Java Remote Method Invocation

Des Sockets à Java-RMI

- **But**: applications client/serveur, cad
  - Réparties, et
  - multi-threadées coté serveur pour servir chaque client efficacement
- **Solution**: De la programmation (manuelle!) par sockets
  - Orienté flux, avec démarrage d'une thread par client
- … à de l'invocation de fonctions coté serveur déclenchée automatiquement
- Dans un contexte Java, et selon une philosophie SOA:
  - Portabilité
  - Polymorphisme
  - Génération dynamique de code
  - Chargement dynamique de classes, y compris durant exécution

Java-RMI

- RMI signifie Remote Method Invocation
- Introduit dès JDK 1.1
- Partie intégrante du cœur de Java (API + runtime support)
  - La partie publique de RMI est dans java.rmi
- RMI = RPC en Java + chargement dynamique de code
- Mêmes notions de stubs et skeletons qu'en RPC
- Fonctionne avec l'API de sérialization (utilisée également pour la persistance)
- Possibilité de faire interagir RMI avec CORBA et DCOM

Université Nice SOPHIA ANTIPOLIS

Notions de base

- RMI impose une distinction entre
  - Méthodes locales
  - Méthodes accessibles à travers le réseau
  - Distinction dans
    - Déclaration
    - Usage (léger d'un point de vue syntaxe; moins "léger" d'un point de vue sémantique – voir un prochain cours)
- Un objet qui a des méthodes accessibles à distance est appelé objet distant
- Le stub (=proxy) est (type-) compatible avec l'objet appelé
  - Il a la "même tête"
- Le skeleton est générique

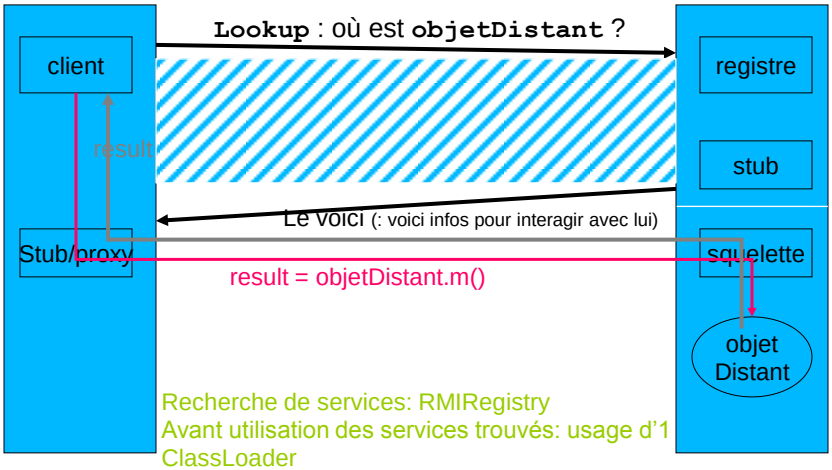
Université Nice SOPHIA ANTIPOLIS

Du vocabulaire et quelques concepts

- Notions de stubs et skeletons (idem qu'en RPC)
  - *Proxy: (person with) authority or power to act for another "mandataire"*
  - *Stub: the short part of something which is let after the main part has been used or left, "morceau restant", "talon/souche"*
    - In distributed programming, the stub in most cases is an interface which is seen by the calling object as the "front-end" of the "remote proxy mechanism", i.e. "acts as a gateway for client side objects and all outgoing requests to server side objects that are routed through it".
    - The proxy object implements the Stub
  - *Skeleton: ossature, charpente*
    - In distributed programming, skeleton acts as gateway for server side objects and all incoming clients requests are routed through it
    - The skeleton understands how to communicate with the stub across the RMI link (>=JDK 1.2 –generic-- skeleton is part of the remote object implementation extending Server class, thanks to reflection mechanisms)

Université Nice SOPHIA ANTIPOLIS

Interaction Objet Client <-> Objet ServDistant



Université Nice SOPHIA ANTIPOLIS

Implémenter un objet distant

- Les seules méthodes accessibles à distance seront celles spécifiées dans l'interface Remote
  - Écriture d'une interface spécifique à l'objet, étendant l'interface java.rmi.Remote
- Chaque méthode distante doit annoncer lever l'exception java.rmi.RemoteException
  - Sert à indiquer les problèmes liés à la distribution
- L'objet distant devra fournir une implémentation de ces méthodes

Implémenter un objet distant

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface MonInterfaceDistante extends Remote {
      public void echo() throws RemoteException;
}
```

Cette interface indique que tout objet qui l'implémentera aura la méthode echo() qui sera appelable à distance

Implémenter un objet distant

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class MonObjetDistant extends UnicastRemoteObject
implements MonInterfaceDistante {

    public MonObjetDistant() throws RemoteException {}
    public void echo() throws RemoteException{
        System.out.println(« Echo »);
     }
}
```

L'objet distant doit implémenter les méthodes de l'interface
Hériter de java.rmi.server.UnicastRemoteObject (autre sol: exportObject)
qui est une classe support dont chaque instance sera associée à un port TCP = point entrée vers l'objet distant
Et avoir un constructeur sans paramètre levant aussi l'exception

Utiliser un objet distant

- Pour utiliser un objet distant il faut
  - Connaître à l'avance son interface !
  - Le trouver! (obtenir un stub/proxy implantant l'interface et indiquant la socket d'entrée vers hôte+port où tourne l'objet)
  - L'utiliser (=invoquer les méthodes offertes)
- RMI fournit un service de nommage permettant de localiser un objet par son nom : le registry (même hôte que l'objet distant)
  - L'objet s'y enregistre sous un nom « bien connu » (de tous)
  - Les clients demandent une référence vers cet objet via ce nom

