Intergiciel - concepts de base

Ada Diaconescu, Laurent Pautet & Bertrand Dupouy

ada.diaconescu _at_ telecom-paristech.fr

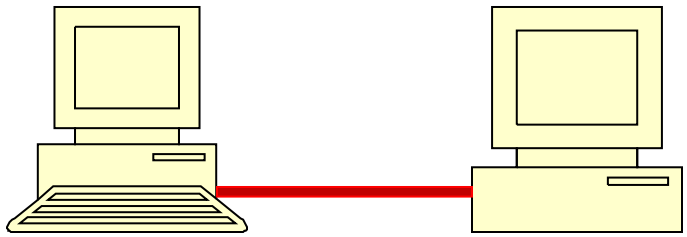
Rappel : système réparti

- Système constitué de multiples ressources informatiques

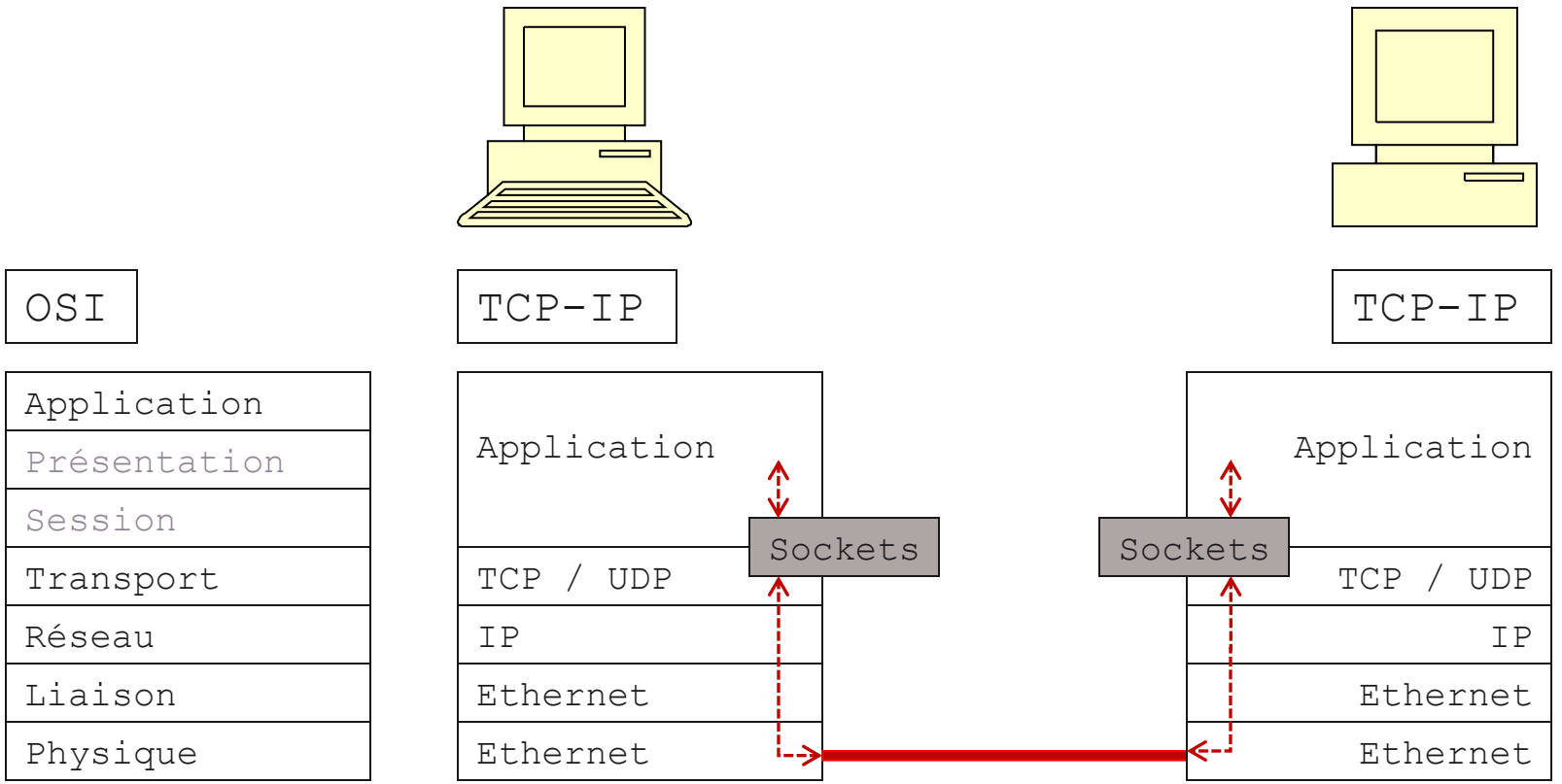
  (ex : processus logiciels, équipements matériels, …)

  interconnectées via un support de communication

  (ex: réseaux, communication interprocessus, mémoire partagée, …)



Rappel : modèle OSI et TCP/IP



Des sockets aux intergiciels

- Limites de l'API socket
  - Travail fastidieux, répétitif, sujet aux erreurs
  - Configuration, construction de requêtes, déploiement, exécution, …
- Solution :
  - Factoriser les opérations sur les sockets à l'aide de bibliothèques de plus haut niveau
- Intergiciel (« middleware ») : abstractions pour la répartition
  - Fournit un modèle de répartition : entités inter-agissantes suivant une sémantique claire
  - Au dessus des briques de base de l'OS (dont les sockets)
  - Définit un cadre complet pour la conception et la construction d'applications distribuées

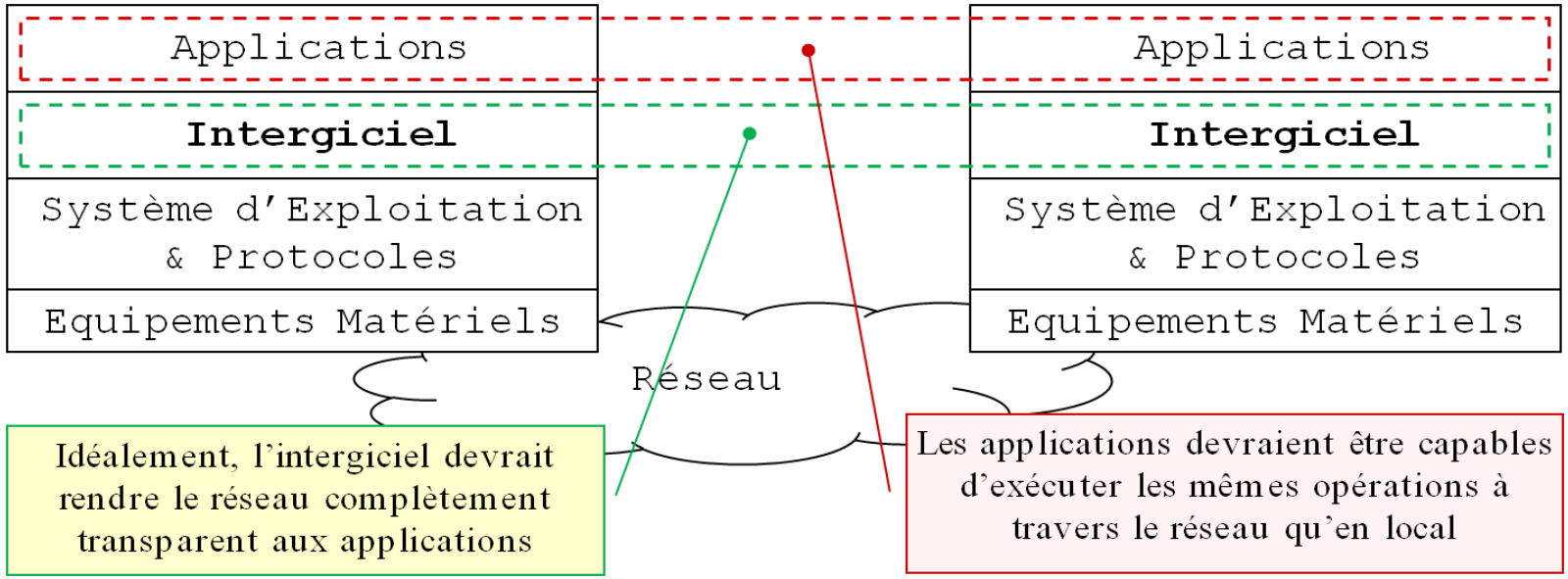
Intergiciels - définitions de base

- Logiciel réutilisable qui fait la *médiation* entre les applications et le réseau

