

Licence en génie logiciel et systèmes d'information
Niveau: 3^{ème} année
Semestre 1

Chapitre 4: Middleware
Java RMI

Dhikra KCHAOU
dhikrafsegs@gmail.com

Plan du chapitre

1. Introduction
2. Présentation d'un Middleware
3. Fonctions d'un middleware
4. Accès à un objet distant via un middleware
5. Architecture générale d'un middleware
6. Fonctionnement général d'un appel d'opération à distance
7. Présentation du Middleware Java RMI
8. Exemple

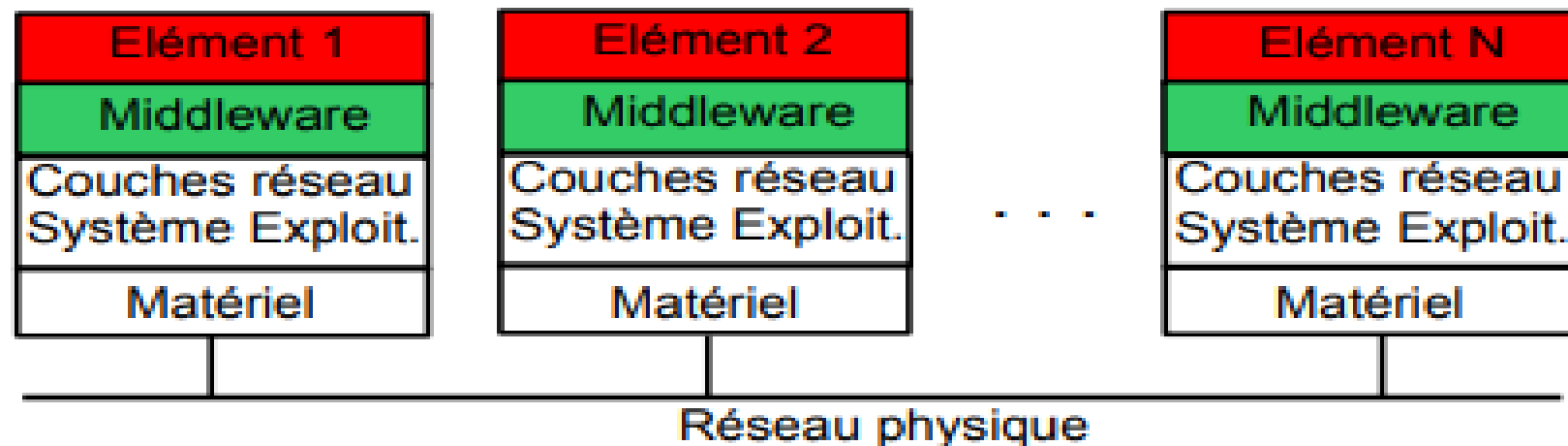
Introduction

Motivations : L'interface fournie par les systèmes d'exploitation et les systèmes de communication est encore très complexe pour être utilisée par les applications:

- ▶ Hétérogénéité (matérielle et logicielle)
- ▶ Complexité des mécanismes (bas niveau)
- ▶ Nécessité de gérer la répartition
- ▶ **Solution**: Introduire une couche logicielle intermédiaire (répartie) entre les niveaux bas (systèmes et réseaux) et le niveau haut (Applications)
→ **Intergiciel** ou **Middleware**

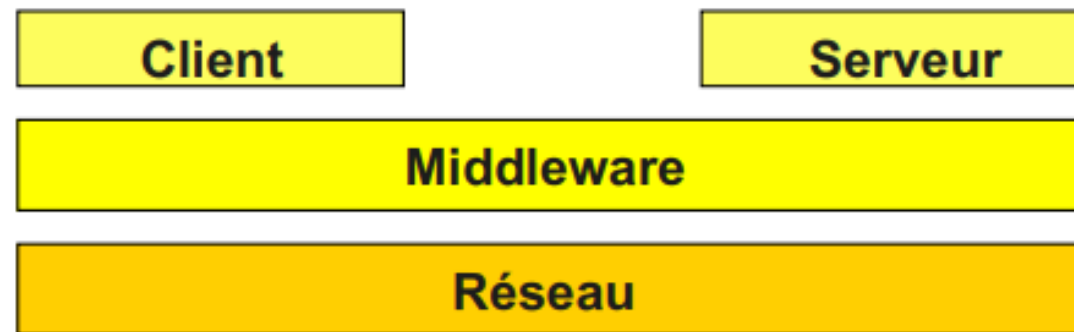
Middleware

- ▶ Middleware ou intergiciel : couche logiciel
 - ▶ S'intercale entre le système d'exploitation/réseau et les éléments de l'application réparti.
 - ▶ Offre un ou plusieurs services de communication entre les éléments formant l'application ou le système distribué.



Middleware

- ▶ Grossièrement, **un middleware = la gestion du protocole applicatif + l'API d'accès à la couche transport + des services complémentaires.**
- ▶ C'est un ensemble de services logiciels construits au dessus d'un protocole de transport afin de permettre l'échange de requête/réponse entre le client et le serveur de manière transparente.



Middleware

- ▶ Complément de services du réseau permettant la réalisation du dialogue client/serveur :
 - ▶ prend en compte les requêtes de l'application cliente
 - ▶ les transmet de manière transparente à travers le réseau jusqu'au serveur
 - ▶ prend en compte les données résultat du serveur et les transmet vers l'application cliente
- ▶ L'objectif essentiel du middleware est d'offrir aux applications une interface unifiée permettant l'accès à l'ensemble des services disponibles sur le réseau : l'API

Fonctions d'un Middleware

- ▶ Masquer l'hétérogénéité (machines, systèmes, protocoles de communication)
- ▶ Fournir une API (Application Programming Interface) de haut niveau:
 - ▶ Permet de masquer la complexité des échanges
 - ▶ Facilite le développement d'une application répartie
- ▶ Rendre la répartition aussi invisible (transparente) que possible
- ▶ Fournir des services répartis d'usage courant (*Service de nommage* pour connaître les éléments présents, ...)

Service de nommage d'un middleware

- ▶ Un middleware utilise un service de nommage pour enregistrer, identifier et rechercher les éléments et services connectés via le middleware
- ▶ Le Service d'un élément = (interfaces d') opération(s) offerte(s) par un élément et pouvant être appelée(s) par d'autres.
- ▶ Un élément enregistre les opérations qu'il offre
- ▶ Pour rechercher une opération ou un élément, on distingue 2 modes:
 - ▶ On demande à recevoir une référence sur un élément qui offre une opération compatible avec celle que l'on cherche
 - ▶ On sait identifier l'élément distant et on demande à récupérer la référence sur cet élément
- ▶ Une fois la référence sur l'élément distant connu, on peut appeler l'opération sur cet élément via cette référence

Les grandes familles de middleware

▶ Appel de procédure/méthode à distance

- ▶ Extension de l'appel local d'opération dans le contexte des systèmes distribués
- ▶ Une partie serveur offre une opération appelée par une partie client
- ▶ Permet d'appeler une procédure/méthode sur un élément/objet distant (presque) aussi facilement que localement
- ▶ Exemples (dans ordre historique d'apparition)
 - ▶ **RPC** (Remote Procedure Call) : solution de Sun pour C/Unix
 - ▶ **CORBA** (Common Object Request Broker Architecture) : standard de l'OMG permettant l'interopérabilité quelque soit le langage ou le système
 - ▶ **Java RMI** (Remote Method Invocation) : solution native de Java

▶ Envoi ou diffusion de messages: Famille des **MOM (Message Oriented Middleware)**

