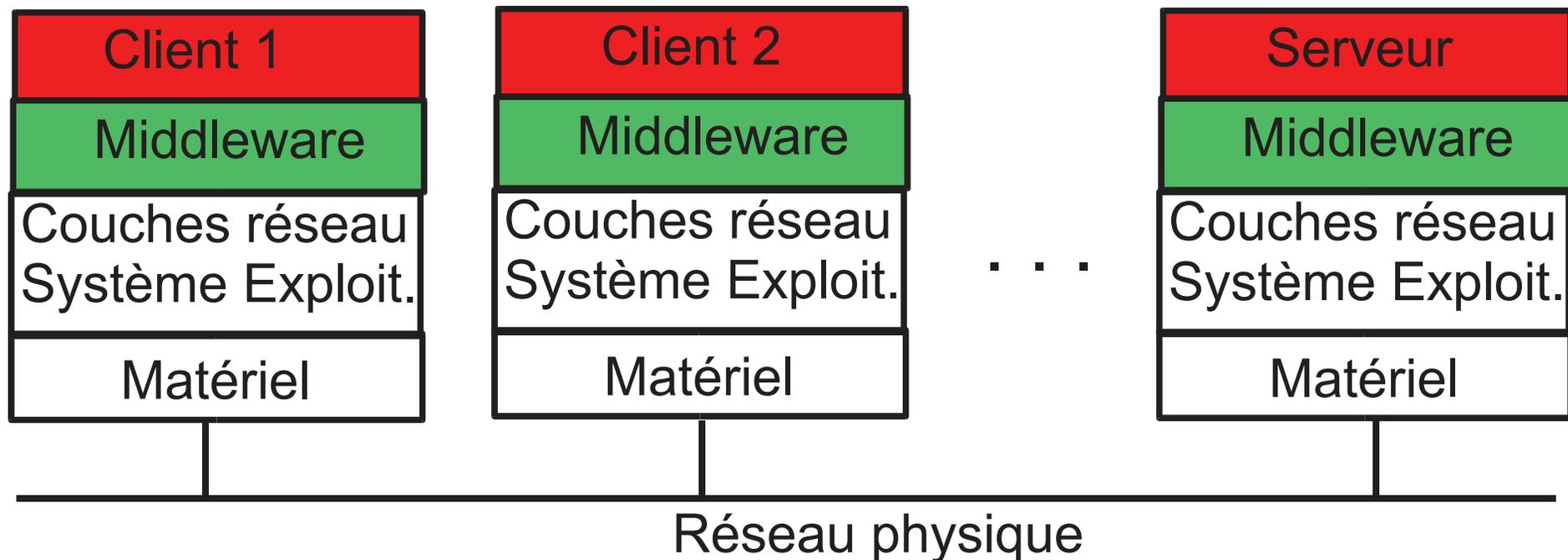


# Middleware

- ◆ Middleware (intergiciel en français) : couche logiciel
  - ◆ S'intercale entre le système d'exploitation/réseau et les éléments de l'application distribuée
  - ◆ Offre un ou plusieurs services de communication entre les éléments formant l'application ou le système distribué
    - ◆ Services de plus haut niveau que les communications via sockets TCP/UDP, augmentation du niveau d'abstraction



# *Middleware*

- ◆ Plusieurs grandes familles de middleware
  - ◆ Appel de procédure/méthode à distance
    - ◆ Extension « naturelle » de l'appel local d'opération dans le contexte des systèmes distribués
      - ◆ Une partie serveur offre une opération appelée par une partie client
    - ◆ Permet d'appeler une procédure/méthode sur un élément/objet distant (presque) aussi facilement que localement
    - ◆ Exemples (dans ordre historique d'apparition)
      - ◆ RPC (Remote Procedure Call) : solution de Sun pour C/Unix
      - ◆ CORBA (Common Object Request Broker Architecture) : standard de l'OMG permettant l'interopérabilité quelque soit le langage ou le système
      - ◆ Java RMI (Remote Method Invocation) : solution native de Java
  - ◆ Envoi ou diffusion de messages ou événements
    - ◆ Famille des MOM (Message Oriented Middleware)
    - ◆ Event Service de CORBA, JMS de Java
  - ◆ Mémoire partagée
    - ◆ Accès à une mémoire commune distribuée
    - ◆ JavaSpace de Java