Utiliser un objet distant

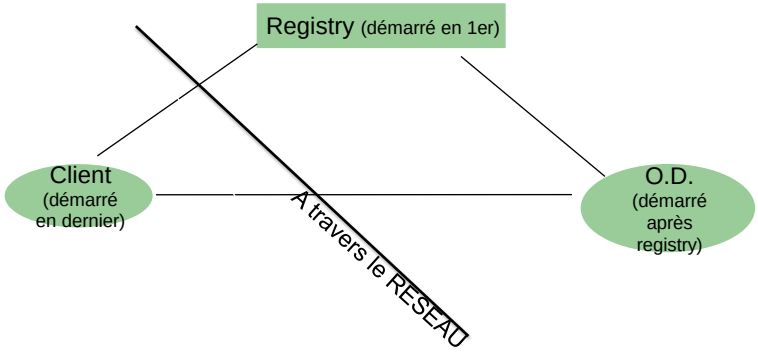
```
import java.rmi.RemoteException;

public class Client {
      public static void main(String[] args) {
         MonInterfaceDistante mod = … // du code pour
                                    //trouver l'objet, cad mod est un stub/proxy vers l'objet
          try {
             mod.echo();
          } catch (RemoteException e) {
            e.printStackTrace();
          }
      }
}
```


Trouver un objet distant

- L'objet doit d'abord s'enregistrer dans le registry
  - Programme lancé préalablement sur la même machine que l'objet distant : rmiregistry
  - Utilise le port 1099 par défaut, sinon ajouter numéro voulu
  - Possibilité de le démarrer depuis l'application (class LocateRegistry)
- Il agit comme un service d'annuaire "en plus simple": serveur de nommage (juste association nom->objet)
- Les noms, selon l'API utilisée: des URLs complètes ou partielles
  - protocole://machine:port/nom
  - Protocole, machine et port sont optionnels
    - Objet toto sur la machine locale: ///toto
- Des méthodes de gestion existent entre autre dans la classe Naming
  - L'objet distant appelle Naming.bind ou Naming.rebind
  - Le client appelle Naming.lookup

Démarrage d'une application RMI

- L'objet distant (O.D.) s'enregistre dans le registry
- Le client demande une référence au registry
- La référence sert ensuite pour appeler les méthodes sur O.D.



Générer les stubs et skeletons

- Une fois l'objet distant écrit, il est possible de générer les classes des stubs (et des skeletons -> plus la peine depuis JDK 1.2)
- Outil fourni dans l'outillage Java: rmic
- Prend le nom complet de la classe distante (package+nom) et travaille sur le fichier compilé (.class)
- Génère 2 (ou juste 1) fichiers (même nom classe _Stub et _Skel)
- Ne met dans le stub que les méthodes spécifiées dans l'interface distante
- Possibilité de voir le code source avec l'option –keep (instructif!)
  - rmic -keep className
- Plus nécessaire depuis JDK1.5 d'invoquer explicitement rmic
  - Invocation à rmic faite côté machine serveur à chaque demande d'enregistrement de la référence de l'objet distant dans le rmiregistry

RMI: la pratique en résumé

- Écrire l'interface distante
- Écrire le code de l'objet distant (une seule classe ou une par item ci-après)
  - Implémenter l'interface (et étendre UnicastRemoteObject)
  - Ajouter le code pour le registry (en général dans le main ou le constructeur)
- Compiler
- Générer les stub et skeleton (optionnel)

- Écrire le client
  - Obtenir une référence vers l'objet distant
  - Utiliser ses méthodes distantes
- Compiler

- Exécuter:
  - Démarrer le rmiregistry PUIS Démarrer le serveur
  - Démarrer le client
  - Debugger :)

Résumé - RMI

- Un objet accessible à distance (objet distant) doit
  - Avoir une interface qui étend Remote et dont les méthodes lèvent une RemoteException
  - Sous classer UnicastRemoteObject et avoir un constructeur sans paramètre levant une RemoteException
- Pour trouver une référence vers un objet distant, on passe par un service de nommage, le RMIregistry

Compléments de RMI

Passage de paramètres

- Le but de RMI est de masquer la distribution
- Idéalement, il faudrait avoir la même sémantique pour les passages de paramètre en Java centralisé et en RMI
- C'est-à-dire passage par copie
  - Copie de la valeur pour les types primitifs
  - Copie de la référence pour les objets (en C, c'est équivalent à passer un pointeur, i.e. la valeur de l'adresse de l'objet en mémoire)
    - Par abus de langage, on dit "passage par référence"
- En Java, on ne manipule jamais des objets!
  - La valeur d'une variable est soit un type primitif, soit une référence (adresse mémoire) vers un objet

Passage de paramètres

```
public void foo(int a) {
    a=a+1;
}
```

```
public class MonInt {
   public int i;
   …..
}
public void foo(MonInt a) {
    a.i=a.i+1;
}
```

```
int x = 10;
foo(x);
// que vaut x ici? … tjs 10…
```

```
MonInt x = new MonInt(10);
foo(x);
// que contient x ici? … 11
```

