



Client - Serveur et Interopérabilité

Anne Marie Dery

**Object-
oriented
Design**

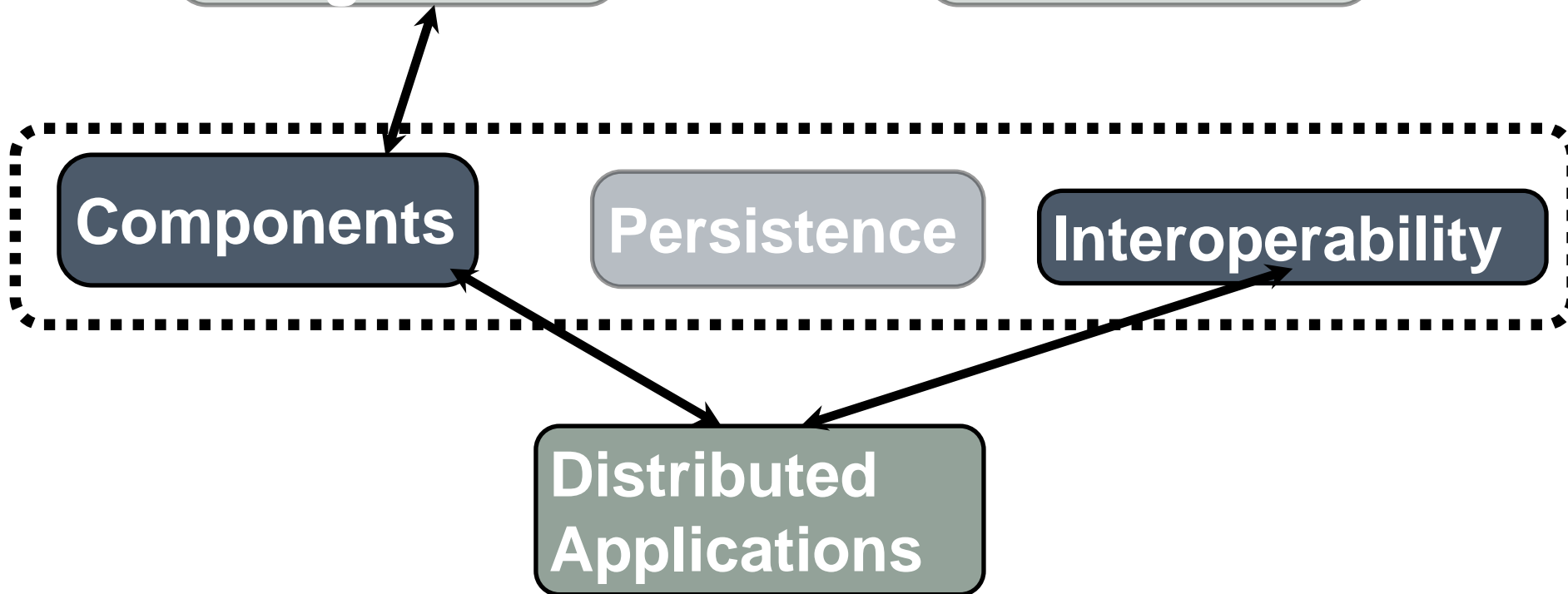
Databases

Components

Persistence

Interoperability

**Distributed
Applications**



Voir ou revoir des concepts de base

- Protocoles de transports
- Protocoles d'application
- Distinguer:
 1. transport des données
 2. sérialisation des données
 3. définition et gestion des services
- Comprendre comment on définit des services
- Distinguer couplage fort et couplage faible
 - RMI, Corba, Web Services, Composants J2E...

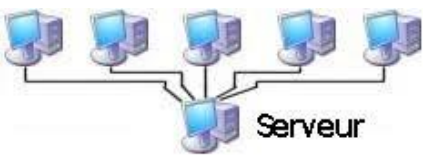
Client - Serveur



1/ Protocole d'application

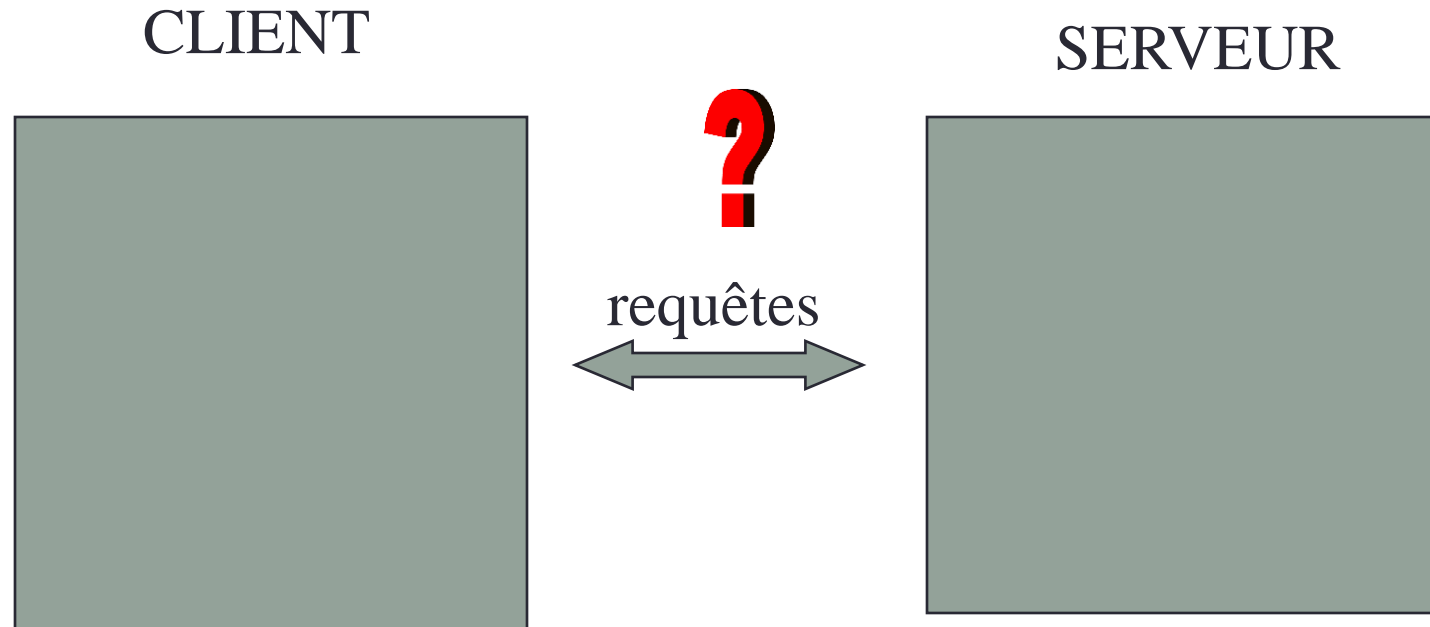


2/ Marshalling Unmarshalling



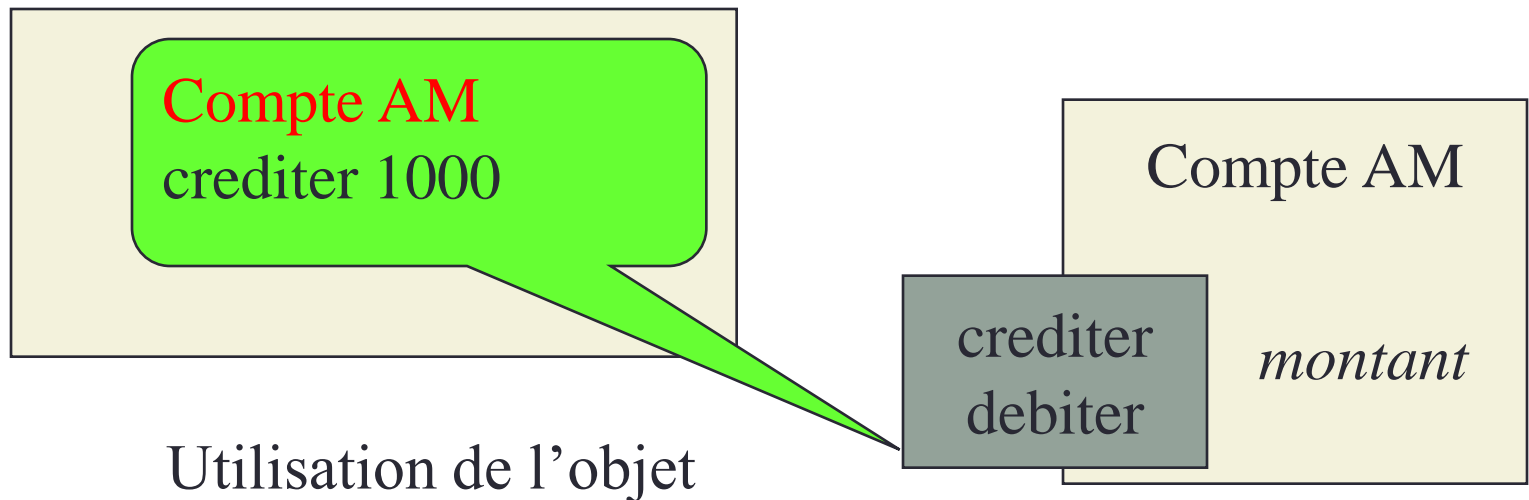
3/ Communication réseau

Requêtes Client Serveur : appeler les services proposés par le serveur



Ensemble des requêtes = protocole d'application du service

Exposer le protocole d'application : liste des services offerts par un serveur



Equivalent à la javadoc d'une classe Java

Interface = partie visible de l'objet

Implémentation = partie privée inaccessible depuis d'autres objets

Modifier l'implémentation des classes sans altérer l'utilisation

Interface = contrat entre l'objet et le monde extérieur

Protocole SMTP, RFC1822/3



```
HELO
MAIL From: pinna@essi.fr
RCPT To: pinna@essi.fr
DATA
From: pinna@essi.fr
Subject: Qui est là ?\n");
"Vous suivez toujours ?
QUIT
```

Approche réseau : Questions préliminaires

- ⦿ **Protocoles** de transport TCP et UDP ?
- ⦿ Utilisation des **adresses Internet** ?
- ⦿ Utilisation des **ports** ?
- ⦿ Programmation **sockets** : gestion d'entrées/sorties

Client : ?

Serveur : ?

Serveur de noms ?
(DNS, LDAP) ?

Un peu de vocabulaire

Client : entité qui fait l'appel

Sockets : moyen de communication entre ordinateurs

Adresses IP : adresse d'un ordinateur

Serveur : entité qui prend en charge la requête

Serveur de noms (DNS) : correspondances entre noms logiques et adresses IP (**Annuaire**)

Port : canal dédié à un service

Protocole : langage utilisé par 2 ordinateurs pour communiquer entre eux

protocole de transport : comment véhiculer les données – construction de la trame réseau

protocole d'application : comment le client et le serveur structurent les données échangées



Architecture client serveur

Un hôte établit une communication avec un autre hôte qui fournit un **service**



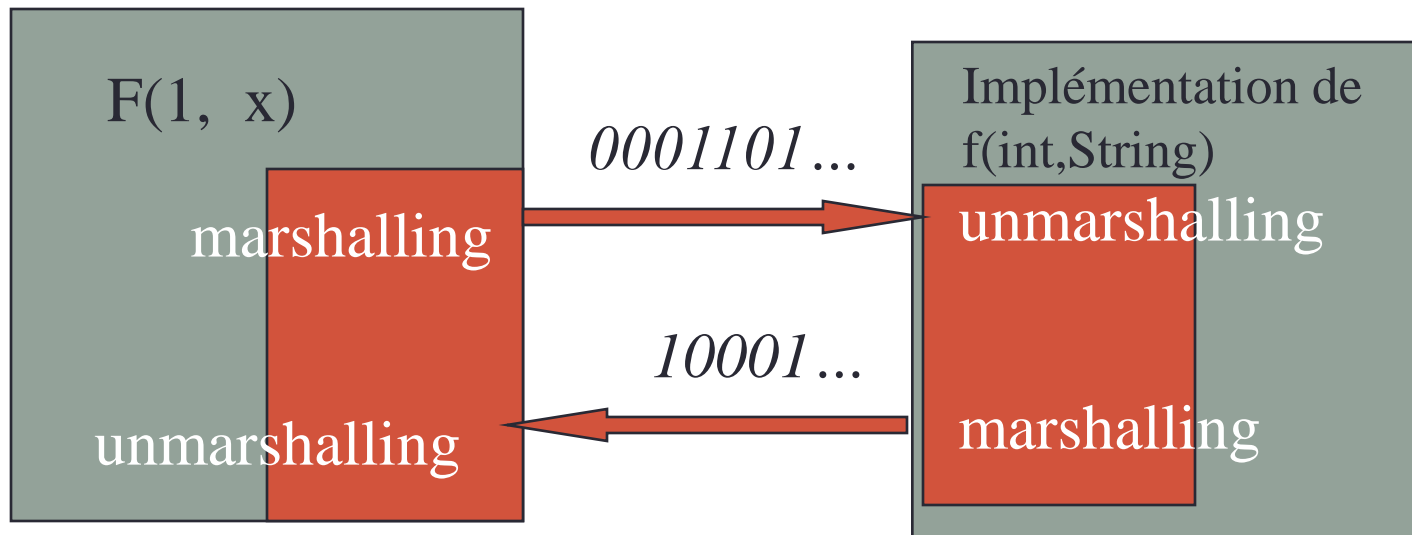
Appel à Distance

Marshalling : Préparer le format et le contenu de la trame réseau

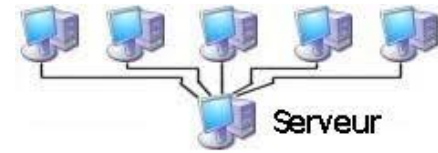
Unmarshalling : Reconstruire l'information échangée à partir de la trame réseau

CLIENT

SERVEUR

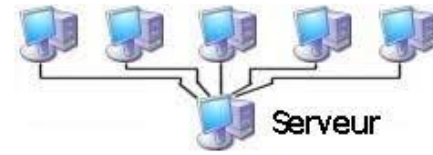


Comment cela fonctionne au niveau du réseau ?



- Identification de la machine qui abrite le serveur par le client
- Identification du serveur sur la machine
- Canal de communication entre le serveur et le client
- **Construction de la trame réseau**
- Echange **du protocole d'application au dessus d'un protocole de transport**

Programmation Socket : échanges sur le réseau



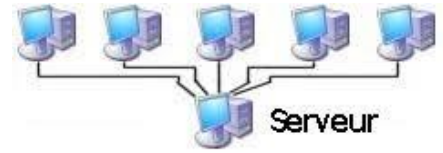
Les applications client/server communiquent via des sockets

- ◎ Deux types de transports via les socket API:
 - › Datagramme (non fiable)
 - › Orienté flux d'octets (fiable)

socket

Une porte à travers laquelle l'application peut à la fois envoyer et recevoir des messages d'une autre application

Exemple socket Java



Communication par flot
de données TCP

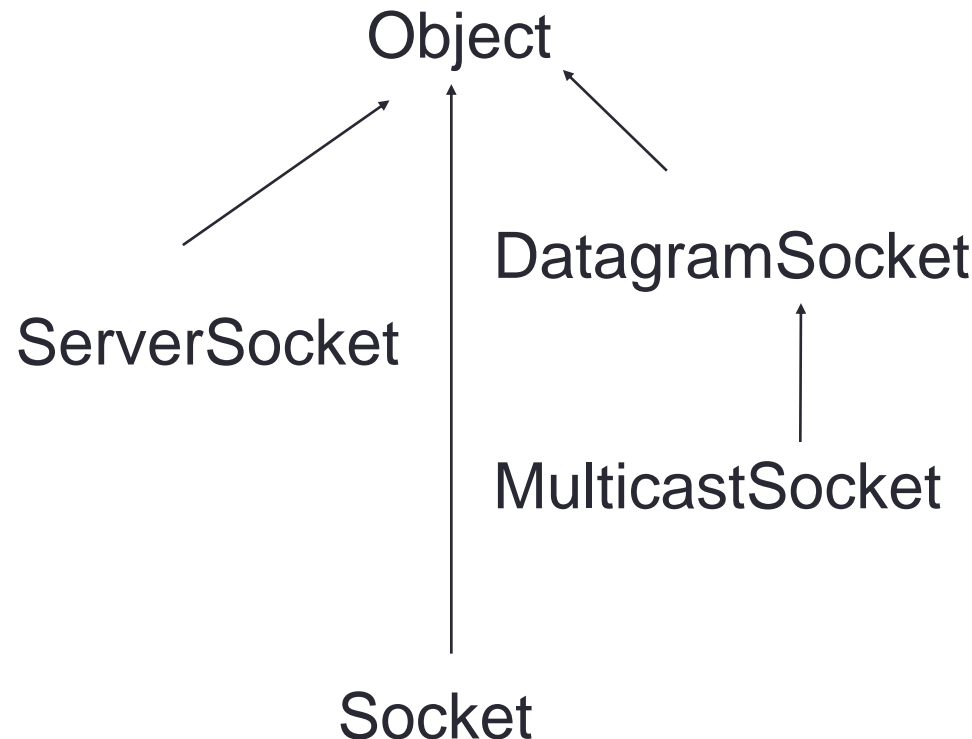
- fortement connectée
- synchrone
- type client-serveur

Communication par
messages UDP

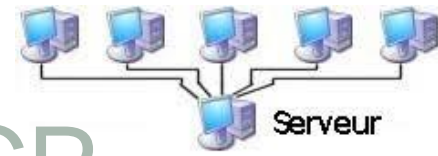
- en mode datagramme
- en mode déconnecté

Communication réseau
par diffusion UDP

- En Java : sockets
dans le package **java.net**



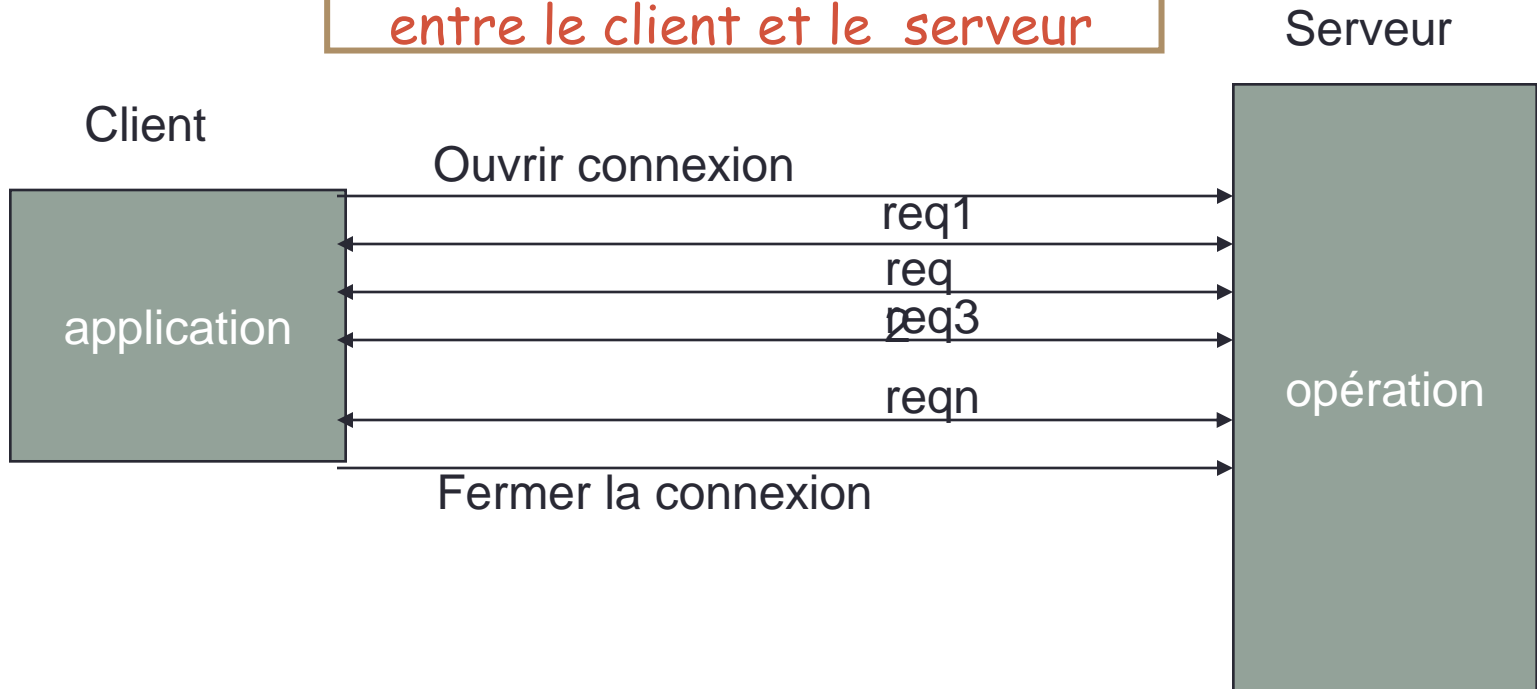
Flot de requêtes du client vers le serveur avec des Sockets TCP



