's stub in the registry
            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();
        }
    }
}
```

# Etape 5 : Implémentation d'un client

A) Demande un stub d'objet distant enregistré sur ***rmiregistry***
→ ***java.rmi.lookup***.

→ Si la classe du stub n'est pas connue

→ récupération auprès de ***java.rmi.server.codebase***

B) Invoque des méthodes sur le stub.

# Etape 5 : Implémentation d'un client

```
package example.hello;
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);


                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(); }
        } }
```

Le programme client

→ obtient un stub pour le registre sur l'hôte du serveur

→ recherche le stub de l'objet distant par nom dans le registre

→ appelle la méthode *sayHello* sur l'objet distant à l'aide du stub.

# Etape 5 : Implémentation d'un client

- Ce client obtient d'abord le stub du registre en appelant la méthode statique *LocateRegistry.getRegistry* avec le nom d'hôte  spécifié sur la ligne de commande.
  - Si aucun nom d'hôte n'est spécifié, *null* est utilisé comme nom d'hôte indiquant que l'adresse de l'hôte local doit être utilisée.
- Le client appelle la méthode distante *lookup* sur le stub de registre pour obtenir le stub de l'objet distant à partir du registre du serveur.
- Le client invoque la méthode *sayHello* sur le stub de l'objet distant, ce qui provoque les actions suivantes :
  - L'environnement d'exécution côté client ouvre une connexion au serveur à l'aide des informations d'hôte et de port dans le stub de l'objet distant, puis sérialise les données d'appel.
  - L'environnement d'exécution côté serveur accepte l'appel entrant, distribue l'appel à l'objet distant et sérialise le résultat (la chaîne de réponse « Hello, world ! ») au client.
  - Le runtime côté client reçoit, désérialise et renvoie le résultat à l'appelant.Le message de réponse renvoyé par l'appel distant sur l'objet distant est ensuite imprimé dans *System.out*.

# Exécution

# 1 : Compiler les fichiers sources

- Exemple de commande de compilation :

```
javac -d destDir Hello.java Server.java Client.java
```

où *destDir* est le répertoire de destination dans lequel placer les fichiers de classe.

- Remarque :
  - Si le serveur doit prendre en charge les clients s'exécutant sur des machines virtuelles antérieures à la version 5.0, une classe de stub pour la classe d'implémentation d'objet distant doit être pré-générée à l'aide du compilateur *rmic,* et cette classe de stub doit être mise à disposition des clients à télécharger.

# 2 : Démarrer le registre Java RMI

- Pour démarrer le registre, exécutez la commande ***rmiregistry*** sur l'hôte du serveur.
  - Cette commande ne produit aucune sortie (en cas de succès) et est généralement exécutée en arrière-plan.
  - Exemple,
    - Sur Linux : ***rmiregistry &***
    - Sur Windows : ***Start rmiregistry***
- Par défaut, le registre s'exécute sur le port TCP 1099.
- Pour démarrer un registre sur un autre port, il faut spécifier le numéro de port à partir de la ligne de commande.
  - Exemple, pour démarrer le registre sur le port 2021 sur Windows : ***Start rmiregistry 2021***
- Si le *registry* est exécuté sur un port autre que 1099, il est nécessaire de spécifier le numéro de port dans les appels à *LocateRegistry.getRegistry* dans les classes *Server* et Client.
  - Exemple : si le registre s'exécute sur le port 2021 dans cet exemple, l'appel à *getRegistry* sur le serveur serait :

```
Registry registry =
LocateRegistry.getRegistry(2021);
```

# 3 : Exécution côté serveur

- Pour démarrer le serveur, il faut exécuter la classe *Server* :
  - Sur Linux : ***java -classpath classDir -Djava.rmi.server.codebase=file:classDir/ example.hello.Server &***
  - Sur Windows : ***start java -classpath classDir -Djava.rmi.server.codebase=file:classDir/ example.hello.Server***
    → "classDir" est le répertoire racine de l'arborescence des fichiers de classe.
    → La définition de la propriété système *java.rmi.server.codebase* garantit que le registre peut charger la définition de l'interface distante ;
  - L'exécution du serveur donnerait :*Server ready*
  - Le serveur reste en cours d'exécution jusqu'à ce que le processus soit terminé par l'utilisateur (généralement en tuant le processus).
