Chapitre 3 : Programmation réseau

3.1 Introduction et rappels sur les sockets

Une Socket est une interface de programmation (API) avec les services du système d'exploitation pour exploiter les services de communication du système (local ou réseau). Une socket est un demi-point de connexion d'une application. Une socket est caractérisée par une adresse.

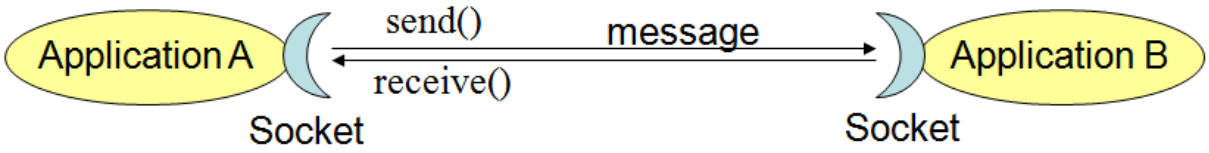


Figure 1.5 :Schéma de socket

Couche 7	Applicative	Logiciels	NFS
Couche 6	Présentation	Représentation indépendante des données	XDR
Couche 5	Session	Établit et maintient des sessions	RPC
Couche 4	Transport	Liaison entre applications de bout en bout, fragmentation, éventuellement vérification	TCP, UDP, Multicast
Couche 3	Réseau	Adressage et routage entre machines	IP
Couche 2	Liaison	Encodage pour l'envoi, détection d'erreurs, synchronisation	Ethernet
Couche 1	Physique	Le support de transmission lui-même	

Figure 3.1 :Les couches réseau

Les Sockets en Java sont uniquement orientées transport (couche 4). Deux API pour les sockets :

- java.net : API bloquante (étudié ici)
- java.nio.channels (> 1.4) :

Mode connecté : la communication entre un client et un serveur est précédée d'une connexion et suivi d'une fermeture.

- Facilite la gestion d'état
- Meilleurs contrôle des arrivées/départs de clients
- Uniquement communication unicast
- Plus lent au démarrage

Mode non connecté : les messages sont envoyés librement

- Plus facile à mettre en œuvre
- Plus rapide au démarrage

- Il ne faut pas confondre connexion au niveau transport et au niveau applicatif!
- HTTP est un protocole non connecté alors que TCP l'ai
- FTP est un protocole connecté et TCP aussi
- RPC est un protocole non connecté et UDP non plus

Liaison par flux : Socket/ServerSocket (TCP)

- Utilise le mode connecté : protocole de prise de connexion (lent)
- Sans perte : un message arrive au moins un fois
- Sans duplication : un message arrive au plus une fois
- Avec fragmentation : les messages sont coupés
- Ordre respecté
- Communication de type téléphone

Liaison par **datagram** : DatagramSocket/DatagramPacket (UDP)

- Non connecté : pas de protocole de connexion (plus rapide)
- Avec perte : l'émetteur n'est pas assuré de la délivrance
- Avec duplication : un message peut arriver plus d'une fois
- Sans fragmentation : les messages envoyés ne sont jamais coupés
  ⇒ soit un message arrive entièrement, soit il n'arrive pas
- Ordre non respecté
- Communication de type courrier

Une socket est identifiée par

- Une adresse IP : une des adresses de la machine (ou toutes)
- Un port : attribué automatiquement ou choisi par le programme

`static InetAddress InetAddress.getByAddress(byte ip[])`	`construit un objet d'adresse ip`
`static InetAddress InetAddress.getByName(String name)`	`renvoie l'Adresse IP de name`
`static InetAddress InetAddress.getLocalHost()`	`renvoie notre adresse`
`String InetAddress.getHostName()`	`renvoie le nom symbolique`
`byte[] InetAddress.getHostAddr()`	`renvoie l'adresse IP`

```
public class Main {
  public static void main(String args[]) {
    byte ip[] = {195, 83, 118, 1 };
    InetAddress addr0 = InetAddress.getByAddress(ip);
    InetAddress addr1 = InetAddress.getByName("ftp.lip6.fr");
    …
  }
}
```

En Java, il est possible de représenter dans un objet une adresse de Socket sans protocole attaché. Le tableau suivant résumé les différentes instructions.

