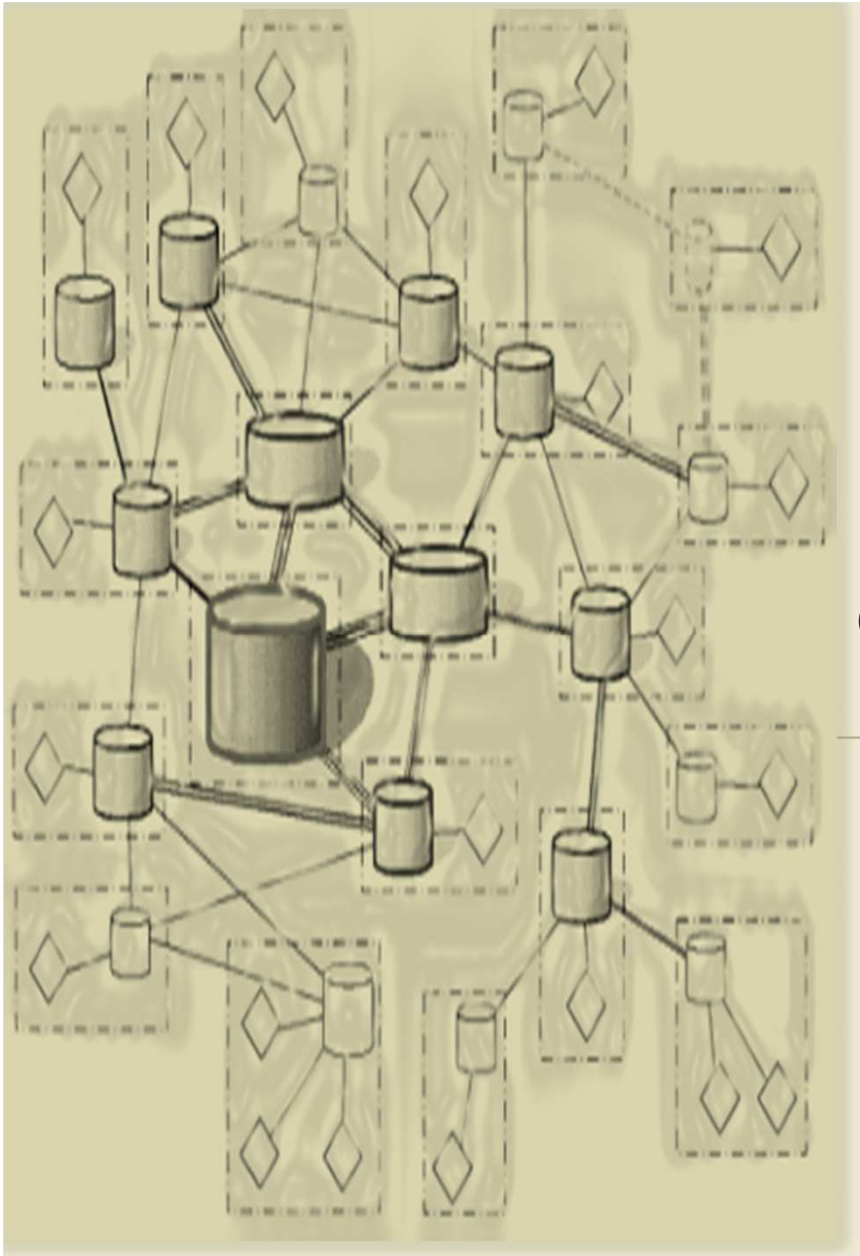


DÉVELOPPEMENT D'APPLICATIONS RÉPARTIES

Chapitre 2: Client Serveur et RPC

©Amen Ben Hadj Ali

amenbha@hotmail.com

ISI-L3SIL

2011-2012

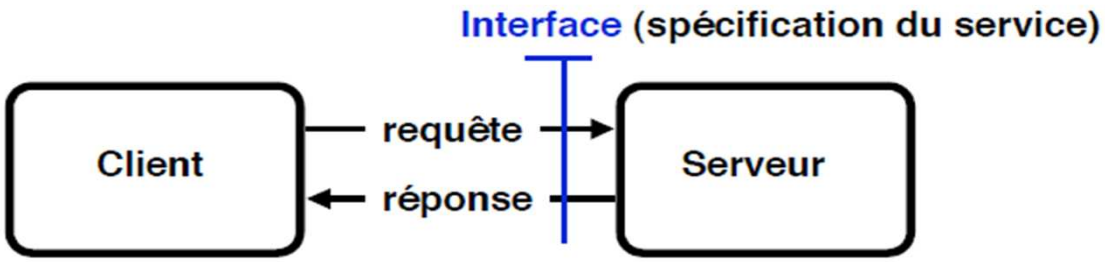
Chapitre 2- le modèle client Serveur

Plan

1. Présentation du modèle CS
2. RPC
3. Mise en œuvre de RPC

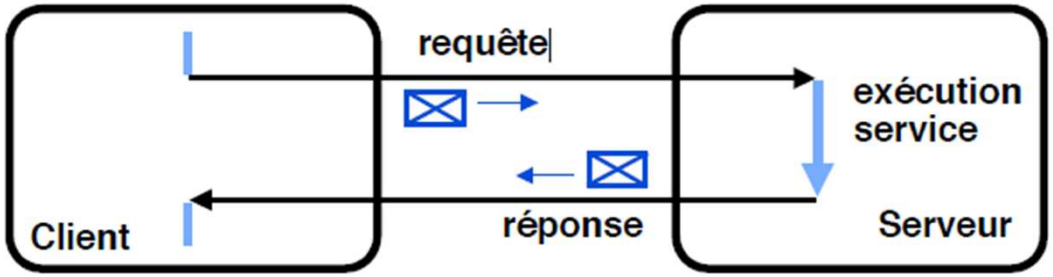
Le modèle Client-Serveur : principe

- Le client demande l'exécution d'un service
- Le serveur réalise le service
- Client et serveur sont (en général, pas nécessairement) localisés sur deux machines distinctes
- Indépendance interface-réalisation



Le modèle Client-Serveur : fonctionnement

- Le dialogue entre le client et le serveur s'effectue par un échange de messages (plutôt que par partage de données, mémoire ou fichiers) transitant à travers le réseau reliant les deux machines
- Requête : paramètres d'appel, spécification du service requis
- Réponse : résultats, indicateur éventuel d'exécution ou d'erreur
- Communication synchrone (dans le modèle de base) : le client est bloqué en attente de la réponse