  - Il faut prévoir une procédure d'arrêt avec arrêt des objets distants :
    - ***public static void Naming.unbind(String name)***
    - ***public static boolean UnicastRemoteObject.unexportObject(Remote,boolean force)***

# 3 : Exécution côté serveur

- Le stub doit être dans le ***CLASSPATH*** ou chargeable via FS ou HTTP
- Un fichier ***.policy*** autorise l'usage du ***accept*** et du ***connect*** sur les sockets java

  → ***Djava.security.policy=./hello.policy helloServer hostreg:1099***

- Possibilité d'utiliser une ligne comme : ***System.setProperty( "java.security.policy", "hello.policy")***;
  - Exemple de fichier (ici : ./hello.policy)

```
grant{
    permission java.net.SocketPermission "*:1024-65535","connect,accept";
    permission java.net.SocketPermission ":80","connect";
    // permission java.security.AllPermission;
    }
```

# 4 : Exécuter le client

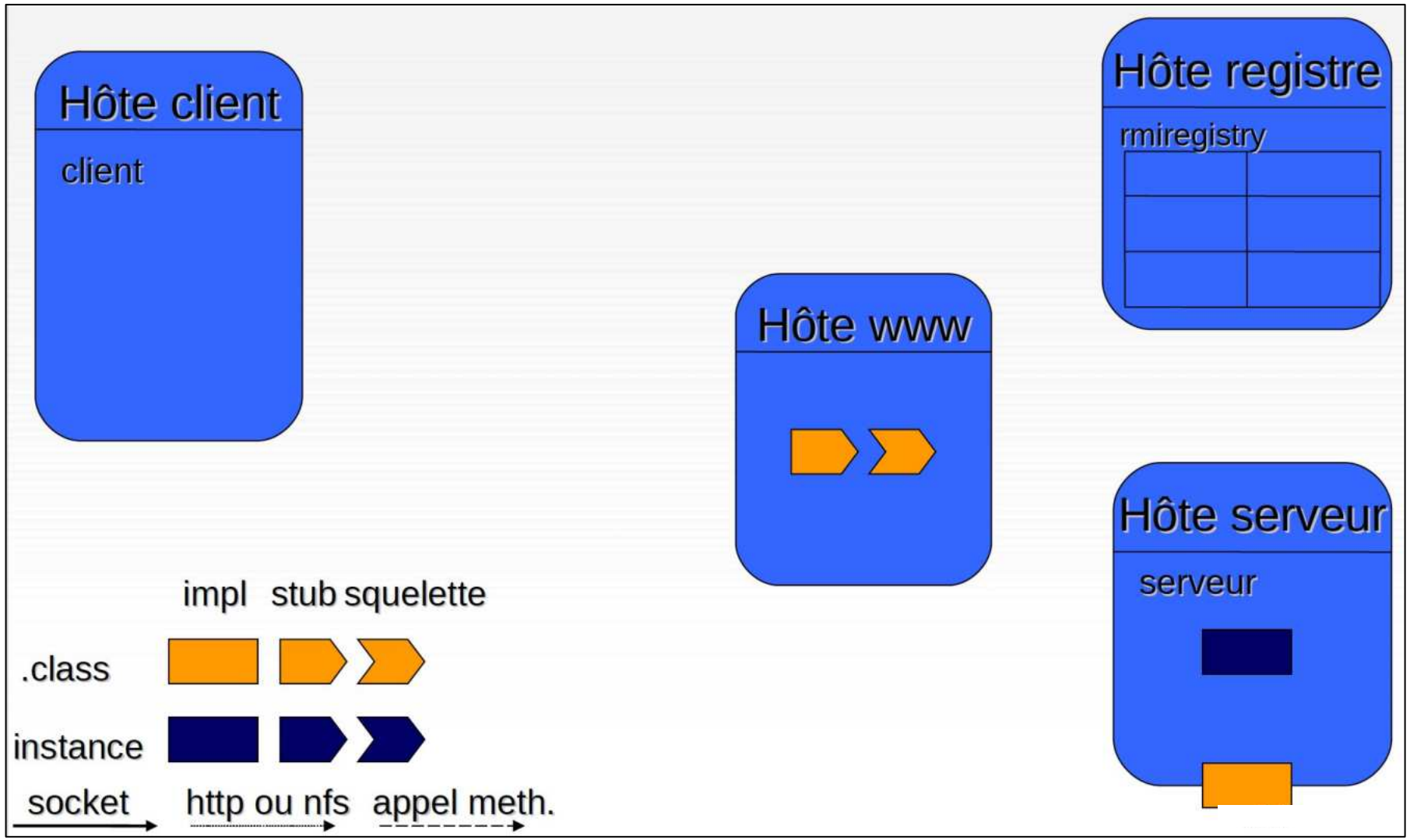
- Une fois le serveur prêt, le client peut être exécuté comme suit :

```
java -classpath classDir example.hello.Client
```

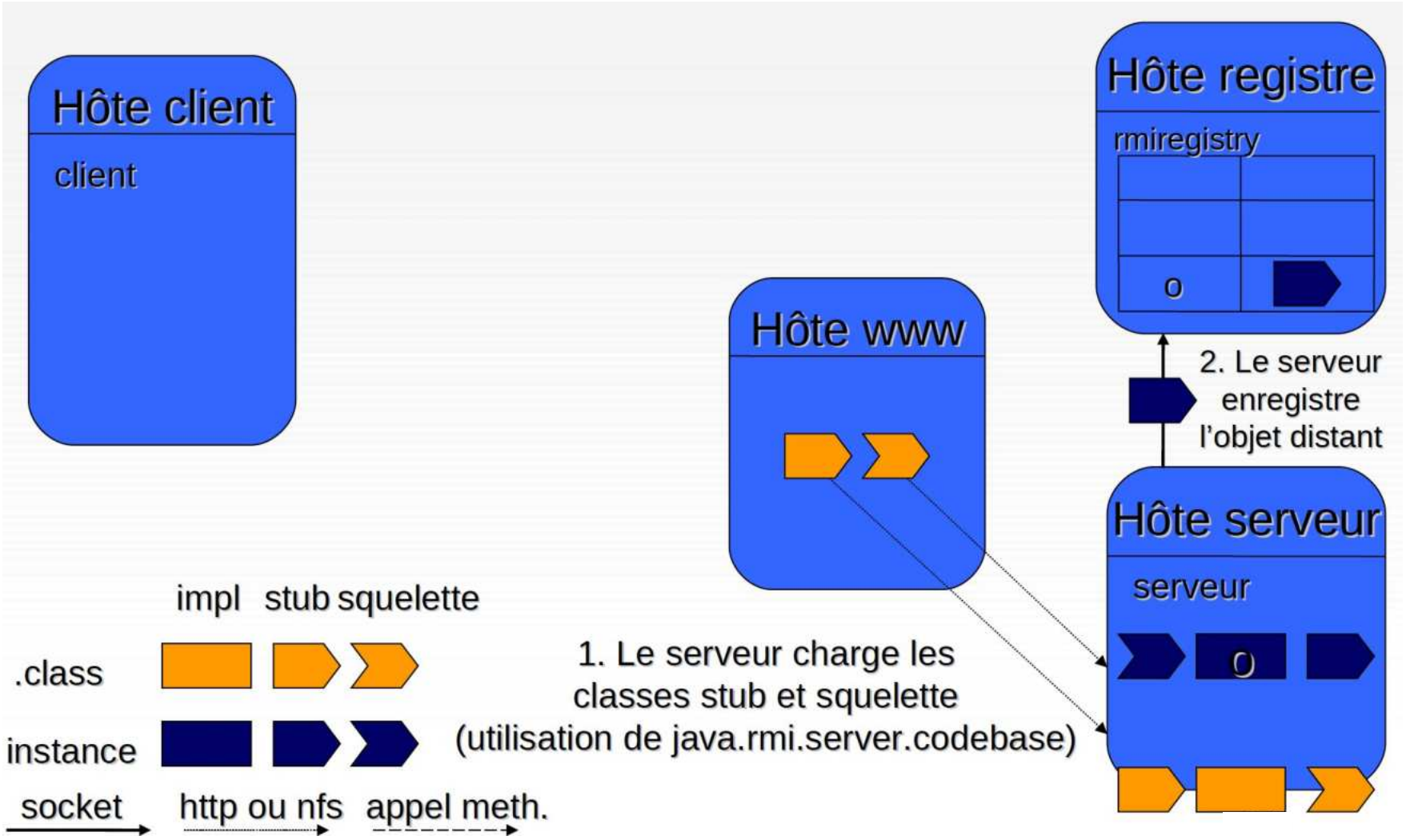
  - *classDir* est le répertoire racine de l'arborescence des fichiers de classe.

