# *Chapitre 1* Introduction aux applications réparties

**Système centralisé :** tout est localisé sur la même machine et accessible par le programme

- L'exécution s'effectue sur une seule machine
- Accès local aux ressources (données, code, périphériques, mémoire ...)

**Système réparti** : un système informatique dans lequel les ressources ne sont pas centralisées.

- composé de plusieurs systèmes calculatoires autonomes
- sans mémoire physique commune (sinon c'est un système parallèle)
- qui communiquent par l'intermédiaire d'un réseau quelconque

# Pourquoi utiliser un système réparti?

Les systèmes répartis sont populaires pour plusieurs raisons:

- Accès distant: un même service peut être utilisé par plusieurs acteurs, situés à des endroits différents
- Redondance: des systèmes redondants permettent de pallier une faute matérielle, ou de choisir le service équivalent avec le temps de réponse le plus court
- Performance: la mise en commun de plusieurs unités de calcul permet d'effectuer des calculs parallélisables en des temps plus courts
- partage des données: les systèmes ont besoins de communiquer entre eux  des données
- Capacité d'évolution : un systèmes répartis peut être évoluer facilement en ajoutant d'autres service ou d'autres composants
- Minimisation du coût : le rapports prix/performance est amélioré

- Les systèmes répartis sont équipés d’une couche logiciel dédié à la coordination des activités du système ainsi qu’au partage des ressources.
    - Pour remédier au problème de l’hétérogénéité des ressources matériels et logiciels (ordinateur, systèmes d’exploitation, langage de programmation).
    - Pour cacher la complexité du matériel et des communications.
    - Pour fournir des services communs de plus haut niveau d’abstraction.

- Du point de vue de l'utilisateur, le système apparaît comme une unique entité.

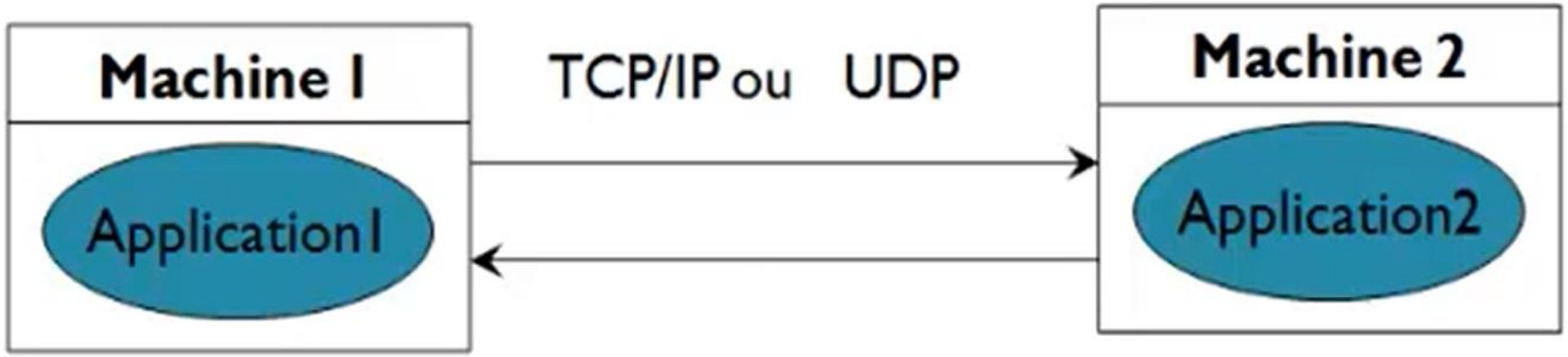
## Applications réparties

- Une application répartie est une application composée de plusieurs entités(ou applications) s'exécutant indépendamment et en parallèle sur un ensemble d'ordinateurs connectés en réseau pour répondre à un problème spécifique en utilisant les services généraux fournis par le système.

  ⇒ **Les applications réparties**

  - ne se résument pas à l'échange de données en réseau.

    Exemple recherche sur le web: un navigateur Internet
  - Mais mettent en œuvre réellement une coopération des programmes entre eux.




## Technologies d'accès :

- Sockets ou DataGram
- RMI (JAVA)
- CORBA (Multi Langages)
- EJB (J2EE)
- Web Services (HTTP+XML)

# Inconvénients de la répartition

- La gestion du système est totalement décentralisée et distribuée

  ⇒ Une mise en œuvre plus délicate

    - Gestion des erreurs
    - Suivi des exécutions

  ⇒ Pas de vision globale instantanée

- Délais des transmissions: si le débit d'information est très important

  ⇒ Goulot potentiel d'étranglement

- Les applications réparties imposent un mode de développement spécifique, et alourdissent le test et le déploiement du logiciel.

- **Problèmes réseaux:**
  - Si problème au niveau du réseau, le système peut tomber en panne.
- **Problème serveur:**
  - En générale un serveur est central au fonctionnement du système. Si le serveur plante, le système ne fonctionne pas.
- **Administration plus lourde:**
  - Installation
  - Configuration
  - Surveillance
- **Coût:**
  - Formation
  - Achat des environnements
- **Sécurité.** Accès aux données confidentielles. Authentification. Utilisation des ressources (périphériques, logiciels licenciés). Malveillance.

# Outils de communication

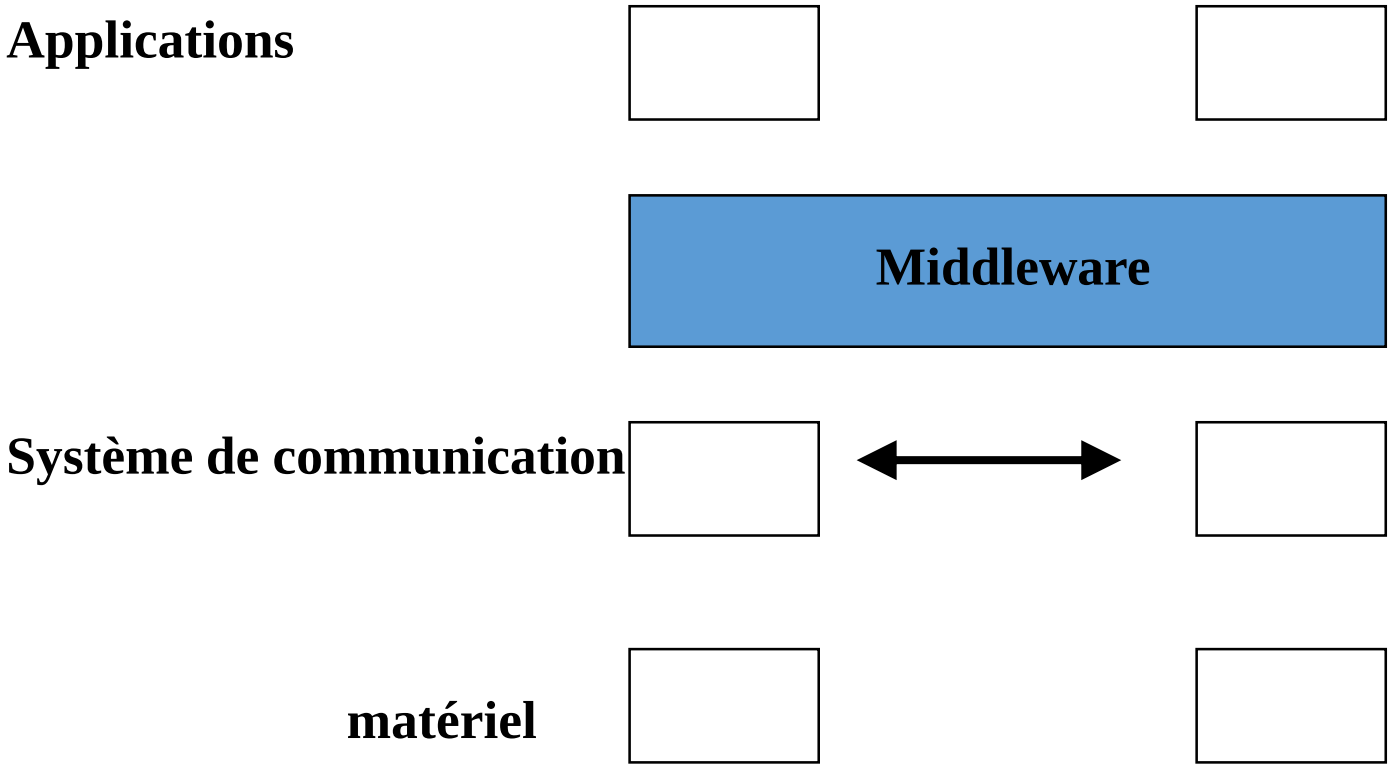
La communication entre applications a toujours été un problème majeur des systèmes répartis.

- Outils de base : les sockets (les prises, point de communication)

- Les technologies d'appel de procédure à distances (RPC, remote Procedure call) regroupant les standard tels que CORBA, RMI, DCOM (Microsoft DNA), .NET,  SOAP. Leur principe réside dans l'invocation d'un service (i.e. d'une procédure ou d'une méthode d'un objet) situé sur une machine distante indépendamment de sa localisation ou de son implémentation.

Un intergiciel (middleware) est un programme qui s'insère ente deux applications et qui va permettre de faire communiquer des machines entre elles, indépendamment de la nature du processeur, du système d'exploitation, du langage, Il est  destinée à :

- Faciliter l'interopérabilité des applications : Unifier l'accès à des machines distantes
- Masquer l 'hétérogénéité des machines & des systèmes : Etre indépendant des systèmes d'exploitation et du langage de programmation des applications
- Faciliter la programmation réseau
- Masquer la répartition des traitements et des données
- Fournir des services associés pour gérer : Fiabilité, la Sécurité


Applications
Middleware
Système de communication
matériel

# Modes synchrones et asynchrones

- Mode synchrone lorsque l'application cliente est bloquée en attente d'une réponse à une requête au serveur. Exemple: analogie du mode synchrone est le téléphone: les deux correspondants sont présents au moment de l'appel et chacun attend la réponse de l'autre pour parler.

  => ce mode permettre de s'assurer que le message a été envoyé et que l'on a bien reçu une réponse: on obtient plus de fiabilité

- Mode asynchrone lorsque l'appel n'est pas bloquant : l'application continue son déroulement sans attendre la réponse. Exemple: semblable à la communication par courrier.

  => ce mode permet de libérer du temps de calcul pour d'autre actions afin de paralléliser les tâches: on obtient plus de performance.

# Différents types des Middleware

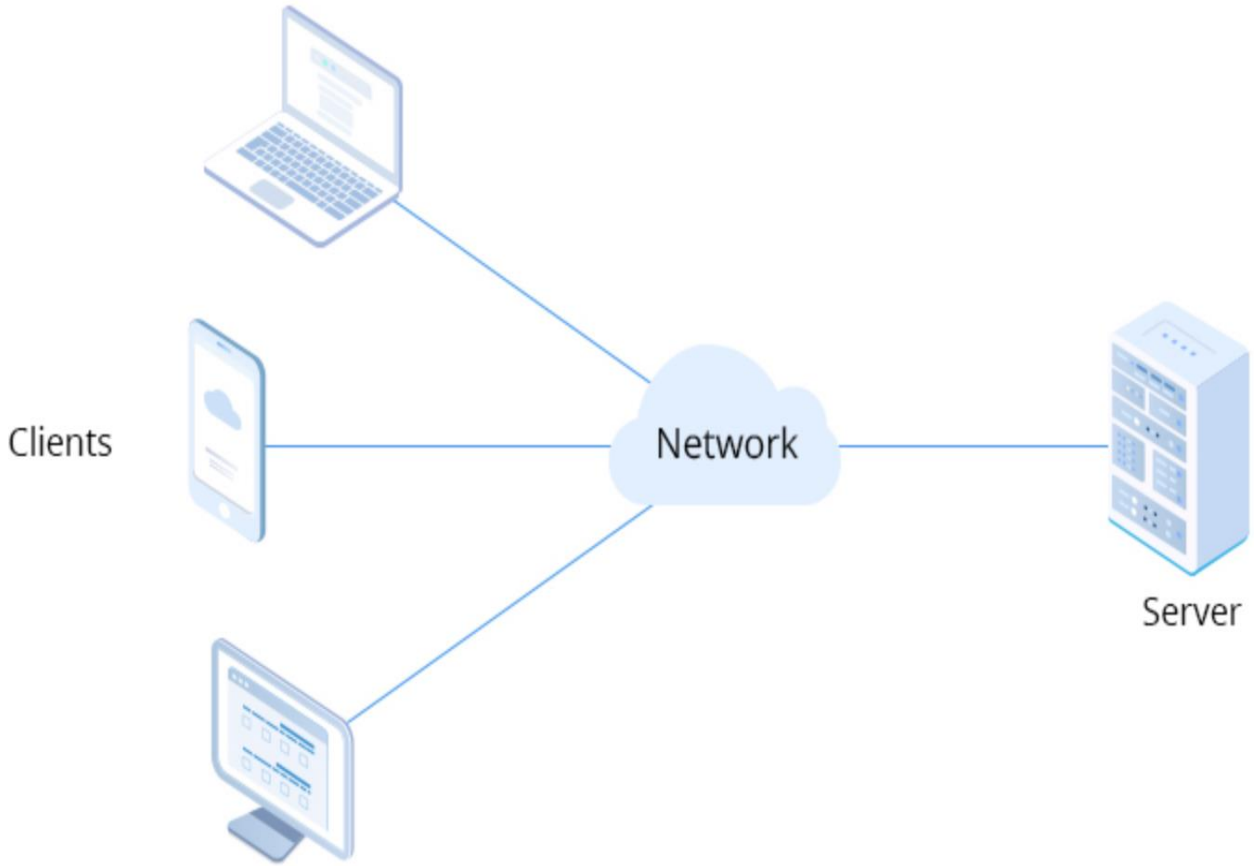
- Objets
  - JAVA RMI, CORBA (ORBIX, VisiBroker, OpenORB, …)
  - DCOM
- Composant
  - J2EE (Websphere, Weblogic, JBOSS)
  - .Net
- Web-Service

# modèles d'exécution

## *Le Modèle d'exécution client-serveur*

Une application client/serveur est une application qui fait appel à des services distants au travers d'un échange de messages (les requêtes) plutôt que par un partage de données (mémoire ou fichiers). Dans un modèle client/serveur une application se divise en deux parties :

- **serveur** : programme offrant un service sur un réseau (par extension, machine offrant un service). Donc c'est un programme qui s'exécute sur un réseau d'ordinateurs et qui est capable d'offrir un service. Le serveur reçoit une requête à travers le réseau, effectue le traitement nécessaire et retourne le résultat de la requête.
- **client** : un programme qui demande un service (envoi la requête) au serveur et attend la réponse. Il est toujours l'initiateur du dialogue.




**Modèle client-serveur**

## Avantages

**Gestion centralisée :** Toutes les données sont centralisées sur un serveur central, ce qui simplifie les contrôles en termes d'administration, de mise à jour des données et des logiciels.

**Sécurité :** Les données et les informations sont stockées sur un seul serveur plutôt que sur plusieurs clients, permettant ainsi d’assurer une meilleure protection contre les menaces extérieures.

**Évolutivité :** Il est possible d'ajouter ou supprimer des clients sans interrompre le fonctionnement normal des autres appareils.

## inconvénients

Le client-serveur a tout de même quelques inconvénients parmi lesquelles :

**Coût élevé :** Les serveurs peuvent être coûteux à mettre en place et à maintenir.

**En cas de panne :** une défaillance du serveur central risque de perturber tous les ordinateurs ou autres dispositifs du réseau client-serveur.

**Congestion du trafic :** Lorsqu'un grand nombre de clients envoient des demandes au même serveur, celui-ci risque de ne pas être en mesure de supporter la charge, ce qui peut entraîner un problème de congestion du trafic.