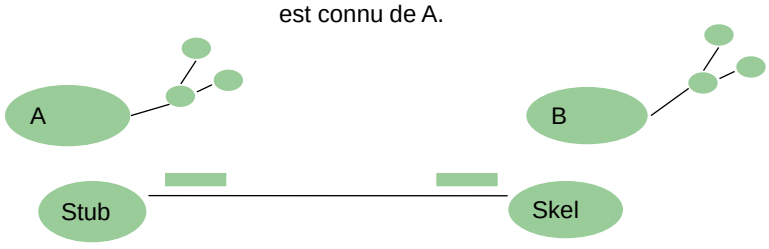

Passage de paramètres

- Peut-on faire la même chose en RMI ?
- Très facile pour les types primitifs, il suffit de les envoyer sur le réseau, ils sont automatiquement copiés
  - C'est le rôle du stub lorsque le client invoque une méthode du serveur, et veut passer les valeurs de ces types primitifs en paramètre
- Plus compliqué pour les objets, car il faudrait
  - Envoyer la valeur de l'objet (facile objet sérialisable)
  - Ramener les éventuelles modifications (traitement ad-hoc)
  - Gestion de la concurrence (des modifs côté serveur) non triviale
- Mais on peut avoir besoin d'un passage par référence
  - RMI le permet, seulement pour des références vers des objets distants
  - C'est quoi une référence vers un objet distant? Son Stub!
  - Il suffit donc de simplement copier le stub

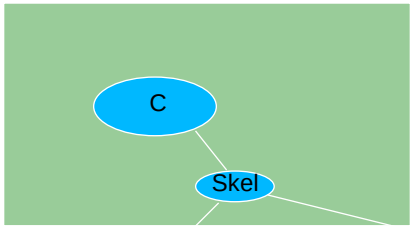
Passage de paramètres

- Un objet référencé non RMI, est passé par copie profonde

Ex: A appelle une méthode sur B, avec un paramètre correspondant à un objet qui est connu de A.

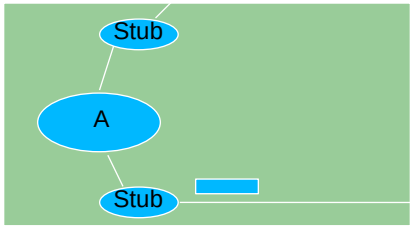


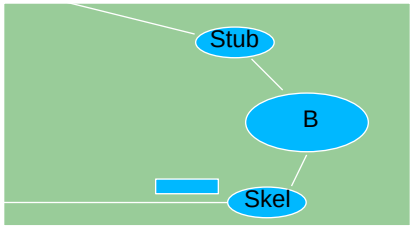
Passage de paramètres



Les références distantes sont passées par copie du stub

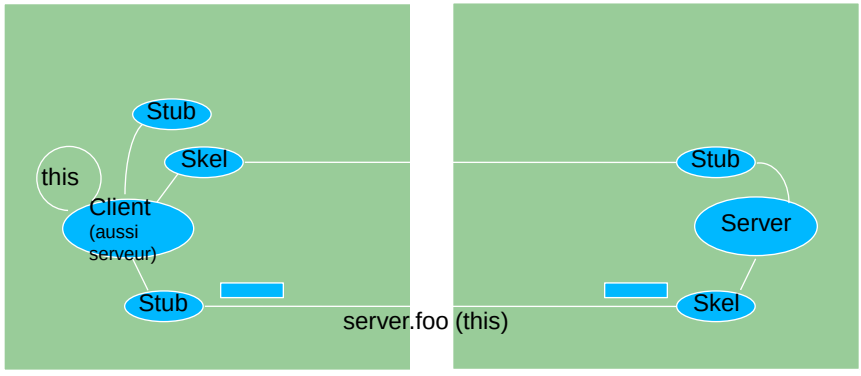
Ex: A appelle une méthode sur B, avec un paramètre correspondant à C (qui est lui même un objet distant)





Passage de paramètres

Une référence locale d'un objet remote est automatiquement convertie en référence distante



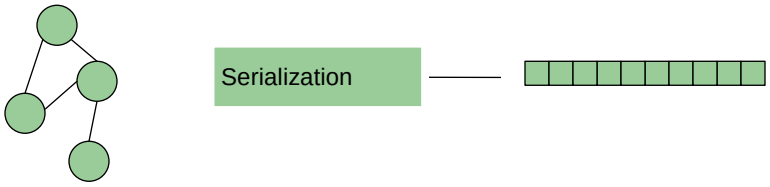
Résumons

- Toute variable de type primitif est passée par copie
- Tout objet est passé par copie
- Tout objet distant (abus de langage, on devrait dire référence distante) est passé par référence

- Mais comment copier un objet?
  - Il ne faut pas copier que l'objet
  - Il faut aussi copier toutes ses références
- Très fastidieux à faire à la main
- Heureusement, une API le fait pour nous: la Serialization

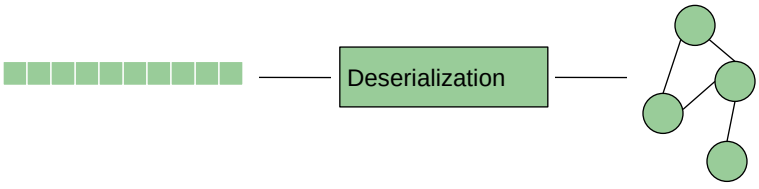
La Sérialisation

- Mécanisme générique transformant un graphe d'objets en flux d'octets
  - L'objet passé en paramètre est converti en tableau
  - Ainsi que tout ceux qu'il référence
  - Processus récursif (copie profonde)
- Fonctionnement de base
  - Encode le nom de la classe
  - Encode les valeurs des attributs



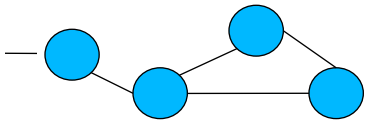
La Désérialisation

- Processus symétrique de la sérialisation
  - Prend en entrée un flux d'octets
  - Crée le graphe d'objet correspondant
- Fonctionnement de base
  - Lit le nom de la classe
  - Fabrique un objet de cette classe (suppose qu'on a donc la classe!)
  - Lit les champs dans le flux, et met à jour leur valeur dans la nouvelle instance



Gestion des cycles

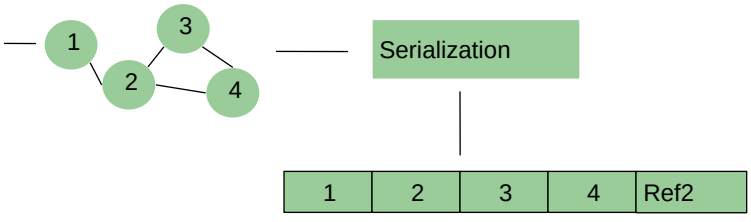
- La sérialisation est un processus récursif
- Que se passe-t-il quand il y a un cycle dans le graphe d'objets?