# Exécution : illustration

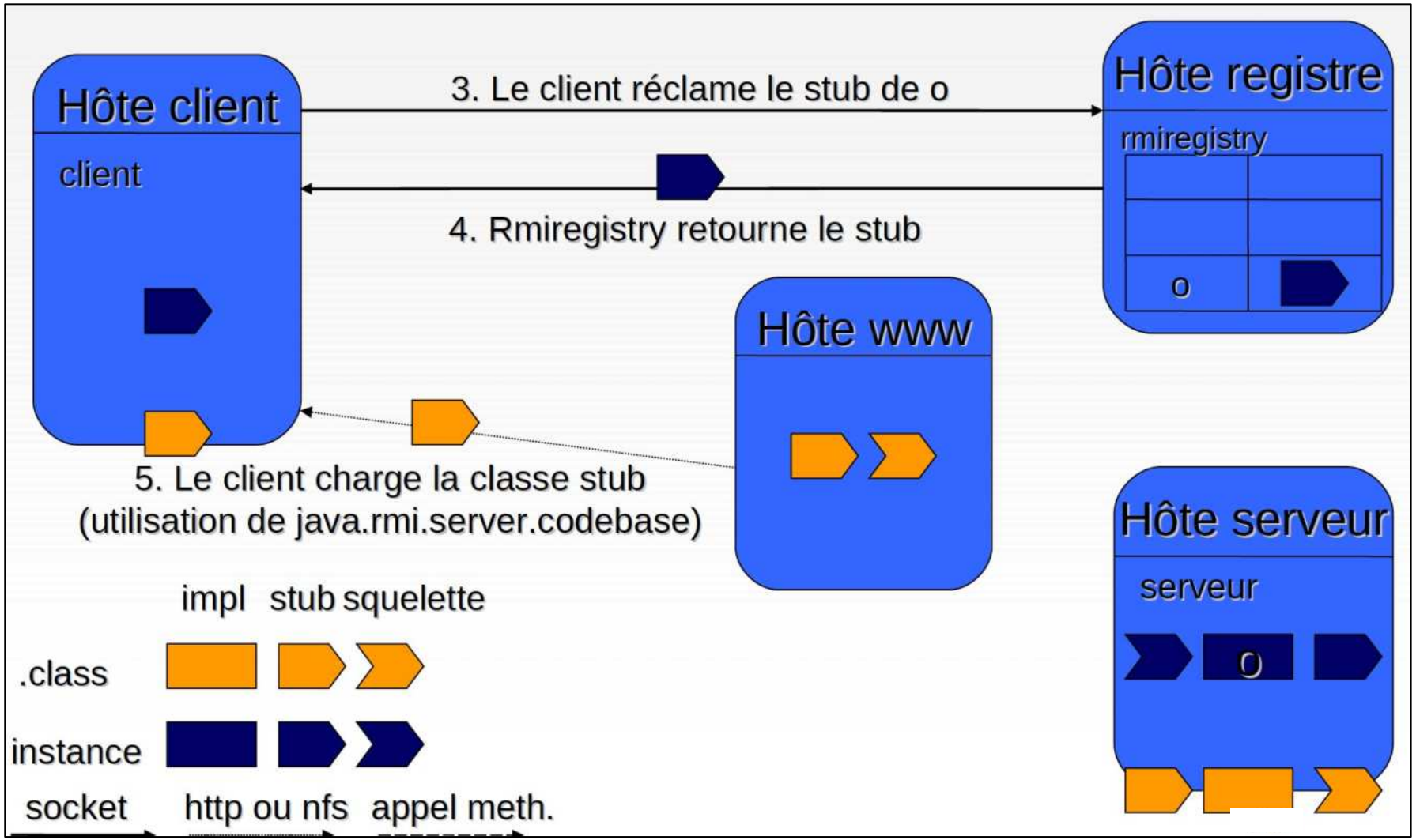
# Exécution : illustration



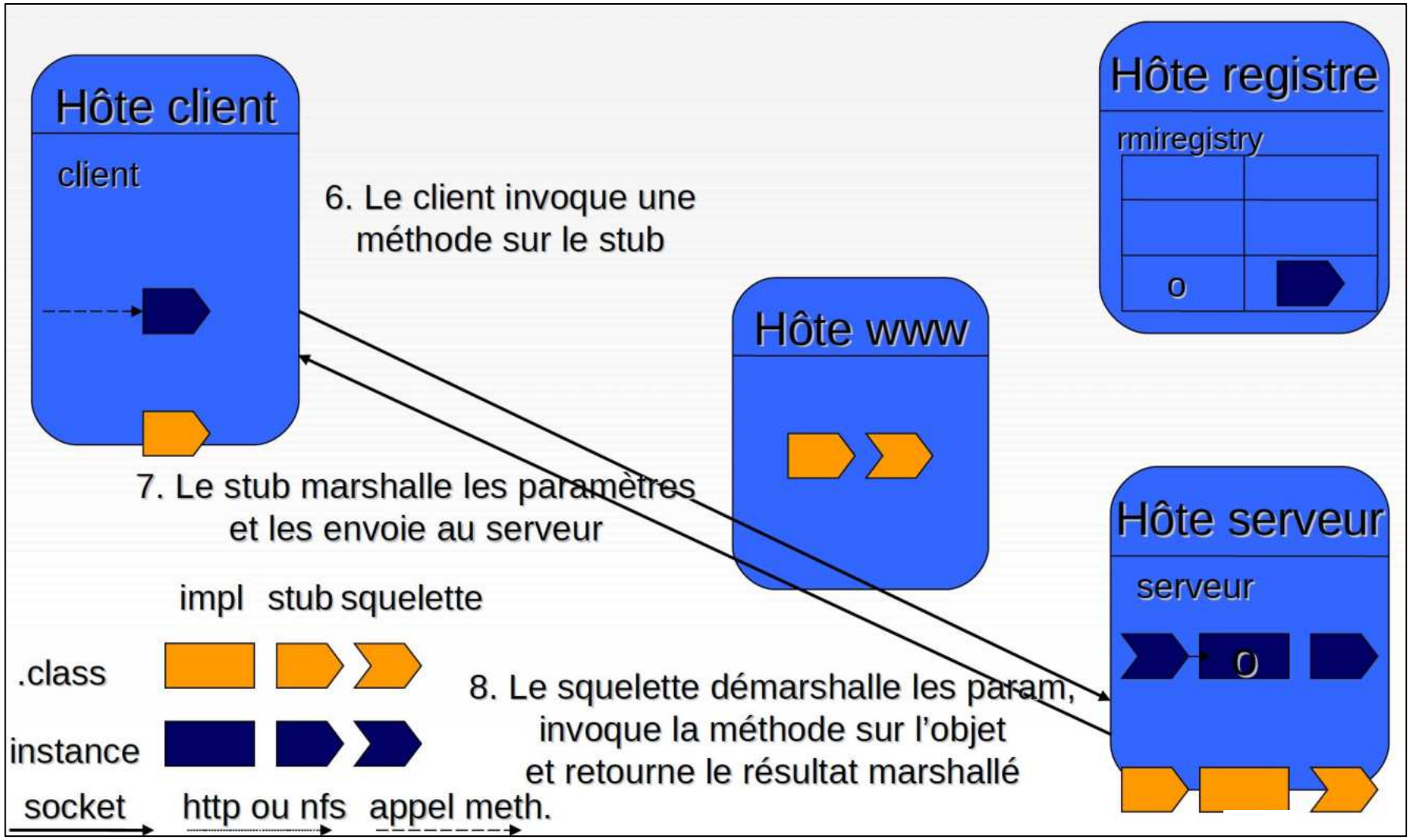
# Enregistrement de l'objet



# Récupération du stub



# Invocation d'une méthode



# Le passage de paramètres

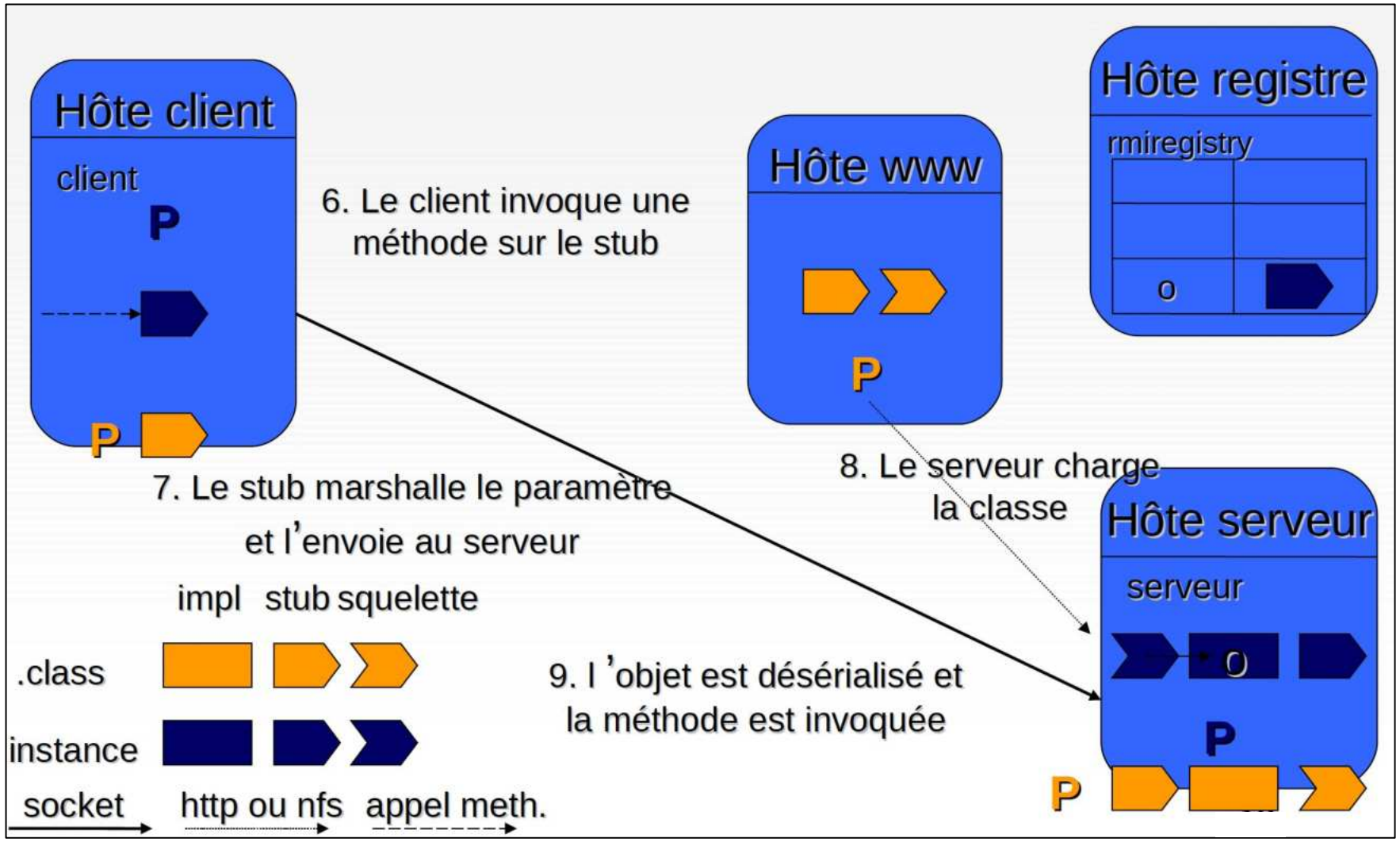
# Le passage de paramètres

- Les paramètres des méthodes invoquées sur un objet distant sont soit :
  - Une valeur de type primitif
    - Passage par valeur
  - Un objet d'une classe sérialisable
    - L'objet est sérialisé et envoyé à l'objet distant qui le désérialise avant de l'utiliser
  - Un objet d'une classe qui implémente l'interface Remote
    - C'est l'objet stub qui est sérialisé et envoyé à l'objet distant
- Sinon une exception est levée

# Passage d’un paramètre de classe inconnue du serveur

- Le client invoque une méthode avec un paramètre inconnu du serveur.