;

## Modèle peer to peer (pair à pair)

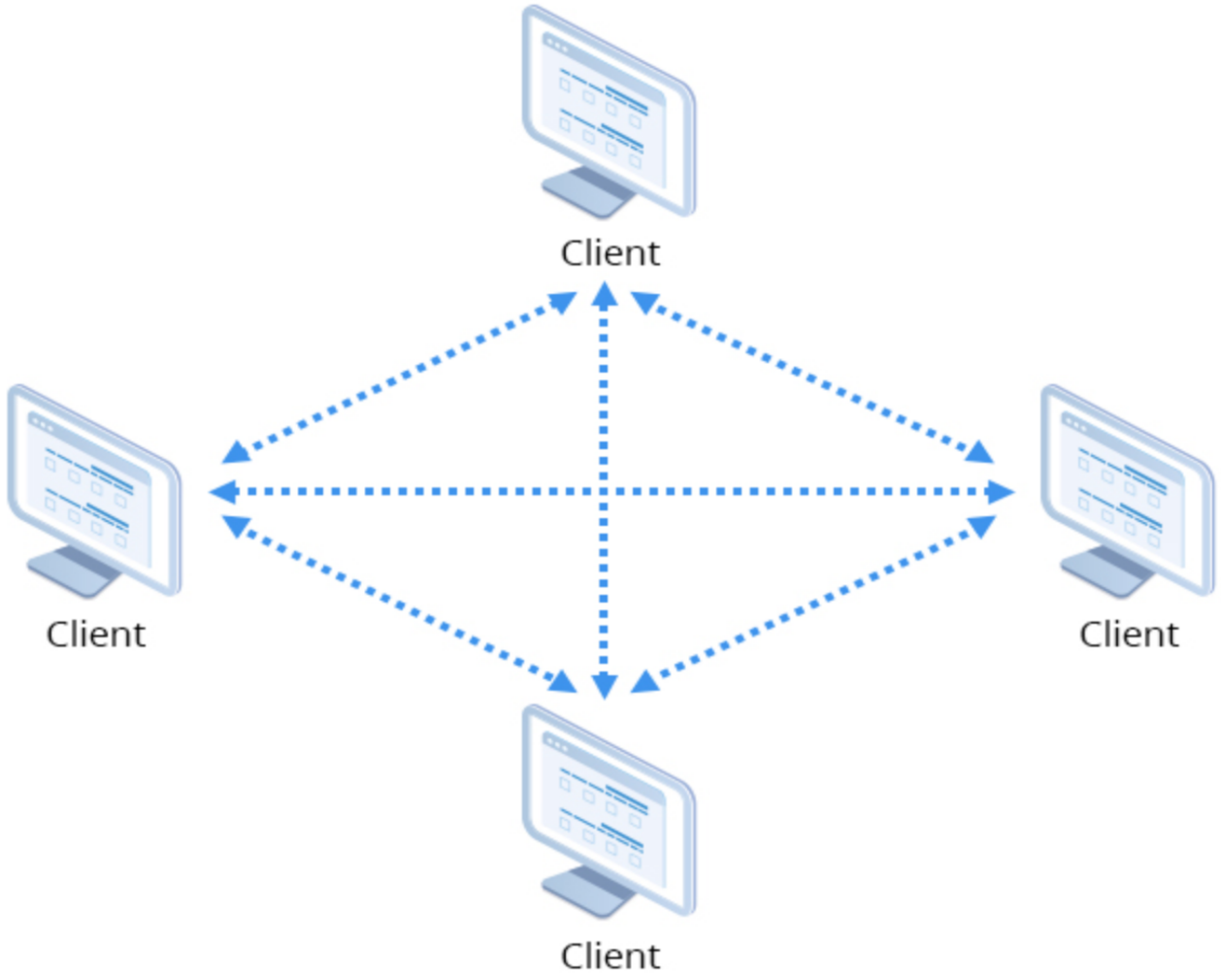
- Chaque ordinateur connecté au réseau est susceptible de jouer tour à tour le rôle de client et celui de serveur.
- Peut être centralisée (les connexions passant par un serveur central intermédiaire) ou décentralisée (les connexions se faisant directement).

### avantage

- Permet d'utiliser au mieux l'ensemble des ressources que représente chaque nœud d'un réseau.
- Permet un équilibre dans l'utilisation des ressources d'Internet tant au niveau de la bande-passante, que de l'espace de stockage ou de la capacité de traitement
- Réduire le coût du matériel utilisé

### Inconvénients

- La décentralisation des données et des service rend ce modèle très difficile à administrer
- La sécurité est moins facile à assurer, compte tenu des échanges transversaux ;




**Modèle peer to pee**

## Modèle à base de code mobile:

Un agent mobile est une entité logicielle qui se déplace d’un site à un autre en cours d’exécution pour accéder à des données ou à des ressources distante

## avantages

- satisfaire la tolérance aux pannes,
- réduire le trafic sur le réseau,
- suivre un composant matériel mobile
- Équilibrer la charge de calul

## Inconvénient

- La mise en place d'une politique de sécurité  des ressources et des données des machines hôtes et la préservation de l'intégrité et de la confidentialité des agents eux-mêmes et de leurs communications est une tâche délicate
- Dans le cas d'une faible interaction, cette approche devient moins efficace, car elle est pénalisée par la taille de l'agent.

# Exemples de systèmes répartis

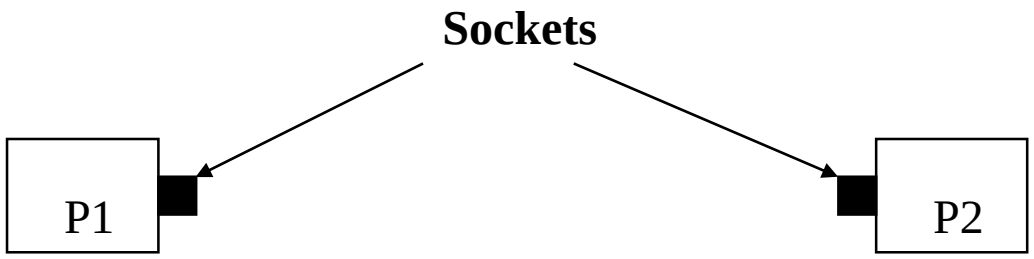
On rencontre des systèmes répartis dans la vie de tous les jours:

- Calcul scientifique
- Travail coopératif
- Téléconférence
- WWW, FTP, Mail
- Guichet de banque,
- Réservation des billets, des hôtels, …

## I. Définition:

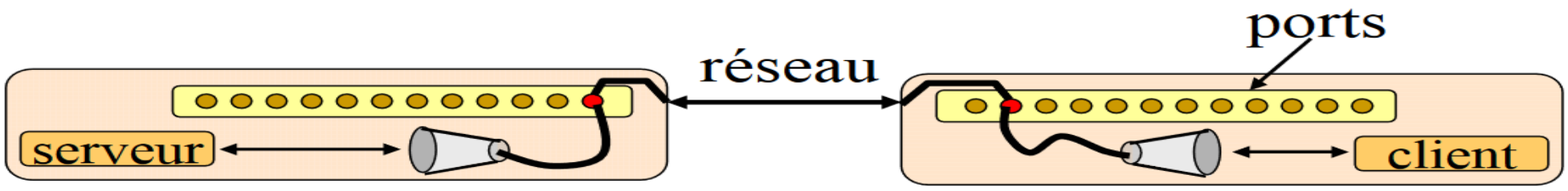
Un(e) socket désigne un point de communication par lequel un processus peut envoyer ou recevoir des informations.

- Elle est associée à un processus et identifiée dans ce processus par un descripteur de socket « socket descriptor »
- Sous UNIX, ce point de communication est représenté par une variable entière (int), manipulée comme un descripteur de fichier.
- Les sockets sont l'outil de base (de bas niveau) permettant la communication entre processus



# Principes de bases

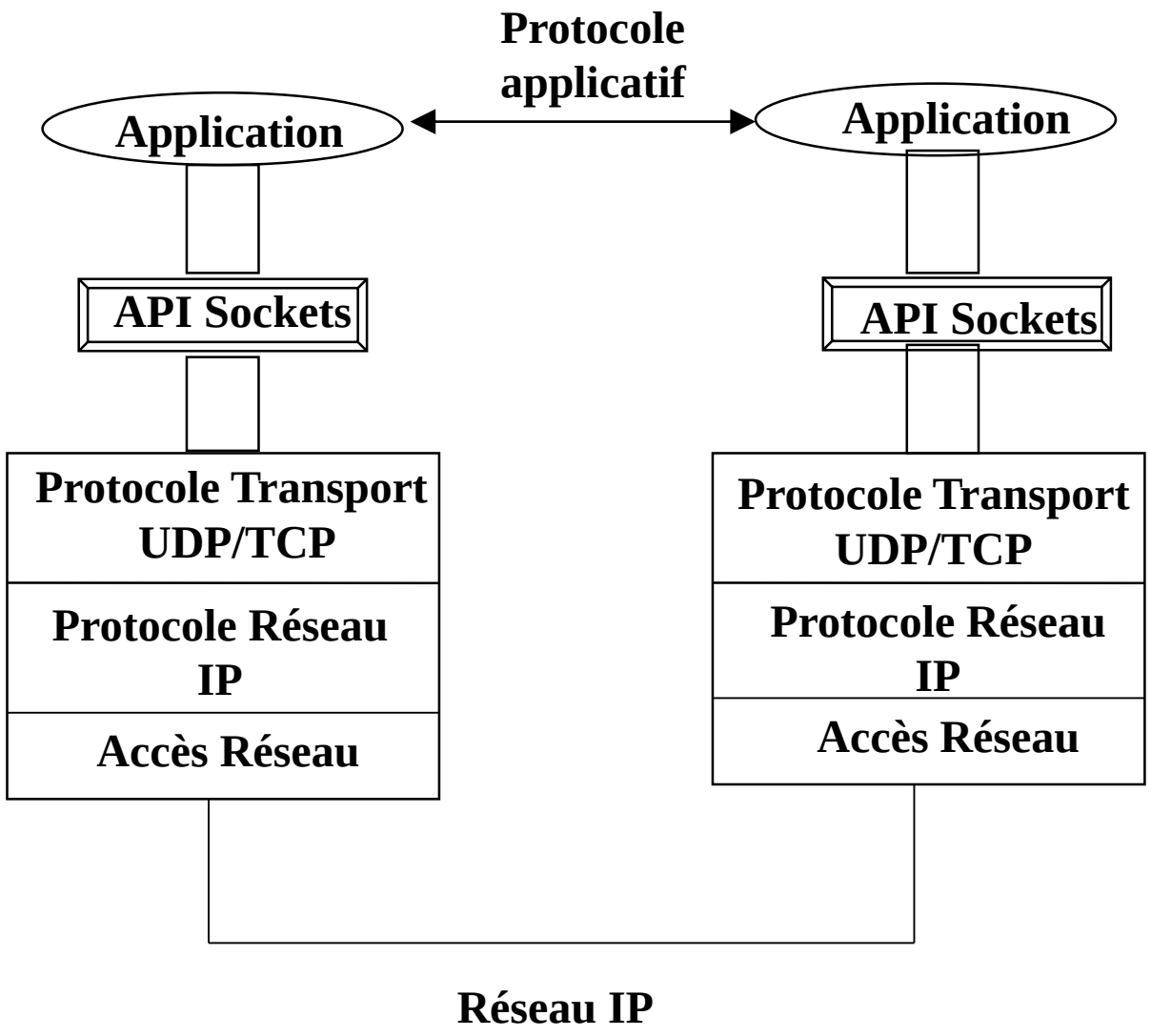
- Chaque processus crée une socket,
- Chaque socket sera associée à un port de sa machine hôte,
- Les deux sockets seront explicitement connectées si on utilise un protocole en mode connecté ...,
- Chaque processus lit et/ou écrit dans sa socket,
- Les données vont d'une socket à une autre à travers le réseau,
- Une fois terminé chaque machine ferme sa socket.



Le terme socket désigne aussi une bibliothèque d'interface de programmation ( pour les communications) entre les applications et les couches réseau.

⇒ c’est un ensemble de primitives (fonctions/procédures) d’accès aux deux protocoles de transport d’Internet : TCP et UDP

⇒ Il s'agit d'une interface souple mais de bas niveau.


Protocole
applicatif
Application
Application
API Sockets
API Sockets
Protocole Transport
UDP/TCP
Protocole Réseau
IP
Accès Réseau
Protocole Transport
UDP/TCP
Protocole Réseau
IP
Accès Réseau
Réseau IP

## Expression des adresses des machines et des numéros de ports

Une communication met en jeu 2 « extrémités » identifiées par :

- le type de protocole (UDP ou TCP): La couche Transport utilise le numéro de port pour livrer les données au processus approprié sur l'ordinateur de destination
- l'adresse IP: La couche Internet utilise l'adresse IP pour transférer des données d'un ordinateur à un autre.
- le numéro de port associé au processus
    - serveur : port local sur lequel les connexions sont attendues
    - client : allocation dynamique par le système

- Le concept de port

  Un port est  une manière d'identifier différentes communications qui passent par une même interface réseau. Il est:

  – référencé par les processus pour établir des communications entre processus

  – identifié sur une machine par un nombre entier (16 bits)

    - *Les numéros de port p.n. (port numbers) sont locaux aux machines: un même p.n. existe sur différentes machines.*

- Adressage

Adressage pour communication entre applications

- Adresse « réseau » d'une application noté

  *@IP : port* ou *nomMachine:port*

  - Adresse IP : identifiant de la machine sur laquelle tourne l'application
    - Exemple d'adresse IP: 192.129.12.34
  - Numéro de port : identifiant local de l'application sur la Couche transport (TCP ou UDP). Ce numéro est un entier:
    - de 0 à1023 : ports réservés. Ils sont assignés par l'IANA (Internet Assigned Numbers Authority). Ils donnent accès aux services standard :
      - Exemple : 80 = HTTP, 21 = FTP, 53: DNS, …
      - Exemple: 192.129.12.34:80 : accès au serveur Web tournant sur la machine d'adresse IP 192.129.12.34
    - p.n. > 1024
      - ports utilisateurs disponibles pour placer un service applicatif quelconque

Les @IP et les #port doivent être sous format réseau.

Certaines fonctions d'information sur le réseau répondent dans le format réseau : gethostbyname(...), gethostbyaddr(...), ...
Sinon il existe des fonctions de conversions de valeurs « machine » en valeur « réseau » (et réciproquement) :

```
#include <netinet/in.h>
    u_short htons(u_short);
        // host-to-network, short integer (entier court local vers réseau)
    u_long htonl(u_long);
        // host-to-network, long integer (entier long local vers réseau)
    u_short ntohs(u_short);
        // network-to-host, short integer (entier court réseau vers local)
    u_long ntohl(u_long);
        // network-to-host, long integer (entier long réseau vers local)
```

# Structures de données des adresses réseaux

- les adresses de socket sont conservées dans des structures sockaddr.  Le contenu de sockaddr dépend du protocole utilisé
- Exemples de spécialisation
  - Contexte IP: communication distante
    - Fichier <netinet/in.h>
    - struct sockaddr_in
  - Contexte Unix : communication locale à la même machine via des sockets
    - Fichier <sys/un.h>
    - struct sockaddr_un

## Les primitives générales de manipulation

### Création d'une socket

L'appel de la fonction socket ( ) créé un point de communication (une socket) pour permettre au processus de communiquer avec l'extérieur.

int socket(int *domaine*, int *type*, int *protocol*);

- *domaine* : définit l'ensemble des sockets avec lesquelles la socket créée pourra communiquer (spécifie les entités avec lesquelles la communication sera réalisé) et le format des adresses possibles.
  - AF_INET : domaine IP (contexte Internet)
    - Format des adresses: sockaddr_in (fichier standard <netinet/in.h>)
  - AF_UNIX: domaine UNIX (contexte local)
    - Format des adresses: sockaddr_un (fichier standard <sys/un.h>)

Remarque: Les appels sockets pour le domaine AF_UNIX pour faire communiquer deux processus se trouvant sur une même machine sont identiques à ceux pour le domaine AF_INET, ce qui change c'est les structures associées aux adresses.

- *type* : précise le protocole (le style) de communication:
  - SOCK_DGRAM: socket orienté vers la transmission de datagramme.
  - SOCK_STREAM: socket orienté vers l’échange de séquence continue de caractères (échange de flots d'octets). .

Remarque: Si on utilise les sockets au-dessus d'UDP, la taille d'un échange est limitée à quelques kilo-octets: de deux Ko à huit Ko selon les systèmes.

- *protocole* : spécifie le protocole à utiliser pour les communications, par exemple:
  - pour le type SOCK_DGRAM le seul protocole utilisé est UDP dans le domaine AF_INET
  - pour le type SOCK_STREAM  le seul protocole utilisé est TCP dans le domaine AF_INET

✓ Dans le fichier <netinet/in.h> on définit les constantes symbolique pour identifier les deux protocoles UDP et TCP pour le domaine AF_INET.

- La constante IPPROTO.UDP identifie le protocole UDP
- La constante IPPROTO.TCP identifie le protocole TCP

Exemple :

```
#include <netinet/in.h>
s1=socket(AF_INET, SOCK_DGRAM, IPPROTO.UDP);
s2=socket(AF_INET, SOCK_STREAM, IPPROTO.TCP);
```

⇒ Dans la pratique, on remplace l'argument *protocole* par la valeur 0 pour considérer le protocole par défaut associé à chaque type de socket.

