

Modèle client-serveur



Daniel Hagimont

IRIT/ENSEEIH

**2 rue Charles Camichel - BP 7122
31071 TOULOUSE CEDEX 7**

**Daniel.Hagimont@enseeiht.fr
<http://hagimont.perso.enseeiht.fr>**

Remerciements

Michel Riveill

Plan



- Principes généraux
 - Modèle client-serveur
 - Appel de procédure à distance (Remote Procedure Call)
- Application dans l'environnement Java
 - Java Remote Method Invocation (RMI)

Modèle client-serveur Définition



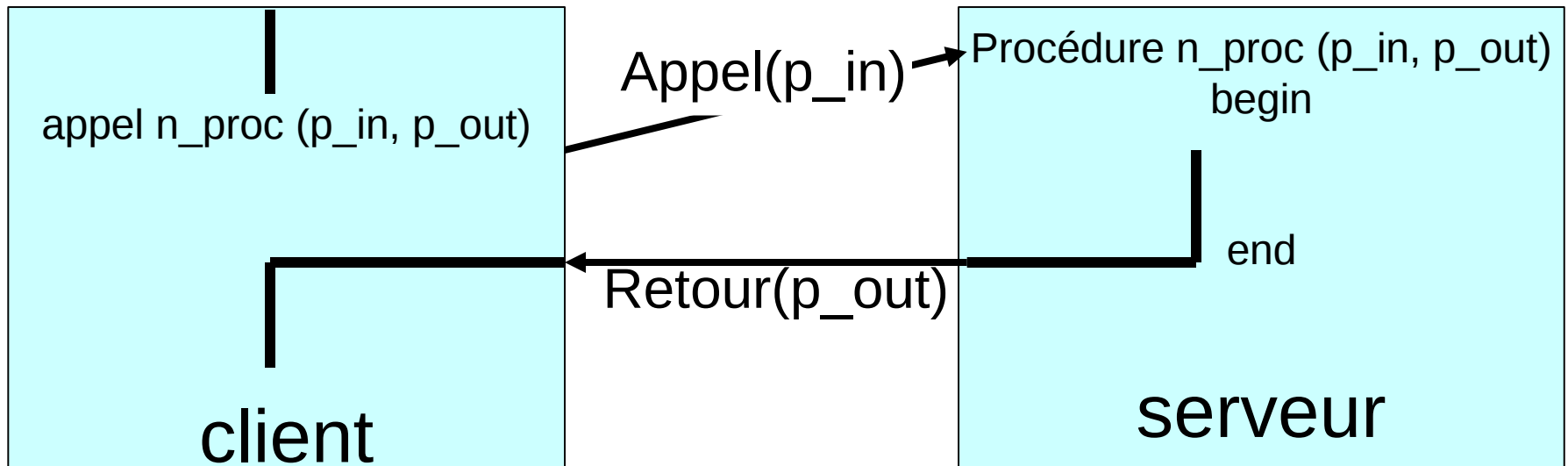
■ Application client/serveur

- application qui fait appel à des services distants au travers d'un échange de messages (les requêtes) plutôt que par un partage de données (mémoire ou fichiers)
- serveur
 - programme offrant un service sur un réseau (par extension, machine offrant un service)
- client
 - programme qui émet des requêtes (ou demandes de service). Il est toujours l'initiateur du dialogue

Modèle client-serveur

Communication par messages

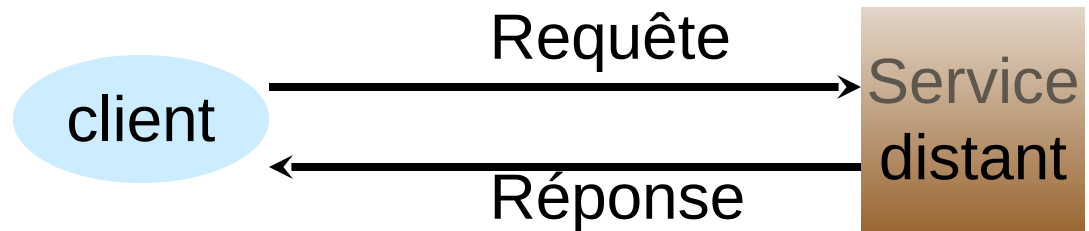
- Deux messages (au moins) échangés
 - Le premier message correspondant à la requête est celui de l'appel de procédure, porteur des paramètres d'appel.
 - Le second message correspondant à la réponse est celui du retour de procédure porteur des paramètres résultats.



Modèle client-serveur

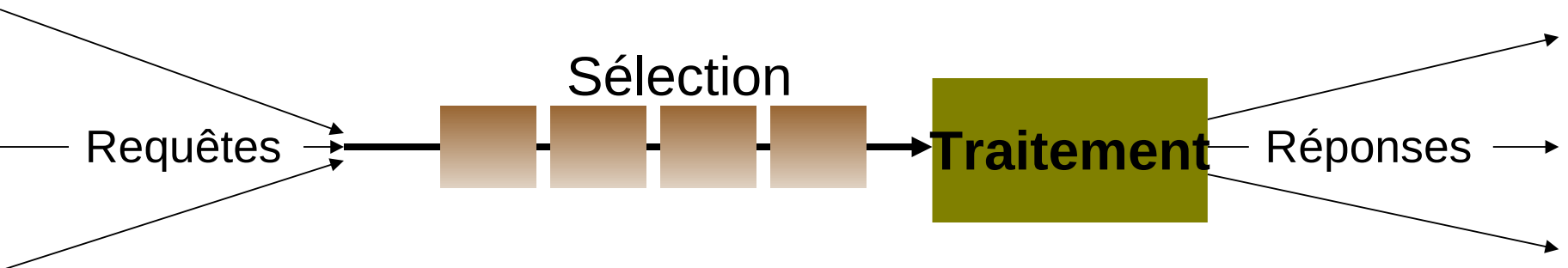
Principe

- Vu du client



- Vu du serveur

- Gestion des requêtes (priorité)
- Exécution du service (séquentiel, concurrent)
- Mémorisation ou non de l'état du client



