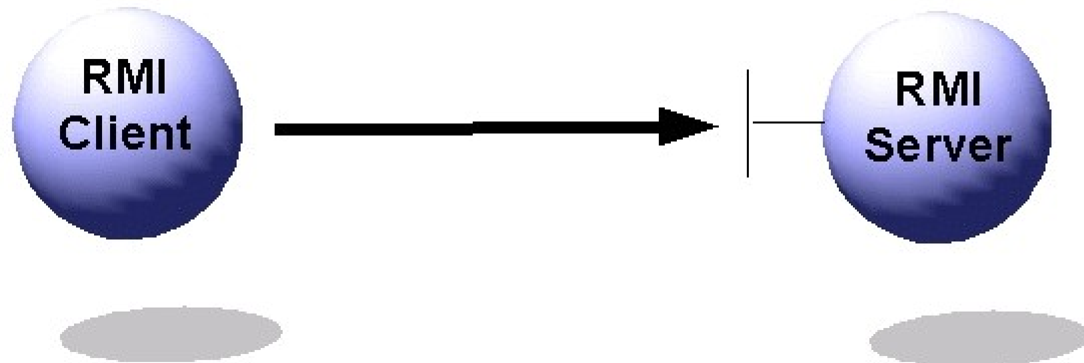
# Programmation répartie
# RPC & RMI



# Plan du cours

- **Introduction**
  - Définitions
  - Problématiques
  - Architectures de distribution
- **Distribution intra-applications**
  - Notion de processus
  - Programmation multi-thread
- **Distribution inter-applications et inter-machines**
  - sockets
  - Middlewares : de RPC vers RMI
  - middlewares par objets distribués (CORBA)
- **Conclusion**

# Les sockets : primitives de communication

- **On a vu :**
  - Les différentes formes de communication
    - Synchrone
    - Asynchrone
    - Par RDV
  - Et leurs implications sur les applications
    - Intégration des actions de communication dans la sémantique opérationnelle des applications
    - Leurs conséquences tels que le blocage mort et/ou vivant
- **Illustration sur les Sockets sous Java**
  - { } de classes java qui permettent de générer des connexions entre les applications et d'échanger un flux de données
  - Une classe pour le serveur ***`ServerSocket`***
  - Une classe pour définir un canal asynchrone **`Socket`**
  - => forme de communication par RDV (notion d'entrée)

# Quelques constatations

- **On a considéré implicitement un mode d'interaction avec le serveur:**
- **Mode d'interaction : on distingue deux choses**
  - Le protocole
  - Le contenu
  - Cas du TD:
    - protocole
      1) Un client se connecte au serveur
      2) Ensuite il envoit un message
      3) Après il envoit et reçoit
    - Contenu
      - Dans 2) le contenu du message sera interprété comme le pseudonyme
      - Et tout ce qui en suit comme des messages pour le tchat.
- **Les données sont échangées sous format de flux et c'est au récepteur de savoir le type et le sens (interprétation).**
- **=> On ne distingue plus l'utilisation des Socket dans un code C ou Java**
  - On a perdu complètement l'aspect orienté objet ou même toute autre structuration telles que les procédures.

# Le Web

- **Le Web est un vaste parc de serveurs et de clients on y distribue des ressources sous plusieurs formats.**
  - Image, son, html etc.
- **Chaque ressource disponible possède sa propre identification URI.**
  - URL ou URN
- **URL est une adresse d'une ressource particulière. On y trouve**
  - l'adresse du serveur qui détient la ressource
  - L'adresse relative de la ressource sur le serveur
  - **Le protocole** que le serveur utilise pour fournir cette ressource
- **Exemple de protocol :**
  - SMTP on a déjà vue en TD
  - HTTP, FTP
  - ....

# Exemple de HTTP

- **C'est un protocole normalisé de communication entre le navigateur et le serveur Web (client-serveur)**
- **Il fonctionne selon trois étapes (protocole):**
  - Établissement d'une connexion TCP avec le serveur sur le port 80 ou autre
  - Envoie d'un message au serveur pour lui réclamer un service (requête).
  - Réception de la réponse
- **Les requêtes sur HTTP peut être de deux formes (contenu)**
  - Réclamer une ressource **`Get`** `adresseRelative version_http`
  - Déposer un certain nombre de données pour faire un traitement `Post formatAttributValeur ressourceDeTraitement`
- **La réponse est un objet multimédia**
  - HTTP comme FTP et d'autres se basent sur un typage MIME

# C'est quoi MIME

- C'est un protocole de transfert de données.
- Il offre un système unique d'échange de données typées.
- Grâce à MIME, le navigateur ou le client Mail sont capables de distinguer des octets correspondant à une image de celle d'une suite d'instructions postscript
- Le typage MIME est utilisé par le protocole HTTP pour pour l'encodage des ressources.
- Le protocole HTTP spécifie un attribut pour le type mime du contenu de la requête ou encore de la réponse.

# Analyse du cas du Web

- **On a des clients (navigateurs) et des serveurs.**
- **Les serveurs peuvent rendre des services plus ou moins complexes.**
- **Quel sont les éléments essentiels pour connecter deux applications ?**
- **Cas de HTTP**
  - Mise en place d'un protocole adapté à la nature de l'échange (orienté message requête-réponse).
    - Ensemble de convention du format, du message et leurs signification
  - Mise en place d'une convention d'encodage de données échangées (Typage MIME)
  - Un mécanisme d'identification d'une application distante (URI,....)

# Pour une application donnée

- **Quel est le format de services que l'application peut offrir**
  - Cas du Web : des ressources
  - Cas des langages procédureaux : une procédure
  - Cas de l'orienté objet : les méthodes d'un objet.
- **Quel sera la nature de l'échange pour réaliser le service**
  - Protocole :  séquence plus contenu
- **Une méthode de transfert des artéfacts locaux à chacune des applications**
- **Un mécanisme de localisation de services offerts par les applications**
- **Un mécanisme d'amorçage qui permet une transparence**

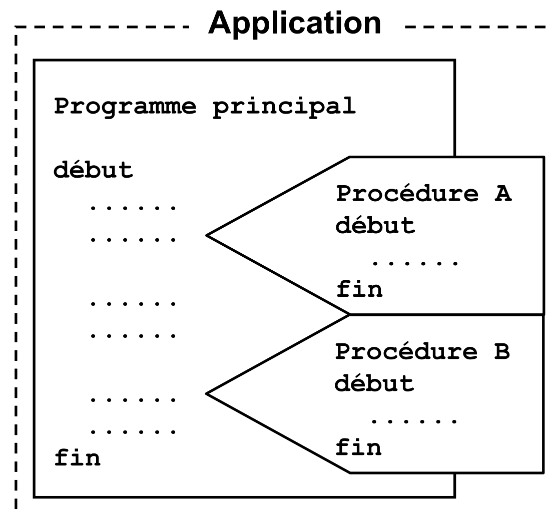
# Objectif de ce cours

***Etudier les mécanismes de mise en place des services middellware ou intergiciel permettant la connexion entre deux applications.***

- **Cas des langages procéduraux**
  - RPC : Remote Procedure Call
- **Cas de l'orienté Objet (java)**
  - RMI: Remote Method Invocation

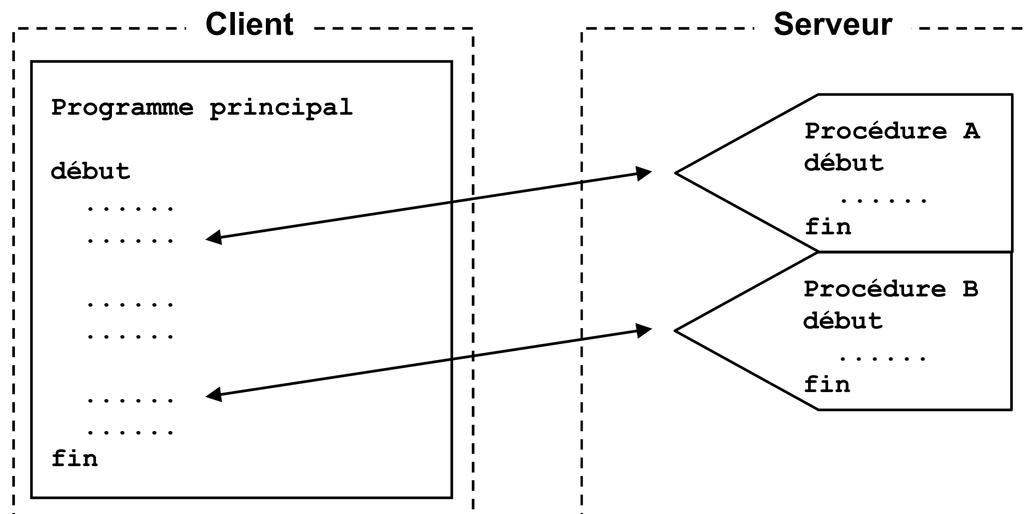
# Programmes et appel de procédures

- **Applications structurées en 2 parties**
  - le programme principal
  - un ensemble de procédures
- **Principe de fonctionnement**
  - programme principal et procédures sont compilés et liés
  - au moment de l'exécution
    - le programme principal appelle les procédures en transmettant des paramètres d'entrée
    - les procédures s'exécutent et retournent leurs résultats dans les paramètres de sortie