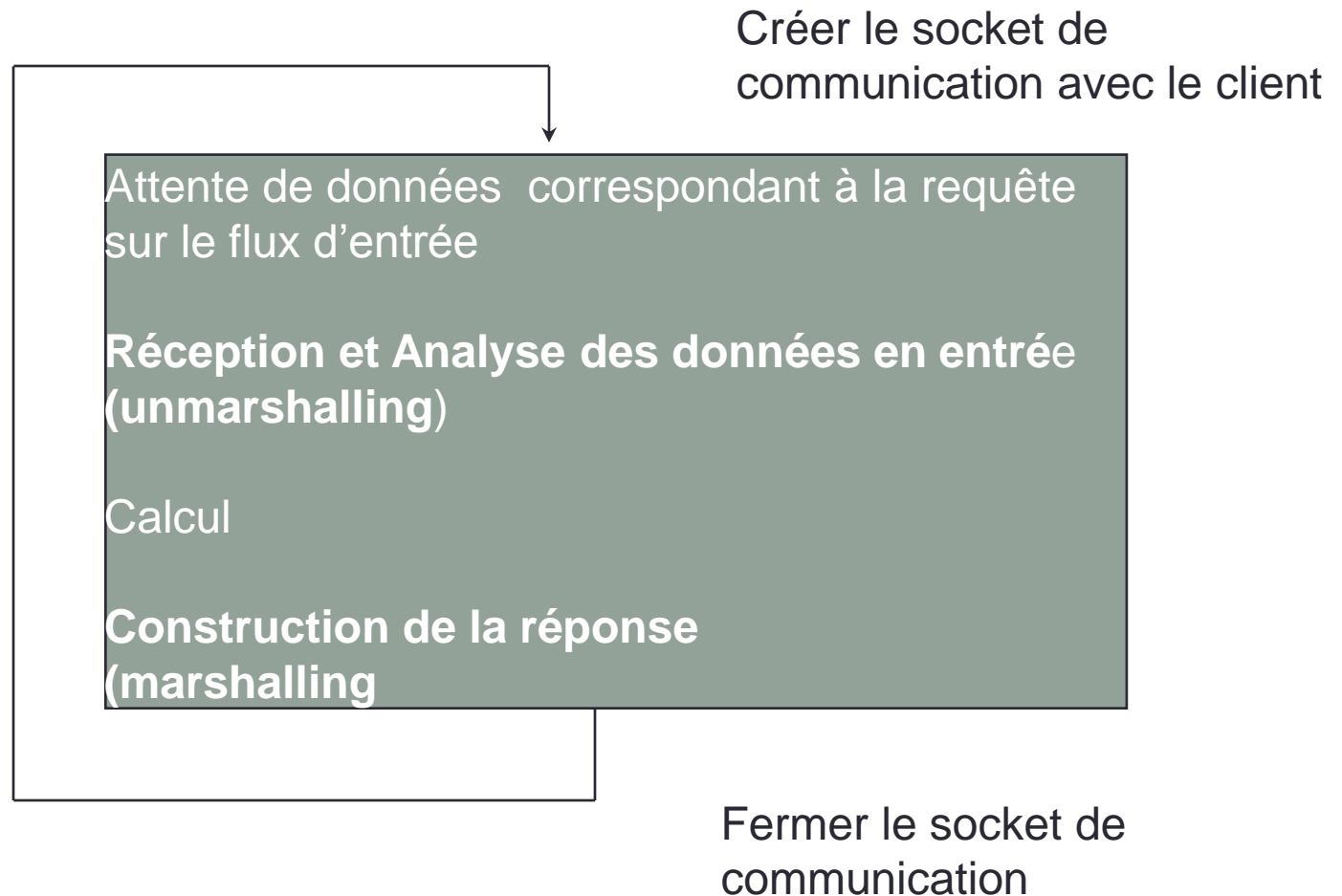
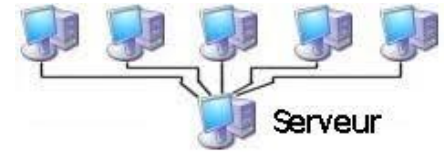
Point de vue application

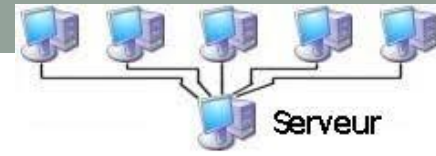
TCP fournit un transfert fiable,
conservant l'ordre de transfert
des octets ("pipe")
entre le client et le serveur

SESSION



Scénario d'un serveur





Scénario d'un client

Créer le socket de connexion
avec le serveur
Attendre que la connexion
soit établie
Récupérer la socket de
communication

Préparer la requête (marshalling)

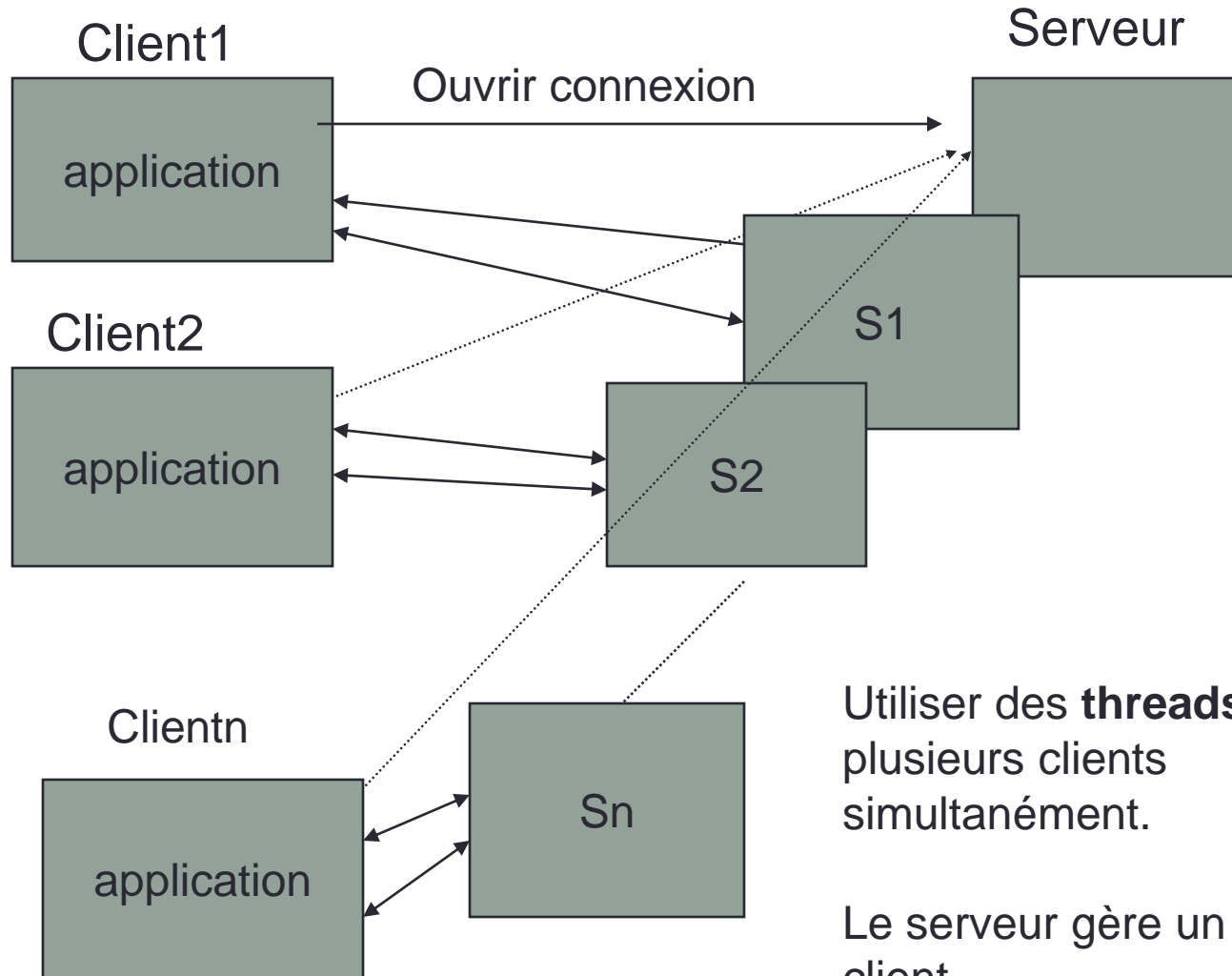
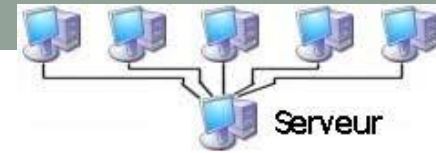
l'envoyer sur le flux de sortie

Attendre les données correspondant à la
réponse sur le flux d'entrée

les lire, les **analyser (unmarshalling)** et les
traiter

Fermer le socket

Plusieurs clients

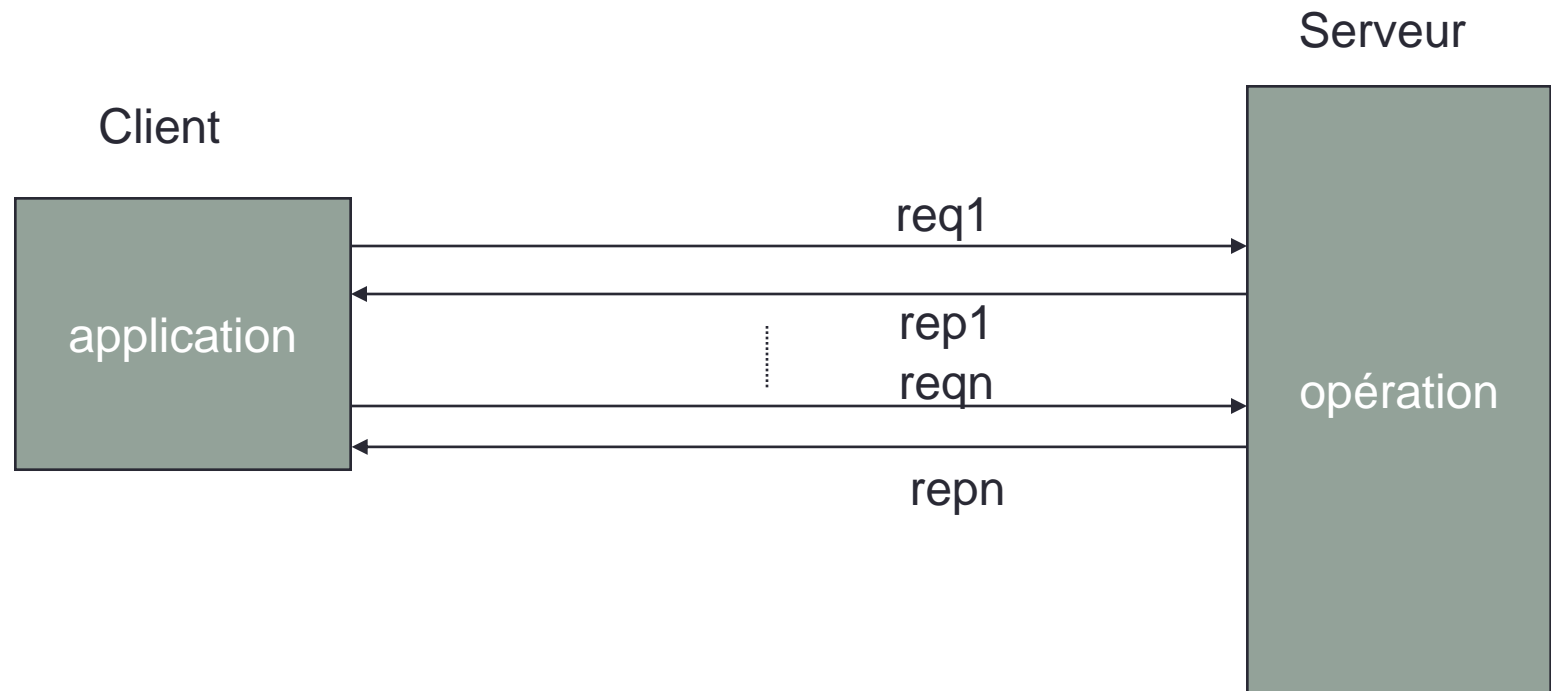
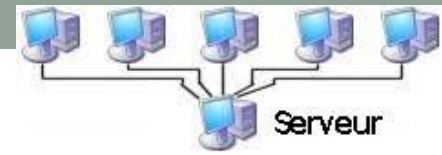


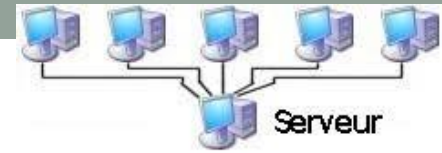
Utiliser des **threads** pour accepter plusieurs clients simultanément.

Le serveur gère un thread par client

Communication par message :

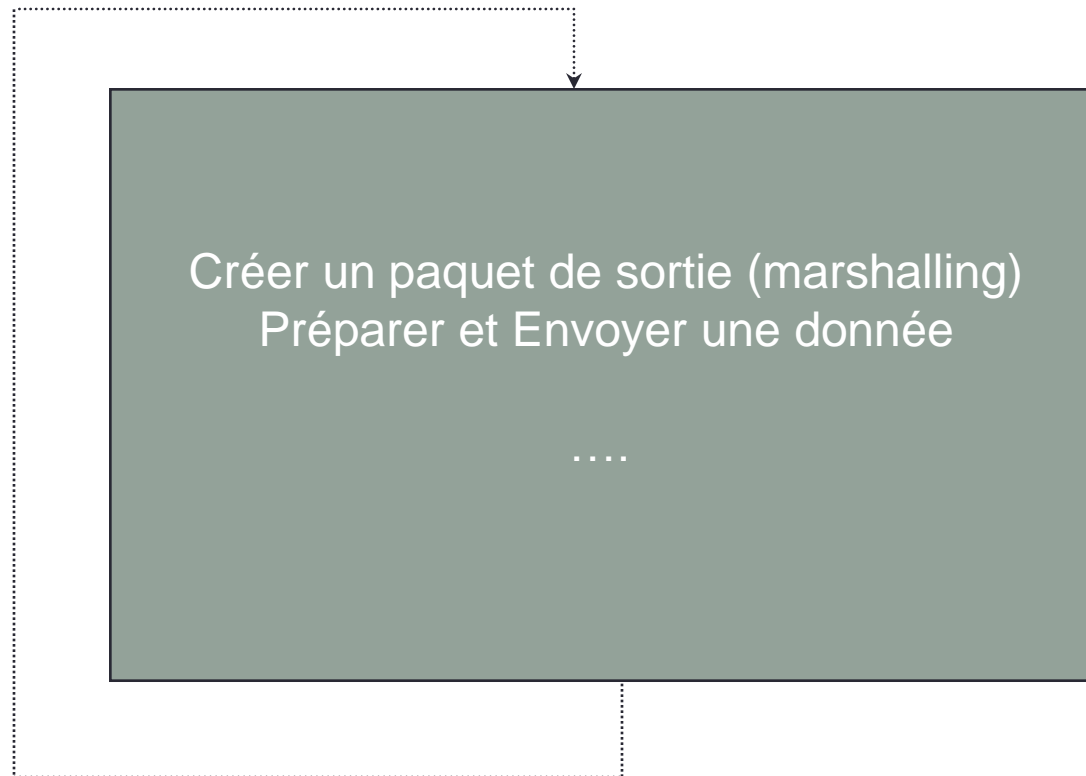
Envoi de datagrammes



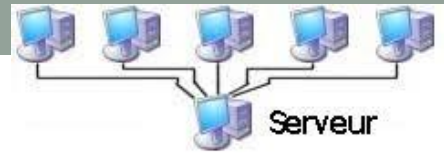


Scénario d'un client

Créer le socket d'entrée



Fermer le socket d'entrée



Scénario d'un serveur

Créer le socket d'entrée

Création d'un paquet d'entrée
Attente de données en entrée (unmarshalling)
Réception et traitement des données en entrée

....

Fermer le socket d'entrée

Marshalling et unmarshalling



l'information qui est lue doit être du même type et du même format que celle qui est écrite

Quel moyen connaissez vous en java ?

Interface Serializable



- Ne contient pas de méthode
- -> enregistrement et récupération de toutes les variables d'instances (pas de static)
 - + informations sur sa classe (nom, version), type et nom des variables
 - 2 classes compatibles peuvent être utilisées

Objet récupéré = une copie de l'objet enregistré •

Sérialisation-Desérialisation

Persistance et transfert réseau



- Enregistrer des objets dans un flux
 - Récupérer des objets dans un flux
-
- Via la méthode **writeObject()**
 - Classe implémentant l'interface **OutputStream**
 - Exemple : la classe `OutputStream`
 - Sérialisation d'un objet -> sérialisation de tous les objets contenus par cet objet
 - Un objet est sauvé qu'une fois : cache pour les listes circulaires

Exemple : la classe `InputStream`

- Via la méthode **readObject()**
 - Classe implémentant l'interface **InputStream**

RPC : Remote Procedure Call



1/ Définition d'un contrat



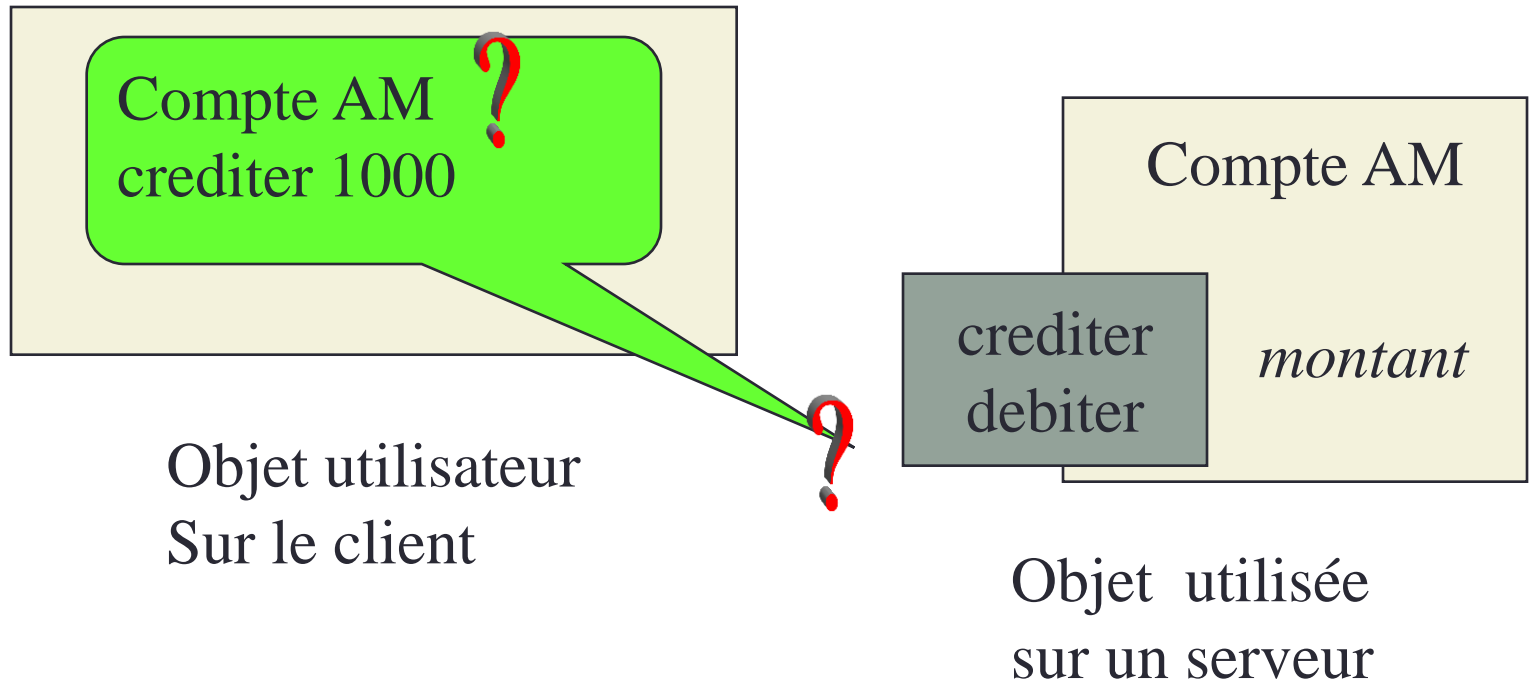
2/ Automatisation
du Marshalling



3/ Couplage FORT



Pour les objets distants



crediter et débiter <-> services proposés par le serveur

Un serveur peut abriter plusieurs objets distants et plusieurs types d'objets distant

Que peut on générer à partir d'une description des services ?