- Gère et facilite l'implantation de l'*interaction* entre des applications qui s'exécutent sur des plates-formes distantes et/ou hétérogènes

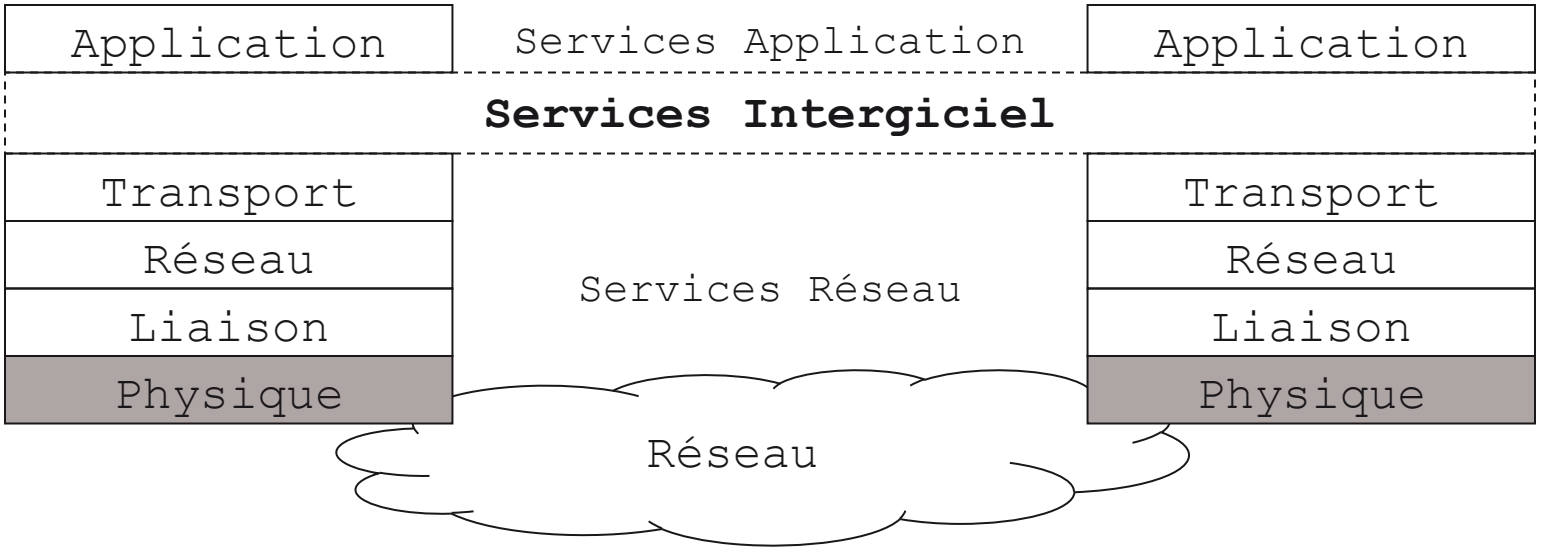
Intergiciel – positionnement (1)

- Au dessus du réseau
- En dessous des applications (logique de business), bases de données, …
- Sur toutes les machines participantes



Intergiciel – positionnement (2)

- L'intergiciel - où se trouve-t-il par rapport au modèle OSI?



Intergiciel - contenu

- Une collection réutilisable et extensible de services non-fonctionnels :
  - A l'origine : des services de communication
  - Et ensuite : nommage, transactions, sécurité, persistance, gestion d'instances, journalisation (logging), …

Un service *non-fonctionnel* n'est pas conscient des buts fonctionnels (de business) et de la sémantique de l'application qu'ils servent

Intergiciels – importance

- Aide à définir et concevoir un système réparti
  - **Aide les développeurs**, en les protégeant :
    - des détails liées à la plateforme sous-jacente
    - des difficultés liées à la distribution

    => Plus besoin de programmer de sockets
  - **Amortit les coûts de développement** initiaux par la réutilisation et par la dissémination de l'expertise :
    - Communication : création de session, (de)*marshalling*, collecte et « bufferisation » de requêtes, contrôle de la concurrence, …
    - D'autres services non-fonctionnels : nommage, transactions, sécurité, …

Intergiciels – caractéristiques

- Se fonde sur des mécanismes de répartition
  - Passage de messages, sous-programmes distant
  - Objets répartis, objets partagés (mémoire répartie)
  - Transactions
- Fournit un contrôle sur ces mécanismes
  - Langages de description ou de programmation
  - Interfaces de bibliothèques ou de services
- S'accompagne d'un intergiciel de mise en œuvre
  - Bibliothèques + outils

modèle de répartition – Envoi de Messages

- Interface réseau
  - Interface bas niveau - UDP, TCP, ATM
  - IPv6, multicast, SSL/TSL
- PVM/MPI
  - Calcul massivement parallèle
  - Communication de groupe
- Java Messaging Service (JMS)
  - Interface haut niveau (Standard Java)
  - Message Oriented Middleware (MOM) : Point à Point ou Publish / Subscribe

Modèle de répartition – Appel de Procédure à Distance

- RPC (Remote Procedure Call) - Sun
  - Service de nommage ou de localisation (portmapper)
  - Compilateur rpcgen qui produit les souches et squelettes
  - Intégré dans DCE (Distributed Computing Environment)

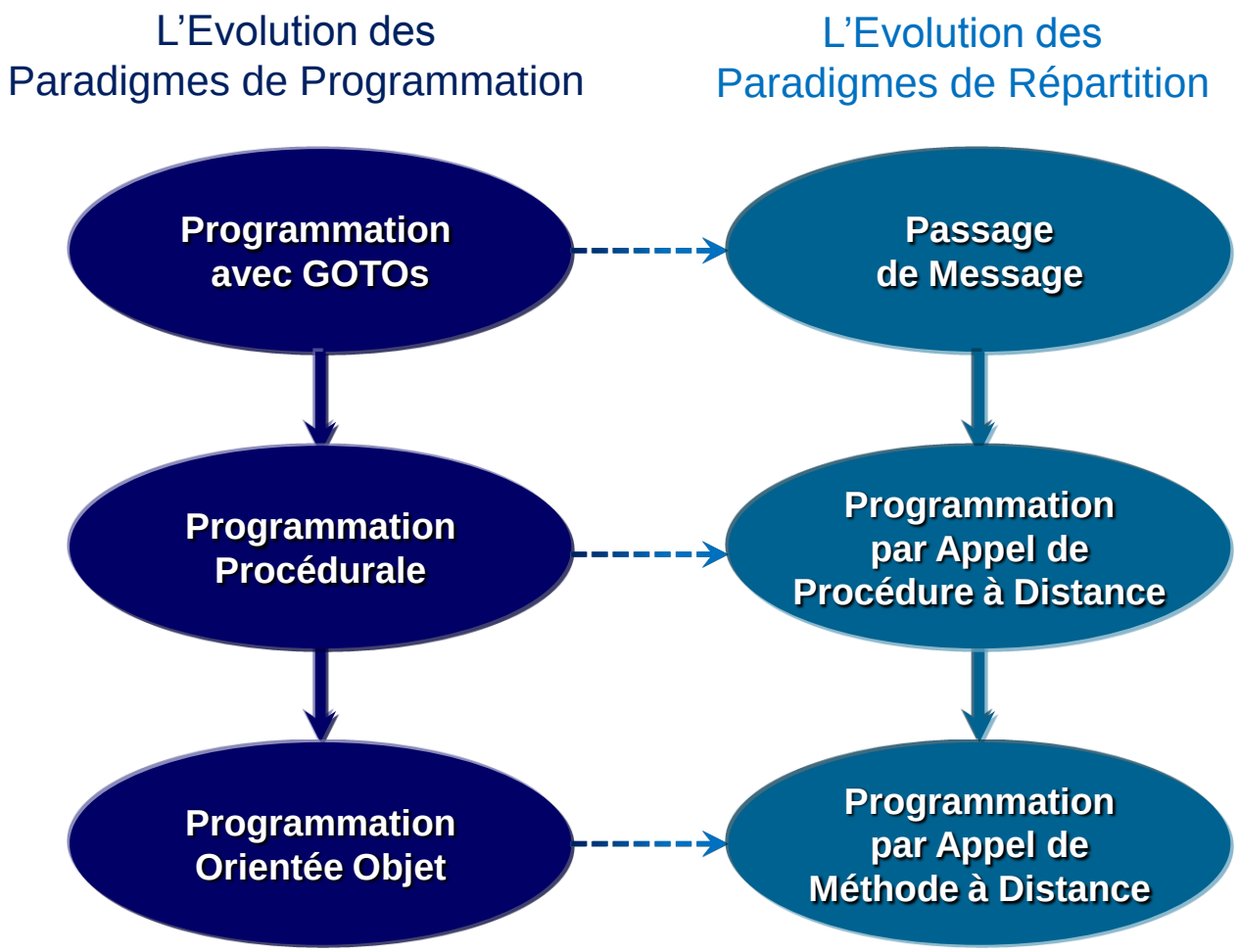
- DSA (annexe des systèmes répartis d'Ada95)
  - Réduit la frontière entre réparti et monolithique (local et uni-thread)
  - Services de nommage et de localisation internes
  - Annulation d'appel de sous-programmes distants
  - Référence sur des sous-programmes distants

