Partie Socket :

Exemple tuto de Socket Client :

```
public class GreetingClient {

   public static void main(String [] args) {
      String serverName = args[0];
      int port = Integer.parseInt(args[1]);
      try {
         System.out.println("Connecting to " + serverName + " on port " +
port);
         Socket client = new Socket(serverName, port);

         System.out.println("Just connected to " +
client.getRemoteSocketAddress());
         OutputStream outToServer = client.getOutputStream();
         DataOutputStream out = new DataOutputStream(outToServer);

         out.writeUTF("Hello from " + client.getLocalSocketAddress());
         InputStream inFromServer = client.getInputStream();
         DataInputStream in = new DataInputStream(inFromServer);

         System.out.println("Server says " + in.readUTF());
         client.close();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Exemple tuto de Socket Server :

```
public class GreetingServer extends Thread {
   private ServerSocket serverSocket;

   public GreetingServer(int port) throws IOException {
      serverSocket = new ServerSocket(port);
      serverSocket.setSoTimeout(10000);
   }

   public void run() {
      while(true) {
         try {
            System.out.println("Waiting for client on port " +
               serverSocket.getLocalPort() + "...");
            Socket server = serverSocket.accept();

            System.out.println("Just connected to " +
server.getRemoteSocketAddress());
            DataInputStream in = new DataInputStream(server.getInputStream());

            System.out.println(in.readUTF());
            DataOutputStream out = new
DataOutputStream(server.getOutputStream());
            out.writeUTF("Thank you for connecting to " +
server.getLocalSocketAddress()
               + "\nGoodbye!");
            server.close();

         } catch (SocketTimeoutException s) {
            System.out.println("Socket timed out!");
            break;
         } catch (IOException e) {
            e.printStackTrace();
            break;
         }
      }
   }

   public static void main(String [] args) {
      int port = Integer.parseInt(args[0]);
      try {
         Thread t = new GreetingServer(port);
         t.start();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

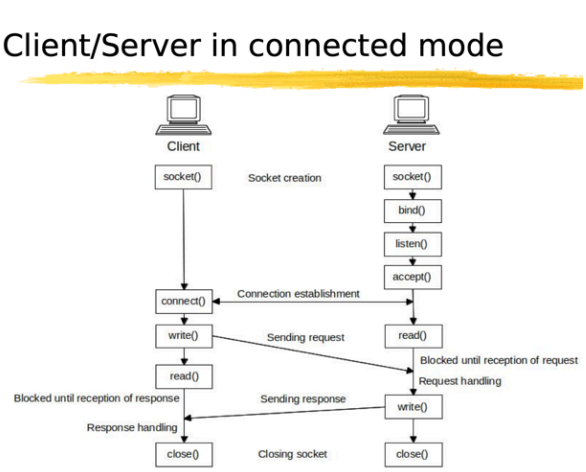
Partie Cours Sockets :

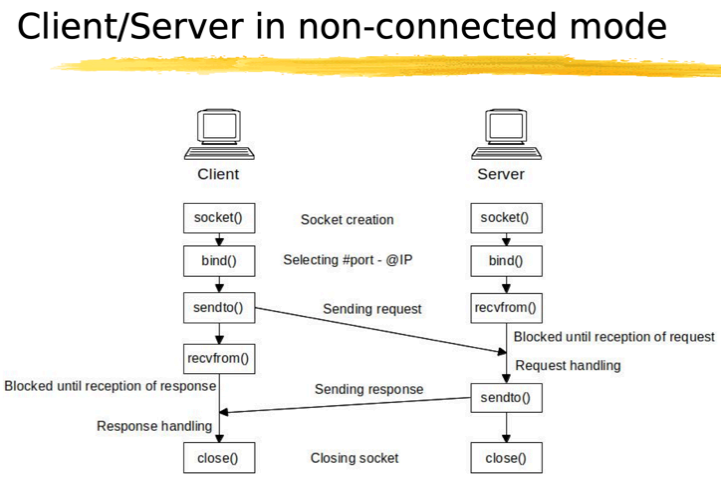
Socket : API de programmation de réseau de communication, grâce à laquelle un développeur exploite facilement les services d'un protocole réseau. Ils permettent entre autres de construire des applications client/serveur.
Les sockets possèdent une @IP, un #port et un protocole (comme TCP ou UDP par exemple).

