Modèle client-serveur

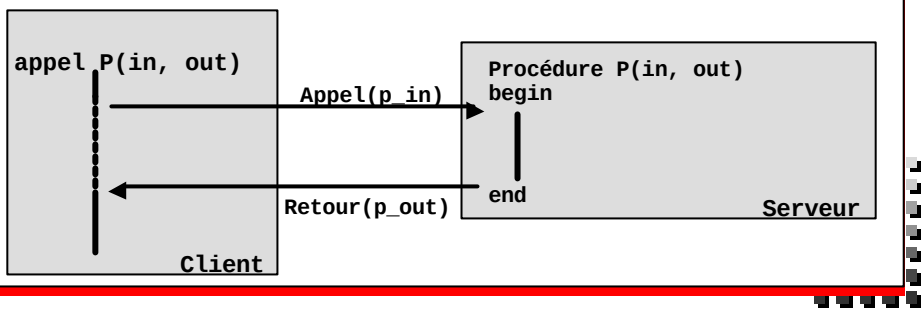
- Définition - Présentation
- Exemple : Les sockets
  - Unix
  - Java
- Le RPC
  - Modèle - Fonctionnement
  - Extension
  - Exemple : Le RPC Sun

Modèle client-serveur définition

- **Application client/serveur**
  - 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)
  - Client
    - programme qui émet des requêtes (demandes de service). Il est toujours l'initiateur du dialogue
  - Serveur
    - programme offrant un service sur un réseau (par extension, machine offrant un service)

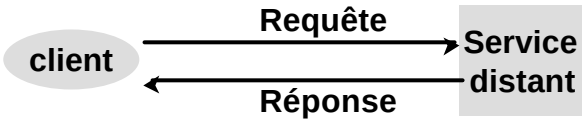
Modèle client-serveur
Communication par messages

- **Deux messages (au moins) échangés**
  - Le premier message correspondant à la requête est celui de l'appel de procédure, porteur des paramètres d'appel.
  - Le second message correspondant à la réponse est celui du retour de procédure porteur des paramètres résultats.

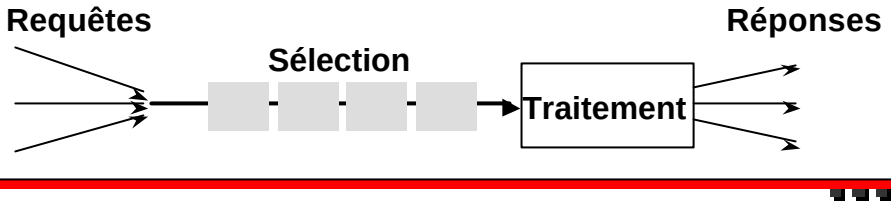


Modèle client-serveur
Principe

- Vu du client

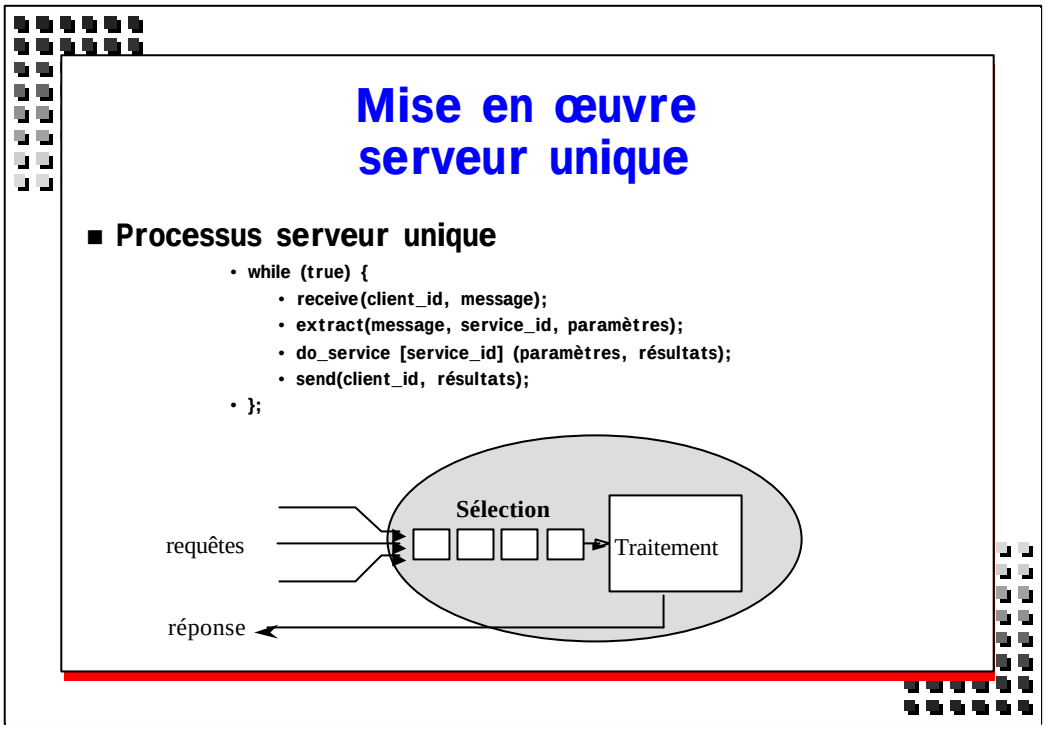


- Vu du serveur
  - Gestion des requêtes (priorité)
  - Exécution du service (séquentiel, concurrent)
  - Mémorisation ou non de l'état du client



Modèle client-serveur
Gestion des processus

- **Clients et serveurs sont dans des processus distincts**
  - Le client est suspendu lors de l'exécution de la requête
  - Eventuellement, exécution concurrente de plusieurs requêtes chez le serveur
    - Plusieurs processus
    - Plusieurs threads (processus légers) dans un même espace virtuel.

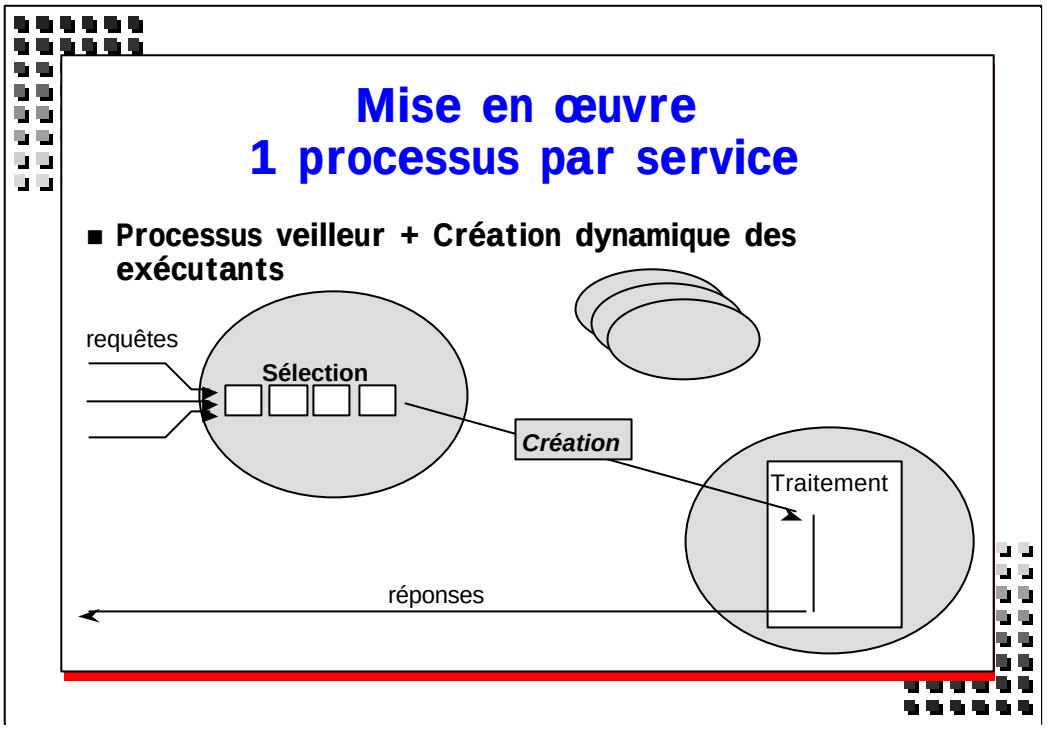


Un seul et unique processus serveur reçoit la requête, il en extrait le l'identification du service requis par le client (service_id) et les paramètres nécessaires à l'exécution de ce service.

Exemple: dans le cas d'un serveur de noms (DNS), le service_id peut-être DNS_LOOKUP pour identifier une requête d'interrogation de la base et les paramètres une chaîne de caractère contenant le nom de la machine dont on veut retrouver l'adresse IP.

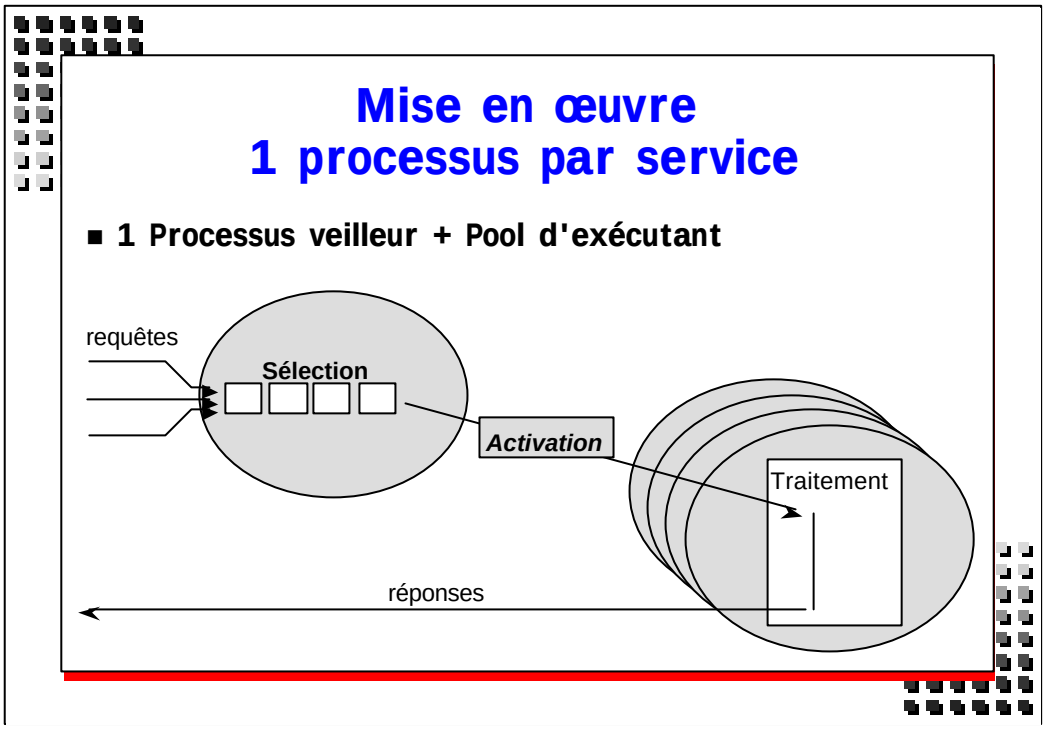
Une fois le service identifié, le serveur extrait les paramètres du message puis exécute le traitement approprié ; il récupère alors les résultats et construit un message de réponse qu'il transmet au client.

Cette mise en œuvre simpliste, ne permet pas (en première approximation) de parallélisme dans le traitement des requêtes ; elle est donc essentielement utilisée dans la cas ou le temps de traitement des requêtes est court.



Dans cette mise en œuvre, un processus veilleur reçoit les requêtes et crée un processus pour traiter chacune ; une fois la requête traitée, ce processus émet la réponse à destination du client puis disparaît.

Ce type de mise en œuvre est en général utilisé en mode connecté lorsque la durée du traitement peut être très longue : exemple ftp, telnet, etc.



Cette mise en œuvre est une variante de la précédente ; le processus veilleur, au lieu de créer un processus par requête à exécuter, distribue chaque requête à un processus disponible d'un ensemble statique d'exécutant.

Cette mise en œuvre est en général utilisée pour parallèliser les traitements lorsque la durée des traitements est "raisonnable" ; elle peut être observée par exemple dans l'implémentation Unix de NFS (un ensemble de processus nfsd répondant cycliquement aux requêtes), ou dans l'implémentation du serveur Web Apache (processus httpd).

Différents types de service
Sans données rémanente/persistante

- situation idéale ou le service s 'exécute uniquement en fonction des paramètres d 'entrée
  - pas de modification de données rémanente sur le serveur
- solution très favorable
  - pour la tolérance aux pannes
  - pour le contrôle de la concurrence
- exemple :
  - calcul d 'une fonction scientifique
  - serveur de noms, serveur Web, etc.

Différents types de service avec donnée rémanente/persistante

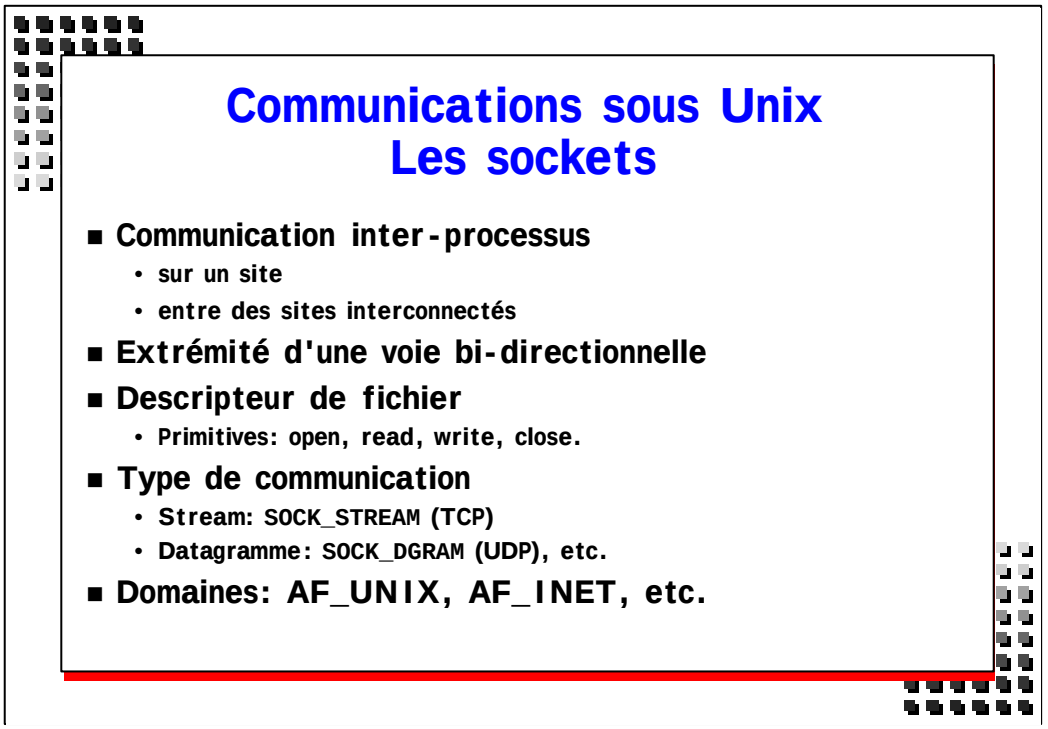
- Les exécutions successives manipulent des données persistantes
  - modification du contexte d 'exécution sur le site distant
  - problèmes de contrôle de la concurrence
  - difficultés en cas de panne en cours d 'exécution
- Exemples
  - Serveur de fichier réparti
    - Pose de verrou pour les opérations d 'écriture
  - Etat d'un objet manipulé par ses méthodes

Différents types de service mode sans état

- Les appels de procédure s'exécutent sans lien entre eux
  - il peut y avoir modification de données globales mais chaque opération s'effectue sans lien avec celles qui l'ont précédé
- exemple
  - NFS (Network File System de SUN - SGF réparti)
    - Lecture (fichier F, bloc N)
    - Ecriture (fichier F, bloc N)

Différents types de service mode avec état

- Les appels successifs s 'exécutent en fonction d'un état laissé par les appels antérieurs
  - gestion de l 'ordre des requêtes est indispensable
- exemple
  - lecture d'un enregistrement d'un fichier en accès séquentiel (dépendant du pointeur courant)
  - appel de méthode sur un objet



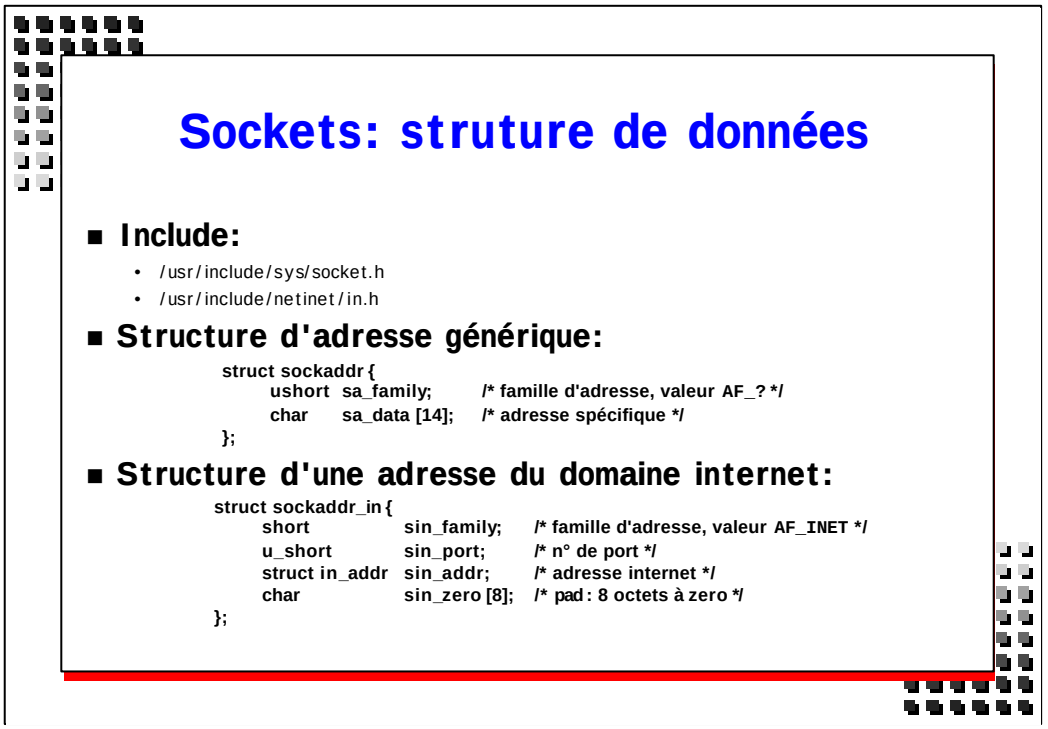
Les sockets permettent la communication entre processus d'une même machine ou de machines interconnectées. Un socket est l'extrémité d'une voie bi-directionnelle établie entre deux processus. Il est vu par l'utilisateur (programmeur) comme un descripteur de fichier ordinaire ; en d'autres termes, on peut leur appliquer les opérations d'entrées/sorties définies sur les fichiers (read, write, close) en plus des opérations spécifiques définies ci-après.

