**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(…)

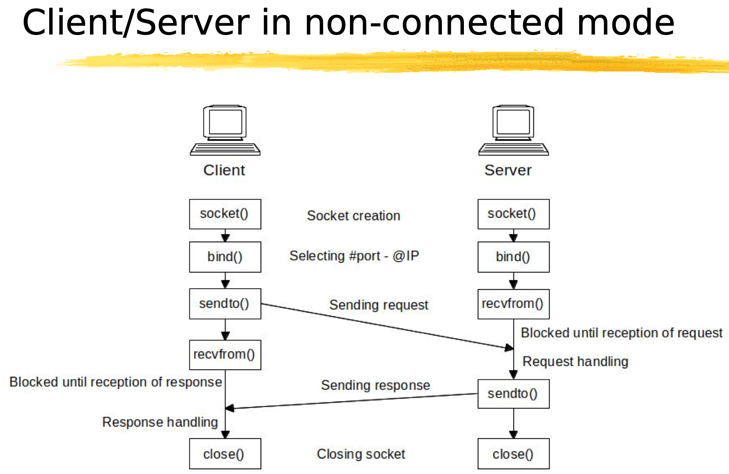
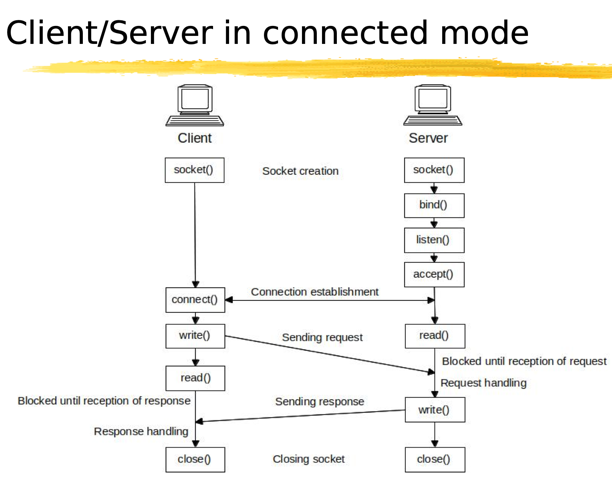
Transfert de data :

Mode connecté : read(…), write(…), send(…), recv(…)

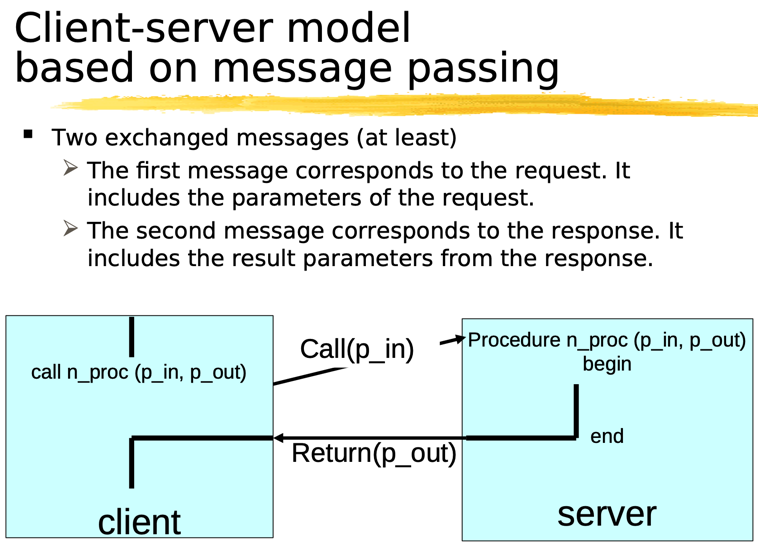
Mode non connecté: sendto(…), recvfrom(…), sendmsg(…), recvmsg(…)

Fermer un dialogue :

Close(…), shutdown(…)