- SOAP (Simple Object Access Protocol)
  - Protocole léger de communication d'informations structurées
  - Le plus souvent : RPC à base de HTTP + XML (protocole uniquement)

modèle RPC pour les Objets répartis – Appel de Méthode à Distance

- Approches dépendantes du langage de programmation
  - Modula-3 : extension au langage Modula-2
  - RMI (« Remote Method Invocation ») : extension de Java
  - DSA (annexe systèmes répartis d'Ada95) : extension au langage Ada95
- Approches indépendantes du langage de programmation
  - CORBA (OMG IDL - langage de définition d'interface)
  - DCOM, .NET (Microsoft IDL)

Motivations – Distribution (Analogie)



Motivations - Hétérogénéité

- Appel de méthodes à distance sur objets répartis hétérogènes, indépendamment des :
  - Langages de programmation
  - Systèmes d'exploitation
  - Plates-formes d'exécution
  - Fournisseurs de technologie
  - Protocoles réseau
  - Formats des données

⇒ Consensus pour l'interopérabilité.

- Mais, de nombreux intergiciels restent spécifiques à
  - un langage, un vendeur, ou un Système d'Exploitation (SE)

Java RMI -
Remote Method Invocation

Ressources :

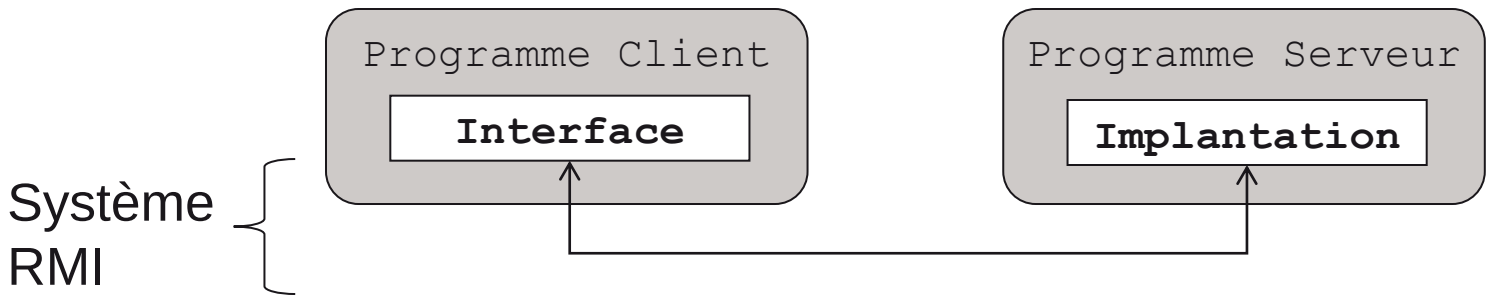
- http://docs.oracle.com/javase/tutorial/rmi/index.html
- http://docs.oracle.com/javase/6/docs/technotes/guides/rmi
- …

Objectifs de Java RMI

- Définir une architecture distribuée qui s'intègre dans le langage Java de façon transparente, donc qui permette :
  - L'invocation de méthodes sur des objets situés sur des machines virtuelles différentes, de façon similaire à une invocation locale ;
  - L'utilisation de l'environnement de sécurité Java classique lors de l'exécution d'applications distribuées ;
  - L'affranchissement du programmeur de la gestion de la mémoire, en définissant une extension distribuée au garbage collector.

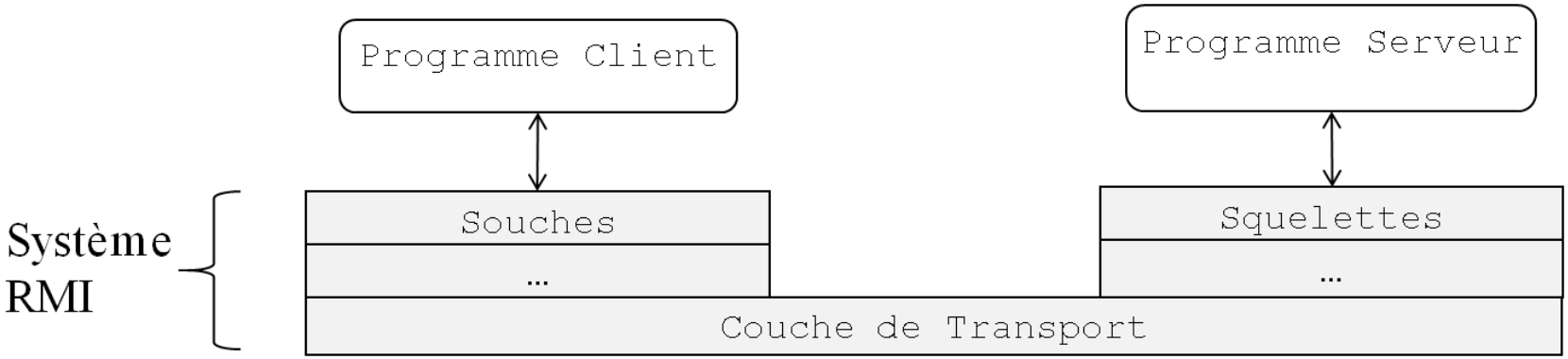
Architecture RMI - Interface

- Séparation de deux concepts :
  - *Définition* d'un comportement – Java Interface
  - *Implantation* d'un comportement – Java Class
- Dans un système réparti :
  - Le client s'intéresse à la description d'un service – Interface
  - Le serveur fournit l'implantation - Class