2 modes :

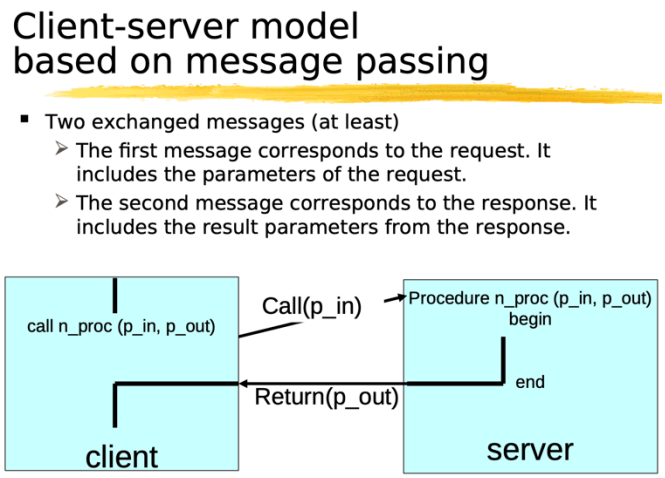
- Mode connecté (TCP) : 1 seule primitive pour l'envoi et la réception et pas de taille maximale de message (utilisation de Stream of Bytes), pas de problèmes de connexion.
- Mode non connecté (UDP) : Plus efficace, permet broadcast/multicast, mais les problèmes de connexion doivent être pris en charge par l'application.

L'API Socket :
Ouvrir un dialogue :
Client : bind(…) ou connect(…)
Serveur : bind(…) ou listen(…) ou accept(…)
Transfert de data :
Mode connecté : read(…), write(…), send(…), recv(…)
Mode non connecté: sendto(…), recvfrom(…), sendmsg(…), recvmsg(…)
Fermer un dialogue :
Close(…), shutdown(…)



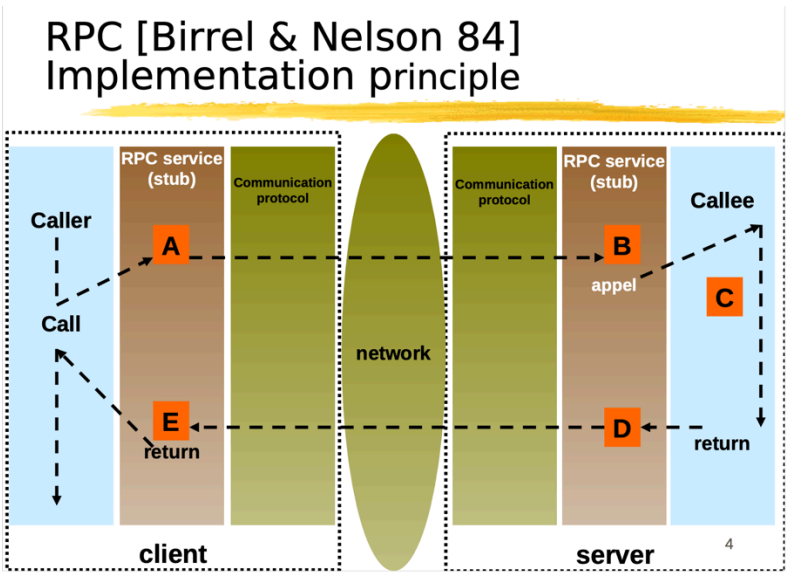


Partie Cours Client-Serveur RMI :



Modèle avec au moins 2 messages : 1 pour la requête, puis suspension de l'activité du client, puis la réponse du serveur.