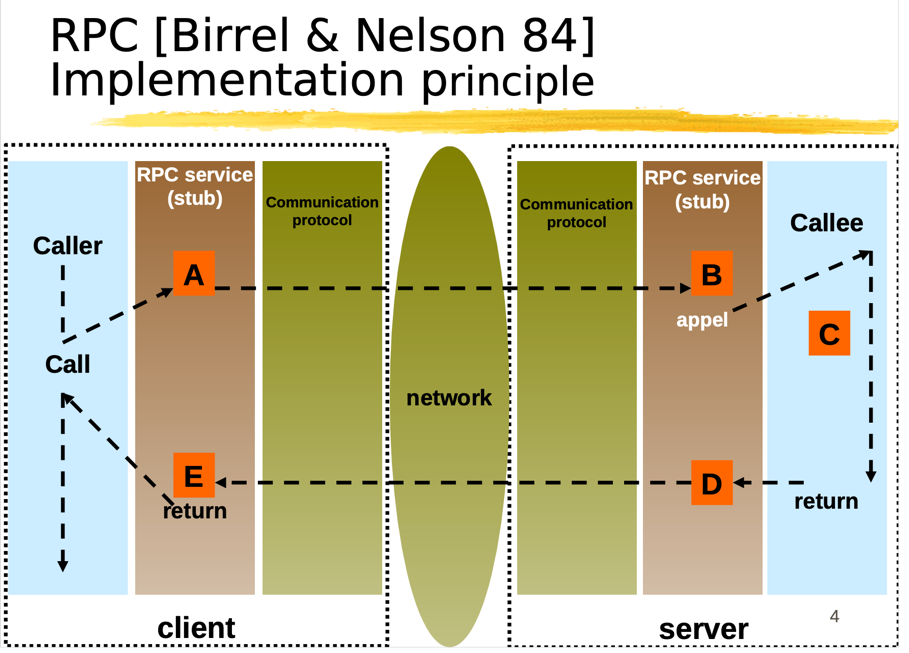


**Partie Cours Client-Serveur RMI :**



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.



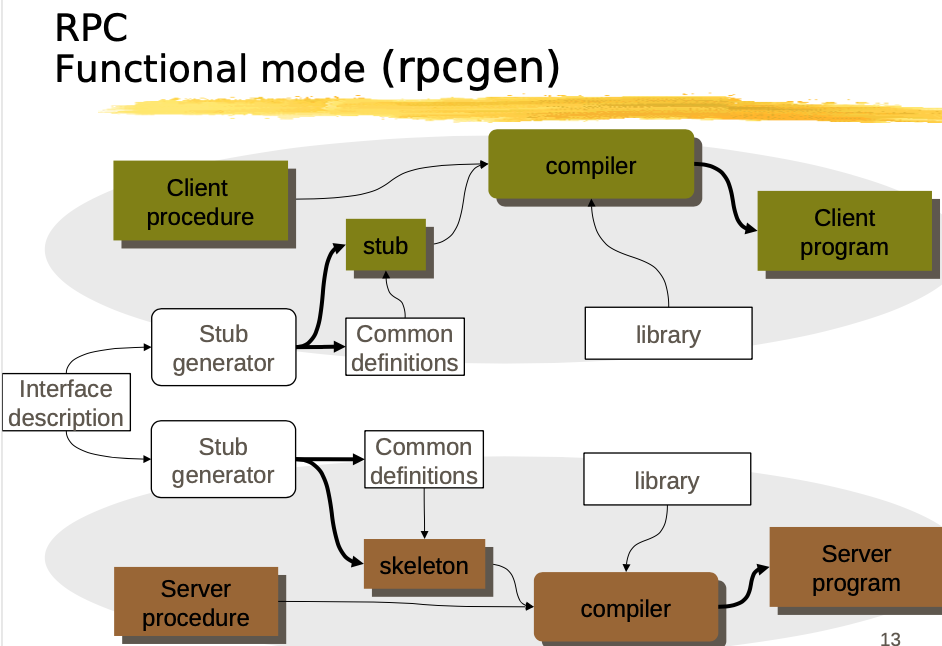
Possible perte de message :

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.

