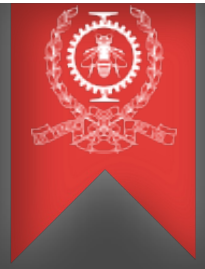


Communication par objets répartis

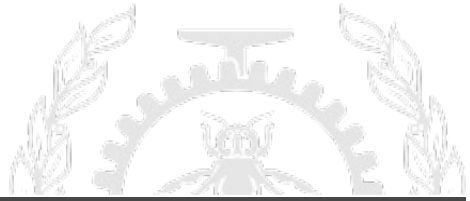
Module 5

INF8480 Systèmes répartis et infonuagique

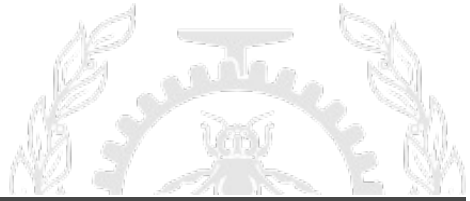
Michel Dagenais

École Polytechnique de Montréal
Département de génie informatique et génie logiciel

Sommaire



Communication par objets répartis

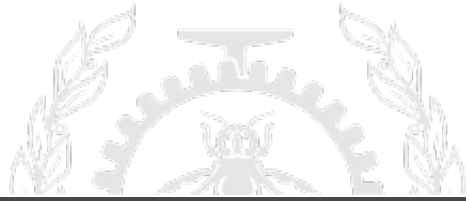


Appel de méthodes à distance

- Modèle objet: références aux objets, interfaces, méthodes, exceptions, ramasse-miettes.
- Proxy: se présente comme un objet local, ses méthodes sérialisent les arguments, transmettent la requête au module de communication et retournent la réponse reçue.
- Répartiteur: reçoit les requêtes dans le serveur et les communique au squelette correspondant.
- Squelette: reçoit les requêtes pour un type d'objet, désérialise les arguments, appelle la méthode correspondante de l'objet référencé et sérialise la réponse à retourner.
- Module de communication: envoi de requête client (type, numéro de requête, référence à un objet, numéro de méthode, arguments), la requête reçue par le serveur est envoyée au répartiteur et la réponse est retournée.

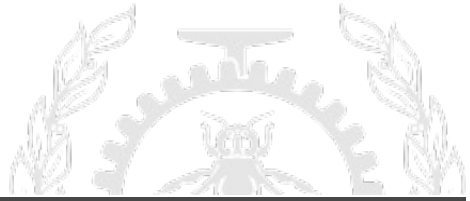
Module des références réseau

- Table des objets importés (pour chaque objet distant utilisé, adresse réseau et adresse du proxy local correspondant),
- Table des objets exportés (pour chaque objet utilisé à l'extérieur, liste des clients, adresse réseau et adresse locale).
- Lorsqu'une référence réseau est recue, son proxy ou objet local est trouvé ou un nouveau proxy est créé et une entrée ajoutée dans la table.
- Lorsqu'un objet réseau local est passé en argument, son adresse réseau le remplace et il doit se trouver dans la table des objets exportés.

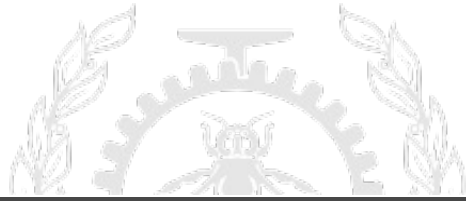


Services pour l'appel de méthodes distantes

- Service de nom: permet d'obtenir une référence objet réseau à partir d'un identificateur/nom.
- Contextes d'exécution: le serveur peut créer un fil d'exécution pour chaque requête.
- Activation des objets: le serveur peut être démarré automatiquement lorsqu'une requête pour un objet apparaˆıt (message augmentation de salaire pour le budget qui est stocké sur disque dans un fichier de chiffrier).
- Stockage des objets persistents.
- Service de localisation.



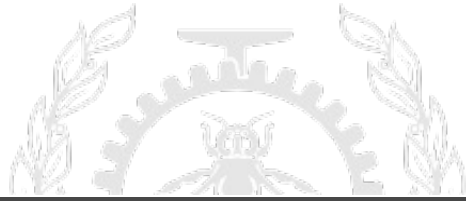
Communication par objets répartis



Gestion de la mémoire en réparti

- Malloc et Free difficiles à appeler au bon moment en réparti!
- Chaque client s'enregistre lorsqu'il utilise un objet et avertit lorsqu'il en a terminé. Le serveur libère un objet si ni lui ni les clients ne l'utilisent.
- Ramasse-miette conventionnel qui travaille en réparti? Très difficile et inefficace puisqu'il faut tout arrêter en même temps lors du ramassage.
- Ramasse-miette local à chaque processus avec notification automatique au serveur lorsqu'un client cesse d'utiliser un proxy.
  - Une référence à un objet O est envoyée de A à B et A cesse de l'utiliser, le message de A (reférence à O inutilisée) peut arriver avant celui de B (utilise O).
  - Cycle de références entre 2 clients ou plus.
  - Si le client disparaˆıt, le serveur doit l'enlever de la liste (limite de validité pour les références, ou besoin de messages de maintien keepalive).