Un algorithme naïf bouclerait à l'infini
Solution:
Repérer les cycles
La sérialisation se souvient des objets déjà sérialisés
Si on veut en sérialiser un à nouveau, alors met juste une référence

Gestion des cycles

- Le flux contient donc 3 types d'information
  - Le nom de la classe
  - Les valeurs des attributs
  - Des références vers d'autres parties du flux



Utiliser la sérialisation

- Par défaut, un objet n'est pas sérialisable
  - Problème de securité: la sérialisation ignore les droits d'accés (les champs même private sont quand même sérialisés…)
  - Levée d'une NotSerializableException
- Il faut donc explicitement indiquer qu'un objet est sérialisable
- Marquage au niveau de la classe
  - Toutes les instances seront sérialisables
  - Les sous classes d'une classe sérialisable sont sérialisables
- Utilisation de l'interface java.io.Serializable
  - Interface marqueur, aucune méthode à implémenter

Utiliser la sérialisation

- RMI fait appel à la sérialisation
  - Totalement transparent
- Mais on peut aussi l'utiliser manuellement
  - Très pratique pour copier des objets
- Étapes
  - Bien vérifier que les objets sont sérialisables
  - Créer des flux d'entrée et de sortie (input et output streams)
  - Utiliser ces flux pour créer des flux objets (object input et object output streams)
  - Passer l'objet à copier à l'object output stream
  - Le lire depuis l'object input stream

Exemple: Comment utiliser la sérialisation

```
static public Object deepCopy(Object oldObj) throws Exception
{
  ObjectOutputStream oos = null;
  ObjectInputStream ois = null;
  try {
    ByteArrayOutputStream bos = new ByteArrayOutputStream();
     oos = new ObjectOutputStream(bos);
     // serialize and pass the object
    oos.writeObject(oldObj);
    oos.flush();

     ByteArrayInputStream bin = new ByteArrayInputStream(bos.toByteArray());
      ois = new ObjectInputStream(bin);
    // return the new object
    return ois.readObject();
  }
  catch(Exception e)  {
    System.out.println("Exception in ObjectCloner = " + e);
    throw(e);
  }
  finally {
    oos.close();
    ois.close();
  }
}
```


Contrôler la sérialisation

- Marquer une classe avec l'interface Serializable indique que tout ses champs seront sérialisés
- Pas forcément acceptable
  - Sécurité
  - Efficacité (pourquoi copier ce qui pourrait être recalculé plus rapidement?)
- Possibilité de contrôle plus fin
  - Marquage d'attributs comme étant non sérialisables: mots clé *transient*
  - Donner à un objet la possibilité de se sérialiser

Contrôler la sérialisation

- Pour modifier la sérialisation par défaut, il faut implémenter 2 méthodes dans la classe de l'objet
  - writeObject() : sérialisation
  - readObject() : désérialisation
- Leur signature est
  - private void writeObject(ObjectOutputStream s) throws IOException
  - private void readObject(ObjectInputStream o) throws ClassNotFoundException,IOException
- Elles seront automatiquement appelées et remplaceront le comportement par défaut
- On écrit dans ces méthodes du code spécifique à la classe dont l'objet est instance

Contrôler la sérialisation

- Dans les méthodes readObject/writeObject il est possible de tout faire
  - pas de limitation théorique
  - Manipulation/modification des attributs de l'objet possibles
- Basé sur les flots (streams de java.io)
  - Implémentation FIFO
  - Donc à faire dans le même ordre que l'écriture
- Symétrie
  - Normalement, lire tout ce qui a été écrit
  - En pratique, RMI délimite le flux et évite les mélanges

Écriture - Lecture

- Utilisation des méthodes de ObjectOutputStream et ObjectInputStream
  - Types primitifs
    - {write|read}Double, {write|read}Int…
  - Objets
    - {write|read}Object
    - Provoque une nouvelle serialization
- Possible de rappeler l'ancienne implémentation
  - Méthodes defaultWriteObject() et defaultReadObject() des streams
  - Très pratique pour ajouter une fonctionnalité

Exemple 1: reproduire le comportement par défaut

```
public class Defaut implements Serializable {
  public Defaut() { }

  private void writeObject(ObjectOutputStream s) throws
                                    IOException {
      s.defaultWriteObject();
    }

private void readObject(ObjectInputStream s) throws
                  IOException, ClassNotFoundException {
    s.defaultReadObject();
  }
}
```

Exemple 2: sauvegarde d'un entier

```
public class Defaut implements Serializable {
   private int valeur;
   private double valeur2
   public Defaut() { }

  private void writeObject(ObjectOutputStream s) throws
                                        IOException {
       s.writeInt(valeur);
    }

private void readObject(ObjectInputStream s) throws
                    IOException, ClassNotFoundException {
     valeur = s.readInt();
  }
}
```