- ▶ Event Service de CORBA, JMS de Java

Familles de middlewares, caractéristiques générales

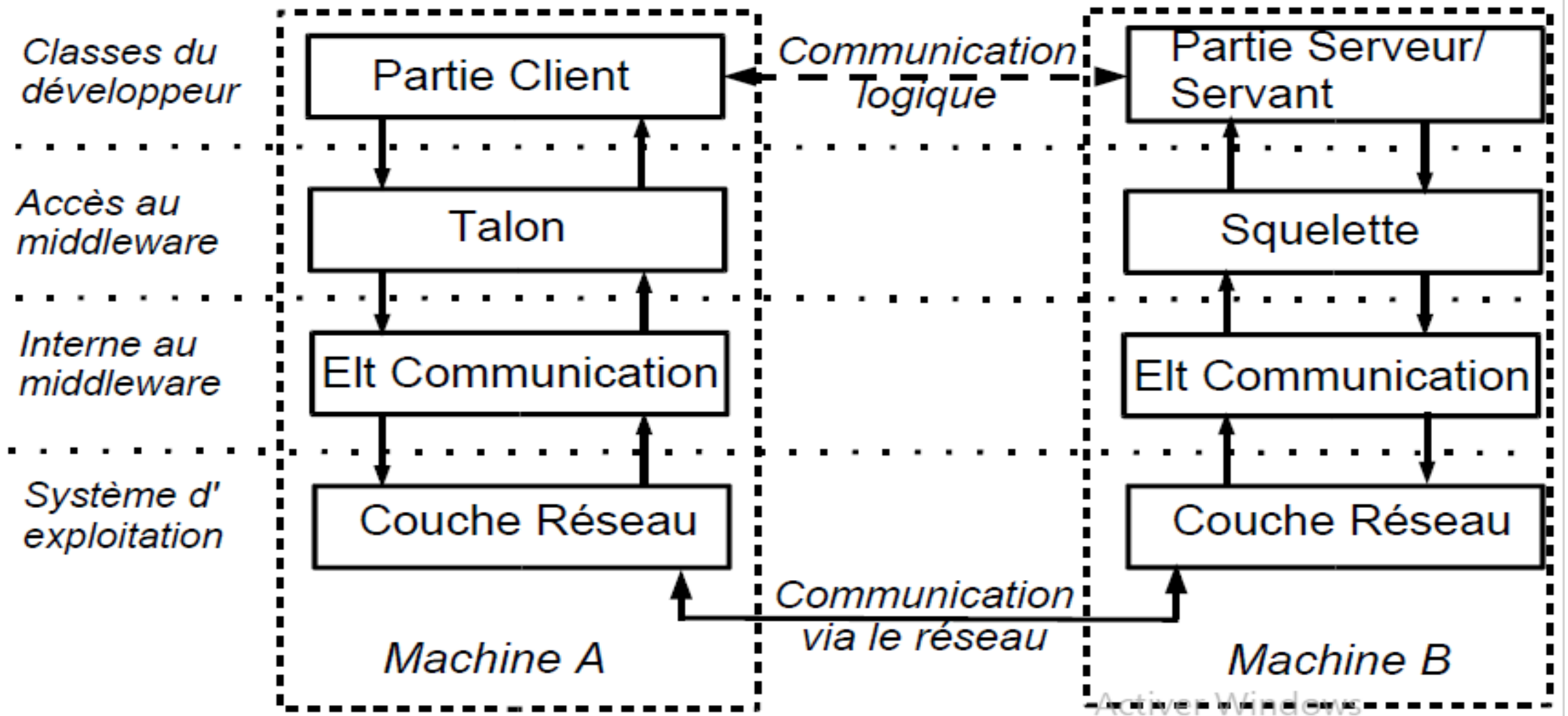
▶ **Modèle RPC/RMI**

- ▶ Interaction forte entre parties client et serveur
- ▶ Appel synchrone (pour le client) d'opération sur le serveur
- ▶ Généralement, le client doit explicitement connaître le serveur
- ▶ Peu dynamique (point de vue serveur) car les éléments serveurs doivent être connus par les clients et lancés avant eux
- ▶ Mode 1 vers 1 : opération appelée sur un seul serveur à la fois

▶ **Modèles à message**

- ▶ Asynchrone et pas d'interaction forte entre éléments (Envoi de message est asynchrone)
 - ▶ Pas de nécessité de connaître les éléments accédant à la mémoire
 - ▶ Permet une plus forte dynamique : ajout et disparition d'éléments connectés au middleware facilités par les faibles interactions et l'anonymat
- ▶ Mode 1 vers n
 - ▶ Diffusion de messages à plusieurs éléments en même temps
 - ▶ Accès aux informations de la mémoire par plusieurs éléments

Architecture générale du Middleware RPC/RMI



Architecture générale du Middleware RPC/RMI

- ▶ Les éléments de l'architecture générale peuvent être classés en 2 catégories:
 - ▶ Éléments communicants réalisés par le développeur
 - ▶ Éléments permettant la communication interne au middleware
- ▶ Description des différents éléments
 - ▶ **Éléments dont le code est généré automatiquement via des outils du middleware**
 - ▶ *Talon (stub)* : élément/proxy du côté client qui offre localement les mêmes opérations (la même interface) que le servant
 - ▶ *Squelette (skeleton)* : élément du côté serveur qui reçoit les requêtes d'appels d'opérations des clients et les lance sur le servant

Description des différents éléments communiquant

- ▶ **Éléments communiquant:**

- ▶ *Partie client* : partie qui appelle l'opération distante
- ▶ *Partie serveur* : partie qui offre l'opération distante via une interface
 - ▶ Appelée aussi « servant » pour certains middleware
 - ▶ A implémenter par le développeur : contient le code des opérations de l'interface
- ▶ Des 2 cotés les éléments de communication entre parties client/serveur sont:
 - ▶ Internes au middleware
 - ▶ Attaquent les couches TCP ou UDP via des sockets pour gérer la communication entre les talons et les squelettes

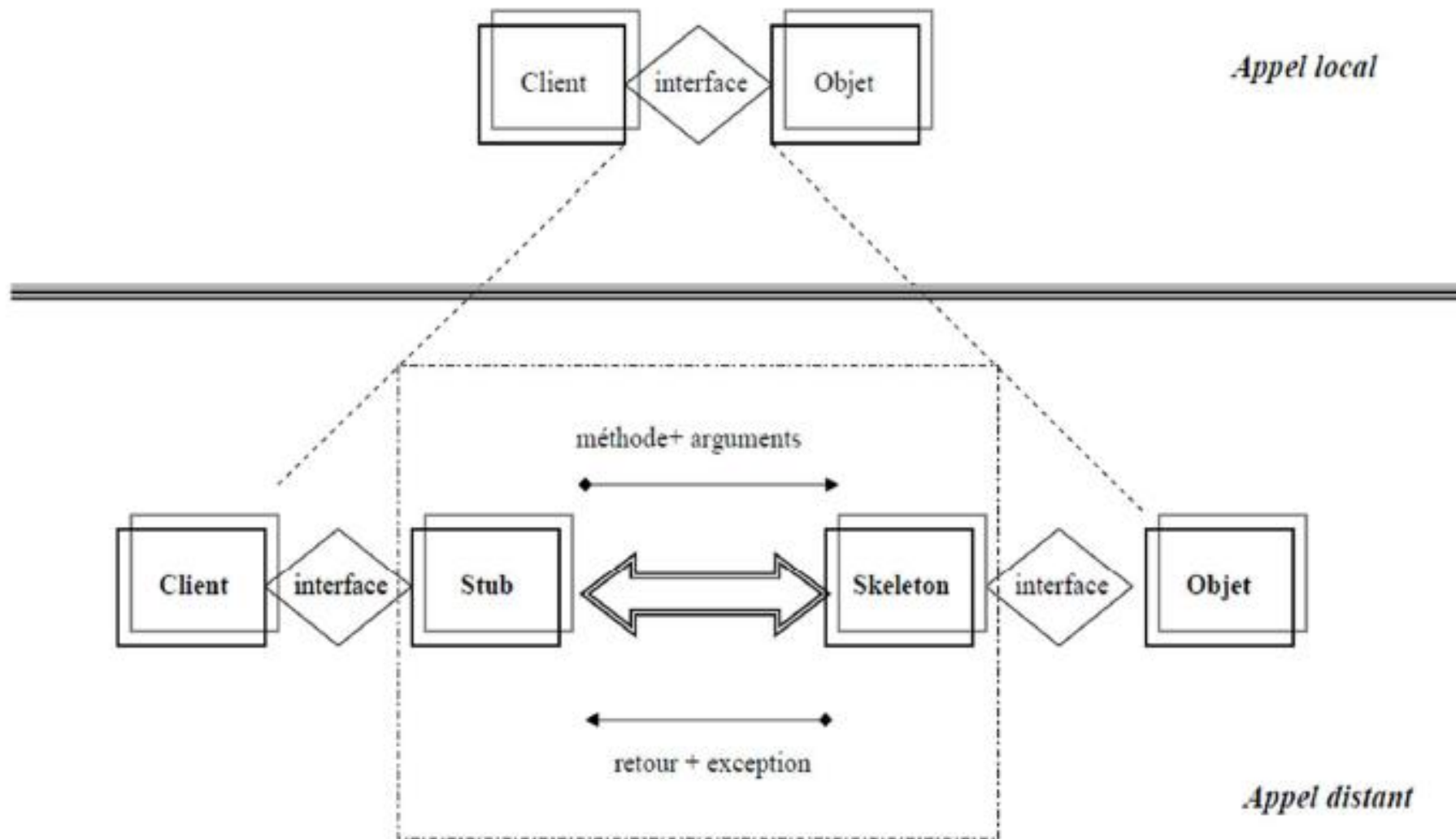
Fonctionnement général d'un appel d'opération à distance

1. La partie client récupère via le service de nommage une référence d'objet sur le servant distant (En réalité, une référence sur le talon local)
2. La partie client appelle une opération sur cette référence d'objet : Appel synchrone : la partie client attend que l'appel de l'opération soit terminé pour continuer son exécution
3. Le talon, qui reçoit cet appel, « compacte » toutes les données relatives à l'appel (identificateur opération + paramètres)
4. L'élément de communication envoie les données à l'élément de communication coté serveur
5. L'élément de communication coté serveur envoie les données au squelette

