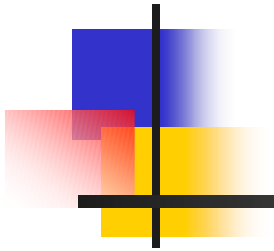


NSY107 - Intégration des systèmes client-serveur

Cours du 27/05/2006, 4 heures

© Emmanuel DESVIGNE
<emmanuel@desvigne.org>

Document sous licence libre (FDL)

Plan du cours

- TP n°1 (fin du TP précédent)
- Critiques de ce TP1
- Quelques outils système C/UNIX
- TP n°2 (en deux étapes)
- XDR (eXternal Data Representation)
- TP n°3
- Critiques de XDR
- Introduction aux RPC

TP n°1 – max. 45 min [1/1]

- Reprise du cours précédent :
  - Programmer (en C ou en JAVA) :
    - un serveur TCP, à l'écoute sur le port 8424. Attend en entrée une (et une seule) chaîne de caractères, et retourne la chaîne en ordre inverse,
    - un client TCP, qui envoie une requête au serveur précédent, récupère la réponse, et affiche le résultat

Critiques de ce TP 1 [1/1]

- Réflexions sur ces TPs :
  - Quels sont les problèmes rencontrés? Pour vous aider à les découvrire :
    - Quels sont les questions que vous vous êtes posés à la lecture du sujet ?
    - Des remarques relatives à la sécurité ?
    - Interconnectez un client développé par un groupe avec un serveur développé par d'autres. Cela fonctionne-t-il ?
    - Simulez (à l'aide de la fonction **`sleep()`**) un traitement long sur le serveur, et lancez simultanément 1, puis 2, ... puis 6 clients simultanément. Que se passe-t-il ?
    - Lancez un client avant le serveur. Que se passe-t-il ?
    - Lancez serveurs sur la même machine. Que se passe-t-il ?
    - Simulez l'arrêt du serveur avant qu'il ne réponde (fonction **`exit()`**). Comment réagit le client ?

Quelques outils système UNIX [1/6]

- Pour éviter qu'un client ou un serveur soit bloqué en attente d'un message qu'il pourrait ne jamais recevoir, il est possible de programmer un « *timeout* » en rendant la lecture du socket non bloquante par un :
  - ```
    #include <fcntl.h>
    [...] fcntl(sock, F_SETFL, O_NONBLOCK | \
                               fcntl(sock, F_GETFL, 0));
    ```
- Ou encore par un :
  - ```
    #include <sys/ioctl.h>
    [...] int on=1;
    [...] ioctl(sock, FIONBIO, &on);
    ```

Quelques outils système UNIX [2/6]

- Problème : si le socket est non bloquant, il vous faut alors l'interroger régulièrement pour savoir si vous avez reçu des choses (technique de polling). Cette technique engendre une perte de temps CPU non négligeable ➔