Exemple :

```
// on utilise
        s1 =socket(AF_INET, SOCK_DGRAM, 0);
// au lieu de
    s1=socket(AF_INET, SOCK_DGRAM, IPPROTO.UDP);
```

- Valeurs de retour de la fonction socket()
  - En cas de succès, l’appel de la fonction socket() retourne un descripteur sur la socket créée (un identificateur de socket). Cet identificateur est locale à la machine et n’est pas connu par le destinataire.
  - En cas d'erreur, elle retourne - 1
    - Pour connaître le détail de l'erreur
      - On peut consulter la valeur de la variable errno. Liste des erreurs dans le fichier <ernno.h>
      - On peut afficher un message avec la fonction perror

## Nommage (attachement de la socket à une adresse)

- Après sa création,
  - une socket est désignée par un descripteur obtenu au retour de l'appel de la fonction socket().
  - une socket n'est pas directement accessible par l'extérieur (en dehors du processus qui l'a créé).

- Pour pouvoir être contactée de l'extérieur (accessible par l'extérieur) une socket doit être associée à une adresse. Le format de l'adresse est déterminée par le domaine de la socket
  - Dans le domaine Unix :
    - une adresse = une entrée dans le système de fichiers
  - Dans le domaine internet
    - une adresse = ( adresse de la machine , numéro de port)

L’association socket() – adresse est réalisée par l’appel de la fonction *bind()*.

int *bind* ( int *sock_desc*,
struct sockaddr **adresse*,
socklen_t *longueur_adresse*)

- *sock_desc* = descripteur de la socket,
- **adresse* = adresse sur laquelle la socket sera liée. Pointeur sur la structure contenant les informations sur la machine locale (domaine, nº de port, et adresse IP)
- *longueur_adresse* = taille de l'adresse. C’est la taille de la structure sockaddr obtenue avec la fonction sizeof()

✓La fonction retourne

- 0 en cas de succès (l’attachement a pu se réaliser)
- -1 en cas de problème

✓Problèmes les plus courants

– Adresse déjà utilisée (erreur EADDRINUSE)
– Liaison non autorisée (ex : port < 1024) sur ce port (erreur EACCES)

## Déclaration typique d'une adresse

struct sockaddr est un identifiant « général » qu'on ne l'utilise jamais directement mais une spécialisation selon le type de réseau ou de communication est utilisée

- Cas du domaine AF_UNIX: Le nom de la structure utilisée est *sockaddr_un* définie dans: /usr/include/sys/un.h

```
#include <sys/un.h>
struct sockaddr_un {
        short sun_family;     /* AF_UNIX*/
        char sun_path[108]  /* reference */
}
```

L'attachement d'une socket à une adresse ne peut se faire que si la référence correspondante n'existe pas.

- Cas du domaine AF_INET: Le nom de la structure utilisée est *sockaddr_in* definie dans /usr/include/netinet/in.h

```
#include <netinet/in.h>
    /* Adresse internet d'une socket */
struct sockaddr_in {
    short   sin_family;          /* AF_INET */
    u_short  sin_port;  /* numéro de port associé à la socket */
    struct  in_addr  sin_addr;/* adresse internet de la machine sur 32 bits */
    char sin_zero[8];            /* champs de 8 caracatères nuls : inutilise*/
};

    /* Adresse Internet d'une machine */
struct in_addr {
    u_long s_addr;
}
```

Peut être aussi définie:

```
#include <netinet/in.h>
struct sockaddr_in{
        short sin_family;
        u_short sin_port;
        struct in_addr {
                u_long s_addr;
        } sin_addr;
        char sin_zero[8];
};
```

• Les valeurs des champs sin_addr et sin_port peuvent avoir des valeurs particulières:

- sin_family = AF_INET  /* Domaine Internet */
- sin_addr.s_addr = INADDR_ANY
  - INADDR_ANY : permet d'utiliser n'importe quelle adresse IP de la machine si elle en a plusieurs et/ou évite de connaître l'adresse IP locale.
- sin_port = 0, l'attachement peut se faire sur un port de numéro quelconque. Ceci est particulièrement intéressant pour les clients.

- Remarque: Les valeurs des champs sin_addr, sin_port et s_addr sont sous format réseau

# Identifiants d'une machine

Fichier <netdb.h>

```
    struct hostent {
        char *h_name;                   // nom officiel,
        char **h_aliases;               // liste des alias,
        int h_addrtype;                 // type d'adresse,
        int h_length;                   // longueur de l'adresse,
        char **h_addr_list;             //  liste des adresses
        #define h_addr h_addr_list[0]   // première adresse
};
```

- Type d'adresse : internet (IP v4) par défaut
  - domaine = AF_INET, longueur = 4 (en octets)
- Une machine peut avoir plusieurs adresses IP et plusieurs noms

## Accès aux identifiants de machines distantes

La fonction gethostbyname( ) retourne:

- l'identifiant de la machine dont le nom est passé en paramètre.
- NULL si aucune machine de ce nom est trouvée.

```
#include <netdb.h>
 struct hostent *gethostbyname(char *nom);
```

- La structure retournée est placée à une adresse statique en mémoire.
  - Un nouvel appel de gethostbyname() écrase la valeur à cette adresse.
  - Si on veut la conserver, il est nécessaire de copier la structure avec un memcopy() ou un bcopy().

Exemple: Récupération du client de l'identifiant de la machine distante de nom "PGR"

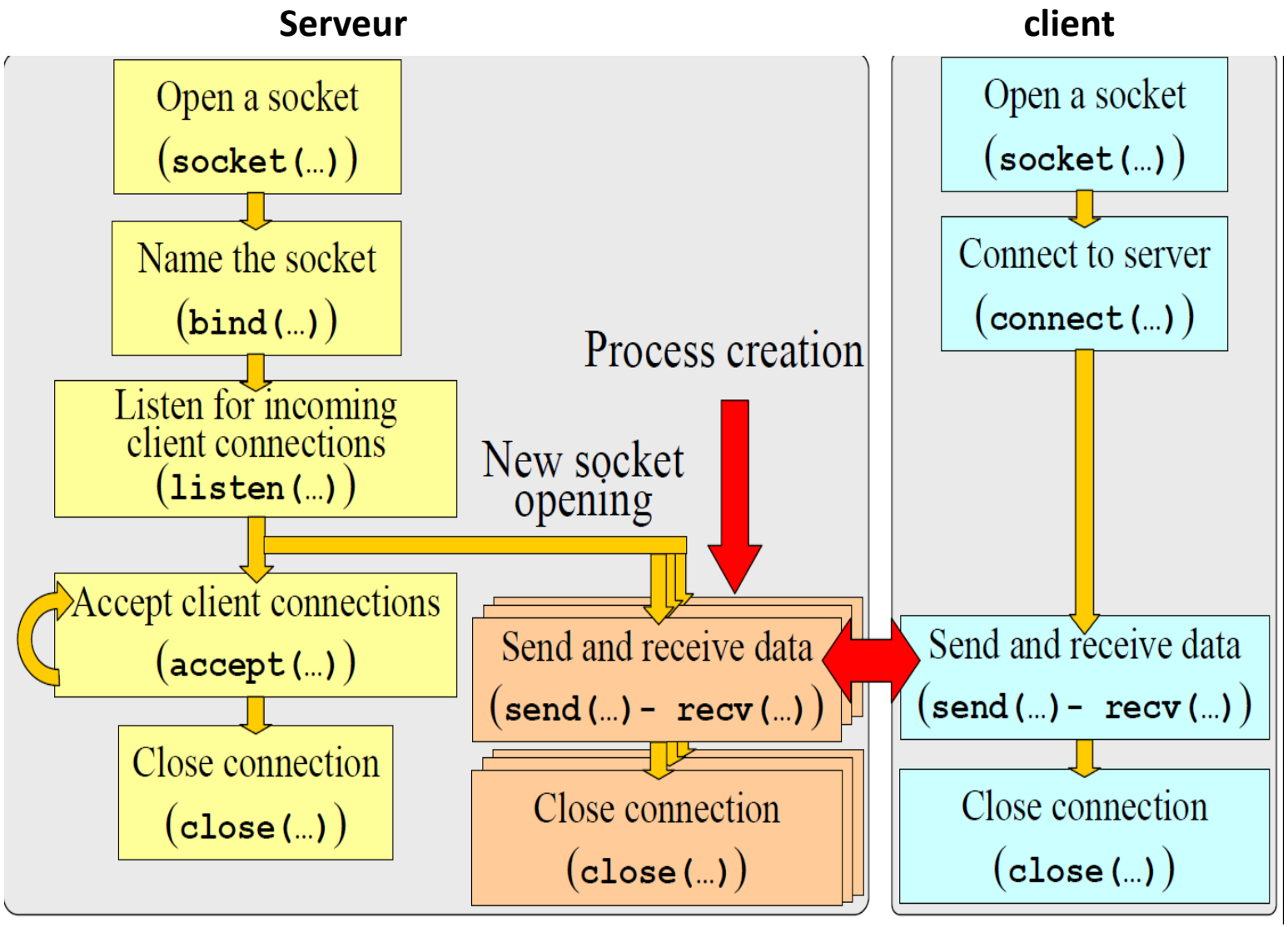
```
static struct sockaddr_in      addr_serveur;    // identification de la socket d'écoute du serveur
struct hostent *host_serveur;                   // identifiants de la machine où tourne le serveur
host_serveur = gethostbyname("PGR");      // récupération identifiants de la machine serveur

// création de l'identifiant de la socket d'écoute du serveur
bzero((char *) &addr_serveur, sizeof(addr_serveur));
    //       #include <string.h>
    //       void bzero (void *s, size_t n);    // met à 0 les n premiers octets du bloc pointé par s.
addr_serveur.sin_family = AF_INET;
addr_serveur.sin_port = htons(4000);    // le même numéro de port définie lors
                                        // de l'appel de  bind (s_ecoute) dans le serveur
memcpy(&addr_serveur.sin_addr.s_addr, host_serveur->h_addr, host_serveur->h_length);
```

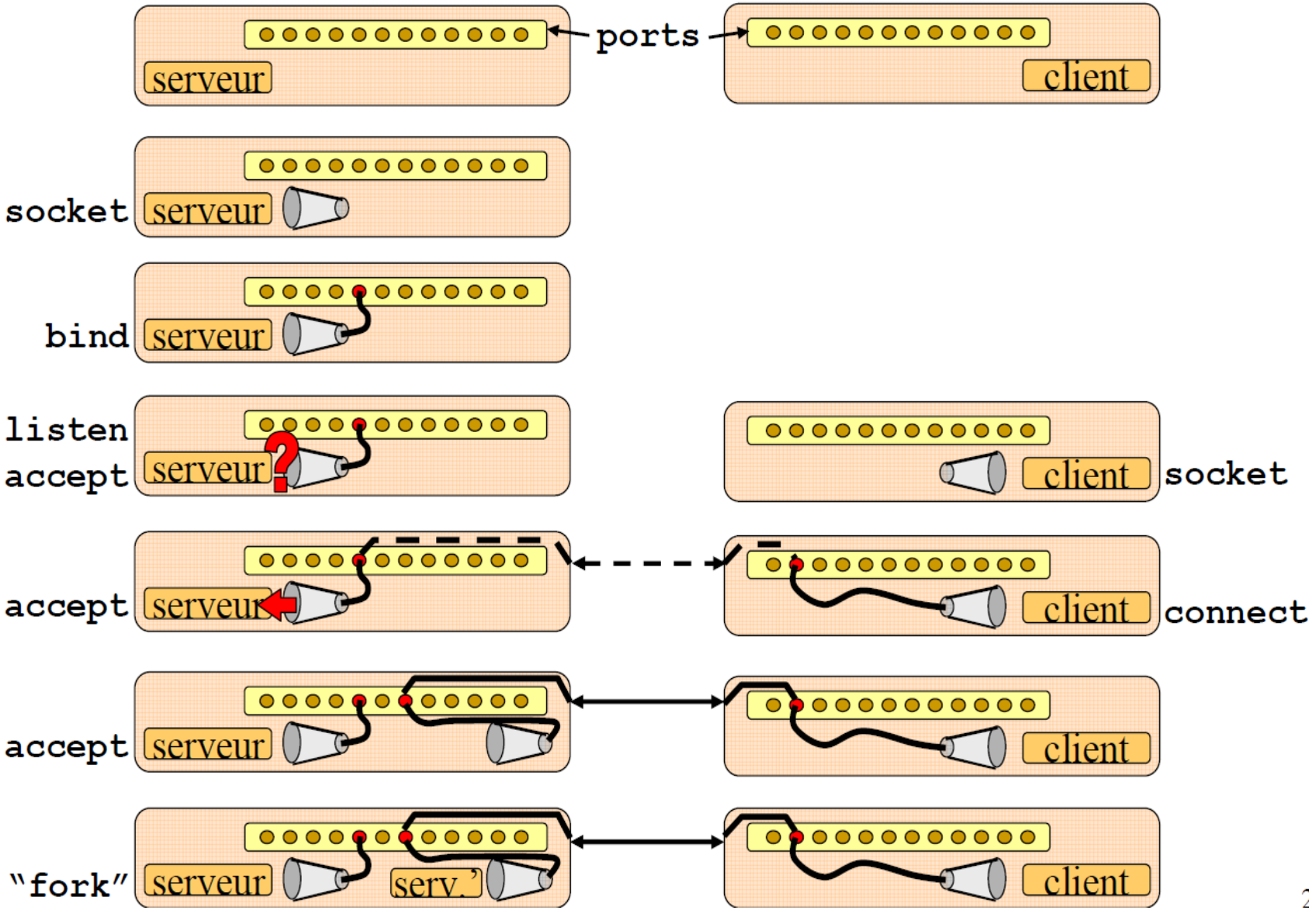
# Communications en mode connecté (mode TCP):