InetSocketAddress(InetAddress addr, int port);	Construit une adresse de Socket
InetSocketAddress(String name, int port);	Construit une adresse de Socket
InetAddress InetSocketAddress.getAddress()	Renvoie l'adresse IP
int InetSocketAddress.getPort();	Renvoie le port
String InetSocketAddress.getHostName();	Renvoie le nom symbolique de l'IP

```
public class Main {
  public static void main(String args[]) {
    byte ip[] = {195, 83, 118, 1 };
    InetAddress addr = InetAddress.getByAddress(ip);
    InetSocketAddress saddr0 = new InetSocketAddress(addr, 21);
    InetSocketAddress saddr1 = new InetSocketAddress("ftp.lip6.fr", 21);
    … } }
```

Listing 3.1 : exemple d'utilisation de InetAddress

Pool de connexions

Afin de permettre à plusieurs clients de se connecter à un serveur unique, il est possible de mutualiser les connexions. Pour ce faire, un pool de connexions est ouvert en permanence et les clients (ré) utilisent les connexions ouvertes.

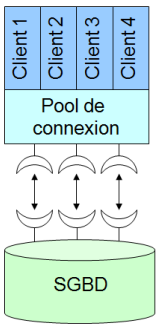


Figure 3.2 : pool de connexions à un SGBD

3.2 Sockets en mode flux

Les sockets en mode flux de Java reposent sur le protocole TCP. Ses propriétés sont :

- Taille des messages quelconques
- Envoi en général bufferisé (i.e. à un envoi OS correspond plusieurs écritures Java)
- Pas de perte de messages, pas de duplication
- Les messages arrivent dans l'ordre d'émission
- Contrôle de flux (i.e. bloque l'émetteur si le récepteur est trop lent)
- Pas de reprise sur panne Trop de perte ou réseau saturé

Les sockets en mode flux nécessitent une phase de connexion. Le Serveur : attend des connexions et le client : se connecte au serveur. Le serveur doit maintenir des connexions avec plusieurs clients.

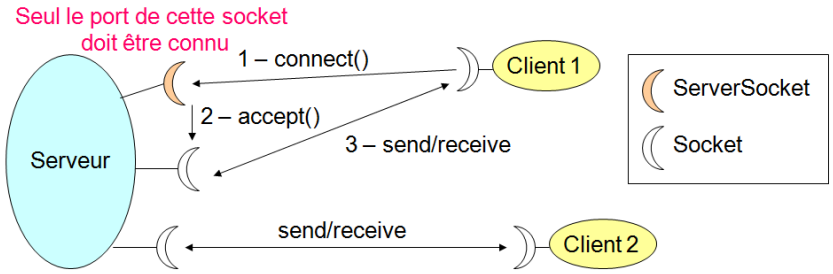


Figure 3.3 : Exemple de Socket avec flux

La socket du serveur possède le port `#port` qui identifie le serveur. La socket de communication du serveur possède un port attribuée automatiquement par Java lors de `l'accept()`. La socket de communication du client possède un port attribuée automatiquement par Java lors du `connect()`. Il est possible de fixer le port de la socket du client par :

```
sc.bind(new SocketAddress(InetAddress.getLocalHost(), #port)).
```

Comme il est possible aussi de fixer le port de la socket de connexion du serveur après sa création par

```
ServerSocket sa = new ServerSocket();

sa.bind(new SocketAddress(InetAddress.getLocalHost(), #port));
```

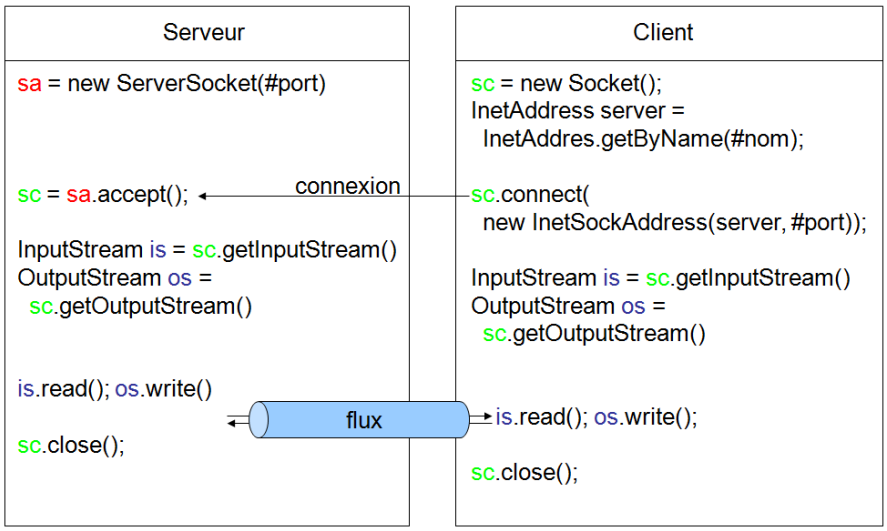
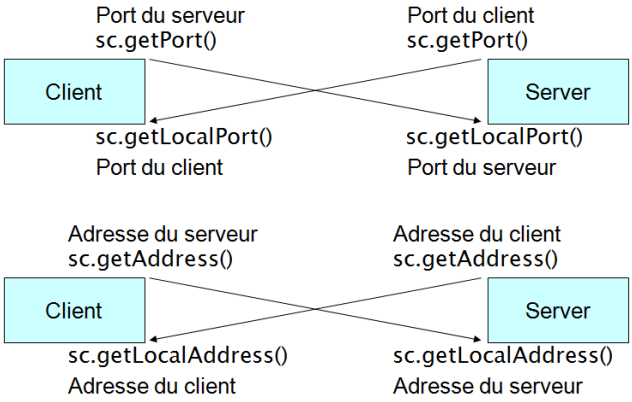