Le modèle Client-Serveur : avantages

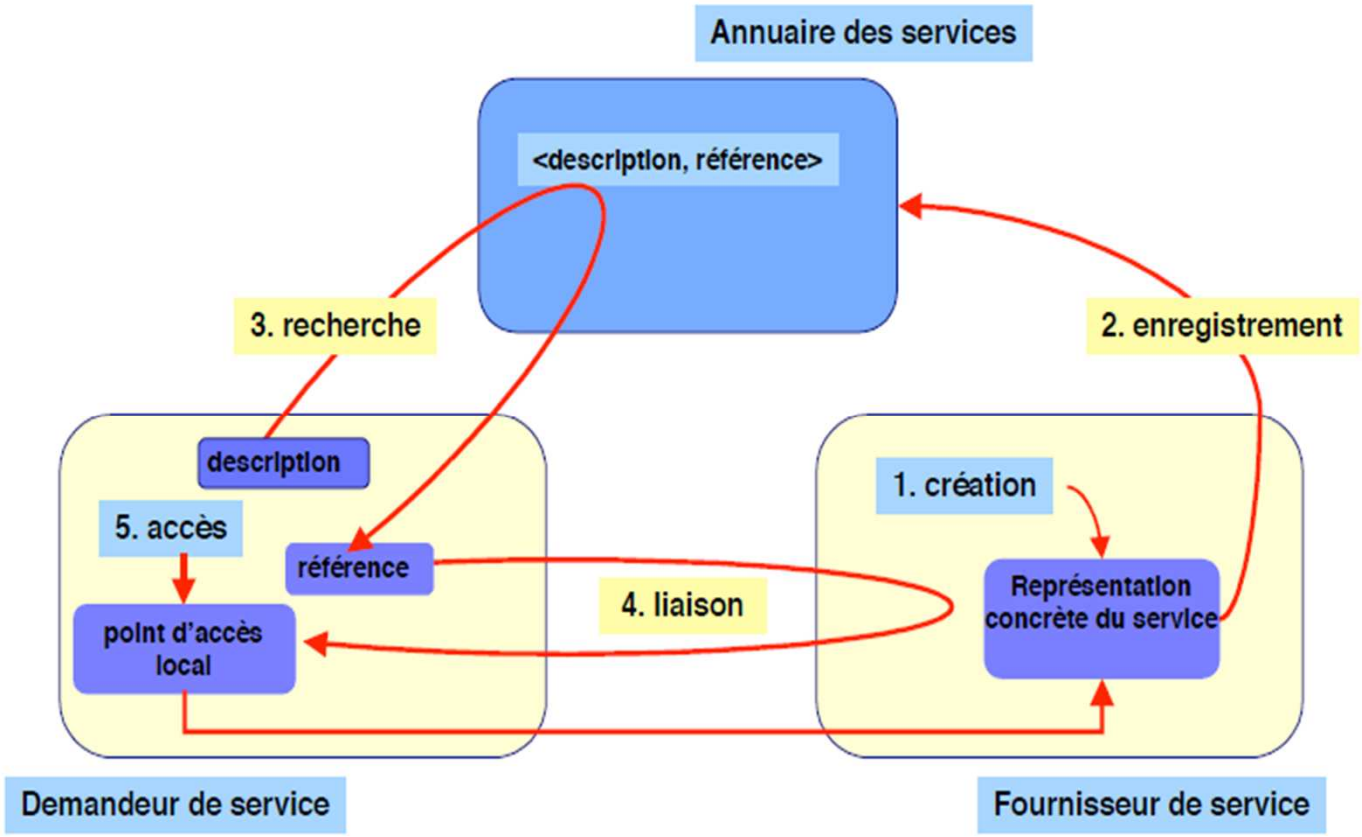
- Structuration
  - fonctions bien identifiées
  - séparation interface du service - réalisation
  - client et serveur peuvent être modifiés (remplacés) indépendamment
- Protection
  - client et serveur s'exécutent dans des domaines de protection différents
- Gestion des ressources
  - le serveur peut être partagé entre de nombreux clients
  - En contrepartie, il doit assurer la gestion des ressources partagées

ces considérations sont indépendantes de la répartition

Le modèle Client-Serveur : exemples

- Serveur de fichiers (NFS)
- Serveur d'impression
- Serveur de calcul
- Serveur d'application (spécifique à l'application)
- Serveur de bases de données
- Serveur de temps
- Serveur de noms (annuaire des services)

Le modèle Client-Serveur : Accès à un service



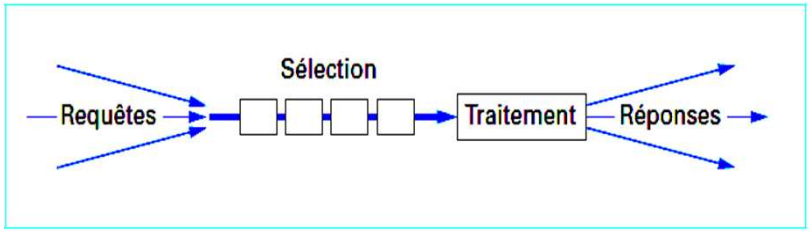
Le modèle Client-Serveur : Mise en oeuvre

- Par des opérations de “bas niveau”
  - Utilisation de primitives du système de communication
  - Exemple : sockets
    - Mode non connecté (UDP)
    - Mode connecté (TCP)
- Par des opérations de “haut niveau”
  - Utilisation d’un middleware spécialisé
  - Contexte : langage de programmation
    - Appel de procédure à distance
  - Contexte : objets répartis
    - Appel de méthodes, création d’objets à distance

Principe de réalisation d'un serveur

La structure du serveur peut prendre plusieurs formes. Le schéma classique d'organisation du serveur est celui d'un processus cyclique qui a trois tâches essentielles

- recevoir, trier et conserver les requêtes avant leur exécution ;
- extraire une requête et exécuter le service demandé ;
- envoyer la réponse au client.

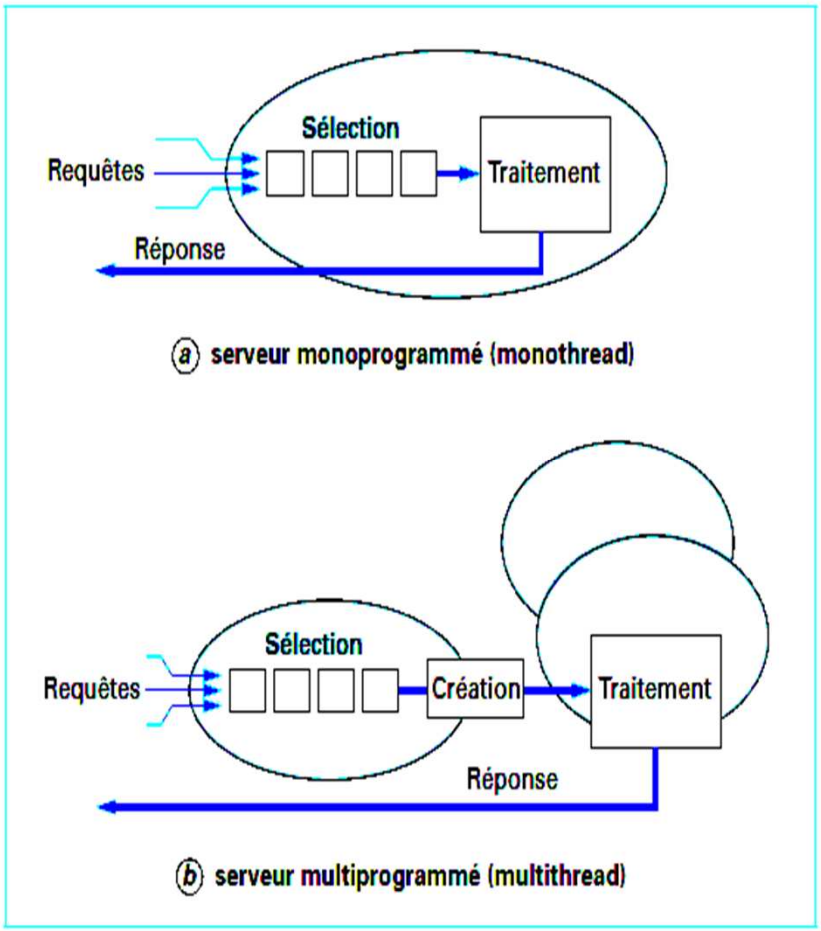



```
while true do
   begin
      Receive (client, message);
      Extract (message, service_id, <params>);
         Case service_id of
         service_1 : begin
            do_service [service_1]
                        (<params>, results);
            end
         ...
         autre_service : begin
            do_service [autre_service]
                        (<params>, results);
            end
         ...
      end
      Send (client, results);
   End
```

Types de serveurs

- L'exécution des services peut être réalisée de façon séquentielle: serveur **séquentiel** (figure a).

- Elle peut être sous-traitée à des flots d'exécution indépendants, **processus** ou processus léger (**thread**): serveur **multiple** ou encore de serveur **multiprogrammé** (figure b).
  - accroît le parallélisme de traitement des requêtes issues des clients *mais*
  - impose une synchronisation des différents flots d'exécution lors de l'accès aux données communes du serveur.



Types de serveurs

- Si le serveur peut servir plusieurs clients, il est organisé comme une famille de processus coopérants pour l'exécution concurrente de plusieurs requêtes et exploiter ainsi un multiprocessus ou des entrées-sorties simultanées.
- Le schéma classique comporte un processus cyclique, le **veilleur** (demon), qui attend les demandes des clients. Lorsqu'une demande arrive, le veilleur active un processus **exécutant** qui réalise le travail demandé. Les exécutants peuvent être créés à l'avance et constituer un pool fixe ou être créés à la demande par le veilleur en utilisant des processus ou des processus légers (threads). Ce dernier schéma est illustré ci-après :

```
while true do      -- processus veilleur
begin
   receive (client, message);
   extract (message, service-id, <params>);
   p := create_processus (client,
                  service_id, <params>);
end ;
```

```
Processus p:      -- processus exécutant
Begin
   do_service [service_id](<params>, results);
   send (client, results);
   exit;      -- autodestruction
end
```

Types de serveurs

On distingue différents types de serveurs selon qu'ils gèrent des données **persistantes** ou non, ou selon qu'ils conservent **l'état** du dialogue avec un client ou non.