Sérialisation et héritage

- Les sous classes d'une classe sérialisable sont sérialisables
- Mais une classe peut-être sérialisable, alors que son parent ne l'est pas
  - La sous-classe est responsable de la sauvegarde/restauration des champs hérités
  - Lors de la désérialisation, les constructeurs sans paramètres seront appelés pour initialiser les champs non sérialisables
  - Le constructeur vide de la classe parent sera appelé
- Source de bugs très difficiles à identifier

RMI avancé

Répartition des classes

- RMI distingue deux types d'objets
  - Ceux accessibles à distance
  - Les autres
- Ils sont souvent sur des machines différentes (un client et un serveur)
- Comment sont réparties (+-placées "en dur") les classes (initialement)
  - Côté Client:
    - Implémentation du client
    - Interface distante avec toutes les classes nécessaires pour décrire les méthodes de cette interface
    - Classe du Stub (objet stub, lui, est téléchargé auprès du registry par exemple, ou suite retour de méthode) si pas générée à la volée
  - Côté Serveur:
    - Interface Distante
      - avec toutes les classes nécessaires pour décrire les méthodes de cette interface,
    - Implémentation du serveur avec les éventuelles sous-classes nécessaires (sous classes pour les paramètres des méthodes remote)

Téléchargement de classes

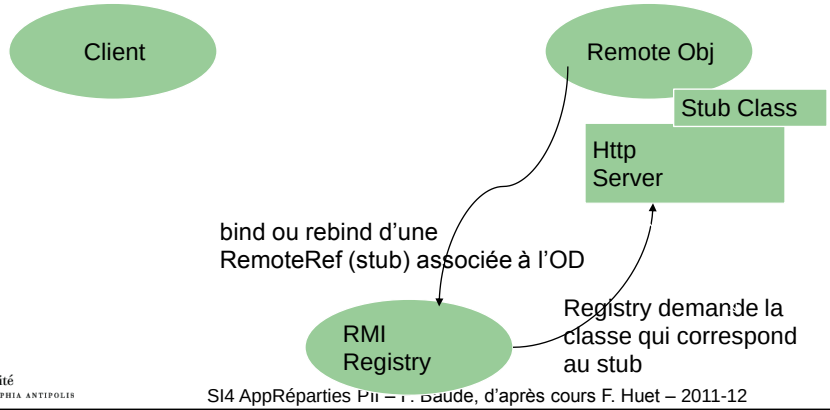
- Dans un monde parfait
  - Les classes sont distribuées
  - Rien ne change, tout est connu
- En pratique
  - Les classes sont *plus ou moins bien* distribuées là où on pense qu'on en aura besoin
    - Certains ordinateurs n'ont pas les classes nécessaires
  - Ex: Appel d'une méthode distante avec en paramètre une sous classe de la classe déclarée dans la signature distante
    - Le serveur doit télécharger le .class dispo côté machine client pour connaitre cette sous-classe
- Solution: pouvoir télécharger les classes manquantes

Téléchargement de classes

- Mécanisme fourni par RMI
- Utilise HTTP
  - Permet de passer tous les firewalls
  - Mais nécessite un serveur HTTP
- Principe:
  - Les flots de sérialisation sont annotés avec des *codebase*
  - Ils indiquent où peut être téléchargée une classe si nécessaire
  - Lors de la désérialisation, si une classe manque, le serveur HTTP indiqué par le codebase est contacté
  - Si la classe est disponible, le programme continue, sinon, ClassNotFoundException

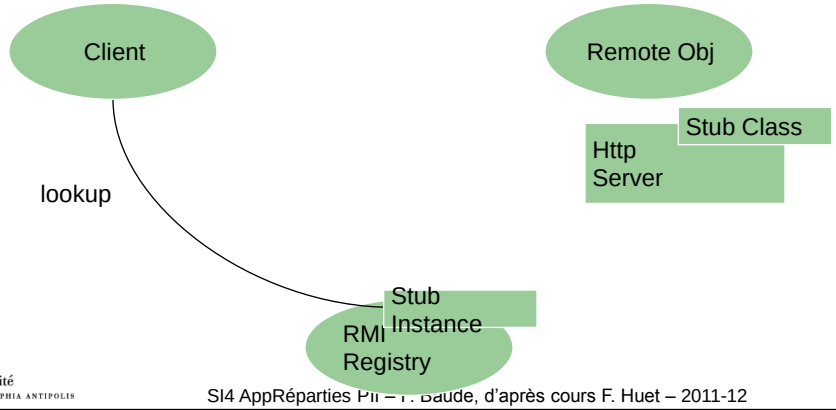
Exemple

- Déploiement minimal : dans ce cas (et sous Hyp. stub créé par rmic)
  - Le stub n'est pas disponible pour le registry ou pour le client
  - Seul le remote object possède la classe de l'objet stub dans son classpath