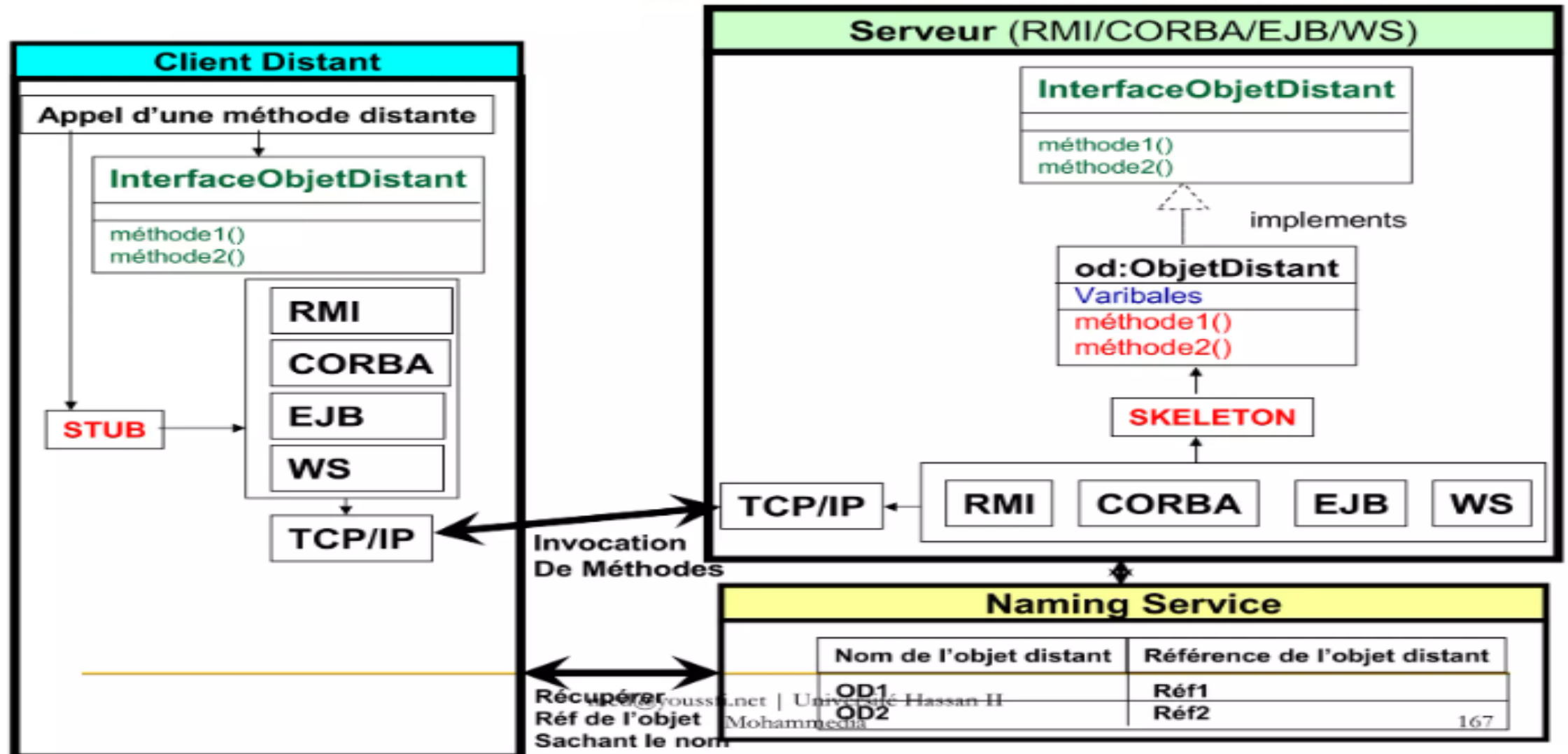
Fonctionnement général d'un appel d'opération à distance

6. Le squelette décompacte les données pour déterminer l'opération à appeler sur le servant
7. L'opération est appelée sur le servant par le squelette
8. Le squelette récupère la valeur de retour de l'opération, la compacte et l'envoie au talon via les éléments de communication Le talon décompacte la valeur et la retourne la valeur à la partie client
9. L'appel de l'opération distante est terminé, la partie client continue son exécution

Appel Local vs Appel à distance

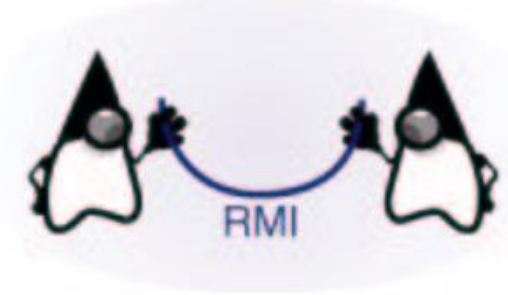


Accès à un objet distant via un middleware



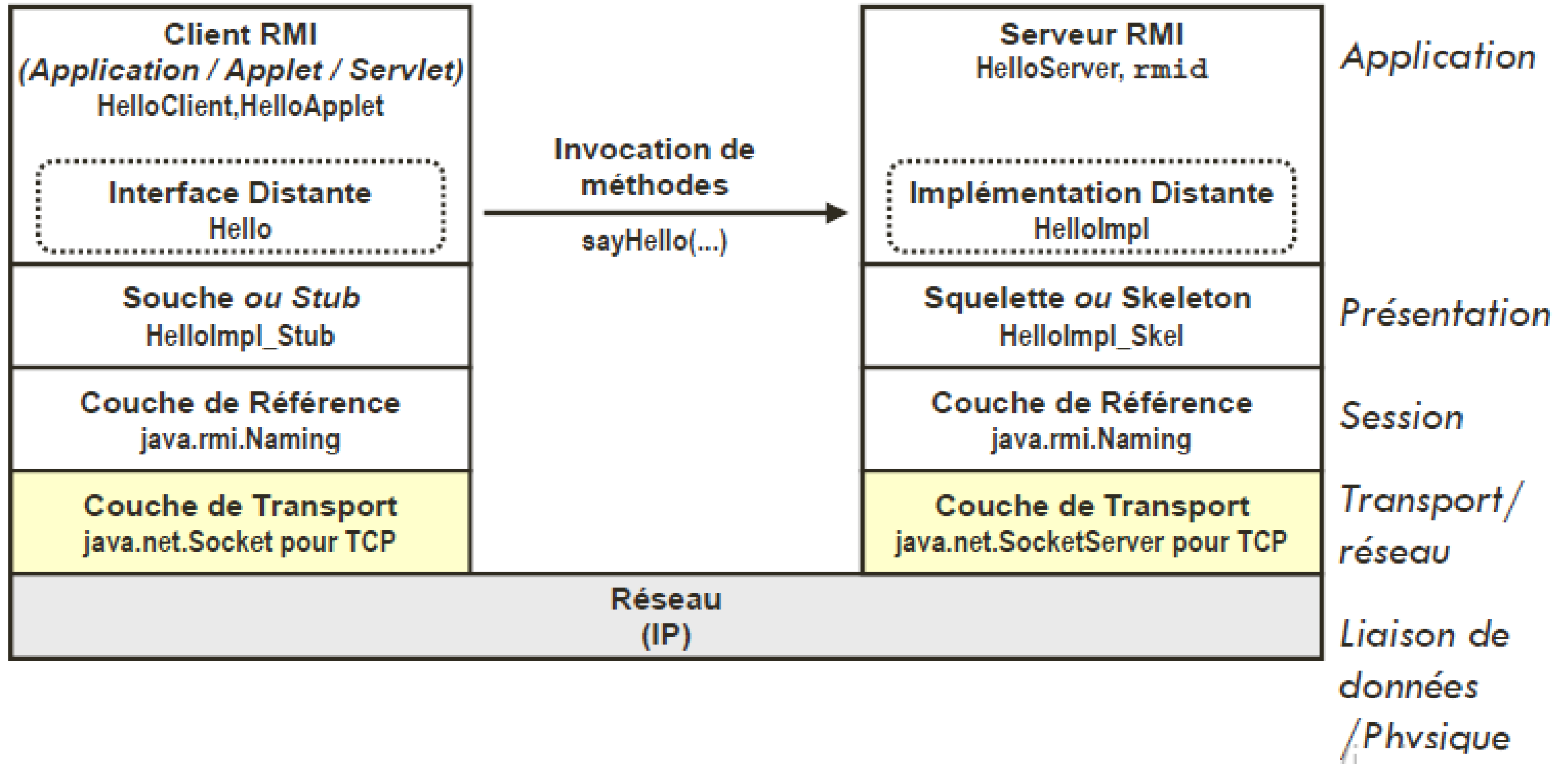
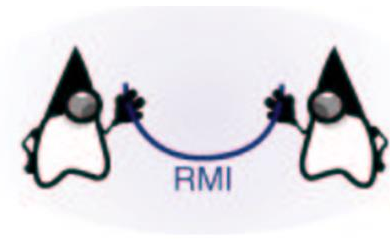
Le middleware java RMI

Le middleware Java RMI



- ▶ Java RMI est un middleware de type RPC/RMI intégré dans Java
- ▶ Le but de RMI est de créer un modèle objet distribué en java (spécifique au langage Java : ne fait communiquer que des éléments écrits en Java)
- ▶ RMI est apparu dès la version 1.1 du JDK
- ▶ Avec RMI, les méthodes de certains objets (appelés objets distants) peuvent être appelés depuis des JVM différentes (espace d'adressages distincts).
- ▶ RMI assure la communication entre le serveur et le client via TCP/IP et ce de manière transparente pour le développeur.
- ▶ Il utilise des mécanismes et des protocoles définis et standardisés tels que les sockets et RMP (Remote Method Protocol).
- ▶ Le développeur n'a pas à se soucier des problèmes de niveaux inférieurs spécifiques aux technologies réseaux.

Structure des couches RMI



Couche des Stubs et Skeletons

- ▶ Les Stubs sont des classes placées dans la machine du client.
- ▶ Lorsque le client fera appel à une méthode distante, cet appel sera transféré au Stub.
- ▶ Le Stub est le représentant local de l'objet distant.
- ▶ Il compacte (marshall) les arguments de la méthode distante et les envoie dans un flux de données vers le service RMI distant.
- ▶ D'autre part, il décompacte (unmarshall) la valeur ou l'objet de retour de la méthode distante.
- ▶ Il communique avec l'objet distant par l'intermédiaire d'un Skeleton.