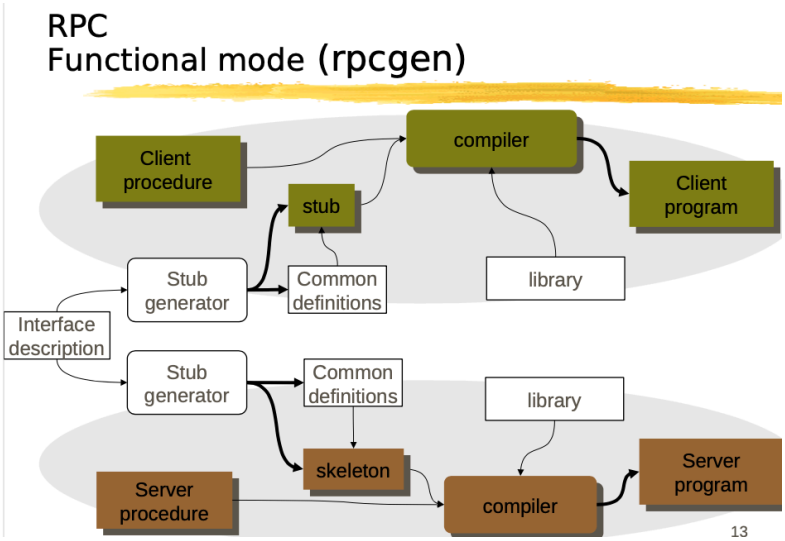
Remote Procedure Call (RPC) : Génère la plupart du code (ex : émission/réception messages) pour faciliter le dév. d'applications possédant des interactions Client/Serveur.



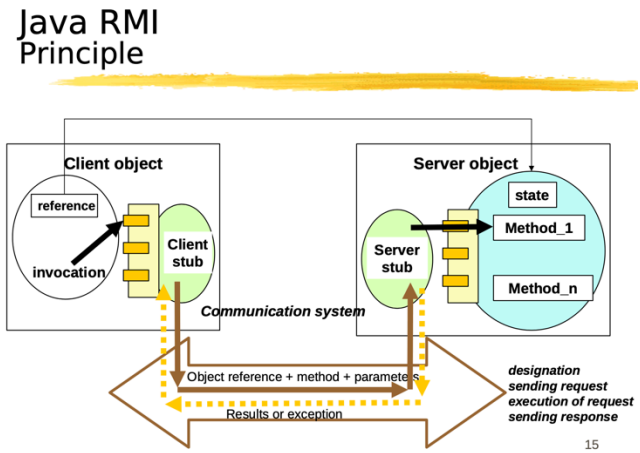
Possible perte de message :
Côté client : si le watchdog (s'assure que l'ordi ne reste pas bloqué) expire ou si on reçoit un message avec un RPC Identifier connu.
Côté serveur : si le watchdog expire on reçoit un message avec un RPC Identifier connu.

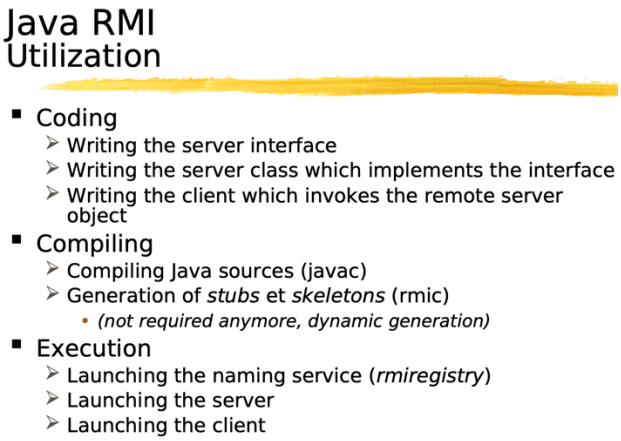
Problèmes du RPC : Performance, Sécurité, Gestion des pannes

Schéma du RPC ci-après



Java Remote Method Invocation (RMI) : Objet base sur RPC intégré en java et facile à utiliser car invoqué comme s'il était en local.