Architecture RMI – plusieurs couches

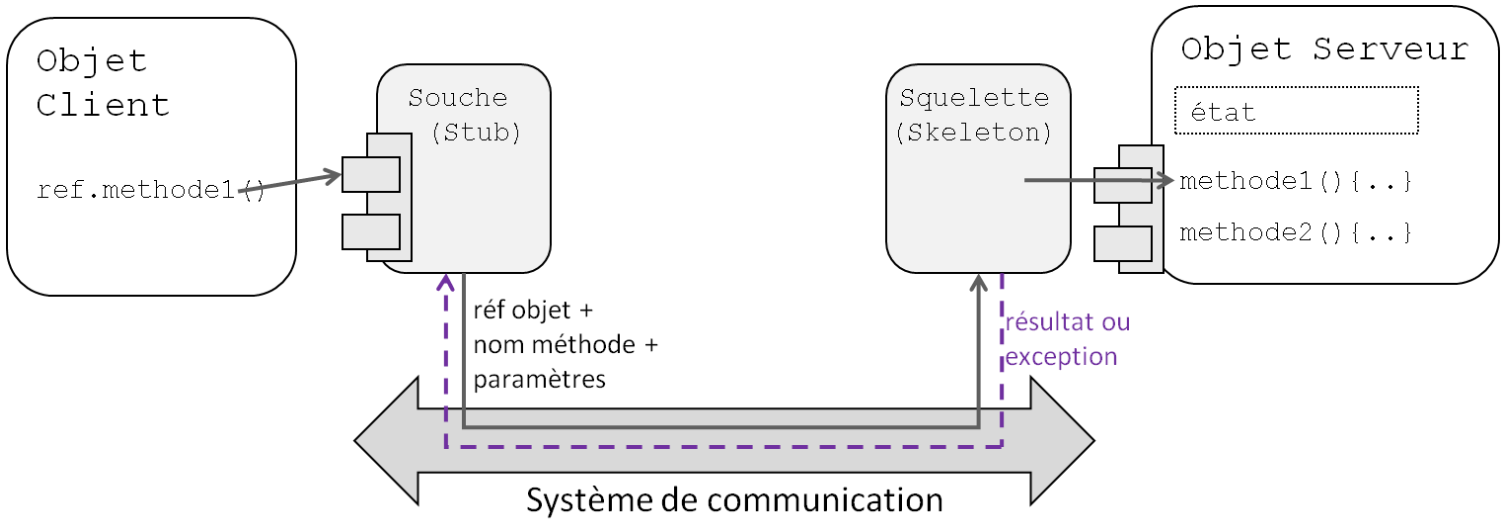
- Souches (stubs) et Squelettes (skeletons)
  - Juste en dessous du programme développeur
  - Transforment les appels Java en messages réseau
- …
- Couche de Transport
  - Fait la connexion entre les JVMs
  - Utilise le protocole TCP/IP

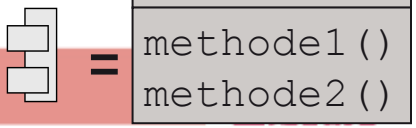


Architecture RMI – détails & exemple

- Stub (Souche)
  - Fait la transition entre l'appel de l'Objet Client et le Réseau
- Skeleton (Squelette)
  - Fait la transition entre le Réseau et l'appel vers l'Objet Serveur

> *Visibles ou invisibles par le développeur, selon la version de JDK*





Objet réparti via RMI - stubs et skeletons

- Extension du mécanisme d'appels de méthodes à la répartition :
  - Souche (« stub ») - Objet « Proxy » qui :
    - Reprend la même signature que l'Objet Serveur
    - Est manipulé par le client
  - Squelette (« skel ») - code côté serveur pour :
    - Traiter la requête, et
    - Transmettre la réponse à l'Objet Client
  - RMI + JVM ajoutent la logique de traitement de la requête:
    - Localisation de l'objet, construction de la requête, transmission, …

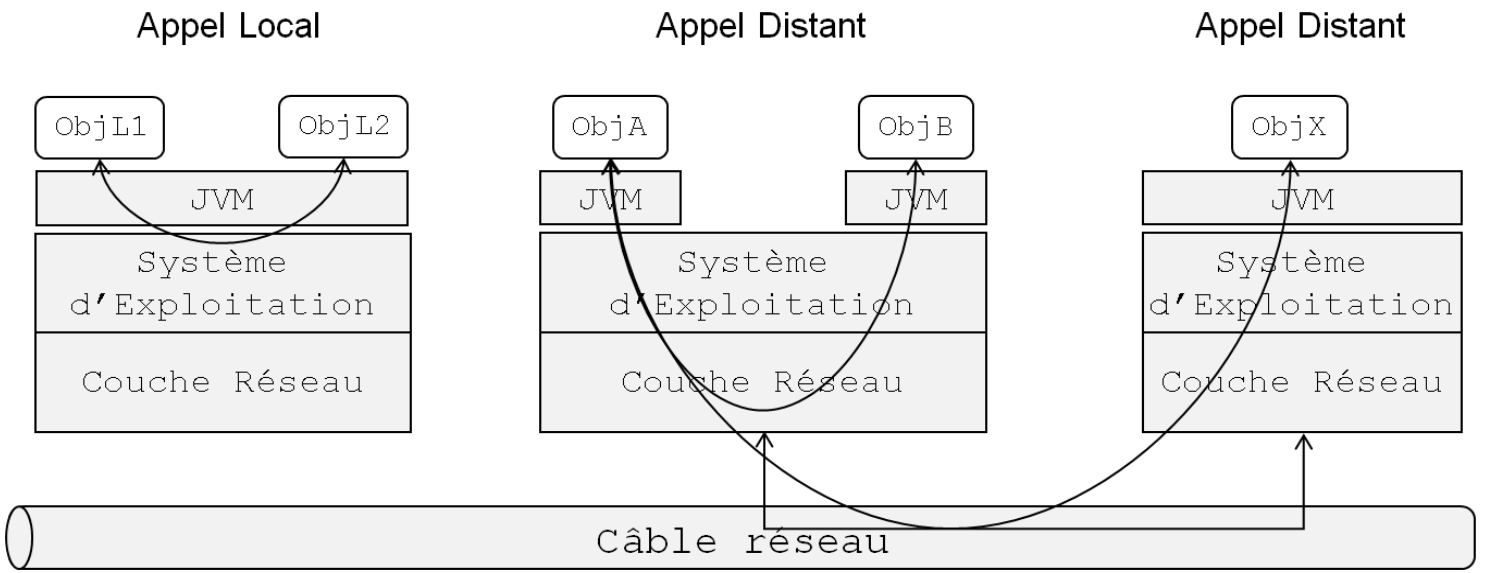
Objet réparti via RMI - terminologie & définitions

- Un objet distant (**`Remote Object`**) est un objet dont les méthodes peuvent être invoquées par des clients situés sur des JVM différentes ;
- Cet objet distant est manipulé au travers d'interfaces (**`Remote Interfaces`**) qui listent les méthodes que les clients pourront invoquer sur ces objets distants ;
- Une référence distante (**`Remote Reference`**) est une référence qui permet d'adresser un objet situé sur une JVM différente ;
- Une invocation de méthode distante (**`Remote Method Invocation`**) appelle une méthode définie dans une interface d'objet distant. Elle a la même syntaxe qu'une invocation de méthode locale.

Passage d'appels distants

Objets s'exécutant sur :

- La même JVM → appel local
- Des JVMs différentes sur la même machine → appel distant
  - Passant par la couche réseau de la machine
- Des machines différentes → appel distant
  - Passant par une connexion réseau entre les machines