Communication par objets répartis

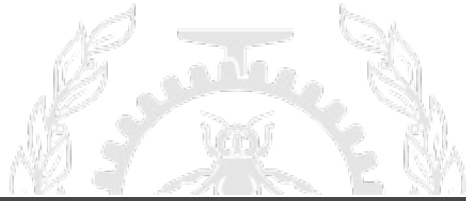


Le Remoting en C#

- Fonctionne entre tous les langages supportés par le CLR: C#, C++, VB...
- Utilise la reflexivité pour générer le code client et serveur dynamiquement.
- Choix d'encodage binaire ou XML.
- Choix de canal TCP ou HTTP.
- Les objets qui héritent de MarshalByRef sont passés par référence à travers le réseau, les autres sont passés par valeur et doivent supporter l'interface ISerializable.
- Un objet déjà créé peut être exporté ou on peut enregistrer un type et l'objet est créé au moment de la requête selon un de deux modes (Singleton, Single call).
- Les références aux objets peuvent avoir une date d'expiration pour éviter d'avoir à vérifier pour les références inutilisées; passé cette expiration elles n'existent plus.

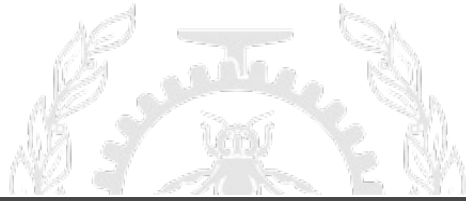
Interface en C# Remoting

```
using System;

abstract public class Server: MarshalByRefObject {
  abstract public string GetInfo();
}
```



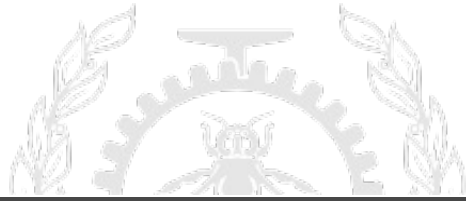
Client en C# Remoting

```
using   System;
using   System.IO;
using    System.Runtime.Remoting;
using   System.Runtime.Remoting.Channels;
using       System.Runtime.Remoting.Channels.Tcp;

class Client
{
  public static void Main(string[] args) {
    int serverPort = 9090;
    host = "localhost";
    TcpChannel channel = new TcpChannel();
    ChannelServices.RegisterChannel(channel);
    Server server = (Server)RemotingServices.Connect(
          typeof(Server),
          "tcp://" + host + ":" + serverPort.ToString() + "/MyServer");
    string info = server.GetInfo();
  }
}
```

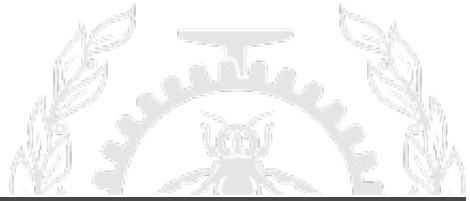


Serveur en C# Remoting

```
using System; using System.IO;
using System.Threading;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

class AServer: Server {
  public override string GetInfo() {
    return "A Server version 0.01"
  }
  public static void Main(string[] args) {
    int serverPort = 9090;
    TcpChannel channel = new TcpChannel(serverPort);
    ChannelServices.RegisterChannel(channel);
    AServer server = new AServer();
    ObjRef serverRef = RemotingServices.Marshal(server, "MyServer",
        typeof(Server));
    Thread.Sleep(20000);
    RemotingServices.Disconnect(server);
    ChannelServices.UnregisterChannel(channel);
  }
}
```



Communication par objets répartis

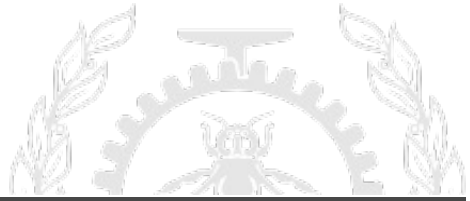


Java RMI

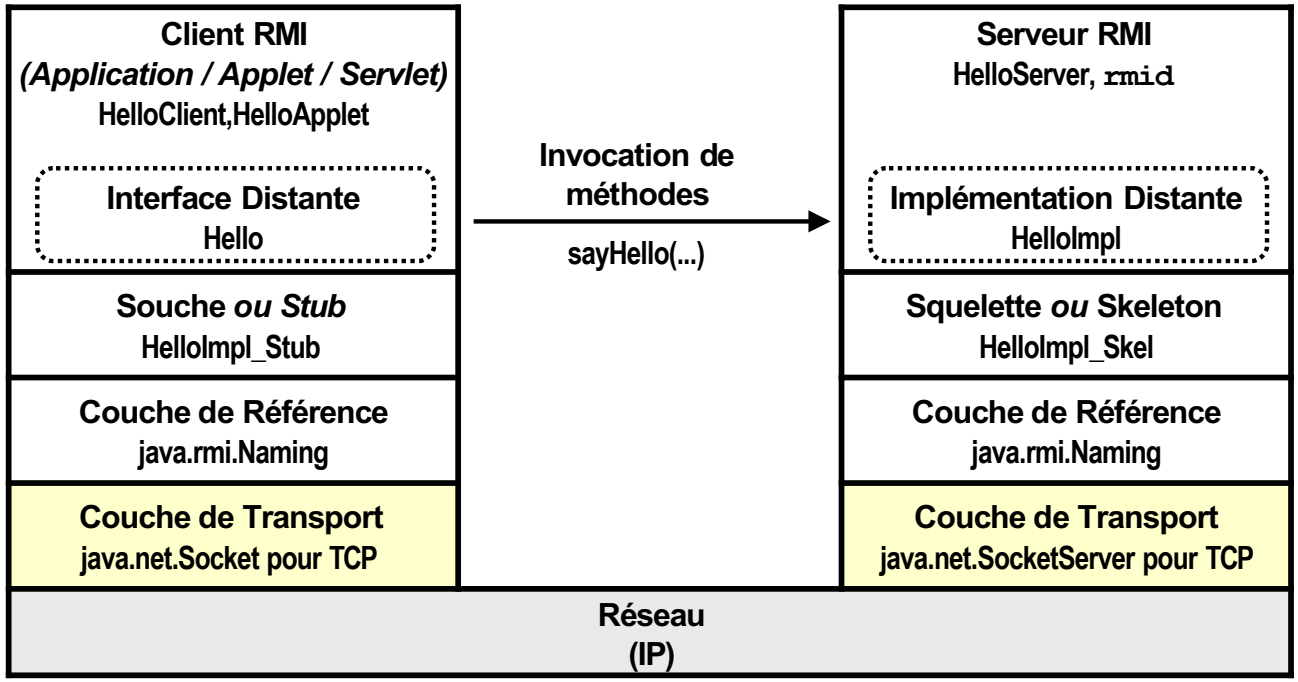
- Ne fonctionne que de Java à Java.
- S'applique à tout type qui implante l'interface Remote.
- Les méthodes doivent accepter l'exception RemoteException.
- Les arguments seront envoyés et la valeur retournée sert de réponse.
- Ces arguments doivent être des types primitifs, des objets réseau, ou des objets qui implantent l'interface Serializable (le graphe complet d'objets rejoints par un argument peut être transmis).
- Le type des objets sérialisés contient une référence à la classe. La classe peut être téléchargée au besoin par le récepteur.
- Le type des objets réseau contient une référence à leur classe proxy (stub) qui peut être téléchargée au besoin.