Utilisation :

Serveur :

```
public class HelloImpl extends UnicastRemoteObject implements Hello {
        String message;
        Int port;
        String URL = "//localhost:"+port

        public HelloImpl(String msg, int p) throws RemoteException {
                port = p;
                message = msg;
        }

        //Implementation de la méthode Remote :
        public void sayHello() throws RemoteException {
        System.out.println(message);
        }

        public  static void main(String args[]) {
        try {
                Registry registry = new LocaleRegistry.createRegistry(port);
                // Créer une instance de l'objet Serveur
                Hello obj = new HelloImpl("hello");
                Naming.rebind(URL, obj);
```

```
            } catch() {…}
        }
}
```

Client :

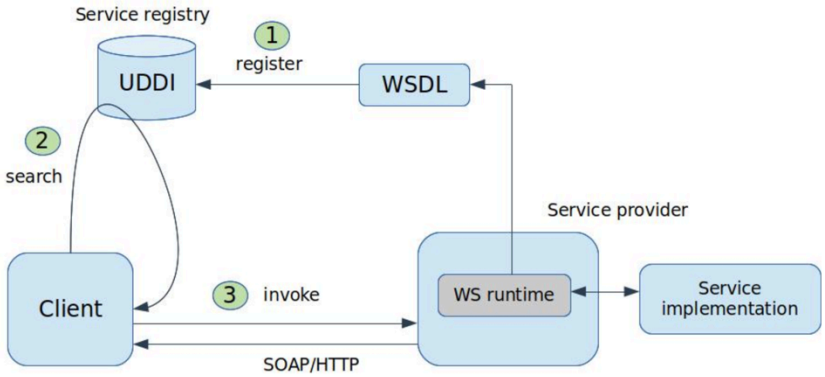
```
Public class HelloClient {
        Int port;
        String URL = “//localhost:”+port

        Public static void main(String args[]) {
            Try{
                Hello obj = (Hello) Naming.lookup(URL);
                Obj.sayHello();
            } catch …..
        }
}
```

Partie Cours Web Services :

Motivations : Faciliter l'intégration de services web en leur fournissant une interaction (et intégration) RPC
Contraintes : Applications développées de manière indep.
Conséquences : Pas de définition de modèle, choix de base XML (pour son adaptabilité).

Architecture of WS



Service Web REST :

Version simplifiée, non standard, qui utilise des méthode http (comme GET, POST, PUT, DELETE), invoque un service dans l'URL, y passe des paramètres, se déploie à travers plein d'environnements.

Service Web SOAP :

Étaient très populaires mais sont maintenant devenus obsolètes depuis les REST.

Partie Cours JMS :

Modèle basé sur les messages :

Modèle Client/Serveur :

Appels synchrones, approprié pour des éléments fortement couplés, connexion 1-1, cible définie

Modèle message :

Communication asynchrone, connexion 1-N, cible indéfinie (on envoie à des groupes : newsgroups).

Coarse-grain :

Applications souvent qualifiées selon le niveau de synchronisation de leurs tâches. Si une application a des tâches qui se synchronisent beaucoup entre elles, on la qualifie de fined-grained (grain fin), sinon de coarse-grained (gros grain i.e. de parallélisme embarrassant).

Middleware basé sur les messages (MOM : Message oriented middlewares):

Communication avec messages :

Environnement classique : socket.

Queue de messages :

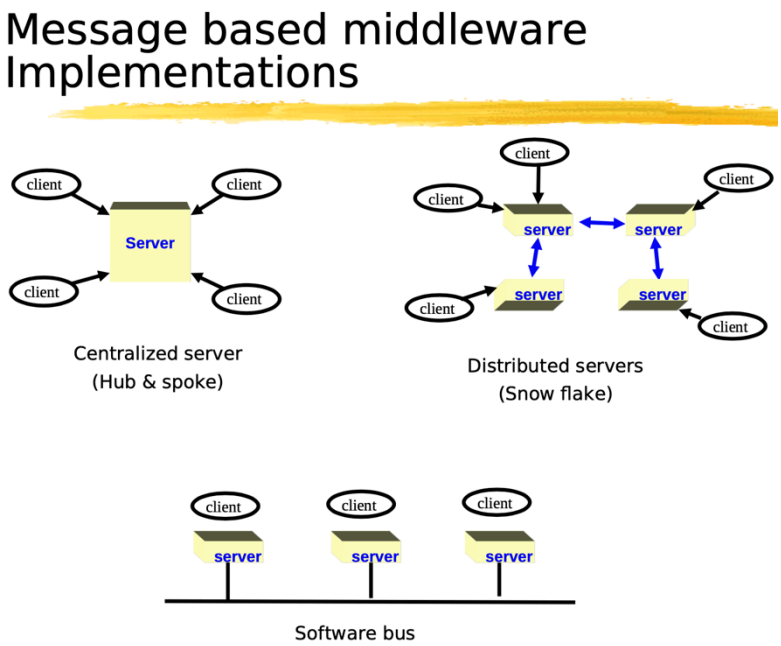
Persistence des messages (reliabilité), indépendance entre émetteur et récepteur (permet de l'asynchrone), permet plusieurs récepteurs (anonymise).