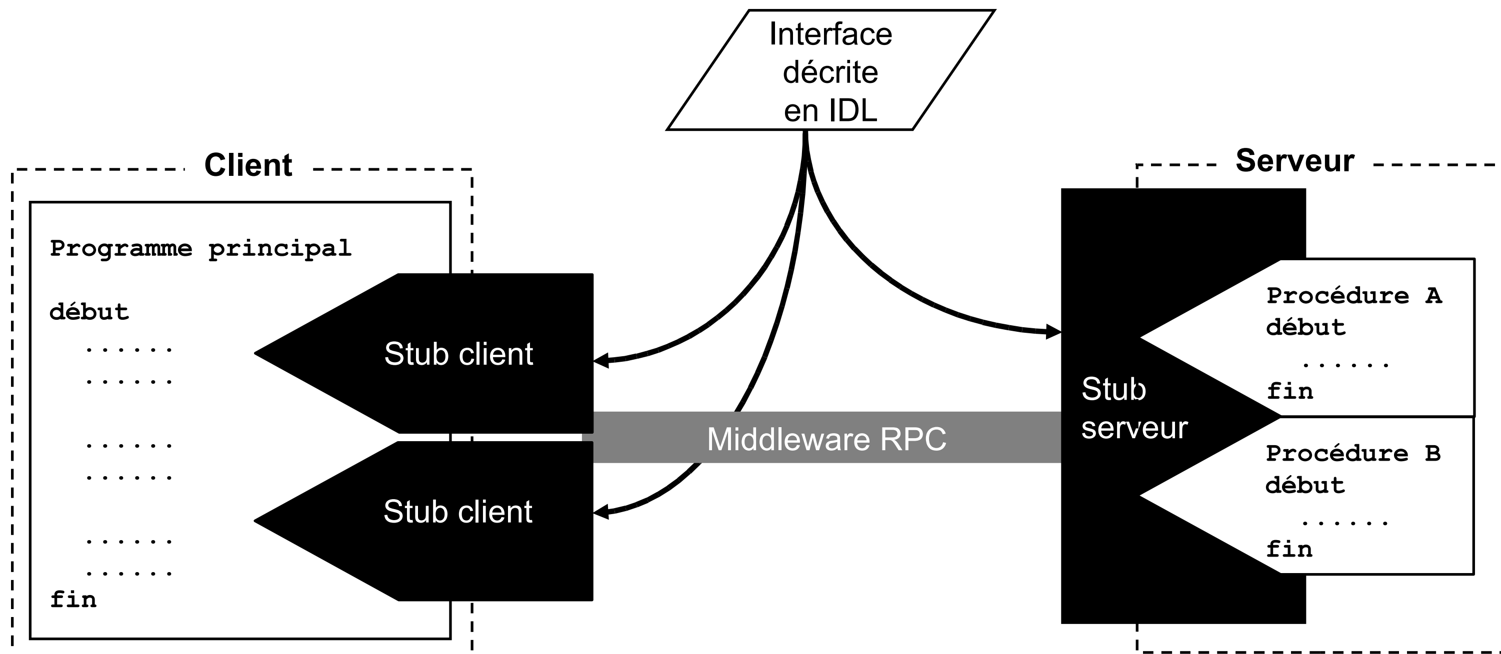
# Analogie avec le client-serveur

- le programme principal se comporte comme un client
- l'ensemble des procédures est assimilable à un ensemble de services disponibles sur un serveur
  - interface d'un service = signature de la procédure
  - interface du serveur = ensemble des signatures des procédures

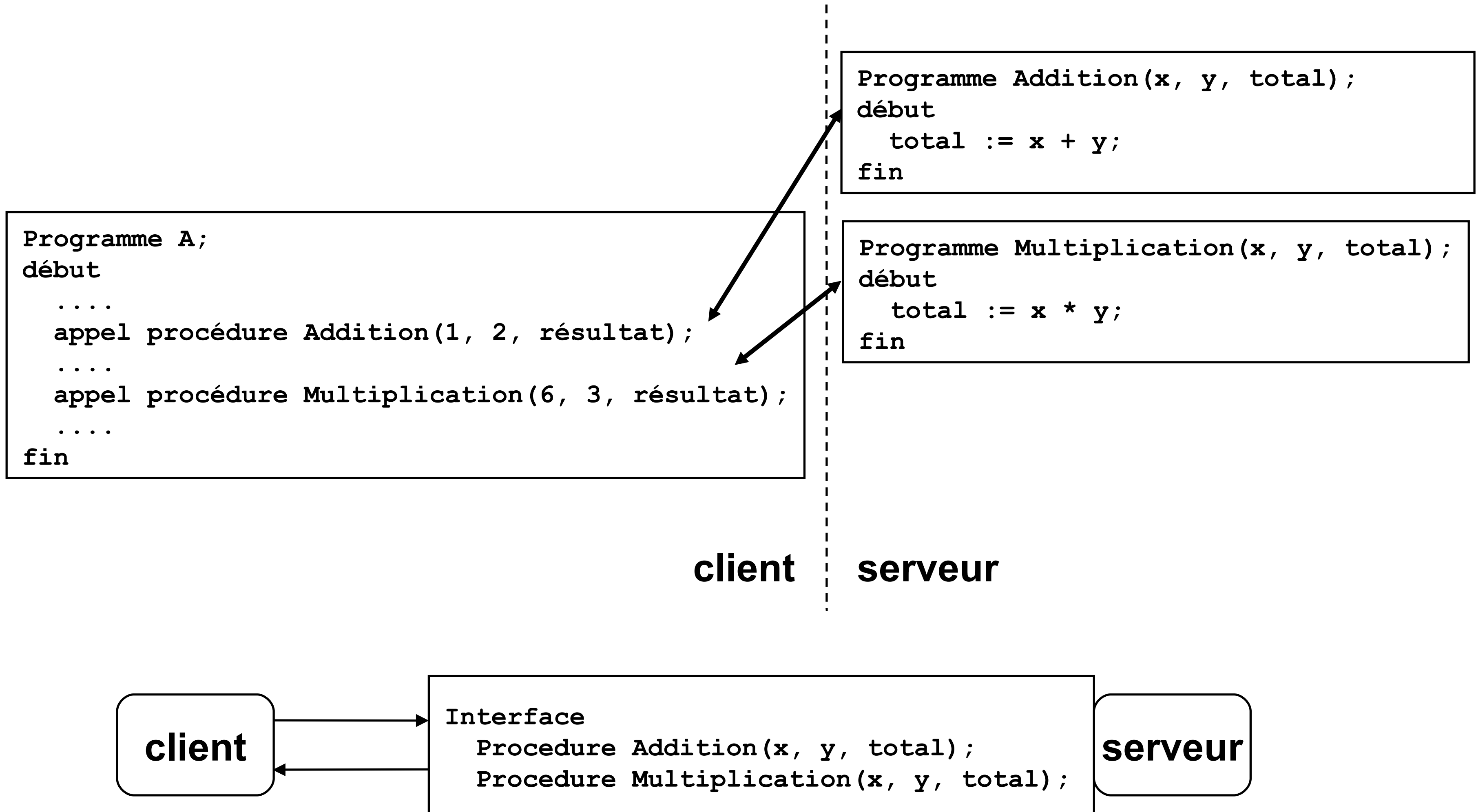


# Middleware par appel de procédures distantes

- **Assure la transparence de localisation**
  - le client appelle les procédures comme si elles étaient locales
  - le middleware assure la communication avec le serveur
  - l'interface du serveur est décrite en IDL
  - le code de préparation d'une requête (stub client) est généré automatiquement à partir de la description IDL



# Exemple : calcul de fonction mathématiques



# Le client RPC

- **Lors de la compilation et de l'édition des liens...**
  - erreur car les procédures ne sont pas présentes
  - procédures émulées par 2 fausses procédures appelées *stubs client*
- **Le *stub client***
  - procédure qui porte le nom de la vraie procédure qu'il remplace
  - donne l'illusion au programme principal que la procédure est bien locale
  - remplace le code de la vraie procédure par un autre code
    - gère la connexion avec le bus middleware
    - transmet les paramètres vers la machine où se trouve la procédure
    - récupère le(s) résultat(s)

# Le serveur RPC

- **Impossible d'avoir un programme exécutable composé uniquement de procédures**
  - nécessité d'un programme principal appelé *stub serveur*
- **Le *stub serveur***
  - permet de créer un exécutable contenant les procédures du serveur
  - gère la communication avec les *stubs client*
    - active la procédure désignée en lui transmettant les paramètres d'appel
    - retourne les valeurs de résultat au stub client