# *Middleware*

## ► Familles de middlewares, caractéristiques générales

### ◆ Modèle RPC/RMI

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

### ◆ Modèles à message ou mémoire partagée

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

# *Middleware*

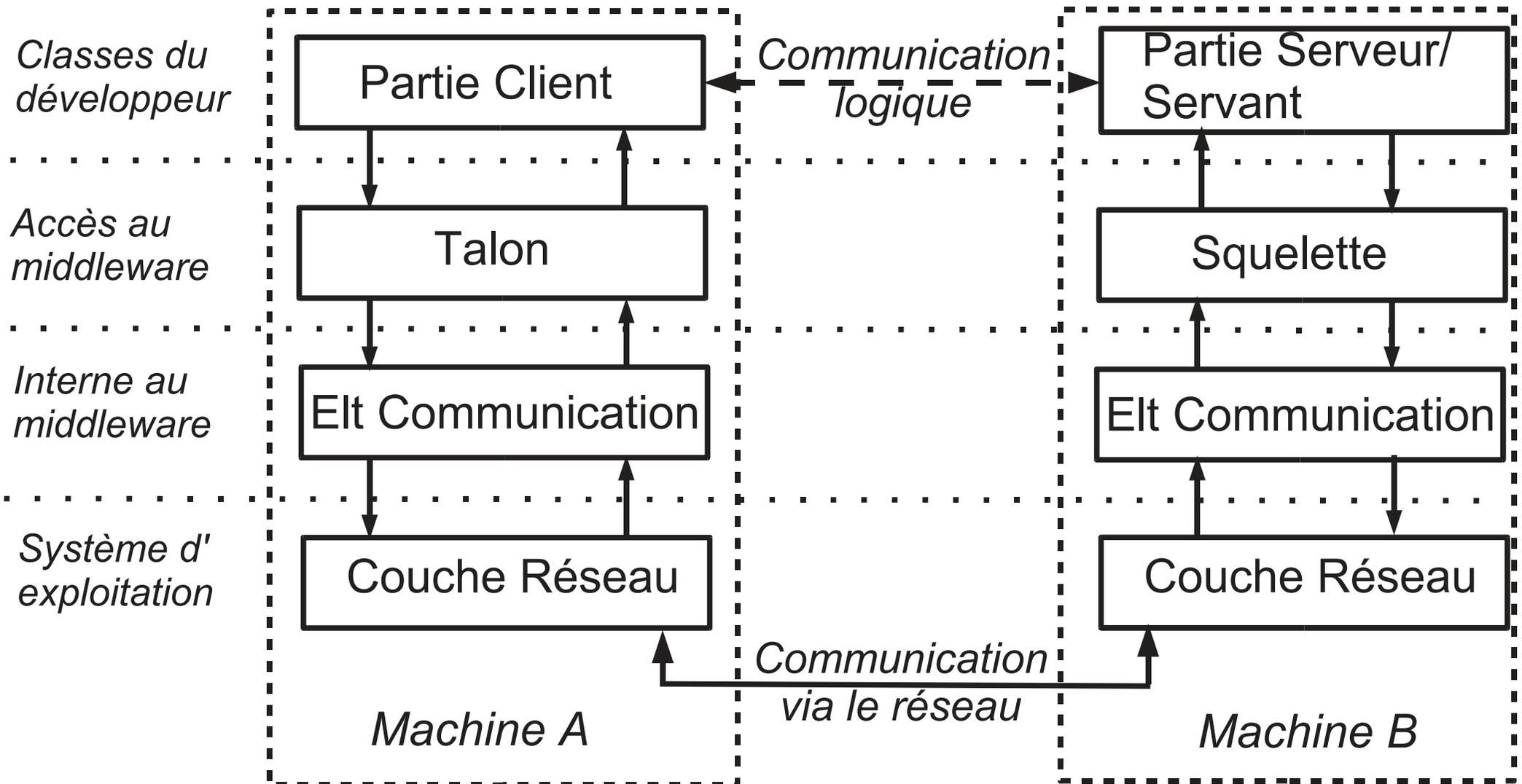
- ◆ Caractéristiques que peut assurer un middleware
  - ◆ Transparence à la localisation
    - ◆ Accès à des objets/ressources distantes aussi facilement que localement et indépendamment de leur localisation
      - ◆ Cas pour tous les middleware
  - ◆ Support de l'hétérogénéité des systèmes et des langages
    - ◆ CORBA par exemple : fait interopérer via du RPC/RMI des éléments logiciels écrits dans n'importe quel langage
  - ◆ Tout autre type de transparence (voir cours intro), selon le middleware
- ◆ Services offerts par un middleware
  - ◆ Un ou plusieurs services de communication/interaction
    - ◆ RPC, diffusion de message, mémoire partagée ...
  - ◆ Service de nommage