Publication/souscription à des newsgroups :

Communication 1-N, le récepteur souscrit à un topic, le rédacteur envoie un message sur un topic.

Events (communication avec callbacks) :

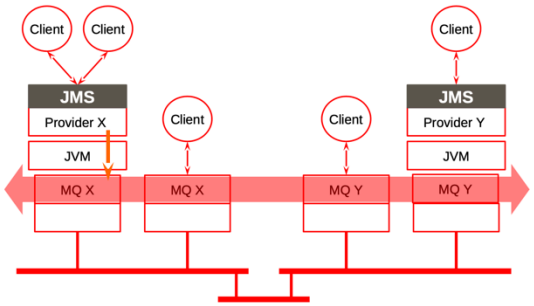
Permet d'associer un événement et une action de manière simple.



Java Messagerie Service (JMS) :

API définissant des interfaces uniformes pour l'accès à des systèmes de messagerie (ex : Oracle WebLogic, Apache ActiveMQ). Utilise les MOMs (c.f. ci-dessus).

JMS: an interface (portability, not Interoperability)



Interoperability : AMQP (Advanced Message Queuing Protocol)

JMS interface

- *ConnectionFactory*: factory to create a connection with a JMS server
- *Connection*: an active connection with a JMS server
- *Destination*: a location (source or destination)
- *Session*: a single-thread context for emitting or receiving
- *MessageProducer*: an object for emitting in a session
- *MessageConsummer*: an object for receiving in a session

- Implementations of these interface are specific to providers ...

Partie Cours ESB :

Entreprise Application Integration (EAI) :

Multi-Socket : Un connecteur par application, l'EAI route les messages entre les applications.
C'est une sorte de hub qui permet l'interconnexion de plusieurs applications.
Les EAI sont *généralement* centralisées (mais pas les ESB c.f. ci-dessous).

Entreprise Service Bus (ESB):

Objectif : fournir un support pour l'intégration de différentes applications dans une infrastructure globale.
Typiquement, dans l'administration de l'N7 on a pleins de logiciels spécialisés qui ne communiquent a priori par du tout ensemble, les ESB sont là pour ça.
Ex d'ESB : IBM WS ESB, Mule, JBoss ESB, Apache ServiceMix
Les ESB sont des EAI décentralisées qui relient les grands standards (XML, WS, JMS …).
Une ESB est faite d'un bus (MOM souvent implémenté en JMS), de data (souvent en XML), d'adaptateurs/connecteurs (des WS), d'un flot de contrôle (routage).

Exemple de Mule :

ESB basée sur Java qui permet aux devs de connecter facilement et rapidement des applications pour échanger de la data suivant la méthodologie Service-Oriented Architecture (SOA), sans regarder les différentes technologies que les applications utilisent comme JMS, WS, HTTP …
Mule permet entre autres de faire du routing de messages, de la reliabilité, de la sécurité, de la convversion de protocole ou même de la transformation de messages.

ENSEEIHT 2ième année Département Sciences du Numérique. Parcours ABLMR. Année 2021-2022

Intergiciels

Examen session 1

Daniel Hagimont

NOM : .....