- Un serveur qui ne gère pas de données persistantes : fournit un résultat calculé uniquement en fonction des paramètres d'appel du service demandé.
  - favorable en ce qui concerne la tolérance aux pannes
  - exemple : le calcul d'une fonction scientifique.

- le serveur avec données persistantes gère des données qui peuvent être modifiées par l'exécution des différents services
  - Contrôler les accès concurrents à ces données partagées pour un serveur « multiprogrammé », et d'autre part de
  - mettre en place une politique de reprise après panne pour assurer la cohérence de ces données.
  - exemple : un objet serveur dont l'état est modifié par l'exécution de ses différentes méthodes.

Types de serveurs

- Dans un serveur sans état, les requêtes émises par un même client s'exécutent sans lien. Il peut y avoir modification de données globales mais le serveur ne garde aucune trace des opérations réalisées par un client (en particulier, l'ordre d'exécution des requêtes n'a pas d'importance).
  - exemple: dans un serveur de fichiers avec accès direct.

- dans certains cas, il est important que le serveur conserve des informations sur les requêtes antérieures, par exemple lorsque l'ordre des appels est un paramètre du service demandé.
  - Un serveur de fichiers permettant l'accès séquentiel en est un exemple (dans ce cas, l'identificateur de l'enregistrement courant est un élément du contexte d'exécution du client sur le serveur).

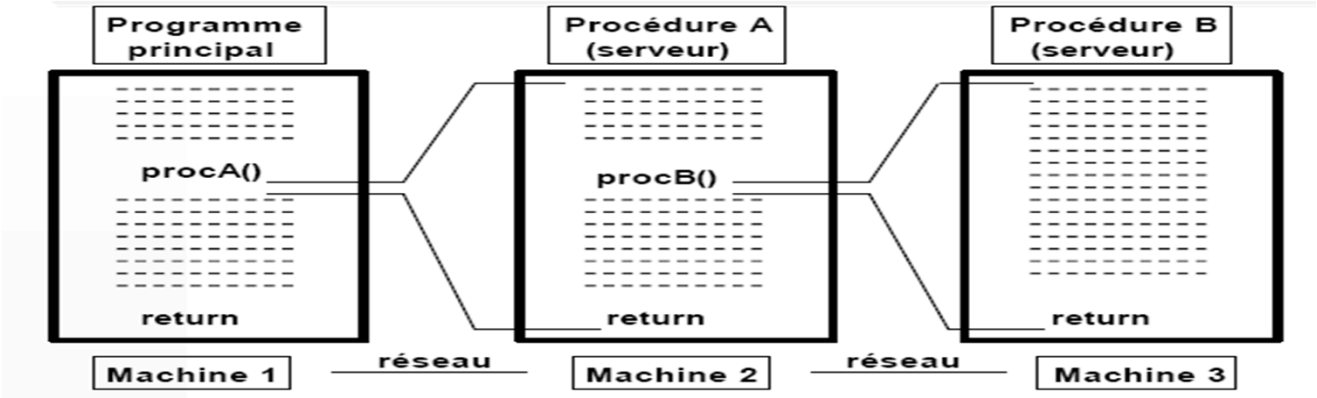
Chapitre 2- le modèle client Serveur

Plan

1. Présentation du modèle CS
2. RPC
3. Mise en œuvre de RPC

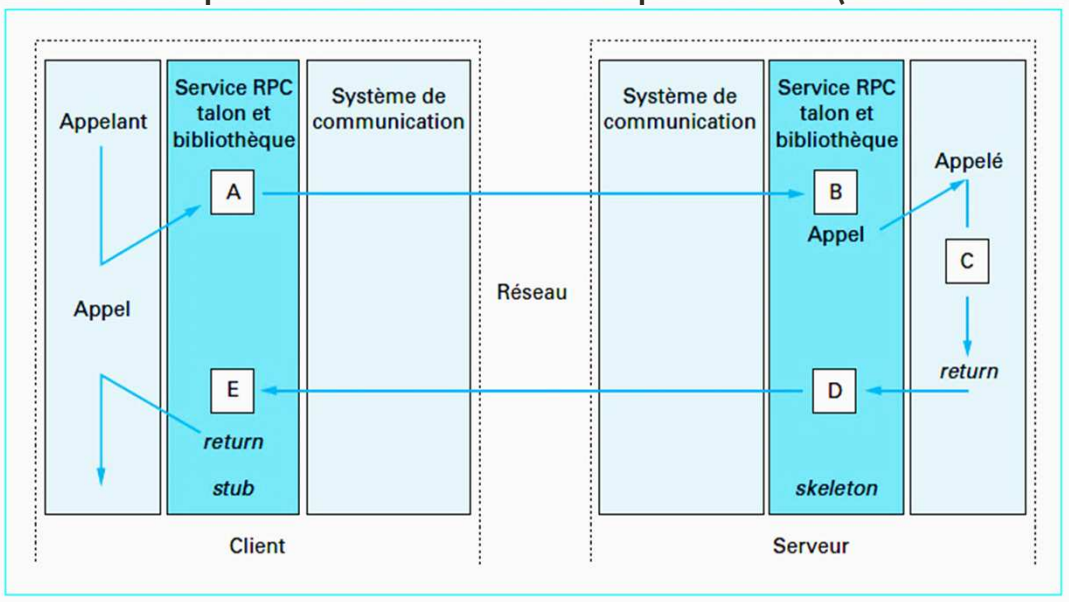
RPC: principe

- RPC : permet à un processus p, implanté sur un site C, d'exécuter une procédure P sur un site distant S, avec un effet globalement identique à celui qui serait obtenu par l'exécution locale de P sur C.
- intérêt : étendre « l'appel de procédure » à un SR
- Un client a une description (ou une interface) de procédures disponibles chez un serveur distant
- Le client invoque une procédure à distance;
- Le serveur exécute localement cette procédure;
- Le serveur renvoie au client le résultat (type de retour) de la procédure.



Mise en œuvre: Appel de procédure à distance

- Utilisation d'un processus s (ou serveur) exécute la procédure P sur le site S. Le processus client p reste bloqué pendant l'exécution de P et il est réactivé au retour de P.
- Deux procédures appelées **talons**. Le talon client (**stub**), sur le site appelant, fournit au processus client une interface identique à celle de la procédure appelée ; le talon serveur (**skeleton**), sur le site appelé, réalise pour le processus serveur une interface identique à celle d'un appel local.
- Chacun des talons, lié au programme du processus client ou serveur sur le site correspondant, fonctionne aussi comme un représentant de l'autre processus (client ou serveur) sur ce site.
- Le rôle des **talons** : factoriser le code qui ne dépend que du service à exécuter, en particulier la réalisation de programmes d'emballage et de déballage qui nécessitent que soit défini un format standard pour la représentation des paramètres.