Spécifications
des données

Interface . Java

Générateurs



Fichiers
générés

Sérialisation et désérialisation
Code du serveur
Appels des clients

Stubs
Skeletons
Proxy

INVOCATION DE MÉTHODE À DISTANCE

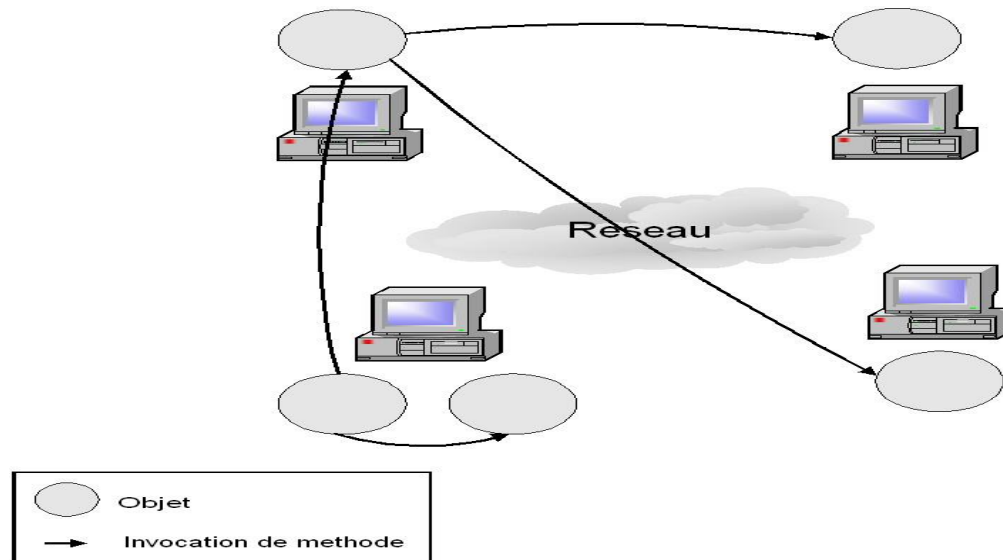
EXEMPLE : JAVA

REMOTE METHOD INVOCATION



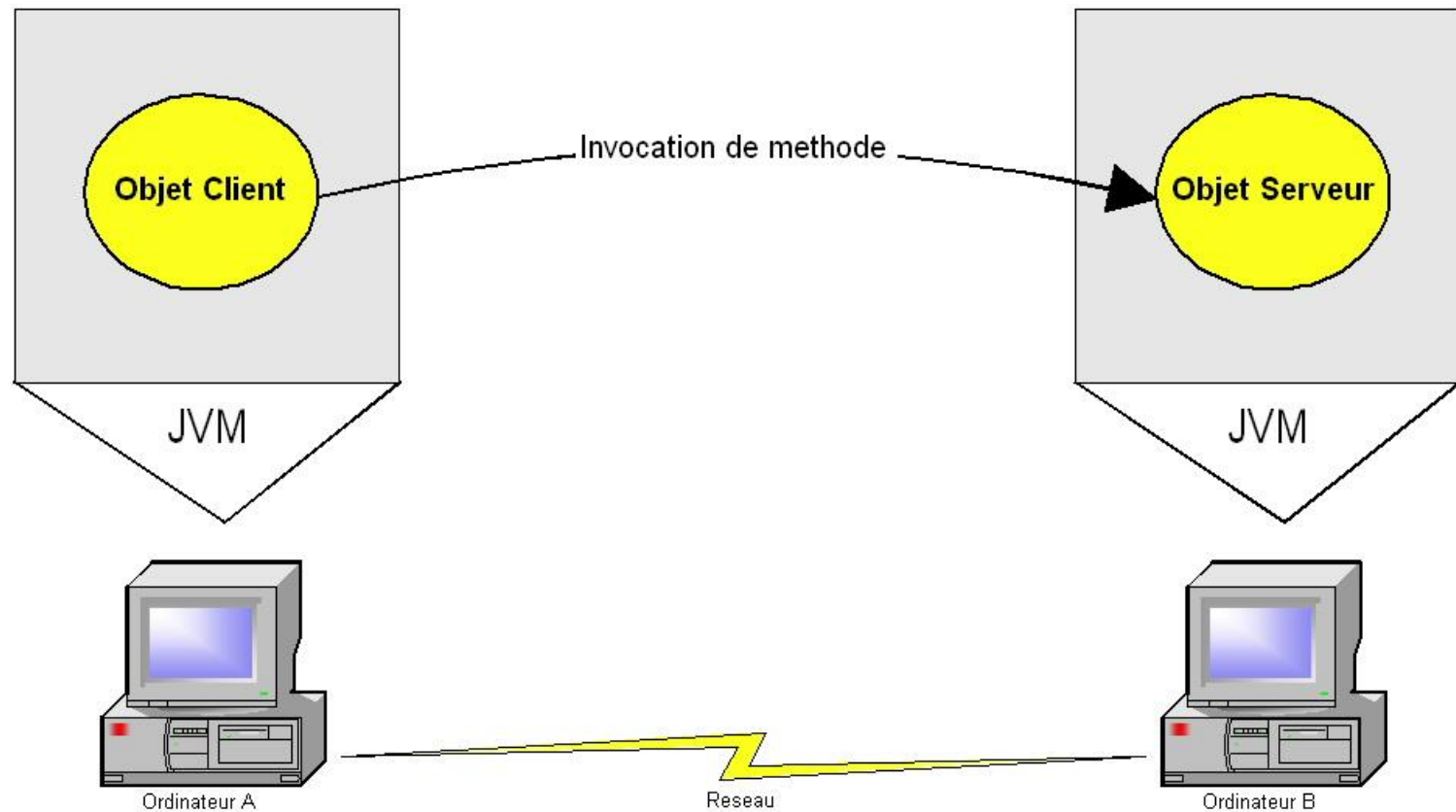
Invocation de méthodes distantes

- Mécanisme qui permet à des objets localisés sur des machines distantes de s'échanger des messages (invoker des méthodes)





Invocation de méthodes distantes

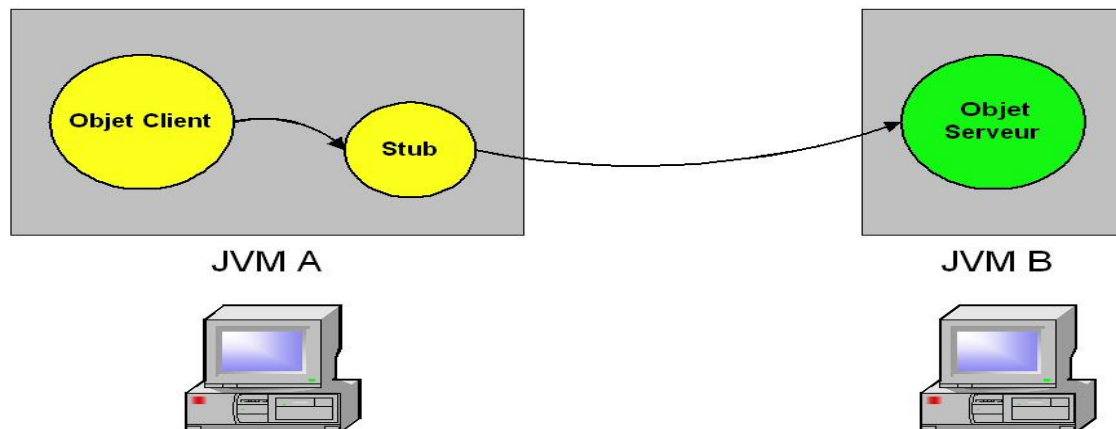


Java *Remote Method Invocation*



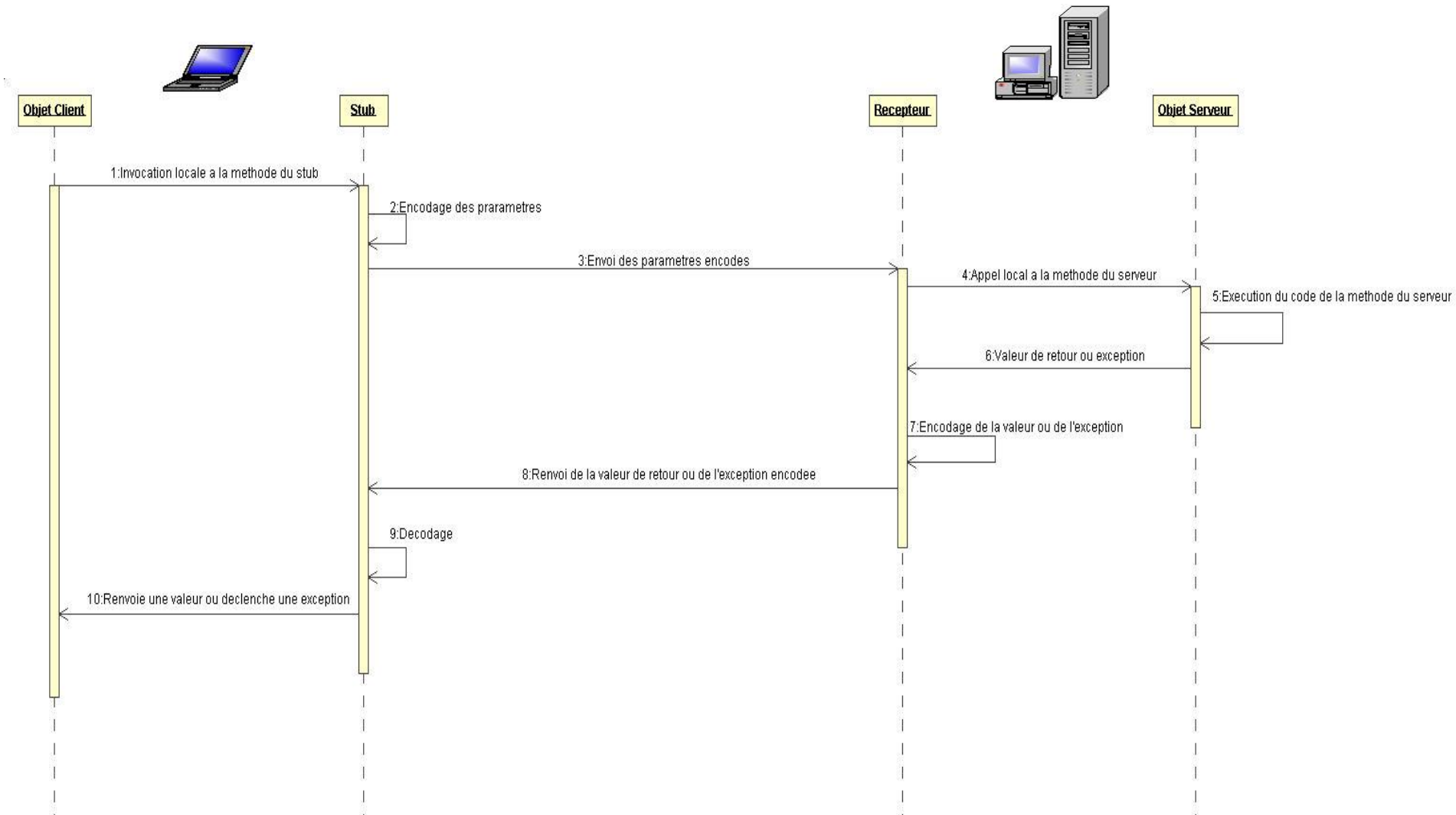
Stubs et encodage des paramètres

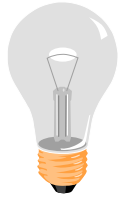
- le code d'un objet client invoque une méthode sur un objet distant
 - Utilisation d'un objet substitut dans la JVM du *client* : le **stub** (souche) de l'objet serveur





Stubs et encodage des paramètres





Du côté du client

- Appel d'une méthode du stub (de façon entièrement transparente)
- le stub construit un bloc de données avec
 - identificateur de l'objet distant à utiliser
 - description de la méthode à appeler
 - paramètres encodés qui doivent être passés
- puis il envoie ce bloc de données au serveur...



Du côté du serveur

- un objet de réception (Skeleton) effectue les actions suivantes :
 - décode les paramètres encodés
 - situe l'objet à appeler
 - invoque la méthode spécifiée
 - capture et encode la valeur de retour ou l'exception renvoyée par l'appel



Encodage des paramètres

- Encodage dans un bloc d 'octets afin d 'avoir une représentation indépendante de la machine
- Types primitifs et «basiques» (int/Integer...)
 - Encodés en respectant des règles établies
 - Big Endian pour les entiers...
- Objets ...



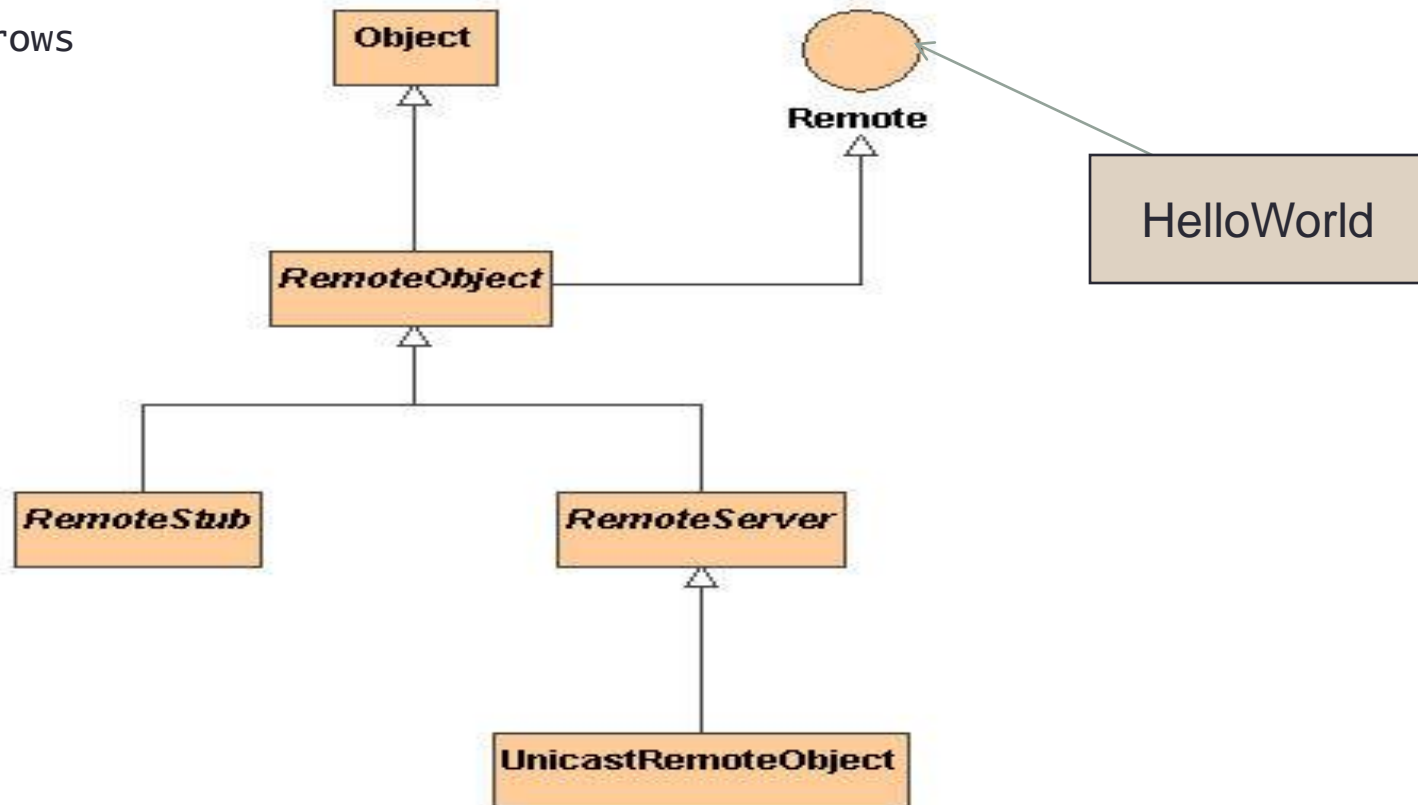
Générateur de stubs

- Les stubs gèrent la communication ainsi que l'encodage des paramètres
- Processus évidemment complexe...
- Entièrement automatique
 - Un outil permet de générer les stubs pour les OD
- En RMI La commande `rmic` du jdk rend transparent la gestion du réseau pour le programmeur
 - une référence sur un OD référence son stub local
- syntaxe = un appel local
 - `objetDistant.methode()`

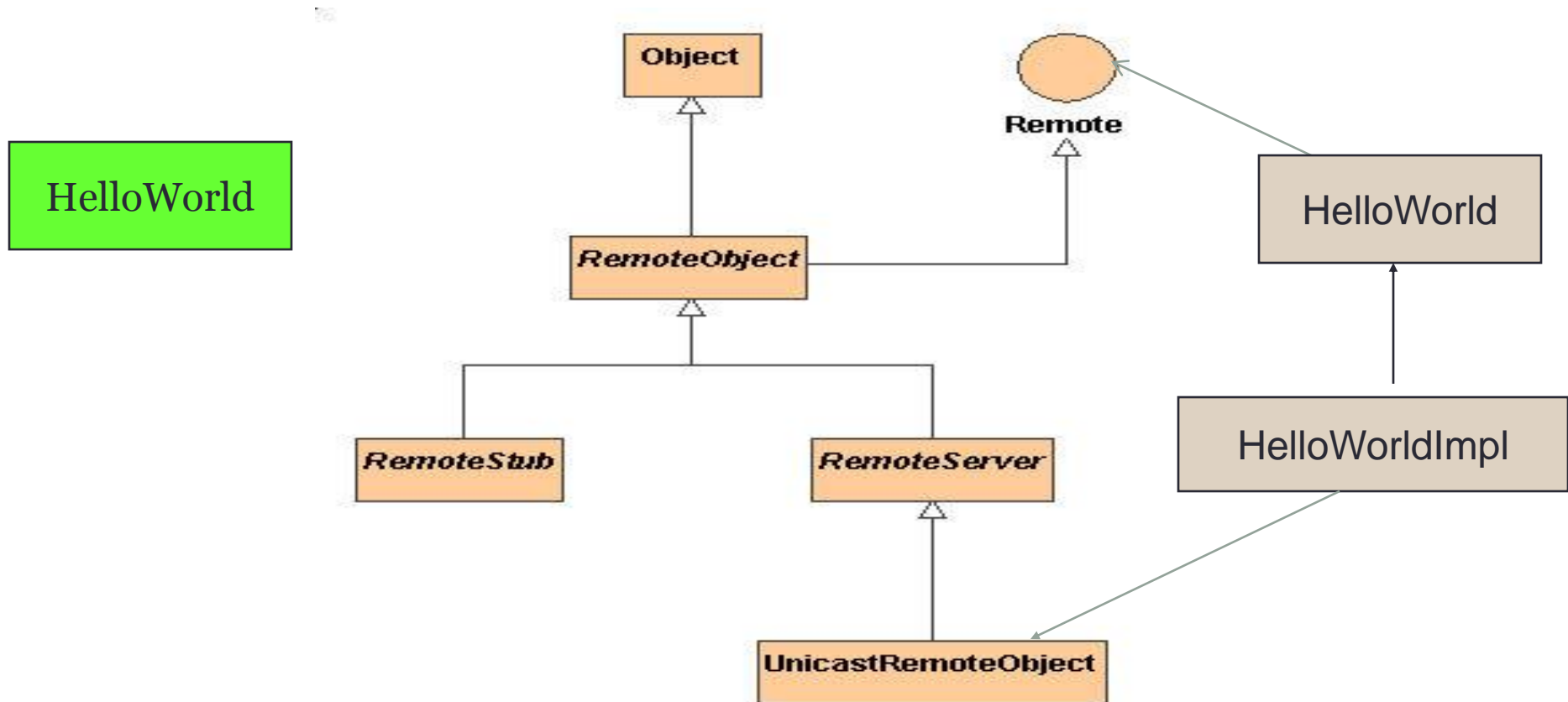
Rôle de l'interface : contrat

```
import java.rmi.*;  
interface HelloWorld extends  
Remote {  
    public String  
sayHello()  
    throws  
RemoteException;  
}
```

HelloWorld



Implémentation côté serveur



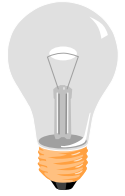


Interface = protocole d 'application
contrat entre un serveur et ses clients
entre l'objet appelé et l'objet appelant

- L 'interface HelloWorld

```
import java.rmi.*;
```

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



Quels paramètres ?

- On sera souvent amenés à passer des paramètres aux méthodes distantes...
- Les méthodes distantes peuvent retourner une valeur ou lever une exception...
- On a deux types de paramètres
 - Les objets locaux
 - Les objets distants



Passage de paramètres: ATTENTION

- Certains objets sont copiés (les objets locaux), d'autres non (les objets distants)
- Les objets distants, non copiés, résident sur le serveur
- Les objets locaux passés en paramètre doivent être sérialisables afin d'être copiés, et ils doivent être indépendants de la plate-forme
 - Les objets qui ne sont pas sérialisables lèveront des exceptions
 - Attention aux objets sérialisables qui utilisent des ressources locales !!!



Comment un client trouve les objets distants ?



1/ Utilisation du contrat



2/ Appel d'une méthode :
Marshalling



3/ Réponse :
Couplage FORT

Référence sur un objet

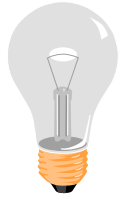
- On sait implanter un serveur d'un côté, et appeler ses méthodes de l'autre

MAIS

- Comment obtient-on une référence vers un stub de notre objet serveur ???