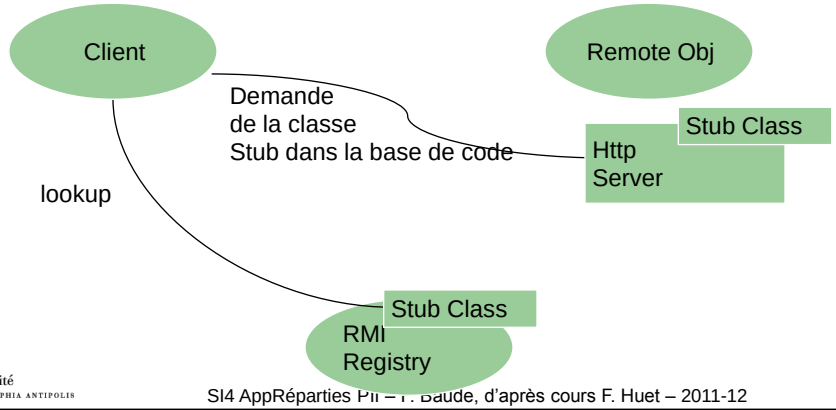


Exemple

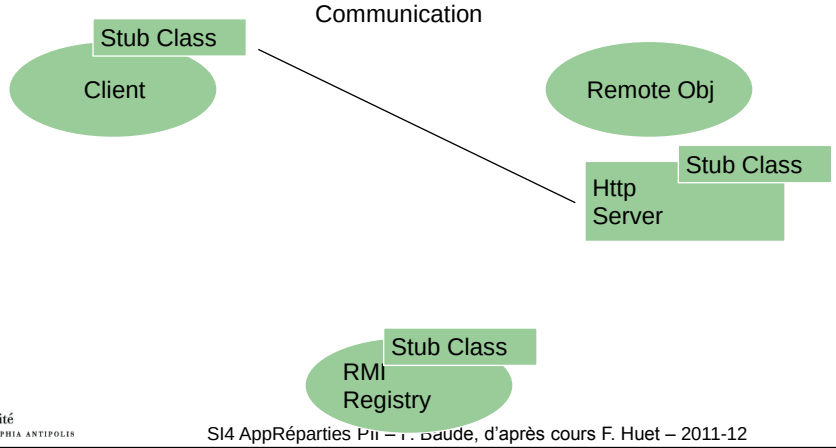
- Quand il contacte le registry, le client doit en plus télécharger la classe correspondant au stub : où est-elle ?