Java RMI

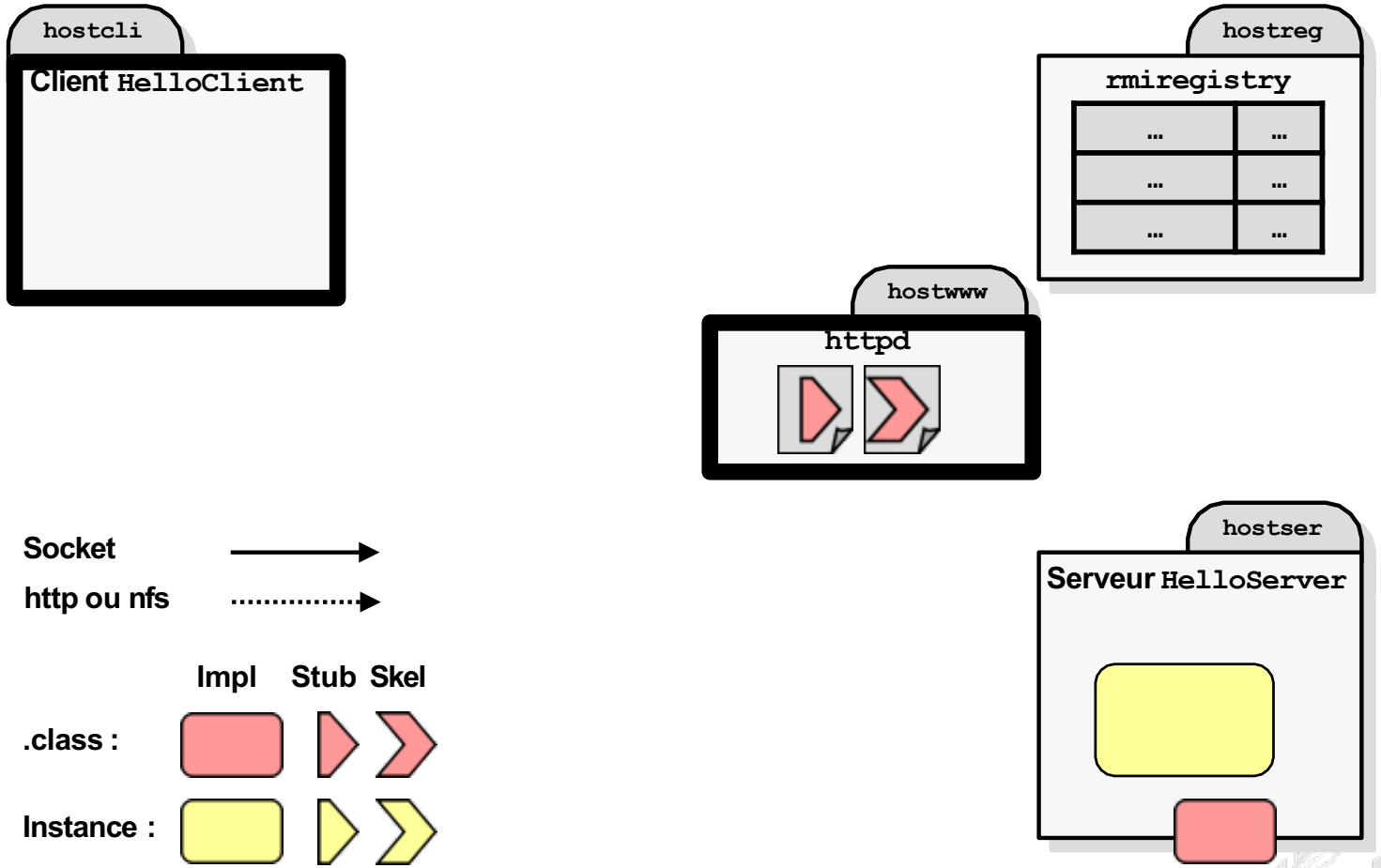
- Le RMIregistry maintient une table (nom, référence réseau) pour obtenir une référence à un objet désiré qui se trouve sur un ordinateur donné. Un nom a la forme: //hostname:port/objectName.
- rmic peut être utilisé pour créer les proxy à partir du code des classes compilées. Le répartiteur est générique et fourni en librairie.
- Les appels distants peuvent être servis par des fils d'exécution différents, surtout s'il viennent de clients (connexions) différents.