Localisation des objets de serveur

- On pourrait appeler une méthode sur un autre objet serveur qui renvoie une référence sur le stub...
- Mais quoi qu'il en soit, le premier objet doit lui aussi être localisé (La poule et l'œuf) !!!
- On a alors recours à un Service de Nommage



Les Services de Nommage

- Obtention d'une première référence sur un objet distant : « bootstrap » à l'aide d'un Service de Nommage ou Annuaire
- Enregistrement des références sur des objets du serveur afin que des clients les récupèrent
- Association de la référence de l'objet à une clé unique
- Recherche par la clé, d'un objet : le service de nommage renvoie la référence distante (le stub) de l'objet enregistré pour cette clé

RMIRegistry en Java



Enregistrement d'une référence



- L'objet serveur HelloWorld (coté serveur bien entendu...)
 - On a créé l'objet serveur et on a une variable qui le référence

```
HelloWorld hello = new HelloWorldImpl();
```

- On va enregistrer l'objet dans le RMIRegistry

```
Naming.rebind("HelloWorld",hello);
```

- L'objet est désormais accessible par les clients



Obtention d'une référence coté client



- sur l'objet serveur HelloWorld
 - On déclare une variable de type HelloWorld et on effectue une recherche dans l'annuaire

```
HelloWorld hello =  
(HelloWorld)Naming.lookup("rmi://www.helloworldserver.  
com/HelloWorld");
```

- On indique quelle est l'adresse de la machine sur laquelle s'exécute le RMIRegistry ainsi que la clé
- La valeur retournée doit être transtypée (castée) vers son type réel

DEVELOPPEMENT



1/ Définition d'un contrat



2/ Générer le Marshalling



3/ Code Serveur et Client
couplage FORT





Processus de développement



- 1) Définir le contrat : la liste des services
définir une interface Java pour un OD
- 2) Implémenter les services dans le serveur
créer et compiler une classe implémentant cette interface
- 3) Créer le serveur (enregistrer le ou les objets dans le serveur de noms, etc)
créer et compiler une application serveur RMI
- 4) Générer le marshalling – unmarshalling
créer les classes Stub (**`rmi.c`**)
- 5) Lancer le serveur (et le serveur de noms)
démarrer **`rmiregistry`** et lancer l'application serveur RMI
- 6) Les clients peuvent maintenant être créés
créer, compiler et lancer un programme client accédant à des OD du serveur

DEPLOIEMENT

1/ Contrat ?



2/ Marshalling ?



3/ Exécutables Serveur ?
Et Client ?





Que doit connaître le client ?

- Le contrat
- Le nom et le service de nommage
- Lorsqu'un objet serveur est passé à un programme, soit comme paramètre soit comme valeur de retour, ce programme doit être capable de travailler avec le stub associé
- Le programme client doit connaître la **classe** du stub



Que doit connaître le client ?

- les classes des paramètres, des valeurs de retour et des exceptions doivent aussi être connues...
 - Une méthode distante est déclarée avec un type de valeur de retour...
 - ...mais il se peut que l'objet réellement renvoyé soit une sous-classe du type déclaré



Déploiement dynamique vs déploiement statique

- Pour ne plus déployer les classes du serveur chez le client
 - Utilisation des **chargeurs de classes** qui téléchargent des classes depuis une URL
 - Utilisation d'un **serveur Web** qui fournit les classes
- Ce que ça change
 - Bien entendu, les classes et interfaces de l'objet distant ne changent pas
 - Le code du serveur ne change pas
 - **le client et la façon de le démarrer sont modifiés**
 - **Et lancer un serveur Web pour nos classes**



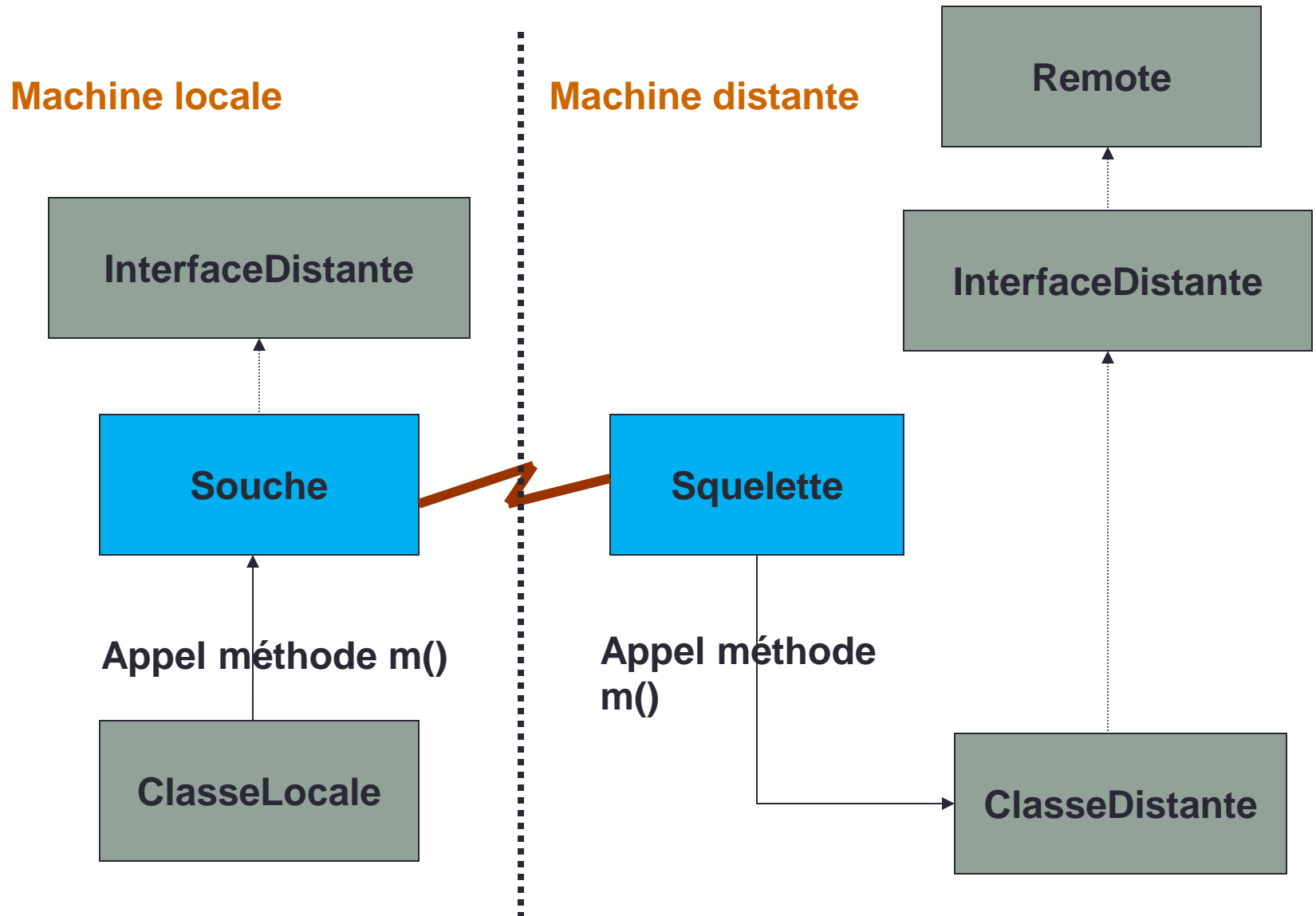
RMI (JAVA) MAIS AUSSI
CORBA (C++ JAVA,
ETC)...

PROTOCOLE D'APPLICATIONS EN RMI

```
public interface Surnoms extends java.rmi.Remote
{
    public Boolean enregistrer(String nom, String surnom) throws
        java.rmi.RemoteException,

        ServeurSurnoms.surnoms.ExisteDeja ;
    ....
}
```

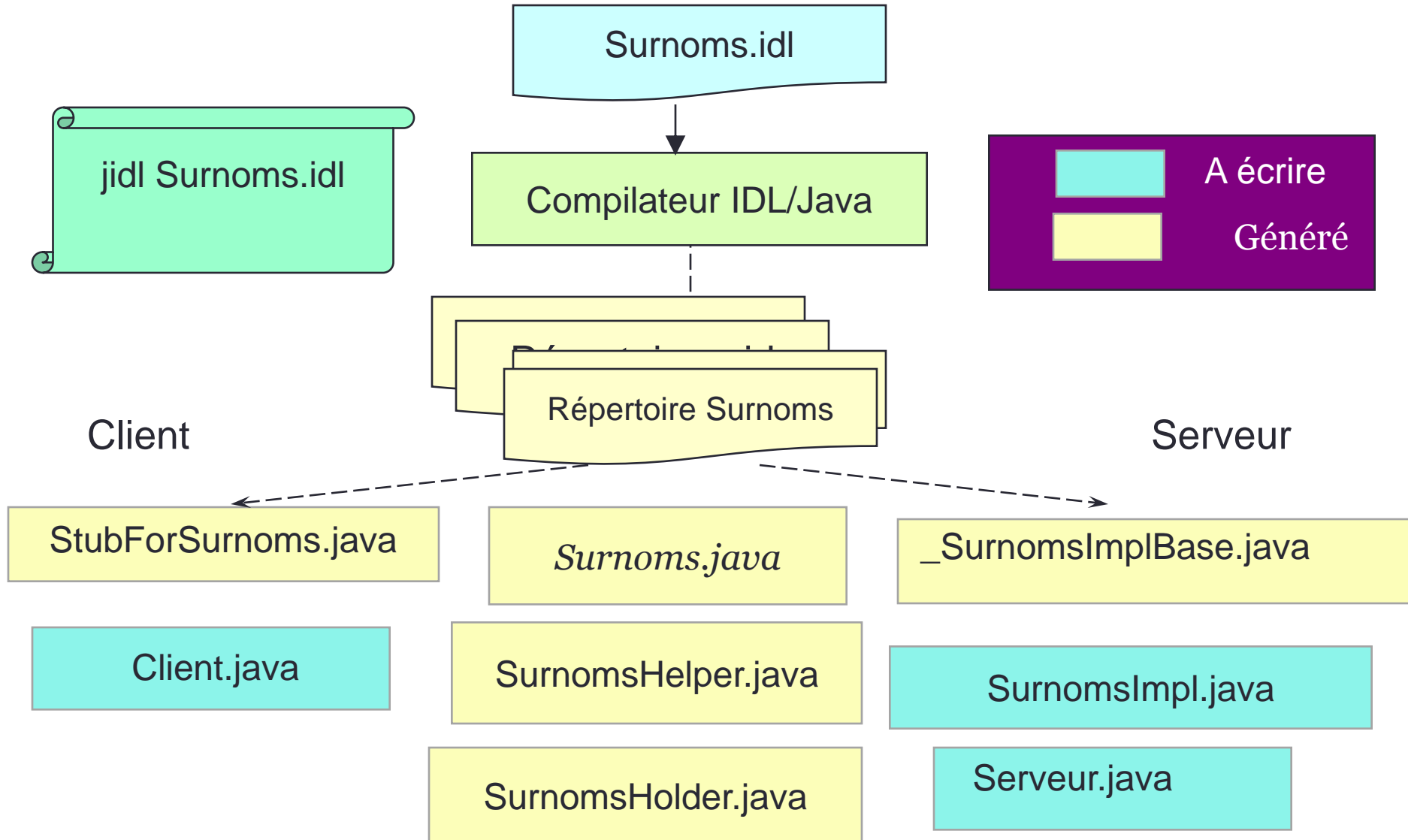
Classes et Interfaces



Protocole d'applications en CORBA

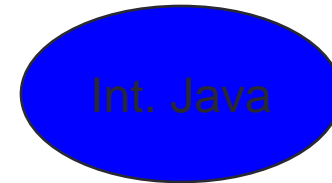
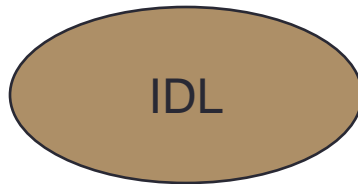
```
module Surnoms {  
  typedef string Nom ;  
  struct Personne {Nom nom;  
                    string surnom;};  
  typedef sequence<Personne> ListePersonnes;  
  interface Surnoms{  
    exception ExisteDeja{string surnom;};  
    boolean enregistrer(in Personne personne) raises (ExisteDeja);  
    .....  
  };  
};
```

Compilation interface IDL



Générateurs

Spécifications
des données



Générateurs



HETEROGENEITE

Fichiers
générés

Stubs Skeletons Proxy

(mise en œuvre de la sérialisation
et désérialisation...)

DES OBJETS DISTRIBUÉS AUX COMPOSANTS

Quels Composants ?

Une vision « simplifiée » : les Web Services

Des composants pratiques : EJB

EXEMPLE HELLO WORLD EN RMI

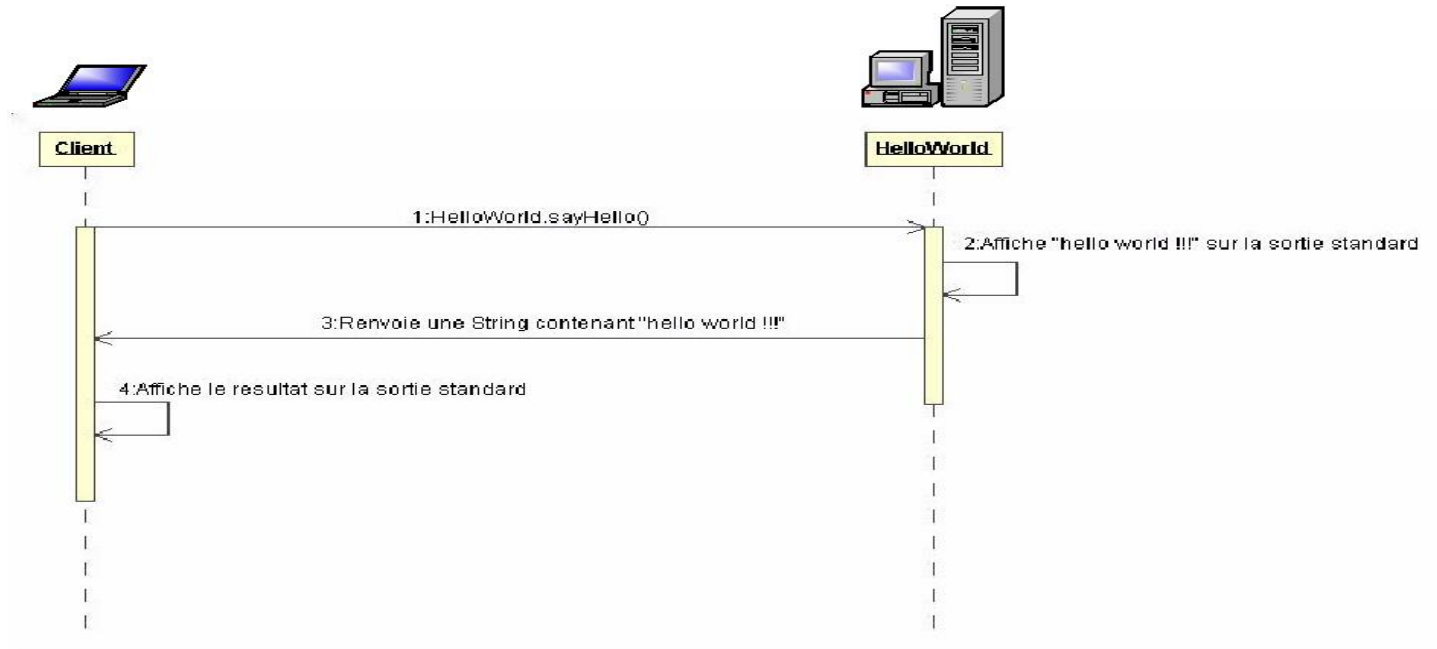
Avec la reflexivite Java



Stubs et rmic

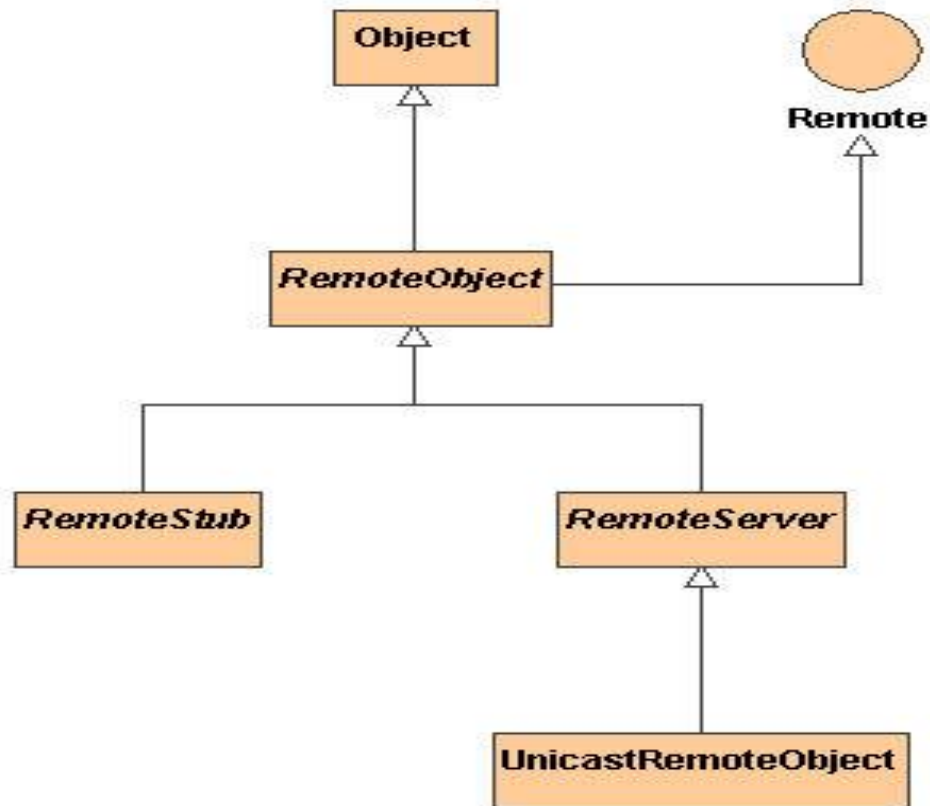
- La commande `rmic` du jdk rend transparent la gestion du réseau pour le programmeur
 - une référence sur un ODréférence son stub local
- syntaxe = un appel local
 - `objetDistant.methode()`

Un exemple : le sempiternel « Hello World » !!!





Interfaces et classes prédéfinies





Interface = protocole d 'application

- L 'interface HelloWorld

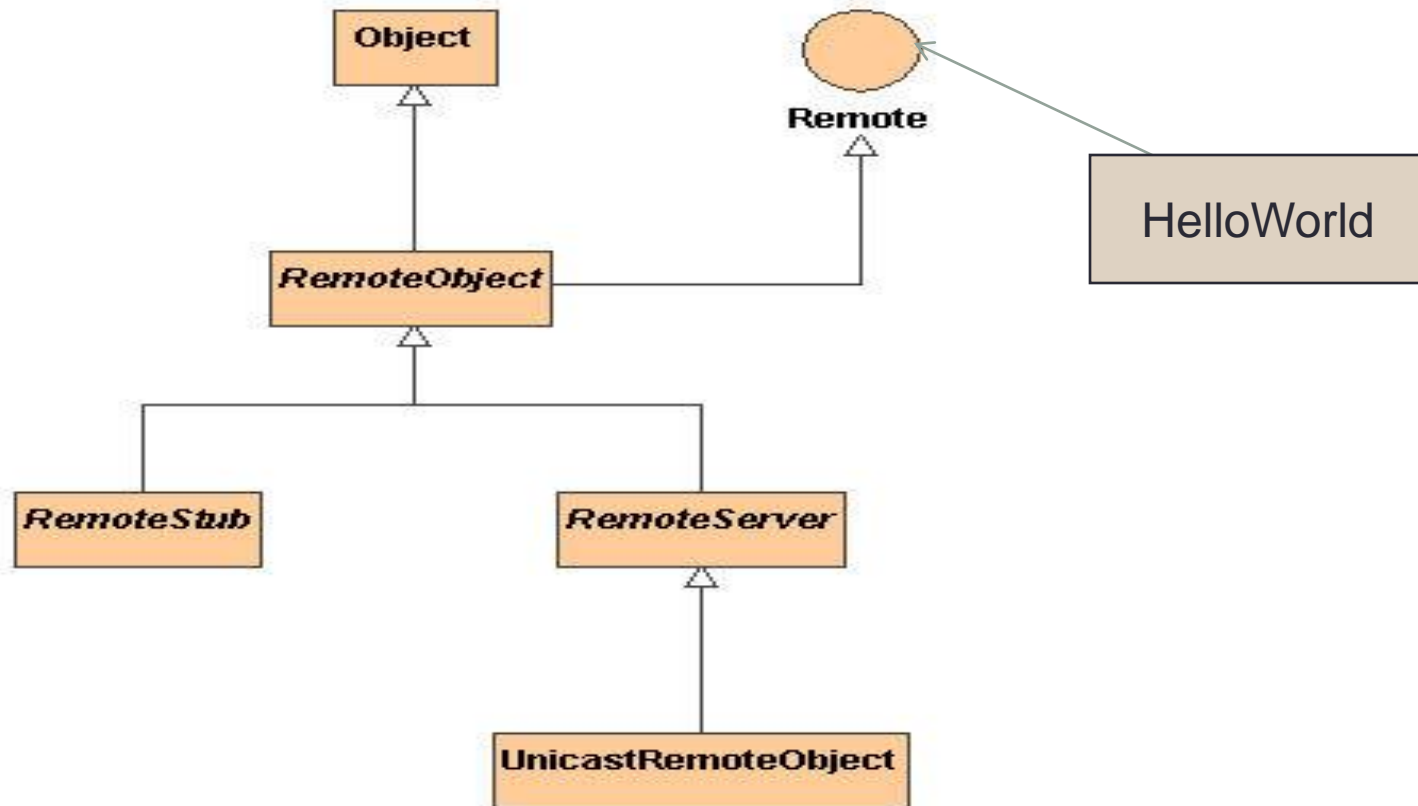
```
import java.rmi.*;
```

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



Rôle de l'interface

HelloWorld





Les exceptions

- L'exception `RemoteException` doit être déclarée par toutes les méthodes distantes
 - Appels de méthodes distants moins fiables que les appels locaux
 - Serveur ou connexion peut être indisponible
 - Panne de réseau
 - ...