Mise en œuvre: Appel de procédure à distance

En l'absence de défaillances, les fonctions du talon client, exécutées par le processus appelant, sont les suivantes :

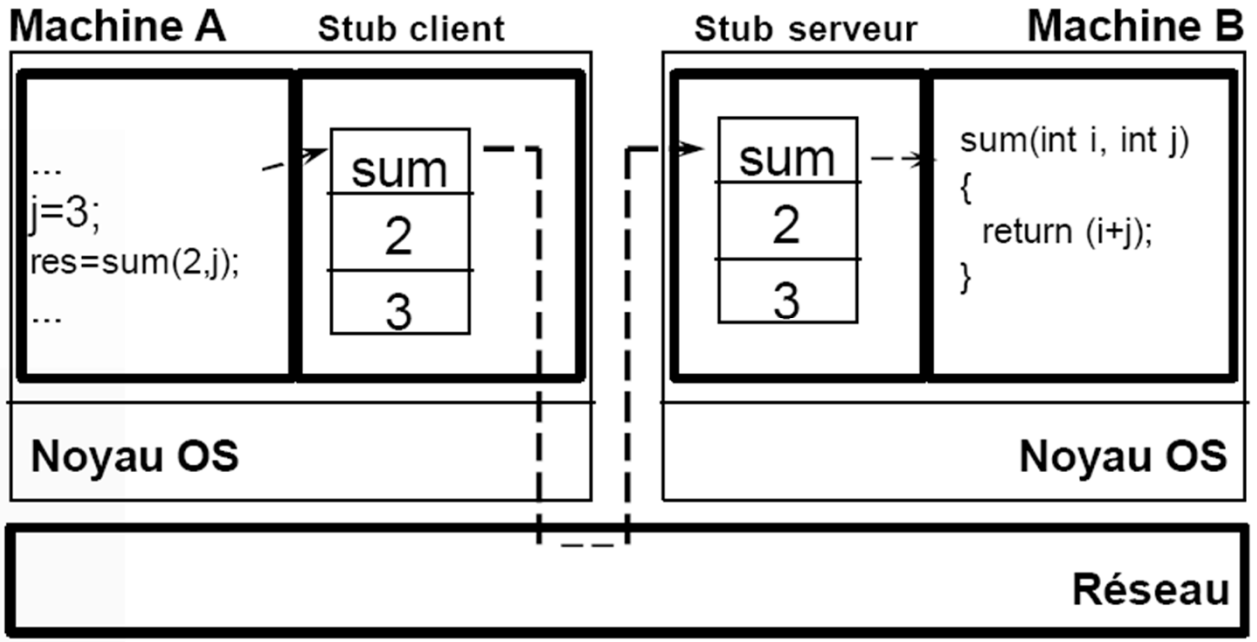
- **au point A :**
  - mettre les paramètres d'appel sous une forme permettant leur transport sur le réseau vers le site appelé : la fonction d'**emballage** (marshalling) ;
  - générer un identificateur pour la requête en cours armer un délai de garde et envoyer vers le site appelé un message contenant l'identité de la procédure appelée et les paramètres d'appel ;
  - mettre en attente le processus client, en attendant la réponse du site appelé ;
- **au point E :**
  - lorsque la réponse arrive, le processus client est réveillé ; il poursuit son exécution dans le talon client en extrayant du message de réponse les résultats, s'il y en a, pour les transformer dans une forme compréhensible par le site local : c'est la fonction de **déballage** (unmarshalling) ;
  - désarmer le délai de garde et acquitter le message de réponse ;
  - exécuter le retour de procédure, avec transmission des résultats ; tout se passe alors, pour le processus client, comme pour le retour d'un appel de procédure local.

Mise en œuvre: Appel de procédure à distance

Sur le site appelé, le processus associé à l'exécution de la requête appelle le talon serveur et exécute les fonctions suivantes :

- **au point B** : à partir du message reçu du client, enregistrer l'identificateur de la requête, déterminer la procédure appelée et déballer les paramètres d'appel ;
- **au point C** : exécuter la procédure appelée en lui passant les paramètres (il s'agit alors d'un appel de procédure local) ;
- **au point D** :
  - au retour de cet appel, l'exécution se poursuit dans le talon serveur, pour préparer le message de retour vers le site appelant, en emballant les éventuels résultats ;
  - armer un délai de garde et envoyer le message de retour au site appelant ;
  - attendre l'accusé de réception du message de réponse, puis terminer l'exécution soit par autodestruction si le processus a été créé à la demande, soit par blocage s'il s'agit d'un processus pris dans un pool.

RPC : Exemple


Machine A
Stub client
Stub serveur
Machine B
...
j=3;
res=sum(2,j);
...
sum
2
3
sum
2
3
sum(int i, int j)
{
return (i+j);
}
Noyau OS
Noyau OS
Réseau

Implantations de RPC

- Le **RPC de Sun/ONC**
- L'**OSF** (Open Software Foundation)
- L'**OMG** (Object Management Group) , au travers de la spécification d'architecture Corba
- L'**environnement DCOM**
- **Java RMI**
- **HTTP**

RPC : Problèmes

Passages de paramètres

- Pour un appel de procédure à distance → recréer les modes de passage de paramètres usuels :
  - **valeur** : ce mode est bien adapté au RPC car, au lieu de recopier sur la pile la valeur du paramètre comme cela est fait dans un appel local, il suffit de transmettre celle-ci dans le corps du message d'appel. Après déballage des paramètres, le talon serveur recopie la valeur reçue dans la pile du processus associé au service appelé ;
  - **copie/restauration** : dans ce mode, la valeur des paramètres est copiée sur la pile de l'appelé comme dans le cas précédent, puis restaurée dans la pile de l'appelant lors du retour ;
  - **référence** impossible en réparti : consiste à recopier sur la pile de l'appelé l'adresse mémoire du paramètre transmis par l'appelant (les espaces d'adressage du client et du serveur sont différents donc aucun sens de passer une adresse)

- La procédure distante n'a pas accès aux variables globales du client, aux périphériques d'E/S (affichage d'un message d'erreur !)
- Un appel de procédure a un fonctionnement synchrone : une instruction suivant un appel de procédure ne peut pas s'exécuter tant que la procédure appelée n'est pas terminée

RPC : Problèmes

Désignation et liaison (principe)

- le **nom** d'un objet →deux fonctions :
  - désigner l'objet,
  - servir de chemin d'accès à l'objet pour permettre son utilisation dans le système.

- Contexte RPC : le nom : id du service service à exécuter + le nom et la localisation du serveur qui exécute ce service.

RPC : Problèmes

- Dans un SR aspects spécifiques :
  - l'espace des noms a une grande taille
  - l'espace des noms évolue dynamiquement et rapidement
    - création de nouvelles entités
    - adjonction permanente de nouveaux composants au système, ou par réorganisation de sa structure interne.
  - la recherche de l'indépendance entre entités logiques réalisant le service et la localisation physique (transparence) conduit à utiliser fréquemment la **liaison dynamique** entre noms et objets. Cette propriété permet de modifier la localisation des services sans modification de nom
  - sous-ensembles du réseau ne sont pas accessibles à un instant donné (pannes ou déconnexions). La panne d'une partie du service de localisation ne doit pas compromettre la réalisation de la désignation dans le reste du système.