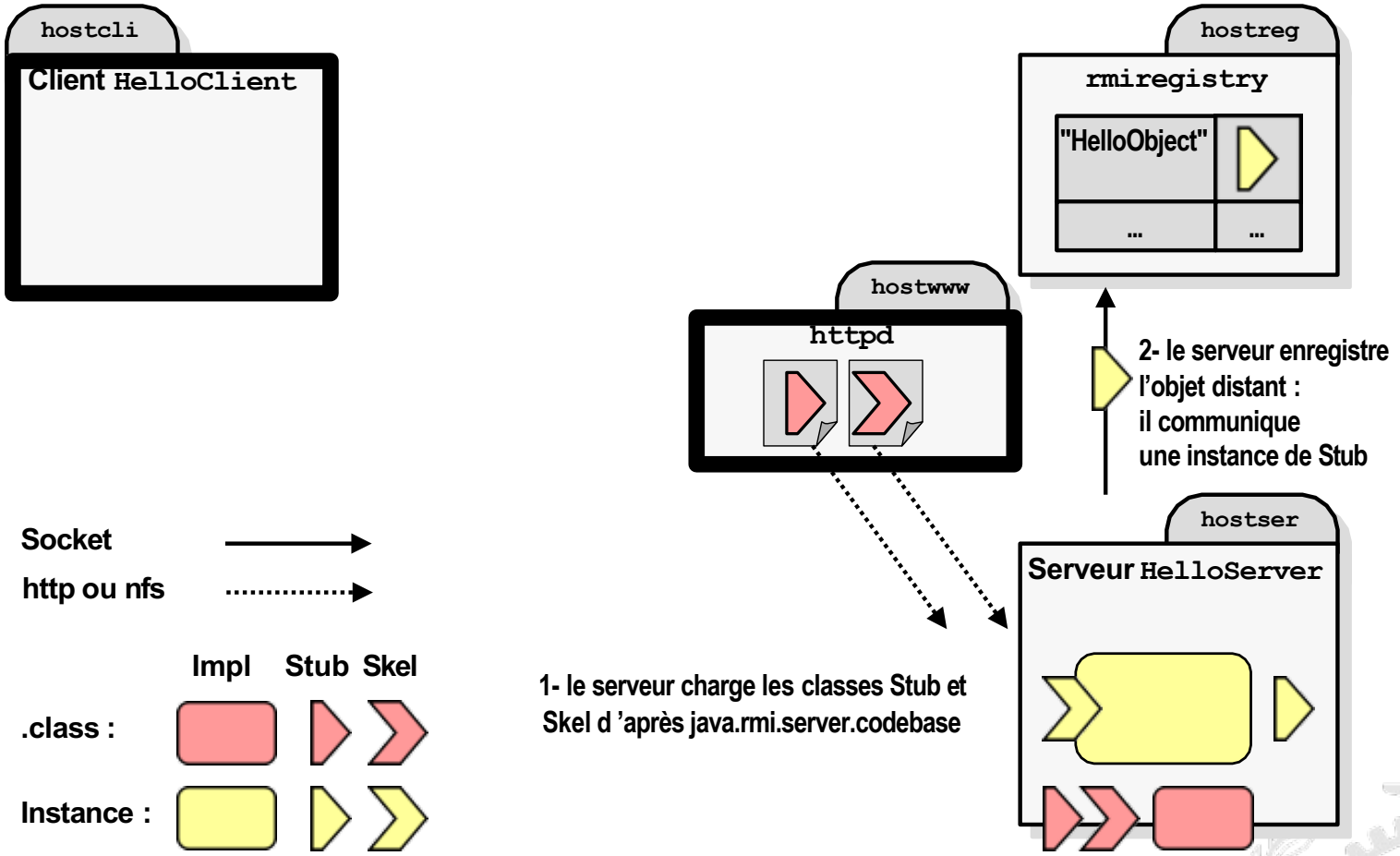
Structure logique des couches RMI



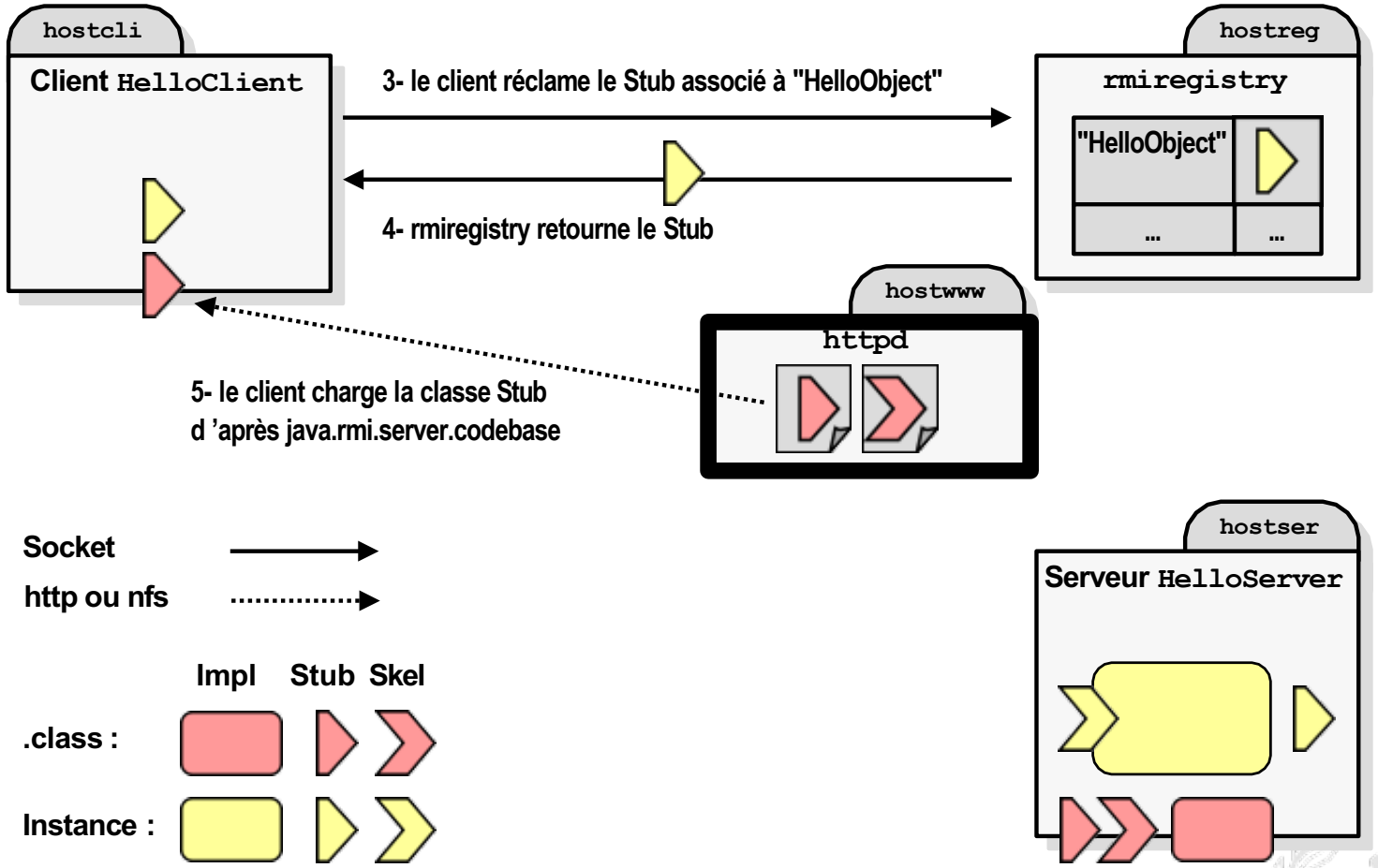
RMI: la configuration



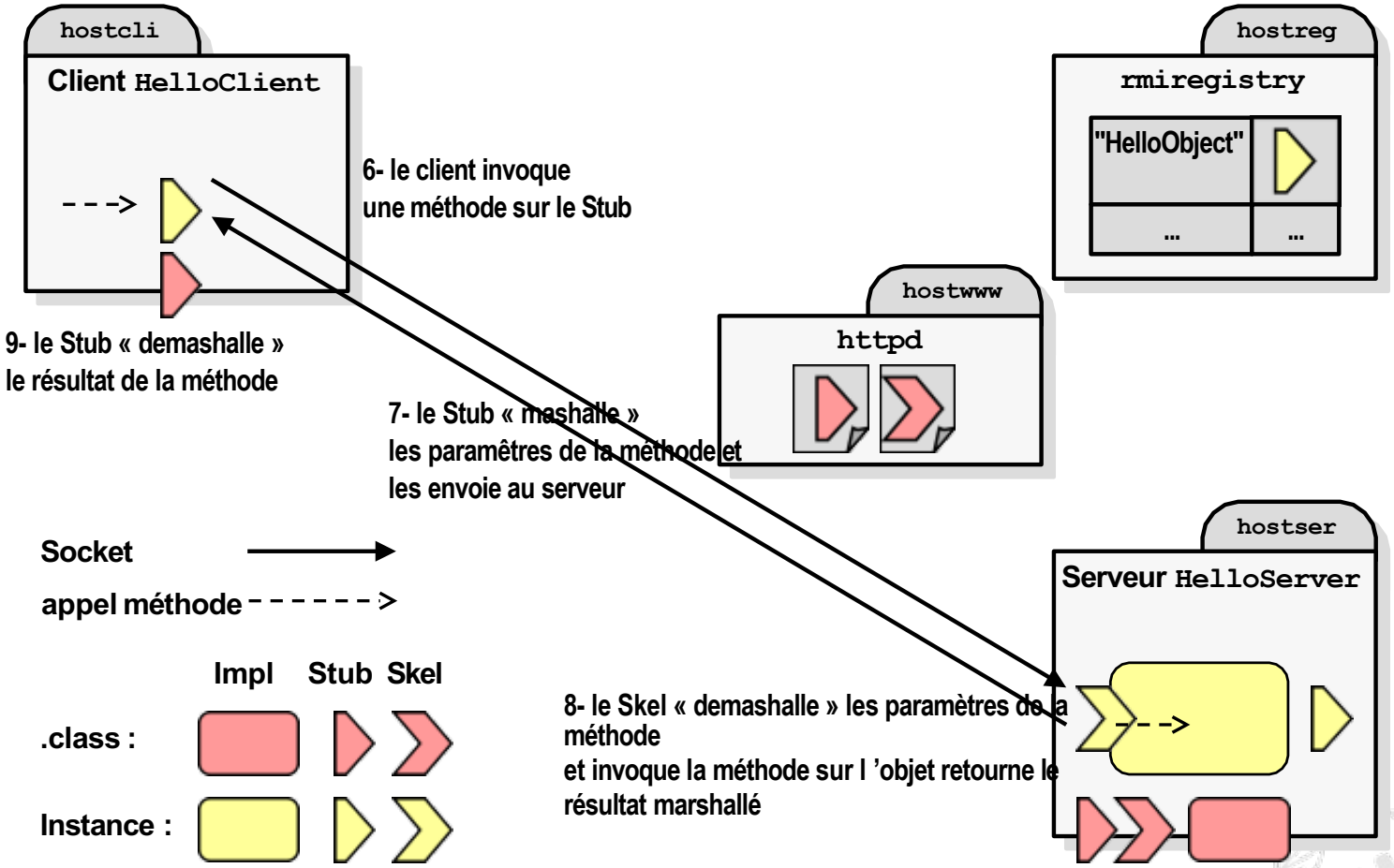
RMI: l'enregistrement de l'objet



RMI: la récupération du Stub



RMI: invocation d'une méthode

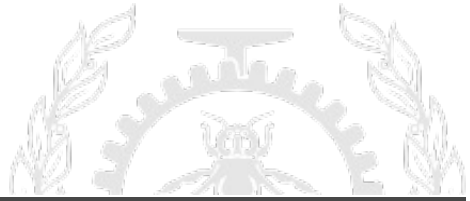


Création et manipulation d'objets distants

- 5 Packages
  - java.rmi : pour accéder à des objets distants
  - java.rmi.server : pour créer des objets distants
  - java.rmi.registry : lié à la localisation et au nommage d'objets distants
  - java.rmi.dgc : ramasse-miettes pour les objets distants
  - java.rmi.activation : support pour l'activation d'objets distants
- Etapes du développement
  - Spécifier et écrire l'interface de l'objet distant.
  - Ecrire l'implémentation de cette interface.
  - Générer les Stub/Skeleton correspondants.
  - Ecrire le serveur qui instancie l'objet implémentant l'interface, exporte son Stub puis attend les requêtes via le Skeleton.
  - Ecrire le client qui réclame l'objet distant, importe le Stub et invoque une méthode de l'objet distant via le Stub.

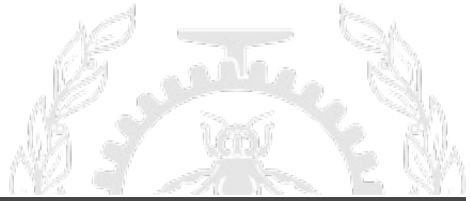
Exemple RMI: interfaces réseau

```
// Interface r´eseau Forme
package examples.RMIShape;