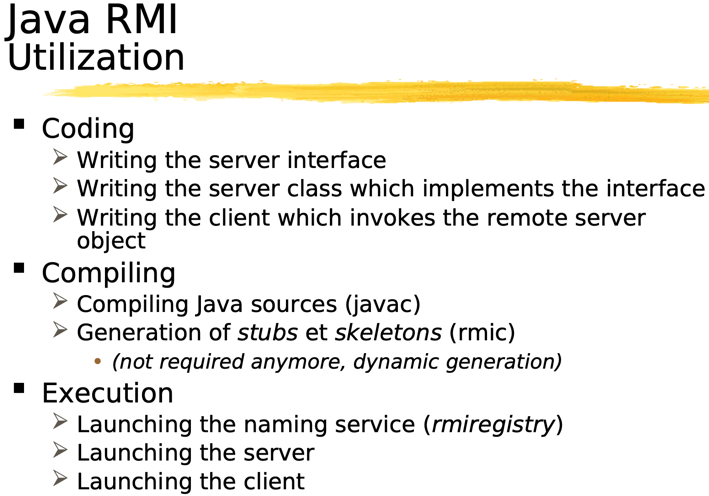
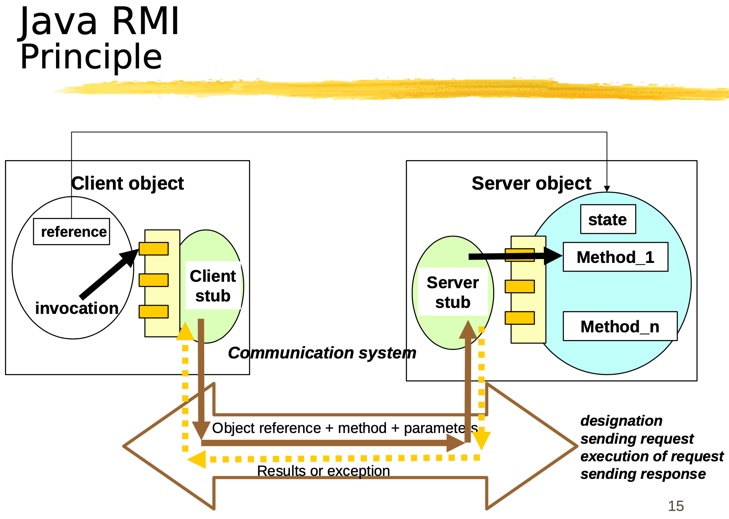
Côté serveur : si le watchdog expire on reçoit un message avec un RPC Identifier connu.

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 …..