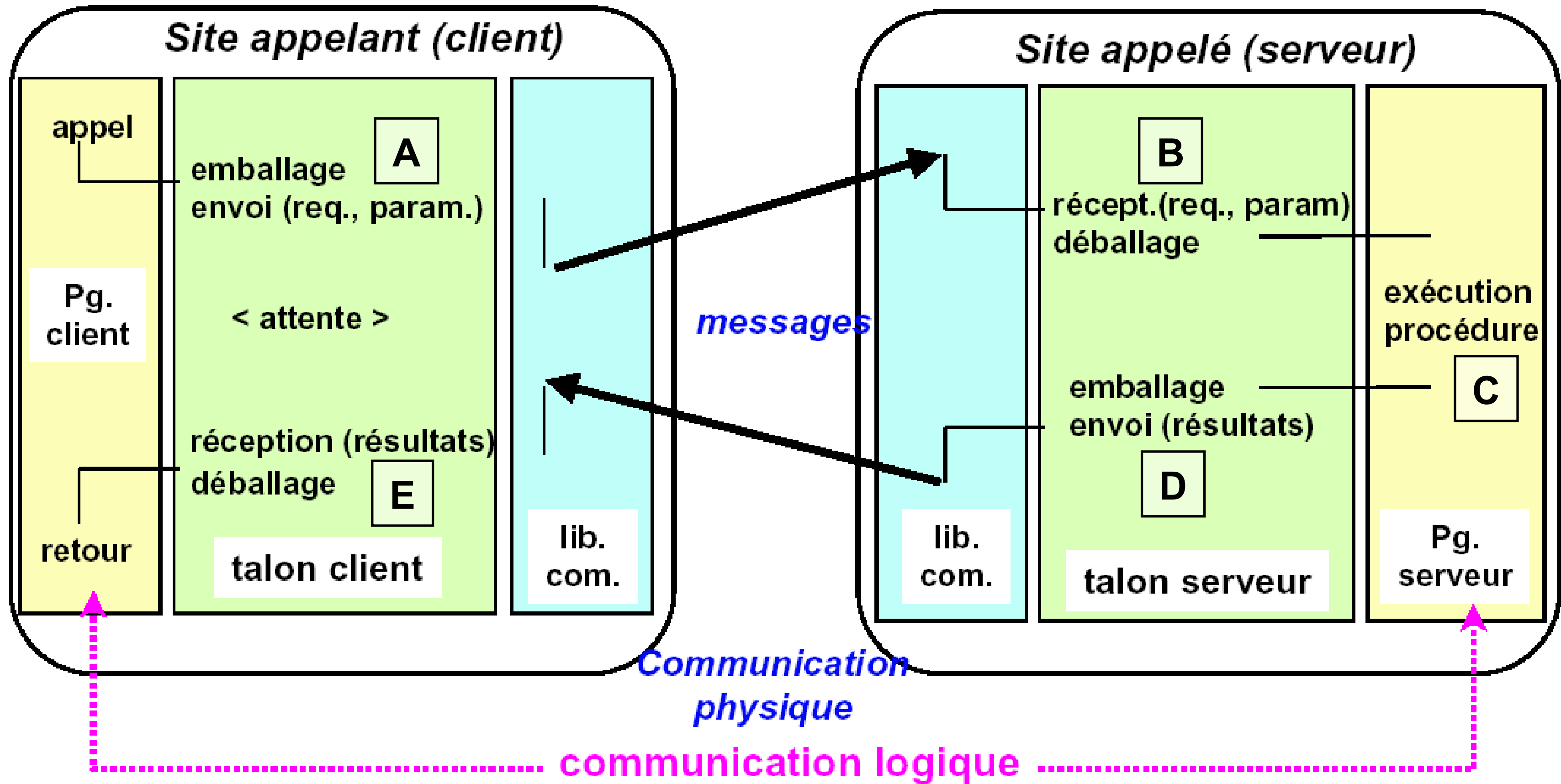
# Caractéristiques

- codes du client et du serveur indépendants du système de communication; le client ne sait pas si la procédure est locale ou éloignée
- le code du client n’a pas à préparer le message ni à localiser le serveur => à la charge du middleware RPC
- système de dialogue entièrement externe au client et au serveur décrit dans un langage spécifique (IDL) à partir duquel est généré automatiquement le code nécessaire
- structure de communication construite au moment de la compilation
- communication synchrone → le client attend la réponse à son appel de procédure avant de continuer son traitement
- technologie RPC entièrement standardisée (inclut IDL + services nécessaires à la communication)

# RPC : les difficultés

- **Appel de procédure local**
  - appelant et appelé sont dans le même espace virtuel
  - même mode de pannes
  - appel et retour de procédure sont des mécanismes internes considérés comme fiables
    - sauf aspects liés à la liaison dynamique de la procédure et à la vérification de la protection
  - dans certains langages
    - mécanisme d'exception pour transmettre les erreurs de l'appelé à l'appelant
- **Appel de procédure à distance**
  - Appelant et appelé sont dans 2 espaces virtuels différents
  - pannes du client et du serveur sont indépendantes
  - pannes du réseau de communication (perte du message d'appel ou de réponse)
  - temps de réponse du serveur long
    - charge du réseau ou du site serveur

# RPC [Birrel et Nelson 84] : principe de réalisation



# RPC (A) : principe de fonctionnement

- **Côté de l'appelant**
  - le client réalise un appel procédural vers la procédure talon client
    - transmission de l'ensemble des arguments
  - au point A
    - le talon collecte les arguments et les assemble dans un message (empaquetage - parameter marshalling)
    - un identificateur est généré pour le RPC
    - un délai de garde est armé
    - le talon transmet les données au protocole de transport pour émission sur le réseau. Mais où??
    - pb : détermination de l'adresse du serveur

# RPC (B, C et D) : principe de fonctionnement

- **Côté de l'appelé**
  - le protocole de transport délivre le message au service de RPC (talon serveur/skeleton)
  - au point B
    - le talon désassemble les arguments (dépaquetage - unmarshalling)
    - l'identificateur de RPC est enregistré
    - l'appel est ensuite transmis à la procédure distante requise pour être exécuté (point C). Le retour de la procédure redonne la main au service de RPC et lui transmet les paramètres résultats (point D)
  - au point D
    - les arguments de retour sont empaquetés dans un message
    - un autre délai de garde est armé
    - le talon transmet les données au protocole de transport pour émission sur le réseau

# RPC (E) : principe de fonctionnement

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

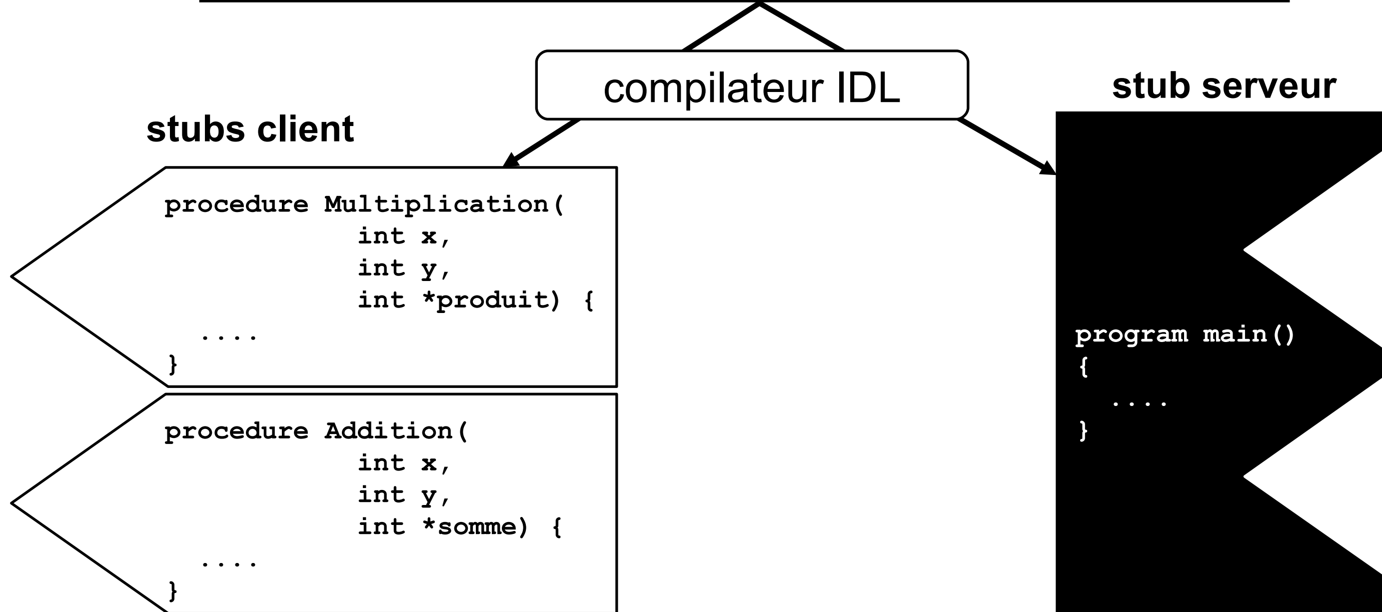
# La notion de contrat entre client et serveur

- **Le contrat**
  - formalise le dialogue entre le client et le serveur
    - permet au client et au serveur d'avoir la même compréhension des échanges effectués
  - répond aux questions
    - que transmet-on ?
    - où envoie-t-on les données ?
    - qui reçoit les données ?
    - comment sait-on que le travail est terminé ?