Les sockets permettent d'utiliser une interface commune pour l'écriture d'applications réparties, il en existe de plusieurs types selon le protocole choisi pour la communication :

• Circuits virtuels, via TCP : utilisation d'un mode connecté, taille illimité des messages, ordonnancement, fiabilité.

• Datagramme, via UDP.

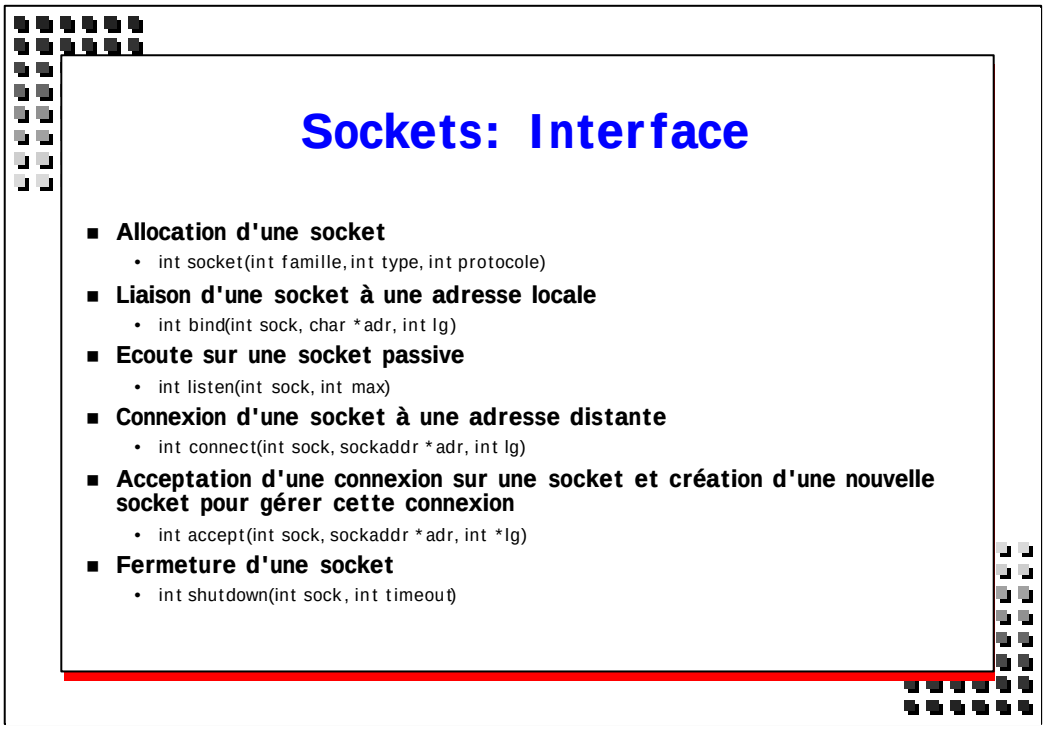
La communication par socket fonctionne à l'intérieur d'un domaine qui définit les entités accessibles. Deux domaines sont prédéfinis : un domaine local (AF_UNIX) interne à un système, un domaine global (AF_INET) qui correspond à l'espace de noms d'Internet.



Au dessus de la couche réseau IP, deux protocoles de transport sont fournis UDP et TCP. L'adressage à ce niveau est réalisé par des numéros de portes (16 bits), une adresse est donc constituée de l'adresse IP (4 octets) de la machine et du numéro de port. Certains ports sont universellement connus et attribués par une administration centrale (SNMP, Telnet, Ftp, Pop, etc.).

Le protocole UDP est un protocole de transport de type datagramme sans connexion, il reprend l'essentiel des fonctions d'IP en fournissant en outre l'adressage par porte propre à la couche transport. Un datagramme UDP est un paquet IP complété des adresses de portes source et destination, et d'un CRC.

Le protocole TCP permet l'établissement d'une connexion entre deux processus, et la communication sur ce circuit virtuel par messages de taille quelconque. TCP fournit des primitives d'attente de connexion, d'établissement et de fermeture d'une connexion, l'émission et la réception de données sur un circuit virtuel. TCP fournit un contrôle d'erreur et un contrôle de flux.



La primitive `socket` permet de créer la structure de données système permettant la manipulation d'une socket. Cette socket est créé dans le domaine spécifié, avec le mode de communication spécifié.

La primitive `bind` permet la liaison d'un descripteur de socket à une adresse transport (@IP + port).

La primitive `listen` permet de spécifier la taille maximale de la file d'attente d'un socket (nombre de demande de connexions non traitée en attente).

La primitive `connect` initialise la connexion avec un autre processus, il s'agit d'une connexion entre le socket local spécifié par son descripteur `sock`, et un socket distant dont l'adresse est donnée par `adr`.

La primitive `accept` crée un nouveau sock sur lequel seront fait les échanges de données ultérieurs. Le descripteur de ce socket est rendu en résultat. L'adresse du socket source de la connexion est rendue dans `adr`.

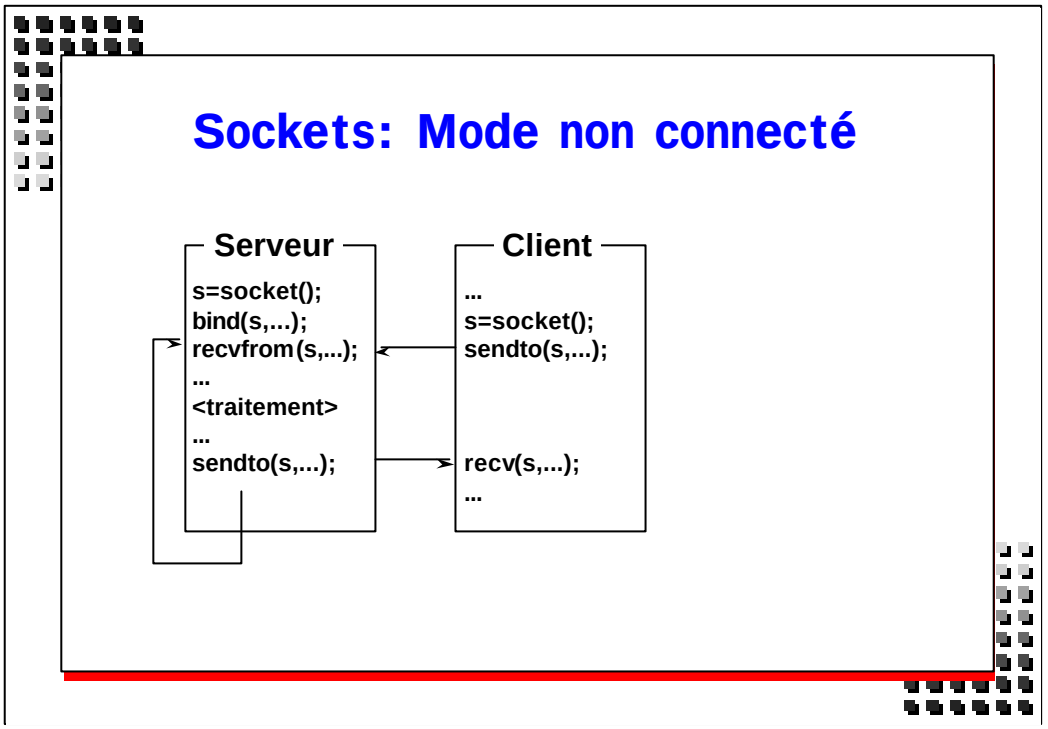
La primitive `shutdown` (`close`) permet de fermer une connexion.

Sockets: Interface

- **Transfert en mode déconnecté (datagramme)**
  - **Emission de requêtes ou de données**
    - int sendto (int sock, char *msg, int lg, int opt, sockaddr *dest, int lg)
  - **réception de données et de l'adresse de l'émetteur**
    - int recvfrom(int sock, char *msg, int lg, int opt, sockaddr *src, int *lg)
- **Transfert en mode connecté (circuit virtuel)**
  - **Emission de requêtes ou de données**
    - int send((int sock, char *msg, int lg, int opt)
  - **Réception de données**
    - int recv((int sock, char *msg, int lg, int opt)

Autres appels

- **Récupération des adresses Internet:**
  - **gethostbyname**
- **Récupération du numéro de port du serveur:**
  - **getservbyname**
- **Conversion de données:**
  - **htonl, ntohl**
  - **htons, ntohs**

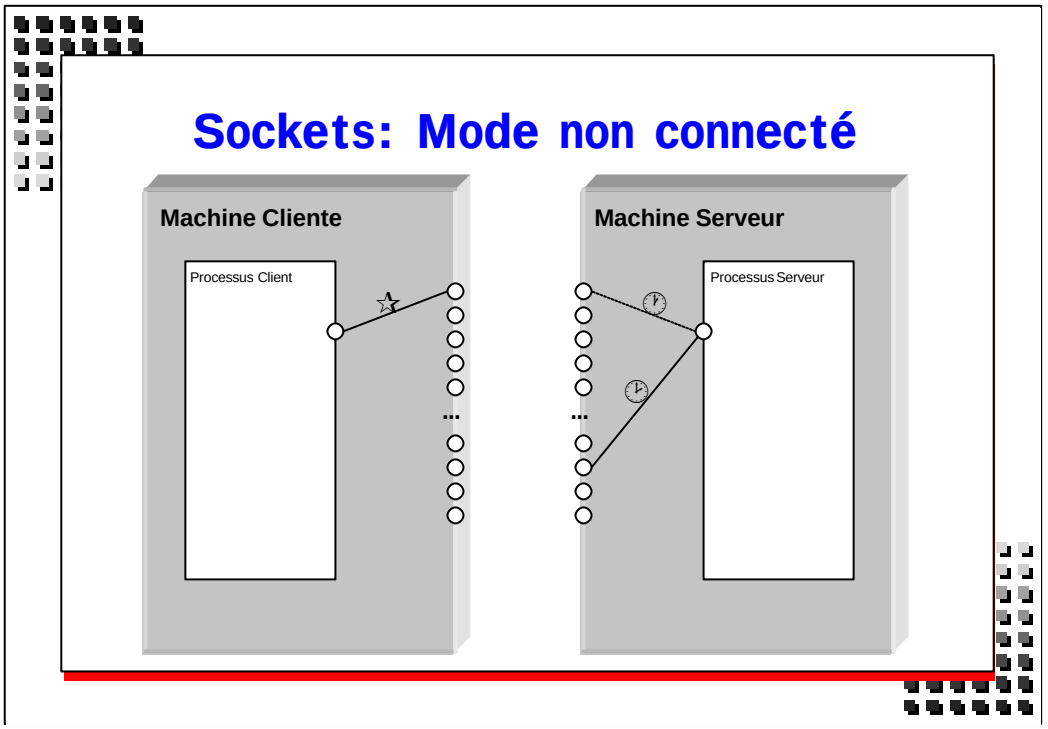


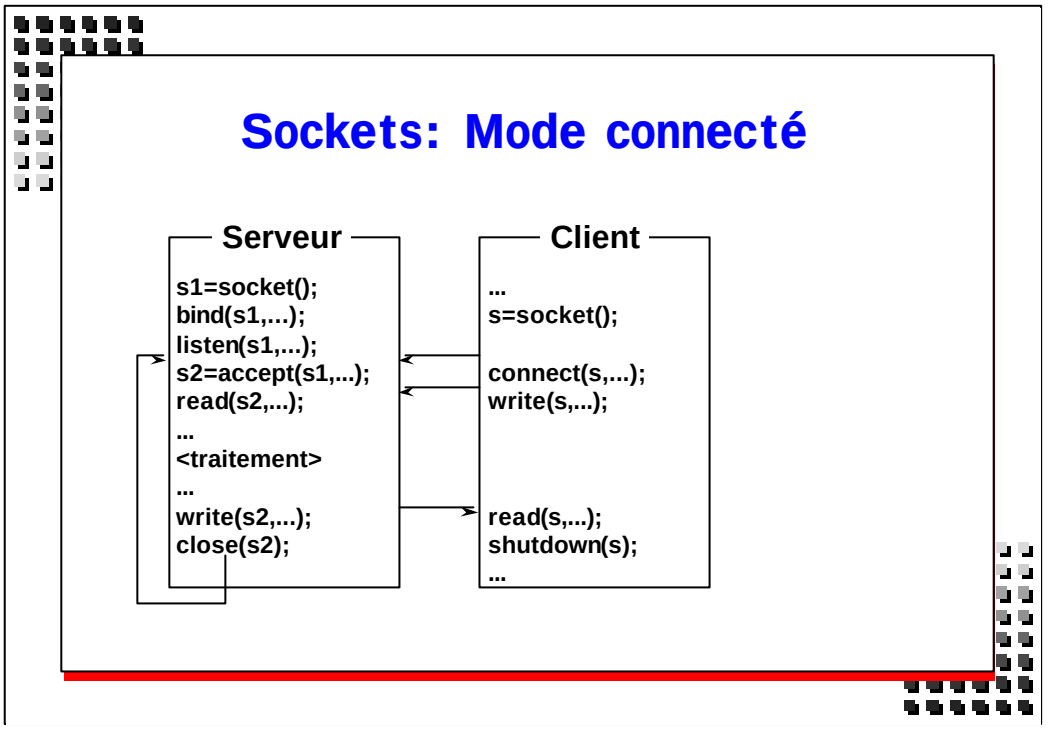
Serveur

1. Création d'une socket
2. Lien de la socket à l'adresse du serveur (N° IP, port)
3. Réception d'une requête
4. Traitement
5. Retour du résultat en utilisant l'@ fournie par recvfrom

Client

1. Création d'une socket
2. Envoi de la requête au serveur (N° IP, port)
3. Attente de la réponse
4. Suite du programme


Sockets: Mode non connecté
Machine Cliente
Processus Client
Machine Serveur
Processus Serveur