# ***Middleware***

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

# Middleware RPC/RMI

- ◆ Architecture générale, plusieurs couches



# *Middleware RPC/RMI*

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

# *Middleware RPC/RMI*

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

# *Middleware RPC/RMI*

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

# *Middleware RPC/RMI*

- ◆ Fonctionnement général d'un appel d'opération à distance (fin)
  8. Le squelette récupère la valeur de retour de l'opération, la compacte et l'envoie au talon via les éléments de communication
    - ◆ Même si pas de valeur de retour, on renvoie quelque chose au talon pour l'informer de la fin de l'appel d'opération
  9. Le talon décompacte la valeur et la retourne la valeur à la partie client
  10. L'appel de l'opération distante est terminé, la partie client continue son exécution

# *Middleware RPC/RMI*

- ◆ Gestion des problèmes de communication entre partie client et servant
  - ◆ Problèmes potentiels
    - ◆ Le servant est planté ou inaccessible
    - ◆ La requête d'appel d'opération n'a pas été reçue coté servant
    - ◆ La réponse à la requête n'a pas été reçue par la partie client
    - ◆ L'exécution de l'opération s'est mal passée
    - ◆ ...
  - ◆ En cas de problèmes, coté client, on n'a pas reçu la réponse à la requête
    - ◆ Doit renvoyer la requête au serveur a priori
    - ◆ Mais coté serveur, si l'opération modifie un état doit éviter de rappeler l'opération a priori
    - ◆ Plusieurs cas possibles pour gérer cela

# *Middleware RPC/RMI*

- ◆ 4 sémantiques possibles pour l'appel d'opérations
  - ◆ « Peut-être »
    - ◆ On ne sait pas si l'opération a été appelée
  - ◆ « Au moins une fois »
    - ◆ L'opération a été appelée au moins une fois mais peut-être plusieurs fois également
      - ◆ Problème si opération modifie un état (opération non idempotente)
  - ◆ « Au plus une fois »
    - ◆ L'opération est appelée une seule fois ou on reçoit un message informant qu'un problème a eu lieu et que l'opération n'a pas été appelée
  - ◆ « Une fois »
    - ◆ Cas idéal mais difficile à atteindre

# ***Middleware RPC/RMI***

- ◆ Trois caractéristiques pour gérer les requêtes d'appels d'opérations
  - ◆ Retransmission de la requête
    - ◆ Le client peut redemander au serveur d'appeler l'opération en cas de non réponse reçue de sa part
  - ◆ Filtrage des duplicats de requêtes
    - ◆ Si le serveur détecte qu'il s'agit d'une redemande d'appel d'opération, il n'exécute pas une nouvelle fois l'opération
  - ◆ Retransmission des résultats
    - ◆ Le serveur garde un historique des valeurs de retour des opérations et peut renvoyer le résultat retourné pour une opération en cas de nouvelle demande de cette même opération

# *Middleware RPC/RMI*

- ◆ Trois combinaisons principales de caractéristiques et sémantiques associées
  - ◆ Pas de retransmission de la requête
    - ◆ Sémantique « peut-être »
  - ◆ Retransmission des requêtes mais pas de filtrage des duplicats de requêtes
    - ◆ Sémantique « au moins une fois »
  - ◆ Retransmission des requêtes, filtrage des duplicats de requêtes et utilisation de l'historique des réponses
    - ◆ Sémantique « au plus une fois »

# *Garbage collector distribué*

- ◆ Dans la plupart des mises en oeuvre de langages objets : garbage collector
  - ◆ « Ramasse miette » en français
  - ◆ But : supprimer de la mémoire locale (du programme, de la JVM en Java) les objets n'étant plus référencés par aucun autre objet
- ◆ Objets distribués (via un middleware comme Java RMI)
  - ◆ Capacité à référencer des objets distants comme des objets locaux
  - ◆ Pour nettoyage des objets non référencés : doit alors prendre aussi en compte les références distantes
  - ◆ Distributed Garbage Collector (DGC)