# Exemple de contrat

```
/* OSF IDL RPC code */
[uuid (xxxxxxxx-xxxx-xxxx-xxxx) version (N.n)]          ❶
interface serveur_mathématique                          ❷
{
  int Addition([in]  int x,                             ❸
               [in]  int y,                             ❹
               [out] int *somme);                       ❺

  int Multiplication([in]  int x,                       ❸
                     [in]  int y,                       ❹
                     [out] int *produit);               ❺
}
```



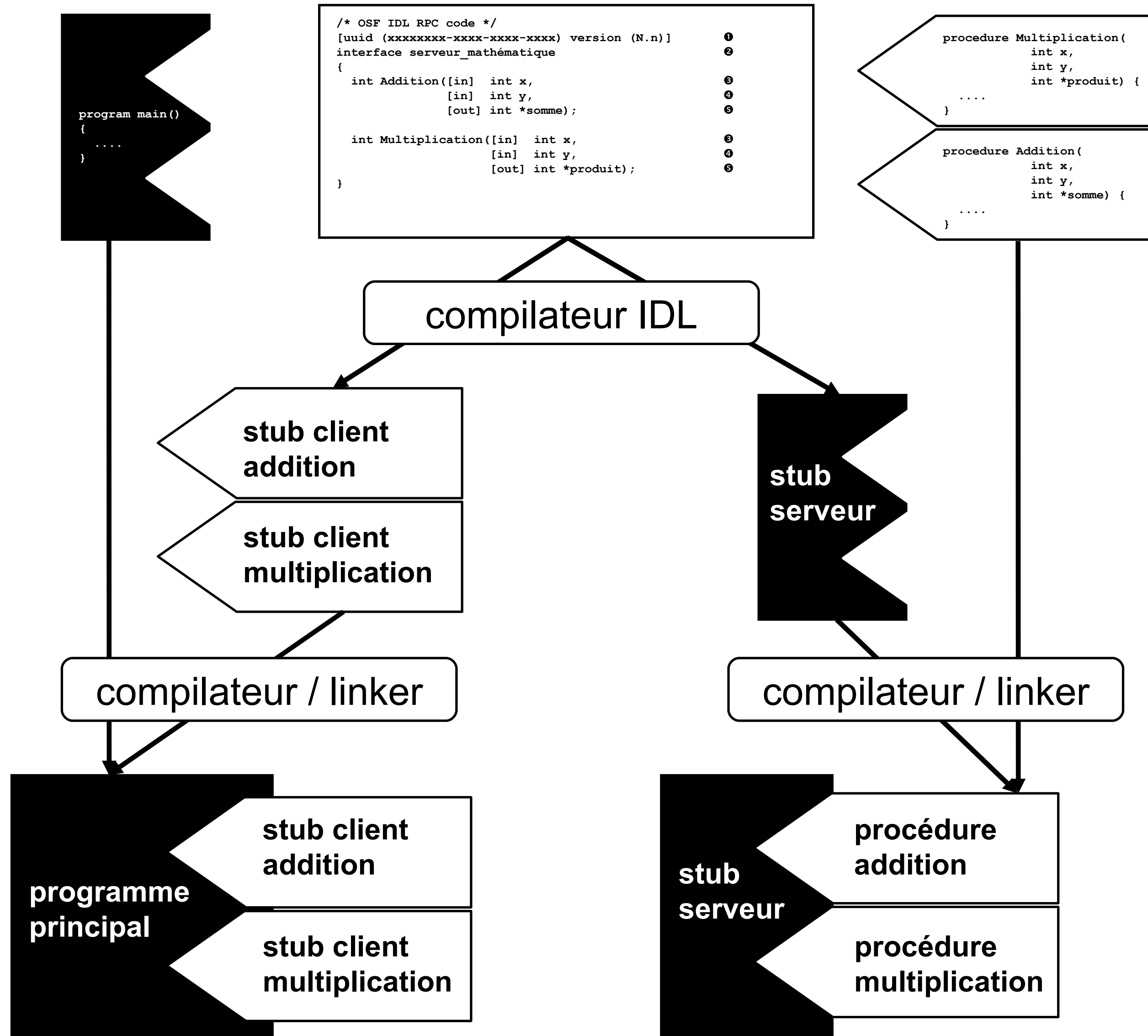
# Caractéristiques du contrat

- **Le contrat**
  - ❶ : est repéré par un n° d'identifiant unique (uuid = universal unique identifier)
    - utilisé lors de chaque requête du client
  - ❷ : définit et nomme l'interface offerte par le serveur
    - contient un sous-ensemble des services disponibles sur le serveur
    - ❸ : définit la signature des services
      - nom de la procédure
      - paramètres caractérisés par leur mode d'utilisation
        - paramètres d'entrée [in] (❸ et ❹)
        - paramètres de sortie [out] (❺)
        - paramètres d'entrée et de sortie [in][ou]
  - est rédigé dans un langage informatique spécifique (IDL = Interface Definition Language)
    - plusieurs versions (OSF IDL RPC pour la technologie RPC)
  - est distribué entre le client et le serveur
- **Le compilateur IDL**
  - génère les stubs client et serveur (le plus souvent en langage C)

# Construction du client et du serveur

- **Eléments à développer**
  - le programme principal de l'application (le client)
    - n'importe quel langage de programmation
    - le plus souvent en C
  - les procédures composant l'application (le serveur)
  - le contrat décrivant les échanges entre client et serveur
    - écrit en langage IDL
- **Etapes de déploiement**
  - générer les stubs client et serveur
    - compilateur IDL
  - construire l'exécutable client
    - compiler le programme principal et les stubs client
    - lier les deux
  - construire l'exécutable serveur
    - compiler le stub serveur et les procédures
    - lier les deux

# Construction du client et du serveur



# Représentation des données

- **Problème classique dans les réseaux**
  - Conversion nécessaire si le site client et le site serveur
    - n'utilisent pas le même codage
      - caractères (ASCII, EBCDIC, Unicode)
      - taille des mots mémoire (16, 32, 64 bits)
      - numérotation des octets dans un mot mémoire (big endian, little endian)
    - utilisent des formats internes différents (type caractère, entier, flottant, …)
    - solution placée classiquement dans la couche 6 du modèle OSI présentation
  - dans réseau : passage de paramètres par valeur
    - émulation des autres modes

# Passage des paramètres

- **Valeur**
  - pas de problème particulier
- **Copie/restauration**
  - valeurs des paramètres sont recopiées
  - pas de difficultés majeures mais redéfinition des solutions définies pour les réseaux
  - optimisation des solutions pour le RPC
  - bonne adaptation au langage C (faiblement typé)

# Passage des paramètres

- **Référence**
  - utilise une adresse mémoire centrale du site de l'appelant… aucun sens pour l'appelé
  - solutions
    - interdiction totale
      - introduit une différence entre procédures locales et procédures distantes
    - simulation en utilisant une copie de restauration
      - marche dans de très nombreux cas
      - mais violation dans certains cas de la sémantique du passage par référence
      - exemple de pb :

```
procedure double_incr (int *x, int *y) {
     x := x+2;
     y := y+1;
}
a := 0 ;
double_incr (a, a)
```

**résultat : a = 2 ou a = 1 ?**

    - reconstruire l'état de la mémoire du client sur le site serveur
      - solution très coûteuse
    - utilisation d'une mémoire virtuelle répartie
      - nécessite un système réparti avec mémoire virtuelle

# Passage des paramètres

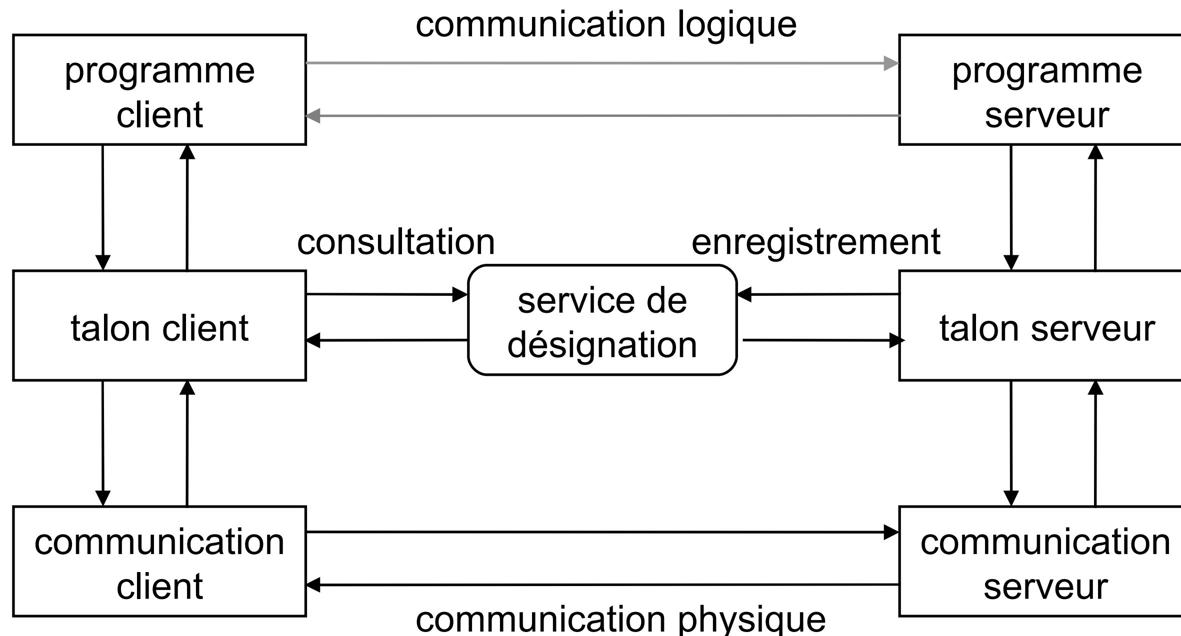
- **Solutions généralement prises**
  - IN : passage par valeur (aller)
  - OUT : passage par valeur (retour)
  - IN-OUT : interdit ou passage par copie-restauration
    - ATTENTION : n 'est pas équivalent au passage par référence