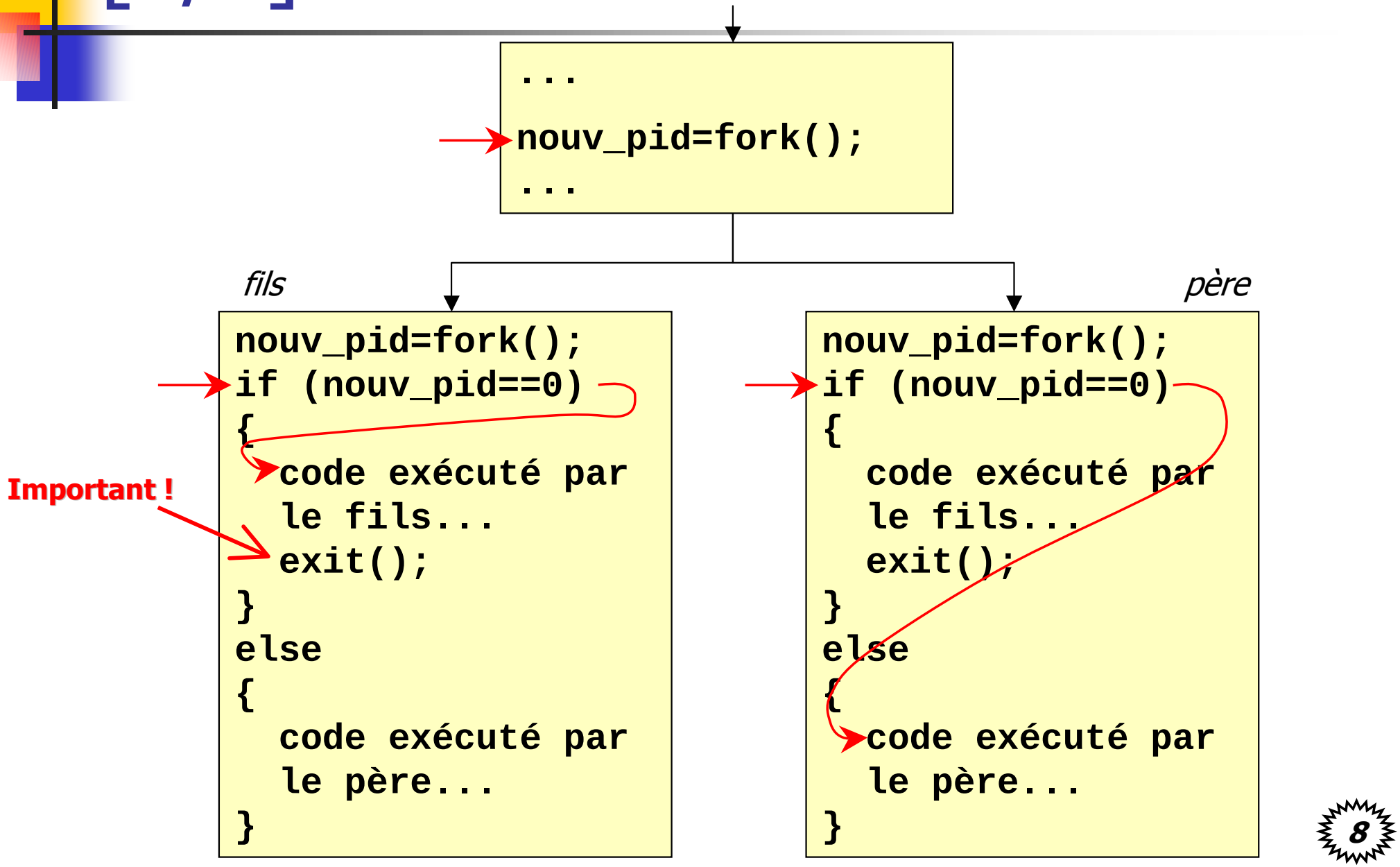
  Polling = technique à proscrire

- Solution élégante : utiliser la fonction :
  - **`select()`** (Cf. « **`man select`** »)

Quelques outils système UNIX [3/6]

- Pour éviter que le serveur soit bloqué par la requête d'un client, en général, le serveur délègue le travail à réaliser à un autre processus (i.e. à un autre programme qui tournera en parallèle), afin de retourner le plus vite possible à l'écoute d'une autre requête provenant d'un autre client. La fonction pour créer un autre processus est :
  - **`fork()`** (Cf. « **`man fork`** »)

Quelques outils système UNIX [4/6]



Quelques outils système UNIX [5/6]

- Après le **`fork()`**, le fils traitera la requête, alors que le père fermera le socket ouvert par **`accept()`** et retournera à l'écoute d'une autre requête. Rq : à noter qu'au lieu de créer une nouvelle tâche, il est possible de créer un nouveau *thread* (processus léger).
- Problème : lorsque le fils a fini son traitement, le système ne libère pas les ressources (mémoire, etc.) qu'il possède. On dit alors que le processus reste à l'état de « zombie ».