# *Garbage collector distribué*

- ◆ Fonctionnement général d'un garbage collector (en local)
  - ◆ Un compteur de référence est associé à chaque objet
  - ◆ Quand un objet récupère une référence sur l'objet, on incrémente de 1 le compteur de cet objet
  - ◆ Quand un objet ne référence plus l'objet, on décrémente le compteur de 1
  - ◆ Quand compteur a pour valeur 0, on peut supprimer l'objet de la mémoire car plus aucun objet ne l'utilise plus
- ◆ En distribué peut utiliser un fonctionnement similaire
  - ◆ En prenant en compte à la fois les références locales et distantes

# *Garbage collector distribué*

- ◆ Contraintes supplémentaires en distribué
  - ◆ Coordination entre garbage collectors sur chaque machine pour assurer gestion des références
    - ◆ Référence sur un objet distant = référence locale sur un proxy pour cet objet
    - ◆ Quand localement un proxy n'est plus référencé par aucun objet, on peut informer le GC gérant l'objet distant de la perte de référence
  - ◆ Si une machine distante est plantée ou est inaccessible
    - ◆ Ne pourra pas préciser à un GC que l'objet n'est plus référencé
    - ◆ Solution : utiliser une durée de bail (*lease*)
    - ◆ Si au bout d'un certain temps, un objet distant n'a pas utilisé un objet local, localement, on considère que l'objet distant ne référence plus l'objet local

# *Java RMI*

# *Java RMI*

- ◆ Java intègre en natif deux middlewares de type RPC/RMI
  - ◆ Java RMI
  - ◆ Une mise en oeuvre de CORBA
- ◆ Java RMI
  - ◆ Spécifique au langage Java : ne fait communiquer que des éléments écrits en Java
  - ◆ Suit l'architecture et les règles de fonctionnement des middlewares de type RPC/RMI
  - ◆ Cas particulier du squelette coté serveur : depuis Java 1.2, plus besoin de squelette
    - ◆ L'appel de l'opération est fait via une invocation dynamique
    - ◆ Plus besoin d'identifier la méthode à appeler en décodant les données reçues (comme on le faisait dans le TP2 avec décodage des messages)

# Java RMI

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

# Interface

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

# Exemple d'interface

## ◆ Exemple d'interface, définissant 2 opérations

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

- ◆ Les opérations sont quelconques, avec type de retour et paramètres de types primitifs ou de n'importe quelle classe
- ◆ Lors de l'implémentation des opérations de l'interface
  - ◆ On ne traite pas directement les cas d'erreurs correspondant à la levée d'une exception `RemoteException`
  - ◆ C'est le middleware/Java qui le fera côté client en cas de besoin

## *Exemple d'interface*

- ◆ La classe Rectangle permet de décrire les coordonnées d'un rectangle

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

- ◆ Note : les attributs sont publics par souci de simplification

# ***Servant : exportation***

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

# Servant

## ◆ Exemple pour l'interface Irectangle

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

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

# Registry

- ◆ Une partie client doit pouvoir récupérer une référence sur l'objet distant implémentant l'interface
  - ◆ Utilise les services du « registry » (registre, annuaire ...)
  - ◆ Le registry est lancé à part à coté des applis Java
  - ◆ Un objet accède au registry via la classe `Naming`
- ◆ Identification d'un objet distant
  - ◆ Via une URL de la forme `rmi://hote:port/nomObj`
    - ◆ `hote` : nom de la machine distante (sur laquelle tourne un registry)
    - ◆ `port` : port sur lequel écoute le registry
    - ◆ `nomObj` : nom donné à un objet offrant des opérations
  - ◆ Si pas précision du port : port par défaut utilisé par le registry
  - ◆ Si pas précision de l'hôte : registry local par défaut (`localhost`)

# Registry

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

# Registry

## ▶ Classe `java.rmi.Naming`

- ◆ 5 opérations statiques pour enregistrer des objets et récupérer leurs références (lèvent toutes plusieurs exceptions – voir API)
  - ◆ `void bind(String name, Remote obj)`
    - ◆ Enregistre un objet sous le nom `name` (erreur si déjà un objet avec ce nom)
  - ◆ `void rebind(String name, Remote obj)`
    - ◆ Enregistre un objet sous le nom `name`, en écrasant la précédente liaison objet/nom si elle existait
  - ◆ `void unbind(String name)`
    - ◆ Supprime du registry la référence vers l'objet nommé `name`
  - ◆ `String[] list(String name)`
    - ◆ Retourne l'ensemble des noms des objets enregistrés sur le registry
  - ◆ `Remote lookup(String name)`
    - ◆ Retourne une référence sur l'objet dont le nom est passé en paramètre
    - ◆ Exception `NotBoundException` levée si objet pas trouvé par le registry
- ◆ Attribut `name` dans les 5 opérations : sous la forme URL de type RMI