import java.rmi.*;
import java.util.Vector;

public interface Shape extends Remote {
  int getVersion() throws RemoteException;
  GraphicalObject getAllState() throws RemoteException;
}
// Interface r´eseau dessin (liste de Forme)
package examples.RMIShape;
import java.rmi.*;
import java.util.Vector;

public interface ShapeList extends Remote {
  Shape newShape(GraphicalObject g) throws RemoteException;
  Vector allShapes() throws RemoteException;
  int getVersion() throws RemoteException;
}
```



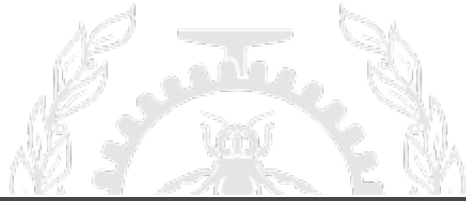
Exemple RMI: serveur de dessin

```
package examples.RMIShape;
import java.rmi.*;

public class ShapeListServer {
  public static void main(String args[]){
    System.setSecurityManager(new RMISecurityManager());
    System.out.println("Main OK");
    try{
      ShapeList aShapelist = new ShapeListServant();
      System.out.println("After create");
      Naming.rebind("ShapeList", aShapelist);
      System.out.println("ShapeList server ready");
    }
    catch(Exception e) {
      System.out.println("ShapeList server main " + e.getMessage());
    }
  }
}
```



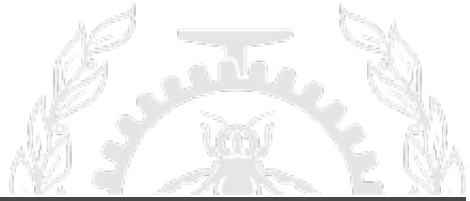
Exemple RMI: dessin exporté par le serveur

```
package examples.RMIShape;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;