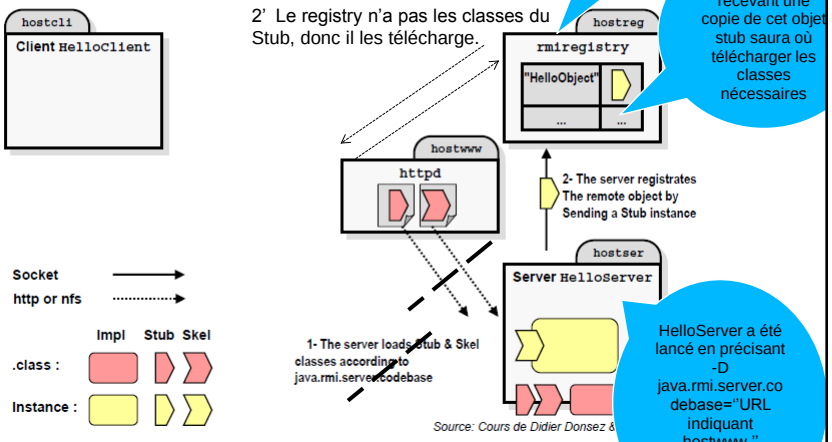
Exemple



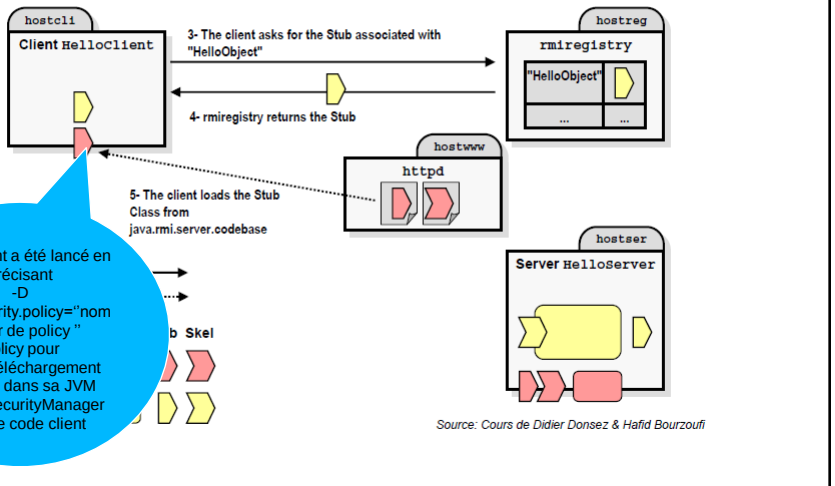
Exemple



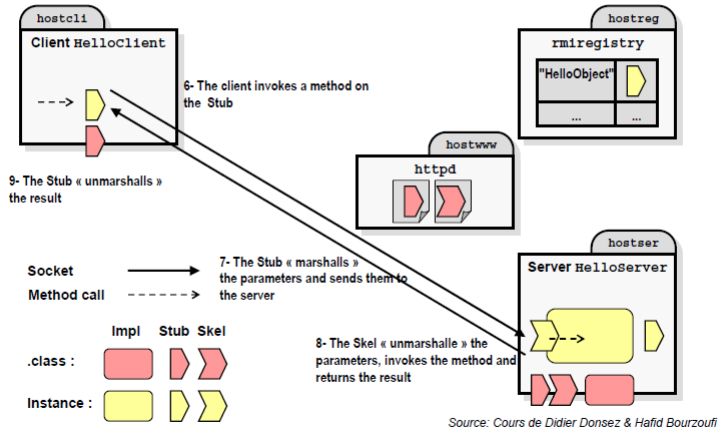
Object Registration



La récupération du Stub

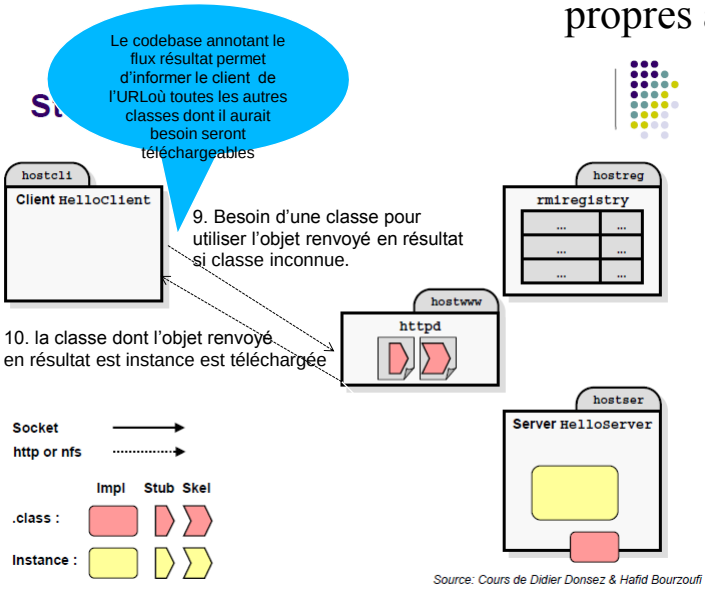


Remote Method Invocation



Source: Cours de Didier Donsez & Hafid Bourzoufi

Et le téléchargement d'autres classes que celles propres à RMI

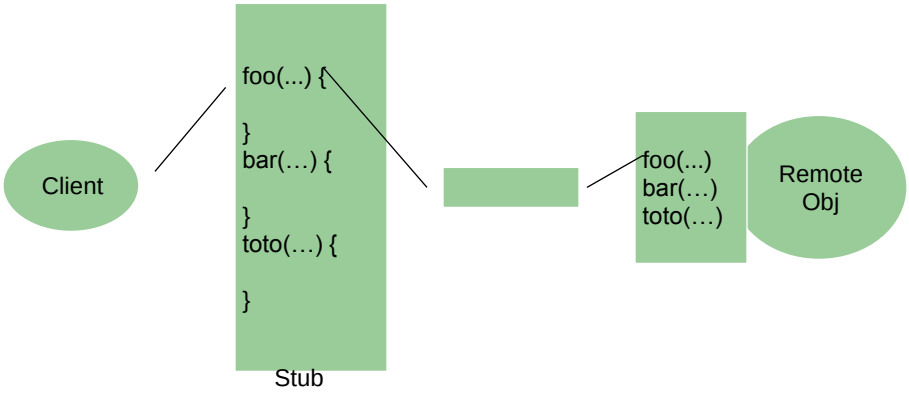


Source: Cours de Didier Donsez & Hafid Bourzoufi

Le Stub

- Le rôle du stub est de se faire passer pour l'objet distant
  - Il implémente l'interface distante
  - Exemple de ce qu'il affiche si on invoque toString():
    Proxy[HelloWorld,RemoteObjectInvocationHandler[UnicastRef [liveRef: [endpoint:[193.51.208.206:10003](remote),objID:[75300dd9:145187ca75d:-7fff, 2812291573724520304]]]]]
- Il doit aussi convertir l'appel de méthode en flot
  - Facile pour les paramètres
  - Pour la méthode, il suffit de la coder sur quelques octets, convention avec le skeleton
- Et attendre le résultat en retour
  - Lecture sur une socket et désérialisation
- Donc un stub, c'est très simple!
  - On peut générer son bytecode à la réception, pour désérialisation
    - A condition d'avoir sur le client le fichier .class de l'interface distante …

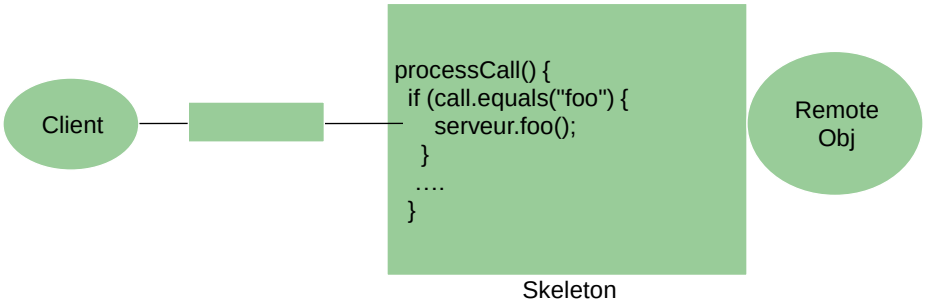
Le Stub



Le Skeleton

- Le skeleton appelle les méthodes sur l'objet distant
- Et retourne le résultat
- Est-il dépendant de l'objet distant?
  - Oui si implémentation statique (naïf)

Le Skeleton (version naïve)



Le Skeleton

- Y'a-t-il moyen de séparer le skeleton de l'objet appelé?
  - Oui, si on a un moyen de dire "je veux appeler la méthode dont le nom est foo" sans l'écrire explicitement
- Reflection
  - Capacité qu'a un programme à observer ou modifier ses structures internes de haut niveau
  - Concrètement, le langage permet de manipuler des objets qui représentent des appels de méthodes, des champs…
  - On fabrique un objet qui représente une méthode, et on demande son exécution
  - Partie intégrante de Java. Vital pour RMI, la sérialization…

Exemple de reflexion

```
String firstWord = "blih";
String secondWord  = "blah";
String result = null;
Class c = String.class;
Class[] parameterTypes = new Class[] {String.class};
Method concatMethod;
Object[] arguments = new Object[] {secondWord};

concatMethod =  c.getMethod("concat",parameterTypes);
result = (String) concatMethod.invoke(firstWord,arguments);
```