# Désignation

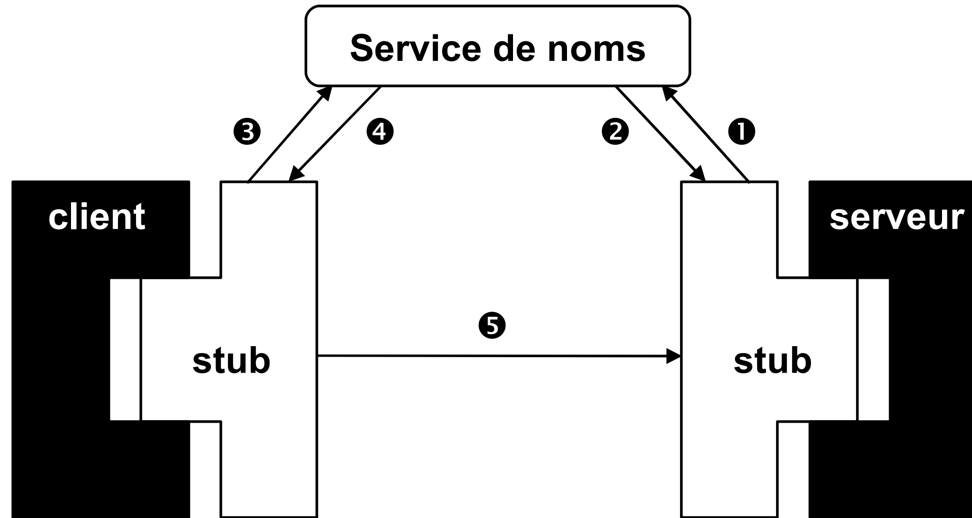
- **Objets à désigner**
  - le site d'exécution, le serveur, la procédure
  - désignation globale indépendante de la localisation
    - possibilité de reconfiguration des services (pannes, régulation de charge, …)
- **Désignation**
  - statique ou dynamique
    - statique : localisation du serveur est connue à la compilation
    - dynamique : non connue à la compilation, objectifs :
      - séparer connaissance du nom du service de la sélection de la procédure qui va l'exécuter
      - permettre l'implémentation retardée

# Liaison et fonctionnement

- **Liaison (détermination de l'adresse du serveur)**
  - liaison statique (pas d'appel à un serveur de nom ou appel lors de la compilation)
  - liaison au premier appel (appel du serveur de nom lors du premier appel)
  - liaison à chaque appel (appel du serveur de nom à chaque appel)



# Liaison - solution classique : DNS Internet




- ❶ **le serveur s'enregistre auprès du serveur de noms**
  - adresse, interfaces
- ❷ **le service de noms confirme au serveur qu'il est connu**
  - le serveur se met en attente de connexions
- ❸ **le client demande au service de noms l'adresse du serveur**

# RPC : les limites

- **Mécanisme de bas niveau**
  - la notion de procédure n’existe pas dans les méthodes d’analyse
- **N’assure pas tous les services souhaités d’un bus de communication**
  - fiabilité du transfert
    - appels perdus si serveur ou réseau en panne
    - gestion des erreurs et des pannes à la charge du client
  - concept de transaction
  - diffusion de messages
    - communication 1-1
    - pas de communication 1-n
- **Outils de développement**
  - limités à la génération automatique des talons
  - peu d’outils pour le déploiement et la mise au point d'applications réparties

# RPC : les problèmes

- **Traitement des défaillances**
  - congestion du réseau ou du serveur
    - la réponse ne parvient pas avant une date fixée par le client (système temps critique)
  - panne du client pendant le traitement de la requête
  - panne du serveur avant ou pendant le traitement de la requête
  - erreur de communication
- **Problèmes de sécurité**
  - authentification du client
  - authentification du serveur
  - confidentialité des échanges
- **Désignation et liaison**
- **Aspects pratiques**
  - adaptation à des conditions multiples (protocoles, langages, matériels)
  - gestion de l'hétérogénéité

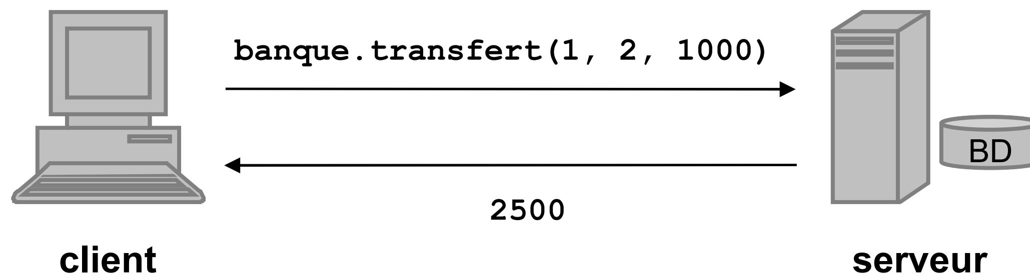
# Cas de la programmation objet : RMI

# RPC « à objets »

- **Motivations**
  - tirer parti des bonnes propriétés de l'objet (encapsulation, modularité, réutilisation, polymorphisme, composition)
  - objet : unité de désignation et de distribution
- **Objets "langage"**
  - représentation propre au langage : instance d'une classe
  - exemple : Java RMI
- **Objets "système"**
  - représentation "arbitraire" définie par l'environnement d'exécution
  - interopérabilité entre objets écrits dans des langages différents
  - exemple : CORBA

# Java RMI (Remote Method Invocation)

- **Eléments d'une "invocation"**
  - référence d'objet ("pointeur" universel)
  - identification d'une méthode
  - paramètres d'appel et de retour (y compris signal d'exception)
    - passage par valeur : types élémentaires et types construits
- **Java RMI**
  - intégré au JDK 1.1
  - gère tous ces détails automatiquement
  - fonctionne sur le modèle client-serveur
  - étend aux objets le principe de RPC



# Caractéristiques attendues

- invoquer une méthode d'un objet se trouvant sur une autre machine exactement de la même manière que s'il se trouvait au sein de la même machine

```
objetDistant.methode();
```

- utiliser un objet distant (OD), sans savoir où il se trouve, en demandant à un service « dédié » de renvoyer son adresse

```
objetDistant = ServiceDeNoms.recherche("monObjet");
```

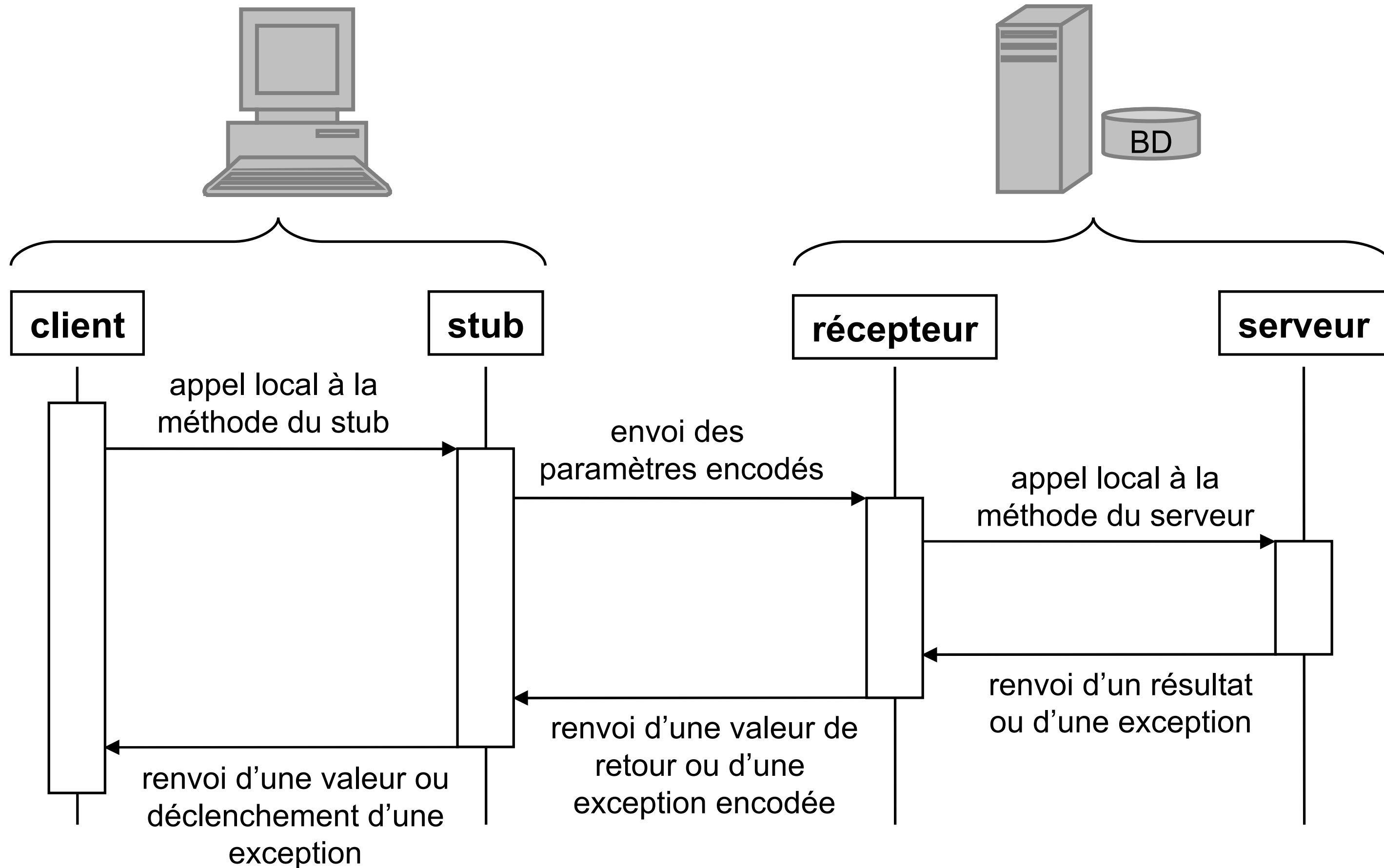
- pouvoir passer un OD en paramètre d'appel à une méthode locale ou distante

```
resultat = objetLocal.methode(objetDistant);
resultat = objetDistant.methode(autreObjetDistant);
```