Figure 3.4 : Exemple de programmation de Socket avec flux

Pour retrouver les adresses IP et les ports, les instructions suivantes peuvent être utilisées:

- `getPort()` permet de récupérer le port distant de la socket
- `getLocalPort()` permet de récupérer le port local d la socket
- `getAddress()` permet de récupérer l'adresse de la machine distante
- `getLocalcAdress()` permet de récupérer l'adresse de la machine locale



Remarque : les flux associés aux sockets peuvent être encapsulés dans n'importe quels autres flux

```
ObjectInputStream is = new ObjectInputStream(new
                           GZipInputStream(sc.getInputStream()));

ObjectOutputStream os = new ObjectOutputStream(
             new GZipOutputStream(sc.getOuputStream()));
```

3.3 Sockets en mode datagram

Les Sockets en mode datagram de Java reposent sur le protocole UDP. De nombreuses protocoles utilisent UDP, comme DNS, TFTP, RIP, ainsi que le Multicast. Ses propriétés sont:

- Taille des messages fixe et limitée (64ko)
- envoi non bufferisé
- Possibilité de perte de messages, Duplication
- Les messages n'arrivent pas forcement dans l'ordre d'émission
- Aucun contrôle de flux
- Pas de détection de panne (même pas assuré que les messages arrivent)
- **Faible latence** (car aucun contrôle de flux, pas de connexion)

DatagramSocket : Socket orientée Datagram et est liée à un port. Elle communique uniquement via des DatagramPacket, pas d'utilisation de flux.

DatagramPacket : représente un message. Si la taille du buffer de réception est trop petite, la fin du message est perdue.