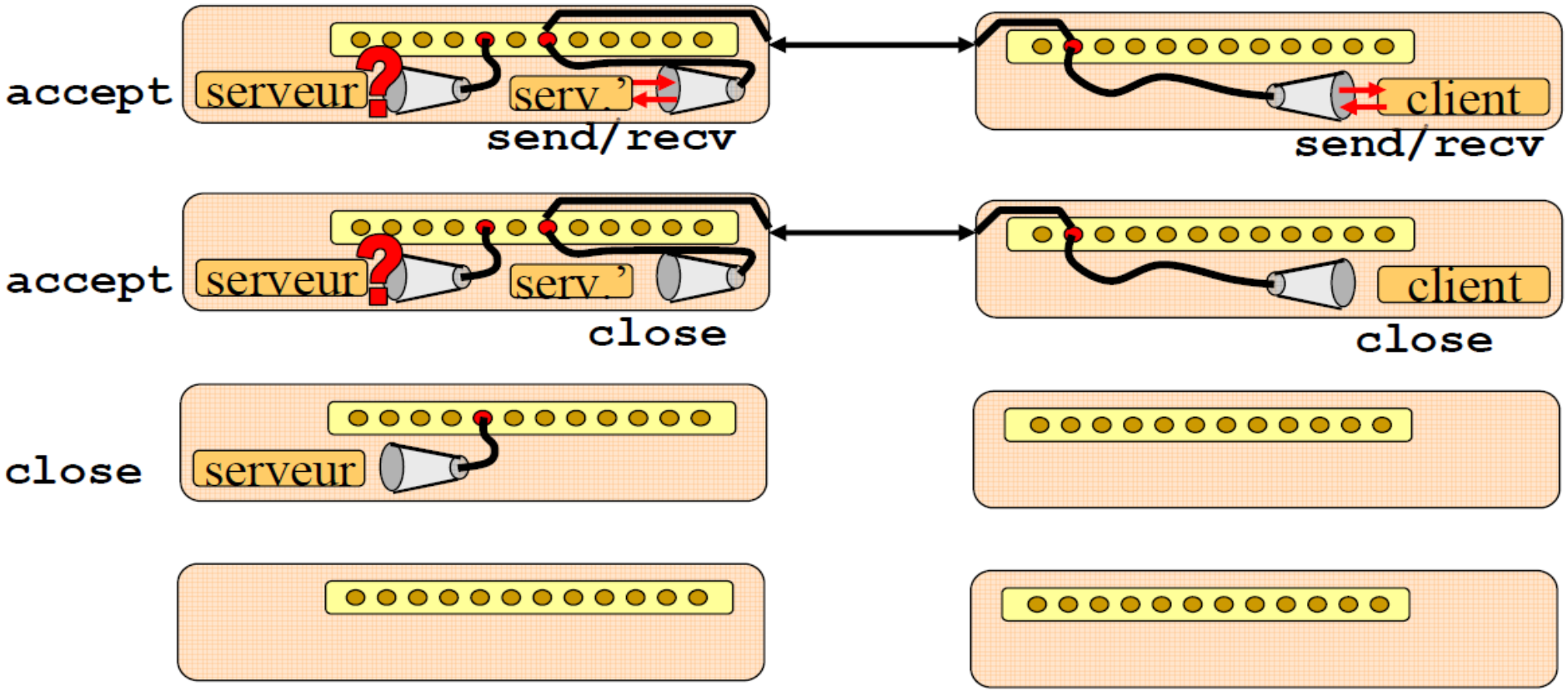
## Déroulement de la communication

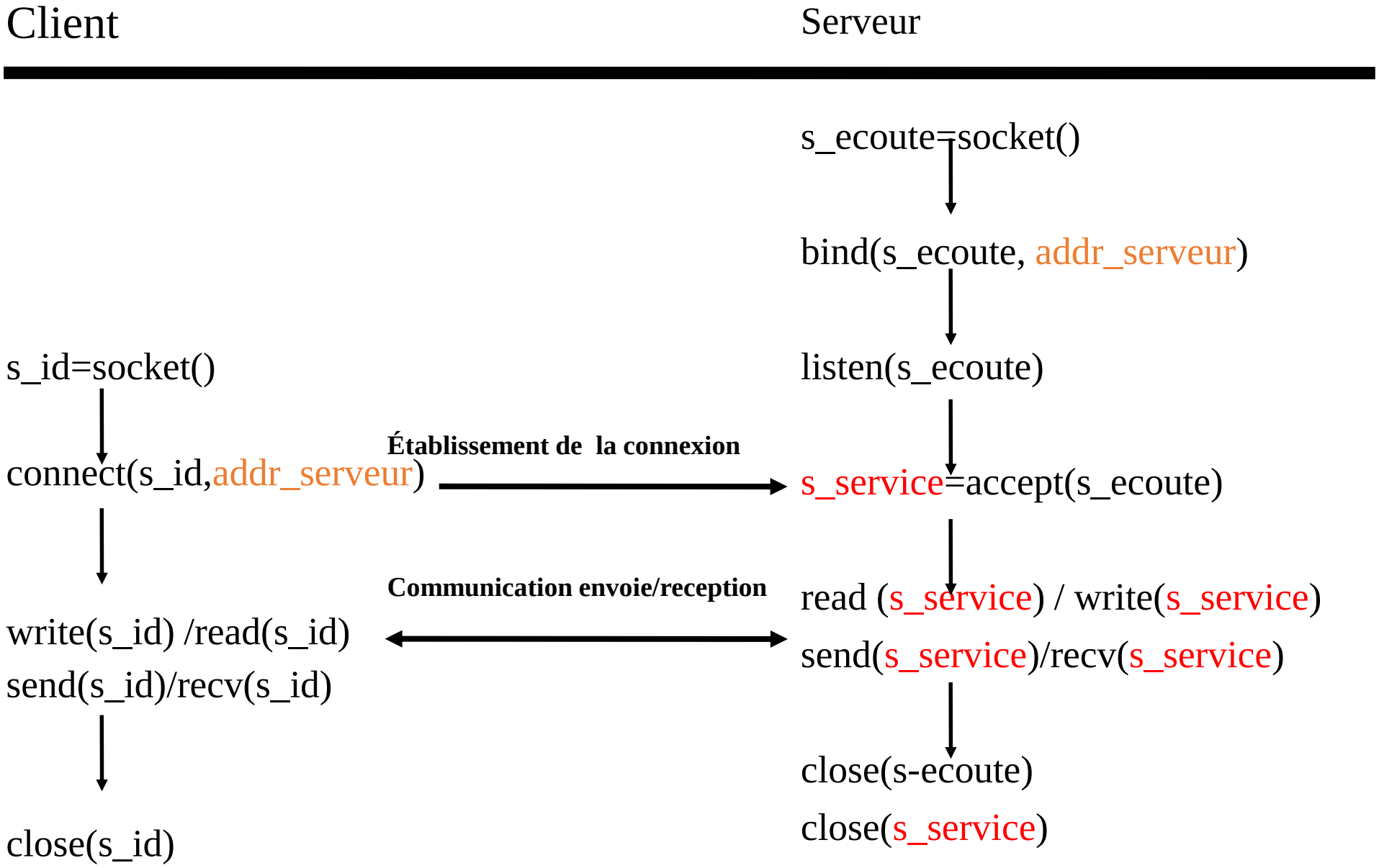
- L'appelé (ou serveur) :
  - créé une socket  (socket d'écoute);
  - associe une adresse socket (adresse Internet et numéro de port) au service:  "binding";
  - se met en attente des connexions entrantes ;
  - pour chaque connexion entrante:
    - "accepte" la connexion (une nouvelle socket (socket de service) est créée);
    - crée un processus pour dialoguer avec le client
    - le nouveau processus continue le dialogue avec le client sur la nouvelle socket
    - Le serveur attend en parallèle de nouvelles demandes de connexions
    - ferme la nouvelle socket.

- L'appelant (ou client) :
  - créée une socket ;
  - se connecte au serveur en donnant l'adresse Internet du serveur et le numéro de port du service. Cette connexion attribue automatiquement un numéro de port au client;
  - Dialogue avec le serveur sur la socket ;
  - ferme la socket.


Serveur
client
Open a socket
(socket(...))
Name the socket
(bind(...))
Listen for incoming
client connections
(listen(...))
Accept client connections
(accept(...))
Close connection
(close(...))
Process creation
New socket
opening
Send and receive data
(send(...)- recv(...))
Close connection
(close(...))
Open a socket
(socket(...))
Connect to server
(connect(...))
Send and receive data
(send(...)- recv(...))
Close connection
(close(...))


ports
serveur
client
socket
bind
listen
accept
socket
accept
connect
accept
"fork"
serv.'


accept
serveur
serv.'
send/recv
client
send/recv
accept
serveur
serv.'
close
client
close
close
serveur


Client
Serveur
s_ecoute=socket()
bind(s_ecoute, addr_serveur)
listen(s_ecoute)
s_id=socket()
Établissement de la connexion
connect(s_id,addr_serveur)
s_service=accept(s_ecoute)
Communication envoie/reception
write(s_id) /read(s_id)
send(s_id)/recv(s_id)
read (s_service) / write(s_service)
send(s_service)/recv(s_service)
close(s_id)
close(s-ecoute)
close(s_service)

Remarque:

accept() retourne un nouveau descripteur de socket (socket de service), qui sera utilisé pour l'échange de données avec le client.

Sur quel type d'événement se bloquent ces fonctions:

- Côté client :
  - connect() attend qu'un serveur effectue un accept();
  - write() se bloque si le tampon d'émission est plein;
  - read() attend qu'un caractère au moins ai été reçu, à la suite d'une opération d'écriture effectuée par le serveur.
- Côté serveur:
  - accept() attend qu'un client effectue un connect();
  - read() attend un caractère, émis par un client;
  - write() se bloque si le tampon d'émission est plein.

# Mise en place de canal de communication

## Coté client

## Ouverture d'une connexion

- L'appel de la fonction *connect()* permet d'ouvrir une connexion avec la partie serveur (connexion sur la socket d'écoute du serveur)

int connect(int s_id, struct sockaddr *addr_s_ecoute, int length_addr);

  - Paramètres
    - s_id : descripteur de la socket coté client
    - addr_s_ecoute : identifiant de la socket d'écoute coté serveur (socket_serveur)
    - length_addr : taille de l'adresse utilisée
  - Retourne
    - 0 en cas de succès,
    - -1 sinon

## Coté serveur:

## Définition de la taille de la file d’attente sur une socket

- La fonction listen() permet de configurer le nombre maximum de connexions en attente (le nombre maximum de connexions pendantes) à un instant donné.

  int listen(int s_id, int nb_con_attente);

  - Paramètres
    - s_id : descripteur de la socket d'écoute à configurer (socket chez le serveur)
    - nb_con_attente : nombre max de connexions pendantes autorisées (taille de la file d’attente sur la socket shez le serveur). Le nombre de connexion pendantes doit être inférieur à la valeur de la constante SOMAXCONN (fichier <sys/socket.h>). Cette valeur dépend des systèmes d'exploitation
    - Exemple: SOMAXCONN = 128 sous Linux
  - Retourne
    - 0 en cas de succès
    - -1 en cas de problème

## Attente d’une demande de connexion:

accept() est la fonction qui attend la connexion d'un client sur une socket d'écoute

s_service= accept(int s_ecoute, struct sockaddr *addr_client, int *length_addr);

- Paramètres
  - s_service : c’est le numéro de la socket (appelée socket de service) renvoyé par l’appel de la fonction accept() dès qu’il reçoit une demande de connexion.
  - s_ecoute : la socket d'écoute (à lier sur un port précis).
  - addr_client : contiendra l'addresse du client qui se connectera
  - length_addr : contiendra la taille de la structure d'address

- Retourne un descripteur de socket
  - La socket de service qui permet de communiquer avec le client qui vient de se connecter
  - Retourne -1 en cas d'erreur
  - Fonction bloquante jusqu'à l'arrivée d'une demande de connexion

## envoi/réception données

- Si la connexion a réussi
  - un « tuyau » de communication directe est établit entre la socket du coté client et la socket de service du coté serveur

- Fonctions de communication
  - On peut utiliser les fonctions standards pour communiquer
    - write()/send() pour l'envoie
    - read()/recv() pour la reception

## Envoie des données :

ssize_t  write(int s_id, void *ptr_mem, size_t  taille);

ssize_t  send(int s_id, void *ptr_mem, size_t  taille, int option);

- Paramètres
  - s_id : descripteur de la socket
  - ptr_mem : adresse de la zone mémoire des  données à envoyer.
  - taille : nombre d'octets à envoyer
  - option prend la valeur 0

- En cas de succès, elle retourne le nombre d'octets envoyés ou -1 en cas de problème

## Réception des données

ssize_t read(int s_id, void *ptr_mem, size_t taille);

ssize_t recv(int s_id, void *ptr_mem, size_t taille, int option);

- Paramètres
  - s_id : descripteur de la socket
  - ptr_mem : adresse du début de la zone mémoire où seront écrites les données reçues.
  - taille : taille de la zone mémoire
  - option prend la valeur 0
- Retourne le nombre d'octets reçus ou -1 en cas de pb

## Fermeture de la socket

- La fonction close() gère la fermeture de la connexion au niveau système (fichiers).

  int close(int s_id)

  s_id : descripteur de socket

- Le noyau essaie d'envoyer les données non encore émises avant de sortir du close().

Exemple:

- Serveur attend la connexion du client
- Client
  - envoie la chaîne “bonjour” au serveur
  - et ensuite attend une réponse
- Serveur
  - reçois la chaîne « bonjour »
  - et ensuite envoie un accusé de reception (la chaîne “bien reçu”

- travail en local (Mode AF_UNIX) lancé sur la machine “PGR”
  - on écrit 2 programme qui communique socket_clientTCP_Unix.c et socket_serveurTCP_Unix.c
- refaire le même travail (en local) mais en en mode AF_INET lancé sur la machine “PGR” sur le port 6000

## coté serveur: fichier socket_serveurTCP_UNIX.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#define TAILLEBUF 20

int main(int argc, char *argv[]) {
    static struct sockaddr_un addr_client; // adresse socket coté client
    static struct sockaddr_un addr_serveur; // adresse socket locale
    int lg_addr; // longueur adresse
    int s_ecoute, s_service; // socket d'écoute et de service
    char message[TAILLEBUF]; // buffer qui contiendra le message reçu
    char *chaine_recue; // chaîne reçue du client
    char *reponse = "bien recu"; // chaîne renvoyée au client
    int nb_octets; // nombre d'octets reçus ou envoyés
```

```
s_ecoute = socket(AF_UNIX, SOCK_STREAM, 0); // création socket TCP d'écoute
 if (s_ecoute == -1) {
          perror("creation socket");
          exit(1); }
  addr_serveur.sun_family = AF_UNIX;
  memcpy(addr_serveur.sun_path,"sock",strlen("sock")); // sock  reference de la socket
  if( bind(s_ecoute, (struct sockaddr*)&addr_serveur, sizeof(addr_serveur))== -1 ) {
      perror("erreur bind socket écoute");
      exit(1); }

       // configuration socket écoute : 5 connexions max en attente
 if (listen(socket_ecoute, 5) == -1) {
        perror("erreur listen");
        exit(1); }
```

```
// on attend la connexion du client
 lg_addr = sizeof(struct sockaddr_un);
 s_service = accept(s_ecoute, (struct sockaddr *)&addr_client, &lg_addr);
  if (s_service == -1) {
        perror("erreur accept");
        exit(1);
  }
```

```
// la connexion est établie, on attend les données envoyées par le client
  nb_octets = read(socket_service, message, TAILLEBUF);

  // affichage du message
  chaine_recue = (char *)malloc(nb_octets * sizeof(char));
  memcpy(chaine_recue, message, nb_octets);
  printf("recu message %s\n", chaine_recue);

  // on envoie la réponse au client
  nb_octets = write(s_service, reponse, strlen(reponse)+1);

  // on ferme les sockets
   close(s_service);
  close(s_ecoute);
}
```

coté client: fichier socket_clientTCP_UNIX.c

```
#include <stdio.h>
  #include <stdlib.h>
  #include <sys/types.h>
  #include <sys/socket.h>
  #include <sys/un.h>
  #define TAILLEBUF 20
int main(int argc, char *argv[]) {
    static struct sockaddr_un addr_serveur; // identification socket d'écoute du serveur
    int s_id;  // socket locale coté client
    char *message = "bonjour"; // message à envoyer au serveur
    char reponse[TAILLEBUF];  // chaîne où sera écrit le message reçu
    int nb_octets; // nombre d'octets envoyés/reçus
```