- pouvoir récupérer le résultat d'un appel distant sous forme d'un nouvel objet qui aurait été créé sur la machine distante

```
NouvelObjetDistant = ObjetDistant.methode() ;
```

# RMI : un RPC « à objets »



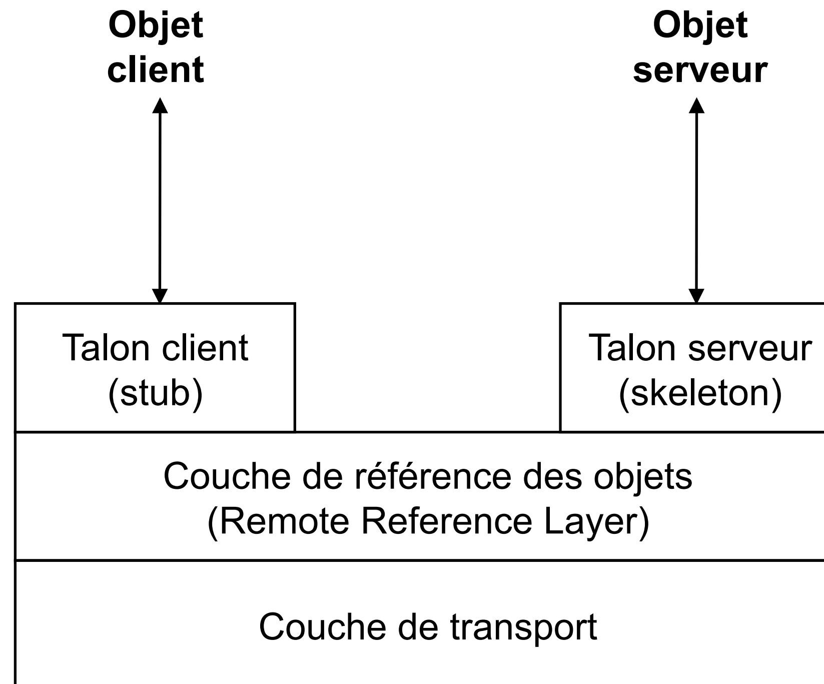
# RMI : Généralités

- **Principe**
  - mécanisme permettant l'appel de méthodes entre objets Java s'exécutant sur des machines virtuelles différentes (espaces d'adressage distincts), sur le même ordinateur ou sur des ordinateurs distants reliés par un réseau
    - utilise directement les sockets
    - code ses échanges avec un protocole propriétaire : R.M.P. (Remote Method Protocol)
- **Objectifs**
  - rendre transparent l'accès à des objets distribués sur un réseau
  - faciliter la mise en œuvre et l'utilisation d'objets distants Java
  - préserver la sécurité (inhérent à l'environnement Java)
    - RMISecurityManager
    - Distributed Garbage Collector (DGC)

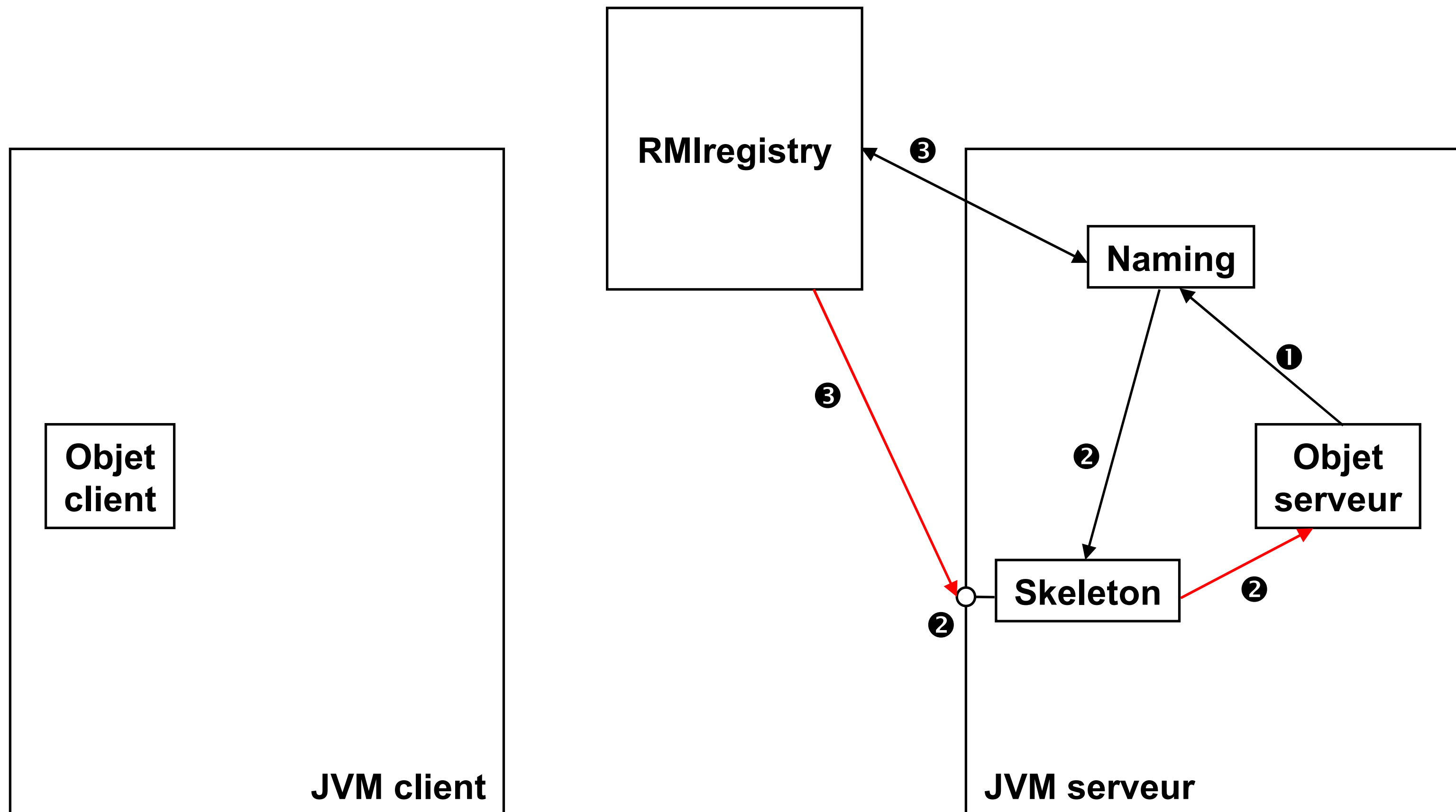
# RMI : objets distants et invocations distantes

- **Objet distant (objet serveur)**
  - ses méthodes sont invoquées depuis une autre JVM
    - dans un processus différent (même machine)
    - dans une machine distante (via réseau)
  - son comportement est décrit par une interface (ou plus) distante Java
    - déclare les méthodes distantes utilisables par le client
  - se manipule comme un objet local
- **Invocation distante (RMI)**
  - action d'invoquer une méthode d'une interface distante d'un objet distant
    - même syntaxe qu'une invocation sur un objet local