Réflexion : utilisation par RMI

Côté Stub: encoder la méthode à invoquer côté serveur en tant qu'un objet

Method m = … getMethod("sayHello"); // serveur offre méthode sayHello qu'on appele en faisant **invoke de m sur la référence distante (notée ref.)**

Aspect générique du skeleton

```
// Stub class generated by rmic, do not edit.
public final class HelloWorldImpl_Stub extends java.rmi.server.RemoteStub
                                        implements HelloWorld, java.rmi.Remote
$method_sayHello_0 = HelloWorld.class.getMethod("sayHello", new java.lang.Class[] {})
public java.lang.String sayHello()  throws java.rmi.RemoteException
  {
    try {
      Object $result = ref.invoke(this, $method_sayHello_0, null, 6043973830760146143L);
      return ((java.lang.String) $result); // appel méthode remote est donc synchrone
    } catch (java.lang.RuntimeException e) {
      throw e;
    } catch (java.rmi.RemoteException e) {
      throw e;
    } catch (java.lang.Exception e) {
      throw new java.rmi.UnexpectedException("undeclared checked exception", e);
    }
```

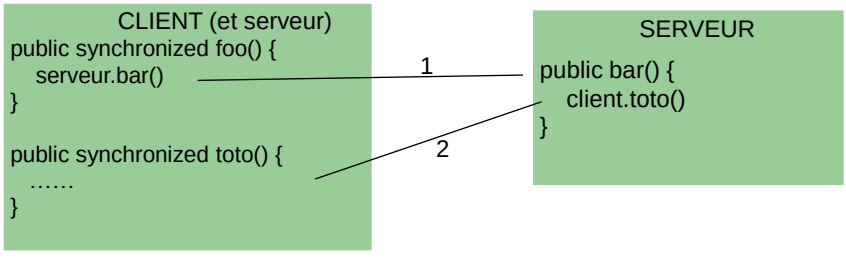

RMI et les threads

- Un appel RMI est initié par un thread côté appelant
- Mais exécuté par un autre thread côté appelé
- Le thread de l'appelant est bloqué jusqu'à ce que le thread du côté appelé ait fini l'exécution
- Si multiples appelants, multiples threads côté appelé
  - Un objet distant est par essence multithread!
  - Il faut gérer la synchronisation des threads (synchronized, wait, notify)
- Pas de lien entre le thread appelant et le thread côté appelé

RMI et les threads

- L'implémentation n'est pas spécifiée
- En pratique
  - Lorsqu'un appel arrive, RMI crée un thread pour l'exécuter
  - Une fois l'appel fini, le thread peut resservir pour un nouvel appel
  - Si multiples appels simultanés, de nouveaux threads sont crées
- Technique du thread pool
- Problème des appels ré-entrants
  - A fait un appel distant sur B, qui fait un appel distant sur A
  - Très courant: cycle dans le graphe d'objets
  - Pas de problèmes dans la plupart des cas (si ce n'est la latence)
  - Gros problèmes si les méthodes sont synchronized: lesquels ?

Interblocage (Deadlock) distribué



2 objets distants communiquent
Cycle dans le graphe d'appels
Le thread ne peut pas entrer dans la méthode toto() tant que l'autre thread n'a pas fini d'exécuter foo
Deadlock!
Parfois difficile à identifier
Pas de vue globale de l'application car elle est répartie
Pas d'information de la part de la JVM

Cours 5ème année (option/Ubinet) : Algorithmique pour les systèmes répartis!

Principe de fonctionnement du GC distribué (DGC)

- Se base sur le GC de chaque JVM concernée
  - Fondé sur un comptage des références : quand un objet n'est plus référencé depuis un objet "racine", alors, il peut être désalloué
- Lorsqu'un stub est envoyé dans une JVM, l'objet distant est donc référencé
  - La JVM qui reçoit ce stub incrémente le compteur de références distantes de l'objet distant (en lui envoyant un message particulier)
- Lorsque le stub n'est plus utilisé, celà a pour effet de décrémenter le compteur de références distantes de l'objet distant
  - [en théorie] Se produit lorsque l'objet dans lequel l'attribut pour stocker le stub est lui-même désalloué, ou bien, l'attribut est modifié
  - [en pratique] Se produit si le client n'utilise pas le stub pendant une certaine durée de temps ('lease'=bail, que l'on peut paramétrer via la propriété java.rmi.dgc.leaseValue=tms).
    - Lorsque le bail pour une référence arrive à 0 (maintenu côté serveur), le GC côté serveur décrémente de 1 le compteur de références
    - Pour éviter ceci, le seul moyen côté client est d'invoquer régulièrement les méthodes!
- Lorsque le compteur de références distantes est parvenu à 0 (donc sur serveur):
  - L'objet distant est considéré comme "garbageable"
  - Il le sera véritablement seulement lorsque plus de références locales vers cet objet

Et les stubs enregistrés dans le rmiregistry ?

- RMI est lui-même un objet distant, qui stocke des stubs
- Tant qu'un stub existe au niveau du registry, cela bloque le GC concernant l'objet distant qu'il référence
  - D'ailleurs, le RMI registry re-initialise son bail régulièrement (pour pas que l'objet serveur soit garbagé !)
- Donc… tant que on a un objet distant pour lequel un *bind* a été effectué, celui-ci est vivant, et occupe des ressources dans sa JVM.
  - Ceci peut être inutile
  - Notion d'objet Activable: est instancié seulement si un client tente d'en invoquer des méthodes à distance.
  - Un stub particulier est stocké dans le RMI registry, même si l'objet n'est pas instancié
    - L'usage d'un tel stub déclenche l'instantiation par le démon rmi
  - http://download.oracle.com/javase/1.4.2/docs/guide/rmi/activation.html