Du côté client

```
HelloWorld hello = ...;  
    // Nous verrons par la suite comment obtenir  
    // une première référence sur un stub  
String result = hello.sayHello();  
System.out.println(result);
```



Du côté Serveur

- Implémentation de la classe qui gère les méthodes de l'interface HelloWorld

```
// Classe d'implémentation du Serveur
```

```
public class HelloWorldImpl  
    extends UnicastRemoteObject  
    implements HelloWorld {
```

```
    public String sayHello() throws RemoteException {
```

```
        String result = « hello world !!! »;  
        System.out.println(« Méthode sayHello invoquée... » +  
result);  
        return result;  
    }  
}
```



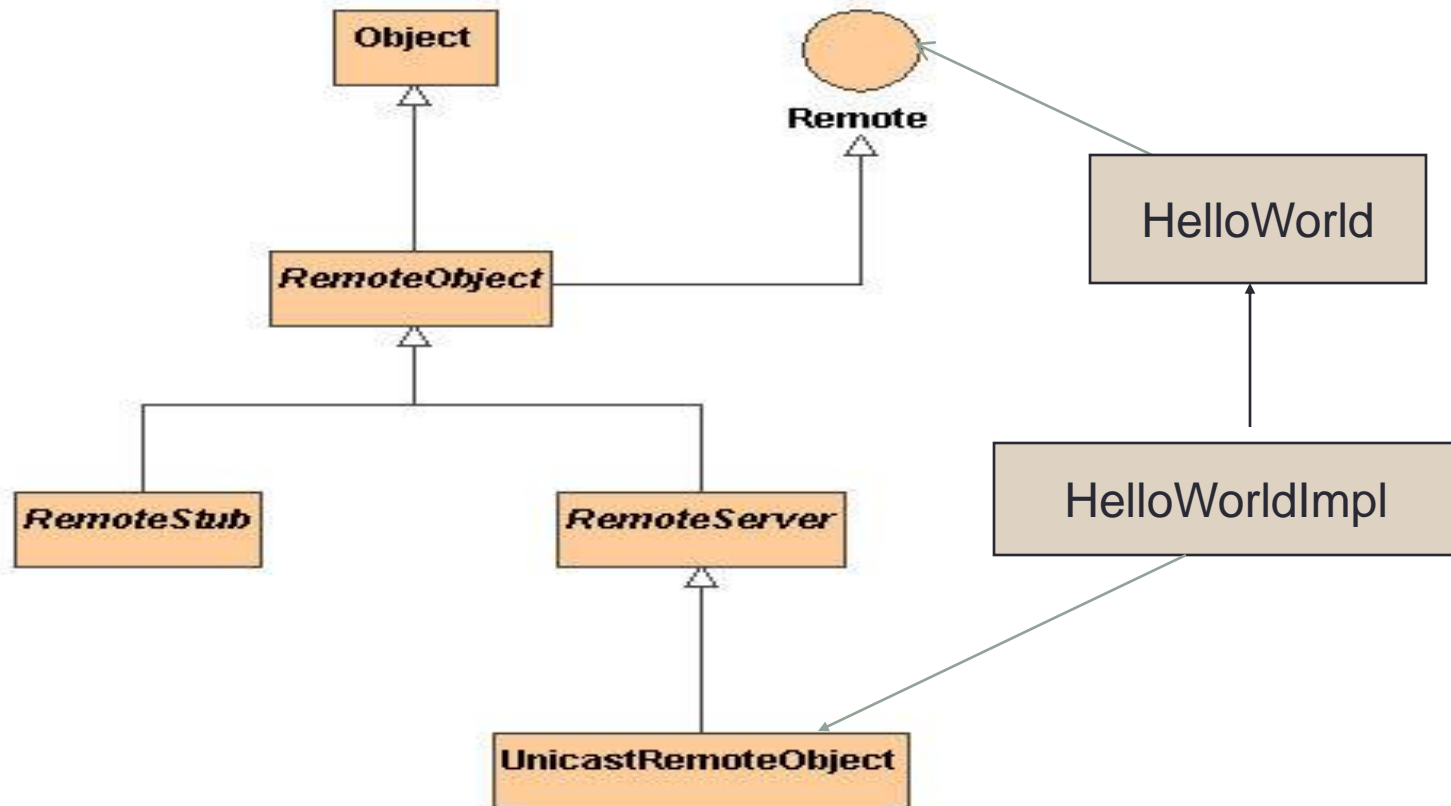

Classe d'implémentation

- doit implanter l'interface `HelloWorld`
- doit étendre la classe `RemoteServer` du package `java.rmi`
- `RemoteServer` est une classe abstraite
- `UnicastRemoteObject` est une classe concrète qui gère la communication et les stubs



Classe d'implémentation

HelloWorld





L 'outil RMIC



- outil livré avec le JDK permet de générer les stubs
 - > `rmic -v1.2 HelloWorldImpl`
 - génère un fichier `HelloWorldImpl_stub.class`

rmic doit être passé pour toutes les classes d'implémentation des OD afin d'en générer les stubs



Le RMIRegistry

- Programme exécutable fourni pour toutes les plates formes
- S'exécute sur un port (1099 par défaut) sur la machine serveur
- Pour des raisons de sécurité, seuls les objets résidant sur la même machine sont autorisés à lier/délier des références
- Un service de nommage est lui-même localisé à l'aide d'une URL



La classe Naming

- du package `java.rmi`
 - permet de manipuler le `RMIRegistry`
 - supporte des méthodes statiques permettant de
 - Lier des références d'objets serveur
 - `Naming.bind(...)` et `Naming.rebind(...)`
 - Délier des références d'objets serveur
 - `Naming.unbind(...)`
 - Lister le contenu du Naming
 - `Naming.list(...)`
 - Obtenir une référence vers un objet distant
 - `Naming.lookup(...)`



Enregistrement d'une référence



- L'objet serveur HelloWorld (coté serveur bien entendu...)
 - On a créé l'objet serveur et on a une variable qui le référence

```
HelloWorld hello = new HelloWorldImpl();
```

- On va enregistrer l'objet dans le RMIRegistry

```
Naming.rebind("HelloWorld",hello);
```

- L'objet est désormais accessible par les clients



Obtention d'une référence coté client



- sur l'objet serveur HelloWorld
 - On déclare une variable de type HelloWorld et on effectue une recherche dans l'annuaire

```
HelloWorld hello =  
(HelloWorld)Naming.lookup("rmi://www.helloworldserver.  
com/HelloWorld");
```

- On indique quelle est l'adresse de la machine sur laquelle s'exécute le RMIRegistry ainsi que la clé
- La valeur retournée doit être transtypée (castée) vers son type réel

Remarque

- Le Service de Nommage n'a pas pour fonction le référencement de tous les objets de serveur
 - Il devient vite complexe de gérer l'unicité des clés
- La règle de bonne utilisation du Naming est de lier des objets qui font office de point d'entrée, et qui permettent de manipuler les autres objets serveurs



Conception, implémentation et exécution de l'exemple



- Rappel
 - On veut invoquer la méthode `sayHello()` d'un objet de serveur distant de type `HelloWorld` depuis un programme Java client
- Nous allons devoir coder
 - L'objet distant
 - Le serveur
 - Le client
 - Et définir les permissions de sécurité et autres emplacements de classes...



Processus de développement



- 1) définir une interface Java pour un OD
- 2) créer et compiler une classe implémentant cette interface
- 3) créer et compiler une application serveur RMI
- 4) créer les classes Stub (**`rmic`**)
- 5) démarrer **`rmiregistry`** et lancer l'application serveur RMI
- 6) créer, compiler et lancer un programme client accédant à des OD du serveur



Hello World : L'objet distant

- Une interface et une classe d'implémentation
- stubs générés automatiquement par `rmic`
- *toutes les classes nécessaires à l'objet de client doivent être déployées sur la machine cliente et accessibles au chargeur de classes (dans le CLASSPATH)*
 - L'interface `HelloWorld` (`HelloWorld.class`)
 - Le stub `HelloWorldImpl_stub` généré par `rmic` pour cet objet



Hello World : Le serveur



- instancie un objet de type `HelloWorld` et attache au service de nommage
- puis objet mis en attente des invocations jusqu'à ce que le serveur soit arrêté

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

public class HelloWorldServer {

    public static void main(String[] args) {
        try {
            System.out.println("Création de l'objet serveur...");
            HelloWorld hello = new HelloWorldImpl();
            System.out.println("Référencement dans le RMIRegistry...");
            Naming.rebind("HelloWorld",hello);
            System.out.println("Attente d'invocations - CTRL-C pour
stopper");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



Serveur (suite)



- Après avoir compilé le tout...
- Pour démarrer le serveur, il faut tout d'abord lancer le RMIRRegistry
 - *Attention* : La base de registres RMI doit connaître les interfaces et les stubs des objets qu'elle enregistre (CLASSPATH) !!!

> rmiregistry &

- et ensuite on lance le serveur

> java HelloWorldServer

Création de l'objet serveur...

Référencement dans le RMIRRegistry...

Attente d'invocations - CTRL-C pour stopper



Hello World : client



- obtenir une référence sur l'objet de serveur HelloWorld, invoquer la méthode sayHello(), puis afficher le résultat de l'invocation sur la sortie standard

```
import java.rmi.*;

public class HelloWorldClient {

    public static void main(String[] args) {
        try {
            System.out.println("Recherche de l'objet serveur...");
            HelloWorld hello =
                (HelloWorld)Naming.lookup("rmi://server/HelloWorld");
            System.out.println("Invocation de la méthode sayHello...");
            String result = hello.sayHello();
            System.out.println("Affichage du résultat :");
            System.out.println(result);
            System.exit(0);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



Le client (suite)



- Il suffit ensuite de lancer le programme

```
> java HelloWorldClient
```

```
Recherche de l'objet serveur...
```

```
Invocation de la méthode sayHello...
```

```
Affichage du résultat :
```

```
hello world !!!
```

- Au niveau du serveur, le message...
Méthode sayHello invoquée... hello world !!!
- ...s'affichera dans la console

COMMENT IMPLEMENTER RMIC GÉNÉRER LES STUBS ?

Avec la Sérialisation et

Avec la réflexivité Java

Java reflection is useful because it supports dynamic retrieval of information about classes and data structures by name, and allows for their manipulation within an executing Java program. This feature is extremely powerful and has no equivalent in other conventional languages such as C, C++, Fortran, or Pascal.

Glen McCluskey

has focused on programming languages since 1988.
He consults in the areas of Java and C++ performance, testing, and technical documentation.

Réflexivité en Java ?

La réflexivité en Java permet
à un programme Java de s'examiner en cours d'exécution
de manipuler ses propriétés internes.

Par exemple, une classe Java peut obtenir le nom de tous ses membres.

Utilisation connue de la réflexivité : l'édition sous Eclipse
L'outil utilise la réflexivité pour obtenir liste des méthodes publiques qui peuvent être envoyées à une instance d'une classe

Que peut on faire ?

- Obtenir des informations sur les classes

 - Simuler l'opérateur *instanceof*

 - Découvrir la liste et le descriptif des méthodes

 - Obtenir des informations sur les constructeurs

 - Avoir des informations sur les variables

- Invoquer des méthodes, des constructeurs, affecter des variables

- Créer des objets et des tableaux

 - dynamiquement lors de l'exécution du programme
 - sans connaître à la compilation

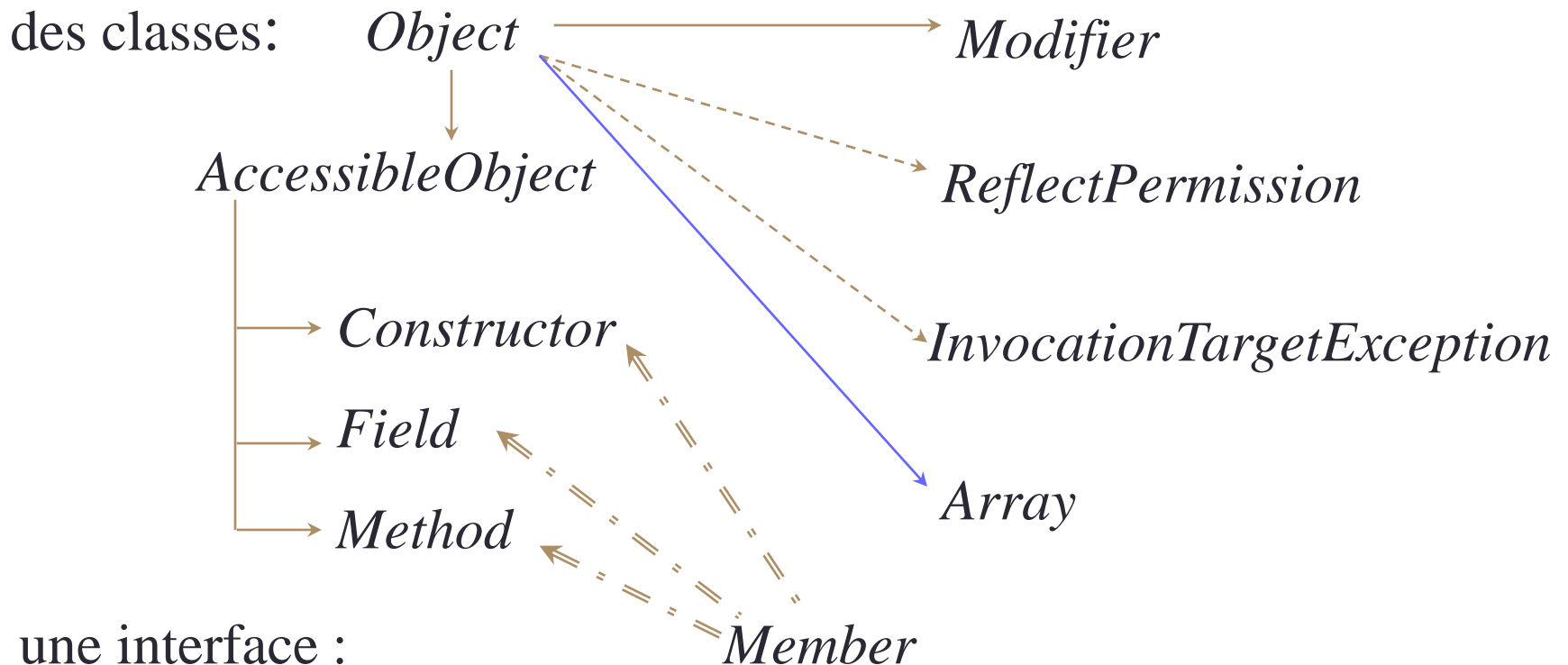
 - le nom et les arguments d'une méthode / constructeur

 - le nom de la variable

 - le type des objets, des tableaux....

Réflexivité = un package Java

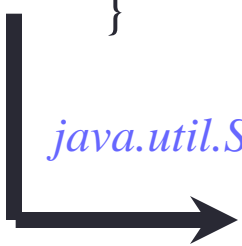
java.lang.reflect.*



Un exemple simple

```
public class DumpMethods {  
    public static void main(String args[])  
        { try {      Class c = Class.forName(args[0]);  
                  Method m[] = c.getDeclaredMethods();  
                  for (int i = 0; i < m.length; i++)  
                      System.out.println(m[i]);  
                  }  
          catch (Throwable e) { System.err.println(e);}  
        }  
    }
```

java.util.Stack



```
public java.lang.Object java.util.Stack.push(java.lang.Object)  
public synchronized java.lang.Object java.util.Stack.pop()  
public synchronized java.lang.Object java.util.Stack.peek()  
public boolean java.util.Stack.empty()  
public synchronized int java.util.Stack.search(java.lang.Object)
```

Oui mais comment ?

Comment travailler avec les classes du package *reflect* ?

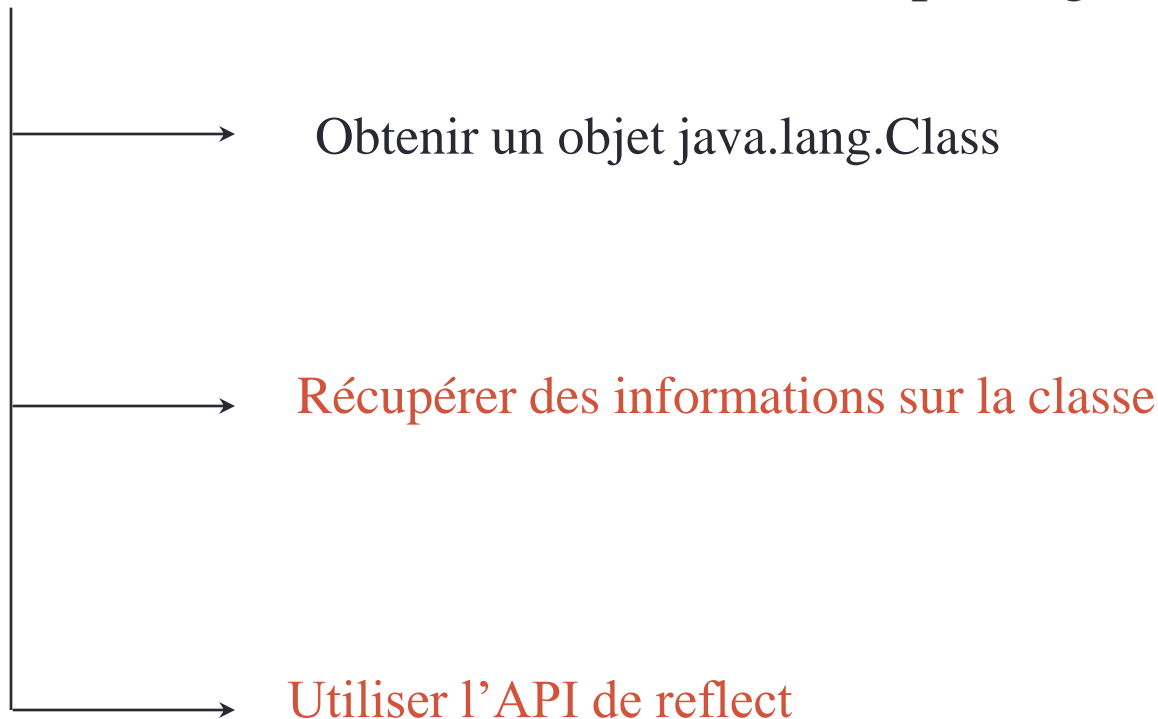


Illustration à partir de l'exemple

Classe *Class*

Les instances de *Class* représentent les classes et les interfaces d'une application Java.

Tous les tableaux sont aussi instances de *Class* (type des éléments, dimension).

Les types primitifs (boolean, byte, char, short, int, long, float, double) et le mot clé void sont aussi des objets *Class*.

Cette classe n'a pas de constructeur public.

Les instances sont créées automatiquement par la VM lorsque les classes sont chargées et par la méthode `defineClass` des class loader.

Obtenir un objet `java.lang.Class`

représentant de la classe que l'on veut manipuler :

Pour les types fondamentaux

Class c = int.class; ou *Class c = Integer.TYPE;*

TYPE est une variable prédéfinie du *wrapper* (Integer, par exemple) du type fondamental.

Pour les autres classes

Class c = Class.forName("java.lang.String");

Récupérer des informations sur la classe

Appeler une méthode sur l'objet classe récupéré :

getDeclaredMethods : pour obtenir la liste de toutes les méthodes déclarées dans la classe

getConstructors : pour obtenir la liste des constructeurs dans la classe

getDeclaredFields : pour obtenir la liste des variables déclarées dans la classe (quelque soit l'accesseur et non héritée)

getFields : pour obtenir la liste des variables publiques accessibles

....

Utiliser l'API de reflect

pour manipuler l'information

Par exemple :