# RMI : architecture (1)



# RMI : mode opératoire côté serveur (1)



# RMI : mode opératoire côté serveur (2)

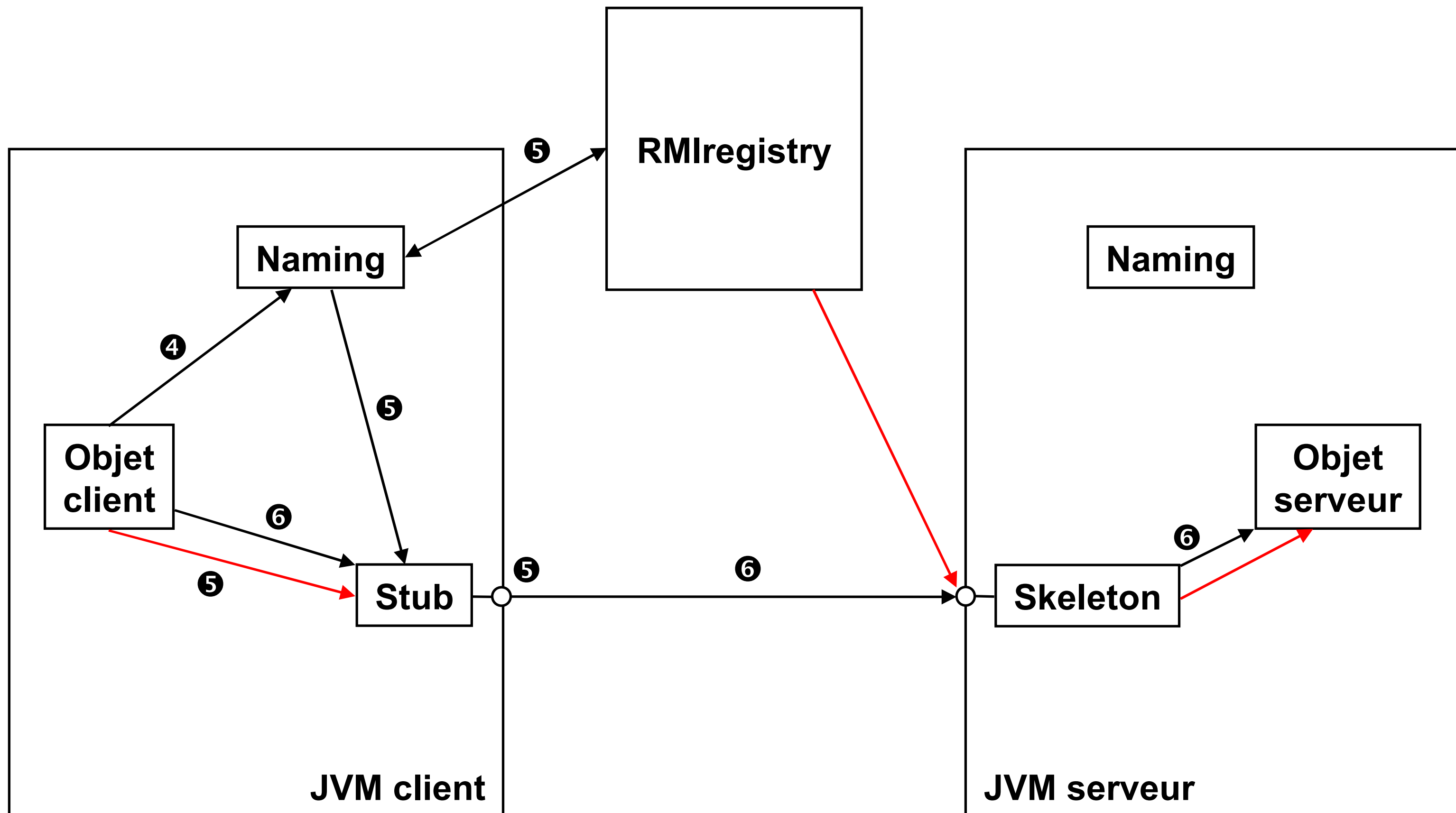
➊ L’objet serveur s'enregistre auprès du Naming de sa JVM (méthode rebind)

➋ L’objet skeleton est créé, celui-ci crée le port de communication et maintient une référence vers l’objet serveur

➌ Le Naming enregistre l’objet serveur, et le port de communication utilisé auprès du serveur de noms

- **L’objet serveur est prêt à répondre à des requêtes**

# RMI : mode opératoire côté client (1)



# RMI : mode opératoire côté client (2)

❹ L'objet client fait appel au Naming pour localiser l'objet serveur (méthode lookup)

❺ Le Naming récupère les "références" vers l'objet serveur, crée l'objet Stub et rend sa référence au client

❻ Le client effectue l'appel au serveur par appel à l'objet Stub

# Les amorces (stub/skeleton)

- **Programmes jouant le rôle d'adaptateurs pour le transport des appels distants**
  - réalisent les appels sur la couche réseau
  - pliage / dépliage des paramètres
- **A une référence d'OD manipulée par un client correspond une référence d'amorce**
- **Les amorces sont générées par le compilateur d'amorces : `rmic`**

# L'amorce client (stub)

- **Représentant local de l'OD qui implémente ses méthodes « *exportées* »**
  - transmet l'invocation distante à la couche inférieure *Remote Reference Layer*
  - réalise le pliage ("*marshalling*") des arguments des méthodes distantes
  - dans l'autre sens, il réalise le dépliage ("*demarshalling*") des valeurs de retour
- **Il utilise pour cela la sérialisation des objets**

# L’amorce serveur (skeleton)

- **Réalise le dépliage des arguments reçus par le flux de pliage**
- **Fait un appel à la méthode de l’objet distant**
- **Réalise le pliage de la valeur de retour**

# Les couches « basses » de l'architecture RMI

- **La couche des références distantes**
  - permet l'obtention d'une référence d'objet distant à partir de la référence locale au Stub
  - ce service est assuré par le lancement du programme **`rmiregister`**
    - à ne lancer qu'une seule fois par JVM, pour tous les objets à distribuer
    - une sorte de service d'annuaire pour les objets distants enregistrés
- **La couche de transport**
  - connecte les 2 espaces d'adressage (JVM)
  - suit les connexions en cours
  - écoute et répond aux invocations
  - construit une table des OD disponibles
  - réalise l'aiguillage des invocations

# Etapes de développement d'une application RMI

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

# Un exemple : un calculateur distribué

- **Cahier des charges**
  - séparer les primitives de calcul et les confier à un serveur dédié
  - opérations à implanter sur le serveur
    - addition
    - soustraction
    - multiplication
    - division
  - éléments à réaliser
    - l'interface **`Calculator.java`**
    - l'objet serveur de calcul **`CalculatorImpl.java`**
    - le programme serveur **`CalculatorServer.java`**
    - le programme client **`CalculatorClient.java`**

# Définir l'interface de la classe distante

- **Rappel**
  - les classes placées à distance sont spécifiées par des interfaces
  - ces interfaces doivent dériver de `java.rmi.Remote`
  - les méthodes de l'interface lèvent une `java.rmi.RemoteException`

- **`Calculator.java`**

```
import java.rmi.*;

public interface Calculator extends Remote {
  public long add(long a, long b) throws RemoteException;
  public long sub(long a, long b) throws RemoteException;
  public long mul(long a, long b) throws RemoteException;
  public long div(long a, long b) throws RemoteException;
}
```

# Définir l'implantation de l'objet distant

- **CalculatorImpl.java**

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

public class CalculatorImpl extends UnicastRemoteObject implements
   Calculator {
  // il faut surcharger le constructeur vide pour déclarer
  // l'exception RemoteException
  public CalculatorImpl() throws RemoteException {
    super();
  }

  public long add(long a, long b) throws RemoteException
  { return a + b; }
  public long sub(long a, long b) throws RemoteException
  { return a - b; }
  public long mul(long a, long b) throws RemoteException
  { return a * b; }
  public long div(long a, long b) throws RemoteException
  { return a / b; }
}
```

# Générer les stubs

- **`rmic CalculatorImpl`**
  - génère le fichier **`CalculatorImpl_Stub.java`**
  - génère le fichier **`CalculatorImpl_Skel.java`**
  - compile les fichiers
  - supprime les fichiers sources (.java)
- **options**
  - **`-keep`** : conserve les fichiers sources
    - permet d'inspecter la manière dont les stubs et skeletons sont implantés
  - **`-v1.2 ou plus`** : ne génère que **`Calculator_Stub`**

# Implanter l'application serveur

- **CalculatorServer.java**

```
import java.rmi.Naming;

public class CalculatorServer {
  public CalculatorServer() {
    try {
      Calculator c = new CalculatorImpl();
      Naming.rebind(
            "rmi://www.sitedistant.fr:1099/CalculatorService", c);
    } catch (Exception e) {
      System.out.println("Trouble: " + e);
    }
  }

  public static void main(String args[]) {
    new CalculatorServer();
  }
}
```

# Implanter l'application client

- **CalculatorClient.java**

```
import java.rmi.Naming;

public class CalculatorClient {
  public static void main(String[] args) {
    try {
      Calculator c = (Calculator) Naming.lookup(
              "rmi://www.sitedistant.fr/CalculatorService");
      System.out.println(c.sub(4, 3));
      System.out.println(c.add(4, 5));
      System.out.println(c.mul(3, 6));
      System.out.println(c.div(9, 3));
    } catch (MalformedURLException murle) {
      System.out.println("java.net.MalformedURLException" + murle);
    } catch (RemoteException re) {
      System.out.println("java.rmi.RemoteException" + re);
    } catch (NotBoundException nbe) {
      System.out.println("java.rmi.NotBoundException" + nbe);
    } catch (java.lang.ArithmeticException ae) {
      System.out.println("java.lang.ArithmeticException" + ae);
    }
  }
}
```

# Mettre en place le système

- **`$ rmiregistry port`**
  - démarre le service de registres RMI
- **`$ java CalculatorServer`**
  - démarre le serveur
  - charge l'implantation en mémoire
  - attend une connexion d'un client
- **`$ java CalculatorClient`**
  - sortie console :