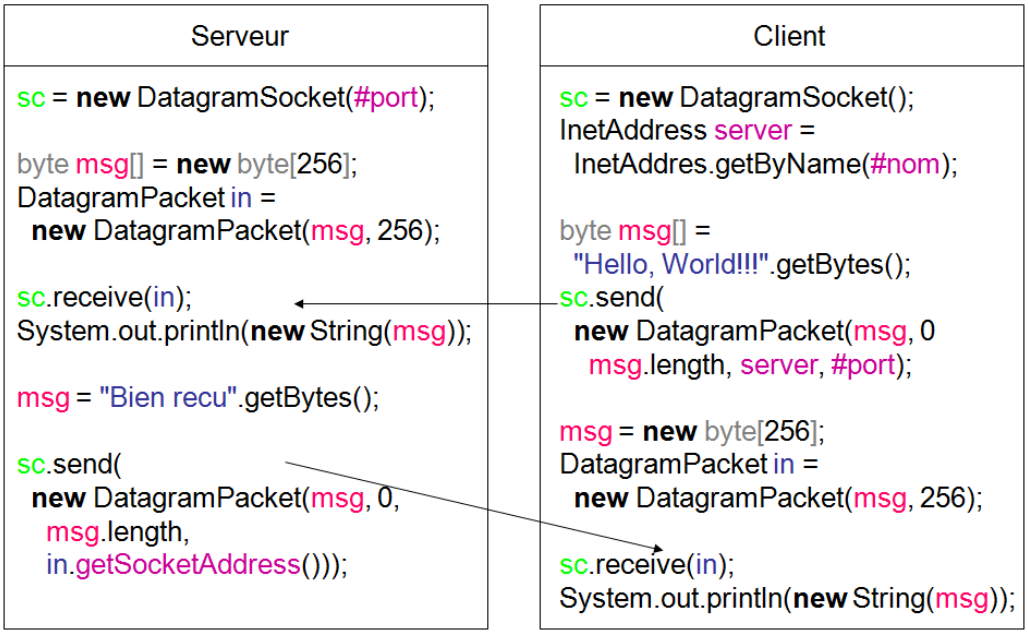


Figure 3.5 : exemple de programmation des sockets datagram

Le client a besoin de connaître l'adresse IP et le port du serveur. Le socket du client se voit assigner un port lors de l'envoi. Une DatagramSocket ne possède pas de flux, elle envoie uniquement de tableaux de bytes, et reçoit uniquement de tableaux de bytes.

```
DatagramPacket serialize(Object o) {
  ByteArrayOutputStream bos = new ByteArrayOutputStream();
  ObjectOutputStream os = new ObjectOutputStream(bos);
  os.writeObject(o);
  byte msg[] = bos.toByteArray();
  return new DatagramPacket(msg, msg.length);
}

Object deserialize(DatagramPacket packet) {
  ObjectInputStream is =
    new ObjectInputStream(
      new ByteArrayInputStream(packet.getData(),
           packet.getOffset(),
           packet.getLength());
  return is.readObject();
}
Object receive(DatagramSocket s) {
       byte buf[] = new byte[????];
       DatagramPacket packet = new DatagramPacket(buf, buf.length);
       s.receive(packet);
       return deserialize(packet);
    }
```

Listing 3.2 : exemple de programmation des sockets datagram

3.4 Sockets en mode multicast

Le multicast consiste à diffuser un message à un groupe de récepteur. Chaque récepteur s'abonne à une adresse IP de classe D, comprise entre 224.0.0.0 et 239.255.255.255. Certaines adresses sont déjà réservées. Comme par exemple la diffusion d'un match de football en direct à un groupe de téléspectateur. Ainsi, l'émetteur : émet à destination de cette adresse IP.

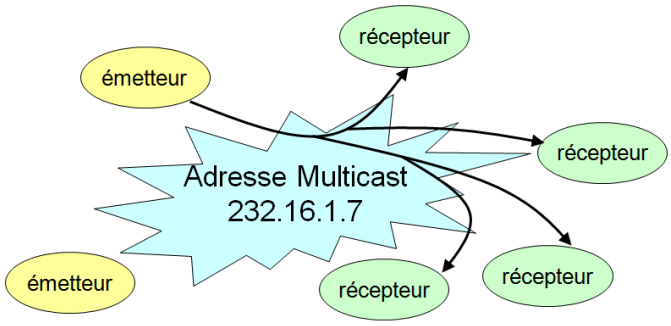


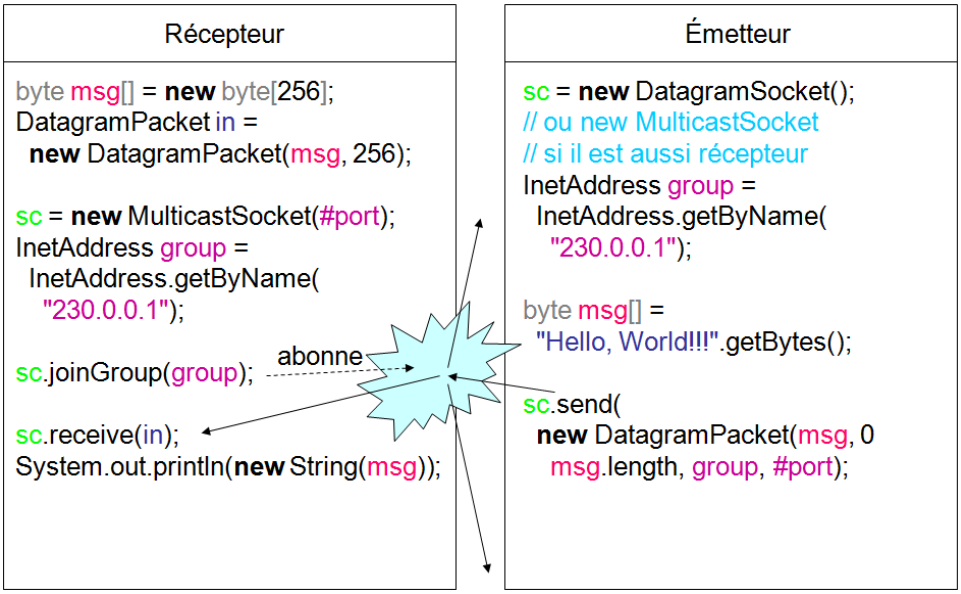
Figure 3.6 communication multicast

Socket en mode multicast de Java repose sur IP Multicast, lui-même basé sur UDP. Il a de ce faite les mêmes propriétés qu'UDP, et qui sont :

- Taille des messages fixe et limitée (64ko),
- envoi non bufferisé,
- Possibilité de perte de messages ou de duplication pour certains récepteurs,
- Les messages n'arrivent pas forcement dans l'ordre d'émission, pas forcement dans le même ordre chez tous les récepteurs,
- Aucun contrôle de flux.