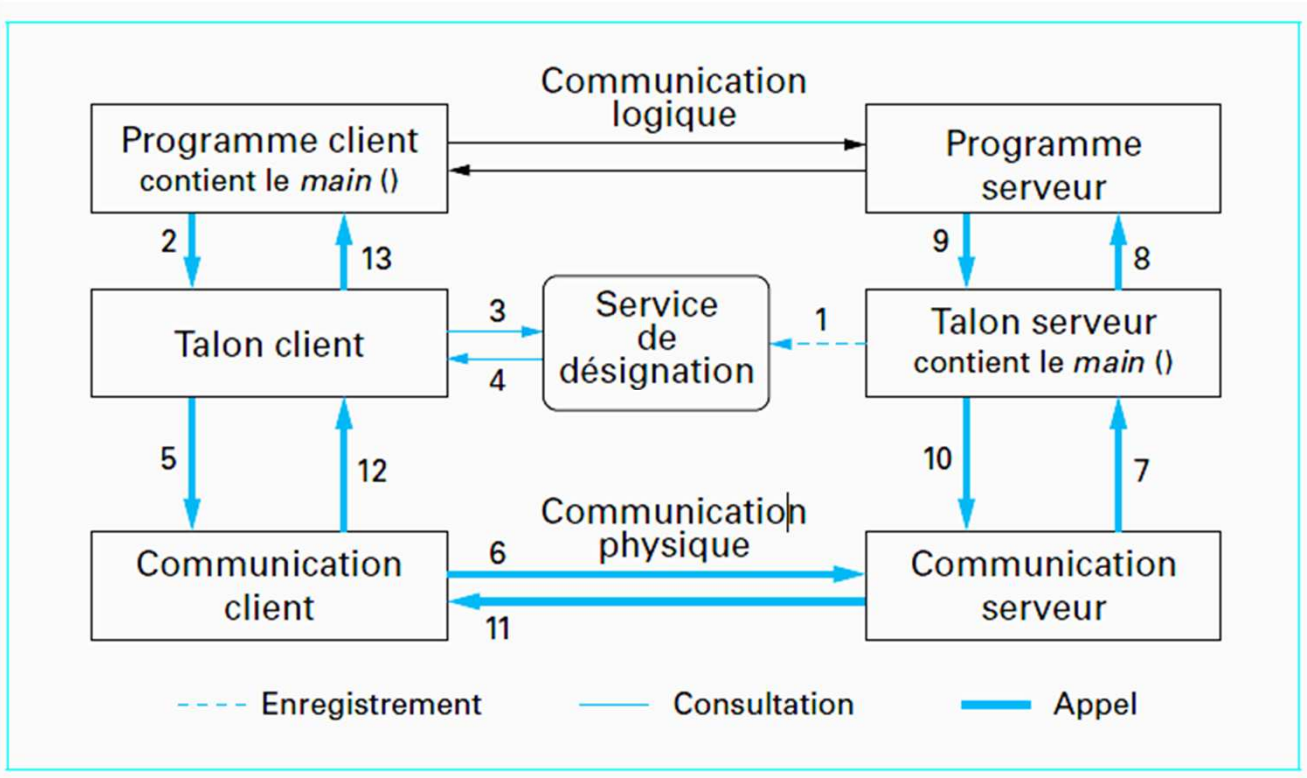
RPC : Problèmes

Désignation et liaison (mise en œuvre)

- Pour localiser un service et y accéder: il doit être enregistré dans un **service de désignation** lui-même connu du client.
  - Le serveur enregistre dans un service de désignation généralement préexistant et connu de tous, différentes informations comme : son nom externe, son nom interne ou sa localisation (son adresse), et une description des services qu'il offre. Les actions d'enregistrement du service dans un annuaire ou de consultation de l'annuaire sont pris en charge par les talons serveur et client.
  - exemple de tels services : le DNS d'Internet
- Pour effectuer un appel, le client transforme le nom externe (symbolique) en une chaîne d'accès repose sur le nom interne (adresse). Transformation = liaison peut être effectuée :
  - **statiquement** : la correspondance entre nom du service et adresse est fixée une fois pour toutes, lors de l'écriture du programme ou dans une phase préalable à l'exécution. Le nom est immuable.
  - **dynamiquement** : l'adresse du serveur est recherchée par le client lors du premier appel au serveur ou lors de chaque appel → plus de souplesse dans la mise en oeuvre du service et permet une modification de la localisation du serveur.

RPC : Problèmes

Désignation et liaison



RPC : Problèmes

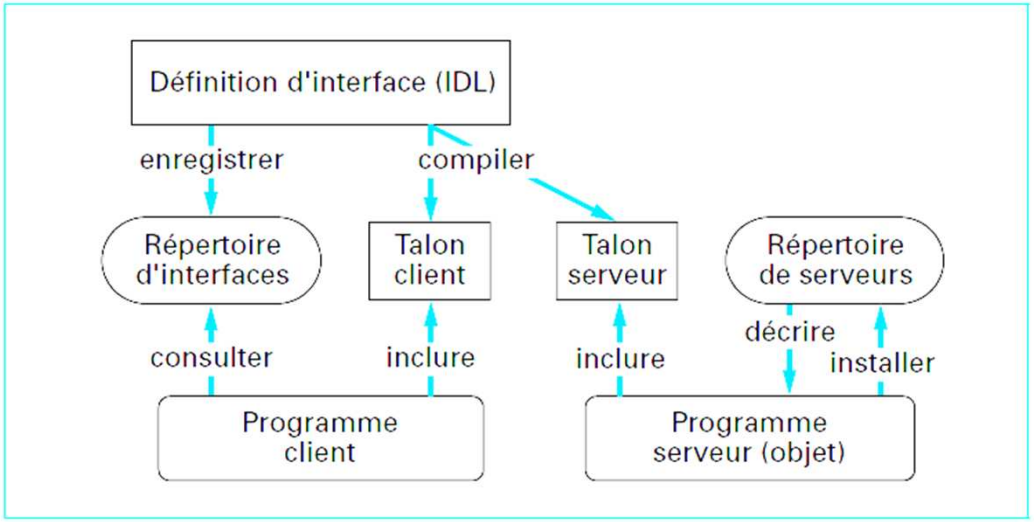
Appel statique et appel dynamique

- La plupart par des RPC offrent uniquement un mode de construction statique des talons clients et serveurs.
- Pour offrir plus de souplesse, certains RPC (Corba) permettent la construction dynamique de requêtes (appels dynamiques DII : Dynamic Invocation Interface) ou la construction dynamique de talons serveurs (DSI : Dynamic Skeleton Interface)
- l'appel dynamique impose de pouvoir récupérer, lors de l'exécution, l'interface du service que l'utilisateur souhaite accéder.

Pour mettre en place cette fonction

la description d'interface est enregistrée

dans un **répertoire d'interfaces.**



Clients et serveurs dans le modèle CORBA

RPC : Problèmes

Problème d'hétérogénéité

L'hétérogénéité des données se trouve à différents niveaux :

- le codage des données en mémoire. Deux représentations coexistent.
- le codage des données par les compilateurs .
  - Exemple : 2 compilateurs C n'ont pas nécessairement la même manière de coder des structures et d'aligner ou non les données sur des mots mémoires. Ces spécificités dépendent en même temps du compilateur (et de sa volonté d'optimiser soit la place mémoire, soit les temps d'accès) et des contraintes des processeurs ;
- le codage des données par les applications: les applications manipulent des données structurées qu'il est nécessaire de faire migrer entre stations (sons, images, documents, etc.).

RPC : Problèmes

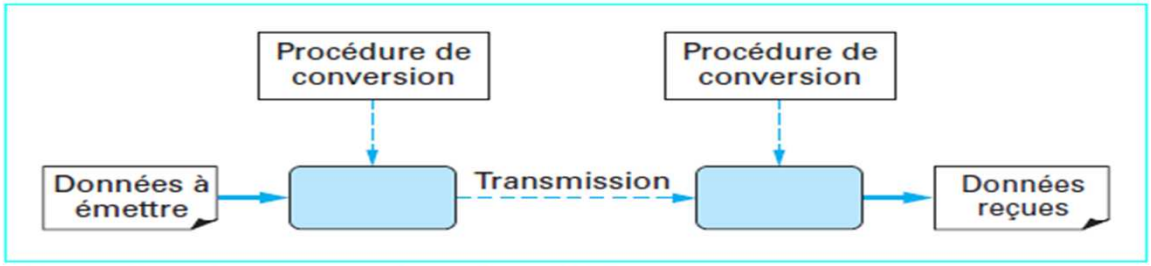
Problème d'hétérogénéité : Codage des données par les processeurs

- Codage/décodage de chaque type de donnée de toute architecture à toute autre architecture
- Format universel intermédiaire (XDR, etc.) :
  - solutions de bas niveau, imposent l'utilisation d'une double transformation (côté client et côté serveur) parfois inutile dans le cas où le serveur et le client utilisent déjà le même codage