Couche des Stubs et Skeletons

- ▶ Le Skeleton doit être placé sur la machine serveur.
- ▶ Il compacte (unmarshall) les paramètres de la méthode distante, les transmet à l'objet local et compacte les valeurs de retour à renvoyer au client.
- ▶ Les Stubs et les Skeletons sont donc des intermédiaires entre le client et le serveur qui gèrent le transfert distant des données.
- ▶ On utilise le compilateur **rmic** pour la génération des Stubs et des Skeletons avec le JDK (avant la version 1.2 de Java).
- ▶ On passe en paramètre à **rmic** le nom de la classe qui implémente l'interface.
 - ▶ Génère et compile automatiquement le fichier **ClassImpl_Stub.java**
 - ▶ Pas de manipulation directe dans notre code de ce fichier

Couche des Stubs et Skeletons

- ▶ Depuis la version 1.2 de java, **rmic** n'est plus utilisée:
 - ▶ le compilateur génère uniquement le stub
 - ▶ Le skeleton est le serveur lui-même et fonctionne directement avec l'implémentation des méthodes exportées : Lorsque votre objet est construit, le constructeur **UnicastRemoteObject** est appelé. Cela connecte l'objet à l'infrastructure interne RMI qui gère l'écoute des sockets et la répartition des méthodes à distance. En d'autres termes, il « exporte » l'objet.
- ▶ **RMIClassLoader** a pour fonction de charger le Stub et le Skeleton utilisés par le système RMI ainsi que d'autres classes utilitaires secondaires.
- ▶ Le Stub et le Skeleton sont générées lors de la compilation de l'implémentation et de l'interface.

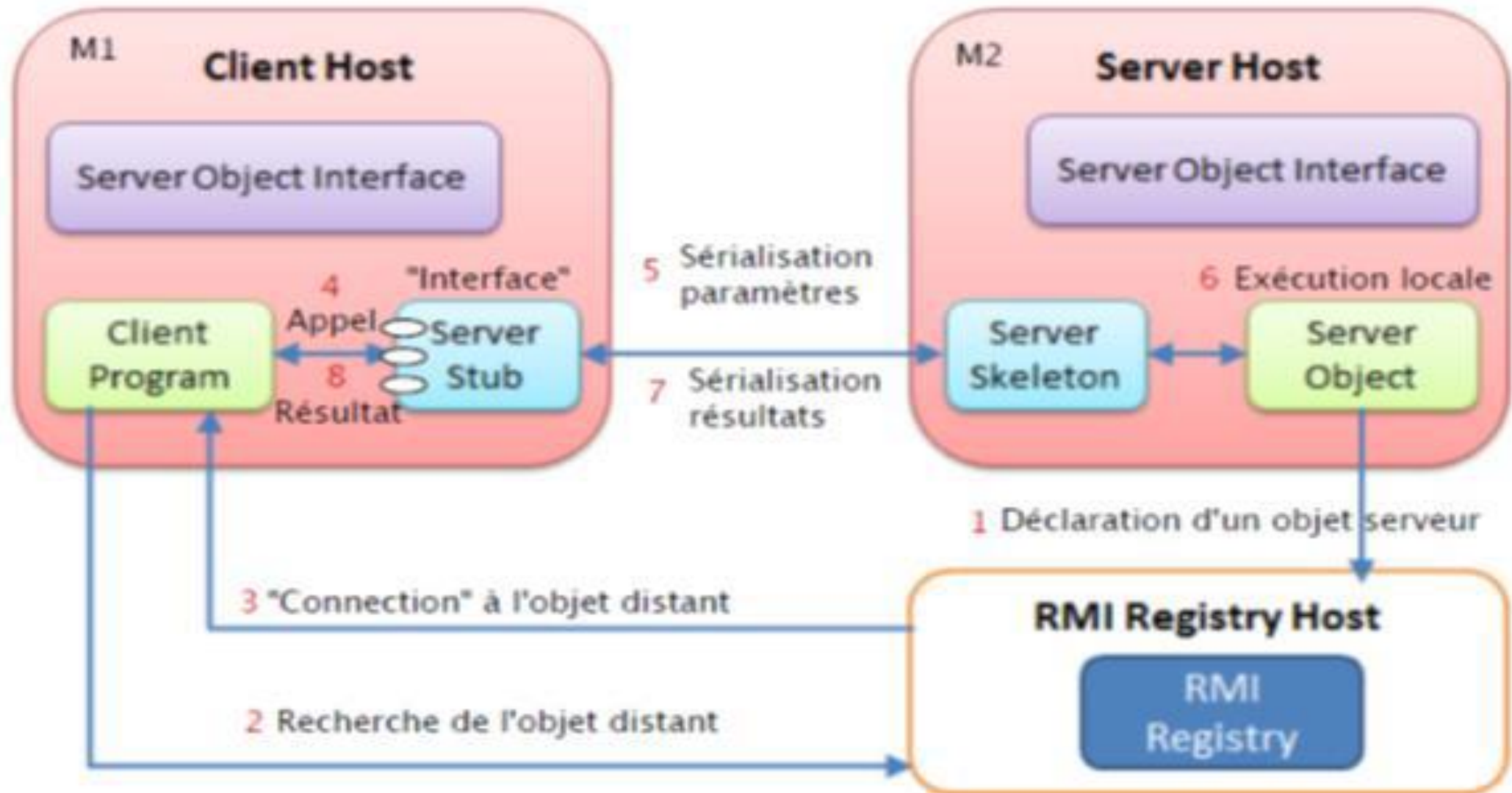
Couche de référence : Remote Reference Layer

- ▶ Permet d'obtenir une référence d'objet distribué à partir de la référence locale au stub.
- ▶ Cette fonction est assurée grâce à un service de noms **rmiregistry** (qui possède une table de hachage dont les clés sont des noms et les valeurs sont des objets distants).
- ▶ Un objet accède au registry via la classe **Naming**
- ▶ Identification d'un objet distant
 - ▶ Via une URL de la forme **rmi://hote:port/nomObj**
 - ▶ hote : nom de la machine distante (sur laquelle tourne un registry)
 - ▶ port : port sur lequel écoute le registry 1099
 - ▶ nomObj : nom donné à un objet offrant des opérations

Couche Transport

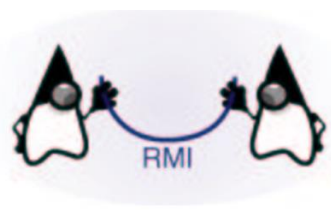
- ▶ La couche transport est basée sur les connexions TCP/IP entre les machines.
- ▶ Elle fournit la connectivité de base entre deux JVM.
- ▶ Elle construit une table des objets distants disponibles.
- ▶ Elle écoute et répond aux invocations.
- ▶ Cette couche utilise les classe ServerSocket et Socket.
- ▶ Elle utilise le protocole RMP Remote Method Protocol.

Fonctionnement du middleware Java RMI



Démarche de développement avec Java RMI

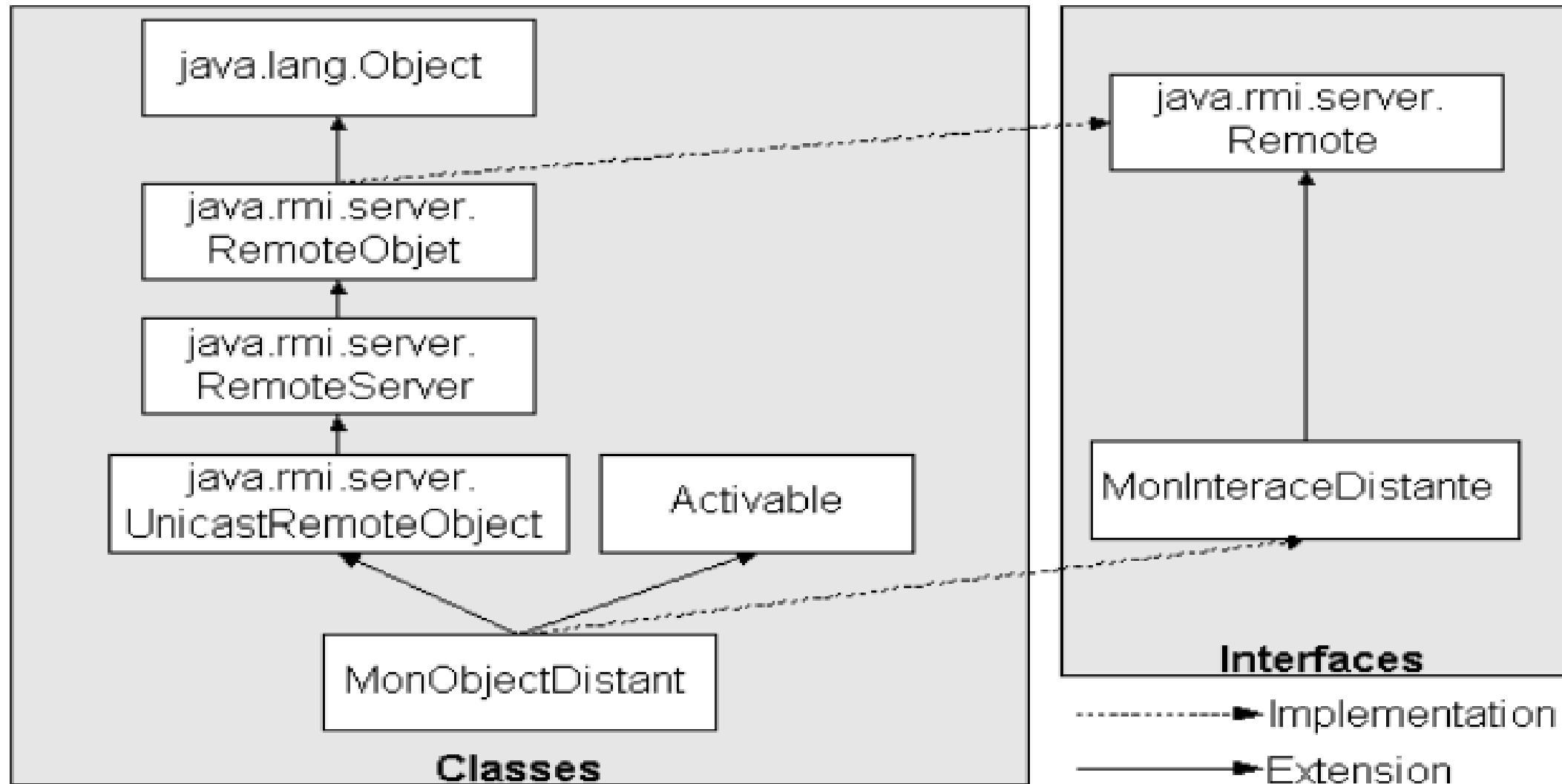
- ▶ Le développement coté serveur consiste à :
 - ▶ Définir de l'interface qui contient les méthodes distantes,
 - ▶ Créer la classe qui implémente cette interface,
 - ▶ Écrire la classe qui instancie l'objet distant et l'enregistre dans le registre de RMI (rmiregistry) en lui assignant un nom.
- ▶ Le développement coté client consiste à :
 - ▶ Obtenir la référence de l'objet distant à partir du registre en utilisant son nom.
 - ▶ Appeler les méthodes distantes en utilisant la référence de l'objet récupérée.