Durée: 1h30, documents autorisés

*Lire l'ensemble des énoncés avant de commencer à répondre. La **clarté**, la **précision** et la **concision** des réponses, ainsi que leur **présentation matérielle**, seront des éléments importants d'appréciation. Donnez essentiellement les programmes Java demandés. Dans vos programmes, vous n'avez pas à programmer les imports et le traitement des exceptions.*

PROBLÈME (Sockets et RMI) (16 points)

Nous voulons implanter en Java un service de diffusion de message à un groupe d'applications, les applications pouvant s'insérer dynamiquement dans le groupe. Ce service permet à des applications réparties de s'insérer dans le groupe (un seul groupe est géré), les messages envoyés au groupe étant reçus par toutes les applications insérées dans le groupe. La procédure d'insertion dans le groupe est implantée en utilisant Java-RMI et la diffusion de message dans le groupe est implantée avec des Sockets.

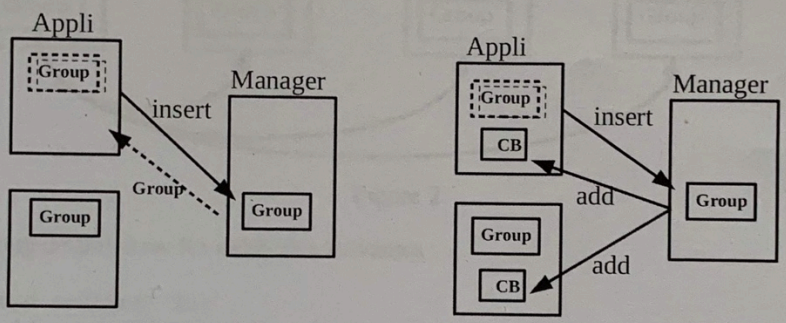


Figure 1

La procédure d'insertion dans le groupe est structurée comme suit :

1) Un objet réparti Manager d'interface/classe Manager/ManagerImpl permet à une application de s'insérer dans le groupe. La méthode d'insertion prend en paramètre (par copie) un objet de la classe Host (ci-dessous) contenant les coordonnées (host/port) de l'application permettant la communication par messages. Le Manager inclut un objet Group d'interface/classe Group/GroupImpl qui inclut la liste des Host présents dans le groupe. L'objet Group est retournée (par copie) en résultat de la méthode d'insertion pour que l'application s'insérant récupère l'état courant du groupe (Figure 1 – gauche).

2) La méthode d'insertion prend un paramètre de type CallBack (par référence) lui permettant d'être avertie de l'arrivée d'une nouvelle application dans le groupe. Le Manager gère donc une liste des Callbacks permettant d'avertir les membres du groupe de l'arrivée d'un nouveau membre (Figure 1 – droite).

La classe Host est donnée :

```
public class Host implements Serializable {
    public Host(String h, int p) { host=h; port=p; }
    public String host;
    public int port;
}
```

Les interfaces ci-dessus définissent donc les méthodes suivantes :

Manager

public Group insert (Host h, CallBack cb);

CallBack

```
public void addNewMember (Host h);
```

La diffusion des messages fonctionne comme suit :

1) Au début, chaque application démarre un thread chargé de recevoir les messages (en acceptant des connexion TCP sur hostname + port qui sont des paramètres (args) de l'application). Ce thread se contente d'afficher les messages reçus sur le terminal.
2) Ensuite, comme décrit ci-dessus, l'application s'insère dans le groupe en appelant le Manager. Elle récupère ainsi l'état courant du groupe. Lors de l'insertion, elle passe en paramètre un objet CallBack pour être avertie de l'arrivée d'un nouveau membre.
3) Enfin les applications peuvent communiquer en utilisant le groupe (Group) contenant les coordonnées de toutes les applications. Pour ce faire, Group fournit une méthode send() permettant d'envoyer un message (une String). Chaque envoi de message est implanté avec une nouvelle connexion TCP et l'envoi du message (Figure 2). Vous pouvez utiliser la sérialisation pour envoyer/recevoir un message.