public class ShapeListServant extends UnicastRemoteObject
    implements ShapeList{
  private Vector theList;
  private int version;

  public ShapeListServant()throws RemoteException{
    theList = new Vector();
    version = 0;
  }
```



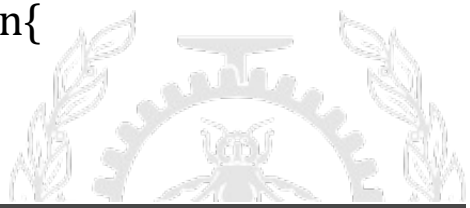
Exemple RMI: dessin exporté par le serveur (Suite)

```
    public Shape newShape(GraphicalObject g) throws RemoteException{
      version++;
      Shape s = new ShapeServant( g, version);
      theList.addElement(s);
      return s;
    }

    public  Vector allShapes()throws RemoteException{
      return theList;
    }

    public int getVersion() throws RemoteException{
      return version;
    }
}
```



Exemple RMI: forme exportée par le serveur

```
package examples.RMIShape;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class ShapeServant extends
    UnicastRemoteObject implements Shape {
  int myVersion;
  GraphicalObject theG;

  public ShapeServant(GraphicalObject g, int version)throws RemoteException{
    theG = g;
    myVersion = version;
  }

  public int getVersion() throws RemoteException {
    return myVersion;
  }

  public GraphicalObject  getAllState() throws RemoteException{
    return theG;
  }
}
```