```
s_id = socket(AF_UNIX, SOCK_STREAM, 0);
if (s_id == -1) {
      perror("creation socket");
      exit(1);
 }
 addr_serveur.sun_family = AF_UNIX;
 memcpy(addr_serveur.sun_path,"sock",strlen("sock"));
       // connexion de la socket client locale à la socket coté serveur
 if (connect(s_id,   (struct sockaddr *)&addr_serveur,
                    sizeof(struct sockaddr_un)) == -1) {
        perror("erreur connexion serveur");
        exit(1);
   }
```

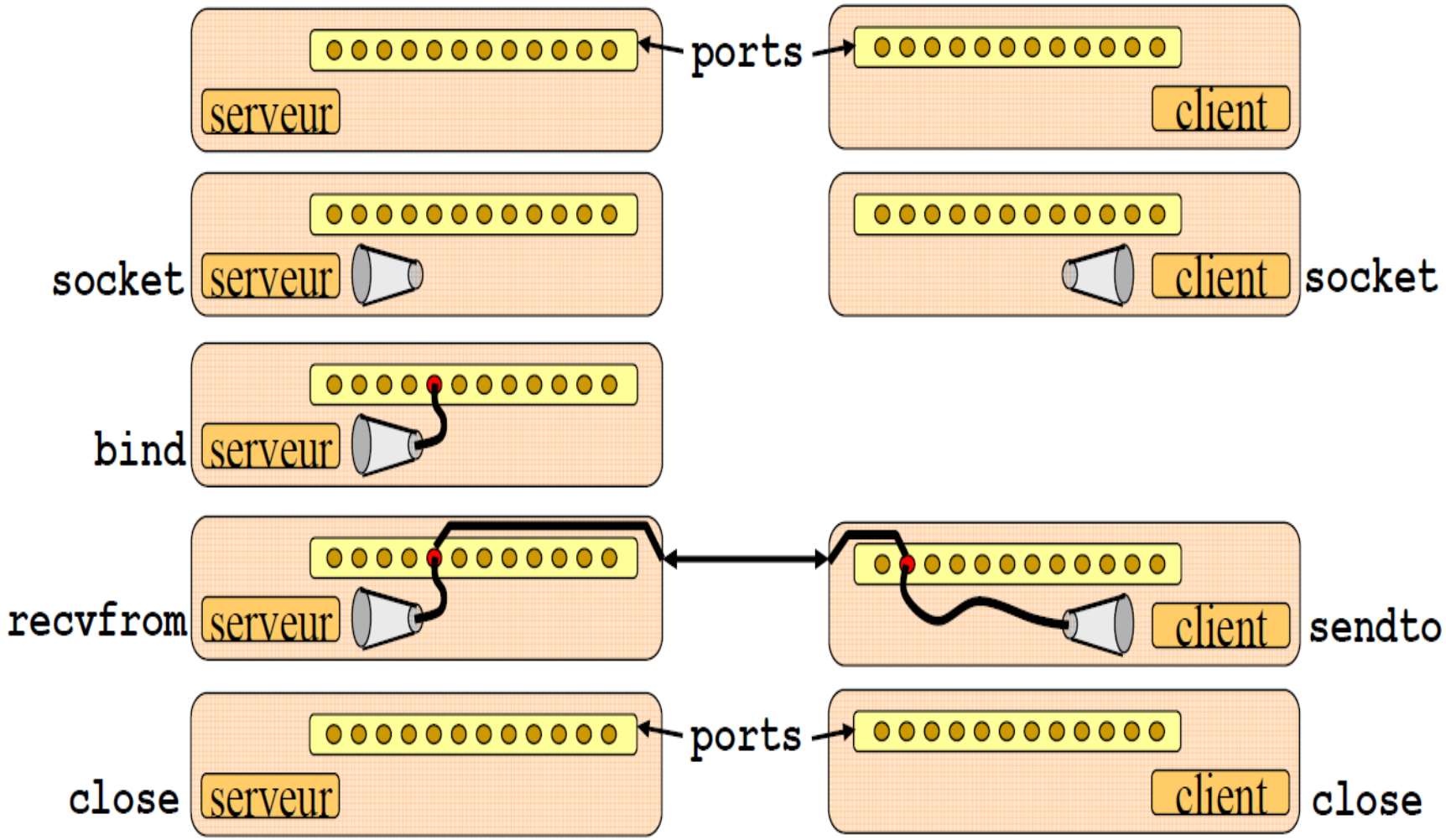
```
  nb_octets = write(s_id, message, strlen(message)+1);
            // connexion etablie, on envoie le message
   nb_octets = read(s_id, reponse, TAILLEBUF);
            // on attend la réponse du serveur
   printf(" reponse recue : %s\n", reponse);
     // on ferme la socket
   close(s_id);
}
```

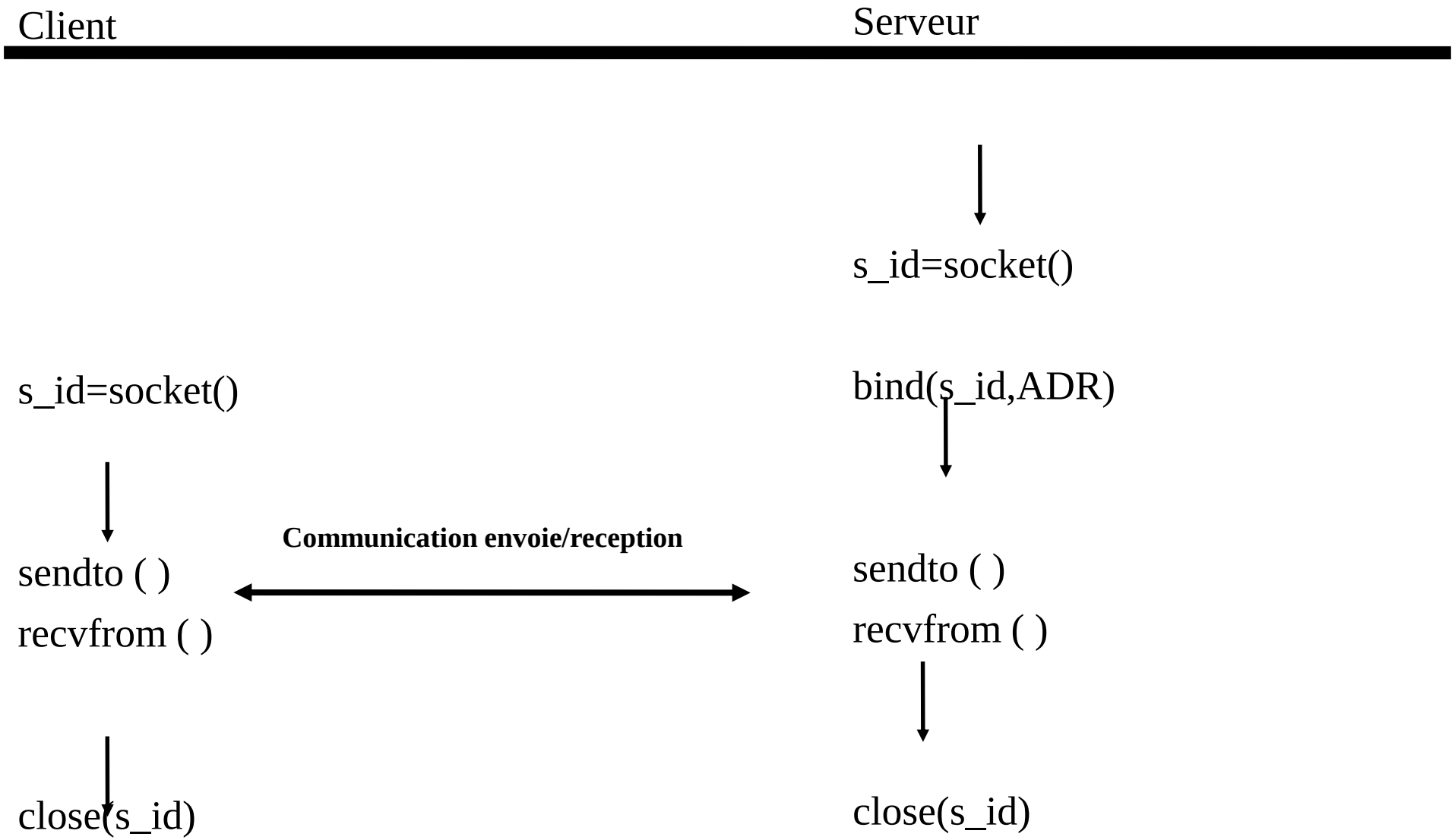
# Communications en mode datagramme (mode UDP)

Dans cette partie nous nous intéressons aux communications interprocessus par l'intermédiaire de sockets du type SOCK_DRAM.

## Déroulement de la communication

- Client (appelant) :
  - créée une socket ;
  - lit ou écrit sur la socket ;
  - le client ne nomme pas sa socket(elle sera attachée automatiquement à un port lors de l'émission)
- Serveur (appelé) :
  - créée une socket ;
  - la nomme (il l'attache à un de ses ports (un port précis)):"binding" ;
  - lit ou écrit sur la socket.


ports
serveur
client
socket
serveur
client
socket
bind
serveur
recvfrom
serveur
client
sendto
close
serveur
ports
client
close


Client
Serveur
s_id=socket()
bind(s_id,ADR)
s_id=socket()
Communication envoie/reception
sendto ( )
recvfrom ( )
sendto ( )
recvfrom ( )
close(s_id)
close(s_id)

# Envoi/réception de données en mode UDP

## Envoi de données en UDP sur socket

int sendto( int s_id, void *message, int m_taille, int option, struct sockaddr *adresse,
int l_adresse)

s_id = descripteur de la socket qui envoie les données,

*message = pointeur vers la zone de données à envoyer (adresse du début du message transmis)

m_taille = taille du message (longueur des données) en octets,

option = 0,

*adresse = adresse de la socket destinatrice,

l_adresse = longueur de l'adresse

- 2 types d'informations à passer en paramètre
  - Lien vers les données à émettre
  - L'adresse identifiant le destinataire (couple @IP/port)
- Valeurs de Retour
  - Nombre de caractères (réellement) envoyés
  - -1 en cas de problème

## Réception de données en UDP sur une socket

int recvfrom(int d_socket, void *message, int longueur, int option, struct sockaddr *adresse,  int *longueur_adresse)

d_socket = descripteur de la socket qui attend les données,

*message = adresse où le contenu du message sera récupérer

m_taille = taille réservé pour la réception du message,

option = 0

*adresse = adresse de la socket émettrice,

*longueur_adresse = longueur de l'adresse, à initialiser

- Paramètres à fournir
  - Lien et taille de la zone de données qui contiendra les données reçues
  - longueur_adresse : longueur de l'adresse, à initialiser.

- L’appel de la fonction *recvfrom()* retourne
  - Le nombre de caractères reçus
  - -1 en cas de problème

- En retour via les pointeurs
  - Zone de données initialisée avec les données
  - Adresse : contient l'adresse de la socket émettrice (longueur_adresse contient la taille de cette adresse)

## Remarques

- Taille de la zone de données : Dans le cas ou le message reçu est de longueur supérieure à l'espace réservé, le mesage est tronqué, le reste des informations étant perdues.
- Taille des datgramme UDP: il est conseillé de ne pas dépasser 8176 octets. Car la plupart des systèmes limitent à 8 Ko la taille des datagrammes UDP
    - Pour être certain de ne pas perdre de données : 512 octets max
    - Si datagramme UDP trop grand : les données sont tronquées

# Application multi-clients

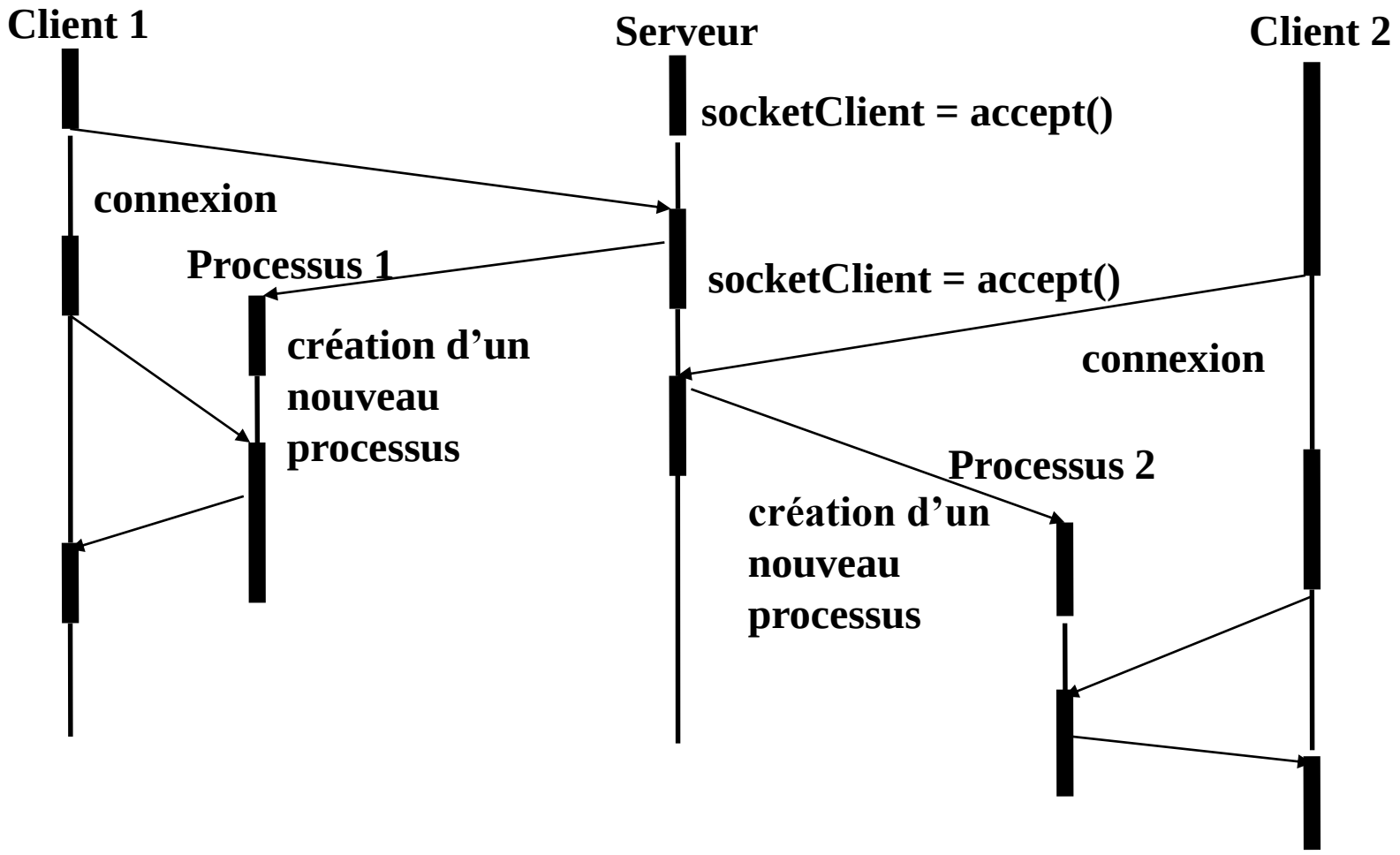
- Application client/serveur classique
  - Un serveur
  - Plusieurs clients
  - Le serveur doit pouvoir répondre aux requêtes des clients sans contraintes sur ordre d'arrivée des requêtes
- Contraintes à prendre en compte
  - Chaque élément (client ou serveur) s'exécute indépendamment des autres et en parallèle des autres

- Fonctionnement de TCP impose des contraintes
  - Attente de connexion : opération bloquante
    - Jusqu'à la prochaine connexion d'un client distant
  - Lecture sur une socket : opération bloquante
    - Tant que des données ne sont pas reçues
- Avec un seul flot d'exécution (processus/thread)
  - Si on ne sait pas quel est l'ordonnancement des arrivées des données des clients ou de leur connexion au serveur
    - Impossible à gérer
- Donc nécessité de plusieurs processus ou threads
  - Un processus en attente de connexion sur le port d'écoute
  - Nouvelle connexion : un nouveau processus est créé pour gérer la communication avec le nouveau client

Boucle de fonctionnement général d'un serveur pour gérer plusieurs clients

```
while(true)
    socketClient = accept();
    newProcessus(socketClient);
```

---



- Création d'un nouveau processus via un fork() à chaque connexion de client
  - Le processus principal fait les accept() en boucle et créé un nouveau processus fils pour la communication avec un nouveau client connecté

```
while(1) {
    s_service = accept(s_ecoute, (struct sockaddr *)&addr_client, &lg_addr);
    if (fork() == 0) {
        // on est dans le fils
        close(s_ecoute);
        // fonction qui gère la communication avec le client
        traiter_communication(s_service);
        exit(0);
    }
    close(socket_service);
}
```

- Pour que le serveur devienne un démon
  - Au départ du lancement du serveur, on crée un fils qui exécute un setsid() et on termine le processus principal
  - Le code du serveur (la boucle principale) est exécutée via ce fils