Exemple :

Soit l'exemple d'un émetteur qui Émet sur une adresse de classe D et un port, qui ne nécessite pas d'abonnement. Le récepteur s'abonne à une adresse de classe D, écoute sur un port donnée, peut se désabonne de la classe D avant de quitter `MulticasrSocket.leaveGroup(InetAddress group)`, et peut rejoindre et quitter le groupe multicast à tout instant. Ci-dessous un exemple.



Listing 3.3: exemple de programmation multicast

3.2 Principes des Systèmes à Objets Répartis

3.2.1 Interactions requêtes/réponse

Une interaction requête/réponse, entre un client et un serveur, consiste en l'envoi du client d'une requête. Pour cela, le programme client doit affecter un numéro à la requête et encoder le message. A la réception de la requête, le serveur décode le message, traite la requête en invoquant la méthode associée, puis envoie la réponse au client.

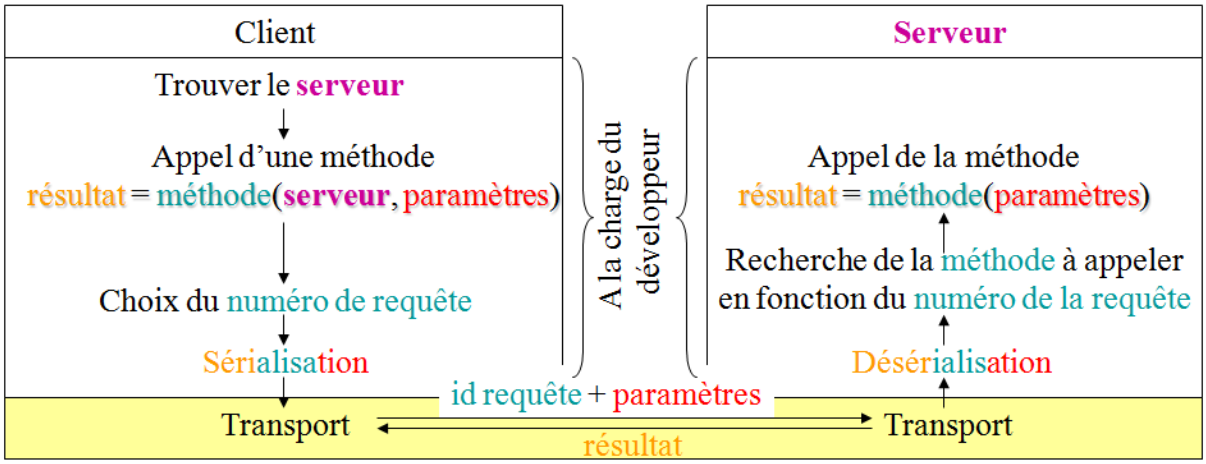


Figure 3.7 Communication par socket

3.2.2 Appel de procédures distantes

Les objectifs d'un protocole d'appel de méthode à distance (par exp: RPC) sont de : *diminuer le développement, masquer la répartition* et de *masquer l'hétérogénéité.*

- *Diminuer le travail de développement* des serveurs et des clients, en prenant en charge la sérialisation/désérialisation des arguments, la conversion appel de méthode en protocole requête/réponse, car il est plus facile pour un serveur d'offrir plusieurs services (méthodes)
- *Masquer une partie de la répartition*, le client et le serveur sont développer comme-ci ils s'exécutent en locale. Ainsi, le client appel une méthode locale, cet appel est délégué au serveur, puis la méthode est exécutée par le serveur.
- *Masquer l'hétérogénéité entre clients et serveurs*, en offrant un format pivot pour les données (xdr, sérialisation Java, CDR…).

Pour ce faire, des objets proxy (mandataires) sont générés automatiquement. Ces objets ont pour but de déléguer les appels distants. Le mandataire généré du côté du client est nommé une souche (stubs en anglais), et celui généré du côté du serveur est nommé un squelette (skeleton).