Différents types de service



- Pas de modification de données rémanentes sur le serveur
 - ex: calcul fonction ou lecture donnée
- Modification du contexte d'exécution sur le site distant
 - ex: serveur de fichiers ou d'objets
 - problèmes de la concurrence et des pannes

Modèle client-serveur

Exemple



- Serveur de fichiers (nfsd)
- Serveur d'impressions (lpd)
- Serveur de calcul
- Serveur base de données
- Serveur de noms (annuaire des services)

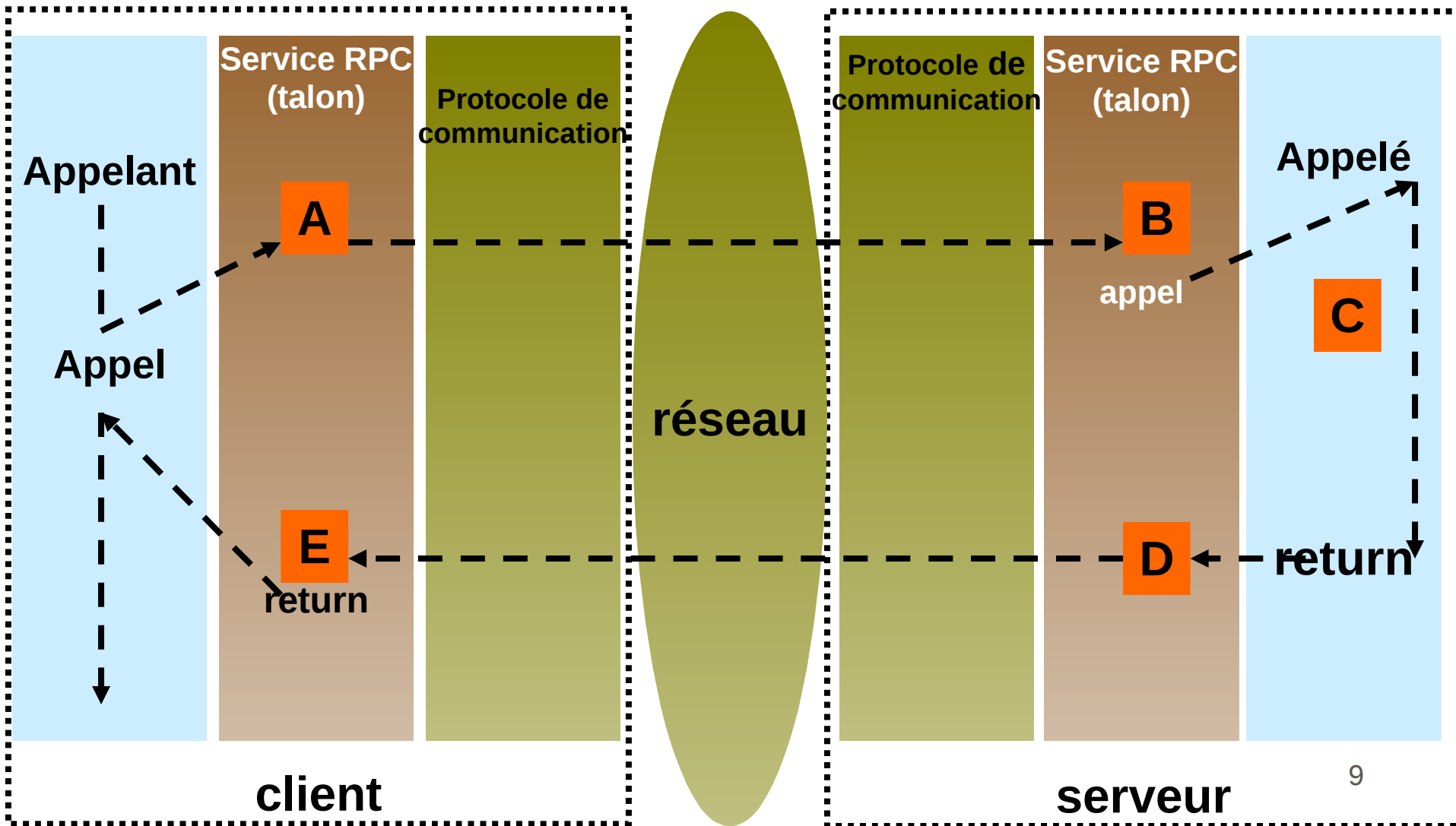
Remote Procedure Call (RPC)

Principes



- Génération du code
 - d'envoi et réception des messages
 - De détection et réémission des messages perdus
- Objectif : le programmeur développe son application comme si elle était centralisée

RPC [Birrel & Nelson 84] Principe de réalisation



RPC (A)

Principe de fonctionnement




■ Côté de l'appelant

- Le client réalise un appel procédural vers la procédure talon client (stub)
 - transmission de l'ensemble des arguments
- au point A
 - le talon collecte les arguments et les assemble dans un message (empaquetage - parameter marshalling)
 - un identificateur est généré pour le RPC et joint au message
 - Un délai de garde est armé
 - Pb : détermination de l'adresse du serveur (annuaire de services)
 - le talon transmet les données au protocole de transport pour émission sur le réseau

RPC (B et C)

Principe de fonctionnement



■ Côté de l'appelé

- le protocole de transport délivre le message au service de RPC (talon serveur/skeleton)
- au point B
 - le talon désassemble les arguments (dépaquetage - unmarshalling)
 - l'identificateur de RPC est enregistré
- l'appel est ensuite transmis à la procédure distante requise pour être exécuté (point C)
- Le retour de la procédure redonne la main au service de RPC et lui transmet les paramètres résultats (point D)

RPC (D)

Principe de fonctionnement



- Côté de l'appelé

- au point D

- les arguments de retour sont empaquetés dans un message
 - un autre délai de garde est armé
 - le talon transmet les données au protocole de transport pour émission sur le réseau

RPC (E)

Principe de fonctionnement



■ Côté de l'appelant

- l'appel est transmis au service de RPC (point E)
 - les arguments de retour sont dépaquetés
 - le délai de garde armé au point A est désarmé
 - un message d'acquiescement avec l'identificateur du RPC est envoyé au talon serveur (le délai de garde armé au point D peut être désarmé)
 - les résultats sont transmis à l'appelant lors du retour de procédure