- Le Stub sérialise l’instance du paramètre et l’envoie au serveur.
- Le serveur charge la classe d’après ***java.rmi.server.codebase***.
- L’objet est désérialisé et la méthode est invoquée.
- Le squelette retourne le résultat au client.

# Passage de paramètre inconnu



# Aller plus loin avec RMI

# Retour sur le transport du code

- Dans certains cas, on a besoin de télécharger du code du serveur vers le client (ou l'inverse).

- Mécanisme de « classpath distribué » : ***codebase***
  - Le ***classpath*** java : où trouver les classes en local
  - Le codebase : où trouver les classes distantes.

- Property java.rmi.server.codebase positionnée :
  - En ligne de commande : ***java -Djava.rmi.server.codebase=http://mycomputer/arch.jar***
  - Ou dans le code :
    - ***System.setProperty( "java.rmi.server.codebase", « http://mycomputer/arch.jar » );***

# Retour sur la sécurité

- Créer un gestionnaire de sécurité
  - ***System.setSecurityManager (new SecurityManager())*** ;
- La politique de sécurité
  - Dans un fichier à part.
  - Localisée via la ***property java.security.policy***.
  - Outil policy tool
    - ***https://docs.oracle.com/javase/7/docs/technotes/guides/security/PolicyFiles.html***

# Retour sur la sécurité : fichier de politique de sécurité

grant [signedBy "signer_names" ,]

[codeBase "URL"] // ending with / includes all class files in the specified directory; ending with /* includes all class and jar files in the directory; ending with /- includes all class and jar files in the directory tree rooted at the specified dir.

[principal principal_class_name "principal_name", ]+

{

[permission permission_class_name ["name",] ["action",]

[signedBy "signer_names"] ; ]+

...

};

Exemples :

grant codeBase "file:/home/jones/src/" {

permission java.security.AllPermission;

};

grant codeBase "file:.${/}bin/" { permission java.io.FilePermission "..${/}*", "read" ; };

donne l'autorisation aux fichiers class chargés depuis le sous-répertoire bin à lire les fichiers présents dans le répertoire parent. ${/} permet d'avoir un séparateur de répertoire portable.

# La classe Naming

- Encapsule le dialogue avec plusieurs objets
- Serveur de liaison
- Méthodes statiques
  - ***bind(String url, Remote r)***
  - ***rebind(String url, Remote r)***
  - ***unbind(String url)***
  - ***Remote lookup(String url)***
  - ***String[] list()***

# La couche de transport

- Par défaut, TCP est utilisé par la couche de transport
- **RMISocketFactory**
  - Dans le cas de ***firewall/proxies***, invocation des méthodes en utilisant un POST HTTP
- La propriété ***java.rmi.server.disableHttp=true*** désactive le tunneling HTTP
- Mais la couche de transport est personnalisable
  - utilisation d'autres classes que Socket et SocketServer basées sur
    - TCP
    - UDP

# Personnaliser la couche de transport

- Ecrire deux sous classes de
  - ***java.rmi.RMIClientSocketFactory*** et
  - ***java.rmi.RMIServerSocketFactory***
    - qui utilisent 2 autres classes de transport que Socketet SocketServer
    - par exemple CompressionSocket et CompressionServerSocket
- Spécifier les factories dans le constructeur de l'objet distant qui hérite de la classe ***UnicastRemoteObject***.

# L’activation d’objets distants

- En JDK1.1, tous les objets distants étaient actifs au démarrage du serveur RMI
- JDK 1.2 introduit le démon ***rmid***
  - ***Rmid*** démarre une JVM qui sert l’objet distant au moment de l’invocation d’une méthode

# Le ramasse miettes distribué dgc

- Ramasse les objets distants qui ne sont plus référencés
- Basé sur le comptage de références
- Interagit avec les GCs locaux de toutes les JVMs
  - maintien des weak references pour éviter le ramassage par le GC local

# Conclusion

- 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 passes par valeur (copie) et doivent être serialisables (étendent l'interface ***java.io.Serializable***).
  - Les objets distants sont passés par référence et sont désignés par leur interface.

- 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***)