Ainsi, il reste à la charge du développeur du client de trouver le serveur et d'invoquer ses méthodes, et il reste à la charge du développeur du serveur d'implémenter ses méthodes accessibles à distances.

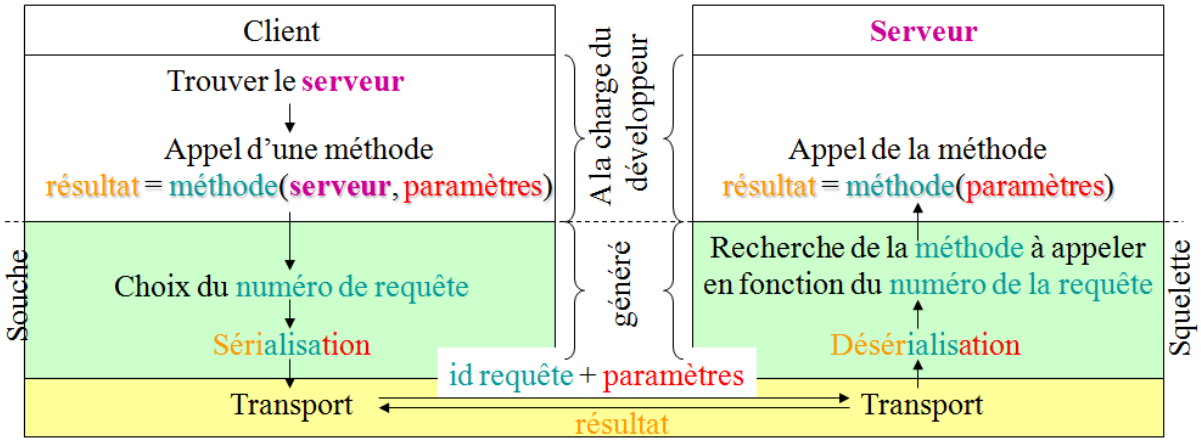


Figure 3.8 : Communication par objets répartis

3.2.3 Modèle des objets répartis

Dans le développement par socket, **l'**appel de méthode n'est pas pris en charge, et il est inadéquat pour un langage objet. Il reste à charge du client de le prendre en considération durant son développement et l'encodage des messages.

Pour rester cohérent avec les principes de l'orienté objets, le paradigme des objets répartis a été défini. Ces objets répartis permettent de définir un code serveur et un code client comme-ci ils s'exécutent localement, et des objets répartis sont générés automatiquement et de manière transparente aux développeurs. Les objets répartis permettent de représenter un objet distant sur une machine et lui déléguer l'appel.

3.2.3.1 Notion de mandataire

Un mandataire (proxy) substitue un représentant à la place d'un objet qui offre les mêmes fonctionnalités (méthodes), c'est un objet réparti. Ce qui permet de répartir le client et le serveur. Son principe est que le client utilise un mandataire à la place de l'objet d'origine du serveur et de manière transparente. Son fonctionnement est le suivant:

- Une classe Serveur S implante une interface I
- Le client utilise uniquement les méthodes de l'interface I
- Le mandataire doit implanter les méthodes de I