Quelques outils système UNIX [6/6]

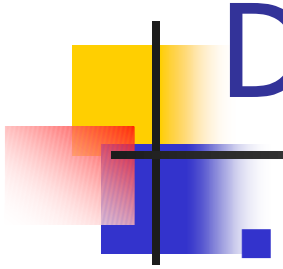
- Solution : utiliser **`signal()`** et **`waitpid()`**

```
#include <sys/wait.h>
#include <signal.h>

[...] void signal_enfant(int non_utilise)
{
    /* Nettoyage des processus fils */
    while(waitpid(-1, NULL, WNOHANG) > 0);
}

[...] int main(int argc, char *argv[])
{
    ...
    signal(SIGCHLD, signal_enfant);
    ...
    if (fork() == 0)
    ...
}
```

TP n°2 – max. 1h30 min [1/1]

- Corriger le code du TP n°1 en fonction des derniers éléments de programmation système qui viennent d'être fournis, afin :
  - Première étape (~45 min) : d'éviter un cas de blocage blocage infini (utilisation de la fonction **`select()`**) ;
  - Deuxième étape (~45 min) : que le serveur puisse répondre à plusieurs requêtes simultanément (utilisation des fonctions **`signal()/waitpid()`** et **`fork()`**).

Le protocole XDR (eXternal Data Representation) [1/7]



- Nous avons commencé à pallier à certains problèmes à l'aide de fonctions système permettant de mettre en place des comptes-à-rebours, du multitâches, etc.
- Pour régler le problème d'interopérabilité lié à la représentation des données, qui peut varier d'une machine à une autre (si elles sont de conception différente), une des solutions (nous en verrons d'autres avec XML) est d'utiliser le mécanisme « XDR »

Le protocole XDR (eXternal Data Representation) [2/7]

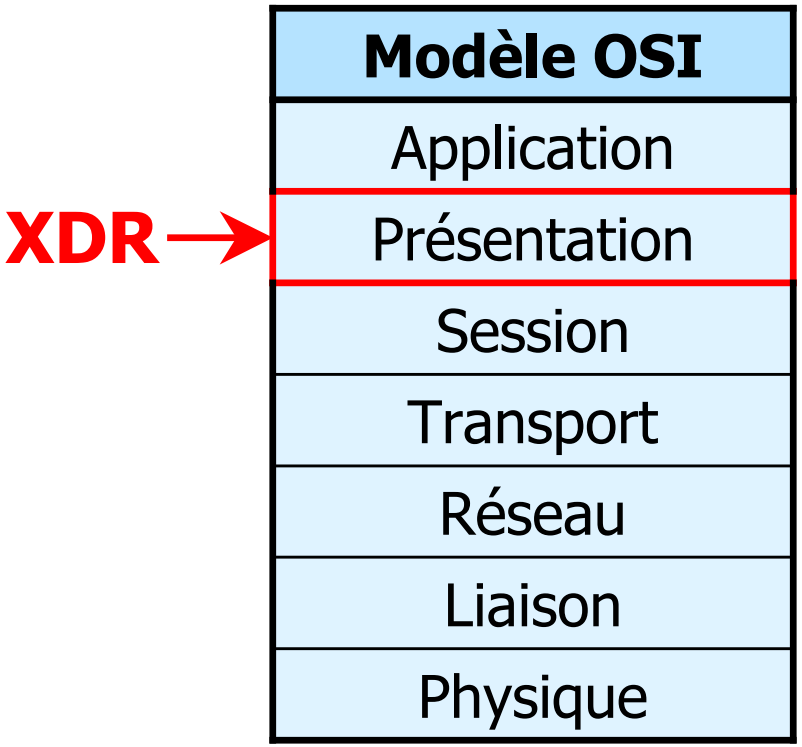
- XDR = eXternal Data Representation, i.e. en français : représentation externe des données
- Proposé et défini par la société SUN, dans le RFC 1832
- XDR est un mécanisme permettant de mettre en forme toute donnée dans un format universel (qui n'est universel que par convention)

Le protocole XDR (eXternal Data Representation) [3/7]

- XDR définit les types de données suivants :
  - Les booléen (vrai/faux),
  - Les entiers (sur 32 bits),
  - Les entiers longs (sur 64 bits, appelés « hyper »),
  - Les nombres à virgule flottante simple précision,
  - Les nombres à virgule flottante double précision,
  - Les énumérations,
  - Les chaînes de caractères (string),
  - Les tableaux (array) de longueur fixe ou variable,
  - Les structures,
  - Les unions.