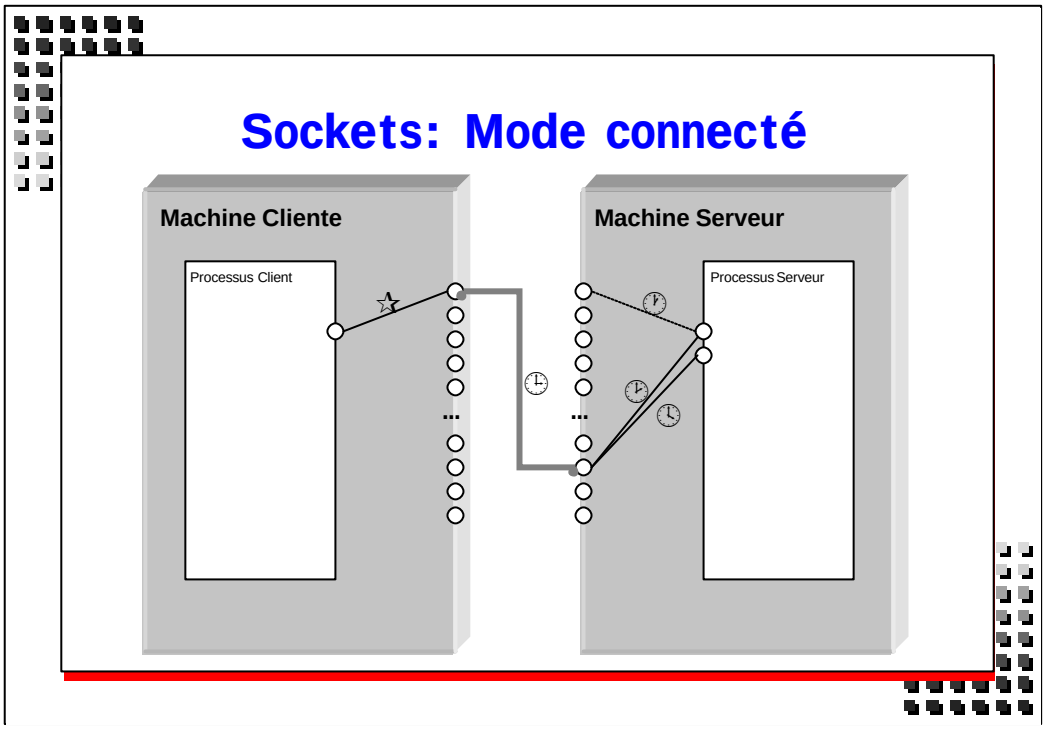


Serveur

1. Création d'une socket
2. Lien de la socket à l'adresse du serveur (N° IP, port)
3. Mise en veille de la socket
4. Acceptation d'une connexion et création d'une socket de transfert (l'@ du client est rendue en retour)
5. Dialogue et traitement
6. Fermeture de la socket esclave et attente d'une nouvelle connexion

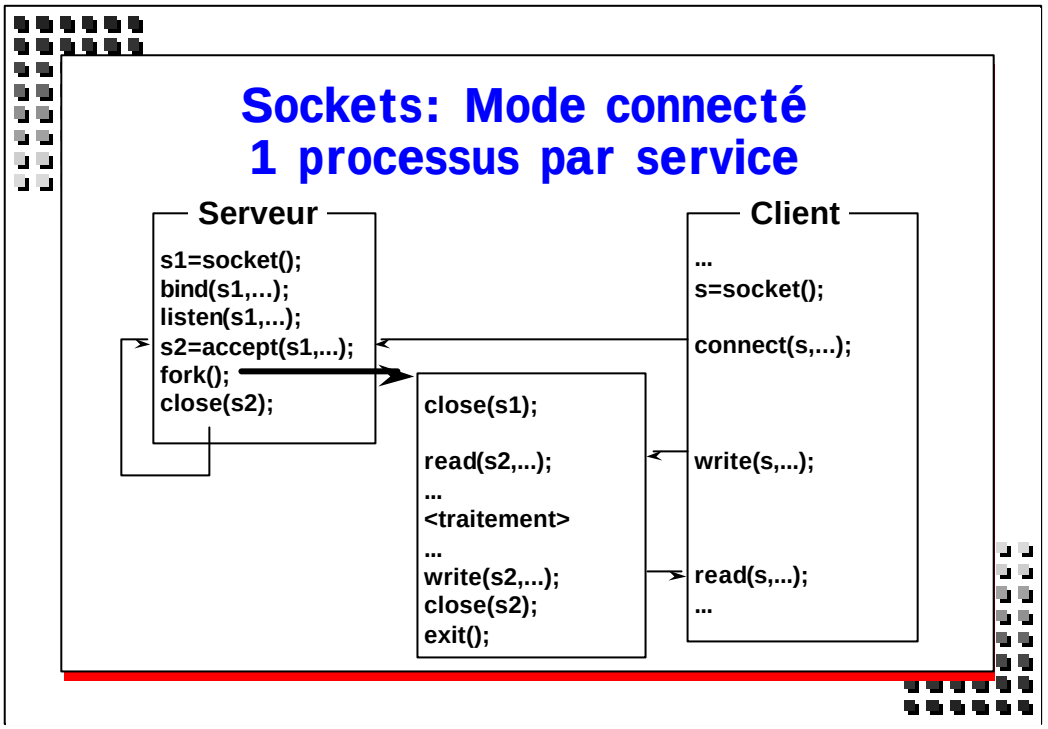
Client

1. Création de la socket
2. Connexion de la socket au serveur (N° IP, port)
3. Dialogue avec le serveur
4. Fermeture de la connexion
5. Suite du programme


Sockets: Mode connecté
Machine Cliente
Machine Serveur
Processus Client
Processus Serveur

Sockets: Mode connecté
Parralèlisation

- **Principe :**
  - Héritage des ressources lors de la création de processus.
- **Mise en œuvre "1 processus serveur par service"**
  - Le processus veilleur attend les connections, crée un processus fils qui hérite de la socket de communication.
  - Le processus fils (serveur) gère la connexion et sert la requête.
  - Le processus père (veilleur) se met en attente d'une nouvelle connexion.
- **Mise en œuvre "Pool de processus serveurs"**
  - La socket est initialisée, puis les processus du pool sont créés et hérite de cette socket.
  - Chacun se met en attente de connexions sur la socket, et traite les requêtes qui lui sont transmises.

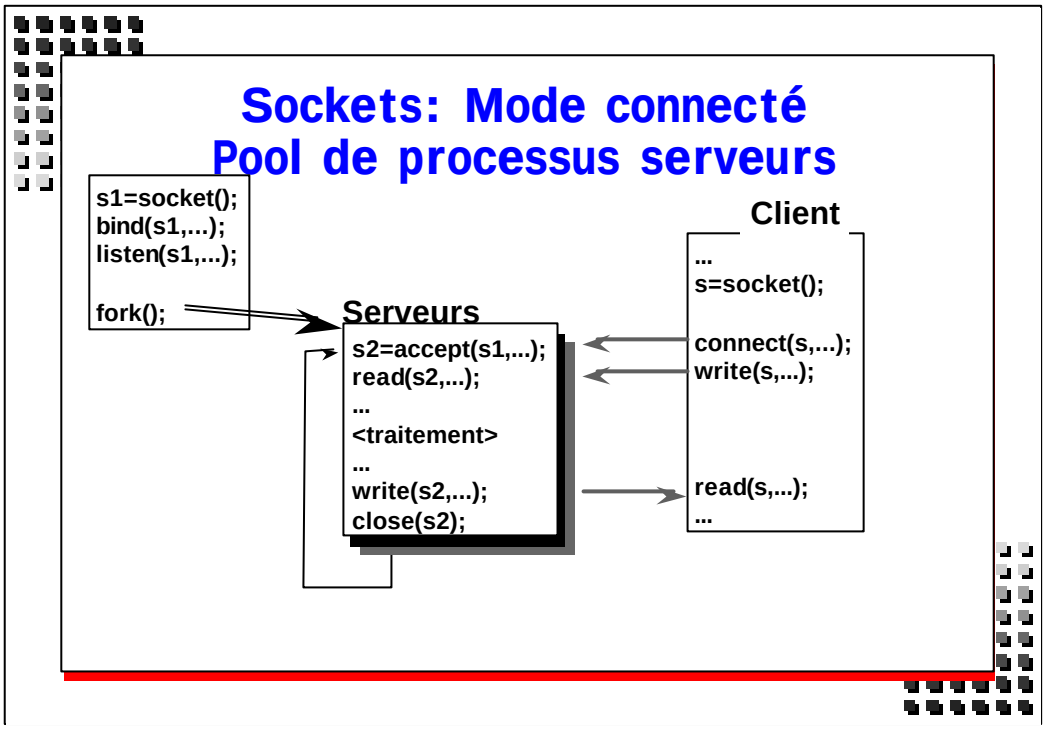


Serveur

1. Création d'une socket
2. Lien de la socket à l'adresse du serveur (N° IP, port)
3. Mise en veille de la socket
4. Acceptation d'une connexion et création d'une socket de communication (l'@ du client est rendue en retour)
5. Création d'un processus pour gérer le dialogue et le traitement
6. Fermeture de la socket esclave et attente d'une nouvelle connexion

Client

1. Création de la socket
2. Connexion de la socket au serveur (N° IP, port)
3. Dialogue avec le serveur
4. Fermeture de la connexion
5. Suite du programme



Initialisation

1. Création d'une socket
2. Lien de la socket à l'adresse du serveur (N° IP, port)
3. Mise en veille de la socket et création des processus serveurs

Serveurs

4. Acceptation d'une connexion et création d'une socket de communication (l'@ du client est rendue en retour)
5. Gestion du dialogue et traitement
6. Fermeture de la socket esclave et attente d'une nouvelle connexion

Client

1. Création de la socket
2. Connexion de la socket au serveur (N° IP, port)
3. Dialogue avec le serveur
4. Fermeture de la connexion
5. Suite du programme

Java et Les Sockets

- **Mode non connecté:**
  - Classe java.net.DatagramSocket
  - Transmission de messages : java.net.DatagramPacket
- **Mode connecté:**
  - Classes java.net.ServerSocket et java.net.Socket
  - Etablissement d'un flot bi-directionnel entre les 2 sockets connectées :
    - Interfaces java.io.InputStream, java.io. OutputStream

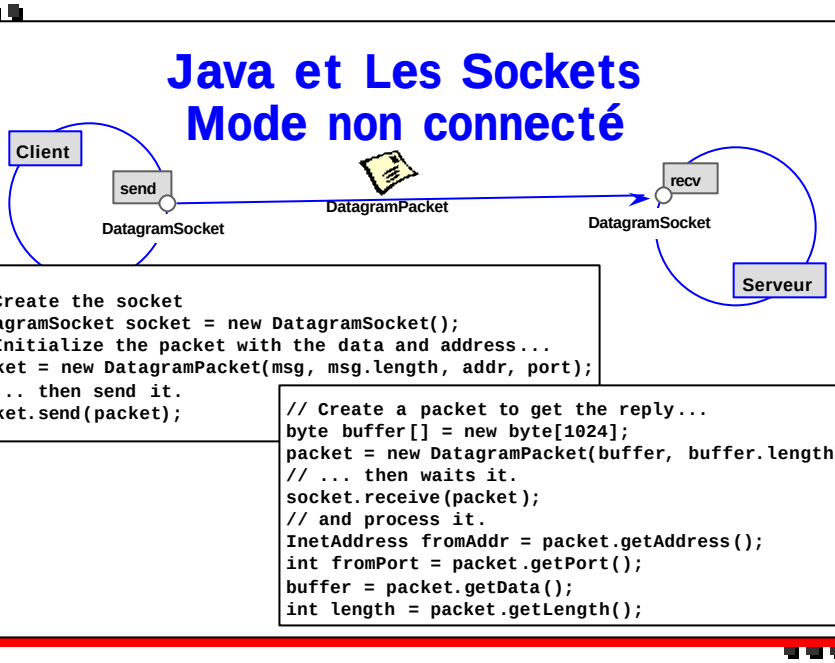
Java et Les Sockets
Mode non connecté

- **Classe DatagramSocket**
  - Constructeur
    - DatagramSocket()
    - DatagramSocket(int port)
  - Emission / Réception
    - send(DatagramPacket p)
      - Le message (contenu et taille) ainsi que l'adresse IP et le port du destinataire sont précisés dans le DatagramPacket.
    - recv(DatagramPacket p)
      - Le buffer alloué pour la réception du message ainsi que sa taille doivent être contenus dans le DatagramPacket lors de l'appel. Lors du retour le contenu et la taille du message sont disponibles, ainsi que l'adresse IP et le port de l'émetteur.

Java et Les Sockets
Mode non connecté

- **Classe DatagramPacket**
  - DatagramPacket(byte[] buf, int length, InetAddress addr, int port)
  - DatagramPacket(byte[] buf, int length)
  - InetAddress GetAddress()
  - int getPort()
  - byte[] getData()
  - int getLength()
  - ! Attention à la réutilisation des DatagramPacket (champs length).

Java et Les Sockets
Mode non connecté




```
// Create the socket
DatagramSocket socket = new DatagramSocket();
// Initialize the packet with the data and address...
packet = new DatagramPacket(msg, msg.length, addr, port);
// ... then send it.
socket.send(packet);
```

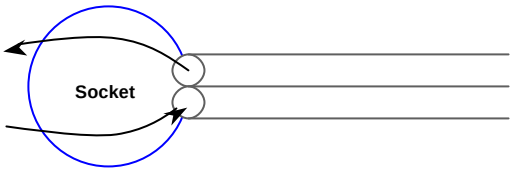
```
// Create a packet to get the reply...
byte buffer[] = new byte[1024];
packet = new DatagramPacket(buffer, buffer.length);
// ... then waits it.
socket.receive(packet);
// and process it.
InetAddress fromAddr = packet.getAddress();
int fromPort = packet.getPort();
buffer = packet.getData();
int length = packet.getLength();
```

Java et Les Sockets
Mode connecté

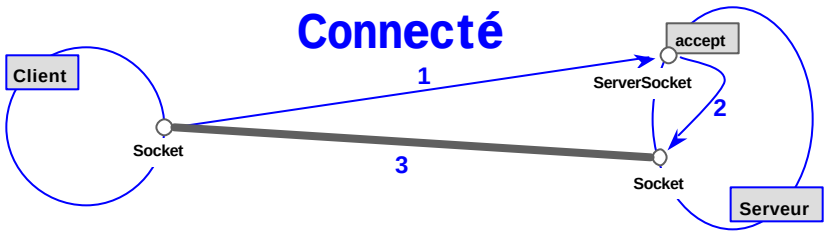
- **Classe ServerSocket**
  - Socket en attente de connexion (cf. listen)
  - ServerSocket(int port)
  - Socket accept()
    - Attend une connexion et l'accepte.
    - Retourne un socket connecté avec l'émetteur

Java et Les Sockets
Mode connecté

- **Classe Socket**
  - Objet permettant la gestion d'une connexion.
  - InputStream getInputStream()
  - OutputStream getOutputStream()



Java et Les Sockets
Connecté



Serveur

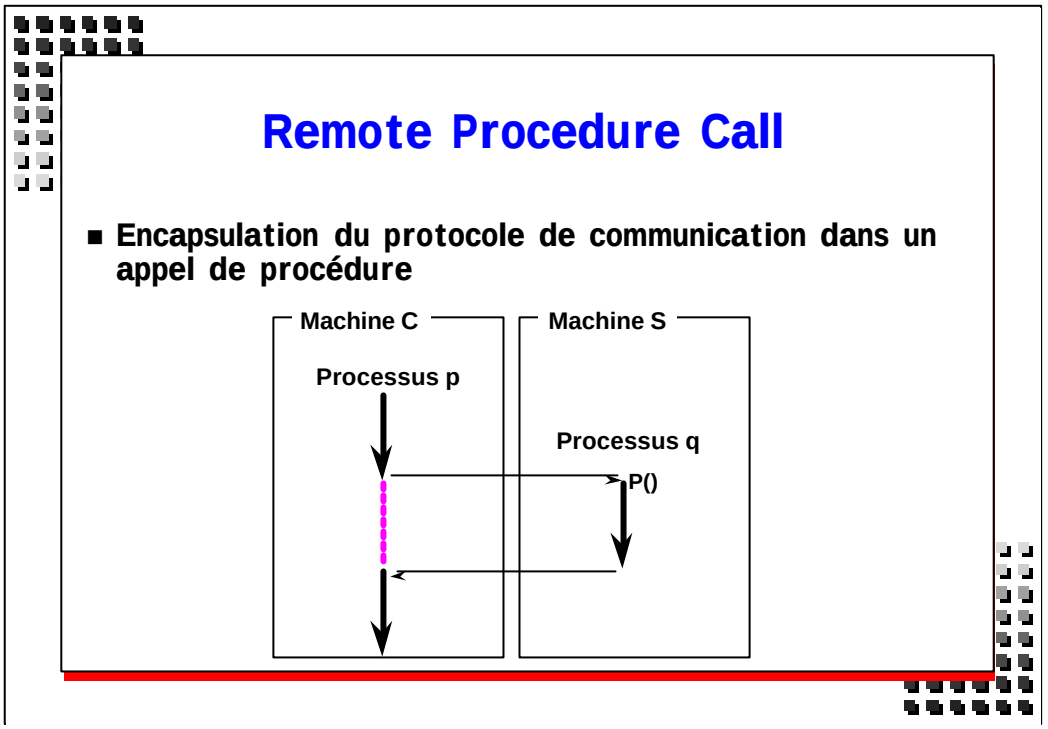
```
ServerSocket server = new ServerSocket(port);
Socket socket = server.accept();
InputStream is = socket.getInputStream();
OutputStream os = socket.getOutputStream();
```

Client

```
Socket socket = new Socket(addr, port);
InputStream is = socket.getInputStream();
OutputStream os = socket.getOutputStream();
```