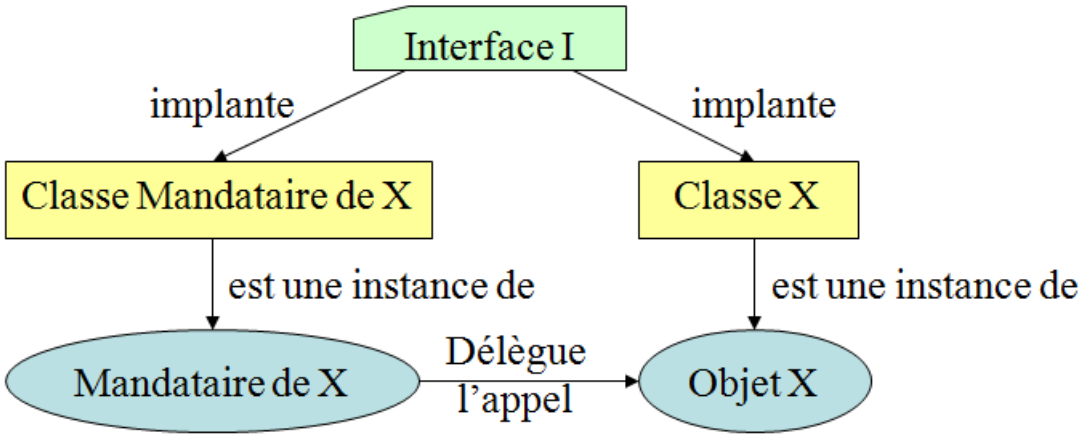


Figure 3.9 : Schéma général de fonctionnement d'un mandataire

3.2.3.2 Paradigme des objets répartis

Paradigme des objets répartis

Souche du client = mandataire spécialisé + mécanisme de construction

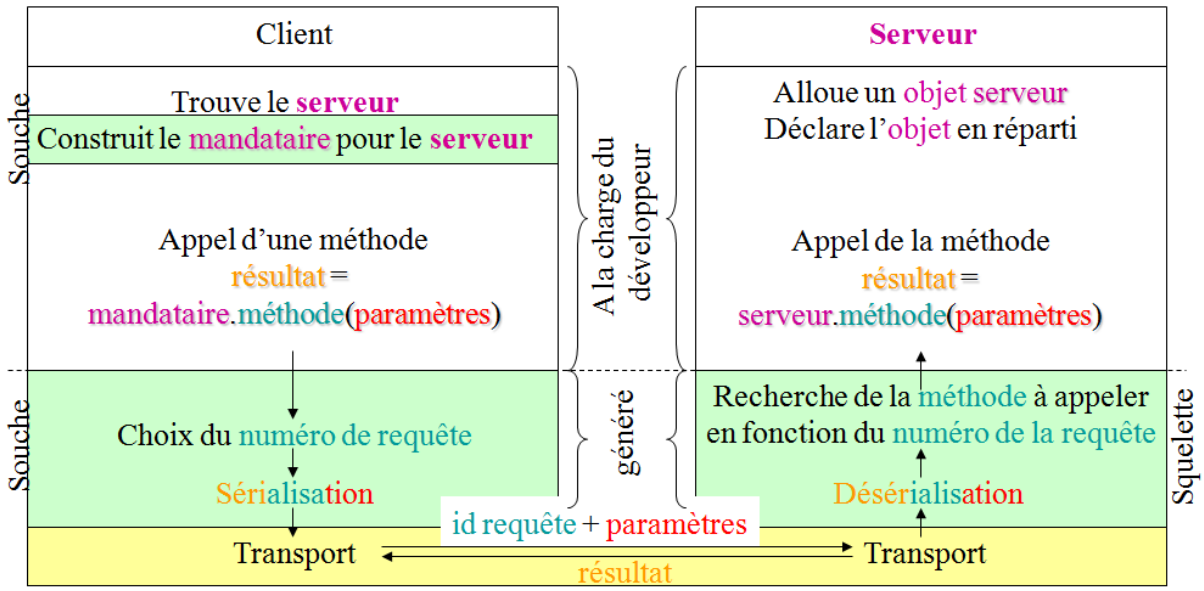


Figure 3.10 : Communication par objets répartis avec recherche du serveur

3.2.4 Construction d'applications

Pour construire une application à objets répartis, il faut suivre les étapes suivantes :

1. définir l'interface de l'objet réparti dans un langage L0, qui décrit les fonctionnalités du serveur
2. Écrire le serveur dans un langage L1
   a. Développer un objet serveur implantant l'interface
   b. Écrire le serveur : allouer un objet serveur + exporter cet objet