Différences entre un appel local ou distant

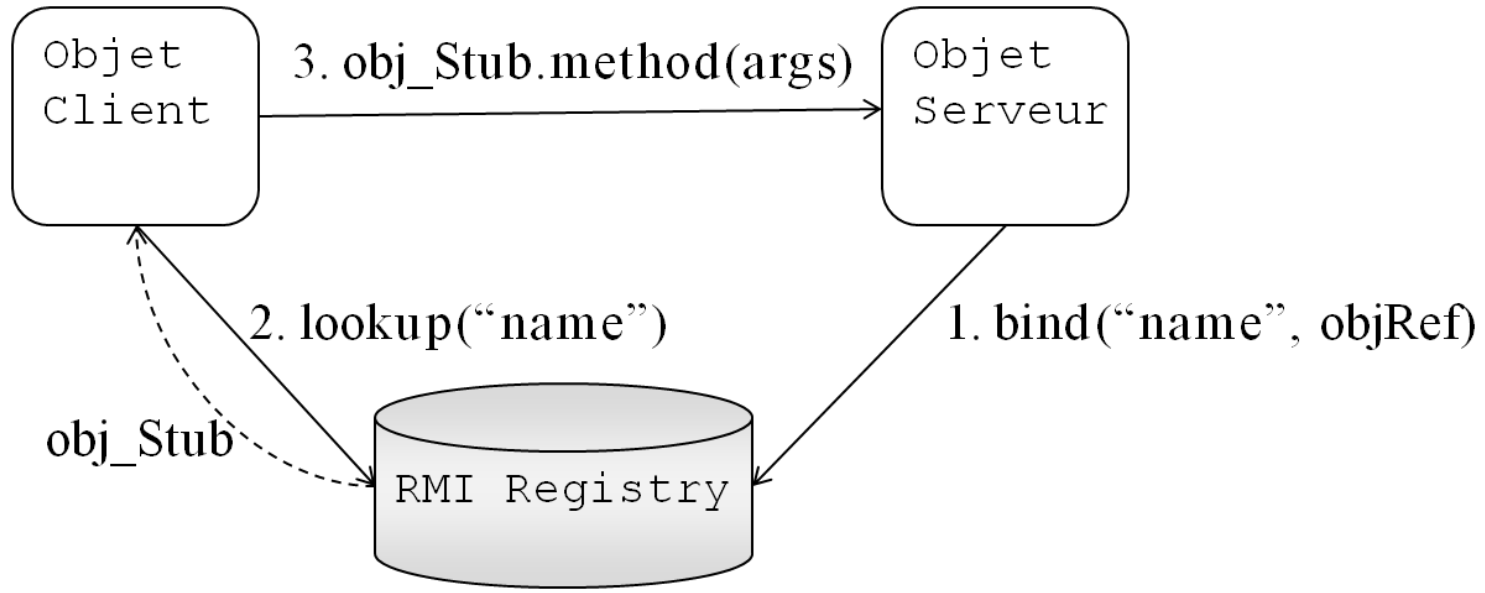
- Les clients manipulent des Proxies qui implémentent les **`Remote interfaces`** - pas directement les objets qui implémentent ces interfaces ;
- Les paramètres de type simple (**`int,`** …) sont passés par copie ;
- Les paramètres de type référence (Objet) sont sérialisés (**`Serializable`**) et passés par copie => différence des paramètres locaux dont les références sont passés par copie ;
- Les objets distants sont passés par référence : les méthodes sont bien exécutées sur l'objet lui-même, quelle que soit la machine virtuelle dans laquelle il réside ;
- Les méthodes distantes provoquent des exceptions supplémentaires : les clients doivent traiter des exceptions liées au caractère distribué des méthodes qu'ils invoquent ;
- Le *bytecode* n'est transmis à la JVM qu'au moment de son utilisation.

Comment trouver les objets ?
RMI Registry

- Les clients utilisent un service de nommage basique – RMI Registry (`rmiregistry`), afin de trouver les références des objets distants
- Au moins un RMI Registry doit se trouver sur chaque machine où se trouvent des objets distants ; (le port d'écoute peut être configuré)
- Les objets distants doivent être enregistrés avec le RMI Registry local afin de pouvoir être trouvés et appelés ;
- Les clients doivent connaître l'adresse (machine et port) du RMI Registry où sont enregistrés les objets distants qu'ils souhaitent appeler
- **Note** : les Services de Nommage (SN) plus avancés n'imposent plus cette contrainte de localisation – une seule instance du SN (à une adresse bien connue) peut être partagée par tous les clients du système réparti.

Appel distant

- Etapes de connexion entre un client et un serveur



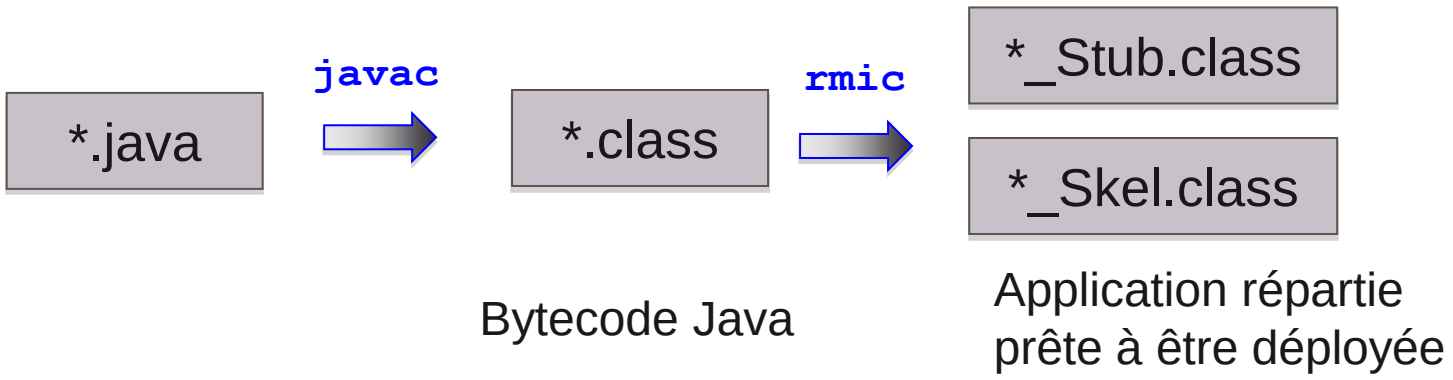
Fonctions RMI

RMI assure les fonctions de :

- **Localisation** des objets distants
  - Le serveur enregistre les objets qu'il exporte auprès de **rmiregistry** en utilisant `bind()`
  - Le client trouve des références sur ces objets exportés et récupère les stubs correspondants en donnant leurs **noms** lors des appels à `lookup()`
- **Communication** transparente grâce au protocole RMI
- **Téléchargement dynamique de bytecode** à partir d'un URL donné (ex : utilisation de serveurs http, ftp, …).

Compilation Java vs Java + RMI

- Le cas réparti nécessite la génération d'éléments de code supplémentaires : codage/décodage des requêtes, gestion des tables de nommages, correspondance requêtes/code utilisateur
- La chaîne de compilation est étendue pour générer le code supplémentaire, qui va scruter le bytecode Java, et rechercher les éléments utiles (implantant les interfaces remote, serializable, …)



rmic – optionnel pour les versions récentes de Java

- Une fois que l'on a écrit est compilé une classe qui implémente les interfaces distantes, on crée le stub et le squelette correspondants grâce à **rmic** :
  - Ces deux classes sont rangées dans des fichiers dont les noms sont créés par ajout de _Stub au nom de la classe pour la souche (« stub »), et _Skel pour le squelette (« skeleton »)

- Notes
  - L'option `-keepgenerated` conserve les sources du stub
  - `Object` contient une méthode `getClass()` qui donne la classe de tout objet : en ajoutant les suffixes adéquats, on obtient les noms du stub et du squelette

Java RMI - 1er exemple

Hello World

A. Diaconescu, L. Pautet & B. Dupouy

Exemple de base

- Soient les fichiers : `Hello.java, HelloServer.java` du côté serveur, et `HelloClient.java` du côté client.
- Contenu de `Hello.java` : l'interface (ce que « voit » le client):

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

  - `Remote` signifie que les méthodes de cet objet Hello doivent être appelables depuis une JVM autre que la JVM locale.

- `HelloServeur.java` implémente cette interface :

```
public class HelloServer implements Hello {
    … …
}
```

Exemple : canevas du serveur

```
//e.g. dans une méthode main :

// Donner un nom au service distant
static final String HELLO_SERVICE_NAME = "HelloService";

// Creation de l'objet qui va être invoque par les clients
HelloServer myServer = new HelloServer();

// export de l'instance myServer (sur un port anonyme - 0)
// obtenir le stub de l'objet distant
Hello stub = (Hello)UnicastRemoteObject.exportObject(myServer, 0);

// Obtenir une référence vers le remiregistry local
//(args[0] donne le port sur lequel écoute le rmiregistry)
int rmiPort = new Integer(args[0]).intValue();
Registry registry = LocateRegistry.getRegistry(rmiPort);

// Enregistrer le service auprès de rmiregistry
registry.rebind(HELLO_SERVER_NAME, stub);
```

Le port sur lequel écoute le RMI Registry

Le nom de l'objet serveur

Visualiser le stub

- Après la compilation du serveur

  il faut faire : **`rmic HelloServer`** , pour avoir :

  - le stub - `HelloServer_Stub.class` - destiné aux clients, et
  - seulement pour les anciennes versions de Java :
    le squelette - `HelloServer_Skel.class`