RPC

Rôle des talons



Talon client - stub

- C'est la procédure d'interface du site client
 - qui reçoit l'appel en mode local
 - le transforme en appel distant en envoyant un message
 - reçoit les résultats après l'exécution
 - retourne les paramètres résultats comme dans un retour de procédure

Talon serveur - skeleton

- C'est la procédure sur le site serveur
 - qui reçoit l'appel sous forme de message
 - fait réaliser l'exécution sur le site serveur par la procédure serveur (choix de la procédure)
 - retransmet les résultats par message

RPC

Perte de message



- Coté client
 - Si le délai de garde expire
 - Réémission du message (avec le même identificateur)
 - Abandon après N tentatives
- Coté serveur
 - Si le délai de garde expire
 - Si on reçoit un message avec un identificateur identique
 - Réémission de la réponse
 - Abandon après N tentatives
- Coté client
 - Si on reçoit une réponse avec un identificateur déjà reçu
 - Réémission du message d'acquiescement

RPC

Problèmes



- Traitement des défaillances
 - Congestion du réseau ou du serveur
 - la réponse ne parvient pas avant une date fixée par le client (système temps critique)
 - Panne du client pendant le traitement de la requête
 - Panne du serveur avant ou pendant le traitement de la requête
 - Panne du système de communication
 - Quelles garanties ?
- Problèmes de sécurité
 - authentification du client
 - authentification du serveur
 - confidentialité des échanges
- Performance
- Désignation
- Aspects pratiques
 - Adaptation à des conditions multiples (protocoles, langages, matériels)

RPC

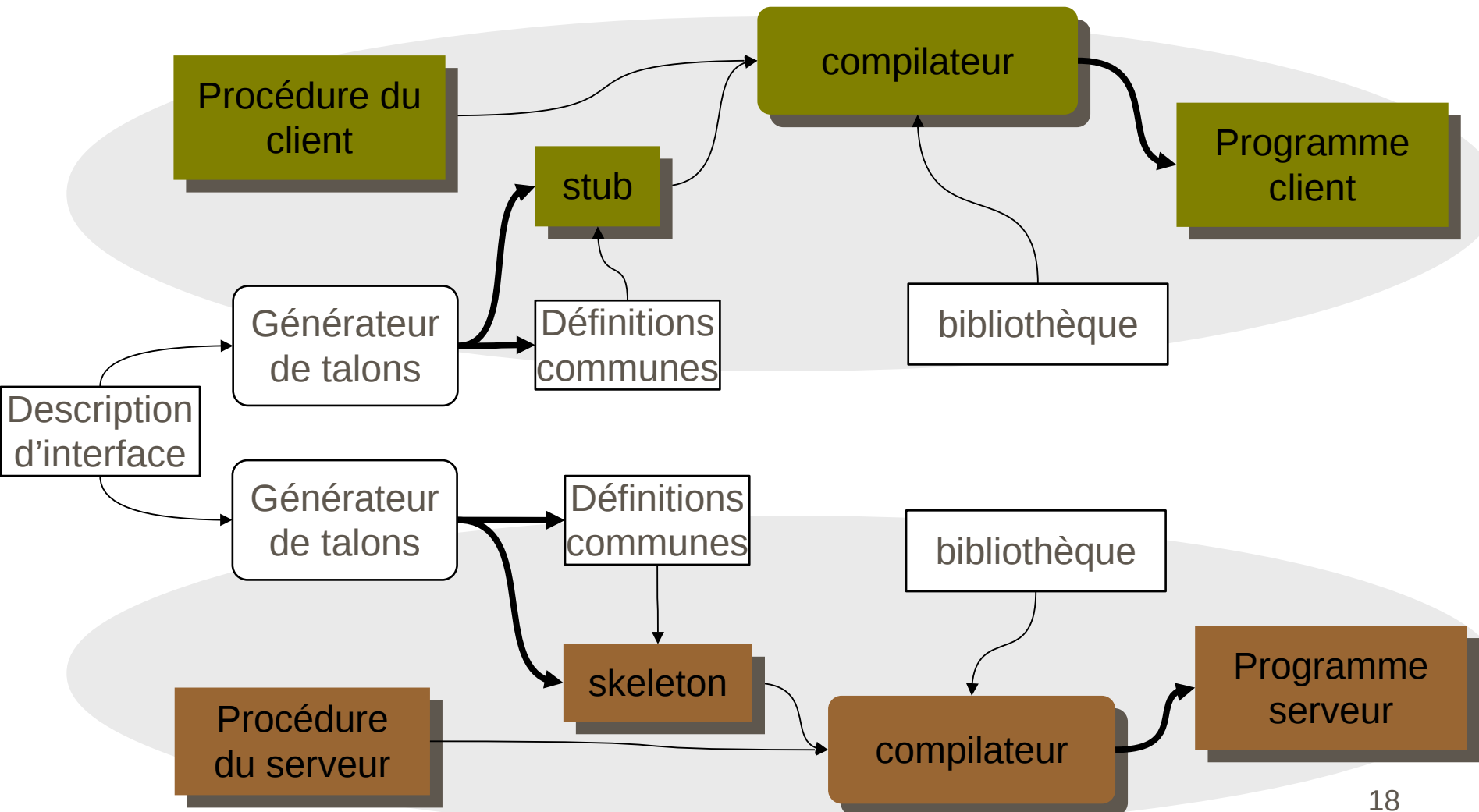
IDL : spécification des interfaces



- Utilisation d'un langage de description d'interface (IDL)
 - Spécification commune au client et au serveur
 - Définition des types et natures des paramètres (IN, OUT, IN-OUT)
- Utilisation de ces définitions pour générer automatiquement :
 - le talon client (ou proxy, stub)
 - le talon serveur (ou squelette, skeleton)

IDL

Mode opératoire (général)