```
Class c = Class.forName("java.lang.String");  
Method m[] = c.getDeclaredMethods();  
System.out.println(m[0]);
```

Classes *Method*, *Field* et *Constructor*

Method fournit les informations et les accès à une méthode (de classe, d'instance ou abstraite) d'une classe ou d'une interface.

Field fournit les informations et accès dynamiques aux champs (*static* ou d'instances) d'une classe ou d'une interface.

Constructor fournit des informations et des accès à un constructeur d'une classe.

Les méthodes d'une classe ?

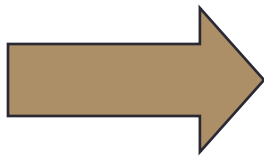
1. récupérer l'objet *Class* que l'on souhaite observer,
2. récupérer la liste des objets *Method* par *getDeclaredMethods* : méthodes définies dans cette classe (public, protected, package, et private)
getMethods permet d'obtenir aussi les informations concernant les méthodes héritées
3. A partir des objets méthodes il est facile de récupérer : les types de paramètres, les types d'exception, et le type de l'argument retourné sous la forme d'un type fondamental ou d'un objet classe.

Exemple d'exécution

```
Class cls = Class.forName("method1");
Method methlist[] = cls.getDeclaredMethods();
for (int i = 0; i < methlist.length; i++) {
    Method m = methlist[i];
    System.out.println("name = " + m.getName());
    System.out.println("decl class = " + m.getDeclaringClass());
    Class pvec[] = m.getParameterTypes();
    for (int j = 0; j < pvec.length; j++)
        System.out.println("param #" + j + " " + pvec[j]);
    Class evec[] = m.getExceptionTypes();
    for (int j = 0; j < evec.length; j++)
        System.out.println("exc #" + j + " " + evec[j]);
    System.out.println("return type = " + m.getReturnType());
}
```

Exemple d'exécution

```
public class method1 {  
    private int f1(Object p, int x) throws NullPointerException  
        {.....}  
    public static void main(String args[]) {....}
```



```
name = f1  
decl class = class method1  
param #0 class java.lang.Object  
param #1 int  
exc #0 class java.lang.NullPointerException  
return type = int  
name = main  
decl class = class method1  
param #0 class java.lang.String  
return type = void
```

COMMENT IMPLEMENTER RMI ?

au dessus des sockets Java

PROGRAMMATION RÉSEAUX

ILLUSTRATION : SOCKETS EN JAVA

Anne-Marie Déry

pinna@polytech.unice.fr



À travailler seuls



Concepts généraux



Mise en œuvre Java



Sockets

Outil de communication pour échanger des données entre un client et un serveur

Canaux de communication (descripteur d'entrée sortie dans lesquels on écrit et sur lesquels on lit)

Gestion similaire des entrées sorties standard (écran, clavier) et des fichiers



Plus précisément un socket

Plusieurs types de sockets :

| | |
|---|-----|
| pour la communication par flot de données | TCP |
| - fortement connectée | |
| - synchrone | |
| - type client-serveur | |

| | |
|---------------------------------------|-----|
| pour communication réseau par message | UDP |
| - en mode datagramme | |
| - en mode déconnecté | |

| | |
|---|-----|
| pour communication réseau par diffusion | UDP |
|---|-----|



Sockets en Java ?

Au dessus de TCP ou UDP



Une infrastructure puissante et flexible pour la programmation réseau

En Java toutes les classes relatives aux sockets sont dans le package **java.net**



Le Package net

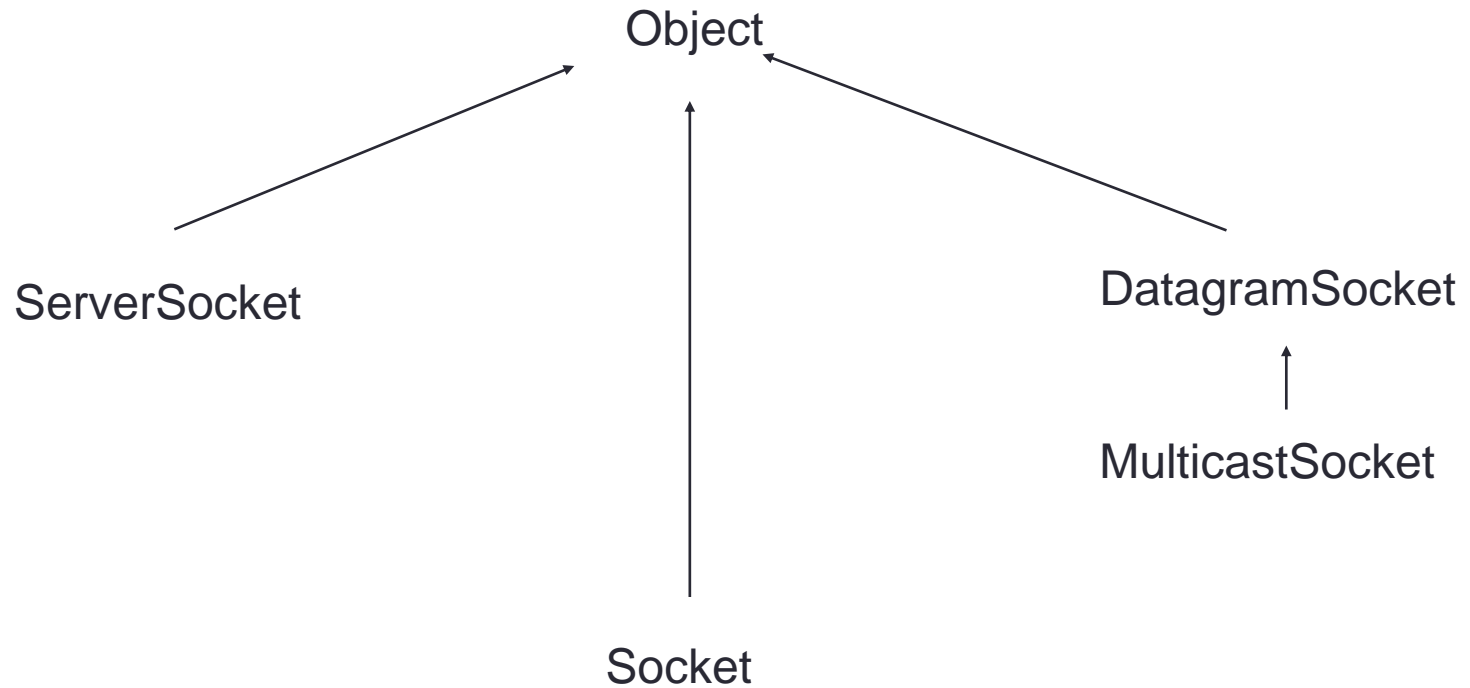
- Des Exceptions
- Des entrées Sorties
- Des Sockets
-



Plusieurs hiérarchies de classes

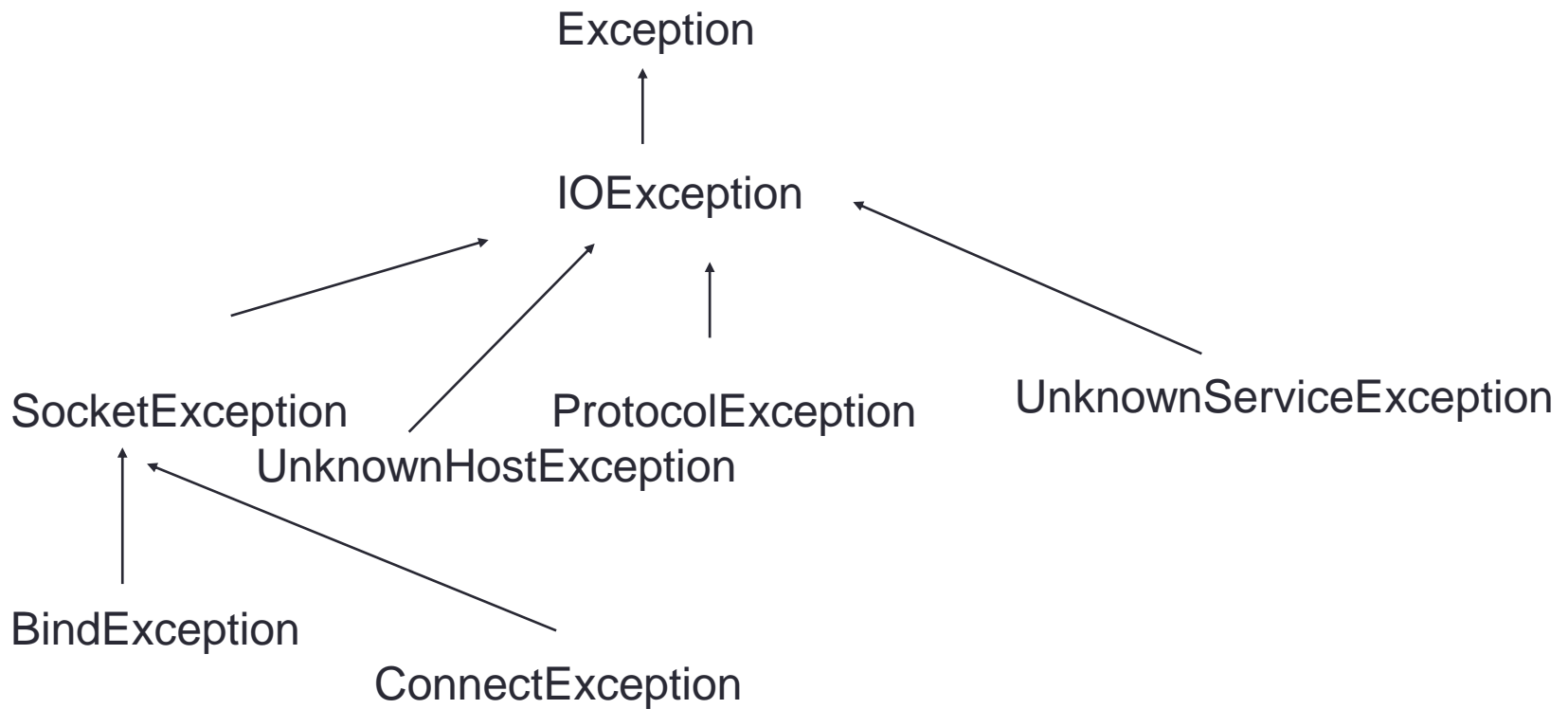


Des types de Sockets





Des exceptions





Java.net.InetAddress : nommage

2 constructeurs : un par défaut qui crée
une adresse vide (cf la méthode **accept** sur Socket)
un qui prend le nom de la machine hôte et
l'adresse IP de la machine.

Des accesseurs en lecture : pour récupérer l'adresse IP d'une
machine (**getByName**, **getAllByName**), des informations sur la
machine hôte (**getLocalHost**, **getLocalAddress**, **getLocalName**)

Des comparateurs : égalité (**equals**) et type d'adresse
(**isMulticastAddress**)

.....

COMMUNICATION CLIENT SERVEUR FORTEMENT CONNECTÉE



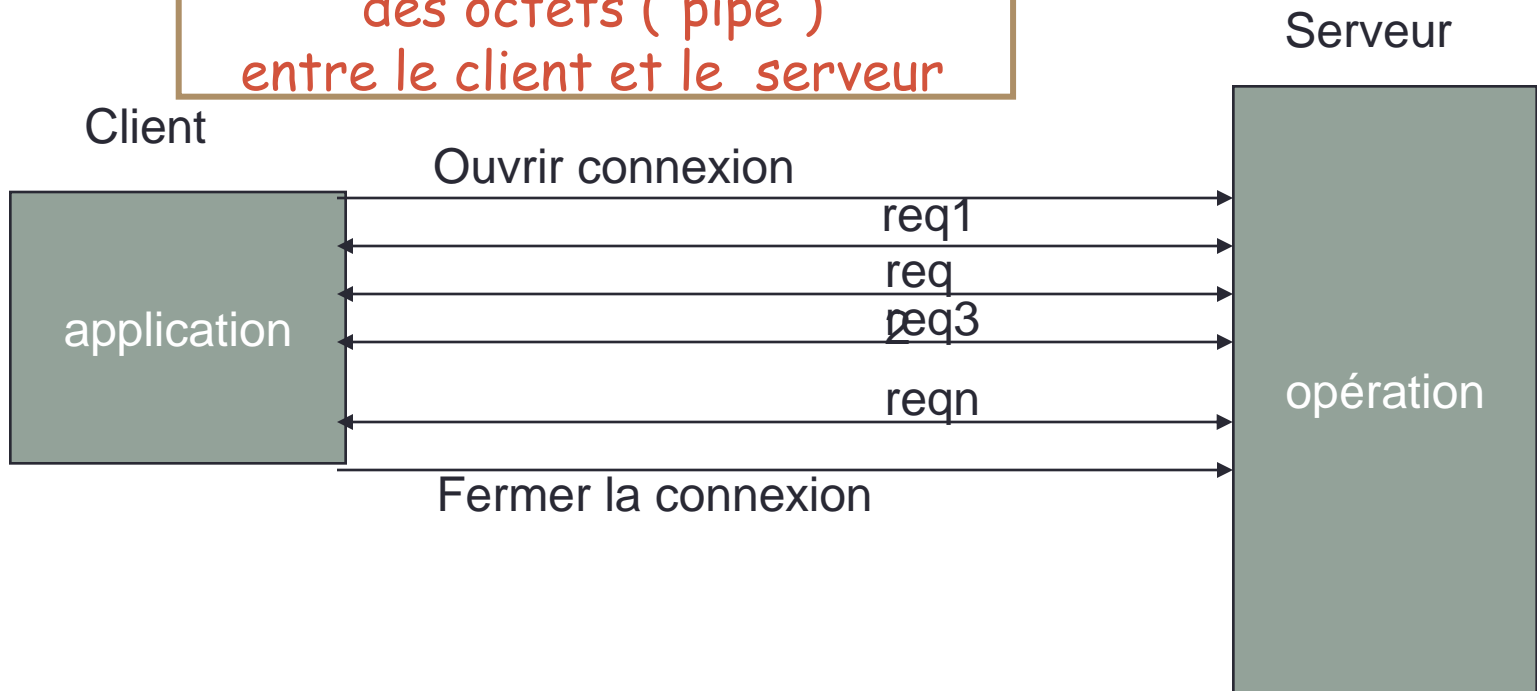
TCP



Flot de requêtes du client vers le serveur

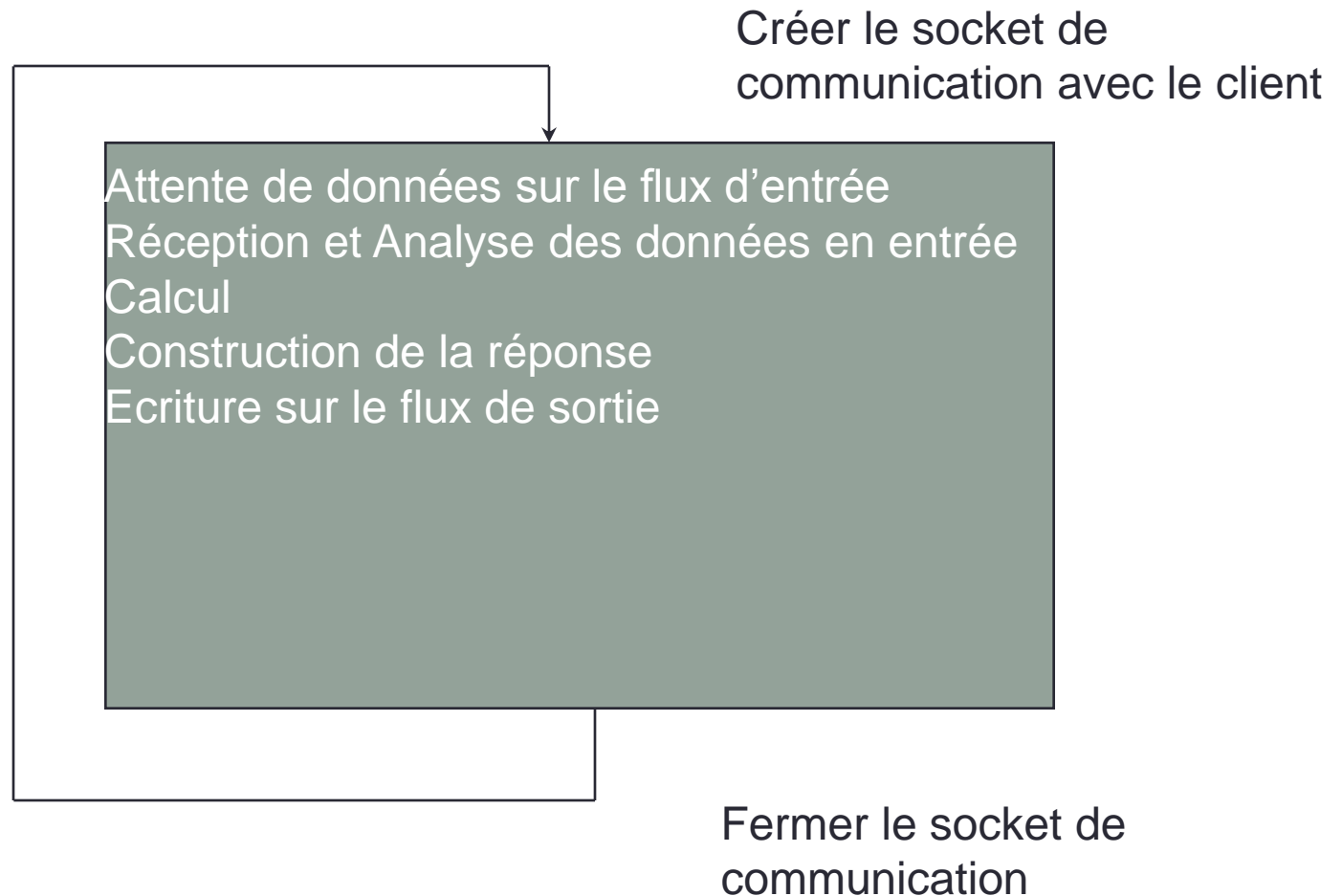
Point de vue application

TCP fournit un transfert fiable,
conservant l'ordre de transfert
des octets ("pipe")
entre le client et le serveur





Scénario d'un serveur pour un client





Scénario d'un client

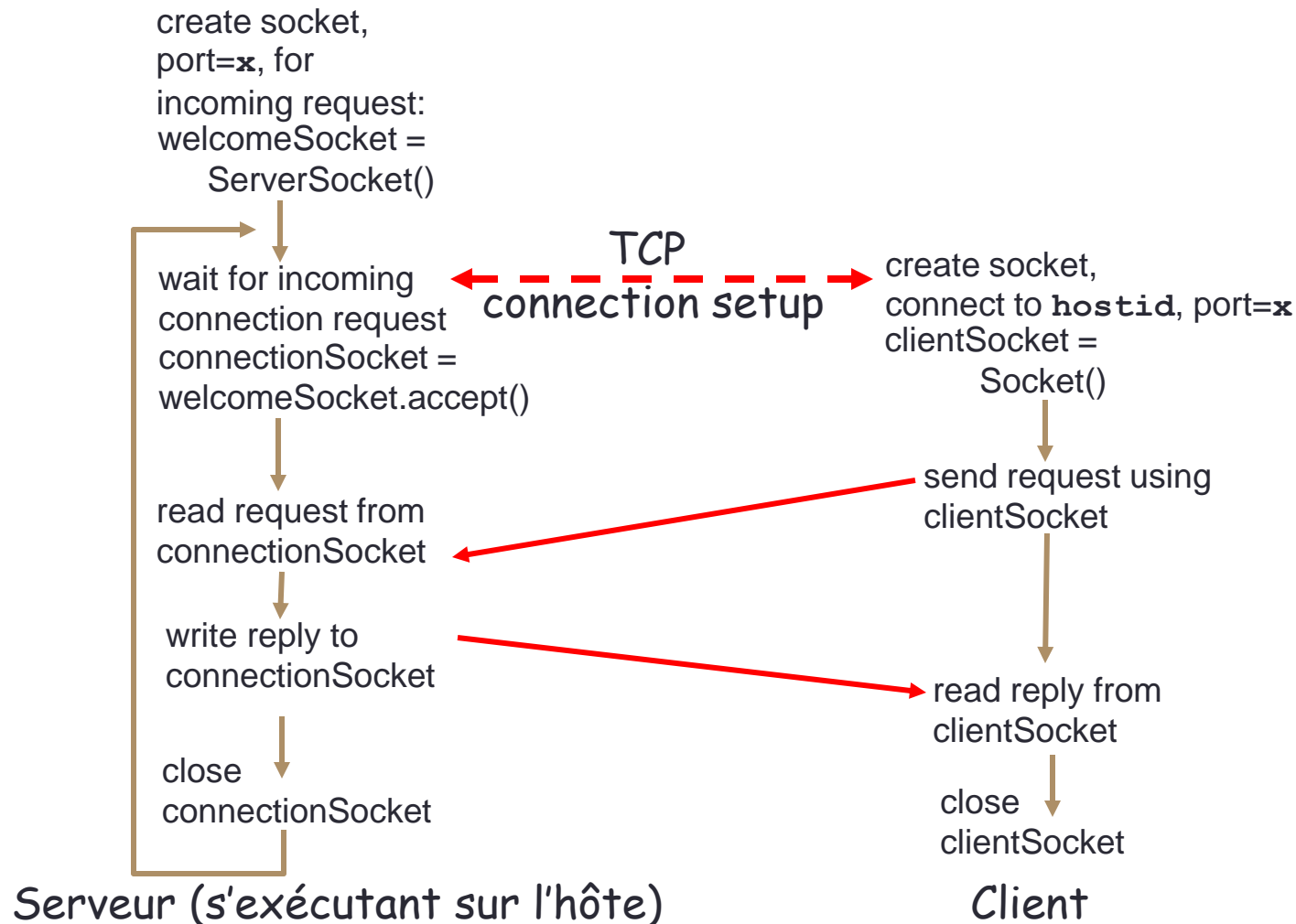
```
graph TD; A[Créer le socket de connexion avec le serveur  
Attendre que la connexion soit établie  
Récupérer la socket de communication] --> B[Préparer la requête  
l'envoyer sur le flux de sortie  
Attendre des données sur le flux d'entrée  
les lire et les traiter]; B --> C[Fermer le socket];
```

Préparer la requête
l'envoyer sur le flux de sortie
Attendre des données sur le flux d'entrée
les lire et les traiter

Créer le socket de connexion
avec le serveur
Attendre que la connexion
soit établie
Récupérer la socket de
communication

Fermer le socket

Interaction Client/server : socket TCP





TCP et Sockets

2 classes : **Socket** et **ServerSocket** (java.net package)
pour les canaux de communication

Classes pour le flot de données `XInputStream` et `XOutputStream`



TRANSFERT DE DONNÉES

Connexion +

« **Marshalling** »



Accepter les connexion - Dans un serveur ?

Créer un objet socket pour écouter les demandes de connexion sur le numéro de port associé au service