```
...
if ( fork() != 0 ) exit(0);
setsid();  // excute le programme dans une nouvelle session
...
while(1) {
      socket_service = accept(...)
      ...
}
```

# *Chapitre 3* **Appel de procédure distante (RPC)**

## Introduction

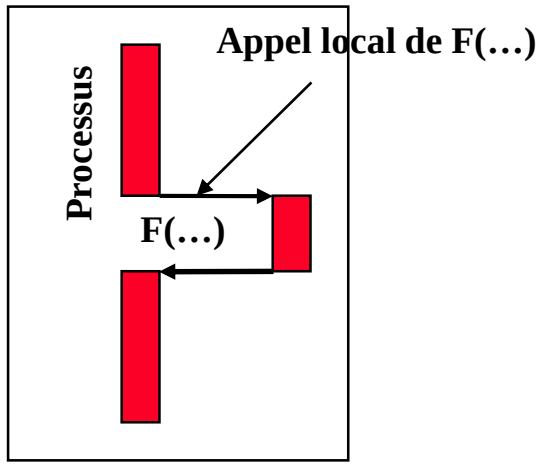
Le concept d'appel de procédure distante ou RPC (Remote Procedure Calls) permet de réaliser des applications client/serveur en gardant le plus possible la sémantique habituelle d'un appel local de fonction

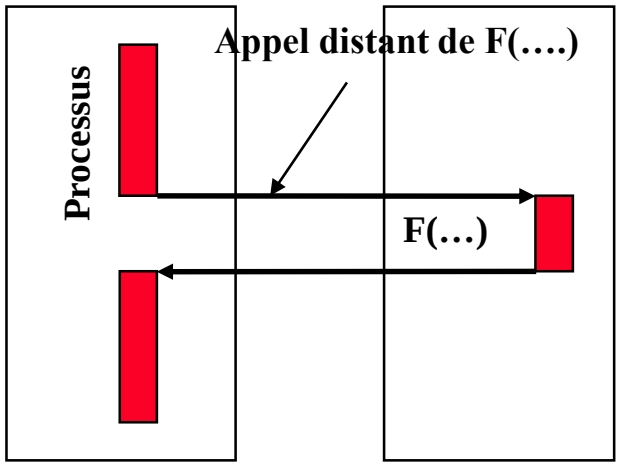
⇒ un programme local à une machine (le client) peut

- exécuter une procédure sur une autre machine distante (le serveur)
- lui passer des données
- et récupérer des résultats.

**Machine A**
**(Client: l'appelant)**

**Machine B**
**(Serveur: l'appelé)**




**Appel local de F(…)**
**Même espace d'adressage**

**Appel distant de F(….) : RPC**
**Espace d'adressage différent**

## Avantages

- Paradigme familier aux programmeurs
  - Formes et effets identiques à l'appel  local
- La sémantique est identique à celle de l'appel local
  - Re-utilisation des programmes existants

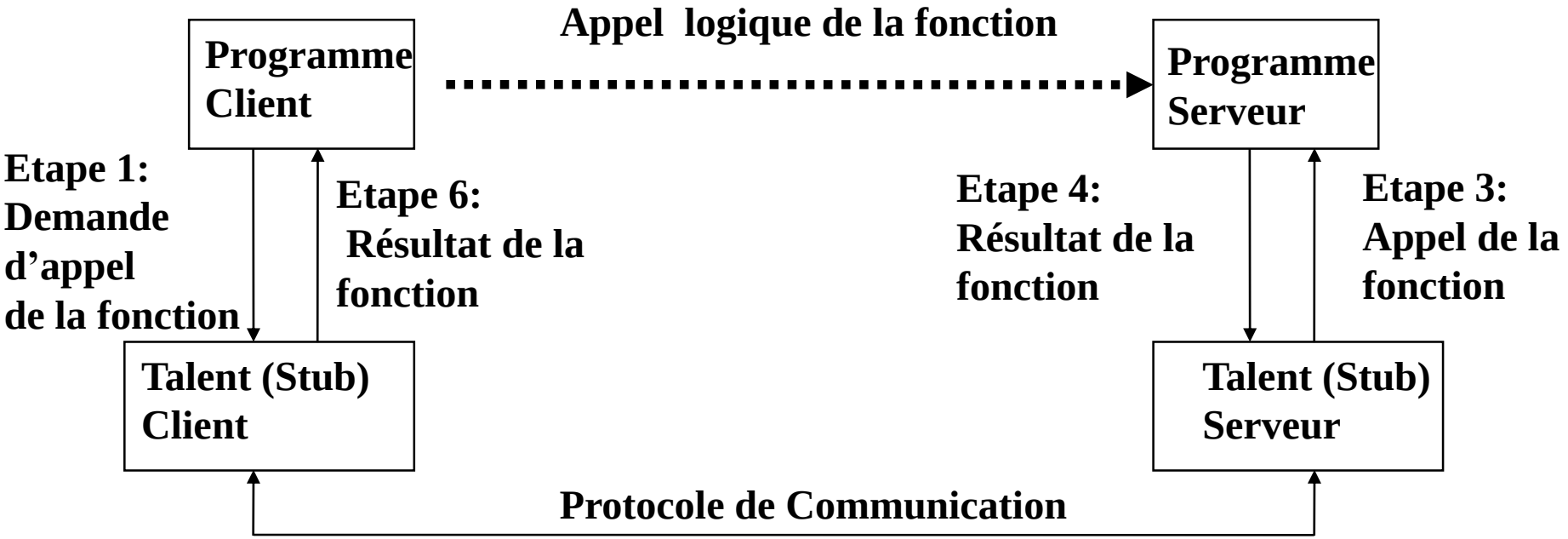
## Difficultés :

- Client et serveur dans deux espaces différents:
  - mode de pannes indépendant,
  - fiabilité du réseau,
  - temps de réponse

- Historique des RPC
  - Introduits en 1984 (par Birell et Nelson)
  - Implémentation par SUN en 1988
  - Utilisés dans le protocole NFS (entre autres)

# Principes de Fonctionnement d’un appel de procédure à distance

- Mode client/serveur
- Le coté client appelle localement la fonction sur un élément spécial (talons ou *stubs*) qui relayera la demande d'appel de fonction coté serveur
- Coté serveur, un élément (talons ou *stubs*) spécial appellera la fonction et renverra le résultat coté client




**Etapes 2 et 5 : Communication via TCP ou UDP**

- Communication via les sockets TCP ou UDP
  - Pour faire transiter les demandes d'appels de fonction et le résultat des appels
  - Toutes les données transitant via les sockets sont codées via XDR

## Talon client

1. Reçoit l'appel local
2. Emballe les paramètres
4. Exécute l'appel distant
5. Place le client en attente
10. Reçoit et déballe les résultats
11. Retourne les résultats au client

## Talon serveur

6. Reçoit le message d'appel
7. Déballe les paramètres
8. Fait exécuter la procédure
9. Emballe et transmet les résultat

## Les RPC sous Linux

- Protocole défini par SUN :
  - Il est à la base de l'implémentation de NFS.
  - Il est Open Source.
  - Il utilise le protocole XDR pour les échange de données (transport des arguments et du résultat).

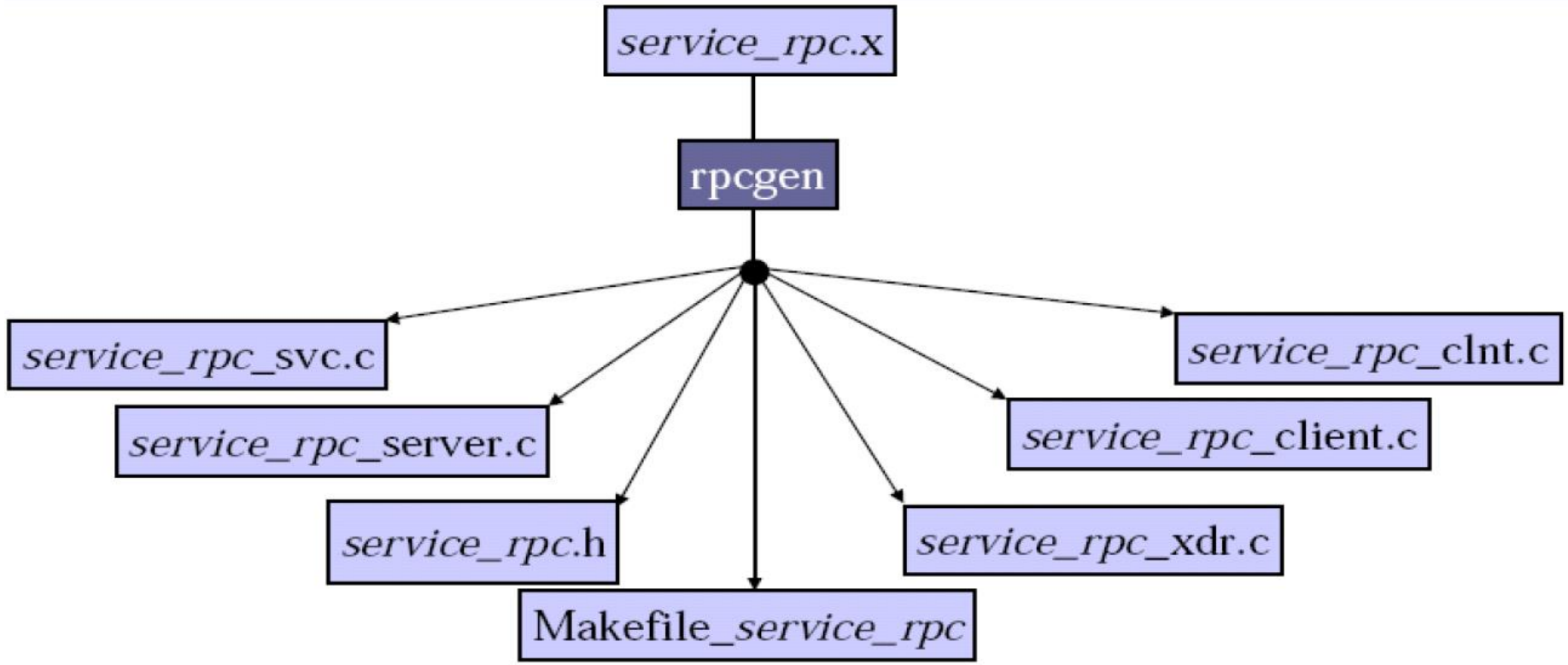
- Fonctionnement :
  - Au niveau serveur :
    - un processus démon attend des connexions (portmaper).
    - il détermine le programme p qui contient la procédure.
    - le programme p décode les paramètres, exécute la procédure et encode le résultat.
    - le démon retourne le résultat.
  - Au niveau client, le talon du client va :
    - déterminer le numéro du programme dans l'intervalle [0x20000000 , 0x3FFFFFFF].
    - déterminer la version du programme à utiliser.

Il existe trois façon de développer des programmes utilisant des RPC :

- Utiliser les fonctions de la couche intermédiaire:
  - Elle comporte peu de fonctions
  - mais le programmeur est limité dans la configuration du système et doit développer l'encodage et le décodage
- Utiliser les fonctions de la couche base :
  - C'est un ensemble complet de fonctions
  - Mais son utilisation est beaucoup plus complexe que la couche intermédiaire
  - Elle possède plus de 20 fonctions
- Utiliser le compilateur rpcgen.
  - Permet de générer automatiquement, le code des talons (stubs), des fonctions XDR associées aux données utilisées par les fonctions et le squelette du serveur et du client

# L’outil RPCGEN

- Principe d'utilisation
  - On décrit dans un fichier (d'extension .x)
    - Les structures de données propres aux fonctions
    - Les fonctions appelables à distance
  - RPCGEN génère ensuite un ensemble de fichiers



# Exemple de fonctionnement

A l'aide de l'exemple suivant, nous illustrons le fonctionnement des RPC. Considérons deux fonctions simples et appelables à distance à savoir.

```
int puiss2(n);
// fonction qui calcule 2 puissance n
float multxy(float x, float y);
// fonction qui calcule x*y
```

Il y a deux programmes

- Un programme client qui appelle les fonctions
- Un programme serveur qui implémente les deux fonctions

# Interface:

Le travail commence par la définition de l'interface, celle-ci étant écrite en utilisant l'IDL (Interface Definition Language) du système RPC. Cette définition est enregistrée dans un fichier d'extension .x, par exemple calcule.x . Dans ce fichier:

- On décrit **les données**
  - les structures utilisées
- On décrit **les fonctions** appelables.
  - Les Règles fondamentales: une fonction ne prend qu'un seul paramètre

  ⇒ on doit définir une structure de données dédiée à la fonction si on veut passer plusieurs valeurs en paramètre.

– On définit un « programme » RPC
  - programme = ensemble de fonctions/services,
  - chaque programme possède un nom avec un numéro associé,
  - un programme peut exister en plusieurs versions
    – chaque version est identifiée par un nom et un numéro associé,
    – chaque version définit une liste de fonctions,
    – chaque fonction est identifiée par un numéro unique

## Exemple Fichier calcule.x

```
struct operandes {
    float x;
    float y;
};
program MATHS_PROG {
     version MATHS_VERSION_1 {
          int PUISS2(int) = 1;
          float MULTXY(operandes) = 2;
     } = 1;
} = 0x20000002;
```

- Nom du programme : MATHS_PROG. L'identifiant du programme en hexadécimal est 0x20000002 = 536870914. Il est choisit dans l'intervalle  réservé pour les utilisateurs à savoir [0x20000000 ,  0x3FFFFFFF].

- Nom de version : MATHS_VERSION_1 de numéro 1
- La liste des procédures que le serveur implémente.
  - Les 2 fonctions PUISS2() et MULTXY() sont numérotées respectivement 1 et  2
  - La structure « operandes » est créée pour la fonction multxy qui nécessite 2 paramètres.
  - Par convention, on écrit les noms de fonctions, programme et versions en majuscule.

# Génération des fichiers

le fichier de définition .x est ensuite traité par l'utilitaire rpcgen pour générer les fichiers

- $ rpcgen calcule.x

- Fichiers générés à partir du .x
  - calcule.h
    - Définition de toutes les structures et des signatures des fonctions
  - calcule_xdr.c
    - Fonctions de codage XDR des structures de données
  - calcule_clnt.c
    - Talon client (stub client)
  - calcule_svc.c
    - Talon serveur (stub serveur)

- Avec l’option -a de rpcgen
  - on génère le fichier calcule_server.c
    - Squelette pour écriture des fonctions du programme
  - on génère le fichier calcule_client.c
    - un squelette pour le programme client

- Le format XDR (eXternal Data Representation) définit les types utilisés pour l'échange de variables entre le client et le serveur. Ce format définit un codage pour les entiers, les flottants ... les structures.

## Coté serveur: Implémentation des fonctions

- Dans le squelette calcule_server.c, généré avec l'option –a , on implémente le code de chacune des fonctions du programme (puiss2() et multxy()) en prenant la signature définie dans le fichier calcule.h
- Dans le fichier .h, les fonctions à implémenter coté serveur sont suffixées par la version et par svc.
- **Exemple**
  - Dans calcule.h

```
extern  int * puiss2_1_svc(int *, struct svc_req *);
extern  float * multxy_1_svc(operandes *, struct svc_req *);
```

# Notes

- Les noms des fonctions dans le fichier .h sont suffixées par la version et par svc par rapport au fichier .x
- Les fonctions définies dans le fichier .h ont un deuxième argument (paramètre svc_req* ) qui n'a pas d'utilité dans notre exemple.
- Un niveau de pointeur en paramètre et retour est ajouté par rapport au .x
  - On doit passer par des pointeurs pour les paramètres.
  - Pour le retour, c'est un pointeur qui est renvoyé. Cette donnée sera retournée par copie au client.
- Notre seul travail consiste à implémenter ces fonctions dans fichier calcule_server.c.

Coté serveur: fichier calcule_server.c.

```
#include "calcule.h"
 int *puiss2_1_svc(int *argp, struct svc_req *rqstp){
      static int  result;
     /*     * insert server code here   */
     int i;
    result=2;
     for (i=1;i<*argp;i++)
          result=result*2;
     return &result;
}
float *multxy_1_svc(operandes *argp, struct svc_req *rqstp){
     static float  result;
    /*       * insert server code here       */
     result=argp->x * argp->y;
     return &result;
}
```

## Remarques

- Il n'y a pas de fonction main(). La fonction main est située dans le stub serveur calcule_svc.c.

- Les variables dans lesquelles on met les valeurs de retour sont déclarées 'static‘
  - En effet, puisqu’on retourne un pointeur sur une donnée qui doit exister après l'appel de la fonction, la variable dans laquelle on met la valeur de retour doit être déclarée 'static'

# Coté client: appel des fonctions

- Dans le squelette calcule_client.c, généré avec l'option –a, ou dans un autre fichier, on appelle les fonctions en prenant la signature définie dans le fichier calcule.h
- Dans le fichier .h, les fonctions à appeler coté client sont suffixées par la version. Ces fonctions sont appelée sur le talon client qui relaie la requête coté serveur sur lequel on appellera les fonctions suffixées par la version et par svc

```
extern  int * puiss2_1(int *, CLIENT *);
extern  float * multxy_1(operandes *, CLIENT*);
```

- Paramètre CLIENT*
  - Caractéristiques de la communication avec la partie serveur

- Avant de pouvoir appeler une fonction sur une machine distance, on doit identifier le programme RPC tournant sur cette machine, en appelant la fonction suivante:

```
CLIENT *clnt_create(
    char *machine,              // nom de la machine serveur
    long numero_programme,      // identifiant  du programme RPC
    long numero_version,        // Numéro de la version du programme
    char *protocole);           // « udp » ou « tcp »
```

- Pour identificateurs programme et version : peut utiliser les noms associés se trouvant dans le .h

- Retourne un identificateur de communication coté client à utiliser pour appel des fonctions ou NULL si problème

# Coté client : fichier client_calcule.c