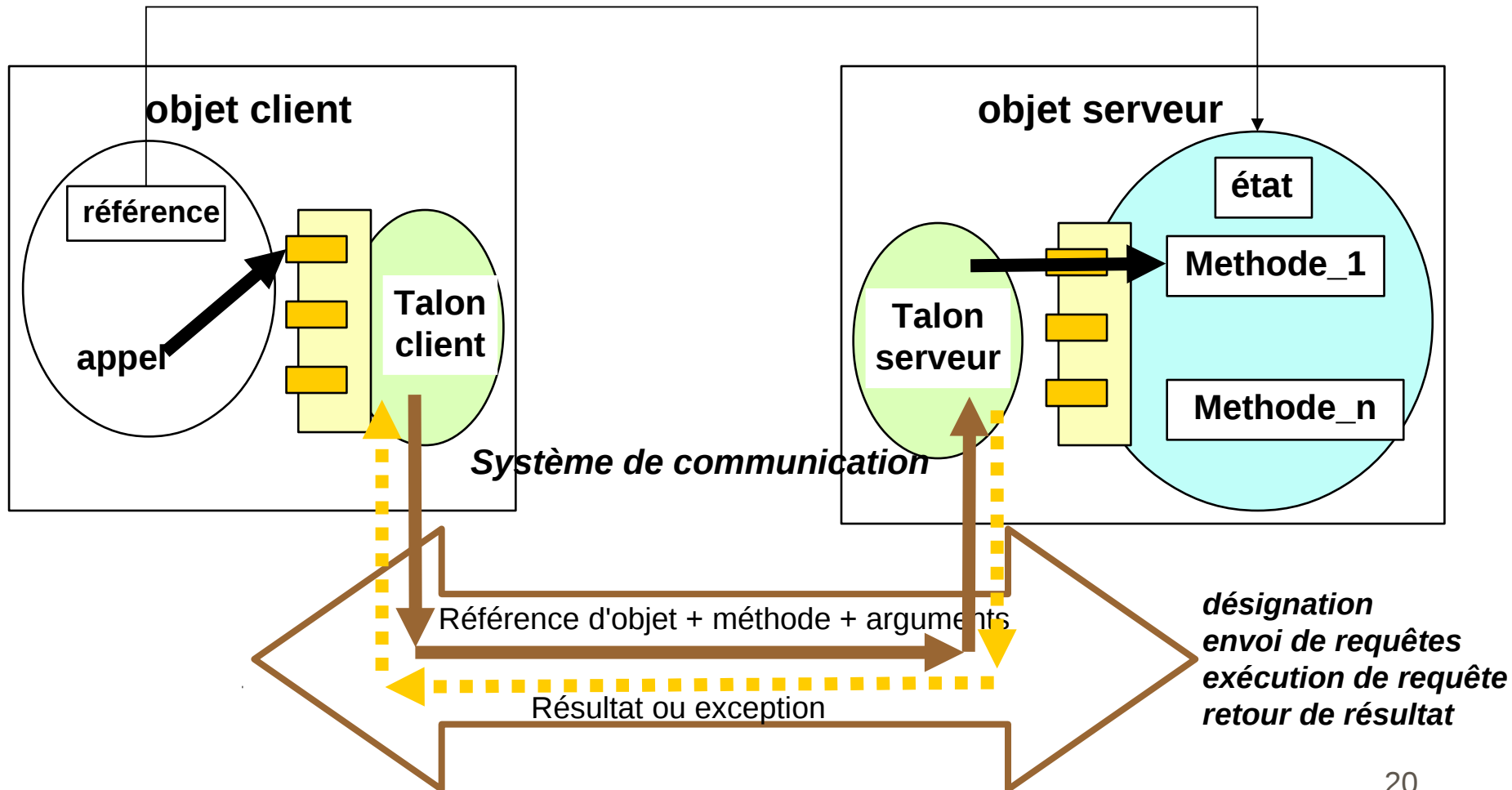
Java Remote Method Invocation RMI



- Un RPC objet intégré à Java
- Interaction d'objets situés dans des espaces d'adressage différents (des *Java Virtual Machines* - JVM) sur des machines distinctes
- Simple à mettre en œuvre : un objet distribué se manipule comme tout autre objet Java