Solution de Sun Microsystems

Format eXternal Data Representation ou XDR

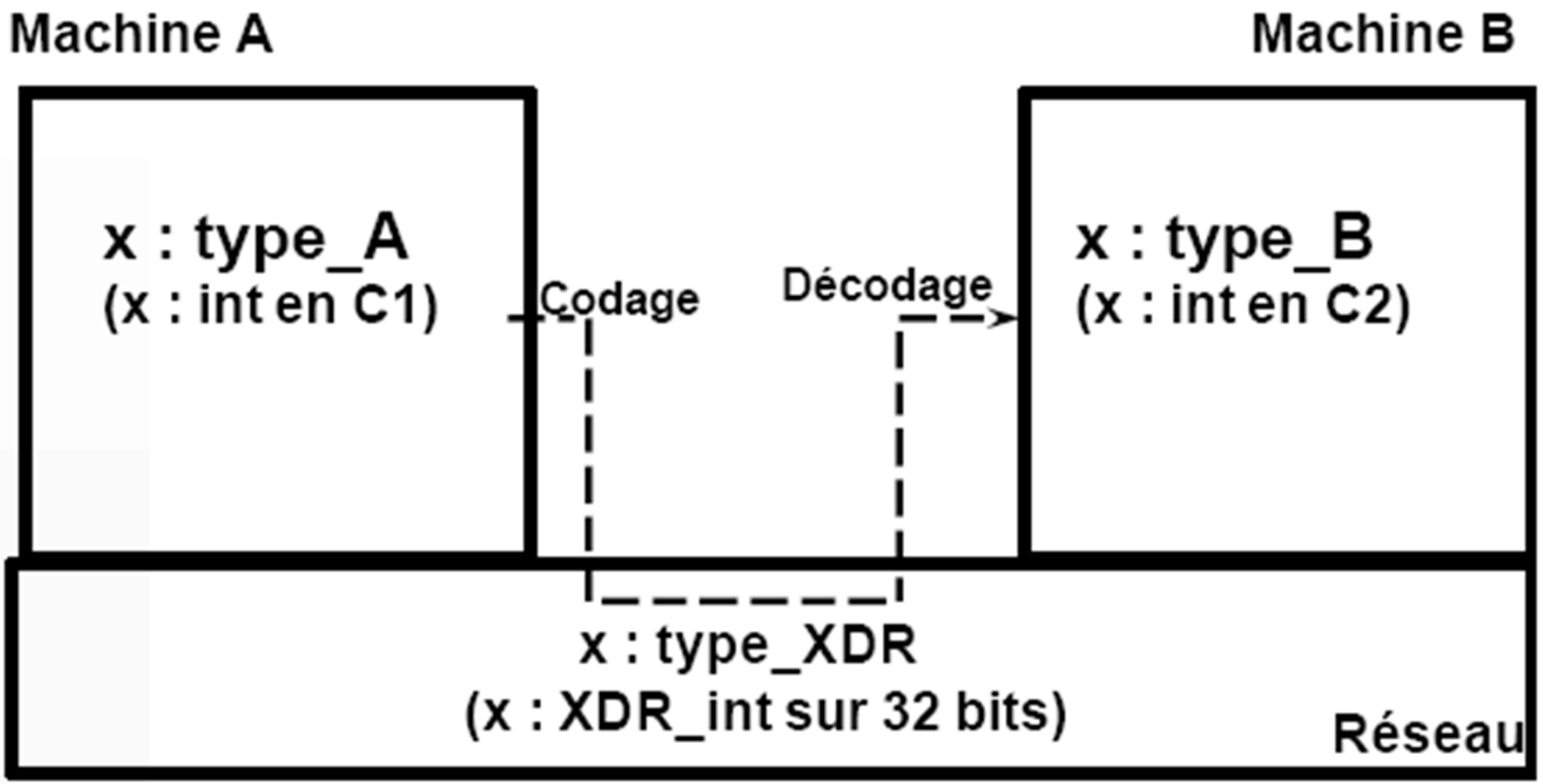
Librairie XDR (types de données XDR + primitives de codage/décodage pour chaque type)



Codage / décodage XDR

RPC : Problèmes

XDR: Principe



RPC : Problèmes

Problème d'hétérogénéité : Codage des données par les applications

- Le développement des documents électroniques a fait émerger la nécessité de posséder des formats de représentation évolutifs et universels.

- XML est un format qui permet de mettre des données structurées dans un fichier texte (ce qui ne signifie pas nécessairement que ce fichier texte soit lisible directement).
  - exemples de données structurées : feuilles de calcul, carnet d'adresses, transactions financières, dessins en CAO, etc.
  - L'intérêt de ces formats : manipuler ces données à travers d'autres logiciels que ceux qui les ont produites, indépendamment d'une plate-forme donnée.

Chapitre 2- client Serveur et RPC

Plan

1. Présentation du modèle CS
2. RPC
3. Mise en œuvre de RPC

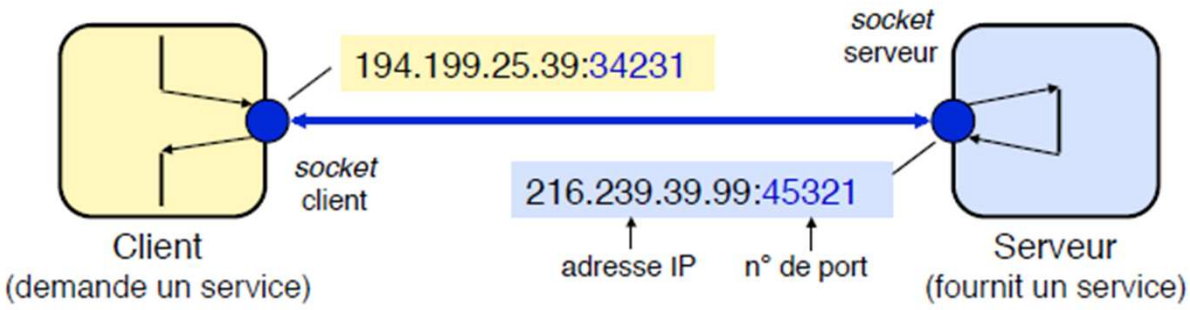
Mise en œuvre de RPC

Les approches pour la réalisation effective d'un appel de procédure à distance utilisent des interfaces de programmation de différents niveaux :

- Envoi de messages (utilisation des *sockets*)
- Appel de procédure avec langage de description d'interface (*IDL*) et d'un générateur de talons (RPCgen de Sun)
- Appel de méthodes à distance : RPC à objets (utilisation de Java RMI).

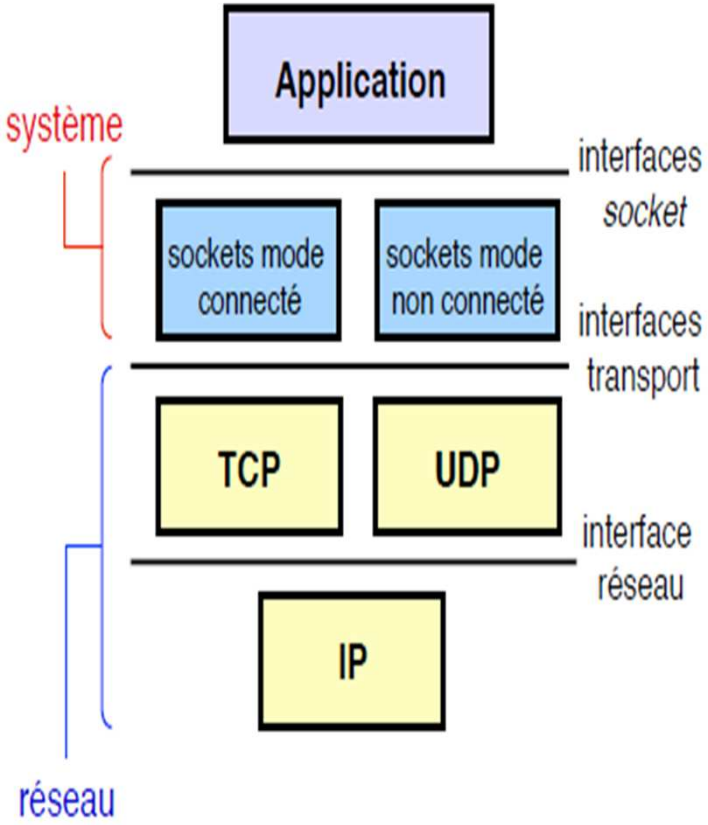
Envoi de messages

- Pour programmer une application client-serveur, il est commode d'utiliser les *sockets*. Les *sockets* fournissent une interface qui permet d'utiliser facilement les protocoles de transport TCP et UDP.
- Une *socket* est un moyen de désigner l'extrémité d'une connexion, côté émetteur ou récepteur, en l'associant à un port.
- Une fois la connexion (bidirectionnelle) établie via des *sockets* entre un
- processus client et un processus serveur, ceux-ci peuvent communiquer en
- utilisant les mêmes primitives (*read*, *write*) que pour l'accès aux fichiers.