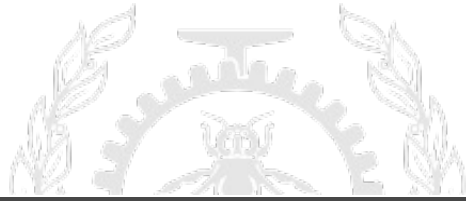


Exemple RMI: client

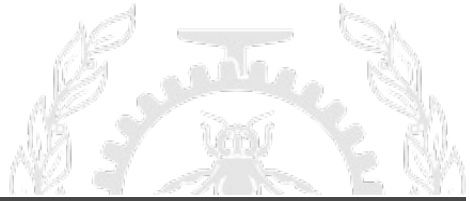
```
// Client
package examples.RMIShape;
import java.rmi.*; import java.rmi.server.*;
import java.util.Vector;
import java.awt.Rectangle; import java.awt.Color;

public class ShapeListClient{
  public static void main(String args[]){
    String option = "Read";
    String shapeType = "Rectangle";
    if(args.length > 0)  option = args[0]; // read or write
    if(args.length > 1)  shapeType = args[1];
    System.out.println("option = " + option + "shape = " + shapeType);

    if(System.getSecurityManager() == null){
      System.setSecurityManager(new   RMISecurityManager());
    }
    else System.out.println("Already has a security manager");
    ShapeList aShapeList = null;
    try{
      aShapeList  = (ShapeList) Naming.lookup("//test.shapes.net/ShapeList");
      System.out.println("Found server");
```

Exemple RMI: client (Suite)

```
        Vector sList = aShapeList.allShapes();
        System.out.println("Got vector");

        if(option.equals("Read")){
          for(int i=0; i<sList.size(); i++){
            GraphicalObject g = ((Shape)sList.elementAt(i)).getAllState();
            g.print();
          }
        } else {
          GraphicalObject g = new GraphicalObject(shapeType,
              new Rectangle(50,50,300,400),Color.red, Color.blue, false);
          System.out.println("Created graphical object");
          aShapeList.newShape(g);
          System.out.println("Stored shape");
        }
      }
      catch(RemoteException e) {
        System.out.println("allShapes: " + e.getMessage());}
      catch(Exception e) {
        System.out.println("Lookup: " + e.getMessage());}
    }
  }
```



Communication par objets répartis



Java Enterprise Edition (EE) étend Standard Edition (SE)

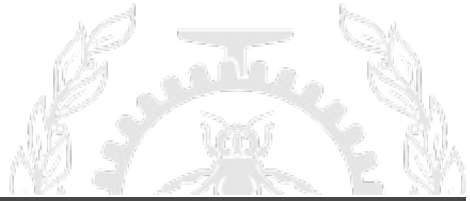
- J2EE 1.2 (1999), 1.3 (2001), 1.4 (2003), Java EE 5 (2006), 6 (2009), 7 (2013), 8 (2017).
- JavaServer Pages (JSP): interfaces pour HTTP.
- Unified Expression Language (EL): langage de script pour les expressions.
- JavaServer Faces (JSF): interface usager.
- Java API for RESTful Web Services (JAX-RS): support pour REST.
- Enterprise JavaBeans (EJB): support de composantes pour les business objects.
- Java Transaction API (JTA): support pour les transactions réparties.
- Java Persistence API (JPA): stockage de l'état dans une base de donnée.
- Bean Validation: annotations de contraintes...

Enterprise Java Beans

- Programmer la logique de l'application séparément du reste de l'environnement: grappe pour le déploiement, interface usager, base de donnée, RPC, sécurité...
- Suppose une architecture classique à trois tiers (interface usager, logique de l'application, base de donnée).
- EJB 1.0 (1998): architecture de base.
- EJB 1.1 (1999): fichiers de méta-données en XML décrivant l'environnement, composantes (beans) de session et d'entité. Interface d'accès à distance.
- EJB 2.0 (2001): interface par message.
- EJB 2.1 (2003): Minuterie et support Web Service.
- EJB 3.0 (2006): POJO avec annotations remplace les méta-données XML.
- EJB 3.1 (2009): Quelques simplifications à l'architecture.
- EJB 3.2 (2013): Changements mineurs.

Conteneurs EJB

- Logiciels comme JBoss (Red Hat), WebSphere (IBM), NetWeaver (SAP), WebLogic (Oracle), Geronimo (Apache), GlassFish (Sun).
- Reçoit les requêtes d'objets/clients locaux ou distants (RMI, RMI-IIOP, Web Services, JMS) qui fournissent l'interface usager.
- Les requêtes sont validées et dirigées vers les objets de session qui sont référencés ou créés au besoin (Business logic).
- Les objets entités sont accédés par les objets de session et leur état est géré et mis à jour dans la base de données selon ce qui a été spécifié dans les annotations (Persistence).

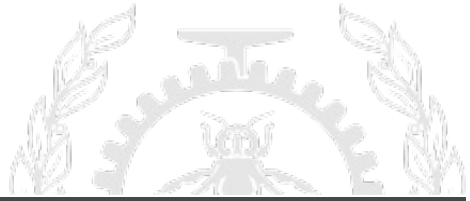


Les rôles selon EJB

- Bean provider: fournisseur des composantes de l'application.
- Application assembler: concepteur de l'application qui assemble les composantes pour obtenir les fonctions désirées.
- Deployer: responsable du déploiement de l'application dans un environnement adéquat.
- Service provider: spécialiste des systèmes répartis qui s'assure du niveau de service désiré.
- Persistence provider: spécialiste des bases de données.
- Container provider: spécialiste de l'environnement d'exécution des composantes Java.
- System administrator: administrateur du système informatique qui s'assure que le système fonctionne selon ce qui a été conçu.