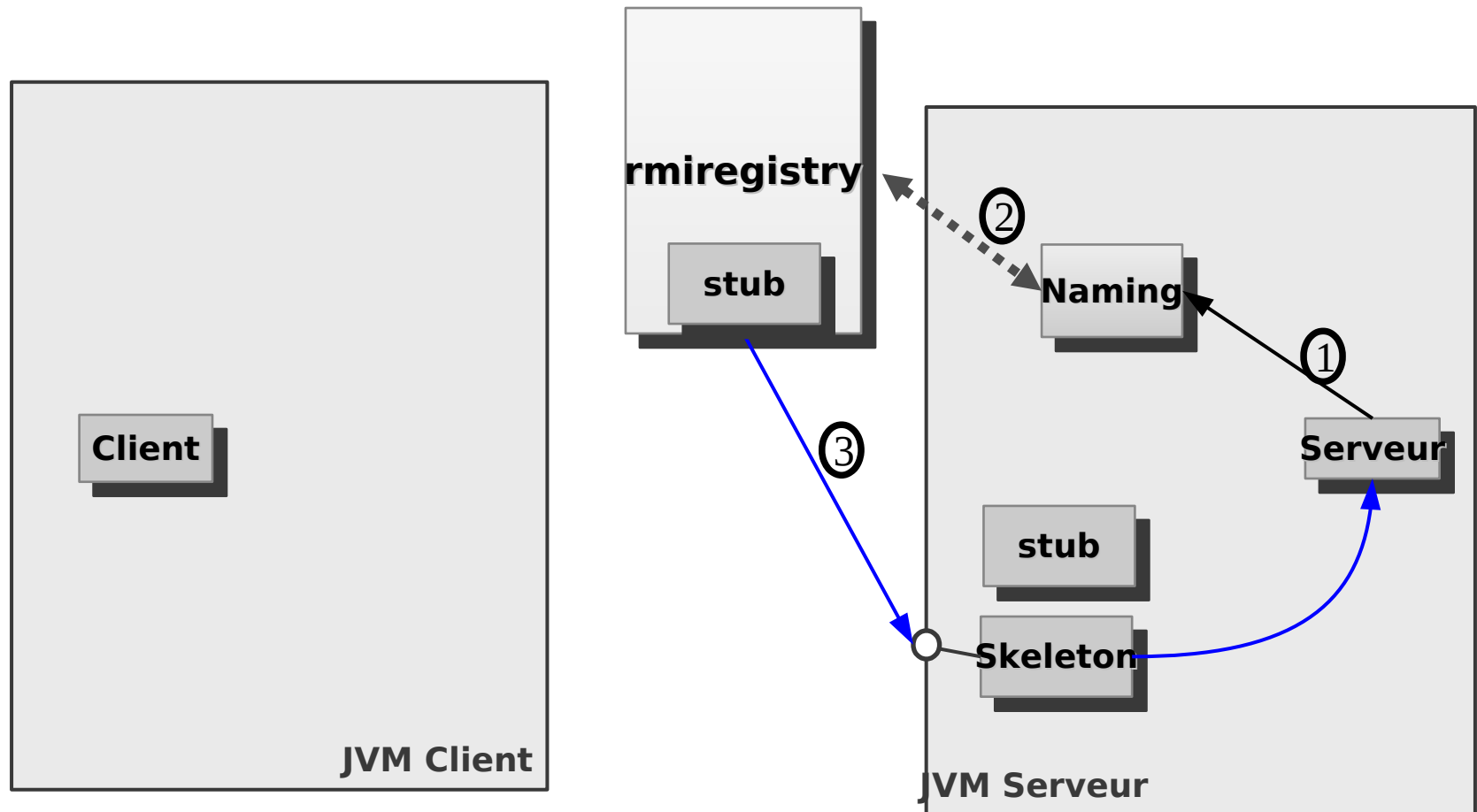
Java RMI

Principe



Java RMI

Coté serveur



Java RMI

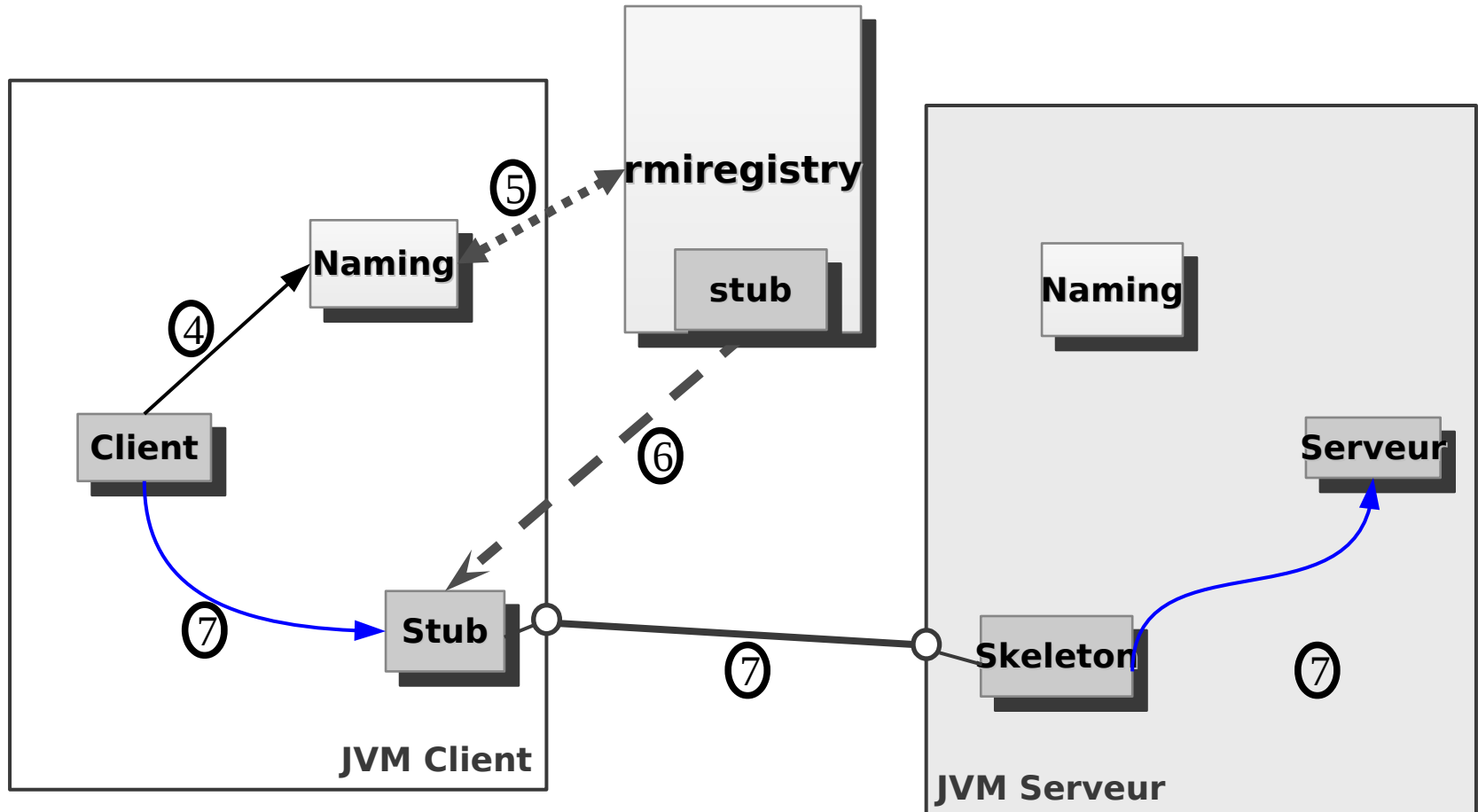
Coté serveur



- 0 - A la création de l'objet, un *stub* et un *skeleton* (avec un port de communication) sont créés coté serveur
- 1 - L'objet serveur s'enregistre auprès d'un annuaire (*rmiregistry*) en utilisant la classe *Naming* (méthode *rebind*)
- 2 - L'annuaire (*rmiregistry*) enregistre le *stub* de l'objet
- 3 - L'annuaire est prêt à donner des références à l'objet serveur

Java RMI

Coté client



Java RMI

Coté client



- 4 - L'objet client fait appel à l'annuaire (*rmiregistry*) en utilisant la classe *Naming* pour localiser l'objet serveur (méthode *lookup*)
- 5 - L'annuaire délivre une copie du *stub*
- 6 - L'objet *stub* est installé et sa référence est retournée au client
- 7 - Le client effectue l'appel à l'objet serveur par appel à l'objet *stub*