# *Génération stub/squelette*

- ◆ Pour pouvoir appeler les opérations à distance, il faut générer le stub (talon)
  - ◆ Et le squelette avant la version 1.2 de Java
- ◆ Utilitaire : `rmic`
  - ◆ Pour notre exemple : `$ rmic RectangleImpl`
  - ◆ On passe en paramètre à `rmic` le nom de la classe qui implémente l'interface (ou les interfaces)
  - ◆ Génère et compile automatiquement le fichier `RectangleImpl_Stub.java`  
(option `-keep` de `rmic` pour conserver les sources Java)
  - ◆ Pas de manipulation directe dans notre code de ce fichier

# *Exemple coté serveur*

- ◆ Coté serveur : instantiation d'un RectangleImpl et enregistrement sur le registry

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

- ◆ Avant d'exécuter ce code : lancer `rmiregistry`
- ◆ On suppose que le serveur et le registry sont lancés sur la machine `scinfr222` avec le port par défaut pour le registry

# Exemple coté client

- ◆ Code client client pour appeler les opérations distantes (exceptions à gérer en plus)

```
Rectangle r1, r2;  
r1 = new Rectangle(10, 10, 20, 30);  
  
// on demande au registry tournant sur la machine scinfr222 (port par défaut)  
// de nous renvoyer la réf de l'objet nommé « opRect »  
IRectangle opRectangle = (IRectangle)  
    Naming.lookup("rmi://scinfr222/opRect");  
  
// une fois la référence obtenue, on appelle normalement les opérations  
// de l'interface sur cet objet (en catchant RemoteException)  
int surface = opRectangle.calculSurface(r1);  
r2 = opRectangle.decalerRectangle(r1, 15, 10);  
  
System.out.println(" surface de r1 = "+surface);  
System.out.println(" position de r2 = "+r2);
```

# *Exemple coté client*

- ◆ Résultat du lookup
  - ◆ Si objet trouvé par le registry distant, retourne un objet implémentant l'interface générique `Remote`
  - ◆ En pratique ici : un objet implémentant `IRectangle`
    - ◆ Comme par exemple `RectangleImpl` instantiée coté serveur
    - ◆ On récupère en fait une référence sur une instance de `RectangleImpl_Stub`
      - ◆ Le talon local coté client qui sert de proxy pour l'accès à l'instance de `RectangleImpl` coté serveur
  - ◆ Pour simplifier, on fait le cast avec `IRectangle`
    - ◆ Pas besoin de savoir l'instance de quelle classe exactement on manipule

# *Passage des paramètres/données*

- ◆ 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
  - ◆ Si classe n'implémente pas `Remote` : passage par valeur
    - ◆ L'objet est cloné, on passe ou récupère une copie de l'objet
  - ◆ Pour 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
    - ◆ Les paramètres ou valeurs de retour sont forcément sérialisés pour être transmis via le réseau

# *Appels en retour*

- ◆ On a vu dans l'exemple précédent
  - ◆ Une partie client appelant une méthode sur une partie serveur
  - ◆ Mais la partie serveur peut avoir besoin d'appeler une méthode sur la partie client
    - ◆ Callback : appel en retour
- ◆ Deux moyens pour le serveur de connaître le client
  - ◆ Rechercher sa référence via son nom dans le registre
  - ◆ Récupérer sa référence via l'appel de la méthode
- ◆ Note sur les termes partie serveur et partie client
  - ◆ Différence entre appelant et appelé de l'opération
  - ◆ Chaque élément peut jouer les 2 rôles indifféremment

# Appels en 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; }  

```

# *Appels en 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

# Appels en retour

## ◆ Implémentation de l'observateur

```
public class Observer extends UnicastRemoteObject
    implements IchangeValue {

    // méthode qui sera appelée par l'observé distant
    public void newValue(int value) throws RemoteException {
        System.out.println(" nouvelle valeur : "+value);
    }

    // méthode à appeler pour s'enregistrer auprès de l'observé
    public void subscribeToObservee() {
        try {
            ISubscription sub = (ISubscription)
                Naming.lookup("rmi://scinfe222/observee");
            sub.subscribe(this); }
        catch (Exception e) { ... } }