Le package `java.rmi`

- ▶ Java propose le package `java.rmi` et ses sous-packages qui contiennent les classes à utiliser ou spécialiser pour des communications via RMI.
- ▶ Règles principales à suivre:
 - ▶ Une **interface d'opérations** appelables à distance doit spécialiser l'interface `java.rmi.Remote`
 - ▶ Une opération de cette interface doit préciser dans sa signature qu'elle peut lever l'exception **RemoteException**
 - ➔ Le servant implémente une interface spécialisant **Remote** et doit spécialiser (ou utiliser des services de) **UnicastRemoteObject**
 - ▶ Les classes des données en paramètres ou retour des opérations doivent implémenter **Serializable**

Hiérarchie des classes dans RMI



Contraintes sur l'interface d'opérations

- ▶ L'interface spécialise **java.rmi.Remote** : précise qu'il s'agit d'une interface de service appelables à distance.
 - ▶ Chaque opération doit préciser dans sa signature qu'elle peut lever l'exception **RemoteException** (L'opération peut également lever des exceptions spécifiques à l'application)
 - ▶ **RemoteException** est la classe mère d'une hiérarchie d'une vingtaine d'exceptions précisant un problème lors de l'appel de l'opération, comme, par exemple:
 - ▶ **NoSuchObjectException** : ne peut pas récupérer la référence sur l'objet distant
 - ▶ **StubNotFoundException** : un stub correct n'a pas été trouvé
 - ▶ **UnknownHostException** : la machine distante n'a pas été trouvée
 - ▶ **AlreadyBoundException** : le nom associé à l'objet est déjà utilisé
 - ▶ **ConnectException** : connexion refusée par la partie serveur
 - ▶ Pour plus d'informations sur l'exception : opération **getCause()** de **RemoteException** peut
-
- ▶ être appelée.

Exemple d'interface

- ▶ Exemple d'interface, définissant 2 opérations

```
public interface IRectangle extends Remote {  
  
    public int calculSurface(Rectangle rect)  
        throws RemoteException;  
  
    public Rectangle decalerRectangle(Rectangle rect,  
        int x, int y) throws RemoteException;  
}
```

- ▶ Uniquement la signature des opérations est déclarée, avec type de retour et paramètres de types primitifs ou de n'importe quelle classe.
- ▶ Lors de l'implémentation des opérations de l'interface, on ne traite pas directement les cas d'erreurs correspondant à la levée d'une exception **RemoteException** (c'est l'affaire du middleware)

Exemple d'une classe implémentant l'interface

```
public class RectangleImpl implements Irectangle {  
  
    public int calculSurface(Rectangle rect)  
        throws RemoteException {  
        return ((rect.x2 - rect.x1)*(rect.y2 - rect.y1));  
    }  
  
    public Rectangle decalerRectangle(Rectangle rect,  
        int x, int y) throws RemoteException {  
        return new Rectangle(rect.x1 + x, rect.y1 + y,  
            rect.x2 + x, rect.y2 + y);  
    }  
  
    public RectangleImpl() throws RemoteException {  
        UnicastRemoteObject.exportObject(this);  
    }  
}
```

- ▶ Les opérations de l'interface sont « standards », rien de particulier à RMI (à part l'exception **RemoteException** dans la signature)
- ▶ La classe peut implémenter d'autres opérations mais seules celles d'interfaces étendant Remote sont appelables à distance.

Contraintes sur la classe implémentant l'interface

- ▶ La classe implémentant l'interface doit utiliser la classe **UnicastRemoteObject**. Cette classe sert à la communication via TCP :
 - ▶ L'objet de la classe implémentant l'interface doit « s'exporter » pour accepter des connexions des clients.
 - ▶ Création et utilisation des ressources, éléments nécessaires à la communication via sockets TCP.
 - ▶ **Unicast** : l'objet de la classe implémentant l'interface n'existe qu'un en seul exemplaire sur une seule machine.
 - ▶ Deux façons d'utilisation:
 - ▶ La classe étend directement la classe **UnicastRemoteObject**
 - ▶ Pas de spécialisation et le constructeur doit exécuter l'instruction suivante :
UnicastRemoteObject.exportObject(this);
 - ▶ Le constructeur par défaut doit préciser dans sa signature qu'il peut lever l'exception
-

▶ **RemoteException.**

Implémentation du servent

- ▶ Côté serveur :
 - ▶ instantiation d'un RectangleImpl et
 - ▶ enregistrement sur le registry

```
try {  
    // instantiation « standard » de l'objet  
    RectangleImpl rect = new RectangleImpl();  
  
    // enregistrement de l'objet sous le nom « opRect » sur le registry local  
    Naming.rebind("opRect", rect);  
}  
catch (Exception e) { ... }
```

- ▶ Avant d'exécuter ce code : lancer rmiregistry

Lancement du registre

- ▶ Lancement du `rmiregistry`
 - ▶ Unix/Linux : **\$ `rmiregistry` [port]**
 - ▶ Windows : **> `start rmiregistry` [port]**
- ▶ En paramètre, on peut préciser un numéro de port : port d'écoute du registry (Par défaut : 1099)
- ▶ Variante de lancement : un objet Java peut à l'exécution lancer un registry :
Opération de **`java.rmi.registry.LocateRegistry`**
`public static Registry createRegistry(int port) throws RemoteException`
- ▶ Lance un registry localement sur le port d'écoute passé en paramètre

Le registre rmiregistry

- ▶ Java propose la classe **java.rmi.Naming** qui offre 5 opérations statiques pour enregistrer des objets et récupérer leurs références (lèvent toutes plusieurs exceptions)
 - ▶ **void bind(String name, Remote obj)** : enregistre un objet sous le nom name (erreur si déjà un objet avec ce nom)
 - ▶ **void rebind(String name, Remote obj)** : enregistre un objet sous le nom name, en écrasant la précédente liaison objet/nom si elle existait
 - ▶ **void unbind(String name)** : supprime du registry la référence vers l'objet nommé name
 - ▶ **String[] list(String name)** : retourne l'ensemble des noms des objets enregistrés sur le registry
 - ▶ **Remote lookup(String name)** : retourne une référence sur l'objet dont le nom est passé en paramètre , Exception `NotBoundException` levée si objet pas trouvé par le registry

→ **Attribut name dans les 5 opérations : sous la forme URL de type RMI**

Classe représentant les données « Rectangle »

- La classe Rectangle permet de décrire les coordonnées d'un rectangle, cette classe doit implémenter l'interface « serializable »:

```
public class Rectangle implements Serializable {  
  
    public int x1, x2, y1, y2;  
  
    public Rectangle(int x1, int y1, int x2, int y2) {  
        this.x1 = x1;  
        this.y1 = y1;  
        this.x2 = x2;  
        this.y2 = y2;  
    }  
  
    public String toString() {  
        return "(" + x1 + ", " + y1 + ") (" + x2 + ", " + y2 + ") ";  
    }  
}
```

Implémentation du client

- ▶ Code client pour appeler les opérations distantes:

```
Rectangle r1, r2;  
r1 = new Rectangle(10, 10, 20, 30);
```

// on demande au registry tournant sur la machine scinfr222 (port par //défaut) de nous renvoyer la réf de l'objet nommé « opRect »

```
IRectangle opRectangle = (IRectangle)  
    Naming.lookup("rmi://scinfr222/opRect");
```

// une fois la référence obtenue, on appelle normalement les opérations de l'interface sur cet objet (en catchant RemoteException)

Implémentation du client

```
int surface = opRectangle.calculSurface(r1);  
r2 = opRectangle.decalerRectangle(r1, 15, 10);  
  
System.out.println(" surface de r1 = "+surface);  
System.out.println(" position de r2 = "+r2);
```

- ▶ Résultat du lookup : Si objet trouvé par le registry distant, retourne un objet implémentant l'interface générique Remote.
- ▶ Dans notre cas, lookup retourne un objet implémentant **Irectangle** comme par exemple **RectangleImpl** instancié coté serveur.