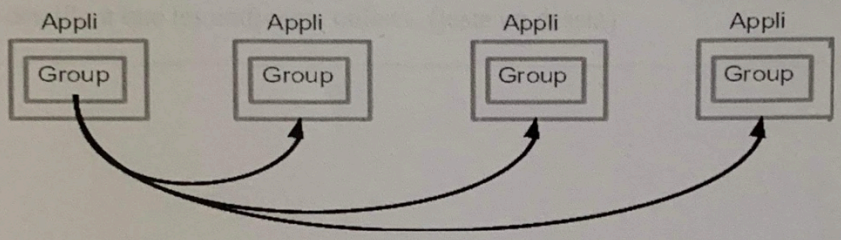


Figure 2

L'interface Group définit donc les méthodes suivantes :

Group

```
public void add(Host h);
public void send(String message);
```

Notez que la méthode add() est appelée dans le Manager et dans le CallBack, alors que la méthode send() est appelée par l'application uniquement.
On ne traitera pas les problèmes liés à la synchronisation.

Vous devez donner l'implantation
- des interfaces : Group, Manager, CallBack
- des classes : GroupImpl, ManagerImpl, CallBackImpl
- une classe Appli qui montre comment on utilise le service.

QUESTIONS DE COURS (4 points)

1) Expliquez la différence entre Web Services SOAP et Web Services REST. Vous devez expliquer dans chaque cas ce qui est **fourni par le serveur exportant** un WS et ce qui est **utilisé par le client utilisant** un WS (2 lignes)

WS SOAP :

WS REST :

2) Supposons que nous disposons de 3 applications : une application A permettant l'exportation de données dans un fichier, une application B permettant l'importation de données dans une base de données, et une application C permettant l'importation de données avec un Web Service. Nous voulons que les données exportées par A soient importées par B et C. Dessinez l'architecture de principe d'interconnexion avec l'ESB Mule en ne détaillant que les endpoints utilisés. (juste un dessin)

Proposition de solution de problème 2021 (de Maxime Henry):

Serializable : On envoie une copie de l'objet et les modifications s'effectuent en local.

Remote : On envoie une référence vers l'objet et les modifications s'effectuent à travers tous les clients.

```
public interface Manager extends Remote {
    public Group insert(Host h, Callback cb) throws RemoteException;
}


public interface Group extends Serializable {
    void add(Host h);
    void send(String msg);
}


public interface Callback extends Remote {
    public void addNewMember(Host h) throws RemoteException;
}



public class GroupImpl implements Group {
    private ArrayList<Host> hosts;

    public GroupImpl() {
        this.hosts = new ArrayList<>();
    }

    public void add(Host h) {
        this.hosts.add(h);
    }

    public void send(String msg) {
        for (Host h : hosts) {
            Socket s =  new Socket(h.host, h.port);
            OutputStream os = s.getOutputStream();
            os.write(msg);
        }
    }
}


public class CallbackImpl extends UnicastRemoteObject implements Callback {
    private Group group;

    private Callback(Group g) throws RemoteException {
        this.group = g;
    }

    public void addNewMember(Host h) throws RemoteException {
        this.group.add(g);
    }
}



public class ManagerImpl extends UnicastRemoteObject implements Manager {
    private Group group;
    private ArrayList<Callback> cbs;
    public final static URL = "//localhost:4000/Manager";

    public ManagerImpl() throws RemoteException {
        this.group = new GroupImpl();
        this.cbs = new ArrayList<>();
    }

    public Group insert(Host h, Callback cb) throws RemoteException {
        for (Callback cb2 : cbs) {
```

```
            cb2.addNewMember(h);
        }
        cbs.add(cb);
        this.group.add(h);
        return this.group;
    }

    public static void main(String args[]) {
        Registry rmi = LocateRegistry.createRegistry(4000);
        Manager m = new ManagerImpl();
        Naming.bind(this.URL, m);
    }
}



public class Application {
    public static void main(String args[]) {
        new Thread() {
            @Override
            public void run()  {
                ServerSocket serv = new ServerSocket(Integer.parseInt(args[1]));
                while(true) {
                    Socket client = serv.accept();
                    InputStream is = client.getInputStream();
                    byte[] t = new byte[1024];
                    is.read(t);
                }
            }
        }.start();

        Manager m = (Manager) Naming.lookup(ManagerImpl.URL);
        Group group = new Group();
        group = m.insert(new Host(args[0], args[1]), new CallbackImpl(group));
        group.send("bonjour");
    }
}
```