Les sockets

Une socket est une interface de communication introduite par les systèmes Unix pour la communication réseau. Il s'agit d'un point d'accès aux services de la couche transport, c'est-à-dire TCP ou UDP. La communication par sockets sur un réseau adopte un modèle client-serveur ; en d'autres termes pour communiquer il faut créer un serveur prêt à recevoir les requêtes d'un client.



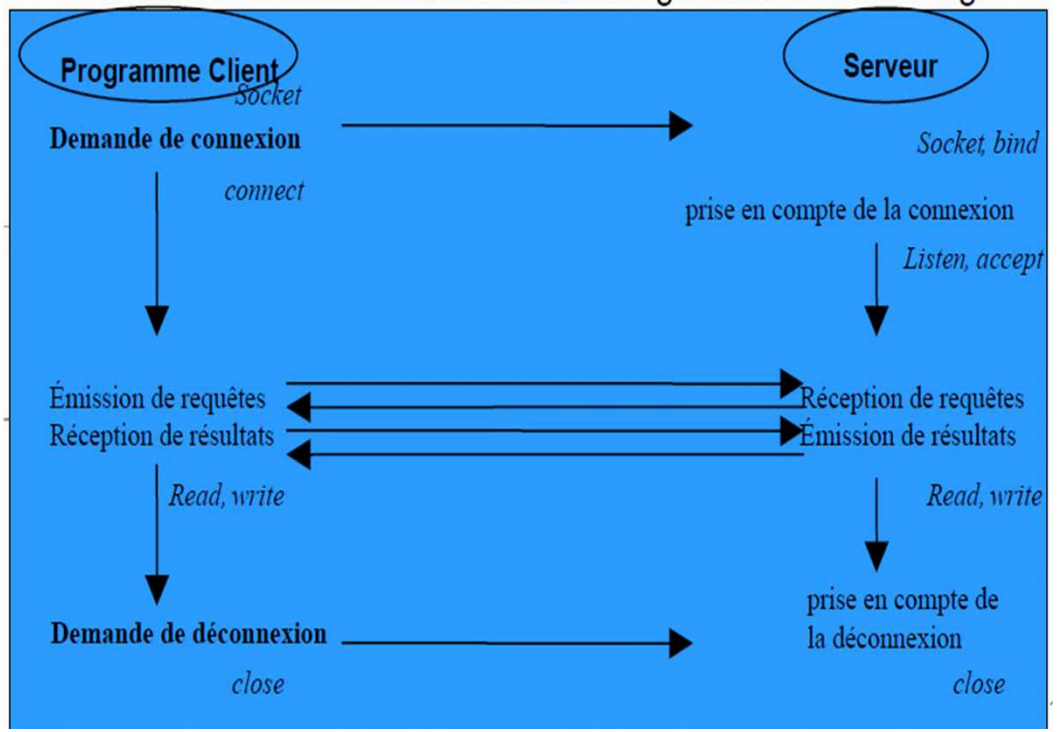
Les sockets

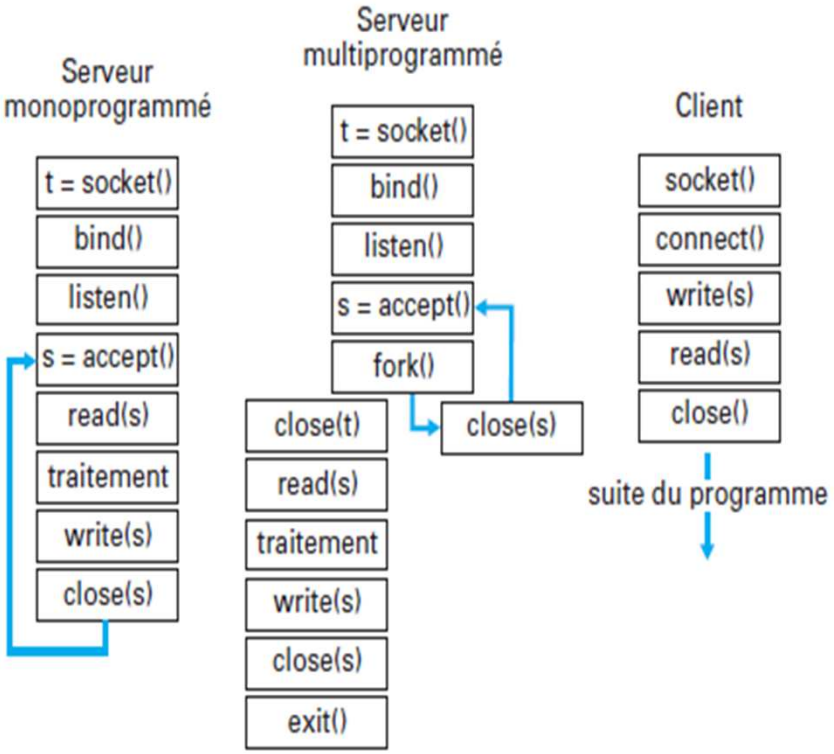
- **TCP (mode connecté) :** une liaison est établie au préalable entre deux hôtes, et ensuite les messages (plus exactement des flots d'octets) sont échangés sur cette liaison
- **UDP (mode non connecté)** :aucune liaison n'est établie. Les messages sont échangés individuellement
- Les protocoles connectés sont généralement préférables pour des transmissions sur une longue durée, sur une longue distance ou avec un important volume de données à transférer.
- Pour les réseaux locaux où le temps de réponse est crucial, on choisit souvent des protocoles non connectés.

CS en mode connecté et non connecté

Mode connecté

- le client initie une connexion avec le serveur par l'appel de *connect* et décide d'y mettre terme en appelant *close*.
- Une fois la connexion établie, aucun des deux sites n'a besoin d'inclure des informations d'adressage dans son message