```
ServerSocket myService;  
try {  
    myService = new ServerSocket(PortNumber);  
}  
catch (IOException e) {System.err.println(e);}
```

Créer un objet socket pour accepter une connexion d'un client : cet objet servira pour tous les transferts d'information de ce client vers le serveur

```
Socket clientSocket = null;  
try {clientSocket = myService.accept();}  
catch (IOException e) {System.err.println(e); }
```



Demander à se Connecter = ouvrir un socket

Dans un client

identifier la machine à laquelle on veut se connecter et le numéro de port sur lequel tourne le serveur implique de créer un socket pour cette communication

```
Socket myClient;  
    try {  
        myClient = new Socket("Machine name", PortNumber);  
    }  
    catch (IOException e) {  
        System.out.println(e);  
    }  
}
```

Machine name : machine à laquelle on veut se connecter
PortNumber port sur lequel tourne le serveur (> 1023)



Comment envoyer une information ?

Côté client : pour envoyer une **requête au serveur**

Côté serveur : pour envoyer une **réponse au client**

1 Créer un flux de sortie pour le socket pour écrire l'information

2 Constituer le contenu des données à émettre (transformer entiers, doubles, caractères, objets en lignes de texte)

Exemples d'entrée sortie : permettent le transfert de données **en format « chaines de caractères »**

DataStream : écrire des types de données primitifs;

```
output= new DataOutputStream(clientSocket.getOutputStream());
```

PrintStream pour afficher des valeurs des types de base (write et println)

PrintStream output

```
try {output = new PrintStream(myClient.getOutputStream());}  
catch (IOException e) {System.err.println(e);}
```



Comment recevoir de l'information ?

Côté serveur : on doit **lire la requête du client**

Côté client : on doit **recevoir une réponse du serveur**

- 1 Créer un flux d'entrée pour le socket et lire l'information sur le flux
- 2 Reconstituer les données émises (entiers, doubles, caractères, objets)
à partir des lignes de texte reçues

`DataInputStream` : lire des lignes de texte, des entiers, des doubles, des caractères...
(`read`, `readChar`, `readInt`, `readDouble`, and **`readLine`**, .)

Côté serveur

```
DataInputStream input;  
try {  
    input = new  
    DataInputStream(clientSocket.get  
InputStream());  
}  
catch (IOException e)  
{System.out.println(e);}
```

Côté client

```
try {input = new  
    DataInputStream(myClient.getInput  
Stream());}  
catch (IOException e)  
{System.out.println(e);}
```

Autres entrées sorties

```
echoSocket = new Socket( "jessica", 7);  
    out = new PrintWriter(echoSocket.getOutputStream(), true);  
    in = new BufferedReader(new InputStreamReader(  
        echoSocket.getInputStream()));
```

Le `BufferedReader` prend un `Reader` en paramètre et non un `Stream`

Utilisation des **`ObjectInputStream`** et **`ObjectOutputStream`** pour **une sérialisation objet**

L'output doit être initialisé en premier sinon blocage à la
Création du flux de sortie.



Entrées sorties : comment procéder ?

Quid du marshalling ?

l'information qui est lue doit être du même type et du même format que celle qui est écrite

ATTENTION au choix de vos entrées sorties – respecter la Cohérence des données transmises



Les flux d'entrée et de sorties doivent être compatibles,
Les flux d'entrée déterminent la sérialisation
La façon de les utiliser peut impacter votre protocole d'applications (writeln)

Le client doit il connaître la nature des E/S du serveur pour être écrit ?



Comment se déconnecter ?

Fermer correctement les flux d'entrée sortie et les sockets en cause.

Fermer les output et input stream avant le socket.

Côté client

```
output.close();  
input.close();  
myClient.close();
```

Côté serveur

```
output.close();  
input.close();  
clientSocket.close();  
myService.close();
```

Sockets (Communication Client serveur)

Le serveur est à l'écoute des requêtes sur un port particulier
Un client doit connaître l'hôte et le port sur lequel le serveur écoute. Le client peut tenter une connexion au serveur

Le serveur connecte le client sur un nouveau no de port
et reste en attente sur le port original

Client et serveur communiquent en écrivant et lisant sur un socket

Ils doivent être d'accord sur les messages échangés : sérialisation et protocole d'applications pour communiquer

Serveur Echo

Un serveur similaire à echo (port 7).
Reçoit un texte du client et le renvoie identique
Le serveur gère un seul client.

```
import java.io.*; import java.net.*;
public class echo3 {
    public static void main(String args[]) {
        ServerSocket echoServer = null;
        String line;
        DataInputStream is;
        PrintStream os;
        Socket clientSocket = null;
        try { echoServer = new ServerSocket(9999);}
        catch (IOException e) {System.out.println(e); }
    try {
        clientSocket = echoServer.accept();
        is = new DataInputStream(clientSocket.getInputStream());
        os = new PrintStream(clientSocket.getOutputStream());
        while (true) {
            line = is.readLine();
            os.println(line);
        }
    }
    catch (IOException e) {
        System.out.println(e);}
    } }
```





Comment écrire un client ?

Toujours 4 étapes

1. Ouvrir un socket.
2. Ouvrir un input et un output stream sur le socket.
3. Lire et écrire sur le socket en fonction du protocole du serveur.
4. Effacer Fermer

Seule l'étape 3 change selon le serveur visé



Client SMTP (Simple Mail Transfer Protocol),

```
import java.io.*; import java.net.*;
public class smtpClient {
    public static void main(String[] args) {
        Socket smtpSocket = null;    // le socket client
        DataOutputStream os = null; // output stream
        DataInputStream is = null;   // input stream
        try { smtpSocket = new Socket("hostname", 25);
            os = new DataOutputStream(smtpSocket.getOutputStream());
            is = new DataInputStream(smtpSocket.getInputStream());
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host: hostname");
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to: hostname");
        }
    }
}
```



Le protocole SMTP, RFC1822/3

```
if (smtpSocket != null && os != null && is != null) {  
    try{os.writeBytes("HELO\n");  
        os.writeBytes("MAIL From: <pinna@essi.fr>\n");  
        os.writeBytes("RCPT To: <pinna@essi.fr>\n");  
        os.writeBytes("DATA\n");  
        os.writeBytes("From: pinna@essi.fr\n");  
        os.writeBytes("Subject: Qui est là ?\n");  
        os.writeBytes("Vous suivez toujours ?\n"); // message  
        os.writeBytes("\n.\n");  
        os.writeBytes("QUIT");  
    }
```



SMTP

```
// attente de "Ok" du serveur SMTP,
String responseLine;
while ((responseLine = is.readLine()) != null) {
    System.out.println("Server: " + responseLine);
    if (responseLine.indexOf("Ok") != -1) {break;}}
os.close();
is.close();
smtpSocket.close();
} catch (UnknownHostException e) {
    System.err.println("Trying to connect to unknown host: " + e);
} catch (IOException){
    System.err.println("IOException: " + e);}
} } }
```

Avez-vous suivi ?

Quel est le protocole d'application pour le service SMTP ?
peut-on le changer ?

Quelle est l'entrée du service sur la machine hôte ?
peut-on la changer ?

Quel est le marshalling associé au protocole d'application ?
peut-on le changer ?

Quel est le protocole de transport ?
peut-on le changer ?



TCP et Sockets

La classe **ServerSocket**

des constructeurs : par défaut,
no de port associé, + taille de la liste de clients en attente +
adresse...

des accesseurs en lecture : no de port sur lequel
le socket écoute, adresse à laquelle il est connecté (**getPort**,
getInetAddress, ...)

des méthodes : **accept** pour accepter une communication avec
un client, **close**

...



TCP et Sockets

La classe **Socket** :

une batterie de constructeurs : par défaut,
no de port + adresse / nom de machine et service distante,
+ no de port + adresse locale,
créent un socket en mode Stream ou DataGramme

des accesseurs en lecture : no de port et adresse à
laquelle il est connecté, no de port et adresse à laquelle il est lié,
input et output Stream associés (**getPort**, **getInetAddress**,
getLocalPort, **getLocalAddress**, **getInputStream**,
getOutputStream...)

des méthodes : **close**

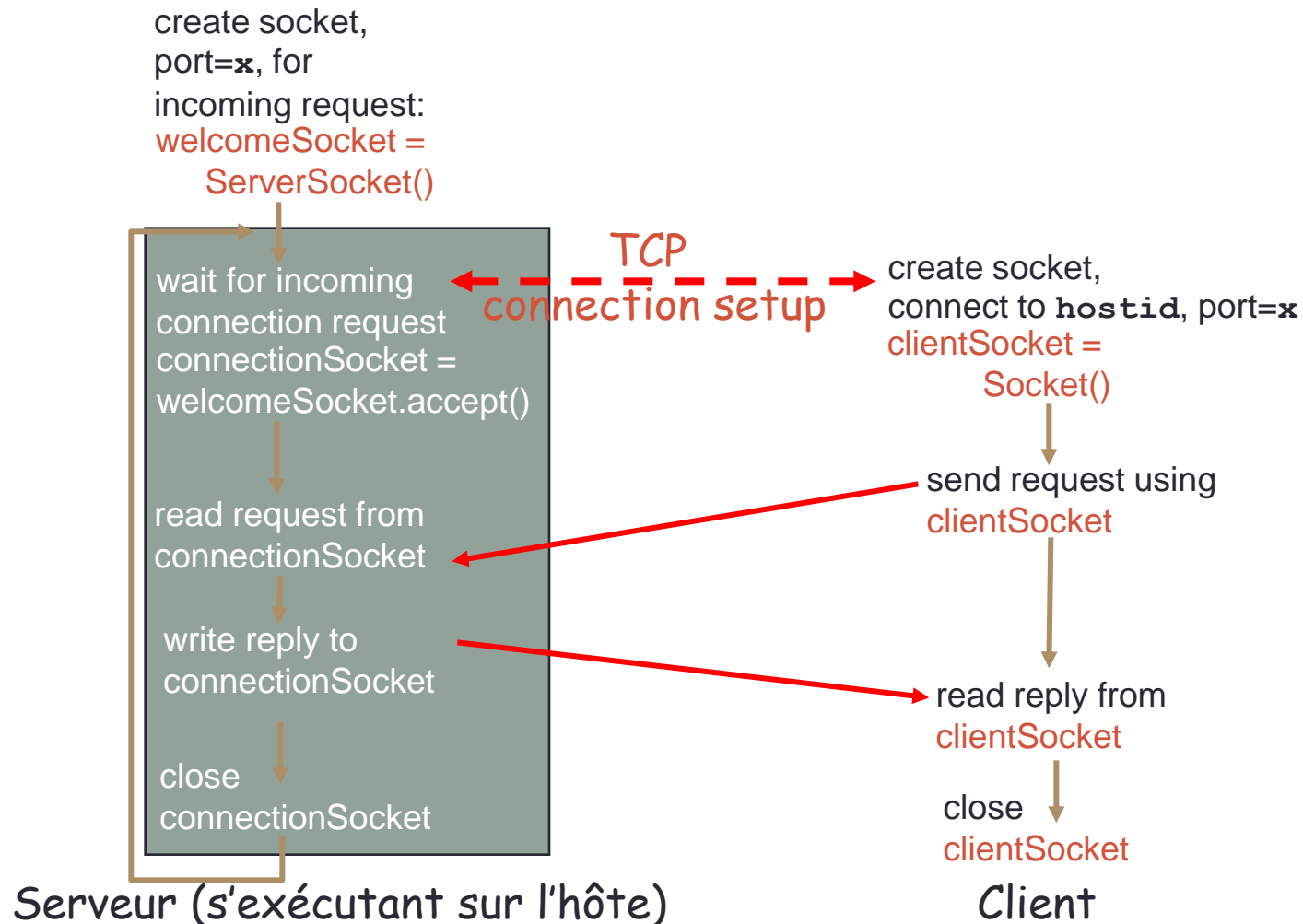
...



APPLICATIONS DISTRIBUÉES ET PARALLÉLISME

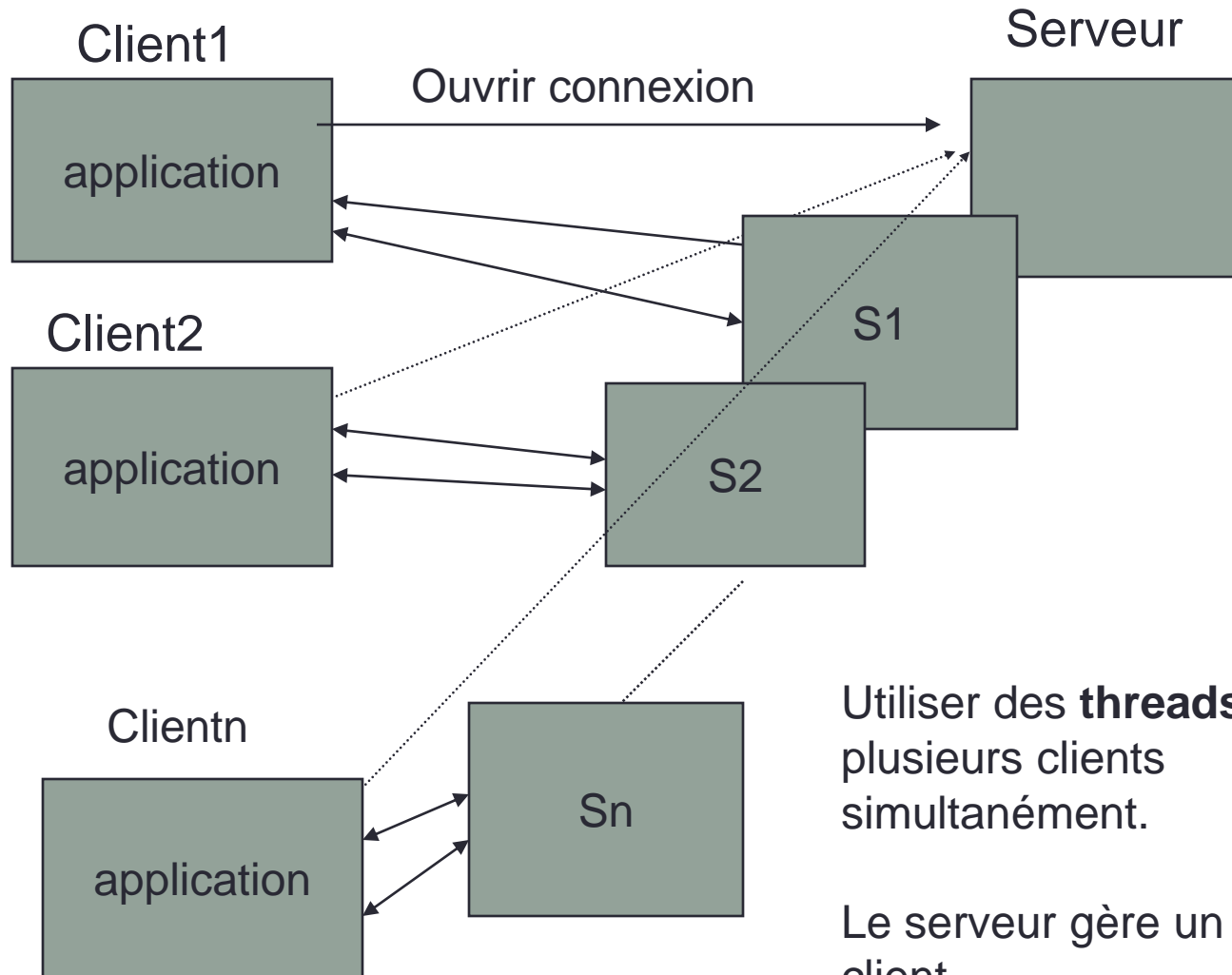
La communication ne doit pas rester bloquée
pour un client

Interaction Client/server : socket TCP





Plusieurs clients



Utiliser des **threads** pour accepter plusieurs clients simultanément.

Le serveur gère un thread par client



Quelques mots sur les Threads

Un thread permet l'exécution d'un programme.

Une application peut avoir de multiples threads qui s'exécutent concurremment (Chaque thread a une priorité).

Chaque thread a un nom. Plusieurs threads peuvent avoir le même. Le nom est généré si non spécifié.

Il y a 2 façons de créer un thread d'exécution.

déclarer une sous classe de **Thread** et surcharger la méthode **run**. Une instance de la sous classe peut alors être allouée et démarrée.

déclarer une classe qui implémente **Runnable** et donc la méthode **run**. Une instance de la classe peut être allouée, passée comme argument à la création d'un thread et démarrée.

Scénario du Serveur Multithreadé

```
while (true)
{ accept a connection ;
  create a thread to deal with the client ;
end while
```

```
public class MultiServerThread extends Thread {
```

```
    private Socket socket = null;
```

```
    public MultiServerThread(Socket socket) {  
        super("MultiServerThread");  
        this.socket = socket; }  
}
```

```
    public void run()
```

```
    { try { PrintWriter out = new  
        PrintWriter(socket.getOutputStream(), true);  
        BufferedReader in = new BufferedReader( new  
        InputStreamReader( socket.getInputStream()));  
        ..... }  
    }
```

```
    out.close(); in.close(); socket.close(); }
```

```
    catch (IOException e) { e.printStackTrace(); } }
```

```
public class MultiServer {
```

```
    public static void main(String[] args) throws  
        IOException
```

```
{    ServerSocket serverSocket = null;
```

```
    boolean listening = true;
```

```
    try { serverSocket = new ServerSocket(4444); }
```

```
    catch (IOException e)
```

```
        { System.err.println("Could not listen on port:  
        4444."); System.exit(-1); }
```