Communication Ex. 5.1

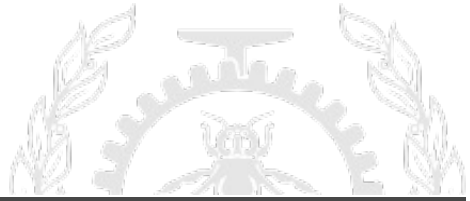
Un appel de procédure à distance (RPC) doit contenir les arguments suivants: string client, string produit, int quantité, int prix. Si les valeurs sont: "Jean Tremblay", "Bicyclette pliante", "2", "80000", combien d'octets seront requis pour encoder cette information avec CORBA CDR (32 bits)?



Communication - Solution - Ex. 5.1

Un appel de procédure à distance (RPC) doit contenir les arguments suivants: string client, string produit, int quantité, int prix. Si les valeurs sont: "Jean Tremblay", "Bicyclette pliante", "2", "80000", combien d'octets seront requis pour encoder cette information avec CORBA CDR (32 bits)?

- Avec CORBA CDR, nous avons 4 octets pour la longueur de client et 16 octets (multiple de 4) pour les 13 lettres de "Jean Tremblay", 4 octets pour la longueur de produit et 20 octets (multiple de 4) pour les 18 lettres de "Bicyclette pliante". Il faut aussi 4 octets pour la quantité et 4 octets pour le prix, soit un total de 4+16+4+20+4+4 = 52 octets.



Communication - Ex. 5.2

Un processus serveur reçoit des requêtes de clients par le biais d'appels de méthode à distance. Le serveur reçoit 25 requêtes par seconde et chaque requête crée un nouvel objet réseau de type *session* qui sera utilisé pendant 350 secondes. On envisage deux stratégies possibles pour déterminer quand les objets réseau peuvent être libérés.

- Pour la première stratégie, une notification est envoyée par le client lorsque l'objet n'est plus utilisé. Cependant, on estime que pour 1% des requêtes, le message de notification ne parviendra pas au serveur et ainsi l'objet ne sera pas libéré et restera en mémoire dans le serveur. Pour cette raison, le serveur est redémarré au milieu de chaque nuit afin de repartir à 0 et que les objets ne s'accumulent pas d'un jour à l'autre.
- Pour la seconde stratégie, l'objet est créé pour une durée de *bail* de 500 secondes, durée qui peut être prolongée au besoin en demandant une extension de *bail* de 500 secondes à la fois.

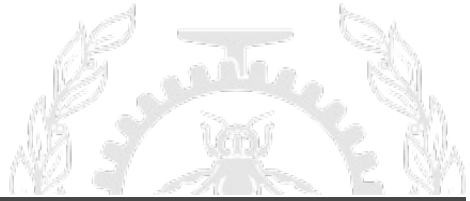
Quel est le nombre d'objets réseau de *session* qui se retrouvent simultanément en mémoire dans le serveur dans le pire cas pour la première stratégie? Pour la seconde?

Communication - Ex. 5.2

Quel est le nombre d'objets réseau de *session* qui se retrouvent simultanément en mémoire dans le serveur dans le pire cas pour la première stratégie? Pour la seconde?

Avec la première stratégie:

- Nous avons 1% des requêtes qui créeront un objet qui ne sera pas libéré avant la fin de la journée.
- Ceci crée une accumulation de 0.01 x 25requêtes/s x 60 s/m x 60m/h x 24 h/jour = 21600requêtes/jour, soit 21600 objets orphelins à la fin de la journée avant le redémarrage.
- En plus, il y a les objets actifs. Lorsqu'une première requête arrive, elle ne sortira qu'après 350s, le nombre de requêtes présentes simultanément (entrées avant que la première ne sorte) sera donc de 25requêtes/s x 350s = 8750requêtes, soit autant d'objets réseau.
- Le total avant de redémarrer est donc de 21600+8750 = 30350.

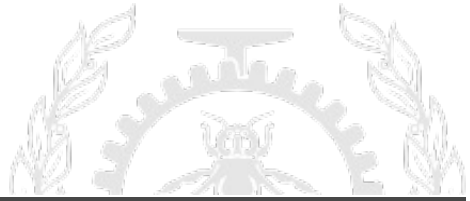


Communication - Ex. 5.2

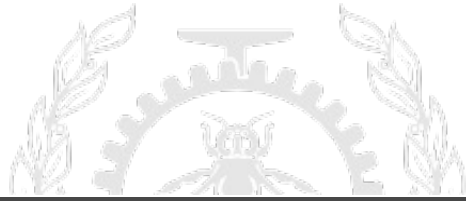
Quel est le nombre d'objets réseau de *session* qui se retrouvent simultanément en mémoire dans le serveur dans le pire cas pour la première stratégie? Pour la seconde?

Avec la seconde solution:

- Nous aurons 25requêtes/s x 500s = 12500requêtes, soit 12500 objets.
- Il y a un peu plus d'objets (que ceux actifs dans la première stratégie) avec la seconde stratégie, car ils sont conservés pour 500s, alors qu'ils auraient pu être libérés au bout de 350s.
- Par contre, on sauve au niveau des messages de notification qui ne sont plus requis, et surtout en évitant les fuites causées par les notifications manquantes. En on n'a pas besoin de redémarrer le système.



Communication par objets répartis



Conclusion

- Java RMI est simple d'utilisation et est utilisé dans des systèmes homogènes Java.
- Le Remoting est simple d'utilisation et fonctionne avec plusieurs langages (C#, C++, VB). Il remplace très avantageusement DCOM.