```
#include "calcule.h"
int main() {
     CLIENT *client;
     operandes param;
     float *resultat;
     int n,*p;
    client = clnt_create("PGR", MATHS_PROG, MATHS_VERSION_1, "udp");
    if (client == NULL) {
        perror(" erreur creation client\n");
        exit(1);
    }
    param.x=12;  param.y=20;
    resultat = multxy_1(&param, client);
    printf(" resultat =   %f \n", *resultat);
    n=16;
    p=puiss2_1(&n,client);
    printf(" 2 puissance 16 =   %d \n", *p);
}
```

- Commentaires
  - Pour la connexion avec le programme RPC distant
    - Le programme distant s'exécute sur la machine PGR
    - On utilise les constantes définies dans cacule.h pour identifier le programme et sa version
    - On choisit une communication en UDP
  - Aucune différence avec l’appel d'une fonction local

## Compilation

- Pour les 2 parties, on a besoin des fonctions de codage XDR
  - $ gcc -c calcule_xdr.c  // calcule_xdr.o
- **Coté client**
  - $ gcc -c calcule_clnt.c // génère calcule_clnt.o
  - $ gcc -c client_calcule.c  // client_calcule.c
  - $ gcc -o client_calcule client_calcule.o  calcule_clnt.o calcule_xdr.o
- **Coté serveur**
  - $ gcc -c calcule_svc.c  // calcule_svc.o

    /* Le talon coté serveur (calcule_svc.c) contient un main() qui enregistre automatiquement le programme comme service RPC */
  - $ gcc -c calcule_server.c // calcule_serveur.o
  - $ gcc -o serveur calcule_svc.o calcule_server.o calcule_xdr.o

- On démarre le serveur sur la machine serveur (RPG dans notre exemple)

  $ ./serveur &

- Ensuite, on peut utiliser la commande rpcinfo pour vérifier qu'il tourne.

- L'option -p  (rpcinfo –p) permet de connaître la liste des programmes RPC actuellement enregistrés sur la machine.

- On lance le client sur la machine cliente

  $$ ./client_calcule

# *Chapitre 4* Sockets Java

## Principe:

- Le serveur crée une socket serveur (socket d'écoute) et se met en attente
  - la classe ServerSocket est utilisée par un serveur pour écouter les connexions entrantes.
- Le client crée une socket client et se connecte à la socket seveur
  - la classe Socket est utilisée par le client pour initialiser une connexion.

Lorsqu'un client réalise une connexion socket, alors

- Une socket client "coté client" est créée
- Une socket service "coté serveur" est créée

La communication effective sur une connexion par socket utilise la notion de flots (flux) de données (java.io.OutputStream et java.io.InputStream).

- Sockets TCP
  - Classe SocketServer
    - Attachement à un port
    - Acceptation d’une demande de connexion à un port local
    - Attente de demandes de connexion
  - Classe Socket
    - Connexion à une machine distante
    - Envoi/Réception de données
    - Fermeture d’une connexion

# Les classes

Plusieurs classes interviennent lors de la réalisation d'une communication par sockets.

- La classe  java.net.InetAddress permet de manipuler des adresses IP.
- La classe java.net.SocketServer  permet de programmer l'interface côté serveur en mode connecté.
- La classe java.net.Socket  permet de programmer l'interface côté client et la communication effective par flot via les sockets.
- Les classes java.net.DatagramSocket et java.net.DatagramPacket permettent de programmer  la communication en mode datagramme.

## La classe java.net.InetAddress

- Cette classe permet de manipuler des adresses IP.

- Champs
  - hostName (exemple : ensa.ump.ma)
  - address (32 bits, exemple : 192.168.50.123)
- Pas de constructeur
- 3 méthodes statiques

  Conversion de nom vers adresse IP : Un premier ensemble de méthodes permet de créer des objets adresses IP d'un site

- public static InetAddress getLocalHost() throws UnknownHostException

  Cette méthode retourne un objet contenant l’adresse IP de la machine locale d'appel.

- public static InetAddress getByName(String host) throws UnknownHostException

  Cette méthode rtourne un nouvel objet InetAddress contenant l'adressse Internet de la machine dont le nom est passé en paramètre. Le nom du site est donné sous forme symbolique (ensa.ump.ma) ou sous forme numérique  (192.168.50.123).

- public static InetAddress[] getAllByName(String host) throws UnknownHostException

  Cette méthode permet d'obtenir les différentes adresses IP d'un site.

Les méthodes suivantes sont applicable à un objet de la classe InetAddress. Elles permettent d'obtenir dans divers formats des adresses IP ou des noms de site.

- public String getHostName()

  Retourne le nom de la machine dont l'adresse est stockée dans l'objet.

- public String getHostAddress()

  obtient l'adresse IP sous forme %d.%d.%d.%d

- public byte[] getAddress()

  obtient l'adresse IP sous forme d'un tableau de 4 octets dans l'ordre réseau.

- La classe  java.net.ServerSocket

Cette classe implante un objet ayant un comportement de serveur via une interface par socket.

public class java.net.ServerSocket

- Gestion d'une connexion
  - Attachement à un port : Création (constructeur ServerSocket()) d'un nouvel objet ServerSocket affecté à un port.
  - Attente et acceptation (accept())d'une demande de connexion à un port local de connextion :
  - Echange d'informations : getInputStream() et getOutputStream()
  - Clôture de la connexion par le client ou le serveur : close()
  - Nouvelle attente

- Les constructeurs ServerSocket

- public ServerSocket(int port) throws IOException
  - Ce constructeur crée un objet serveur à l'écoute du port spécifié

- public ServerSocket(int port, int taillefile) throws IOException
  - Crée un objet serveur à l'écoute du port spécifié. le paramètre taillefile. Spécifie la taille de la file d'attente des demandes de connexion.

- public ServerSocket(int port, int taillefile, InetAddress bindAddr) throws IOException
  - Si la machine possède plusieurs adresses, on peut spécifier l'adresse sur laquelle on accepte les connexions.

- Ces constructeurs correspondent à l'utilisation des primitives socket(), bind() et listen().

- Les méthodes de la classe ServerSocket
  - ✓La méthode accept() pour la prise en compte d'une connxion du client
    - public Socket accept() throws IOException
  - ✓La fermeture de la socket d'écoute s'exécute par l'appel de la méthode close().
    - public void *close()* throws IOException
  - ✓Enfin, les méthodes suivantes retrouvent l'adresse IP ou le port d'un socket d'écoute :
    - public InetAddress getInetAddress()
    - public int getLocalPort()

## Remarques:

- La méthode accept() est bloquante.
- Le temps d’attente peut être limitée par l'appel de la méthode setSoTimeout().

  - public void setSoTimeout(int timeout) throws SocketException

  timeout: délai d’atente exprimé en  millisecondes. À l'expiration du délai d’attente, l'exception java.io.InterruptedIOException est levée.

  - La valeur par défaut 0 équivaut à l'infini.

  - La méthode doit être initialisé avant accept()

- La classe java.net.Socket.

  La classe java.net.Socket est utilisée pour la programmation des sockets connectés, côté client et côté serveur.

  - public class java.net.Socket

  – Connexion à une machine distante:
    - Côté serveur, la méthode accept() de la classe java.net.ServerSocket renvoie une socket de service connecté au client.
    - Côté client, on utilise on utilise les constructeurs socket()
    - Envoi/Réception de données
    - Fermeture d’une connexion

- Constructeurs

  - public Socket(String host, int port) throws UnknownHostException, IOException
  - public Socket(InetAddress address, int port) throws IOException

  Ces deux constructeurs créent une socket connectée à la machine et au port spécifiés par les paramètres « host » (ou « adresse ») et « port ».

  - public Socket(String host, int port, InetAddress localAddr, int localPort) throws UnknownHostException, IOException
  - public Socket(InetAddress addr, int port, InetAddress localAddr, int localPort) throws IOException

  Ce deux constructeurs créent une socket connectée à la machine et au port spécifiés par « host » (ou « adresse ») et « port ». Elles permettent en plus de fixer l'adresse IP (« localAddr »)et le numéro de port (« localPort ») utilisés côté client (plutôt que d'utiliser un port disponible quelconque).

Remarque: Ces constructeurs correspondent à l'utilisation des primitives socket(), bind() (éventuellement) et connect().

Un ensemble de méthodes permet d'obtenir les éléments constitutifs de la liaison établie :

- public InetAddress getInetAddress()
  fournit l'adresse IP distante
- public InetAddress getLocalAddress()
  fournit l'adresse IP locale
- public int getPort()
  fournit le port distant
- public int getLocalPort()
  fournit le port local

• L'opération close() ferme la connexion et libère les ressources du système associées au socket.

- La communication effective sur une connexion par socket utilise la notion de flots de données  (java.io.OutputStream et java.io.InputStream).

- Les deux méthodes suivantes sont utilisés pour obtenir les flots en entrée et en sortie.

  - public InputStream getInputStream() throws IOException

  - public OutputStream getOutputStream() throws IOException

# Exemple:

- Le client envoie le message « Bonjour " au serveur.
- Le serveur (machine distante « PGR ») reçoit le message du client via le port 6000 et envoie un accusé de réception au client.

```
// Code du Client
import java.net.*;      import java.io.*;
public class JavaClient {
    public static void main(String[] args) throws IOException {
        InetAddress addr = InetAddress.getByName("PGR");
       Socket socket = new Socket(addr, 6000);
       try {
         BufferedReader in = new BufferedReader( new InputStreamReader( socket.getInputStream()));
         PrintWriter out = new PrintWriter( new BufferedWriter( new OutputStreamWriter (
                                socket.getOutputStream())),true);
         out.println("Bonjour ");
         String str = in.readLine();
         System.out.println(str);
       }  finally {
           socket.close();
       }
    }
}
```

## Commentaires:

- Pour faire des tests sur une seule machine, il est possible d'utiliser:
  InetAddress addr = InetAddress.getByName("127.0.0.1");  ou
  InetAddress addr = InetAddress.getByName("localhost");  ou
  InetAddress addr = InetAddress.getByName(null);
- Tester et commenter
  System.out.println("socket = " + socket);  et  System.out.println("addr = " + addr);
- Pour garantir la fermeture des sockets,  le code doit être inclus dans un bloc try-finally.
- Ouverture d'une Socket et association d'un flux d'entrée  à cette socket
  InputStreamReader  s1 = new (InputStreamReader( socket.getInputStream())
  BufferedReader in  =  new BufferedReader( s1);
  =  new BufferedReader( new InputStreamReader( socket.getInputStream()));
- association d'un flux de sortie  à la socket
  PrintWriter out = new PrintWriter( new BufferedWriter(
  new OutputStreamWriter( socket.getOutputStream())),true);
  Le tampon de sortie est automatiquement vidé par PrintWriter:

```
// code du serveur
import java.io.*;     import java.net.*;
public class JavaServeur {
     public static final int PORT = 6000;
     public static void main(String[] args) throws IOException {
      ServerSocket s = new ServerSocket(PORT);
      try {  // Le programme s'arrête  ici et attend une demande de connexion:
          Socket socket = s.accept();
          try {
               BufferedReader in = new BufferedReader( new InputStreamReader(socket.getInputStream()));
                PrintWriter out = new PrintWriter(new BufferedWriter(
                                        new OutputStreamWriter(socket.getOutputStream())),true);
               String str = in.readLine();
               System.out.println("Reçu: " + str);
                out.println(" Bien reçu   " + str);
             }  finally {
                  socket.close();   }
       } finally {
             s.close();  }
   }
}
```

## Connexion de plusieurs Clients: Serveur Multithread

Dans l'exemple précédent, le serveur sert un seul client.

Pour serevir plusieurs client (le traitement simultané de plusieurs clients), on utilise en général des serveurs multithread. Dans ce modèle le serveur lance plusieurs threads en parallèle afin de traiter les requêtes des clients simultanément.

On définit une classe auxiliaire JavaUnSeulServeur qui a hérite de Thread et définit la gestion de la requête. Elle a :

- un constructeur qui prend en paramètre la socket et gère la requête
- une méthode run() dans laquelle on définit la gestion de la requête.

```
// Code serveur multithread
import java.io.*;  import java.net.*;
class JavaUnSeulServeur extends Thread {
  public static final int PORT = 6000;
  private Socket socket;
  private BufferedReader in;
  private PrintWriter out;
  public JavaUnSeulServeur (Socket s)  throws IOException {
        socket = s;
         in = new BufferedReader( new InputStreamReader(socket.getInputStream()));
         out = new PrintWriter(new BufferedWriter(
                  new OutputStreamWriter(socket.getOutputStream())),true);
           start();   // lance la méthode run
   }
```

```
    public void run() {
       try {
        System.out.println( "Connection accepted: "+ socket);
              String str = in.readLine();
        System.out.println("Reçu: " + str);
        out.println(" Bien reçu   " + str);
          } catch(IOException e) {
                System.err.println("IO Exception");
           } finally {
                try {
                 socket.close();
            } catch(IOException e) {
                    System.err.println("Socket non fermée ");
                  }
         }
       }
}
```

```
public class JavaServeurMultiThread {
    public static final int PORT = 6000;
    public static void main(String[] args) throws IOException {
        ServerSocket s = new ServerSocket(PORT);
        try {  // Le programme stoppe ici et attend une demande de connexion:
            while (true) {
                Socket socket = s.accept();
            try {
                new JavaUnSeulServeur(socket);   // Lancer un thread pou traiter un client
            } catch(IOException e) {    // En cas d‘échec, fermer l'objet socket,
                                                  // sinon le thread le fermera:
                        socket.close();
                }
            }
        } finally {
                s.close();
    }
}
}
```

# Chapitre 5. JAVA RMI (Remote Method Invocation)

## I. Présentation

RMI (Remote Method Invocation) est un système d'objets répartis destiné au développement d'applications réparties. C'est une API (Application Programming Interface) intégrée depuis JDK 1.1.

- Il rend transparent la mise en oeuvre et l'utilisation d'objets distants Java
- Il est spécifique au langage Java: ne fait communiquer que des éléments écrits en Java, qui s'exécutent éventuellement sur des JVM distinctes,

- Développé par JavaSoft,
- Il assure la communication entre le serveur et le client via TCP/IP de manière transparente pour le développeur, en se basant sur
  - Les mécanisme de sockets ;
  - Le protocole RMP (Remote Method Protocol) pour les échange.

$\Rightarrow$ Le développeur ne voit pas le problèmes de niveaux inférieurs spécifiques aux technologies réseaux.

## II. Principes

- Les méthodes des objets distants peuvent être invoqués depuis des JVM différentes de celle ou se trouve l'objet distant.
- Un objet distant (objet serveur) : ses méthodes sont invoquées depuis une autre JVM
  - dans un processus différent (même machine)
  - ou dans une machine distante (via réseau)
- Un OD (objet distant) sur un serveur est décrit par une interface distante Java
  - déclare les méthodes distantes utilisables par le client
- Les clients (objets clients) des objets distants (objets serveurs) interagissent avec des interfaces distantes, jamais directement avec leurs implémentations.

Même schéma que les RPC

- Le programmeur fournit
  - Description(s) d'interface (en Java)
  - Les objets réalisant les interfaces (l'implémentation de l'interface)
  - Le programme du serveur, instanciant ces objets (l'application serveur)
  - Le programme du client, invoquant ces objets
- L'environnement Java fournit
  - Un générateur de talons nommé rmic
  - Un service de noms (Object Registry)
  - Un middleware pour l'invocation à distance (ensemble de classes)
  - La faculté d'exécuter du code généré ailleurs

# III. Interfaces

- L’interface constitue le cœur des RMI. Elle décrit les caractéristiques de l’objet distant
  - Elle précise les signatures (noms, types de retours, paramètres)  de l’ensemble de méthodes appelables à distance.

  $\Rightarrow$

  - La définition d’un service distant est codé dans une interface Java.
  - L’implémentation est codé dans une classe.

## Interface (suite)

- C’est une interface Java traditionnelle qui:
  - Doit être public, sinon, le client recevra une erreur lorsqu'il tentera d'obtenir un objet distant qui implémente l'interface distante.
  - Doit hériter de l’interface Remote définie dans java.rmi.
    - Le Package java.rmi et sous-packages contient les classes à utiliser ou spécialisé pour des communications via RMI

- Un objet distant passé en argument ou en valeur de retour (soit directement ou inclus dans un objet local), doit être déclaré en tant qu'interface distante et non comme la classe d'implémentation.

# Interface (suite)

- Toutes les méthodes de cette interface peuvent déclencher une exception du type RemoteException. Une méthode de cette interface doit préciser dans sa signature qu'elle peut lever l'exception java.rmi.RemoteException.

- Cette exception est levée :
    - si connexion refusée à l'hôte distant
    - ou bien si l'objet n'existe plus,
    - ou encore s'il y a un problème lors de l'assemblage ou le désassemblage.
- Les objets clients doivent traiter des exceptions supplémentaires
  – Toute invocation distante de méthode doit lever ou traiter cette exception

Exemple d’interface:

```
public interface Echo extends java.rmi.Remote {
    public String echo() throws java.rmi.RemoteException;
}
```

Ou tout simplement par