```
    while (listening)
```

```
        new
```

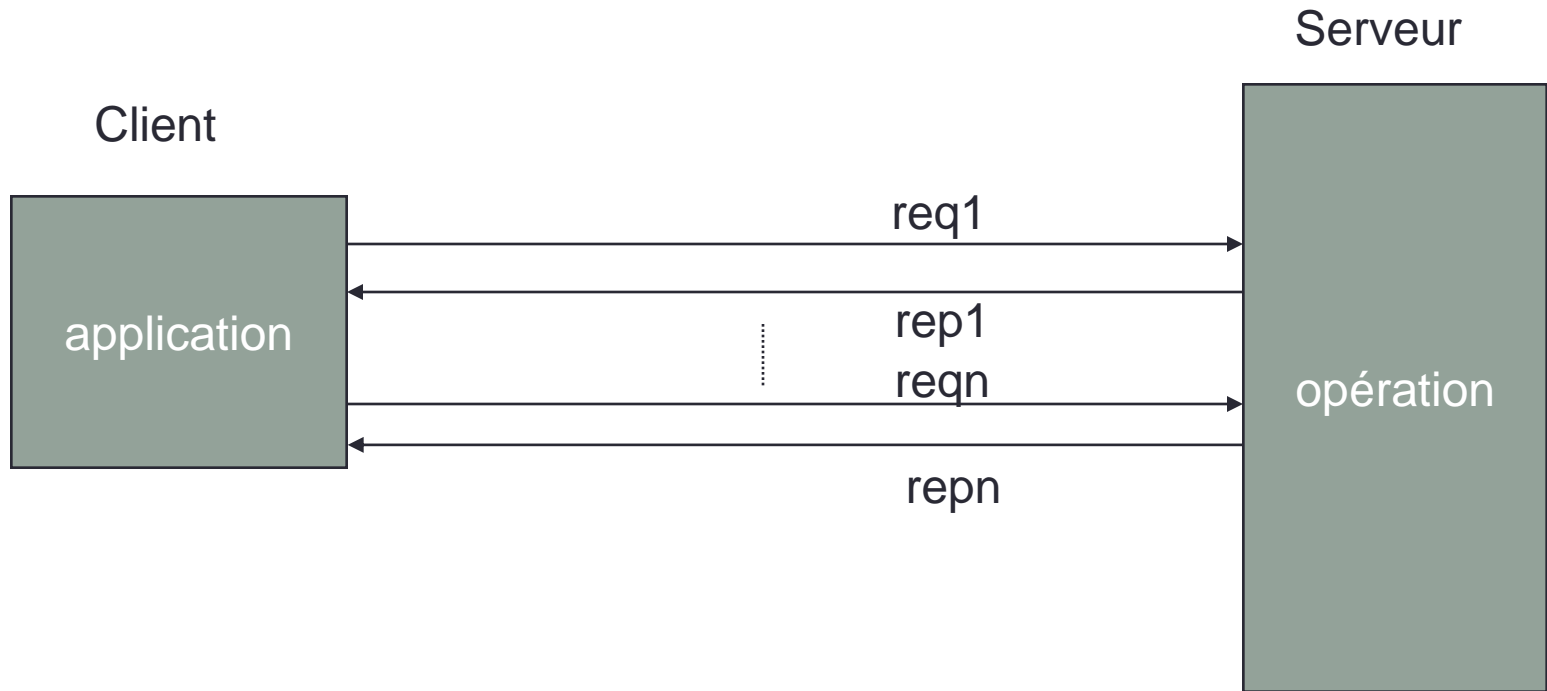
```
        MultiServerThread(serverSocket.accept()).start();  
        serverSocket.close(); } }
```

COMMUNICATION ASYNCHRONE PAR MESSAGES COMMUNICATION PAR DIFFUSION





Communication par message : Envoi de datagrammes



Programmation Socket avec UDP

UDP: pas de “connexion” entre le client et le serveur

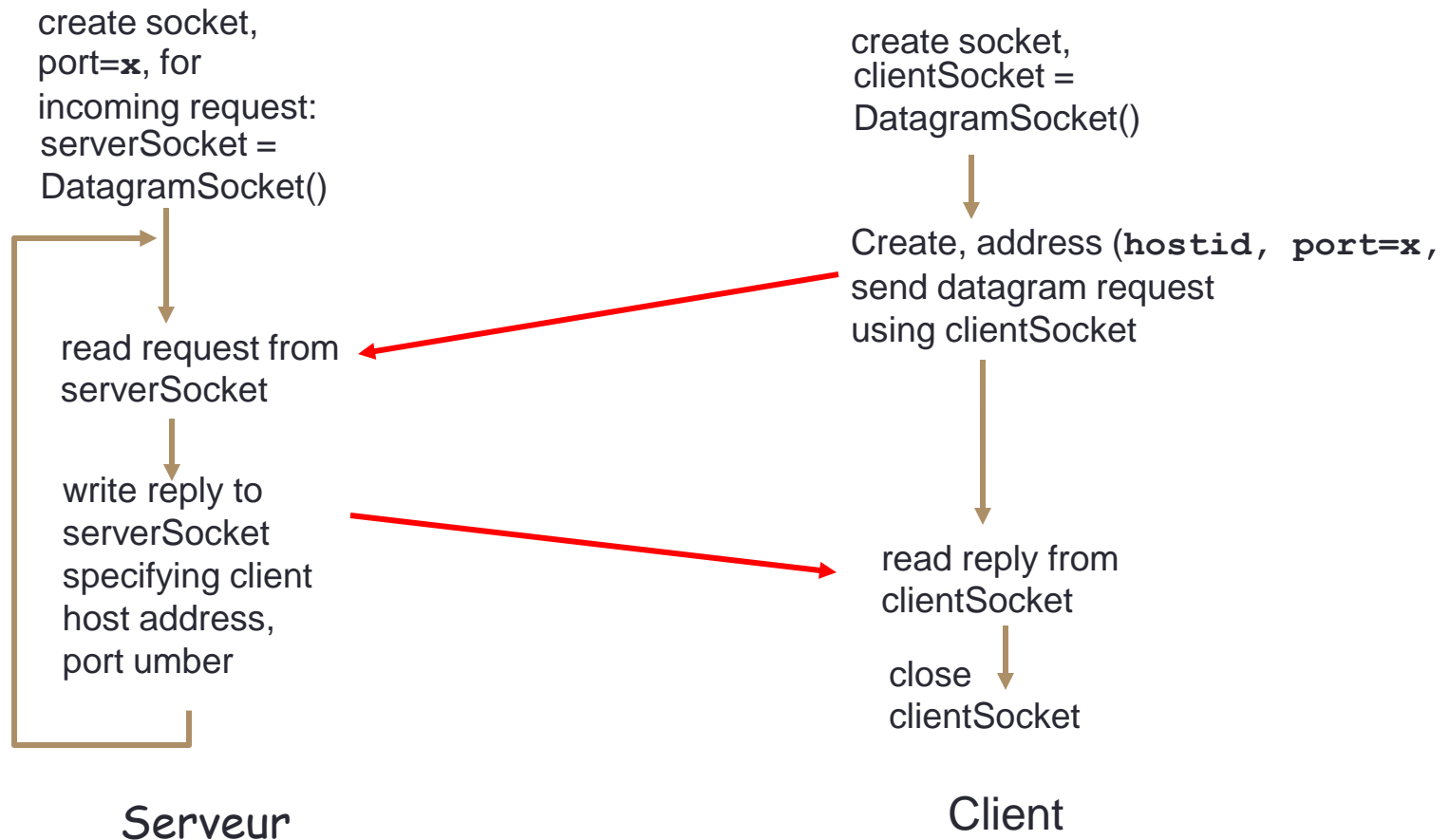
- Pas de lien privilégié entre le client et le serveur
- L'émetteur attache l'adresse IP et le port pour le retour.
- Le serveur doit extraire l'adresse IP et le port de l'expéditeur à partir du datagramme reçu

application viewpoint

UDP fournit un transfert
non fiable de groupes d'octets
("datagrammes")
entre un client et le serveur

UDP: les données transmises peuvent être reçues dans le désordre ou perdues

Client/server socket interaction: UDP





Datagrammes UDP et Sockets

Datagramme = un message **indépendant** envoyé sur le réseau
arrivée, temps d'arrivée et contenu **non garantis**

2 classes : **DatagramPacket** et **DatagramSocket**

packages d'implémentation de
communication via UDP de datagrammes



Exemple

Un serveur de citation qui écoute un socket type datagram et envoie une citation si le client le demande

Un client qui fait simplement des requêtes au serveur

ATTENTION Plusieurs firewalls et routeurs sont configurés pour interdire le passage de paquets UDP



Une Application Client Serveur

Le serveur reçoit en continu
des paquets mode datagramme sur un socket
un paquet reçu = une demande de citation d'un client
le serveur envoie en réponse un paquet qui contient une ligne "quote of
the moment"

L'application cliente envoie simplement un paquet
datagramme au serveur indiquant qu'il souhaite
recevoir une citation et attend en réponse un paquet
du serveur.



La classe QuoteServer

```
socket = new DatagramSocket(4445);
```

Création d'un DatagramSocket sur le port 4445 qui permet au serveur de communiquer avec tous ses clients

```
try { in = new BufferedReader(new FileReader("one-liners.txt"));  
    } catch (FileNotFoundException e)  
    { System.err.println("Couldn't open quote file. " + "Serving time instead.");  
    }  
}
```

Le constructeur ouvre aussi un BufferedReader sur un fichier qui contient une liste de citations (une citation par ligne)



suite

contient une boucle qui tant qu'il y a des citations dans le fichier attend l'arrivée d'un DatagramPacket correspondant à une requête client

sur un DatagramSocket.

```
Byte[] buf = new byte[256];  
DatagramPacket packet = new DatagramPacket(buf, buf.length);  
socket.receive(packet);
```

En réponse une citation est mise dans un DatagramPacket et envoyée sur le DatagramSocket au client demandeur.

```
String dString = null;  
if (in == null) dString = new Date().toString();  
else dString = getNextQuote();  
buf = dString.getBytes();  
InetAddress address = packet.getAddress();  
int port = packet.getPort();  
packet = new DatagramPacket(buf, buf.length, address, port);  
socket.send(packet);
```



Suite

Adresse Internet + numéro de port (issus du DatagramPacket)
= identification du **client** pour que le serveur puisse lui répondre

L'arrivée du DatagramPacket implique une requête
->contenu du buffer inutile

Le constructeur utilisé pour le DatagramPacket :
un tableau d'octets contenant le message et la taille du tableau
+ L'adresse Internet et un no de port.

Lorsque le serveur a lu toutes les citations
on ferme le socket de communication.

`socket.close();`



La classe QuoteClient

envoie une requête au QuoteServer,
attend la réponse
et affiche la réponse à l'écran.

Variables utilisées :

```
int port;  
InetAddress address;  
DatagramSocket socket = null;  
DatagramPacket packet;  
byte[] sendBuf = new byte[256];
```

Le client a besoin pour s'exécuter du nom de la machine sur laquelle tourne le serveur

```
if (args.length != 1) {  
    System.out.println("Usage: java QuoteClient <hostname>");  
    return;  
}
```



La partie principale du main

Création d'un DatagramSocket

```
DatagramSocket socket = new DatagramSocket();
```

Le constructeur lie le Socket à un port local libre

Le programme envoie une requête au serveur

```
byte[] buf = new byte[256];
```

```
InetAddress address = InetAddress.getByName(args[0]);
```

```
DatagramPacket packet = new DatagramPacket(buf, buf.length,  
                                             address, 4445);
```

```
socket.send(packet);
```

Ensuite le client récupère une réponse et l'affiche

Avez-vous suivi ?

Quel est le protocole d'application pour le service de citation ?
peut-on le changer ?

Quel est l'entrée du service sur la machine hôte ?
peut-on la changer ?

Quelle est le marshalling associé au protocole d'application ?
peut-on le changer ?

Quel est le protocole de transport ?
peut-on le changer ?



Classe DatagramSocket

Des constructeurs : par défaut, + no port + Adresse Inet

Des accesseurs en lecture : adresse à laquelle le socket est lié, est connecté, le no port auquel il est lié, connecté, taille du buffer reçu ou envoyé (`getInetAddress`, `getLocalAddress`, `getPort`, `getLocalPort`, `getReceivedBufferSize`, `getSendBufferSize`...)

Des méthodes : pour se connecter à une adresse, pour se déconnecter, pour envoyer un paquet datagramme, pour un recevoir un paquet datagramme (`connect`, `disconnect`, `send`, `receive`)



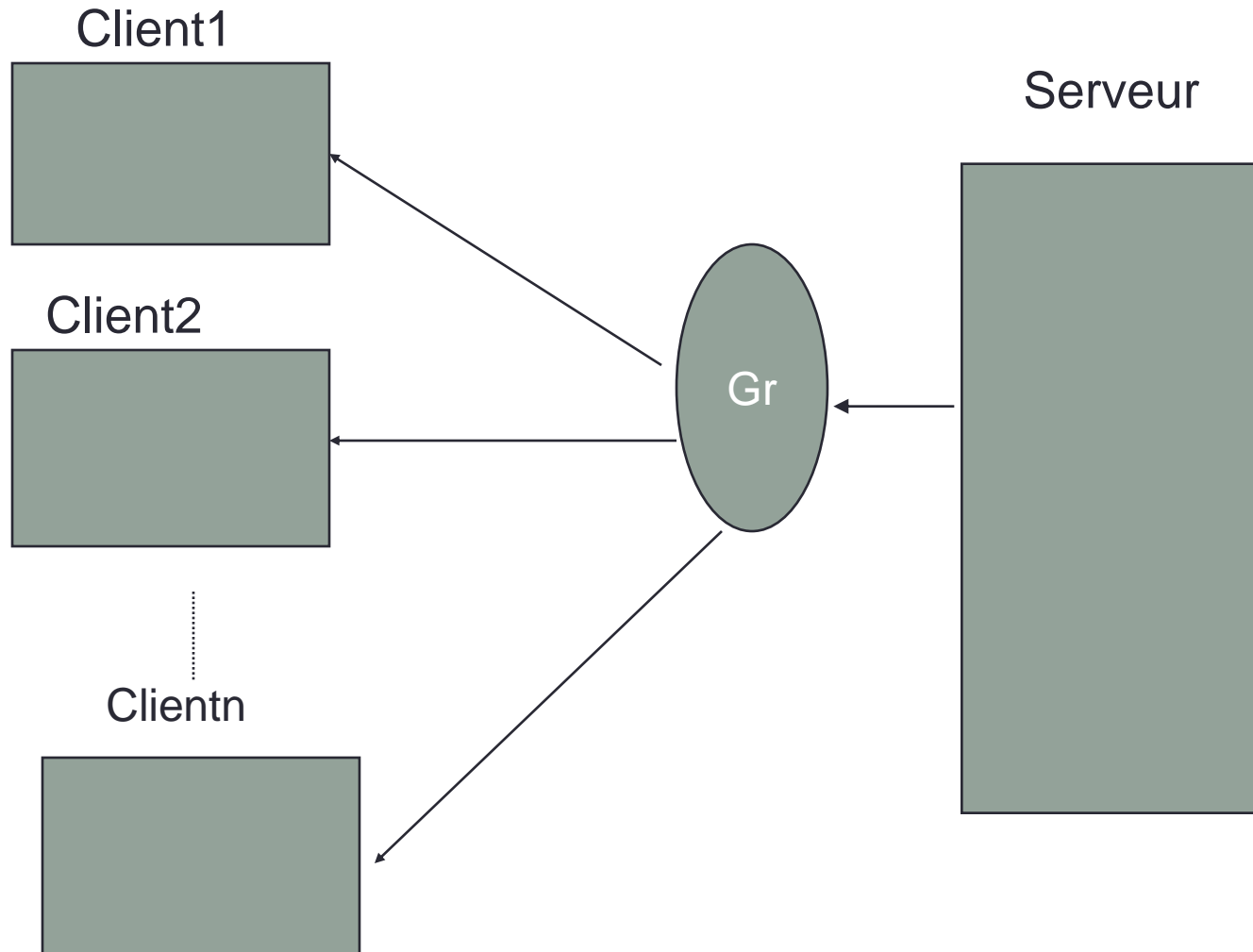
Classe DatagramPacket

Des constructeurs : buffer + longueur de buffer + adresse destination + port...

Des accesseurs en lecture : adresse à laquelle le paquet est envoyé, le no port à laquelle le paquet est envoyé, la donnée transmise (**getAddress, getPort, getData, getLength...**)



Communication par diffusion : Multicast





Ouvrir un socket = demander à se Connecter

Les clients demandent seulement à joindre un groupe

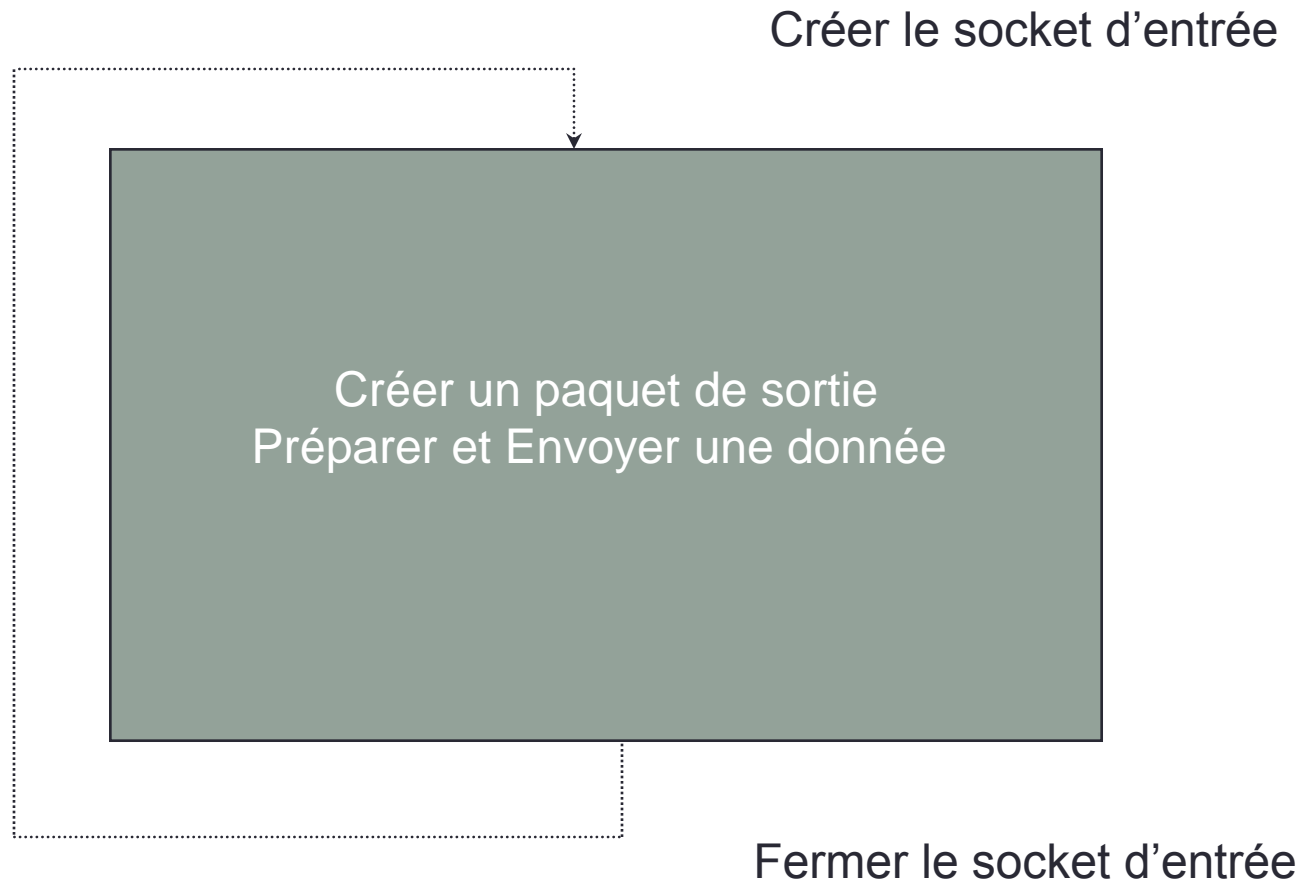


Exemple de multicast

Un serveur de citation qui envoie une citation toutes les minutes à tous les clients qui écoutent (multicast)



Scénario d'un serveur





Scénario d'un client

Créer le socket d'entrée

Création d'un paquet d'entrée
Attente de données en entrée
Réception et traitement des données en entrée

Fermer le socket d'entrée



Classe MulticastSocket

Des constructeurs : par défaut, port à utiliser

Des accesseurs en lecture : adresse du groupe (**getInterface...**)

Des méthodes : pour envoyer un paquet datagramme, pour joindre ou quitter un groupe (**send, joinGroup, leaveGroup**)



Multicast: MulticastSocket

Type de socket utilisé côté client pour écouter des paquets que le serveur « broadcast » à plusieurs clients. .

Une extension du QuoteServer :
broadcaste à intervalle régulier à tous ses clients



Cœur du serveur

```
while (moreQuotes) {  
    try { byte[] buf = new byte[256];  
        // don't wait for request...just send a quote  
        String dString = null;  
        if (in == null) dString = new Date().toString();  
        else dString = getNextQuote();  
        buf = dString.getBytes();  
        InetAddress group = InetAddress.getByName("230.0.0.1");  
        DatagramPacket packet;  
        packet = new DatagramPacket(buf, buf.length, group, 4446);  
        socket.send(packet);  
        try {sleep((long)Math.random() * FIVE_SECONDS);  
            } catch (InterruptedException e) { }  
    } catch (IOException e) { e.printStackTrace();  
        moreQuotes = false;}  
} socket.close();}
```



Différences principales

Le DatagramPacket est construit à partir de
de « l'adresse de plusieurs clients »
L'adresse et le no de port sont câblés

no de port 4446 (tout client doit avoir un MulticastSocket lié à ce no).
L'adresse InetAddress "230.0.0.1" correspond à un identificateur de
groupe et non à une adresse Internet de la machine d'un client

Le DatagramPacket est destiné à tous les clients qui écoutent le port 4446
et qui sont membres du groupe "230.0.0.1".



```
MulticastSocket socket = new MulticastSocket(4446);
InetAddress group = InetAddress.getByName("230.0.0.1");
socket.joinGroup(group);
DatagramPacket packet;
for (int i = 0; i < 5; i++) {
    byte[] buf = new byte[256];
    packet = new DatagramPacket(buf, buf.length);
    socket.receive(packet);
    String received = new String(packet.getData());
    System.out.println("Quote of the Moment: " + received);
}
socket.leaveGroup(group);
socket.close();
```



Un nouveau Client

Pour écouter le port 4446, le programme du client doit créer son MulticastSocket avec ce no.

Pour être membre du groupe "230.0.0.1" le client adresse la méthode joinGroup du MulticastSocket avec l'adresse d'identification du groupe.

Le serveur utilise un DatagramSocket pour faire du broadcast à partir de données du client sur un MulticastSocket. Il aurait pu utiliser aussi un MulticastSocket.

Le socket utilisé par le serveur pour envoyer le DatagramPacket n'est pas important. Ce qui est important pour le broadcast est d'adresser l'information contenue dans le DatagramPacket, et le socket utilisé par le client pour l'écouter.



Synthèse