Java RMI

Utilisation



- **codage**
 - description de l'interface du service
 - écriture du code du serveur qui implante l'interface
 - écriture du client qui appelle le serveur
- **compilation**
 - compilation des sources (javac)
 - génération des *stub* et *skeleton* (rmic)
 - *(plus la peine, génération dynamique)*
- **activation**
 - lancement du serveur de noms (*rmiregistry*)
 - lancement du serveur
 - lancement du client

Java RMI

Manuel d'utilisation



- Définition de l'interface de l'objet réparti
 - interface publique
 - interface : "extends java.rmi.Remote"
 - methodes : "throws java.rmi.RemoteException"
 - paramètres sérializables : "implements Serializable"
 - paramètres références : "implements Remote"
- Ecrire une implémentation de l'objet réparti
 - classe : "extends java.rmi.server.UnicastRemoteObject"

Java RMI

Exemple : Interface



fichier Hello.java

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

Description
de
l'interface

Java RMI

Exemple : Serveur

fichier HelloImpl.java

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

public class HelloImpl extends UnicastRemoteObject
    implements Hello {
    String message;

    // Implémentation du constructeur
    public HelloImpl(String msg) throws java.rmi.RemoteException {
        message = msg;
    }
    // Implémentation de la méthode distante
    public void sayHello() throws java.rmi.RemoteException {
        System.out.println(message);
    }

    ...
}
```

Réalisation
du
serveur

Java RMI

Exemple : Serveur

fichier HelloImpl.java

```
...  
  
public static void main(String args[]) {  
    try {  
        // Crée une instance de l'objet serveur.  
        Hello obj = new HelloImpl();  
        // Enregistre l'objet créer auprès du serveur de noms.  
        Naming.rebind("//ma_machine/mon_serveur", obj);  
        System.out.println("HelloImpl " + " bound in registry");  
    } catch (Exception exc) {... }  
}  
}
```

Réalisation
du
serveur
(suite)

ATTENTION : dans cet exemple le serveur de nom doit être activé avant la création du serveur

Java RMI

Activation du serveur de nom par le serveur

fichier HelloImpl.java

```
public static void main(String args[]) {
    int port;    String URL;

    try {        // transformation d'une chaîne de caractères en entier
        Integer l = new Integer(args[0]); port = l.intValue();
    } catch (Exception ex) {
        System.out.println(" Please enter: java HelloImpl <port>"); return;
    }

    try {
        // Création du serveur de nom - rmiregistry
        Registry registry = LocateRegistry.createRegistry(port);

        // Création d'une instance de l'objet serveur
        Hello obj = new HelloImpl();

        // Calcul de l'URL du serveur
        URL = "://" + InetAddress.getLocalHost().getHostName() + ":" +
            port + "/mon_serveur";
        Naming.rebind(URL, obj);
    } catch (Exception exc) { ... }
}
```

Java RMI

Exemple : Client



fichier HelloClient.java

```
import java.rmi.*;