```
import java.rmi.*;
public interface Echo extends Remote {
    public String echo() throws RemoteException;
}
```

La méthode echo() est appelable à distance.

## IV. Les objets

Java distingue deux familles d'objets

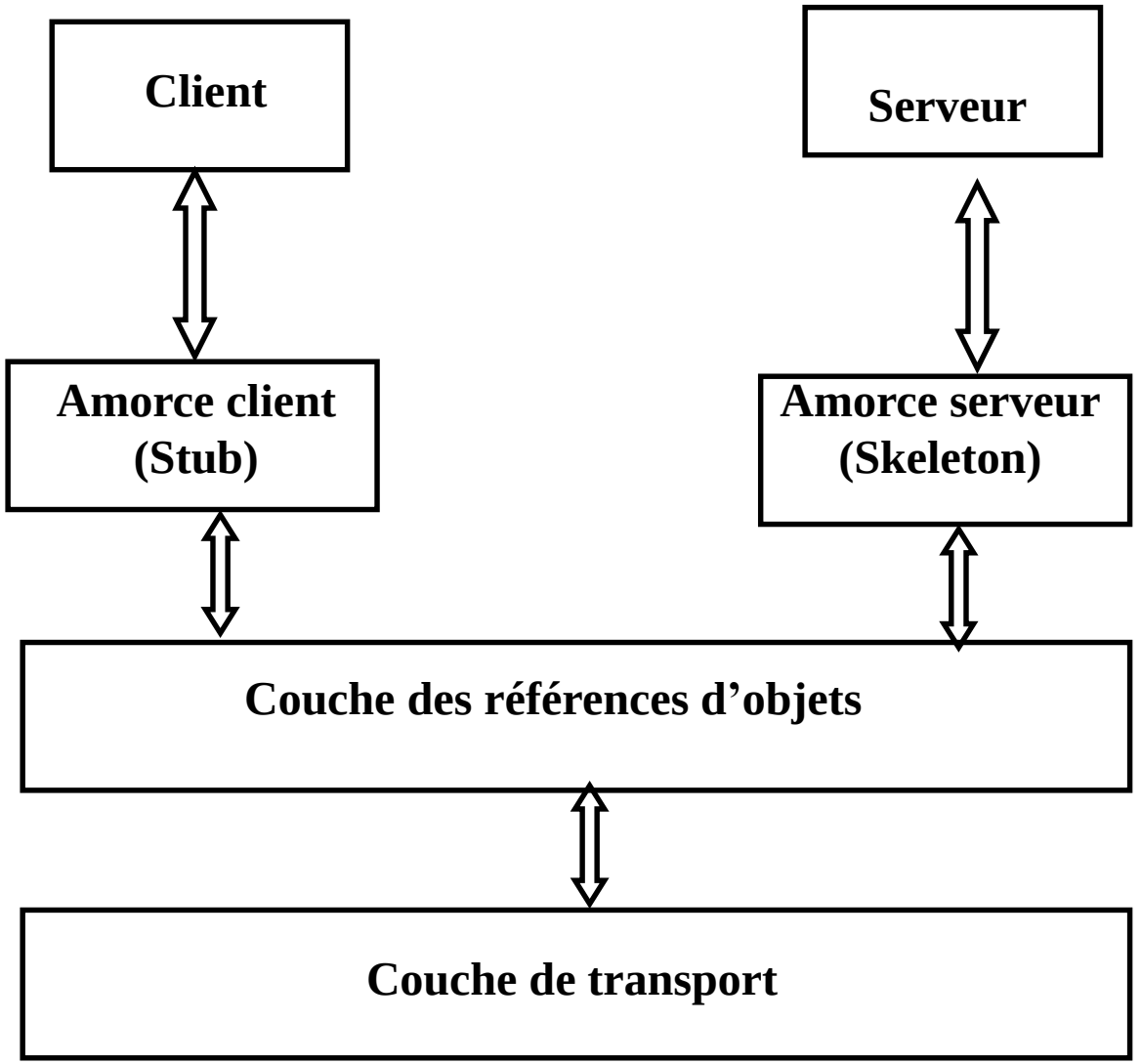
- Les objets locaux
- Les objets distants (OD):
  - ils implantent une interface
  - l'interface étends l'interface marker java.rmi.Remote
- Une référence à un OD peut être passée en argument ou retournée en résultat d’un appel dans toutes les invocations (locales ou distantes)
- Les arguments locaux et les résultats d’une invocation distante sont toujours passés par copie et non par référence
  - leurs classes doivent implémentées
    java.io.Serializable (sérialisation d ’objets)

## V. Architecture RMI

## Les différentes couches de RMI

RMI est essentiellement construit sur trois couches

- Stubs et skeletons
- La couche des références
- La couche transport


Client
Serveur
Amorce client
(Stub)
Amorce serveur
(Skeleton)
Couche des références d’objets
Couche de transport

## 1. Les amorces (Stub/Skeleton)

- Programmes jouant le rôle d'adaptateurs pour le transport des appels distants
  - réalisent les appels sur la couche réseau
  - L'emballage (*marshalling*) / déballage (demarshalling) des paramètres
- A une référence d'OD manipulée par un client correspond une référence d'amorce
- Les amorces sont générées par le compilateur d'amorces : rmic

## 1.1. L’amorce Client (Stub)

- Transmet l’invocation distante à la couche inférieure *Remote Reference Layer*
- Il réalise
  - L’emballage des arguments des méthodes distantes
  - le déballage des résultats
  - Il utilise pour cela la sérialisation des objets
    - il les transforme en un flot de données (flux de pliage) transmissible sur le réseau

## 1.2. L’amorce serveur (Skeleton)

- Réalise le déballage des arguments reçus par le flux d’emballage
- Fait un appel à la méthode de l’objet distant
- Réalise l’emballage de la valeur de retour

## 2. La couche des références distantes

- Permet l'obtention d'une référence d'objet distant à partir de la référence locale au Stub
- Ce service est assuré par le lancement du programme rmiregister
  - à ne lancer qu'une seule fois par JVM, pour tous les objets à distribuer
  - une sorte de service d'annuaire pour les objets distants enregistrés

## 3. La couche de transport

- Réalise les connexions réseau entre les JVM
- Écoute et répond aux invocations
- Construit une table des OD disponibles
- Réalise l’aiguillage des invocations

## VI. Développement d'une application RMI

1. définir une interface Java pour un OD
2. créer et compiler une classe implémentant cette interface
3. créer et compiler une application serveur RMI
4. créer les classes Stub ( _Stub) et Skeleton ( _Skel) à l'aide rmic (Skel n'existe pas pour les versions >1.2)
5. démarrer rmiregister et lancer l'application serveur RMI
6. créer, compiler et lancer un programme client accédant à des OD du serveur

## Mode opératoire coté serveur

1. L'objet serveur s'enregistre auprès du Naming de sa JVM (méthode *rebind*)
2 - L’objet skeleton est créé, celui-ci crée le port de communication et maintient une référence vers l'objet serveur
3 - Le Naming enregistre l'objet serveur, et le port de communication utilisé auprès du serveur de noms

L'objet serveur est prêt à répondre à des requêtes

## Mode opératoire coté client

4 - L'objet client fait appel au Naming pour localiser l'objet serveur (méthode lookup)

5 - Le Naming récupère les "références" vers l'objet serveur, ...

6 - crée l’objet Stub

7 - rend sa référence au client

8 - Le client effectue l'appel au serveur par appel à l’objet Stub

## *Registry*

- Une partie client doit pouvoir récupérer une référence sur l'objet distant implémentant l'interface
  - Utilise les services du « registry » (registre, annuaire ...)
  - Le registry est lancé à part à coté des applis Java
  - Un objet accède au registry via la classe Naming
- Identification d'un objet distant
  - Via une URL de la forme rmi://hote:port/nomObj

  hote : nom de la machine distante (sur laquelle tourne un registry)

  port : port sur lequel écoute le registry

  nomObj : nom donné à un objet offrant des opérations
- Si pas précision du port : port par défaut utilisé par le registry
- Si pas précision de l'hôte : registry local par défaut (localhost)

## Lancement du registry

- À partir de la ligne de commande
  - Unix/Linux : rmiregistry [port]
  - Windows : start rmiregistry [port]
  - En paramètre, on peut préciser un numéro de port : port d'écoute du registry
    - par défaut  1099
  - Peut utiliser un ou plusieurs registry par application distribuée
  - On doit enregistrer un objet sur un registry local
- À partir d'un objet java
  - un objet Java peut à l'exécution lancer un registry
  - Opération de java.rmi.registry.LocateRegistry
    - public static Registry createRegistry(int port) throws RemoteException
  - Lance un registry localement sur le port d'écoute passé en paramètre

# VII. Un exemple

- Un « echo » distribué :
  - invocation distante de la méthode echo() d'un objet distribué avec le mécanisme RMI.
    - 3 programmes coté serveur sont nécessaires :
      - Echo : l'interface décrivant l'objet distant (OD)
      - EchoImpl : l'implémentation de l'OD
      - EchoAppliServer : une application serveur RMI

# 1. Définition de l'interface pour la classe distante

- Les classes placées à distance sont spécifiées par des interfaces qui doivent dériver java.rmi.Remote et dont les méthodes lèvent une java.rmi.RemoteException

```
// fichier Echo.java
import java.rmi.*;
public interface Echo extends Remote {
 public String echo() throws RemoteException;
}
```

## 2. Implémentation de l’OD

- Maintenant, il faut implémenter l’interface distante dans une classe. Par convention le nom de cette classe aura pour suffixe Impl.

- Cette classe doit hériter de java.rmi.server.UnicastRemoteObject qui permet de faire la liaison avec le système RMI.
  - Remarque: Il faut définir explicitement le constructeur de cet objet distant. On doit l’écrire parce qu'il doit émettre RemoteException.

- Dans cette classe, on définie les méthodes distantes que pourrons utiliser les clients.  Elle peut aussi contenir d’autre méthodes non appelables à distance. Dans notre exemple, c’est dans cette classe que nous définissons la méthode echo() qui ne fait que retourner une chaîne de caractère.

```
// fichier EchoImpl.java

 import java.rmi.*;
 import java.rmi.server.*;
 public class EchoImpl extends UnicastRemoteObject implements Echo
    {
     public EchoImpl() throws RemoteException { }
     public String echo() throws RemoteException {
          return "Echo : Bonjour du serveur " ;
     }
}
```

## 3. Définir une application serveur

- Ce programme (Le serveur) doit enregistrer auprès du registre RMI l'objet local qui peut être exécuté à distance grâce aux méthodes bind() ou rebind() de la classe Naming.

Naming.bind() ou Naming.rebind().

- Ces méthodes permet d'associer (enregistrer) l'objet local avec un synonyme dans le registre. L'objet devient disponible pour le client.
- L'objet distant sera accessible par les autres machines en indiquant son nom et la machine sur laquelle est exécuté ce serveur par l'url :

  String url = "rmi://nomServeurRMI:port/nomSynonyme"

  En cas de machine locale on utilise:

  String url = "rmi://localhost /echo", Par défaut : port= 1099

  Si localhost ne marche pas, il faut donner le nom de la machine locale.
- Cet " url RMI " sera utilisé par les clients pour interroger le serveur grâce à l'appel : Naming.lookup(url);

Le programme suivant va créer un objet de type EchoImpl. Ensuite cette objet sera enregistré auprès du Registre RMI sous le nom « echo », grâce à la méthode rebind() de la classe Naming

```
//fichier EchoAppliServer.java:
public class EchoAppliServer {
    public static void main(String args[])  throws RemoteException {
     try {// Création de l'OD
          EchoImpl od = new EchoImpl();
          // Enregistrement de l'OD dans RMI
          String url = "rmi://localhost/echo",
          Naming.rebind(url, od);
     }
     catch(Exception e) {
               e.printStackTrace()
          }
    }
}
```

## 4. Compiler les classes

> javac Echo.java

> javac EchoImpl.java

> javac EchoAppliServer.java

Ces commandes créent les fichiers .class

# 5. Créer les amorces

- Il s’agit d’invoquer le compilateur d’amorces rmic sur la classe compilée de l’OD :

➢ rmic EchoImpl

- 2 classes sont alors créees représentant la souche (ou Stub) et le Skeleton :

EchoImpl_Stub.class

EchoImpl_Skel.class (il n’existe plus à partir de JDK 1.2)

## 6. Lancer rmiregister et l’application serveur

- Il faut d ’abord lancer le rmiregister puis le serveur:

  > rmiregister &

  > EchoAppliServer &
- Ce qui rend maintenant le serveur disponible pour de futurs clients ...

## 7. Définir l’application cliente utilisant l’OD

La méthode lookup() est utilisée par les clients pour obtenir une référence à un objet distant. L’objet retourné est de type Remote.

```
// fichier EchoClient.java
   import java.rmi.*;
   public class EchoClient {
   public static void main(String args[]) throws RemoteException {
         // Recherche de l’OD
          try {
               String url = "rmi://localhost/echo";
               Echo od = (Echo)Naming.lookup(url);
               System.out.println(od.echo());
          }
          catch(Exception e) {
              e.printStackTrace(); }
    }
}
```

## 8. Compiler et lancer l’application cliente

Sur une machine cliente

```
$ javac  EchoClient.java
$  java EchoClient
```

Résultat

```
$  Echo : Bonjour du serveur
```

# VIII. Un autre exemple

- Définition de l'interface pour la classe distante

```
// déplacement d'un point dans le plan
// Fichier IPoint.java
import java.rmi.*;
public interface IPoint extends Remote {
      public Point deplacePoint(Point p,int x, int y)
                  throws RemoteException;
}

// déplacement d'un point dans le plan
```

# 2. Définition de la classe Point

Les classes des données en paramètres ou retour des opérations doivent implémenter Serializable

```
import java.io.*;
public class Point implements Serializable {
   public int x, y;
   public Point(int x, int y){
        this.x = x;
         this.y = y;
    }
    public String toString() {
          return "("+x+","+y+")";
    }
}
```

# 3. Implémentation de l'Interface

```
// fichier IPointImpl.java
 import java.rmi.*;
 import java.rmi.server.*;
 public class IpointImpl extends UnicastRemoteObject implements IPoint {
     public IpointImpl() throws RemoteException { }
     public Point deplacePoint(Point p, int x, int y) throws RemoteException {
             return new Point(p.x + x, p.y + y);
    }
 }
```

# 4. Application serveur

```
//fichier PointAppliServer.java:
import java.rmi.*;
public class PointAppliServer {
    public static void main(String args[]) throws RemoteException {
        try {// Création de l'OD
            IpointImpl od = new IpointImpl();
                // Enregistrement de l'OD dans RMI
            String url = "rmi://localhost/MonPoint";
            Naming.rebind(url, od);
        }
        catch(Exception e) {   e.printStackTrace(); }
    }
}
```

## 5. Côté Client

```
Import java.rmi.*;
public class PointClient {
    public static void main(String args[]) throws RemoteException {
// Recherche de l'OD
        Point p1,p2;
          p1 = new Point(2,4);
        try {
             String url = "rmi://localhost/MonPoint";
             IPoint od = (IPoint)Naming.lookup(url);
             p2=od.deplacePoint(p1,2,2);
             System.out.println(p2);
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Pratiquement: Il y a 2 modes pour récupérer la classe du stub de l'objet distant

- Localement, la classe se retrouve via le CLASSPATH
- A distance, en téléchargeant la classe à partir d'un emplacement précisé (codebase)

Coté Serveur

- Lorsque l'appel à *bind* ou *rebind* est effectué
  - Le registre associe l'objet à un nom
  - Le codebase est stocké dans le registre

Coté Client

- Lorsqu'un client demande une référence à un objet distant
  - Il recherche le *stub* dans le chemin d'accès aux classes locales,
  - Si échec, la recherche se fait dans le codebase

# IX. Gestion de la sécurité

- RMI n’autorise pas le téléchargement dynamique de classes si l’application cliente n’utilise pas de *Security Manager* pour les vérifier.
  - Dans ce cas, seules les classes situées dans le CLASSPATH peuvent être récupérées.
- Avec RMI, on utilise RMISecurityManager avec en plus un fichier décrivant les permissions par exemple connect sur les sockets:
- Exemple, en donnant toutes les permissions
  - A éviter en pratique si on veut bien gérer l'aspect sécurité

```
// Fichier de nom Permissions.policy
grant {
    permission java.security.AllPermission;
};
```

Voir les documentations nécessaire pour approfondir ce sujet

- Dynamique coté client
  - Le serveur est lancé en précisant un codebase
  - Le client est lancé sans codebase et retrouvera les classes via les informations envoyées par le registre
- Dynamique coté serveur
  - Le client est lancé en précisant un codebase
  - Le serveur, en cas d'appel en retour par exemple, récupère les stubs coté client via ce codebase

## Lancement du serveur

$ java -Djava.security.policy=client.policy -Djava.rmi.server.codebase=
http://wwwensa.ump.ma/~maoudi/exemple/ EchoApplicationServeur

## Coté Client:

Sur une autre machine

$ java -Djava.security.policy=client.policy -Djava.rmi.server.codebase=
http://wwwensa.ump.ma/~maoudi/java/ EchoClient