Appel de procédure à distance
Remote Procedure Call

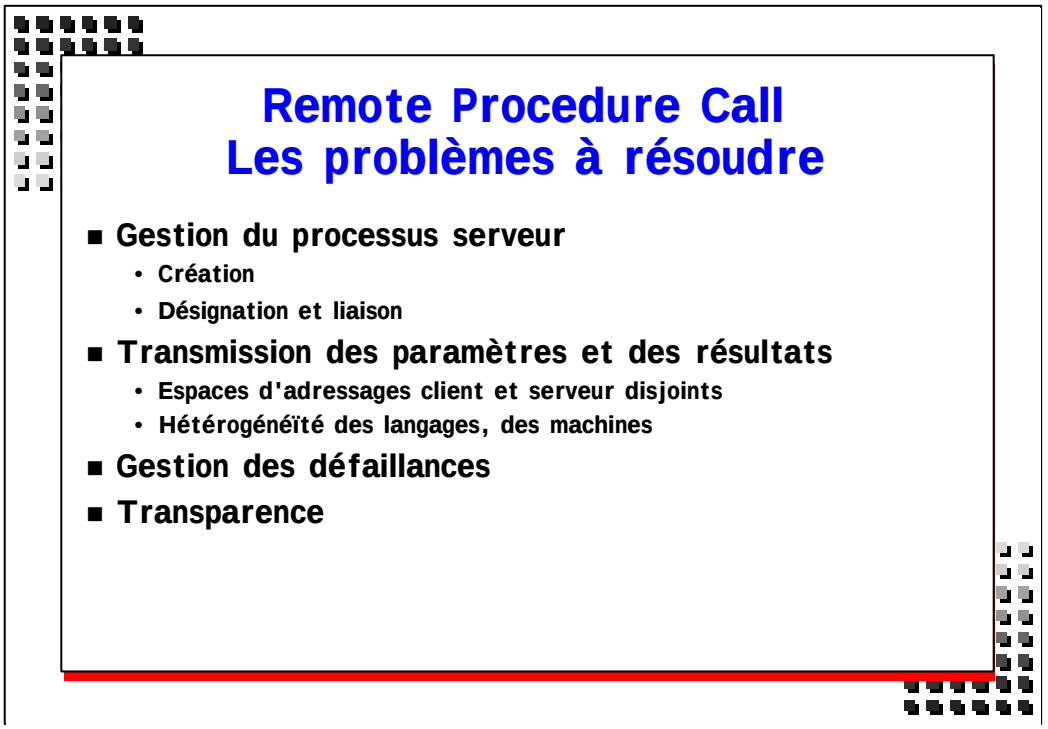
- **Outils de base pour réaliser le mode client-serveur.**
  - **L'opération à réaliser est présentée sous la forme d'une procédure que le client peut faire exécuter à distance par un autre site : le serveur.**
    - **Forme et effet identique à ceux d'un appel local**
      - **Simplicité (en l'absence de pannes)**
      - **Sémantique identique à celle de l'appel local**
  - **Opérations de base**
    - **Client**
      - **doOp (IN Port serverId, Name opName, Msg *arg, OUT Msg *result)**
    - **Serveur**
      - **getRequest (OUT Port clientId, Message *callMessage)**
      - **sendReply (IN Port clientId, Message *replyMessage)**



L'appel de procédure à distance (RPC) est un outil de base pour la construction d'applications réparties. Etant donné un processus p et une procédure P, l'appel de procédure à distance est un mécanisme qui permet à p depuis le site C d'exécuter la procédure P sur le site S avec un effet global identique à celui qui serait obtenu par l'exécution locale de P. L'intérêt de ce mécanisme est de transposer à un système réparti une construction dont la sémantique est bien connue.

Le mécanisme utilisé repose sur l'utilisation d'un processus serveur pour exécuter la procédure P sur le site S. Le processus client reste bloqué pendant l'exécution de P, et il est réactivé lors du retour de P.

Le serveur attend les requêtes, les exécute puis renvoie les résultats.



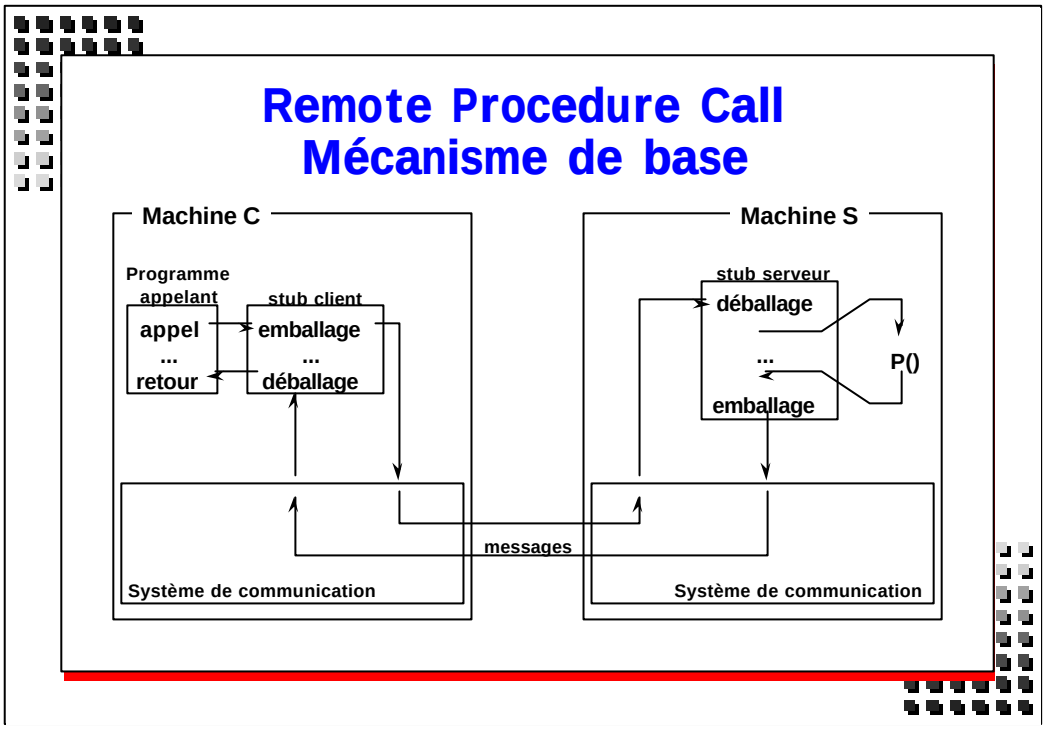
Il faut savoir gérer la création et l'identification du serveur sur le site appelé.

Les processus client et serveur s'exécutant dans des espaces d'adressage disjoints, il faut réaliser la transmission des paramètres à l'appel, et des résultats au retour en utilisant les outils de communication par message.

Les programmes du client et du serveur peuvent être écrits dans des langages différents, et s'exécuter sur des machines dont les représentations des données sont différentes. Il faut donc prévoir un mécanisme de conversion des paramètres et des résultats.

La probabilité des défaillances (site appelant, site appelé, système de communication, etc.) n'est pas négligeable, il faut donc spécifier le comportement de l'appel en cas de défaillances.

Le mécanisme de RPC doit donner, autant que possible, l'illusion d'un appel de procédure classique.



Le principe de réalisation de l'appel de procédure à distance repose sur le principe de transparence: aussi bien pour le processus client que pour le serveur, tout doit se passer comme si l'appel était local. A cet effet on introduit deux procédures appelées talons (stubs). Chacun des talons doit être lié au programme client ou serveur correspondant.

Le talon client sur le site appelant fournit une interface identique à la procédure appelée. Les fonctions exécutées par le stub client sont:

• mise en forme des paramètres d'appel pour permettre leur transport sur le site appelé (marshalling).

• envoi sur le site appelé d'un message contenant l'identité de la procédure appelée et les paramètres.

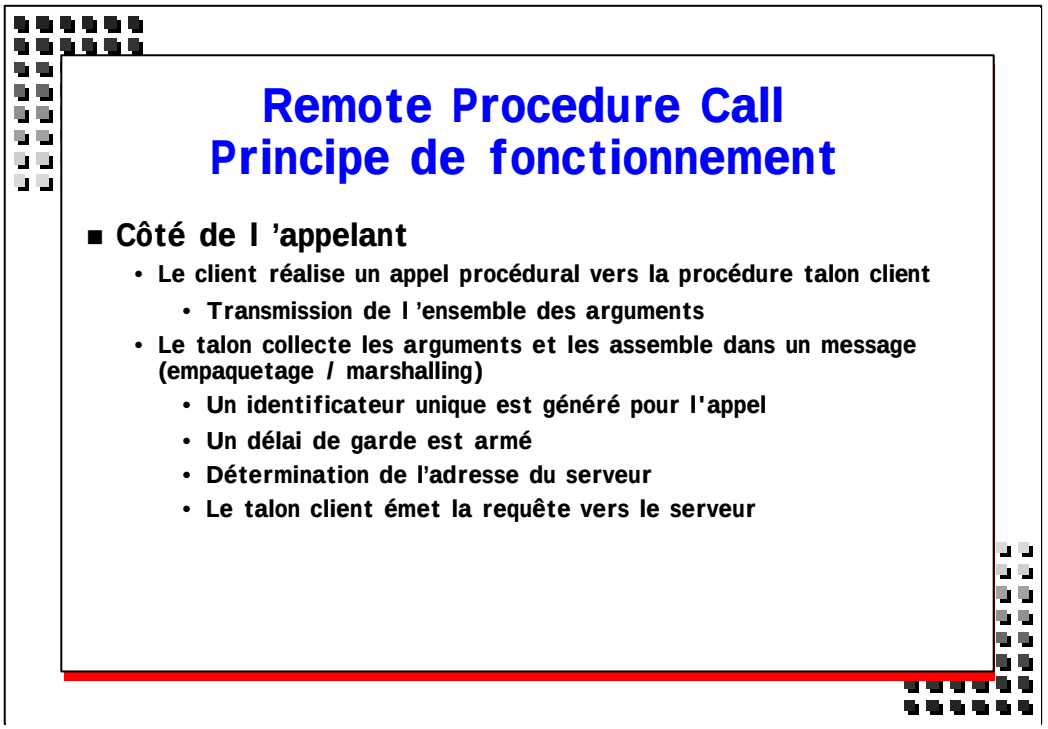
• attente du message de réponse.

• extraction des résultats (unmarshalling) et retour de la procédure.

Le talon serveur est exécuté par le processus serveur sur le site appelé, il exécute les fonctions suivantes:

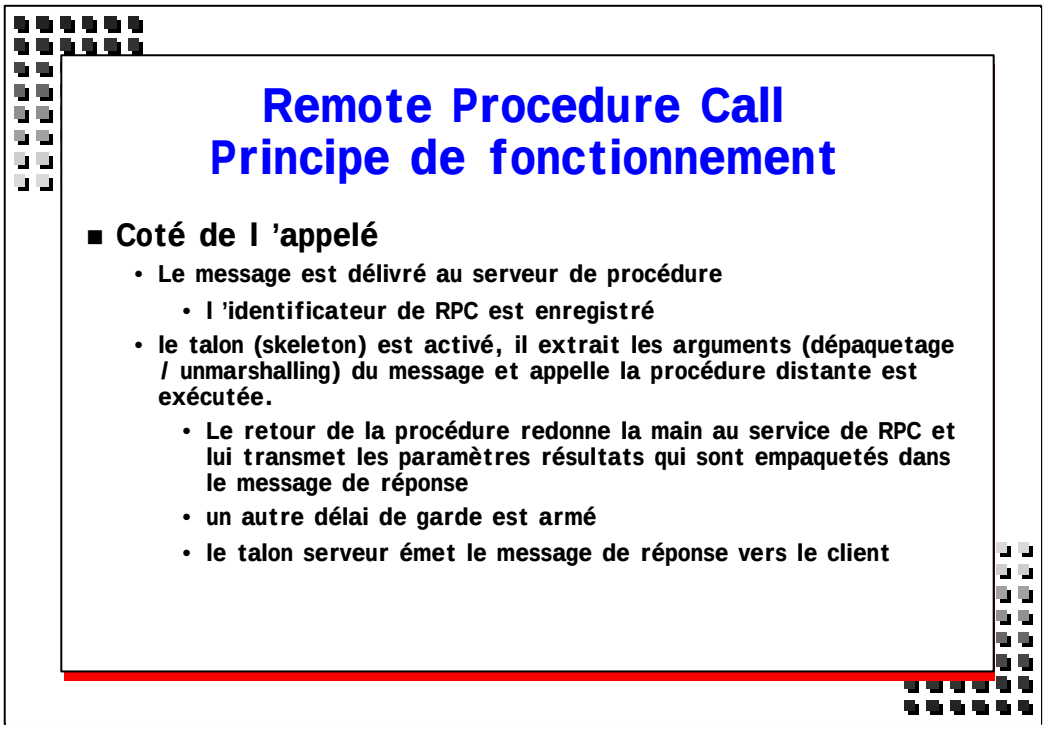
• réception du message, détermination de la procédure appelée, déballage des paramètres et appel de la procédure (appel local).

• au retour de l'appel, emballage des résultats et envoi du message de retour au site appelant.



Le client a l'illusion d'appeler localement la procédure, il appelle en réalité le talon client qui s'est substitué à celle-ci ; ce dernier va gérer le dialogue avec le serveur :

- Création du message d'appel :
  - Identification de la procédure à exécuter sur le serveur.
  - Identificateur unique de l'appel afin de permettre au serveur de distinguer une nouvelle requête d'un message réémis.
  - Ensemble des données nécessaires à cette exécution (avec prise en compte de l'hétérogénéité).
- Mise en place d'un délai de garde :
  - Réémission du message d'appel en cas de perte.
  - Ce délai de garde doit être suffisant pour permettre aux messages d'appel et de retour de transiter, et à la procédure distante de s'exécuter.
- Détermination de l'adresse du serveur (@IP et n° de port par exemple) et émission du message.

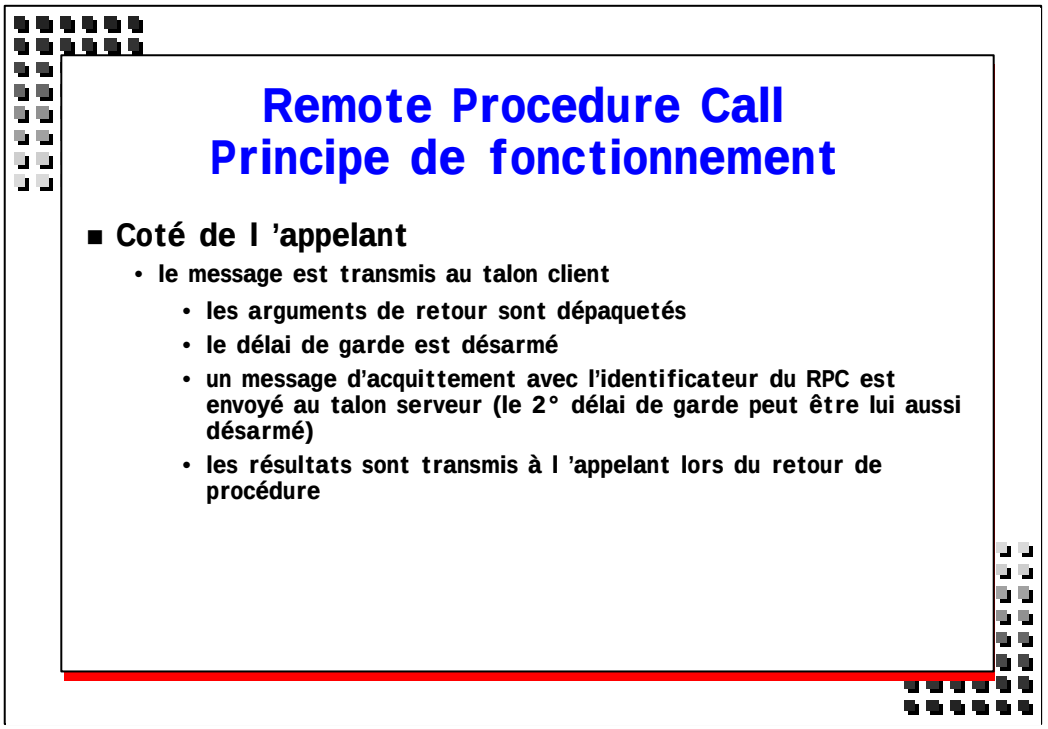


Le serveur reçoit le message :

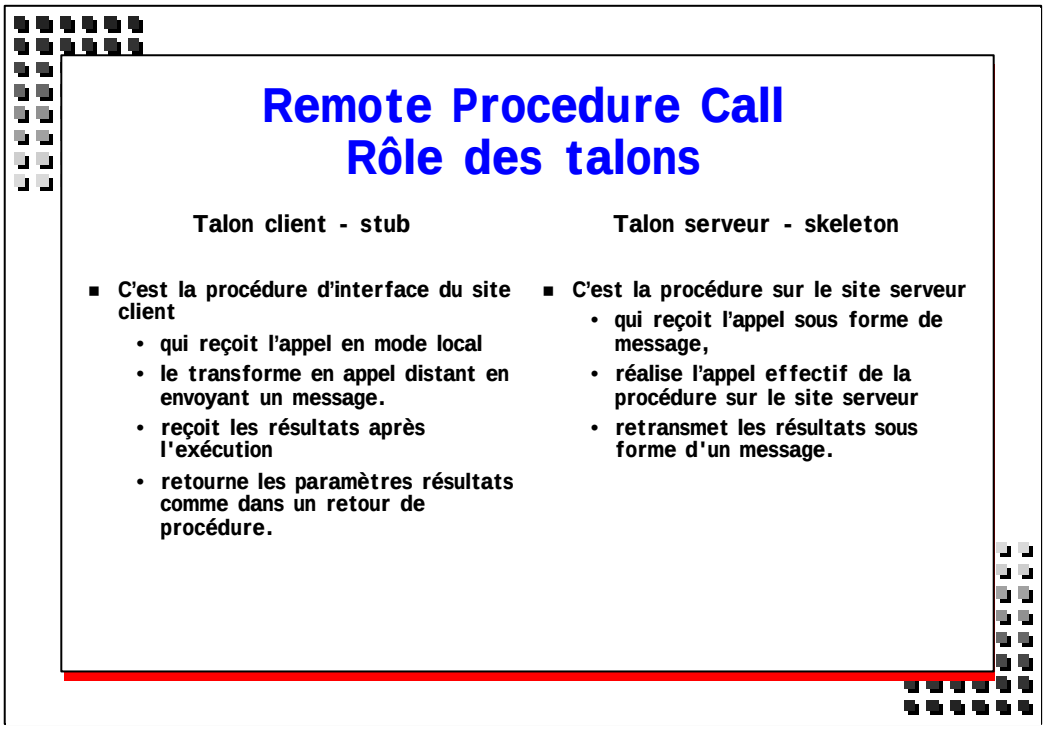
• Il enregistre son identification unique afin de ne pas exécuter de multiples fois la procédure en cas de réémission de message.