```
$ 1
$ 9
$ 18
$ 3
```

# Passage de paramètres

- Lors d'appel de méthodes distantes : 4 cas pour gérer les paramètres ou la valeur retournée selon la classe du paramètre
  - Si classe implémente **`Remote`** : passage par adresse On passe ou récupère la référence sur un objet distant (**stub**)
- Si classe n'implémente pas **`Remote`** : passage par valeur L'objet est cloné, on passe ou récupère une copie de l'objet
  - Pour les types primitifs : passage par valeur également
  - Si classe n'implémente pas **`Serializable`** : objet ne peut pas être paramètre ou la classe ne peut pas être un type de retour (faute ...)
  - Les paramètres ou valeurs de retour sont forcément sérialisés pour être transmis via le réseau

# 2ème exemple : un Appel et retour

- Exemple : implémentation du patron *Observer* Un élément (l'observé) gère une donnée/un état **susceptible de changer**
- D'autres éléments (les observateurs) informent l'observé **qu'ils veulent être tenus au courant des changements de valeur de la donnée**
- Interface d'opérations, coté observateur Une opération qui sera appelée par l'observée quand la **donnée observée (un entier ici) changera de valeur**

```
public interface IChangeValue extends Remote
  {
public void newValue(int value)
throws RemoteException; }
```

# 2ème exemple : un Appel et retour

- **Interface d'opération coté observé**

```
public interface ISubscription extends Remote {
public void subscribe(IChangeValue obs)
throws RemoteException;
public void unsubscribe(IChangeValue obs)
throws RemoteException; }
```

- Une méthode pour s'enregistrer comme observateur de la valeur et une pour se désenregistrer Le paramètre est de de type IChangeValue
- C'est-à-dire un objet implémentant l'interface permettant de signaler un changement de valeur

# 2ème exemple : un Appel et retour

## Implémentation de l'observateur

```
public class Observer extends UnicastRemoteObject
  implements IchangeValue {
public void newValue(int value) throws
  RemoteException {
System.out.println(" nouvelle valeur : "+value);
}//méthode à appeller pour s'enregistrer auprès de
  l'observé
public void subscribeToObservee() {
try {ISubscription sub = (Isubscription)
Naming.lookup("rmi://localhost/observee");
sub.subscribe(this); }
catch (Exception e) { ... } }
// constructeur qui appelle « super » pour exporter
  l'objet
public Observer() throws RemoteException {
super(); }}
```

# 2ème exemple : un Appel et retour

- **Implémentation de l'observé**

```
public class Observee extends UnicastRemoteObject
  implements Isubscription {
// liste des observateurs
protected Vector observerList;
// donnée observée
protected int value = 0;
// enregistrement d'un observateur distant
public synchronized void subscribe(IChangeValue
  obs)throws RemoteException {
observerList.add(obs);
}
// désenregistrement d'un observateur distant
public synchronized voidunsubscribe(IChangeValue obs)
throws RemoteException {
observerList.remove(obs); }
```

# 2ème exemple : un Appel et retour

- **Implémentation de l'observé (suite)**

```
// méthode appelée localement quand la donnée change
  de valeur
public synchronized void changeValue(int newVal) {
value = newVal; IChangeValue obs;
// on informe tous les observateurs distants de la
  nouvelle valeur
for (int i=0; i<observerList.size(); i++)
try {obs = (IChangeValue)observerList.elementAt(i);
obs.newValue(value); }
catch(Exception e) { ... }
}
//on exporte l'objet et initialise la liste des
  observateurs
public Observee() throws RemoteException {
super();observerList = new Vector(); }}
```

# 2ème exemple : un Appel et retour

- Lancement de l'observé

```
Observee obs = new Observee();
Naming.rebind("observee", obs);
for (int i=1; i<=5; i++) {
obs.changeValue(i*10);
Thread.sleep(1000);
}
```

- Lancement d'un observateur

```
Observer obs = new Observer();
obs.subscribeToObservee();
// à partir de là, l'objet observateur sera
  informé des changements
```

// de valeur via l'appel de `newValue`

# 2ème exemple : un Appel et retour

- Les méthodes de l'observateur sont marquées avec **`Synchronized`** Pour éviter des incohérences en cas d'appels concurrents Coté servant.
- En interne de RMI, une opération invoqueé dynamiquement ou par le squelette l'est dans un thread créé à cet usage Accès concurrent possible si plusieurs clients distants demandent en même temps à exécuter une opération
- Toujours avoir cela en tête quand on implémente un servant même si on ne voit pas explicitement l'aspect concurrent

# Chargement dynamique

- **Chargement dynamique des amorces**
  - rappel : un objet client ne peut utiliser un objet distant qu'au travers des amorces
  - RMI permet l'utilisation des OD dont les amorces ne sont pas disponibles au moment de la compilation
  - à l'exécution, RMI réclamera au serveur l'amorce cliente manquante et la téléchargera dynamiquement (byte code)
- **Chargement dynamique des classes**
  - plus généralement, le système RMI permet le chargement dynamique de classes comme les amorces, les interfaces distantes et les classes des arguments et valeurs de retour des appels distants
  - c'est un chargeur de classes spécial RMI qui s'en charge:
    - `java.rmi.server.RMIClassLoader`

# Chargement dynamique

- **Côté servant**
  - Le rmiregistry reclame que le stub soit classé dansle classe path
  - Deux façon pour le faire
  - Soit le palcer dans le classe path
  - Soit specifier à la JVM lors du lancement du serveur où aller les chercher pour le binding

```
java -Djava.rmi.server.codebase=URL
exemple
java -Djava.rmi.server.codebase=file://
                               =http://
```

- **Coté client**
  - Une fois le stub localisé par lookup c'est le **`RMISecurityManager`** qui se charge de chargement dynamique

# La sécurité

- RMI n'autorise pas le téléchargement dynamique de classes (avec **`RMIClassLoader`**) si l'application (ou l'applet) cliente n'utilise pas de **`SecurityManager`** pour les vérifier.
- dans ce cas, seules les classes situées dans le **`CLASSPATH`** peuvent être récupérées
- le gestionnaire de sécurité par défaut pour RMI est **`java.rmi.RMISecurityManager`**
- il doit être absolument utilisé (**`System.setSecurity()`**) pour les applications *standalone*
- pas de problème pour les applets, c'est l'AppletSecurityManager (par défaut) qui s'en charge

- **`RMISecurityManager`**
  - vérifie la définition des classes et autorise seulement les passages des arguments et des valeurs de retour des méthodes distantes
  - ne prend pas en compte les signatures éventuelles des classes

# Les packages

- **`java.rmi`**
  - pour les classes côté client (accéder à des OD)
- **`java.rmi.server`**
  - pour les classes côté serveur (création des OD)
- **`java.rmi.registery`**
  - lié a l'enregistrement et à la localisation des OD
- **`java.rmi.dgc`**
  - pour le *Garbage Collector* des OD

# Aspects avancées : clonage & égalité

- **Passer des objets distant (OD) en paramètre (E/S).**
  - C'est possible ;-)
- **Ce que vous envoyer ou recevez c'est pas l'objet mais un *stub* de l'objet (sans passer par le *Naming*) et donc il faut faire attention à deux choses:**
- **Seul les méthodes hérité des interface qui hérite Remote sont accessible dans le *stub*.**
- **Les méthodes hérités de *Object* et qui peuvent être surchargé dans le l'OD n'ont pas pas la même signification sur le *stub* de l'objet.**
  - Clone()
  - Equals() // est celle de RemoteObject signifie égalité de référence sur le serveur entre deux objet *stub*

# Aspects avancées : sécurité

***java.security.AccessControlException: access denied (java.net.SocketPermission 127.0.0.1:1099 connect,resolve)***

- ***C'est une Exception très souvent rencontré par les programmeur. La cause???***
  - *RMISecurityManager: qui interdit partigement par defaut toutes création de socket sur l'ensemble des port.*
  - *Pour eviter que les object envoyer en paramètres ou en résultat ne puissent se connecter ....*
- ***Pour associer une politique de sécurité à un code JAVA, il faut***
  - *construire un objet instance de SecurityManager*
  - *surcharger les fonctions check... dont on veut changer la politique de sécurité*
  - *invoquer la méthode de la classe System, setSecurityManager, en lui passant cet objet créé plus haut.*

# Aspects avancées : sécurité

- ***Autre solution consite a passer par fichier de politique de sécurité et lancée l'application en lui précisant le fichier de sécurité à considéré.***
  - *java -Djava.security.policy=NomFichier.Policy*
  - *Pour données l'ensemble de droit a vos application serveur et client il faut lui donner les droit nécessaire.*
    - *grant {permission java.security.AllPermission;};*
    - *grant codeBase "file:/home/goubault/-" {permission java.net.SocketPermission "129.104.254.54:1024-", "connect, accept";}*