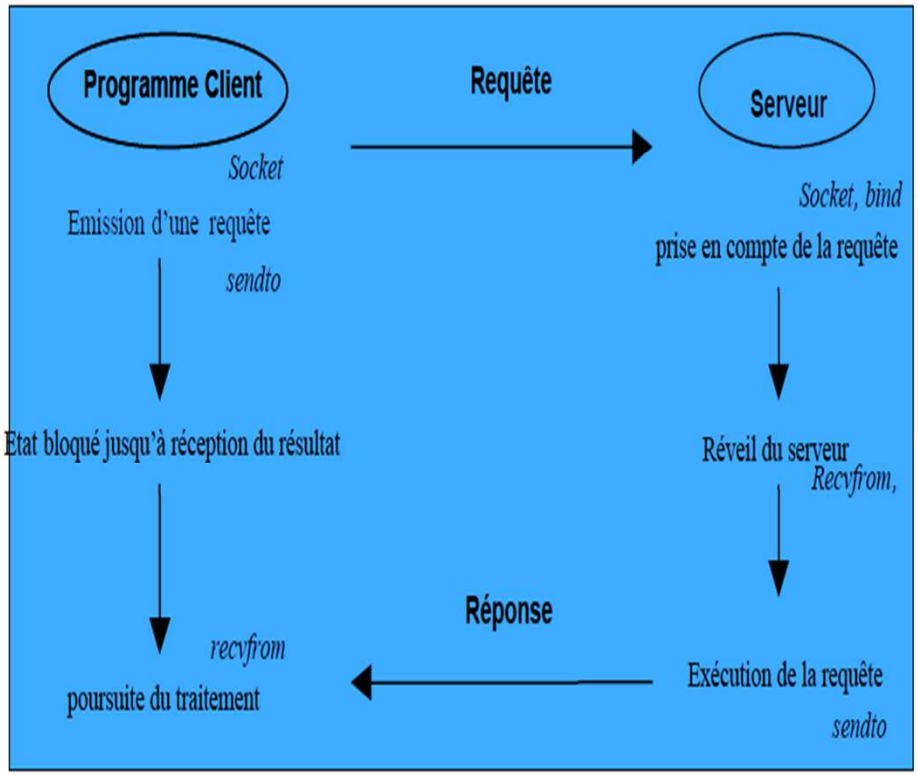


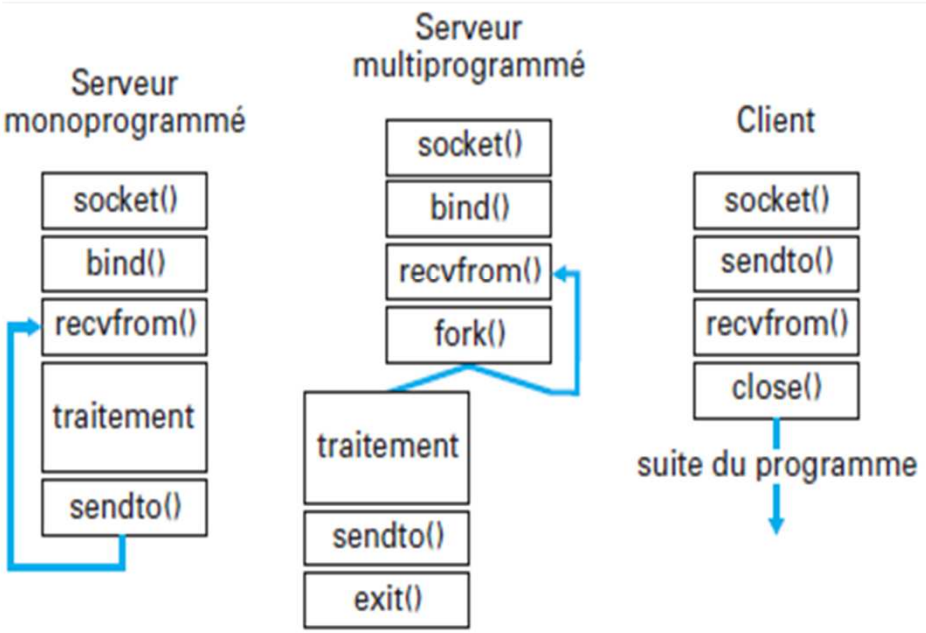
(b) client-serveur en mode connecté

CS en mode connecté et non connecté

Mode non connecté

- le client ne transmet qu'une requête et reste bloqué jusqu'à la réception de la réponse du serveur.





ⓐ client-serveur en mode non connecté

Création d'une socket

La création d'une socket se fait par la fonction socket dont la déclaration se trouve dans <sys/socket.h>. Cet appel permet de créer une structure en mémoire contenant tous les renseignements associés à la socket (buffers, adresse, etc.) ; il renvoie un descripteur de fichier permettant d'identifier la socket créée (-1 en cas dérreur).

```
int socket (
int domain, /* AF_INET pour l'internet */
int type, /* SOCK_DGRAM pour une communication UDP,
                    SOCK_STREAM pour une communication TCP */
int protocole /* 0 pour le protocole par defaut du type */
);
```

Création d'une socket

Une fois la socket créée, il est possible de lui attacher une adresse qui sera généralement l'adresse locale ; sans adresse une socket ne pourra pas être contactée (il s'agit simplement d'une structure qui ne peut pas être vue de l'extérieur). L'attachement permet de préciser l'adresse ainsi que le port de la socket. On attache une adresse à une socket à l'aide de la fonction bind qui renvoie 0 en cas de succès et -1 sinon.

```
int bind (
int descr, /* descripteur de la socket */
struct sockaddr *addr, /* adresse a attacher */
int addr_size /* taille de l'adresse */
);
```

sockaddr

La **structure utilisée avec TCP/IP** est une adresse AF_INET (Généralement les structures d'adresses sont redéfinies pour chaque famille d'adresse). Les adresses AF_INET utilisent une structure *sockaddr_in* définie dans <*netinet/in.h*> :

Exemple

Cet exemple définit une fonction permettant de créer une socket et de l'attacher sur le port spécifié de l'hôte local.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include "creer_socket.h"
/* ************************************************************
* type : type de la socket a créer.
* ptr_port : pointeur sur le numéro de port désire.
* ptr_adresse : résultat final de l'attachement.
************************************************************/
```

Exemple


```
int creer_socket (int type, int *ptr_port, struct
sockaddr_in *ptr_adresse)
{
int desc;
int longueur=sizeof(struct sockaddr_in);
struct sockaddr_in adresse;
/* Creation de la socket */
if ((desc=socket(AF_INET,type,0)) == -1)
{
perror("Creation de socket impossible");
return -1;
}
/* Preparation de l'adresse d'attachement */
adresse.sin_family=AF_INET;
/* Conversion (representation interne) -> (reseau)
avec htonl et htons */
adresse.sin_addr.s_addr=htonl(INADDR_ANY);
adresse.sin_port=htons(*ptr_port);
/* Demande d'attachement de la socket */
if (bind(desc,(struct sockaddr*)&adresse,longueur)
== -1)
{
perror("Attachement de la socket impossible");
close(desc);
return -1;
}
/* Recuperation de l'adresse effective
d'attachement */
if (ptr_adresse != NULL)
getsockname(desc,(struct
sockaddr*)ptr_adresse,&longueur);
return desc;
}
```

Communication en mode non connecté

Il s'agit ici de créer la socket qui recevra le message la demande de connexion. Ensuite on attend le message à l'aide de la fonction recvfrom

```
int recvfrom (
int desc, /* descripteur de la socket */
void *message, /* adresse de reception */
int longueur, /* taille de la zone reservee */
int option, /* 0 pour une lecture simple */
struct sockaddr *ptr_adresse, /* adresse emetteur */
int *long_adresse /* taille de la zone adresse */
);
```

Exemple serveur

On illustre ici le côté serveur par la création d'un processus permettant la réception d'un unique message sur le port passé en argument.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include "creer_socket.h"
```

```
int main (int argc, char *argv[])
{
struct sockaddr_in adresse;
int port,desc_socket,lg=sizeof(adresse);
char message[4096];
if (argc < 2)
{
fprintf(stderr,"udp_serveur num_socket\n");
return 1;
}
/* creation et attachement de la socket */
port=atoi(argv[1]);
if ((desc_socket=creer_socket(SOCK_DGRAM, &port,
&adresse)) == -1)
{
fprintf(stderr,"Creation de socket impossible\n");
exit(2);
}
/* attente du message */
recvfrom(desc_socket,message,4096,0,(struct
sockaddr*)&adresse,&lg);
printf("Message %s", message);
return 0;
}
```

Côté client

Côté client

Il s'agit ici d'envoyer un message sur une machine distante. Pour cela on commence par créer la socket émettrice, puis on prépare l'adresse de destination et on envoie le message à l'aide de la fonction sendto.

```
int sendto (
int desc, /* descripteur de la socket */
void *message, /* message a envoyer */
int longueur, /* longueur du message */
int option, /* 0 pour un envoi simple */
struct sockaddr *ptr_adresse, /* adresse destinataire */
int *long_adresse /* taille de la zone adresse */
);
```

Exemple client

On illustre ici le côté client par la creation d'un processus permettant lénvoi du message "Salut" sur la machine et le port specifiés en argument.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include "creer_socket.h"
int main (int argc, char *argv[])
{
struct sockaddr_in adresse;
int port,desc_socket,lg=sizeof(adresse);
struct hostent *hp;
char message[]="Salut\n";
if (argc < 3)
{
fprintf(stderr,"udp_client machine port_distant\n");
exit(1);
}
```

Exemple client (suite)


```
/* creation et attachement de la socket sur un port quelconque */
port=0;
if ((desc_socket=creer_socket(SOCK_DGRAM, &port, &adresse)) == -1)
{
fprintf(stderr,"Creation de socket impossible\n");
exit(2);
}
/* recherche de l'adresse internet du serveur */
if ((hp=gethostbyname(argv[1])) == NULL)
{
fprintf(stderr,"Machine %s inconnue\n",argv[1]);
exit(3);
}
/* preparation de l'adresse destinatrice */
adresse.sin_family=AF_INET;
adresse.sin_port=htons(atoi(argv[2]));
memcpy(&adresse.sin_addr.s_addr,hp->h_addr,hp->h_length);
/* envoi du message */
sendto(desc_socket,message,strlen(message)+1,0,(struct sockaddr*)&adresse,lg);
exit(0);
}
```