• Il identifie la procédure à appeler et il active le talon serveur correspondant.

Le talon serveur extrait les paramètres du message et réalise l'appel à la procédure. Lors de la terminaison de la procédure, il récupère les paramètres résultats et construit le message de réponse.



Le talon client extrait les paramètres résultats du message de retour afin de les rendre à la procédure cliente (celle-ci aura l'illusion que l'appel de procédure s'est effectué localement), il désarme le délai de garde maintenant inutile et enfin il envoie un message d'acquittement au serveur afin que celui-ci désarme le second délai de garde.



Les talons client et serveurs sont spécifiques ; à chaque procédure correspond une paire stub/skeleton.

Le stub (talon client) possède exactement la même interface que la procédure distante ce qui lui permet de se substituer à celle-ci dans le code du programme client.

Le skeleton (talon serveur) connaît l'interface de la procédure, il peut ainsi réaliser l'appel à partir des informations contenues dans le message.

Chaque talon connaît donc le contenu des messages d'appel et de retour qu'il peut ainsi construire et/ou interpréter.

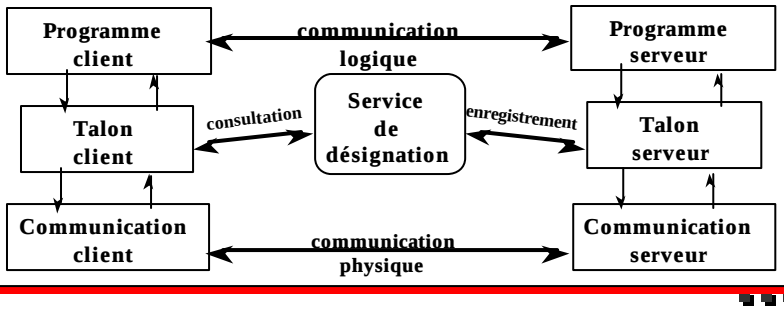
TODO: exemple de code "simpliste" de stubs/skeletons.

Remote Procedure Call
Désignation

- **Objets à désigner**
  - Le site d'exécution, le serveur, la procédure, etc.
- **Désignation**
  - Désignation globale indépendante de la localisation
    - possibilité de reconfiguration des services (pannes, régulation de charge, …)
  - statique ou dynamique
    - statique : localisation du serveur est connue à la compilation
    - dynamique : déterminée à l'exécution, objectifs :
      - séparer connaissance du nom du service de la sélection de la procédure qui va l'exécuter
      - permettre l'implémentation retardée

Remote Procedure Call
Liaison et fonctionnement

- **Liaison (détermination de l'adresse du serveur)**
  - Liaison statique (résolution lors de la compilation)
  - Liaison au premier appel (appel au serveur de nom)
  - Liaison à chaque appel (appel du serveur de nom)



Remote Procedure Call
Gestion des défaillances

- **Entités en œuvre:**
  - Site client
  - Site serveur
  - Système de communication
- **Pannes possibles:**
  - perte du message client ® serveur
  - panne du serveur durant l'appel
  - perte du message serveur ® client
  - panne du client entre l'appel et le retour

Remote Procedure Call
Gestion des défaillances

- **Perte du message d'appel:**
  - **Solution:**
    - **Acquittement des messages et réémission**
  - **Problème:**
    - **Exécution multiples**
  - **Sémantique d'appel:**
    - **Exécution au moins une fois (procédure idempotente)**
    - **Exécution au plus une fois**
  - **Solution:**
    - **Attribution d'une id. unique à chaque appel**

Remote Procedure Call
Gestion des défaillances

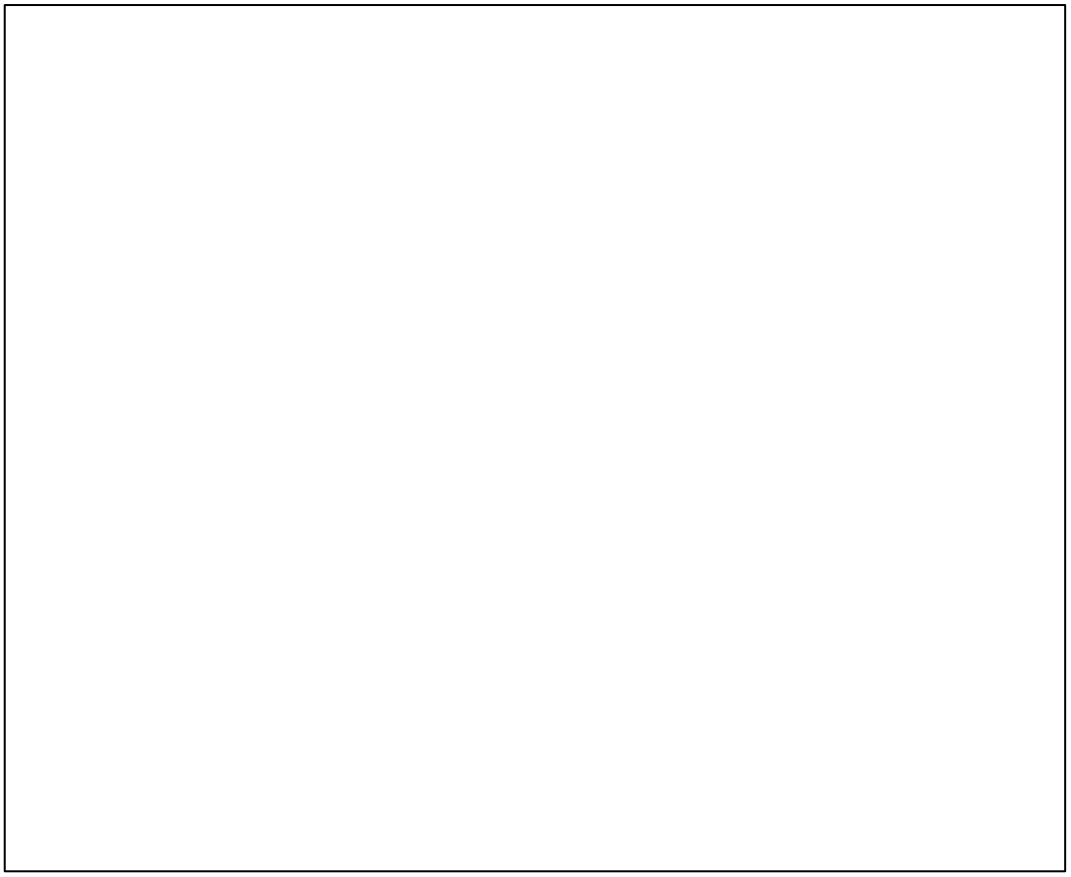
- **Défaillance du serveur:**
  - Réception du résultat ® ok.
  - Délai de garde ➔ 0 ou 1 exécution
    - Panne du serveur
    - Perte du message de retour
  - Exécution au moins une fois ➔ redondance du serveur
- **Perte du message de retour:**
  - Acquittement des messages et réémission
- **Panne du client:**
  - Processus serveur orphelin
- **Optimisation: acquittement retardé (piggybacking)**

Remote Procedure Call
Intégration dans un langage

- **Approche non transparente**
  - l'appel distant est syntaxiquement différent d 'un appel local
    - a_type := remotecall(proc, <args>, <one, maybe>, timeout)
- **Approche transparente**
  - construction automatique de l 'appel distant
    - du coté du client
      - l 'appel à la procédure est remplacé par un appel au talon client (stub) généré par le compilateur
    - du coté du serveur
      - le compilateur produit un talon serveur (skeleton) capable de recevoir l 'appel et de le rediriger vers la procédure
  - Un outil : les langages de définition d 'interface (IDL)

Remote Procedure Call Sémantique

- **Implémentations ayant des sémantiques variées.**
  - retrouver la sémantique habituelle de l'appel de procédure
  - sans se préoccuper de la localisation de la procédure
- **Objectifs difficiles à atteindre**
  - réalisation peu conviviale
  - sémantique différente de l'appel de procédure
    - même en l'absence de panne
    - exemple : passage des paramètres par référence



Remote Procedure Call
Passage de paramètres

- Caractères (ascii/ebcdic), flottants
- Entiers (ordre des octets):
  - little endian

00	00	01	FF

  - big endian

FF	01	00	00

- Structures:
  - Problèmes des pointeurs
  - Problèmes d'alignement

```
struct x {
    short s;
    char c[6];
    int i;
};
```

xx	xx		
a	b	c	d
e	\0		
xx	xx	xx	xx

xx	xx	a	b
c	d	e	\0
xx	xx	xx	xx

Remote Procedure Call
Passage des Paramètres

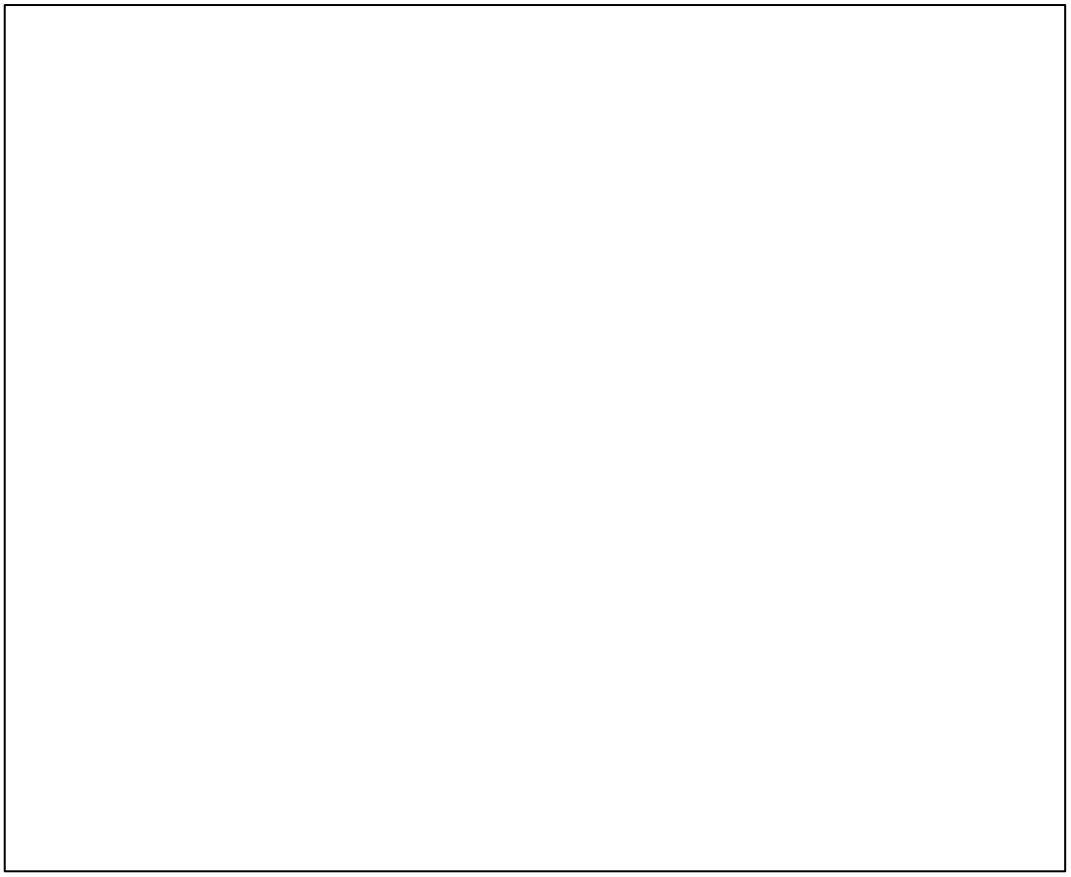
- **Langages classiques (C, C++) :**
  - Valeur, références (par adresse), résultat
  - Référence : utilise une adresse mémoire centrale du site de l'appelant… aucun sens pour l'appelé
- **RPC : passage par valeur, émulation des autres modes**
  - Référence -> valeur, résultat
    - Violation de la sémantique du passage par référence

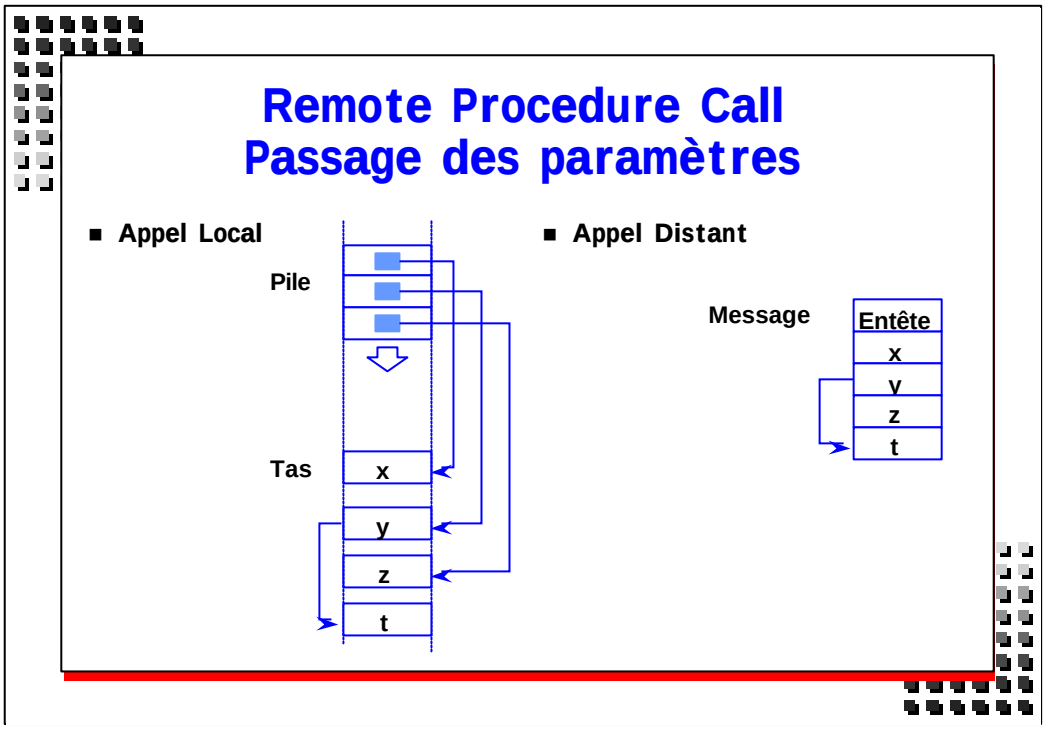
- exemple de pb :
  - a := 0
  - double_incr (a, a)
  - résultat : a = 1 ou 2 ?

```
procédure double_incr (x, y) {
    x := x+1;
    y := y+1;
}
```

Remote Procedure Call
Passage des paramètres

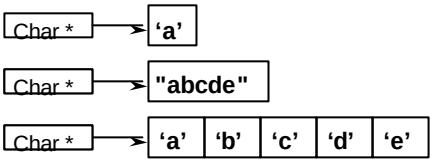
- **Utilisation d 'un IDL**
  - DCE, CORBA, Java RMI…
  - Reconstruction de l'état de la mémoire du client sur le serveur
    - IN : passage par valeur (aller)
    - OUT : passage par valeur (retour)
    - IN-OUT : passage par copie-restauration
  - ATTENTION : n 'est pas équivalent au passage par référence