    // constructeur qui appelle « super » pour exporter l'objet
    public Observer() throws RemoteException {
        super(); }
}
```

# Appels en 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 void unsubscribe(IChangeValue obs)
                                throws RemoteException {
        observerList.remove(obs); }
}
```

# *Appels en 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(); }
}
```

# Appels en 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
```

# Appels en retour

## ◆ Note 1

- ◆ Quand un observateur appelle `subscribe` sur l'observé distant, il passe sa référence « distante » car il implémente une interface `Remote`
- ◆ L'appel de `newValue` se fait donc bien sur l'observateur distant (distant du point de vue de l'observé qui appelle `newValue`)

## ◆ Note 2

- ◆ On peut noter que les observateurs ne se sont pas enregistrés via le registry
- ◆ On peut en effet communiquer avec un objet distant à partir du moment où il s'est exporté
  - ◆ L'enregistrement via le registry n'est utile que pour récupérer sa référence à partir de son nom associé
  - ◆ Inutile ici car les observateurs passent eux-mêmes leurs références à l'observé

# *Appels concurrents d'opérations*

- ◆ 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 invoquée 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

# *Mobilité de code*

- ◆ Référence sur un objet distant
  - ◆ En réalité une référence sur un stub local qui implémente la même interface d'opérations que l'objet distant
  - ◆ Même si la classe du stub n'apparaît pas explicitement dans le code, elle doit pouvoir être retrouvée à l'exécution
  - ◆ Contraintes d'une application distribuée
    - ◆ Les différents objets ne s'exécutent pas dans les mêmes JVM ni sur les mêmes machines : accès aux .class peuvent différer
    - ◆ Malgré cela, on doit pouvoir coté client récupérer la classe du stub
- ◆ 2 modes pour récupérer la classe du stub de l'objet distant
  - ◆ Localement, la classe se retrouve via le CLASSPATH
  - ◆ A distance, en téléchargeant la classe à partir d'un emplacement précisé

# *Mobilité de code*

- ◆ Une propriété de Java permet de préciser une URL où aller télécharger les classes requises
- ◆ `java.rmi.server.codebase`, à positionner en lançant le programme Java
- ◆ URL est de type « file », « ftp » ou « http »
- ◆ Exemples
  - ◆ `$ java -Djava.rmi.server.codebase=file:///home/ecariou/test-rmi/ Serveur`
  - ◆ `$ java -Djava.rmi.server.codebase=http://www.univ-pau.fr/~ecariou/test-rmi/ Serveur`
- ◆ Problème potentiel du téléchargement de code
  - ◆ Sécurité : ne doit pas pouvoir par défaut télécharger n'importe quelle classe de n'importe où

# *Mobilité de code*

- ◆ Gestion de la sécurité
  - ◆ Il n'est pas possible de télécharger des classes sans mettre en place un « security manager »
  - ◆ Classe `java.lang.SecurityManager`
    - ◆ Permet de préciser les permissions pour l'accès aux fichiers, au réseau, à la réflexion sur les objets, à la sérialisation ...
    - ◆ Classe à spécialiser pour préciser les permissions selon le contexte
  - ◆ Pour RMI, il existe une spécialisation :  
`java.rmi.RMI SecurityManager`
    - ◆ Très restrictif par défaut : interdit notamment le téléchargement de classe et à peu près tout le reste aussi ...

# Mobilité de code

- ◆ Gestion de la sécurité (suite)
  - ◆ Avec RMI, deux façons « d'ouvrir » les restrictions
    - ◆ Utiliser une spécialisation « maison » de SecurityManager à la place de RMISecurityManager
    - ◆ Utiliser RMISecurityManager avec en plus un fichier décrivant les permissions : java.policy
  - ◆ Exemple, en donnant toutes les permissions
    - ◆ A éviter en pratique si on veut bien gérer l'aspect sécurité !
    - ◆ Fichier toutesPermissions.policy

```
grant {  
    permission java.security.AllPermission;  
};
```

- ◆ Spécialisation de SecurityManager

```
public class SecurityManagerOuvert extends SecurityManager{  
    public void checkPermission(Permission perm) { }  
}
```

# *Mobilité de code*

## ◆ Gestion de la sécurité (suite)

- ◆ Pour installer un gestionnaire de sécurité, exécuter le code suivant

```
System.setSecurityManager(new RMISecurityManager());
```