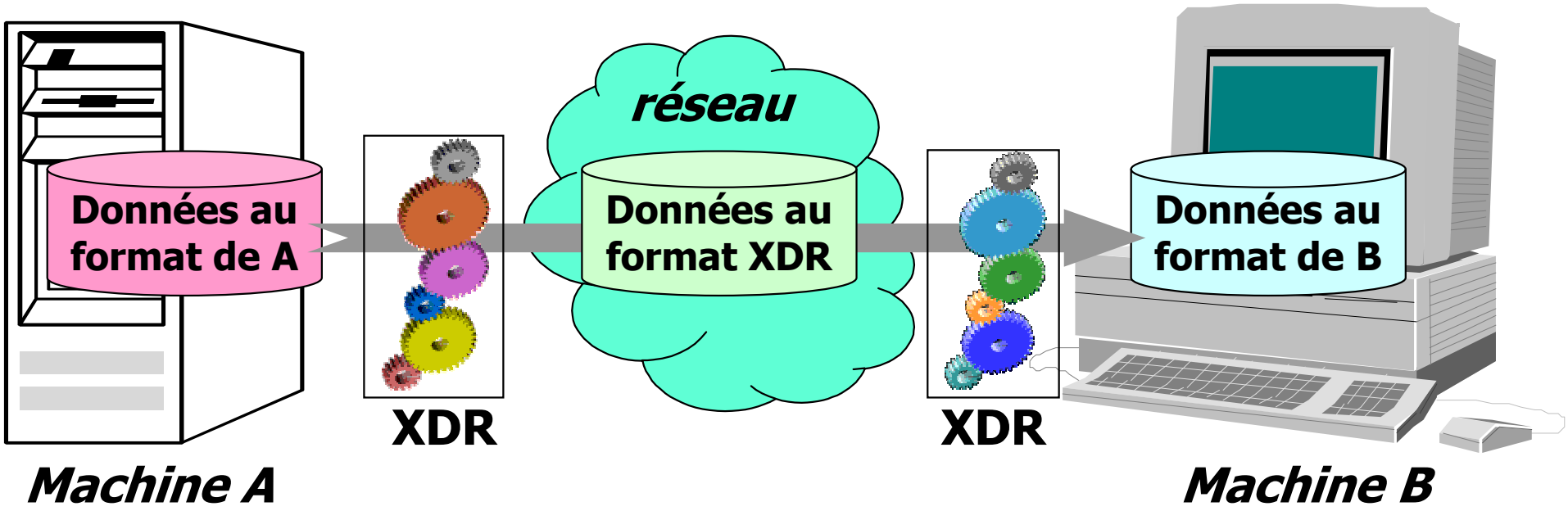
Le protocole XDR (eXternal Data Representation) [4/7]

- En pratique, en plus d'être une définition (un RFC), XDR est aussi une bibliothèque de fonctions qui implémente la couche 6 du modèle OSI



Le protocole XDR (eXternal Data Representation) [5/7]

- Principe schématique (principe général à la couche représentation) :



Le protocole XDR (eXternal Data Representation) [6/7]

- L'API permettant d'utiliser XDR :
  - Les directives « includes » :

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

  - Pour créer un « flot XDR » en mémoire :

```
void  xdrmem(XDR *xdr_ptr, char *adrs,
int longueur, enum xdr_op type_op);
/* L'argument type_op détermine la direction
du flux XDR, et peut valoir : XDR_ENCODE,
XDR_DECODE, XDR_FREE */
```

Le protocole XDR (eXternal Data Representation) [7/7]

- Pour transcoder des données locales vers/depuis un flot XDR :
  xdr_<*type*>(XDR *xdr_ptr, <*type*> *objet_ptr);
  ```
  /*ex: xdr_fload(), xdr_string()... ; ces fonctions retournent une valeur nulle si OK */
  ```
- Pour détruire (libérer les ressources) un « flot XDR » :
  xdr_destroy(XDR *xdr_ptr);
- Cf. « **man xdr** » sur **http://jp.barralis.com/linux-man/**

TP n°3 – max. 45 min [1/1]

- Reprendre le code du TP n°1, et le modifier afin que le flux échangé entre le client et le serveur se fasse selon le protocole « XDR »

Critiques de XDR [1/1]