3. Écrire le client dans un langage L2

   a. Trouver le serveur

   b. Construire un mandataire à partir du serveur

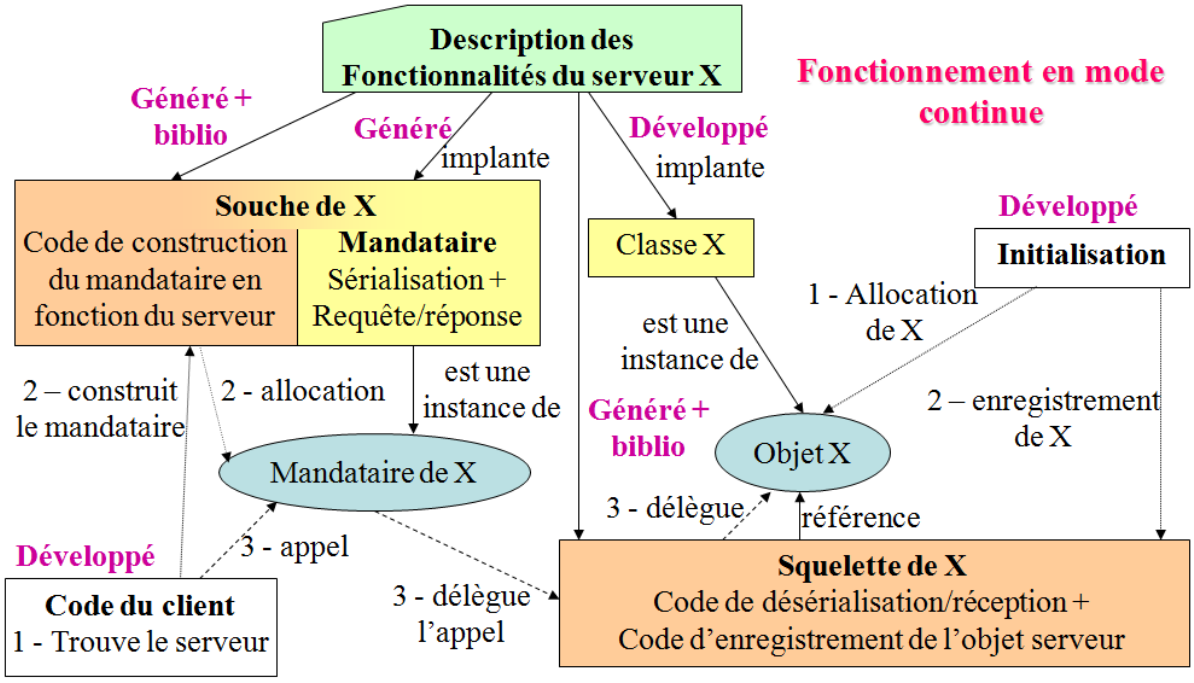


Figure 3.11 : Paradigme des objets répartis

Le modèle à objets répartis n'impose pas que le client et le serveur soient homogènes. Les langages de programmation, les systèmes d'exploitation peuvent être différents, car la couche réseau masque la répartition et la couche de sérialisation masque l'hétérogénéité

3.2.5 Passage de paramètres

Il existe deux façons de passer les paramètres entre clients et serveurs, soit *par copie de paramètres* ou bien *par référence du paramètre*.

Le passage de paramètre par copie, consiste à copier l'objet à distance, lors de l'appel ou lors du retour de l'appel. Ce qui permet d'éviter les accès distants. Ainsi, la copie et l'original n'ont plus de rapport, et si l'original est modifié, la copie de ne l'est pas. **L**es modifications ne sont pas répercutées sur l'original et donc pas forcément mis à jour.

Passage de paramètre par référence consiste à envoyer une référence distante et un mandataire est construit lors de l'appel ou lors du retour. Il permet d'accéder toujours à la dernière version. Un objet passé par référence est un objet réparti. C'est un mandataire qui est envoyé. L'objet n'existe qu'en un seul exemplaire. Les modifications sont partagées par tous les utilisateurs du serveur. Chaque accès à l'objet est effectué à distance.

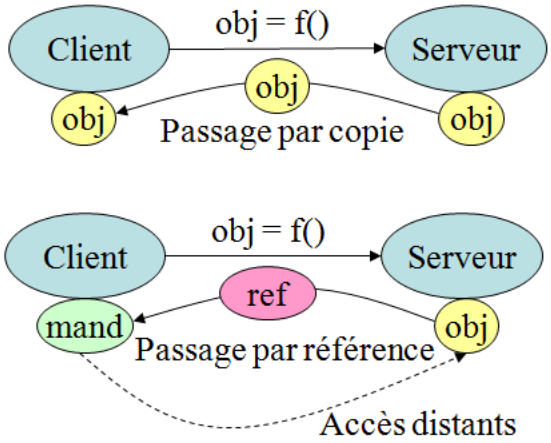


Figure 3.12 Passage de paramètres

3.2.6 Middleware

Un middleware (intergiciel) est une couche logicielle intermédiaire entre les applications et le réseau, qui permet de dialogue entre des applications hétérogènes.

3.2.6.1 Propriétés des middlewares

Les propriétés des middlewares sont

- masquer les problèmes de la répartition, ce qui faciliter la programmation répartie
- masquer hétérogénéité du système sous -jacent (matériel et logiciel), ce qui permet la portabilité des applications
- Offre une interopérabilité d'applications hétérogènes ; c'est-à-dire une sémantique de communication. L'interopérabilité est définie comme la capacité d'un produit ou système à fonctionner avec d'autres produits ou systèmes présents ou futurs et ce sans restriction d'accès ou de mise en œuvre.
- Fournir des services communs à usage général, comme un service de sécurité, de transaction, et de persistance