Remote Procedure Call
Passage des paramètres
Appel Local
Pile
Tas
x
y
z
t
Appel Distant
Message
Entête
x
y
z
t

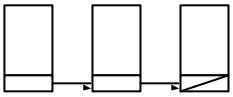
Remote Procedure Call
Passage des Paramètres

- **Chaines de caractères :**

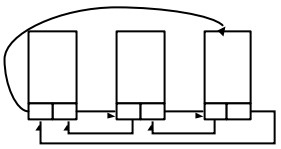


Remote Procedure Call
Passage des Paramètres

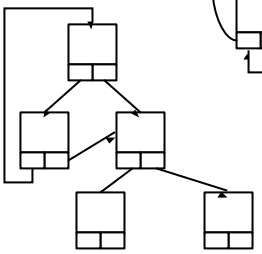
- **Structures contenant des adresses :**
  - **Liste**



  - **Liste doublement chainée**

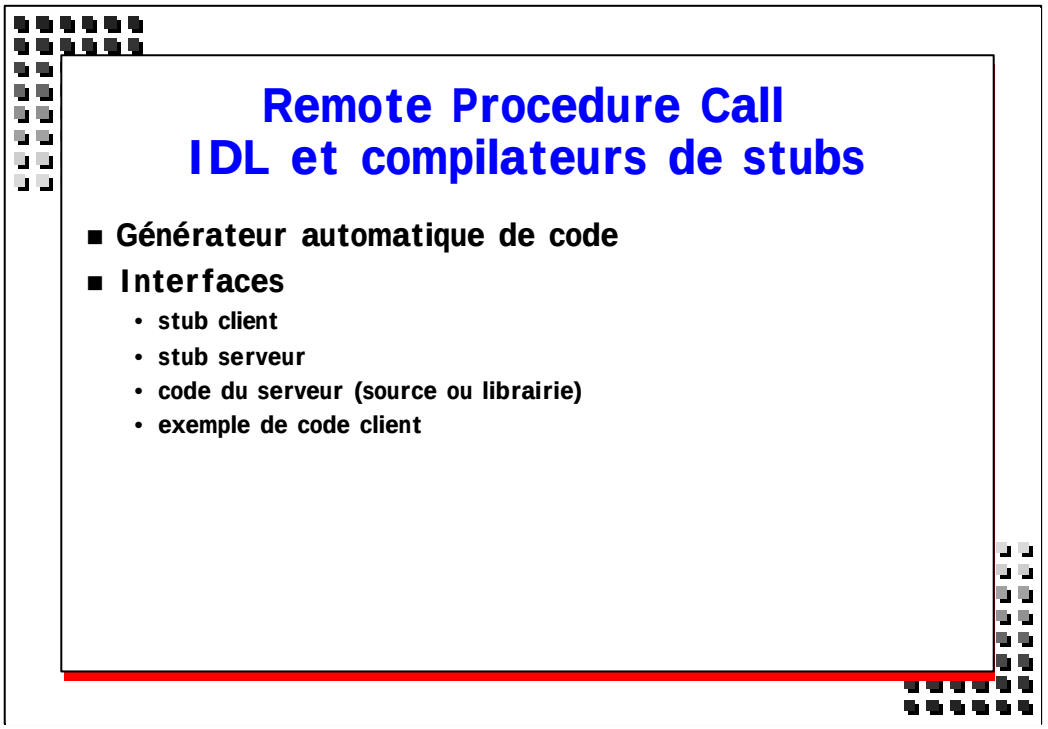


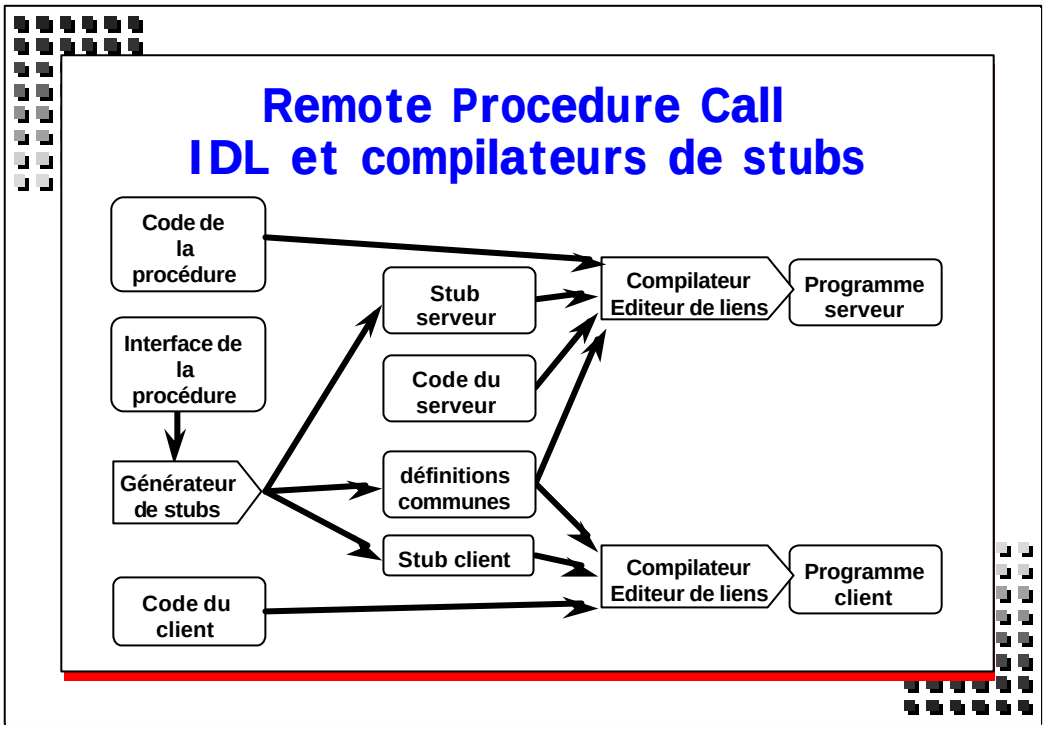
  - **Graphe...**

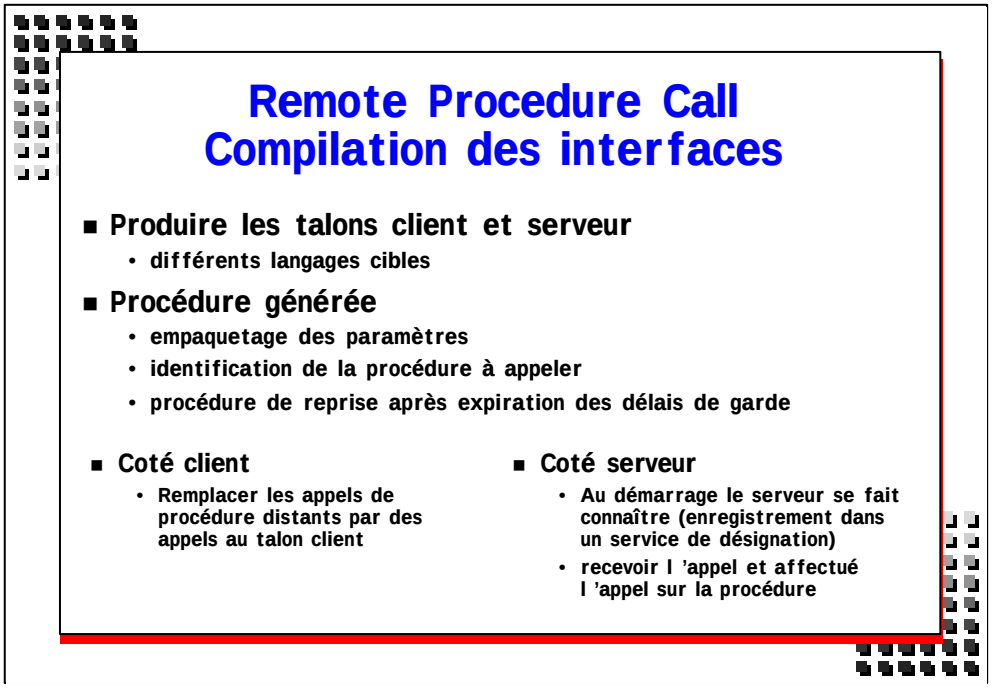


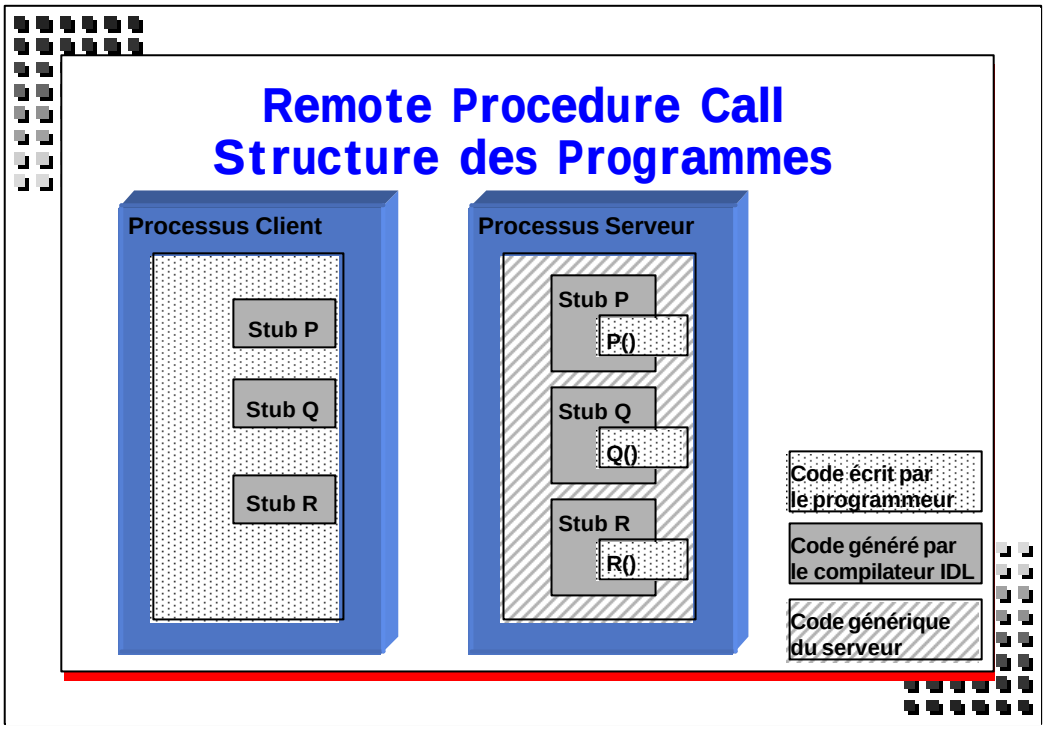
Remote Procedure Call
IDL : spécification des interfaces

- **Utilisation d'un langage**
  - Spécification commune au client et au serveur
  - Définition des paramètres (types et natures : IN, OUT, IN-OUT)
- **Avantage**
  - Traitement de l'hétérogénéité
  - Types de données différents selon les langages
    - Représentations internes différentes selon les systèmes
    - Définition des types indépendante de la représentation
  - Description d'une interface
    - Description des types élémentaires (arguments, résultats, exceptions, etc.)
    - Procédures de conversion pour types complexes (avec pointeurs)


Remote Procedure Call
IDL et compilateurs de stubs
Générateur automatique de code
Interfaces
stub client
stub serveur
code du serveur (source ou librairie)
exemple de code client


Remote Procedure Call
IDL et compilateurs de stubs
Code de la procédure
Interface de la procédure
Générateur de stubs
Code du client
Stub serveur
Code du serveur
définitions communes
Stub client
Compilateur Editeur de liens
Programme serveur
Compilateur Editeur de liens
Programme client


Remote Procedure Call
Compilation des interfaces
Produire les talons client et serveur
différents langages cibles
Procédure générée
empaquetage des paramètres
identification de la procédure à appeler
procédure de reprise après expiration des délais de garde
Coté client
Remplacer les appels de procédure distants par des appels au talon client
Coté serveur
Au démarrage le serveur se fait connaître (enregistrement dans un service de désignation)
recevoir l 'appel et affectué l 'appel sur la procédure

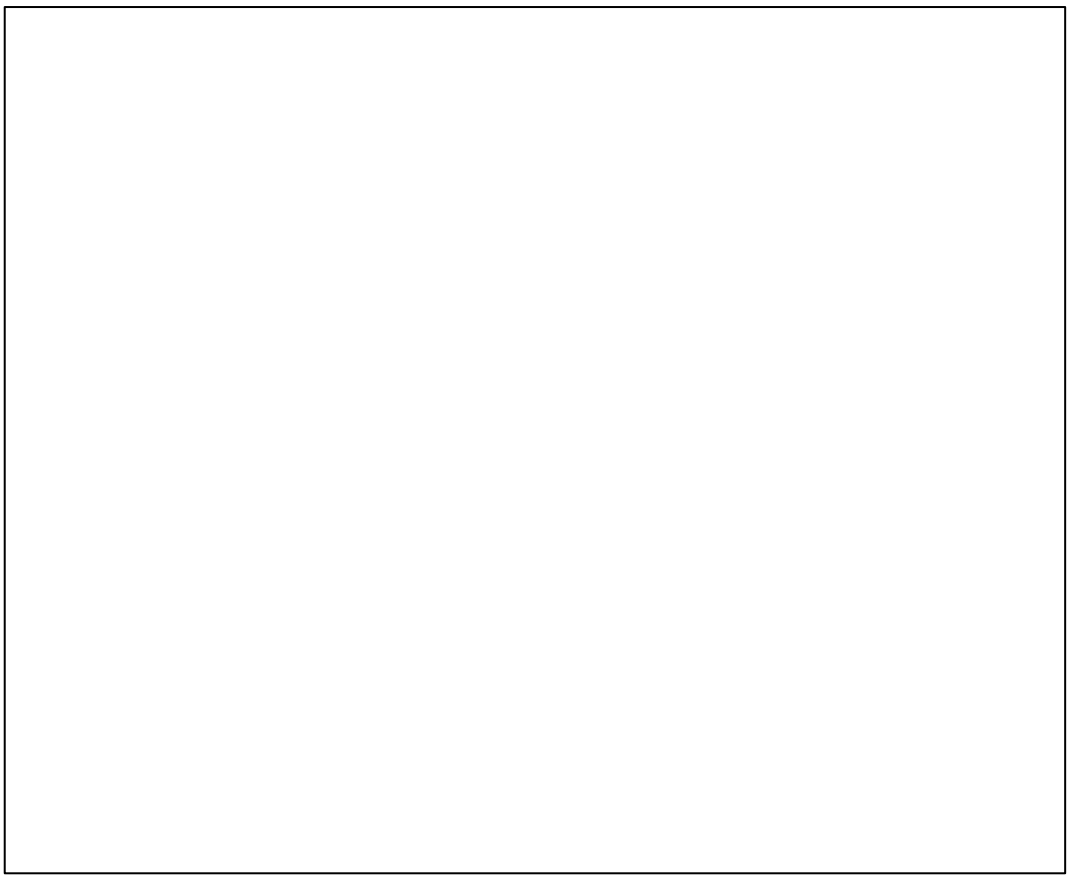

Remote Procedure Call
Structure des Programmes
Processus Client
Stub P
Stub Q
Stub R
Processus Serveur
Stub P
P()
Stub Q
Q()
Stub R
R()
Code écrit par le programmeur
Code généré par le compilateur IDL
Code générique du serveur

Remote Procedure Call
Appel Asynchrone

- **Le client poursuit son exécution après l'émission du message d 'appel**
  - **la procédure distante s'exécute en parallèle avec la poursuite du client et retourne les paramètres résultats en fin de son exécution**
  - **le client récupère les résultats quand il en a besoin**