public class HelloClient {
    public static void main(String args[]) {
        try {
            // Récupération d'un stub sur l'objet serveur.
            Hello obj = (Hello) Naming.lookup("//ma_machine/mon_serveur");
            // Appel d'une méthode sur l'objet distant.
            obj.sayHello();
        } catch (Exception exc) { ... }
    }
}
```

Réalisation
du
client

Java RMI Compilation

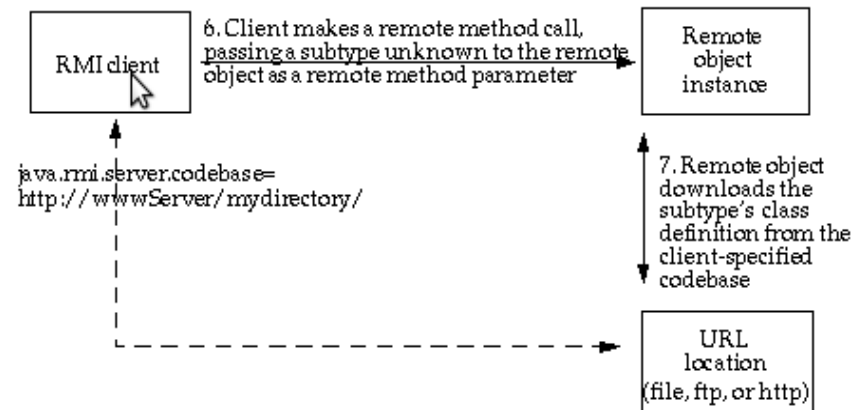


- Compilation de l'interface, du serveur et du client
 - `javac Hello.java HelloImpl.java HelloClient.java`
- Génération des talons (*automatique maintenant*)
 - `rmic HelloServeur`
 - *skeleton* dans `HelloImpl_Skel.class`
 - *stub* dans `HelloImpl_Stub.class`.

Java RMI

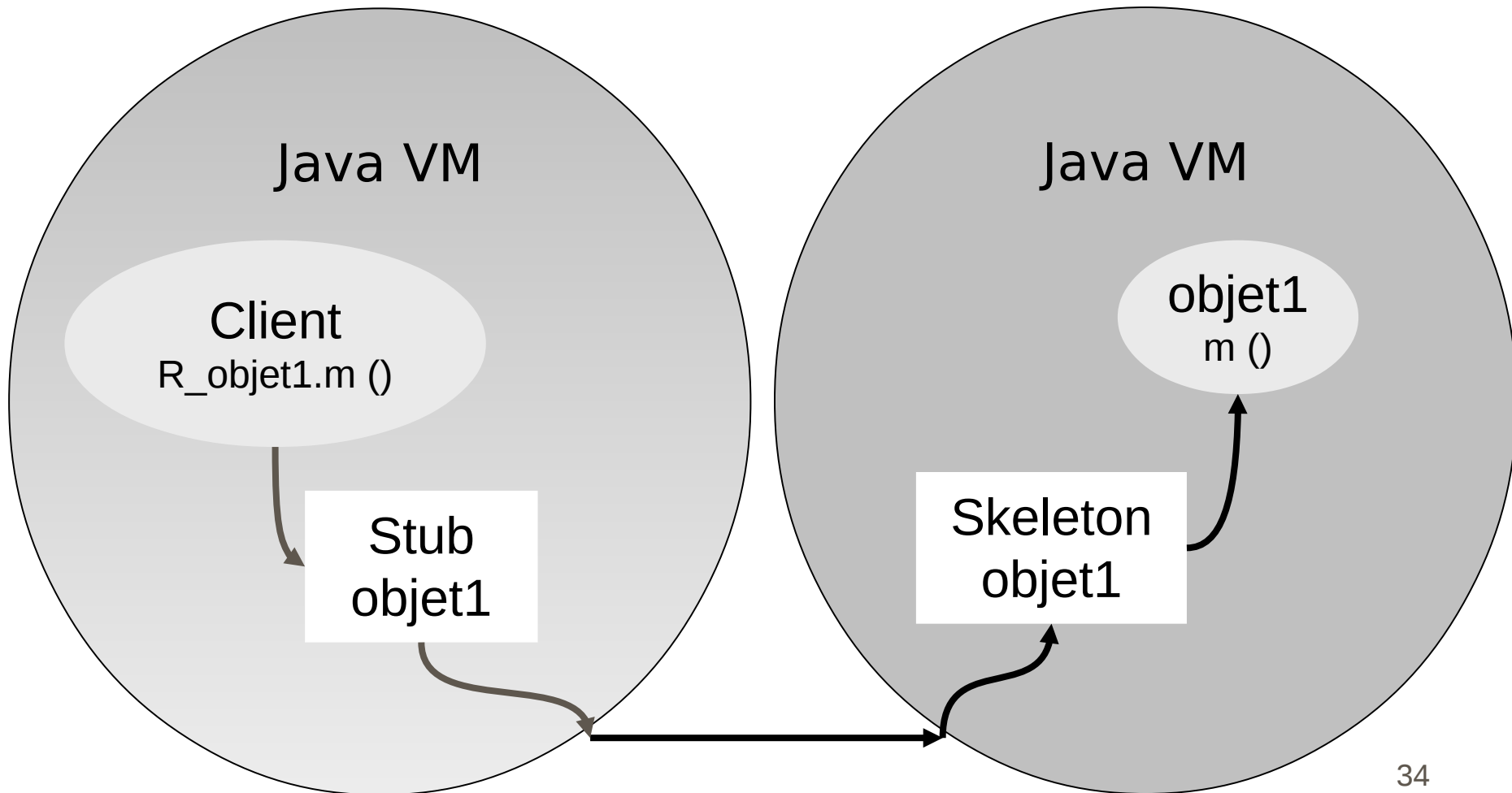
Déploiement

- 1) Activation du serveur de nom
 - rmiregistry &
- 2) Activation du serveur
 - java HelloImpl
 - java -Djava.rmi.server.codebase=http://ma_machine/...
 - path indiquant à quelle endroit la machine virtuelle cliente va pouvoir chercher des classes manquantes
 - Exemple : sérialisation
- 3) Activation du client
 - java HelloClient



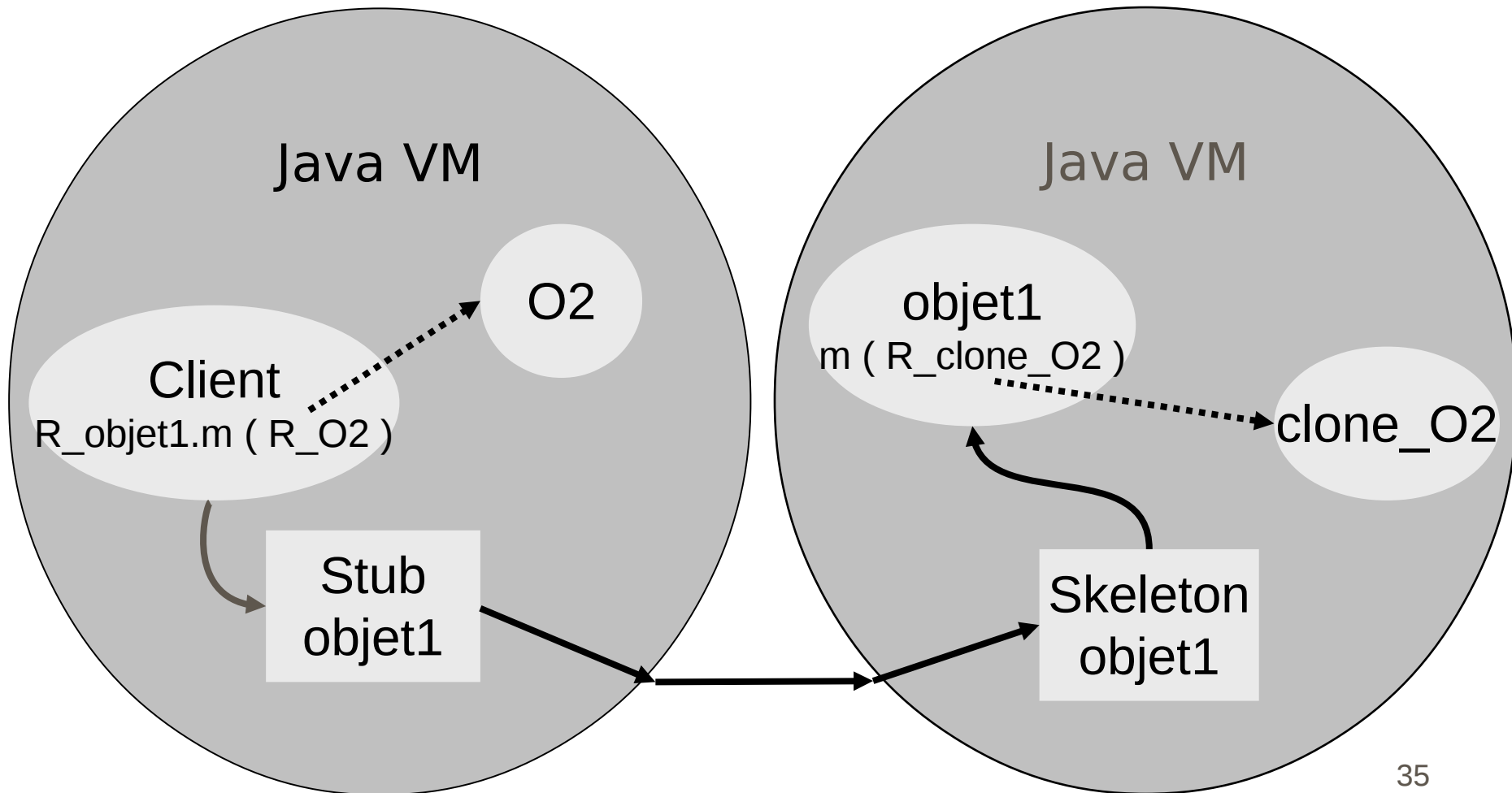
Java RMI

Principe de l'appel de méthode



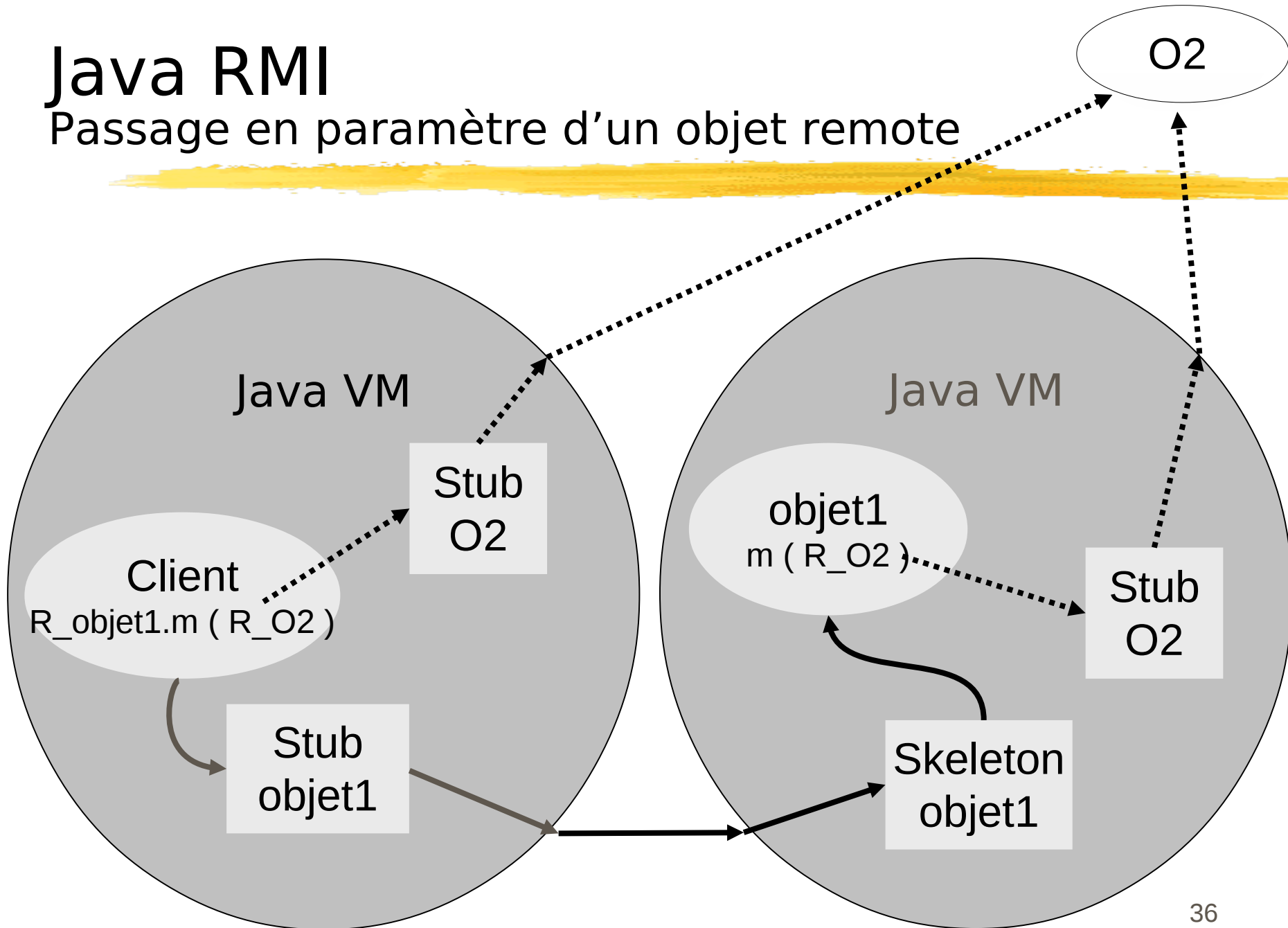
Java RMI

Passage en paramètre d'un objet sérialisable



Java RMI

Passage en paramètre d'un objet remote



Java RMI : bilan