- ◆ Pour utiliser un fichier policy, positionner la propriété `java.policy` au lancement du programme

```
$ java -Djava.security.policy=toutesPermissions.policy  
Serveur
```

- ◆ Ces permissions sont à positionner pour les programmes qui téléchargeront les classes, inutiles pour les autres
- ◆ Comment positionner des permissions plus précises ?
  - ◆ Assez complexe à appréhender et comprendre...
  - ◆ Voir les documentations spécialisées à ce sujet

# *Mobilité de code*

- ◆ Fonctionnement général pour récupérer le stub coté client
  - ◆ Quand serveur enregistre un objet sur le registry, le registry doit pouvoir récupérer la classe du stub distant pour l'associer à l'objet
    - ◆ Soit le serveur est lancé avec un codebase positionné et le registry va l'y chercher
    - ◆ Soit le registry la trouve en local (registry lancé du même répertoire que le serveur par exemple)
  - ◆ Quand un client demande à récupérer la référence sur un objet
    - ◆ Le registry lui envoie le nom de la classe du stub et le codebase associé (sauf si le registry a trouvé la classe en local, il n'envoie pas de codebase)
      - ◆ Si le client trouve cette classe via son CLASSPATH, c'est celle là qui est choisie en priorité
      - ◆ Sinon, il la récupère via le codebase envoyé par le registry
      - ◆ Si on lui avait passé en paramètre un codebase à son lancement, il la cherche également à cet endroit
    - ◆ Le client instancie la classe du stub, l'instancie dynamiquement et utilise cette instance pour accéder à l'objet distant

# *Mobilité de code*

- ◆ Récupération des stubs, principaux cas (modulables entre eux)
  - ◆ Les parties serveurs et clients ont accès aux bonnes classes via leur CLASSPATH
    - ◆ Pas besoin de préciser de codebase, tout sera chargé localement
  - ◆ Dynamique coté client
    - ◆ Le serveur est lancé en précisant un codebase
    - ◆ Le client est lancé sans codebase et retrouvera les classes via les informations envoyées par le registry
  - ◆ Dynamique coté serveur
    - ◆ Le client est lancé en précisant un codebase
    - ◆ Le serveur, en cas d'appel en retour par exemple, récupère les stubs coté client via ce codebase

# *Mobilité de code*

- ◆ Mobilité de code
  - ◆ Ne concerne pas que les stubs
  - ◆ Toute classe nécessaire au bon fonctionnement de l'application peut-être téléchargée au besoin
- ◆ Exemple concret : variante de l'exemple avec les rectangles
- ◆ La classe Rectangle2 spécialise Rectangle

```
public class Rectangle2 extends Rectangle {  
  
    public Rectangle2(int x1, int y1, int x2, int y2) {  
        super(x1, y1, x2, y2);  
    }  
}
```

# *Mobilité de code*

- ◆ Exemple (suite)
- ◆ L'interface est inchangée : `IRectangle`
- ◆ Coté servant, on modifie `decaleRectangle()` pour renvoyer une instance de `Rectangle2`

```
public class RectangleImpl implements IRectangle {  
  
    (...)  
  
    public Rectangle decalerRectangle(Rectangle rect,  
        int x, int y) throws RemoteException  
    {  
        return new Rectangle2(rect.x1 + x, rect.y1 + y,  
            rect.x2 + x, rect.y2 + y);  
    }  
    (...)  
}
```

# *Mobilité de code*

## ◆ Exemple (suite)

- ◆ La partie client (Client.java) va télécharger les classes dynamiquement
- ◆ Elle doit donc positionner un security manager pour commencer

```
// positionne le security manager acceptant tout
```

```
System.setSecurityManager(new SecurityManagerOuvert());
```

```
Rectangle r1, r2;
```

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

```
IRectangle opRectangle = (IRectangle)
```

```
    Naming.lookup("rmi://scinfe222/opRect");
```

```
r2 = opRectangle.decalerRectangle(r1, 15, 10);
```

```
System.out.println("classe de IRectangle = "  
                    +opRectangle.getClass());
```

```
System.out.println("classe de r2 = "+r2.getClass());
```