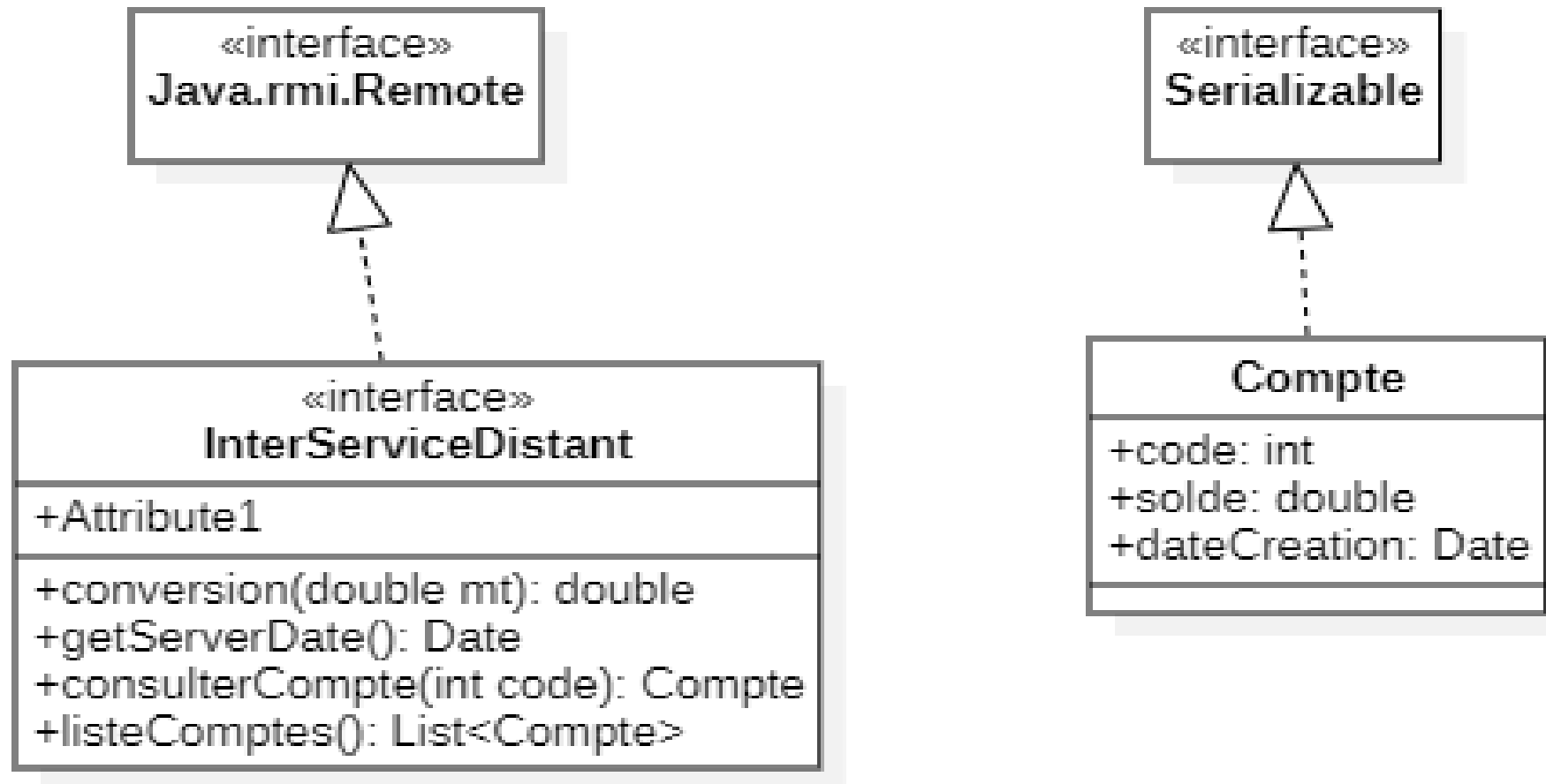
Exemple 2

- ▶ On suppose qu'on veut créer un serveur RMI qui crée un objet qui offre les services distants suivant à un client RMI:
 - ▶ Convertir un montant de l'EURO en Dinar Tunisien.
 - ▶ Envoyer au client la date du serveur
 - ▶ Consulter un compte
 - ▶ Consulter une liste de comptes

Création de l'interface de l'objet distant

- ▶ La première étape consiste à créer une interface distante qui décrit les méthodes que le client pourra invoquer à distance.
- ▶ Pour que ses méthodes soient accessibles par le client, cette interface doit hériter de l'interface **Remote**.
- ▶ Toutes les méthodes utilisables doivent pouvoir lever les exceptions de type **RemoteException** qui sont spécifiques à l'appel distant.
- ▶ Cette interface devra être placée sur les deux machines (serveur et client).
- ▶ Les opérations de l'interface utilisent l'objet métier « **Compte** », donc on doit créer une classe **Compte**.

Création de l'interface de l'objet distant



Création de l'interface de l'objet distant

```
package service;

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Date;
import java.util.List;

import metier.Compte;

public interface IBanqueRemote extends Remote{
    public double conversion (double mt) throws RemoteException;
    public Date getServerDate() throws RemoteException;
    public Compte consulterCompte(int code) throws RemoteException;
    public List<Compte> listeComptes() throws RemoteException;
}
```

Création de la classe Compte

```
package metier;
import java.io.Serializable;
import java.util.Date;
public class Compte implements
Serializable{
    private int code;
    public int getCode() {
        return code;
    }
    public void setCode(int code) {
        this.code = code;
    }
}
```

```
    private double solde;
    public double getSolde() {
        return solde;
    }
    public void setSolde(double solde) {
        this.solde = solde;
    }
    private Date dateCreation;
    public Date getDateCreation() {
        return dateCreation;
    }
    public void setDateCreation(Date dateCreation) {
        this.dateCreation = dateCreation;
    }
    public Compte() {
        super();
    }
}
```

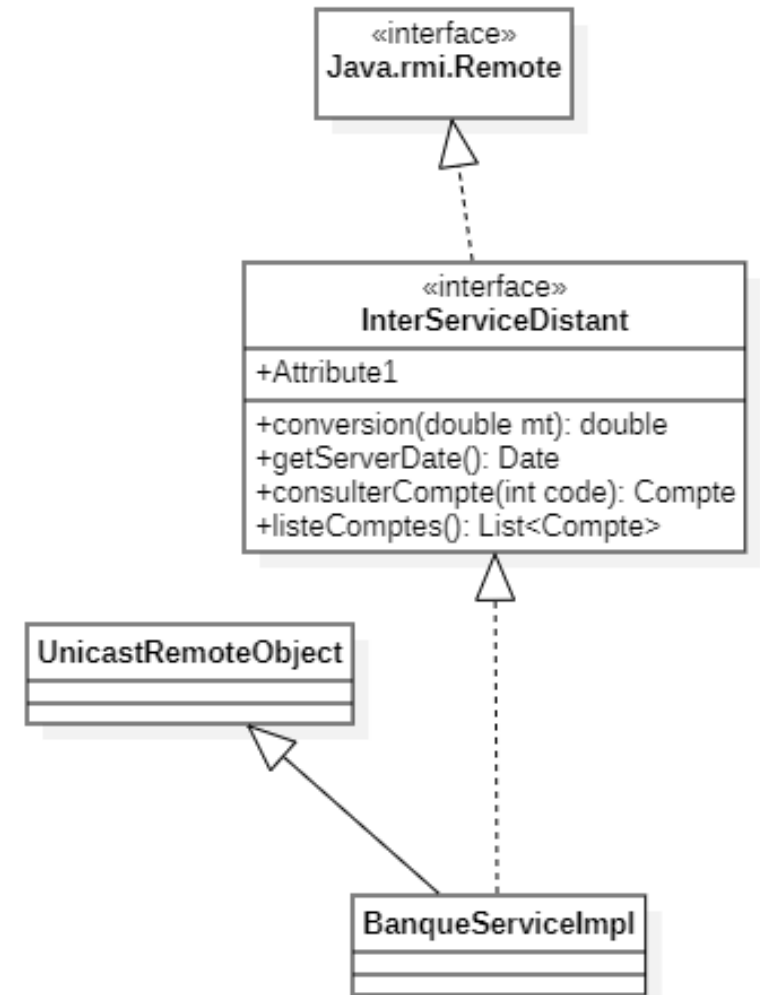
Implémentation de l'interface dans une classe

- ▶ Il faut maintenant implémenter l'interface dans une classe. Par convention le nom de cette classe aura pour suffixe **Impl**.
- ▶ Cette classe doit hériter de la classe **java.rmi.server.RemoteObject** ou de l'une de ses sous classes.
- ▶ La plus facile à utiliser est **java.rmi.server.Unicast.RemoteObject**
- ▶ C'est dans cette classe qu'on nous allons définir le corps des méthodes distantes que pourront utiliser les clients.

Implémentation de l'objet distant

```
ClientRMI.java  *BanqueServiceImpl.java  IBanqueRemote.java  Compte.java

1 package service;
2
3 import java.rmi.RemoteException;
10
11 public class BanqueServiceImpl extends UnicastRemoteObject implements IBanqueRemote {
12
13     protected BanqueServiceImpl() throws RemoteException {
14         super();
15     }
16
17     @Override
18     public double conversion(double mt) throws RemoteException {
19         // TODO Auto-generated method stub
20         return mt*3.5;
21     }
22
23     @Override
24     public Date getServerDate() throws RemoteException {
25         return new Date();
26     }
27
28     @Override
29     public Compte consulerCompte(int code) throws RemoteException {
30         Compte cp=new Compte (1, Math.random()*9000, new Date());
31         return cp;
32     }
33
34     @Override
35     public List<Compte> listeComptes() throws RemoteException {
36         List<Compte> cptes = new ArrayList<Compte>();
37         cptes.add(new Compte (1, Math.random()*9000, new Date()));
38         cptes.add(new Compte (2, Math.random()*9000, new Date()));
39         cptes.add(new Compte (3, Math.random()*9000, new Date()));
40         return cptes;
41     }
42 }
```