- Très bon exemple de RPC
 - facilité d'utilisation
 - intégration au langage Java
 - Passage de référence -> sérialisation ou référence à distance
 - Déploiement du code -> chargement dynamique des classes sérialisables
 - Désignation par des URL


Ouverture :

RPC asynchrone



- Le client poursuit son exécution après l'émission du message d'appel
 - la procédure distante s'exécute en parallèle avec la poursuite du client et retourne les paramètres résultats en fin de son exécution
 - le client récupère les résultats quand il en a besoin (primitive spéciale de lecture)
 - la lecture rend un résultat nul si le résultat n'est pas disponible
 - la lecture bloque le client si le résultat n'est pas disponible
 - avantage : parallélisme plus important
 - critique : le client ne retrouve pas la sémantique de l'appel de procédure

Les limites du modèle client-serveur



- Services pour la construction d'applications réparties
 - le RPC est un mécanisme de “bas niveau”
 - des services additionnels sont nécessaires pour la construction d'applications réparties (désignation, fichiers répartis, sécurité, etc.)
 - CORBA, EJB ...
- Outils de développement
 - limités à la génération automatique des talons
 - peu (ou pas) d'outils pour la construction ou le déploiement d'applications réparties (composants)

Des tutoriaux de programmation RMI plein de Web ...