- Utiliser l'option **`-keepgenerated`** pour obtenir le code source du stub :
  - **`rmic -keepgenerated HelloServer`**

Exemple de base : canevas du client

- Voici comment invoquer une méthode sur l'objet distant :

```
// Donner le nom du service demandé
static final String HELLO_SERVICE_NAME = "HelloService";

// Obtenir une référence vers le remiregistry distant
Registry registry = LocateRegistry.getRegistry(host, port);

// Obtenir une référence vers l'objet distant (via le stub local)
Hello hello = (Hello)registry.lookup(HELLO_SERVICE_NAME);

// Invoquer des méthodes de l'objet distant
message = hello.readMessage();

// Imprimer le message à la console
System.out.println("Message from remote HelloService: " + this.message);
```

Exécution basique de l'exemple (avec partage de code via NFS)

- En étant toujours dans le même répertoire :
  - Lancer **rmiregistry** en arrière plan ;
  - Lancer **le serveur** sur la même machine (<laMachine>) que celle où tourne rmiregistry ;
  - Lancer **le client** sur une machine quelconque, mais en étant dans le même répertoire pour retrouver les stubs : java HelloClient <laMachine>

- Le partage du bytecode (pour les stubs) est assuré par NFS :
  - Le client et le serveur, même s'ils ne sont pas sur la même machine, partagent le même système de fichiers.

- Dans le cas général, il faut faire transiter les stubs par un serveur publique - ex : http, ftp, ….

Synthèse

- Côté serveur
  - Ecrire :
    - Les interfaces "prototypant" les méthodes distantes (extends java.rmi.Remote)
    - Les classes qui les implémentent
    - Une méthode (ex : main) pour créer des instances et les enregistrer avec rmiregistry
  - Lancer :
    - rmic (optionnel) : création du stub et du squelette à partir des fichiers .class des implémentations
    - rendre accessibles par un serveur Web ou Ftp les objets à télécharger (ex : bytecode de stubs)
    - rmiregistry : l'enregistrement des objets distants sera fait lors de l'appel à bind
    - Lancer l'application java du serveur, avec des paramètres de sécurité, ..etc.
- Côté Client
  - Ecrire :
    - L'appel à lookup() sur le rmiregistry pour obtenir le stub de l'objet distant
    - Les appels à l'objets distants via le stub
  - Lancer :
    - Comme un programme java local, avec des paramètres de sécurité en plus

Déploiement de l'application

- Pour lancer un serveur :

```
java -Djava.rmi.server.codebase=http://perso.enst.fr/~$USER/tp/…
     -Djava.security.policy=java.policy
     HelloServer <port> <message>
```

  - -server.codebase donne le nom du serveur web d'où les stubs seront téléchargés,
  - -java.policy donne les droits d'accès qui seront vérifiés par le Security Manager - voici le contenu de ce fichier (donner tous les droits, non-recommandé) :

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

Java RMI - 2ème exemple

Agent distribué

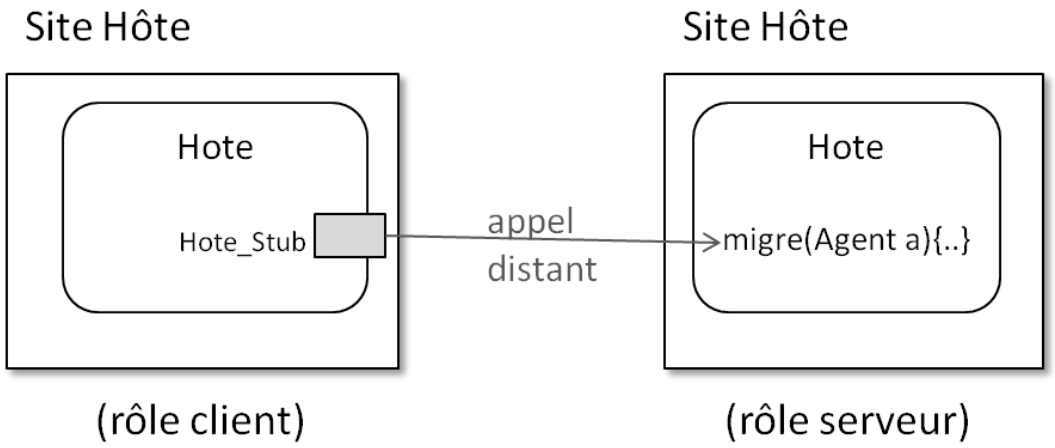
A. Diaconescu, L. Pautet & B. Dupouy

Exemple: agent mobile

- Un client voulant acheter un produit va créer un **agent** au lieu d'interroger lui-même tous les sites détenteurs de magasins vendant le produit.
- L'agent donnera au client le nom du site qui propose le meilleur prix.
- Mise en œuvre :
  - Le client va lancer sa demande depuis un site appelé `Initiator`.
  - Il initialise un tableau de sites à parcourir, puis
  - Il invoque à distance sur le premier site la méthode `migrate()`, à laquelle il a passé l'agent en argument.

Scénario

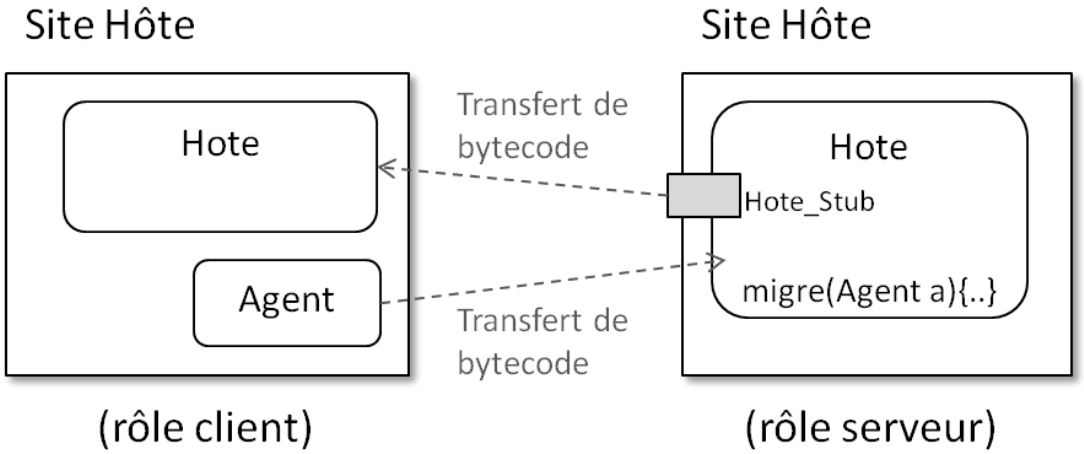
- Un client (Initiateur) va envoyer un Agent interroger successivement plusieurs serveurs (Hôtes)



  - Chaque Hôte prendra successivement le rôle de serveur et de client
  - Le site Initiateur démarre et termine le processus

Transfert de bytecode

- Remarquez les différents objets téléchargés :
  - Le Stub (de Hôte et d'Initiateur) qui permet l'exécution distante de la méthode **`migrate`**
  - L'Agent – transmis en paramètre de la méthode **`migrate`**