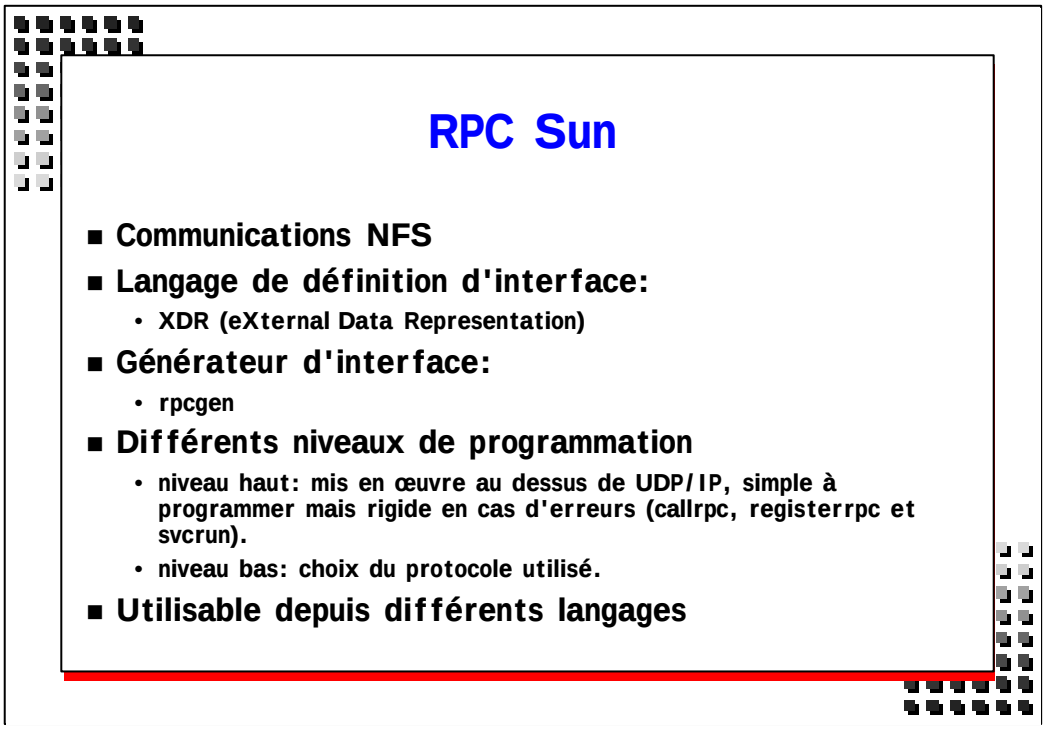
  \+ **parallélisme plus important**

  – **le client ne retrouve pas la sémantique de l'appel de procédure**



Remote Procedure Call
Appel asynchrone avec futur

- **Futur**
  - objet particulier pour la récupération des résultats
  - futur explicite :
    - construction avant l'appel de l'objet dans lequel les résultats seront déposé
  - futur implicite :
    - c'est le mécanisme d'appel qui construit les objets résultats
  - mode d 'utilisation :
    - la lecture rend un résultat nul si le résultat n'est pas disponible
    - la lecture bloque le client si le résultat n'est pas disponible


RPC Sun
Communications NFS
Langage de définition d'interface:
XDR (eXternal Data Representation)
Générateur d'interface:
rpcgen
Différents niveaux de programmation
niveau haut: mis en œuvre au dessus de UDP/IP, simple à programmer mais rigide en cas d'erreurs (callrpc, registerrpc et svcrun).
niveau bas: choix du protocole utilisé.
Utilisable depuis différents langages

RPC Sun

- Un serveur exécute un programme qui exporte plusieurs procédures
- Chaque procédure est identifiée par un triplet d'entiers (32bits):
  - n° de programme
  - n° de version
  - n° de procédure
- Unicité des n° de programmes:
  - 0x0 → 0x1F FF FF FF attribués par Sun
  - 0x20 00 00 00 → 0x3F FF FF FF utilisateurs
  - 0x40 00 00 00 → 0x5F FF FF FF allocation dynamique
- Id de la machine du serveur → utilisateur

RPC Sun: Limitations

- UDP ➔ limite de 8Ko de données
- 1 paramètre d'appel et 1 paramètre de retour
  - Utilisation de structures
  - rpcgen
- Sémantique en cas de panne: au moins un
- Pas de facilités pour écrire des serveurs multiplexés
- L'utilisateur peut fournir ses propres routines de conversion XDR ➔ paramètres quelconques

RPC Sun
Interface niveau haut

- **callrpc**
  - id machine
  - id procédure (n° prog., n° version, n° proc.)
  - type et valeur du paramètre d'appel
  - type et valeur du paramètre de retour
- **registerrpc**
  - id procédure (n° prog., n° version, n° proc.)
  - adresse de la procédure
  - type du paramètre d'appel et de retour
- **svc_run**
  - boucle d'attente des requêtes

RPC Sun
Interface de haut niveau

- **Programme client**

```
main(int argc, char *argv[])
{
        unsigned long cpt;
        int ret;

        if ((ret = callrpc(argv[1],
                           NBPROG, NBVERS, NBPROC,
                           xdr_void, 0,
                           xdr_u_long, &cpt)) != 0) {
                clnt_perrno(ret);
                exit(1);
        }
        printf("compteur = %d sur %s \n", cpt, argv[1]);
        exit(0);
}
```

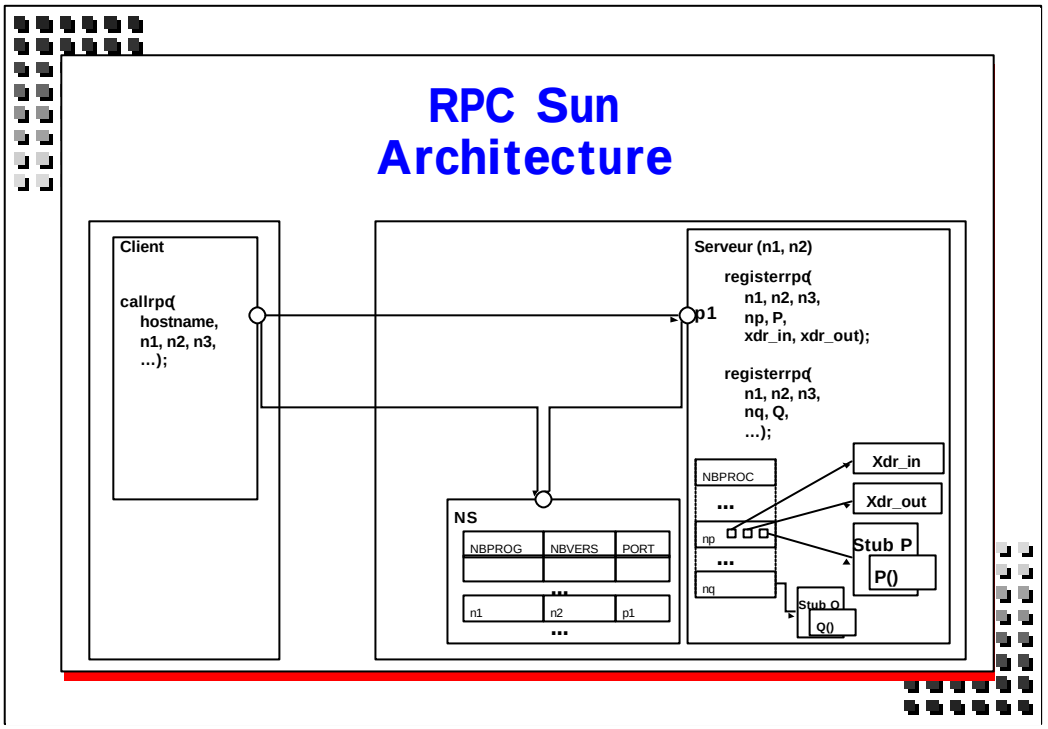
RPC Sun
Interface de haut niveau

- **Programme serveur**

```
int cpt=0;

main()
{
        registerrpc(NBPROG, NBVERS, NBPROC, IncCpt, xdr_void, xdr_u_long);
        svc_run();
        exit(1);
}

char *
IncCpt(char * ParIn)
{
        cpt += 1;
        return ((char *) &cpt);
}
```


RPC Sun
Architecture
Client
callrpc(
hostname,
n1, n2, n3,
...);
Serveur (n1, n2)
registerrpc(
n1, n2, n3,
np, P,
xdr_in, xdr_out);
registerrpc(
n1, n2, n3,
nq, Q,
...);
p1
NS
NBPROG
NBVERS
PORT
n1
n2
p1
NBPROC
np
nq
Xdr_in
Xdr_out
Stub P
P()
Stub Q
Q()

RPC Sun: XDR

- Sérialisation/désérialisation
- Conversion des types de base
  - void, char, short, int, etc.
- Types construits:
  - string: bool_t xdr_string(XDR xdrs, char **s, u_int imax)
  - bytes: bool_t xdr_bytes(XDR xdrs, char **s, u_int imax)
  - array, opaque, union, reference, etc.

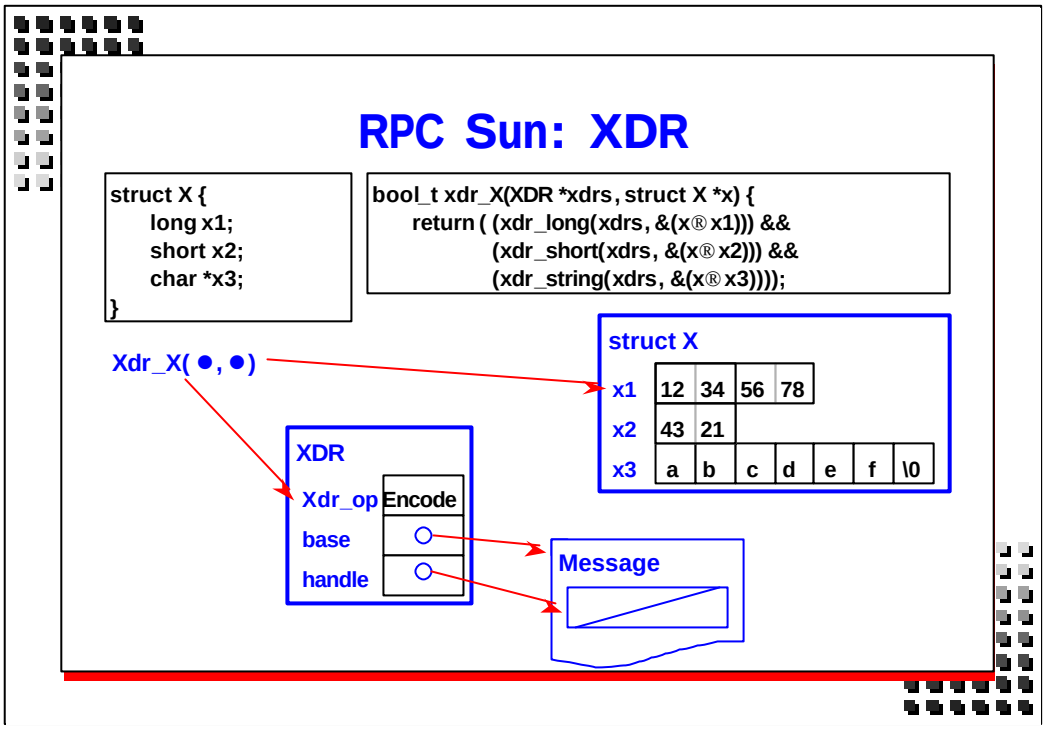
RPC Sun: XDR

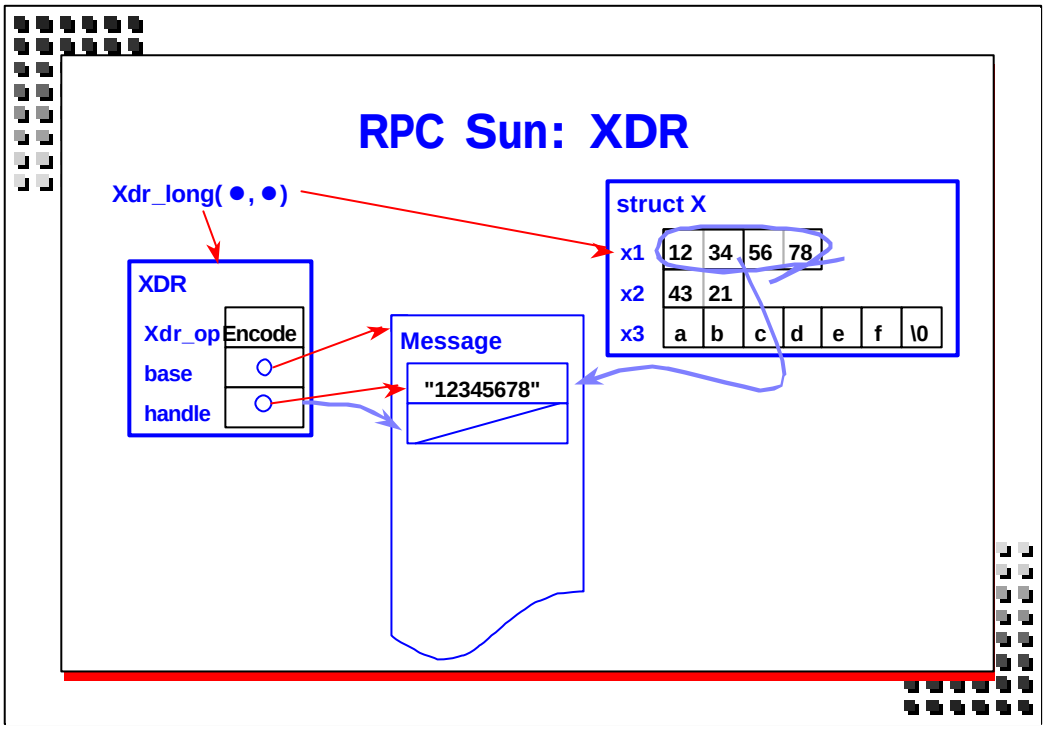
- **Exemple d'extension:**

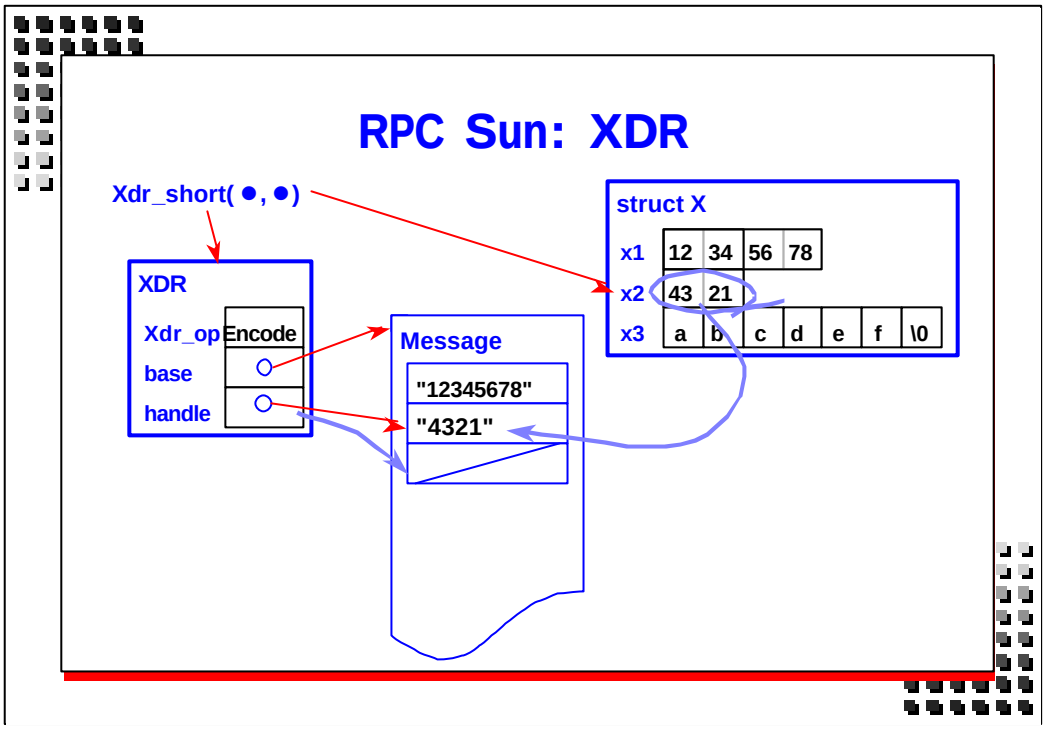
```
struct deuxlong {
        long x1, x2;
};
xdr_deuxlong(XDR *xdrs, struct deuxlong *x)
{
        return (xdr_long(xdrs, &x->x1) && xdr_long(xdrs, &x->x2));
}
```

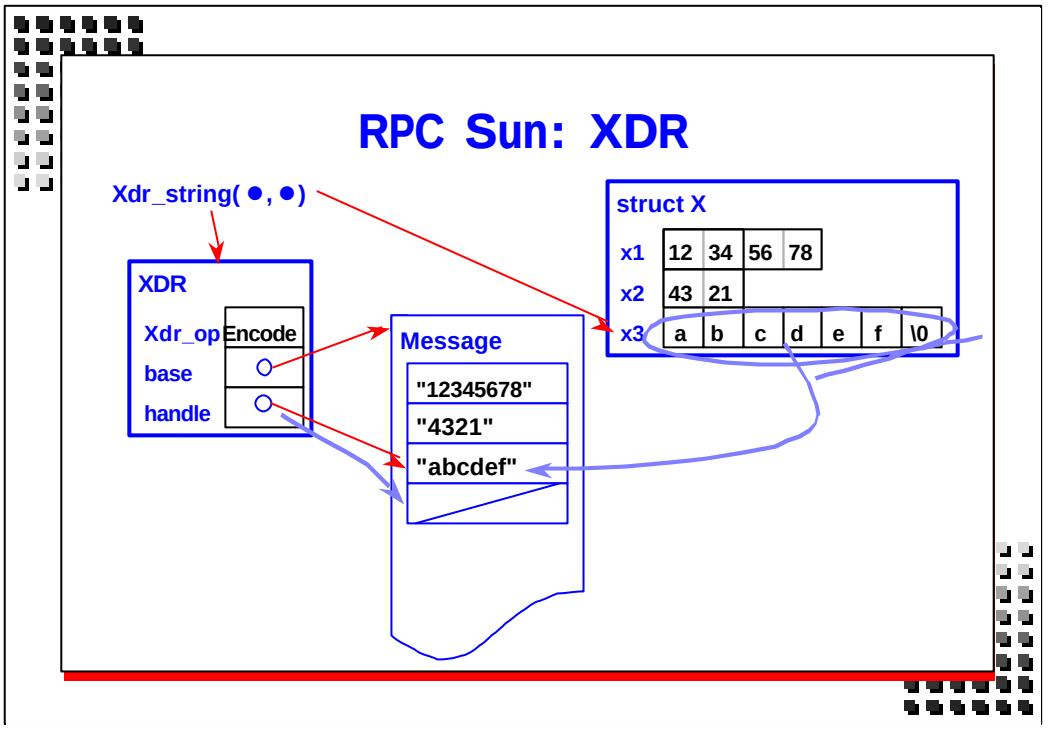
```
struct X {
        struct deuxlong s1;
        long x3;
};
```