}

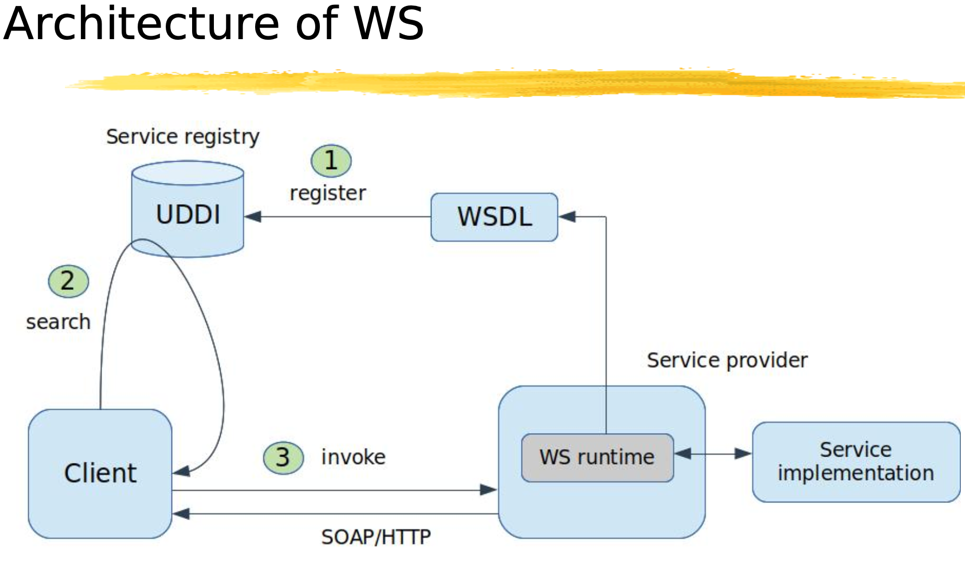
}

**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).



**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 :

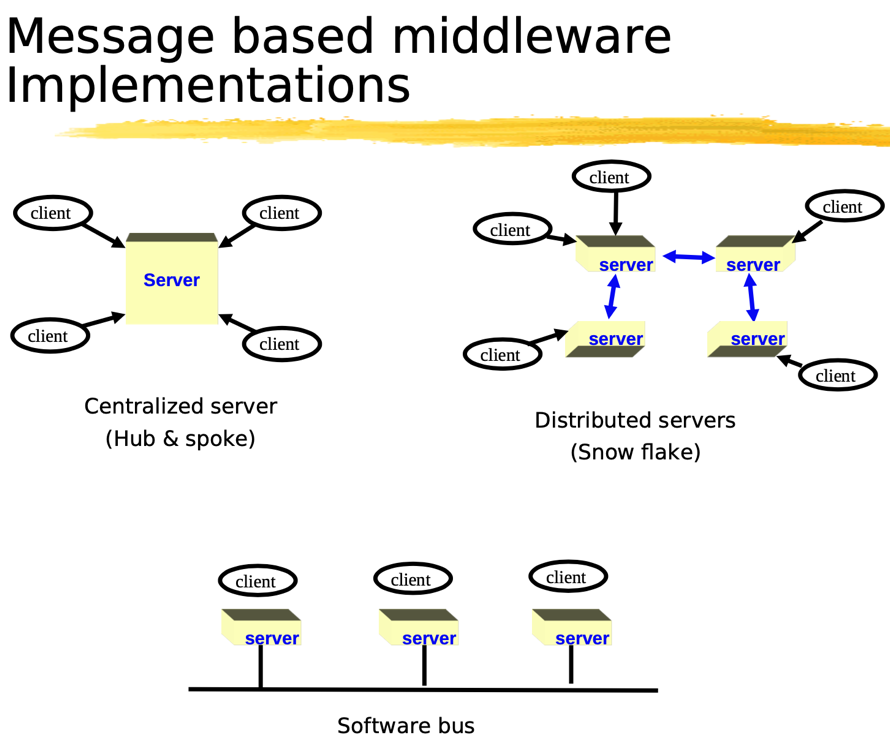
Persistence des messages (reliabilité), indépendance entre émetteur et récepteur (permet de l’asynchrone), permet plusieurs récepteurs (anonymise).

Publication/souscription à des newsgroups :

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

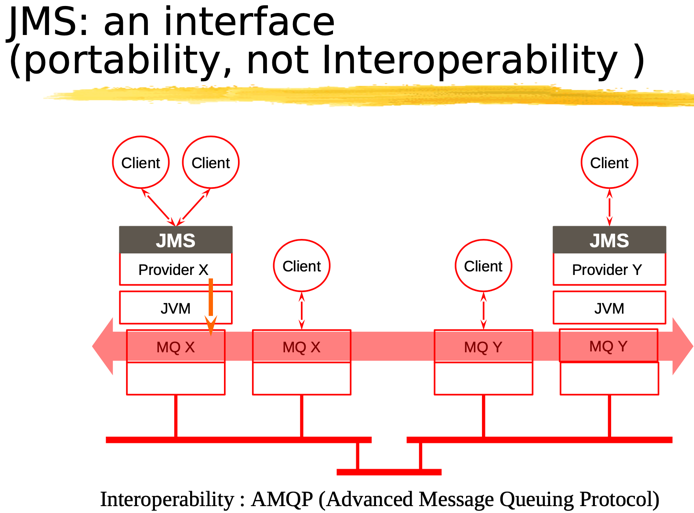
Events (communication avec callbacks) :

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).

Une image contenant texte

Description générée automatiquement

**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.

Une image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquement

**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");

}

}