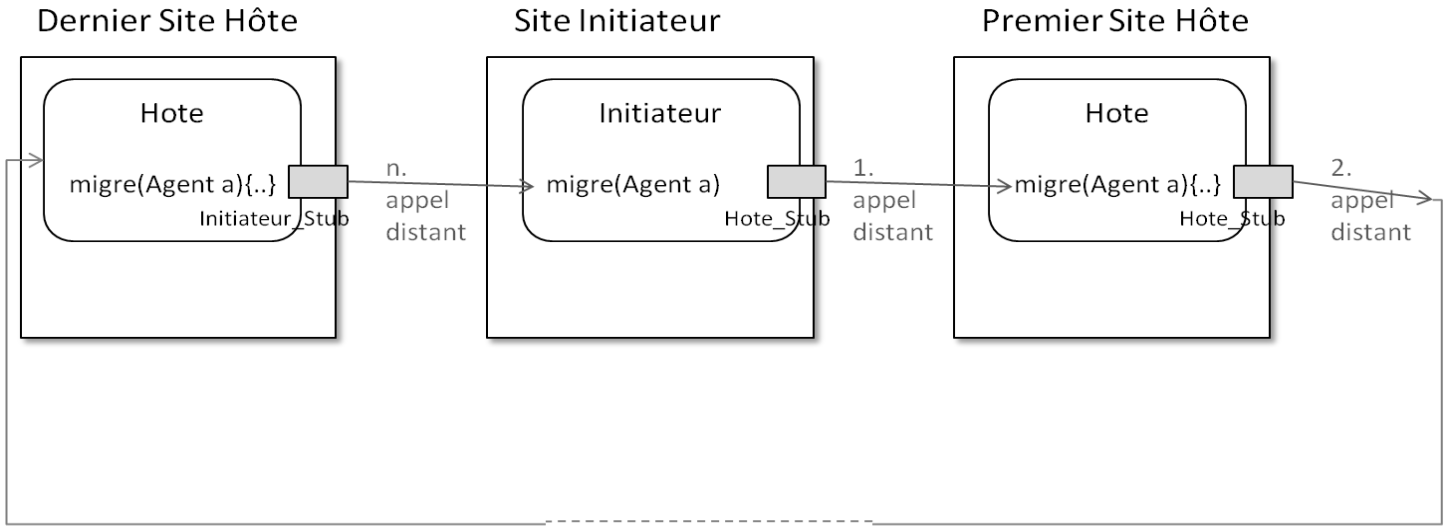
NOTE: il ne faut pas confondre l'envoi de *l'instance agent* (en paramètre de la méthode `migrate`) avec le transfert du *bytecode de la classe Agent*

Récapitulatif du fonctionnement

- La liste des Sites à parcourir se trouve dans l'agent

- Le Site Initiateur appelle le premier Site Hôte (dans la liste)
- Chaque Site Hôte appelle le prochain Site Hôte (en suivant la liste des Sites de l'Agent)
- Le dernier Site Hôte appelle l'Initiateur

- L'agent est transmis en paramètre de chaque appel distant – **`migrate( agent )`**
- La méthode **`migrate`** de l'Initiateur est particulière - elle demande d'afficher les résultats obtenus



Descriptif des transferts de l'agent

- Observez ci-dessous le circuit parcouru par l'agent :

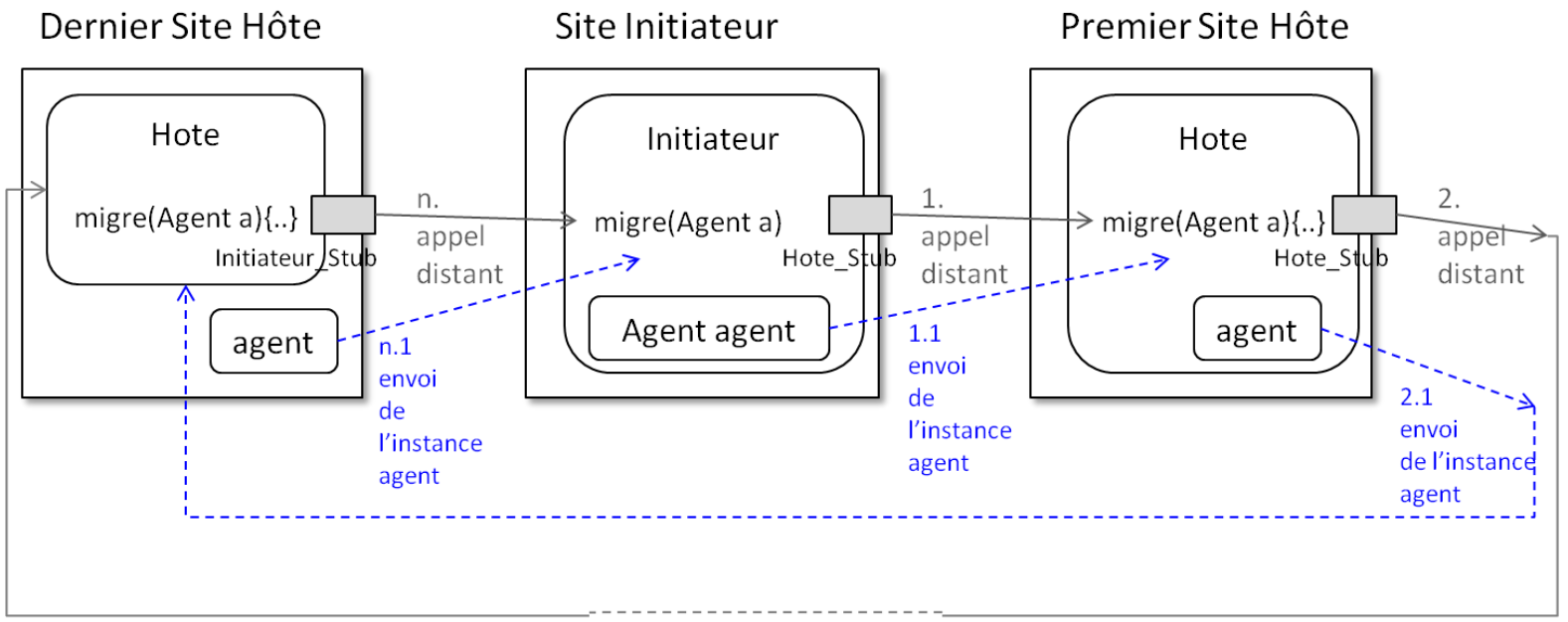
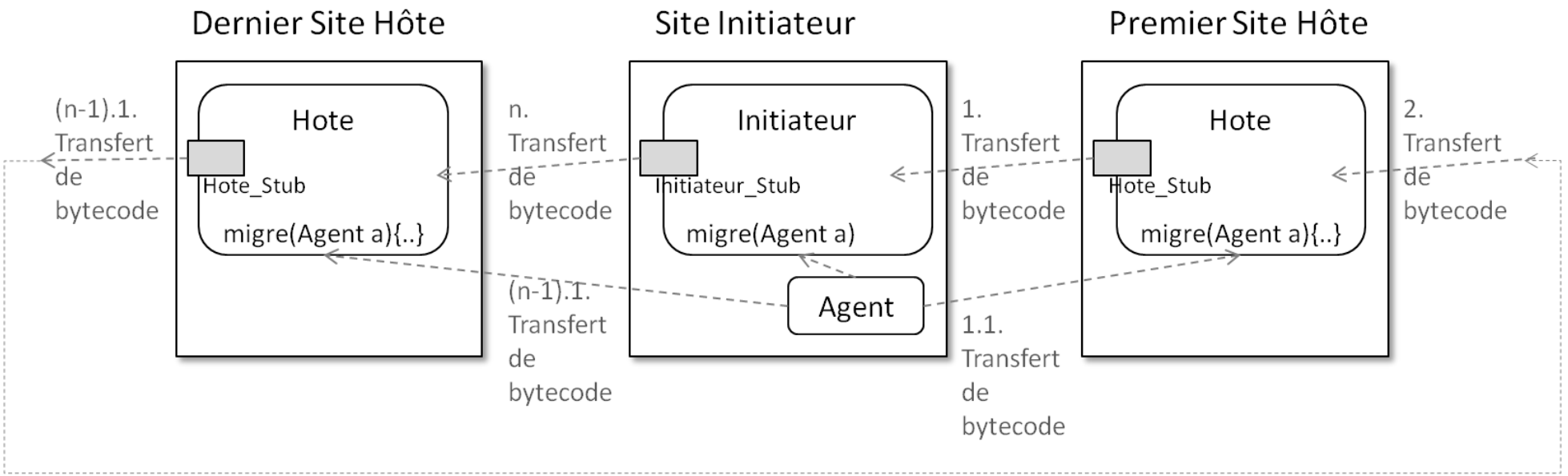
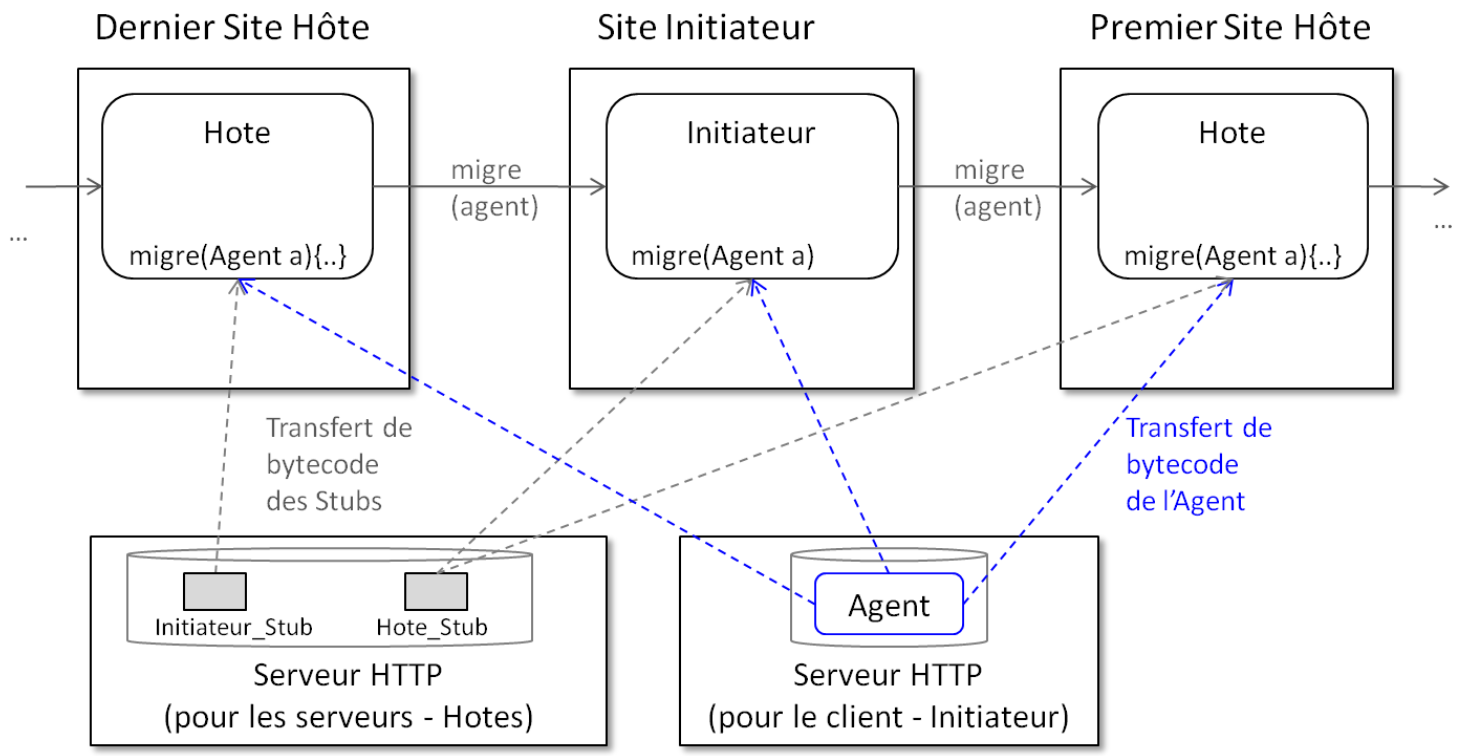