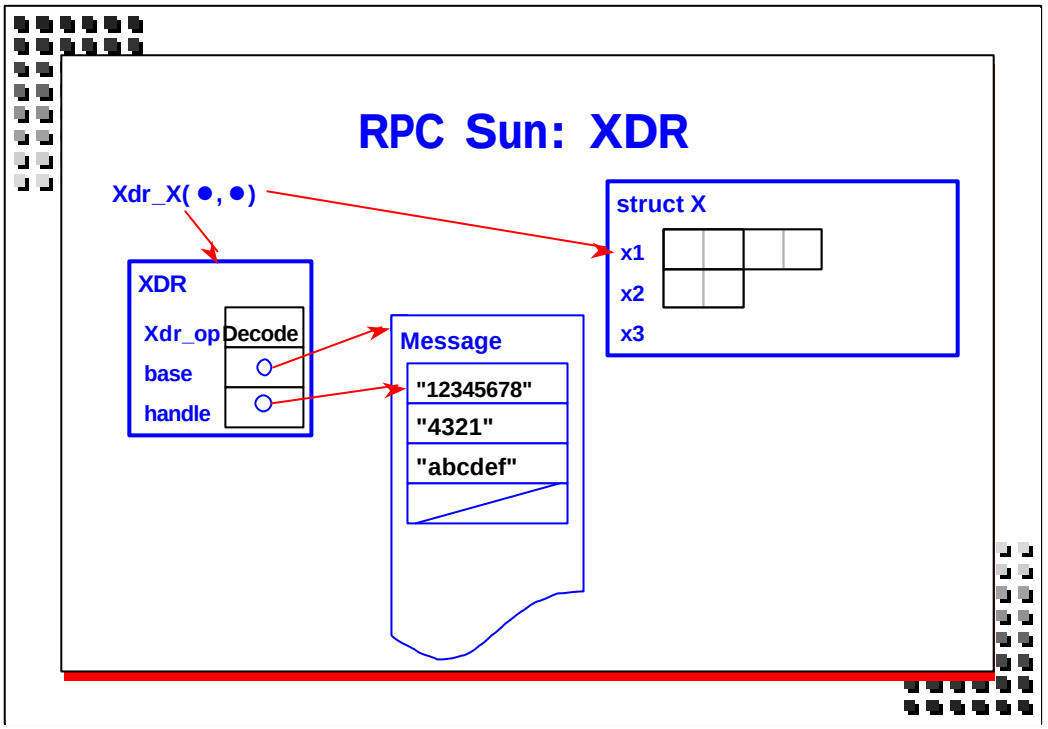
```
struct Y {
        long x1, x2, x3;
};
```

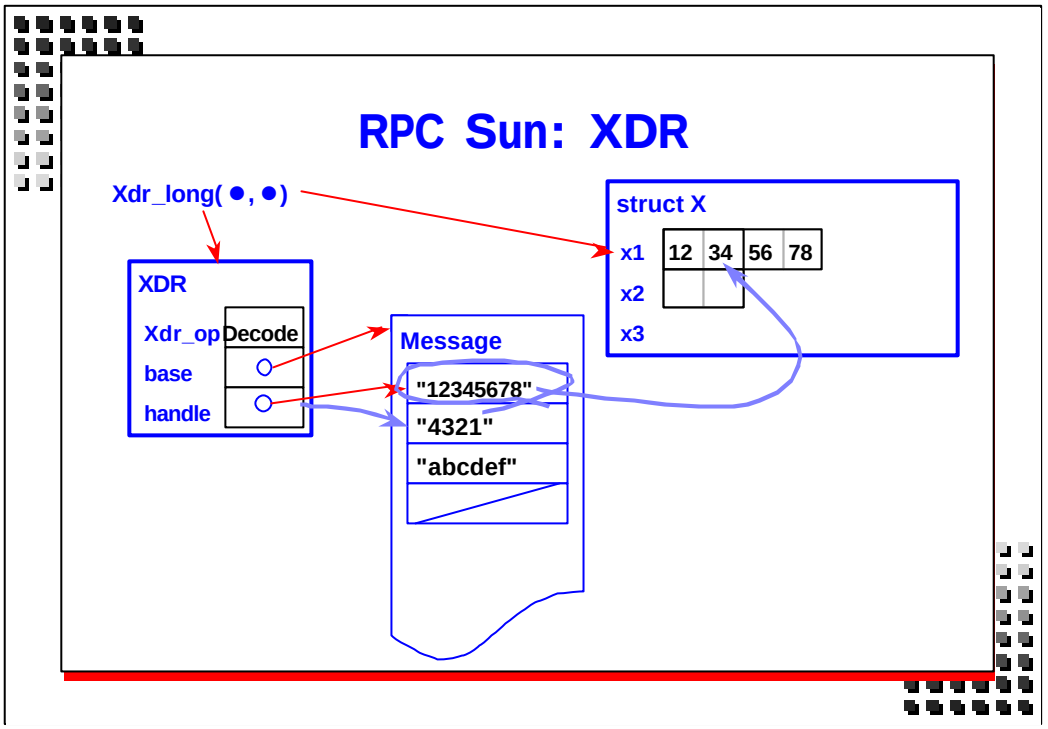

RPC Sun: XDR
struct X {
long x1;
short x2;
char *x3;
}
bool_t xdr_X(XDR *xdrs, struct X *x) {
return ((xdr_long(xdrs, &(x® x1))) &&
(xdr_short(xdrs, &(x® x2))) &&
(xdr_string(xdrs, &(x® x3))));
Xdr_X(●, ●)
struct X
x1
12
34
56
78
x2
43
21
x3
a
b
c
d
e
f
\0
XDR
Xdr_op
Encode
base
handle
Message

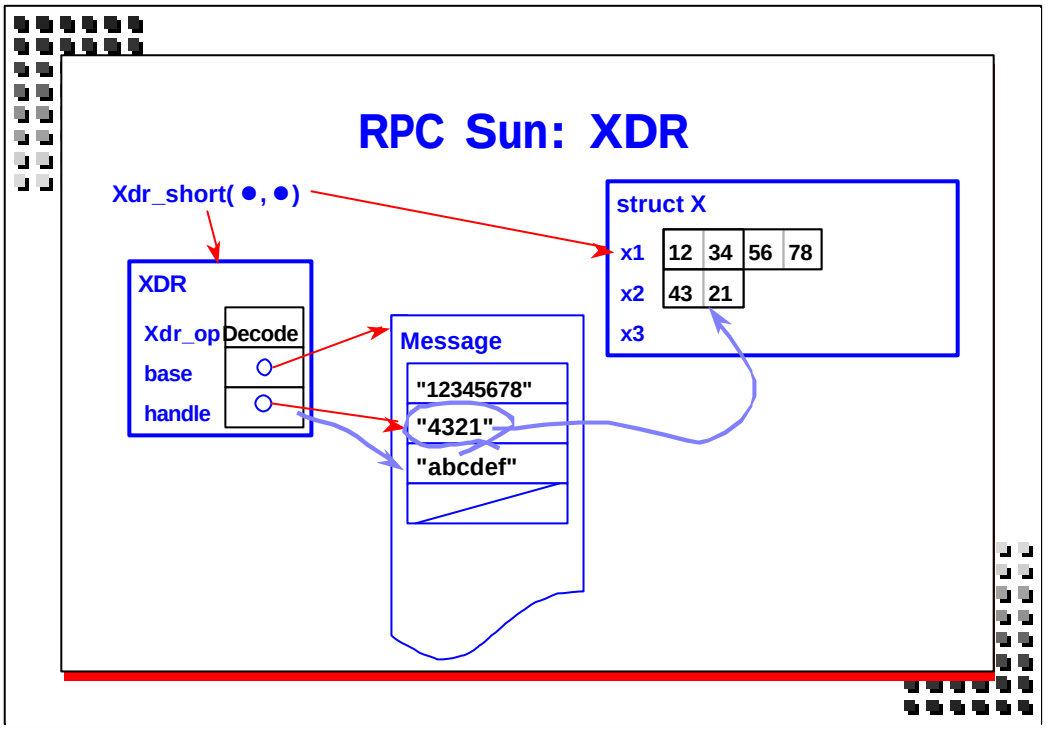

RPC Sun: XDR
Xdr_long(●, ●)
XDR
Xdr_op
Encode
base
handle
Message
"12345678"
struct X
x1
12
34
56
78
x2
43
21
x3
a
b
c
d
e
f
\0

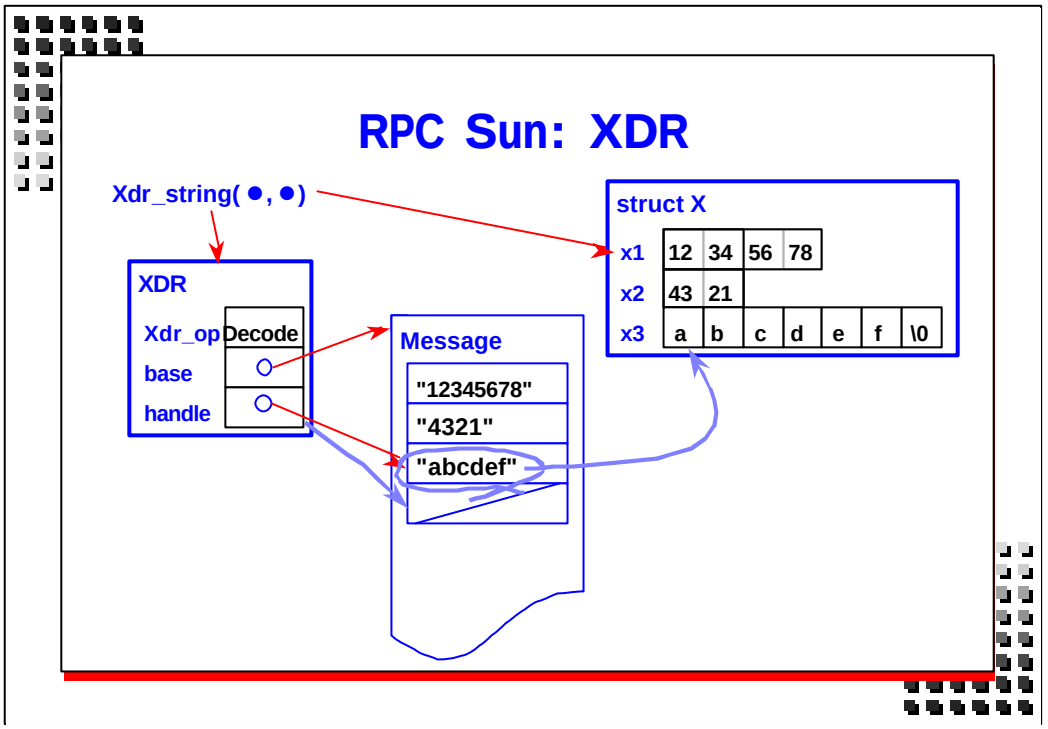

RPC Sun: XDR
Xdr_short(●, ●)
XDR
Xdr_op
Encode
base
handle
Message
"12345678"
"4321"
struct X
x1
12
34
56
78
x2
43
21
x3
a
b
c
d
e
f
\0


RPC Sun: XDR
Xdr_string(●, ●)
XDR
Xdr_op
Encode
base
handle
Message
"12345678"
"4321"
"abcdef"
struct X
x1
12
34
56
78
x2
43
21
x3
a
b
c
d
e
f
\0


RPC Sun: XDR
Xdr_X(●, ●)
struct X
x1
x2
x3
XDR
Xdr_op
Decode
base
handle
Message
"12345678"
"4321"
"abcdef"

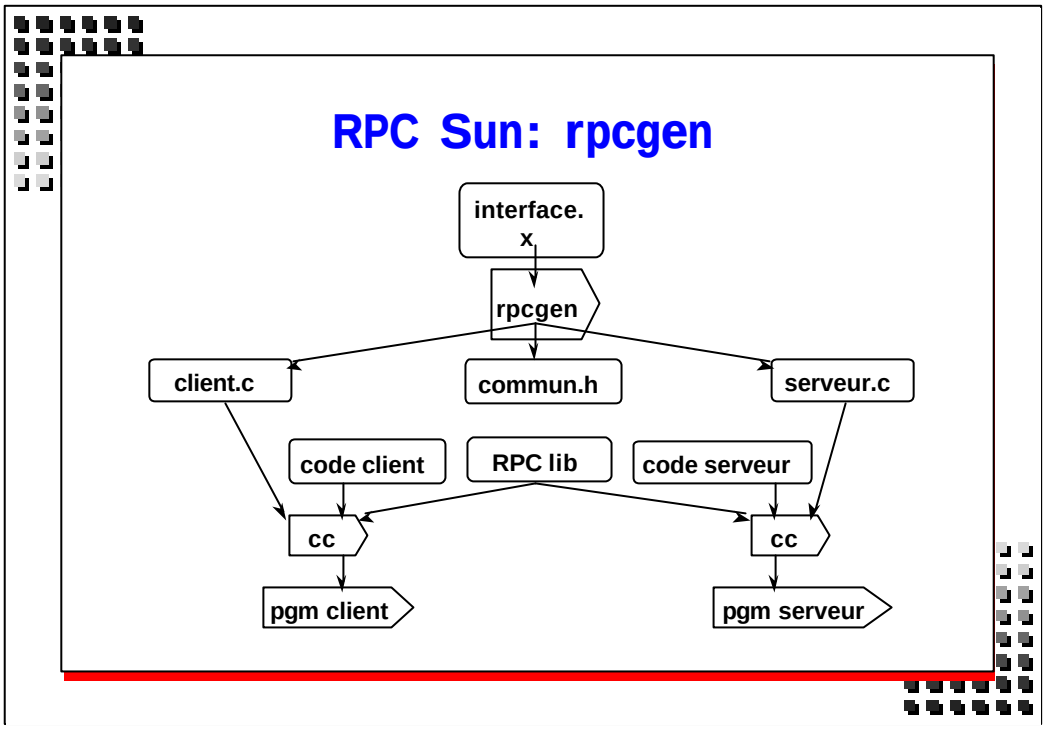

RPC Sun: XDR
Xdr_long(●, ●)
struct X
x1
12
34
56
78
x2
x3
XDR
Xdr_op
Decode
base
handle
Message
"12345678"
"4321"
"abcdef"


RPC Sun: XDR
Xdr_short(●, ●)
struct X
x1
12
34
56
78
x2
43
21
x3
XDR
Xdr_op
Decode
base
handle
Message
"12345678"
"4321"
"abcdef"


RPC Sun: XDR
Xdr_string(●, ●)
XDR
Xdr_op
Decode
base
handle
Message
"12345678"
"4321"
"abcdef"
struct X
x1
12
34
56
78
x2
43
21
x3
a
b
c
d
e
f
\0

RPC Sun: rpcgen

- Compilation des interfaces décrites en "langage RPC"
- Production de code C
  - Stub client
  - Squelette serveur
  - Procédures XDR
  - Définitions communes


RPC Sun: rpcgen
interface.
x
rpcgen
client.c
commun.h
serveur.c
code client
RPC lib
code serveur
cc
cc
pgm client
pgm serveur

RPC Sun: rpcgen

rls.x

```
program DIRPROG {
        version DIRVERS {
                string read_dir(string)=1;
        }=1;
}=0x20000001;
```

rls_client.c

```
/* exemple de programme client */
/* appel au stub client de la function read_dir*/
```

rls_server.c

```
/* framework de la function read_dir */
```

rls.h

```
#include <rpc/rpc.h>

#define DIRPROG ((unsigned long)(0x20000001))
#define DIRVERS ((unsigned long)(1))
#define read_dir ((unsigned long)(1))
```

makefile.rls

```
...
```

rls_clnt.c

```
/* stub client de la function read_dir*/
```

rls_svc.c

```
/* code générique du serveur */
```