# Mobilité de code

- ◆ Exemple (suite)
  - ◆ Le serveur (Serveur.java) est lancé en précisant un codebase pour accéder aux classes requises
    - ◆ Il instancie un RectangleImpl et l'enregistre auprès du registry
    - ◆ 

```
$ java -Djava.rmi.server.codebase=http://www.univ-pau.fr/~ecariou/test-rmi/ Serveur
```
  - ◆ Le registry est lancé d'un répertoire quelconque (et pas celui où se trouvent les classes)
  - ◆ Le client est lancé sans avoir accès via son CLASSPATH aux bonnes classes (RectangleImpl et Rectangle2)
    - ◆ 

```
$ java Client
```
    - ◆ Ce qui affiche le résultat suivant

```
classe de IRectangle = class RectangleImpl_Stub
classe de r2 = class Rectangle2
```

# *Mobilité de code*

## ◆ Exemple (fin)

- ◆ Les classes RectangleImpl et Rectangle2 ont donc été récupérées par le client en les téléchargeant à l'URL <http://www.univ-pau.fr/~ecariou/test-rmi/>

## ◆ Note

- ◆ Si à l'exécution des exceptions UnmarshalException, AccessDeniedException ou ClassNotFoundException sont levées
  - ◆ Correspond souvent à des tentatives de téléchargement de classes mais sans avoir les permissions suffisantes ou sans savoir où aller les récupérer
  - ◆ Si c'est du côté serveur : le registry n'arrive pas à récupérer le stub, le codebase a été oublié côté serveur

## ***Mobilité de code***

- ◆ En conclusion, pour bien gérer l'accès aux classes, 2 grands modes
  - ◆ S'arranger pour qu'à la fois le client et le serveur aient accès à toutes les classes via leur CLASSPATH
  - ◆ Lancer systématiquement le serveur avec le codebase et le client avec un security manager pour pouvoir télécharger les classes au besoin
    - ◆ En lançant le registry sans qu'il ait directement accès aux classes
- ◆ Note
  - ◆ Ne pas oublier de prendre en compte les packages
    - ◆ Et donc de remonter éventuellement dans l'arborescence pour préciser le codebase

# *Garbage collector*

- ◆ Java RMI gère un garbage collector distribué
  - ◆ En local, pour effectuer des actions avant la destruction d'un objet
    - ◆ Redéfinition de la méthode `finalize()` de `java.lang.Object`
  - ◆ Pour la gestion des références distantes
    - ◆ Interface `java.rmi.server.Unreferenced` définissant :  
`public void unreferenced()`
    - ◆ Cette interface peut être implémentée par un objet offrant des opérations appellables à distance
    - ◆ L'opération `unreferenced` est appelée quand plus aucun objet distant ne possède de référence sur l'objet courant
      - ◆ Ce que l'on peut faire par exemple dans cette méthode : « désexporter » l'objet : `unexporteObject(this, true);`

# *Garbage collector*

- ◆ Le garbage collector distribué de RMI gère une durée de bail
  - ◆ Propriété `java.rmi.dgc.leaseValue`
  - ◆ Le bail est de 10 minutes par défaut
  - ◆ Pour le modifier, on peut le faire au lancement du programme
    - ◆ `$ java -Djava.rmi.dgc.leaseValue=20000 Serveur`
    - ◆ La durée est précisée en milli-secondes, 20 secondes donc ici
- ◆ Note
  - ◆ Ne pas oublier que le registry possède aussi une référence sur l'objet si ce dernier a été enregistré dans le registry

# *Exportation de l'objet*

- ◆ Pour exporter un objet, on a vu la classe `UnicastRemoteObject`
  - ◆ L'objet reste alors actif et accessible en permanence
  - ◆ Si la JVM dans lequel il s'exécute est arrêtée, l'objet disparaît et sa référence également
- ◆ Variante possible : *Activable*
  - ◆ Permet de n'activer l'objet que quand il est sollicité
    - ◆ Via un mécanisme et une JVM particuliers
  - ◆ Permet de limiter le nombre d'objets actifs en même temps : gain de mémoire au besoin
  - ◆ Plus complexe à utiliser : voir documentation spécialisée

# *Gestion des firewalls*

- ◆ Élément important dans la construction de toute application distribuée
  - ◆ Prendre en compte les éventuels firewalls pouvant empêcher des communications à distances
- ◆ Plusieurs solutions en RMI, dont l'encapsulation des communications dans des paquets HTTP
  - ◆ Tunneling HTTP
  - ◆ Evite le filtrage sur les ports « non standards »
- ◆ Voir documentation spécialisées pour plus d'infos