Schéma des transferts de bytecode - codebase local via NFS -

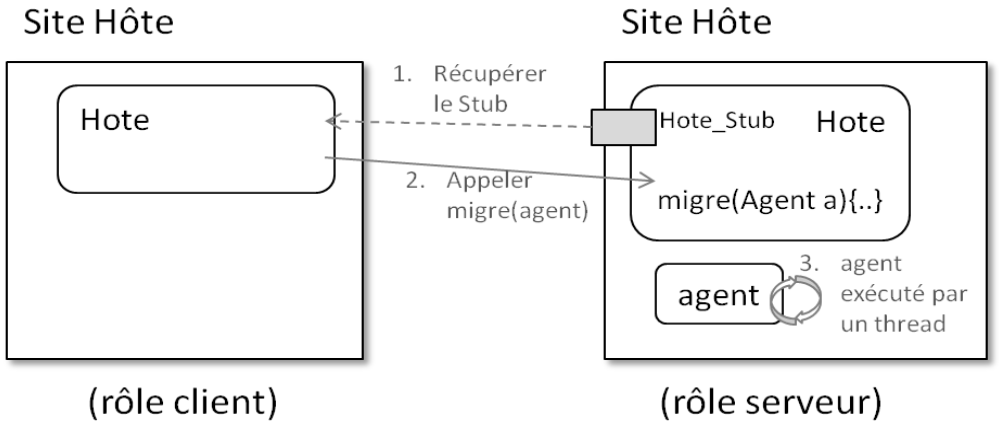


Résumé des transferts de bytecode - codebase sur serveur http -



Le serveur

- Le serveur exécute la fonction **`migrate`** pour le client :
  - il récupère alors le code de l'agent.
  - il le fait exécuter par un Thread qui appelle lui-même migrate sur le site suivant :



- Voici l'interface fournie par le serveur (**`Host.java`**) :

```
public interface Host extends Remote {
    void migrate(Agent a) throws RemoteException;
}
```

- Le code d'Agent n'est pas présent lors de l'écriture de ce serveur

Le serveur

- Implémentation de l'interface (**`HostImplem.java`**) :
  - La méthode migrate invoquée à distance crée un Thread - elle rend donc la main sans attendre la fin du traitement de l'agent (exécution puis migration).

```
public class HostImplem implements Host{

      public HostImplem(String fileName) throws RemoteException{
             super();
             this.myStore = new Store(fileName);
      }

      public void migrate(Agent a){
             AgentThread myThread = new AgentThread(agent, this.myStore);
             myThread.start();
      }
}
```

L'Agent

- Agent.java est l'interface pour les agents.
- L'objet sera implémenté par le "client" (Initiateur).

```
public interface Agent extends Serializable {
    // Traitement effectue par l'agent sur chaque hôte – ex : getMinPrice
    void traitement();
    // Affiche le résultat des traitements : effectué par
    // l'agent lorsqu'il revient sur le site initiateur
    void displayResult();
    // Renvoie le nom de l'hôte suivant a visiter par l'agent
    String getNextHostName ();
}
```

- Serializable indique que les paramètres utilisés seront sérialisés et normalisés lors de l'appel distant (« marshalling »).

Le Thread qui gère l'agent

- ThreadAgent est le support système offert par un Hôte pour traiter un agent : cette classe définit le thread qui va être lancé chaque fois qu'un agent arrive sur un site.

```
class ThreadAgent extends Thread {

  public void run() {
        // On effectue ici le traitement demandé par  l'agent
        ...
        // On fait ensuite migrer l'agent vers l'hote suivant
        String nextHostName = this.theAgent.getNextHostName();
        ...
        Host nextHost  = (Host) registry.lookup(nextServiceName);
        nextHost.migrate(this.theAgent);
  }
}
```

L'Initiateur

- Implémentation de l'Initiateur, c'est à dire du site "client" qui crée et lance l'agent vers les différents serveurs.
  - La méthode migrate est ici particulière : elle ne déclenche pas l'exécution de l'agent mais lui demande d'afficher les résultats obtenus.

```
public class Initiator implements Host {
   public void migrate(Agent a) {
        a.displayResult();
   }
   public static void main(String args[]) {
        AgentImplem agent = new AgentImplem(rmi-port, product, hosts);
        ...
        firstremoteHost.migrate(agent);
   }
}
```

Initiateur

- Voici la commande pour lancer l'initiateur :

```
java
   -Djava.rmi.server.codebase=http://perso.enst.fr/~$USER/tp-rmi/...
   -Djava.security.policy=java.policy
   Initiator <rmirgistry port> <product> <host> <host> ...
```

- server.codebase donne le nom du serveur web d'où l'agent sera téléchargé,
- le fichier java.policy donne les droits d'accès qui seront vérifiés par le Security Manager