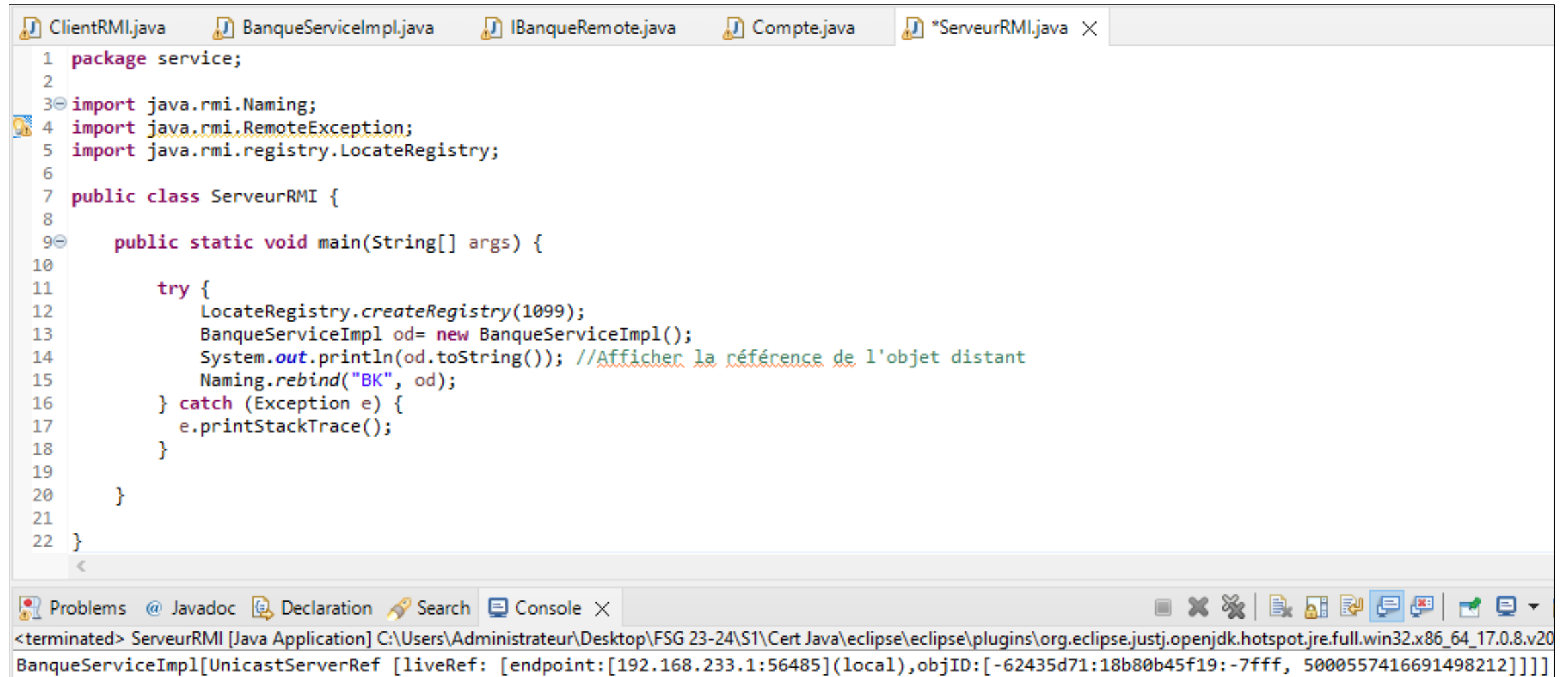


Implémentation du serveur

- ▶ Le serveur doit enregistrer auprès du registre RMI l'objet local dont les méthodes seront disponibles à distance.
- ▶ Cela se fait à l'aide de la méthode statique `bind()` ou `rebind()` de la classe `Naming`.
- ▶ Cette méthode permet d'associer (enregistrer) l'objet local avec un synonyme dans le registre RMI.
- ▶ L'objet devient alors disponible par le client.

```
ObjetDistantImpl od = new ObjetDistantImpl();  
Naming.rebind("rmi://localhost:1099/NOM_Service",od);
```

Implémentation du serveur



```
1 package service;
2
3 import java.rmi.Naming;
4 import java.rmi.RemoteException;
5 import java.rmi.registry.LocateRegistry;
6
7 public class ServeurRMI {
8
9     public static void main(String[] args) {
10
11         try {
12             LocateRegistry.createRegistry(1099);
13             BanqueServiceImpl od= new BanqueServiceImpl();
14             System.out.println(od.toString()); //Afficher la référence de l'objet distant
15             Naming.rebind("BK", od);
16         } catch (Exception e) {
17             e.printStackTrace();
18         }
19
20     }
21
22 }
```

<terminated> ServeurRMI [Java Application] C:\Users\Administrateur\Desktop\FSG 23-24\S1\Cert Java\eclipse\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.8.v20
BanqueServiceImpl[UnicastServerRef [liveRef: [endpoint:[192.168.233.1:56485](local),objID:[-62435d71:18b80b45f19:-7fff, 5000557416691498212]]]]

Naming service

- ▶ Les clients trouvent les services distants en utilisant un service d'annuaire activé sur un hôte connu avec un numéro de port connu.
- ▶ RMI peut utiliser plusieurs services d'annuaire, y compris Java Naming and Directory Interface (JNDI).
- ▶ JNDI inclut un service simple appelé rmiregistry.
- ▶ Le registre est exécuté sur chaque machine qui héberge des objets distants (les serveurs) et accepte les requêtes pour ces services, par défaut sur le port 1099.

Client

- ▶ Le client peut obtenir une référence à un objet distant par l'utilisation de la méthode statique **lookup()** de la classe **Naming**.
- ▶ La méthode **lookup()** sert au client pour interroger un registre et récupérer un objet distant.
- ▶ Elle retourne une référence à l'objet distant.
- ▶ La valeur retournée est du type **Remote**. Il est donc nécessaire de caster cet objet en l'interface distante implémentée par l'objet distant.
- ▶ **Remarque:** le projet Client doit contenir les classes métiers crée au niveau du serveur → A part l'interface, la classe Compte doit exister dans le client.

Implémentation du client

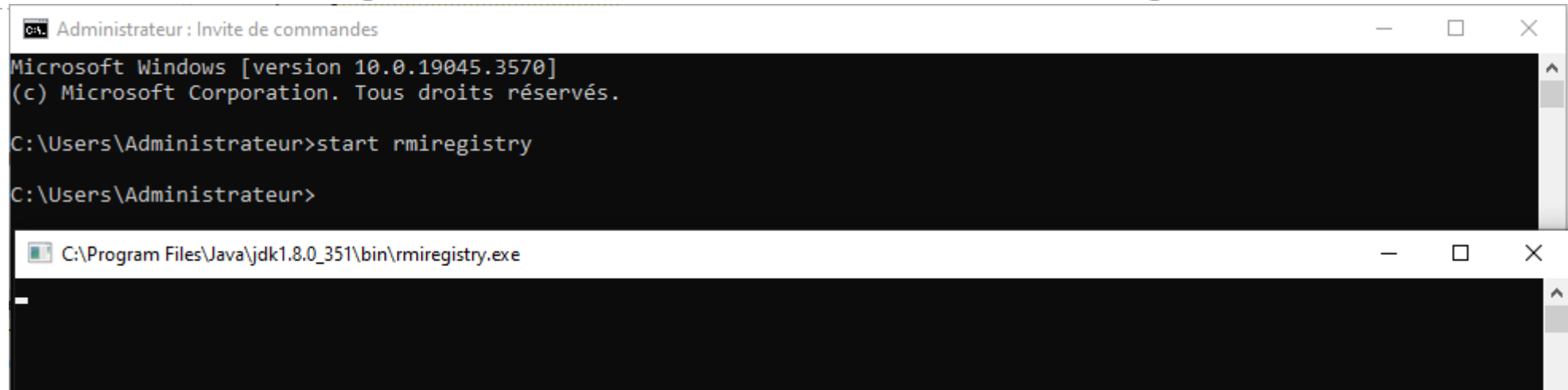
```
*ClientRMI.java X BanqueServiceImpl.java IBanqueRemote.java Compte.java ServeurRM
1 package service;
2 import java.net.MalformedURLException;
3 import java.rmi.Naming;
4 import java.rmi.NotBoundException;
5 import java.rmi.RemoteException;
6 import java.util.List;
7 import metier.Compte;
8 public class ClientRMI {
9
10 public static void main(String[] args) {
11     try {
12         IBanqueRemote stub = (IBanqueRemote) Naming.lookup("rmi://localhost:1099/BK");
13         //System.out.println(stub.conversion(70));
14         //System.out.println(stub.getServerDate());
15
16         System.out.println("Consultation d'un compte à distance");
17         Compte cp=stub.consulterCompte(1);
18         System.out.println("code = "+cp.getCode());
19         System.out.println("Solde = "+cp.getSolde());
20         System.out.println("Date Création= "+cp.getDateCreation());
21
22         System.out.println("Consultation de la liste des comptes");
23         List<Compte> cptes=stub.listeComptes();
24         for (Compte c:cptes) {
25             System.out.println("-----");
26             System.out.println("code = "+c.getCode());
27             System.out.println("code = "+c.getSolde());
28             System.out.println("code = "+c.getDateCreation());
29         }
30     } catch (Exception e) {
31         // TODO Auto-generated catch block
32         e.printStackTrace();
33     }
34 }}
```

```
Problems @ Javadoc Declaration Search Console X
<terminated> ClientRMI [Java Application] C:\Users\Administrateur\Desktop
Consultation d'un compte à distance
code = 1
Solde = 3879.225674929552
Date Création= Mon Oct 30 14:49:24 WAT 2023
Consultation de la liste des comptes
-----
code = 1
code = 5113.1740726169
code = Mon Oct 30 14:49:24 WAT 2023
-----
code = 2
code = 7792.189225593099
code = Mon Oct 30 14:49:24 WAT 2023
-----
code = 3
code = 8203.780775045936
code = Mon Oct 30 14:49:24 WAT 2023
```

Lancement

- ▶ Il est maintenant possible de lancer l'application. Cela va nécessiter l'utilisation de 3 consoles:
 - ▶ La première sera utilisée pour activer le registre. Pour cela, vous devez exécuter l'utilitaire `rmiregistry`
 - ▶ Dans une deuxième console, exécutez le serveur. Celui-ci va charger l'implémentation en mémoire, enregistrer cette référence dans le registre et attendre une connexion cliente.
 - ▶ Vous pouvez enfin exécuter le client dans une 3ème console.
- ▶ Même si vous exécutez le client et le serveur sur la même machine, RMI utilisera la pile réseau et TCP/IP pour communiquer entre les JVM.

Lancement du registre et du serveur avec la ligne de commande



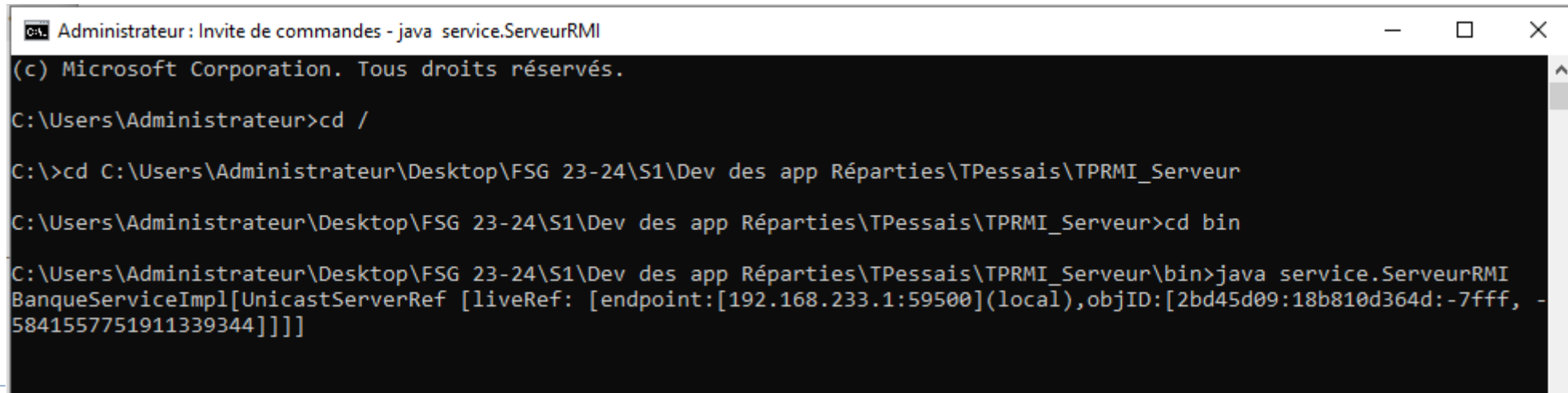
The screenshot shows two overlapping windows. The top window is a Windows command prompt titled 'Administrateur : Invite de commandes'. It displays the Windows version (10.0.19045.3570) and the user's location (C:\Users\Administrateur). The command 'start rmiregistry' has been entered, and the prompt is now 'C:\Users\Administrateur>'. Below it, a smaller window titled 'C:\Program Files\Java\jdk1.8.0_351\bin\rmiregistry.exe' is open, showing a blank black screen.

```
Administrateur : Invite de commandes
Microsoft Windows [version 10.0.19045.3570]
(c) Microsoft Corporation. Tous droits réservés.

C:\Users\Administrateur>start rmiregistry

C:\Users\Administrateur>
```

C:\Program Files\Java\jdk1.8.0_351\bin\rmiregistry.exe



The screenshot shows a Windows command prompt titled 'Administrateur : Invite de commandes - java service.ServeurRMI'. It shows the user navigating to the directory 'C:\Users\Administrateur\Desktop\FSG 23-24\S1\Dev des app Réparties\TPessais\TPRMI_Serveur\bin' and then running the command 'java service.ServeurRMI'. The output shows the server starting and listening on port 59500.

```
Administrateur : Invite de commandes - java service.ServeurRMI
(c) Microsoft Corporation. Tous droits réservés.

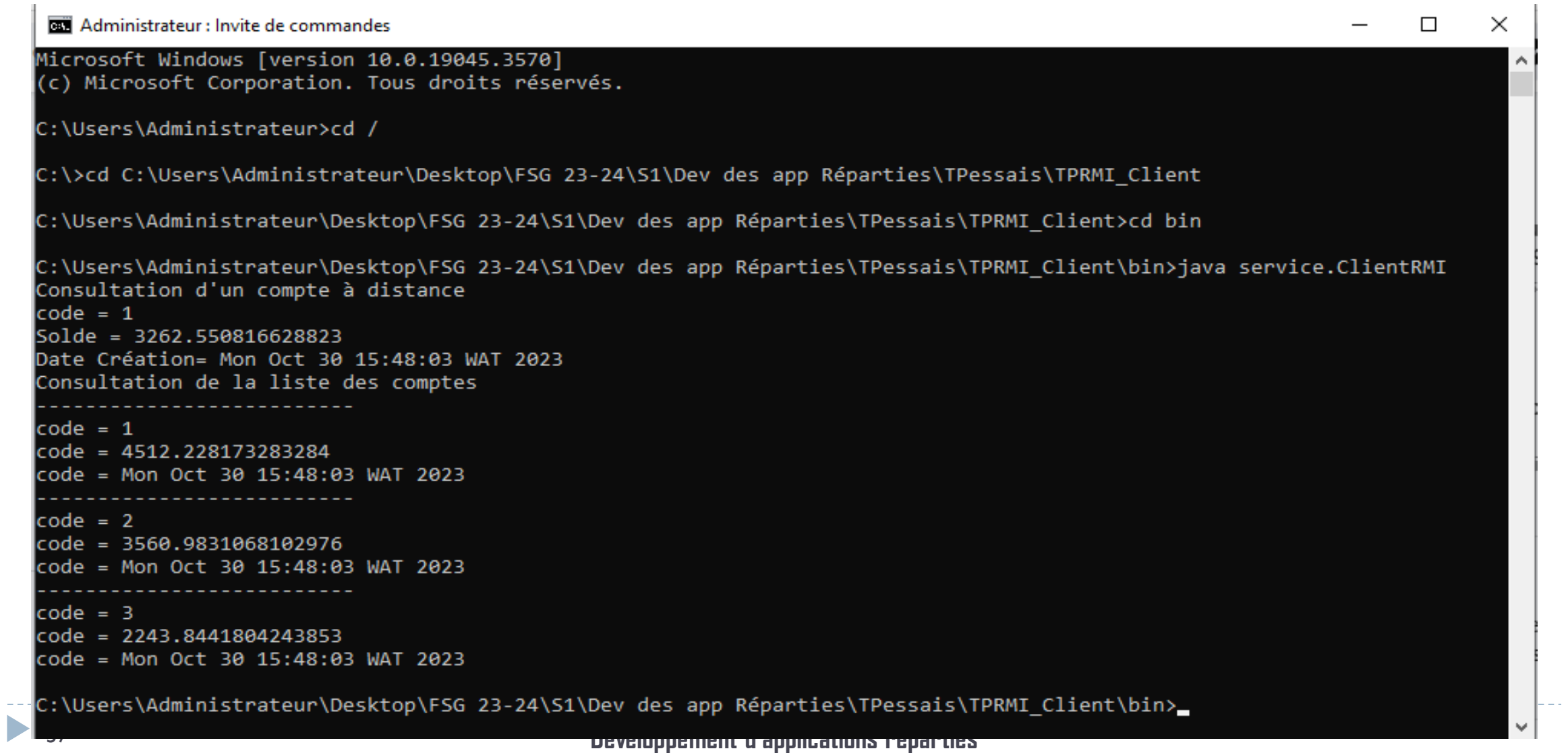
C:\Users\Administrateur>cd /

C:\>cd C:\Users\Administrateur\Desktop\FSG 23-24\S1\Dev des app Réparties\TPessais\TPRMI_Serveur

C:\Users\Administrateur\Desktop\FSG 23-24\S1\Dev des app Réparties\TPessais\TPRMI_Serveur>cd bin

C:\Users\Administrateur\Desktop\FSG 23-24\S1\Dev des app Réparties\TPessais\TPRMI_Serveur\bin>java service.ServeurRMI
BanqueServiceImpl[UnicastServerRef [liveRef: [endpoint:[192.168.233.1:59500](local),objID:[2bd45d09:18b810d364d:-7fff, -5841557751911339344]]]]
```

Lancement du client avec la ligne de commande



```
Administrateur : Invite de commandes
Microsoft Windows [version 10.0.19045.3570]
(c) Microsoft Corporation. Tous droits réservés.

C:\Users\Administrateur>cd /

C:\>cd C:\Users\Administrateur\Desktop\FSG 23-24\S1\Dev des app Réparties\TPessais\TPRMI_Client

C:\Users\Administrateur\Desktop\FSG 23-24\S1\Dev des app Réparties\TPessais\TPRMI_Client>cd bin

C:\Users\Administrateur\Desktop\FSG 23-24\S1\Dev des app Réparties\TPessais\TPRMI_Client\bin>java service.ClientRMI
Consultation d'un compte à distance
code = 1
Solde = 3262.550816628823
Date Création= Mon Oct 30 15:48:03 WAT 2023
Consultation de la liste des comptes
-----
code = 1
code = 4512.228173283284
code = Mon Oct 30 15:48:03 WAT 2023
-----
code = 2
code = 3560.9831068102976
code = Mon Oct 30 15:48:03 WAT 2023
-----
code = 3
code = 2243.8441804243853
code = Mon Oct 30 15:48:03 WAT 2023
-----
C:\Users\Administrateur\Desktop\FSG 23-24\S1\Dev des app Réparties\TPessais\TPRMI_Client\bin>
```

57

Developpement d'applications Réparties