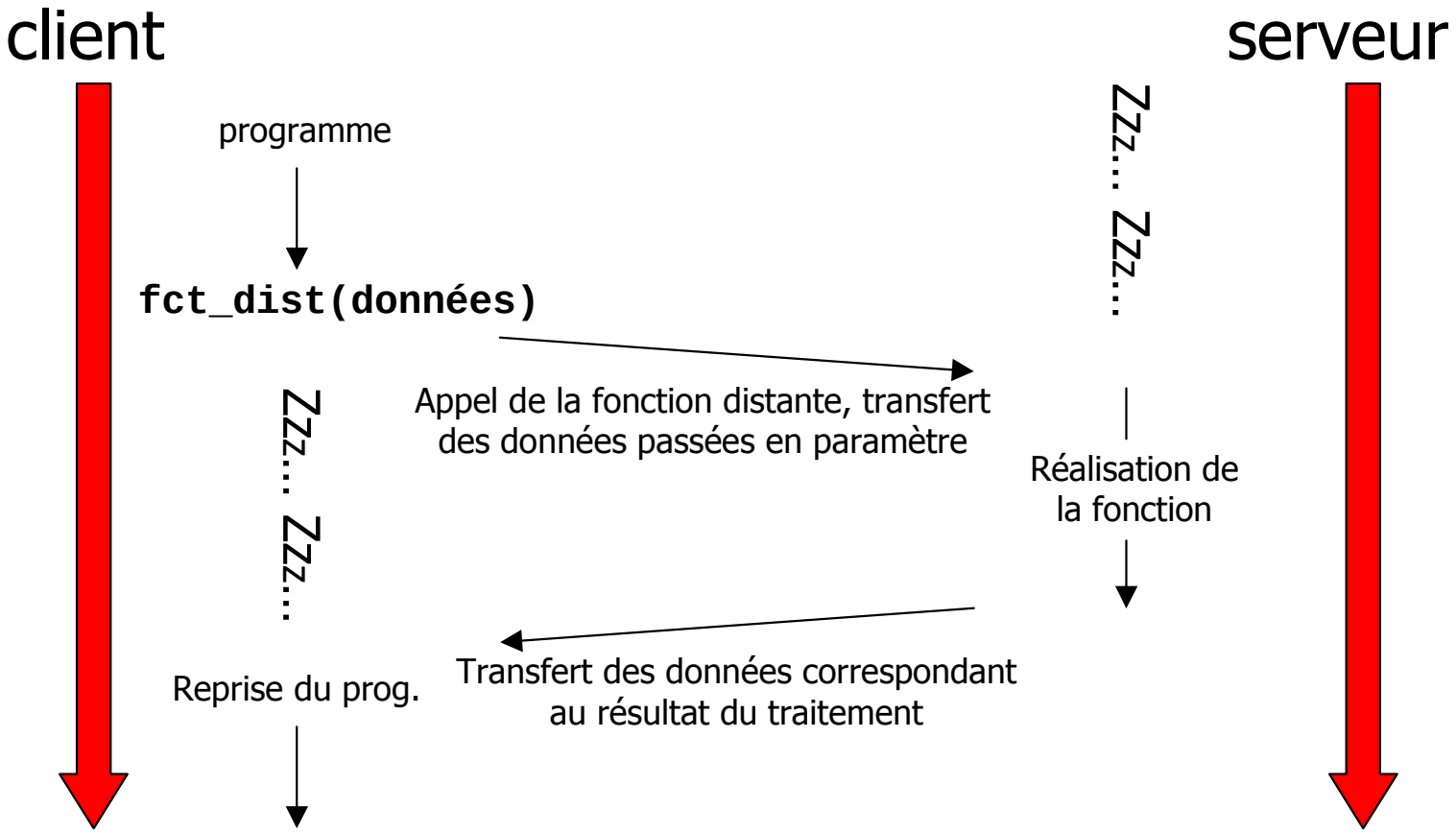
- Les limites d'XDR :
  - Pas de notion d'objet (au sens « programmation objet »),
  - Pas de notion « d'internationalisation » pour les chaînes de caractères (transcode les caractères d'une chaîne, sans traiter les problèmes d'accents, caractères spéciaux, etc.),
  - Peu évolutif (structures et unions permettent de faire des compositions de types, mais il n'est pas simple d'en ajouter de nouveaux ; exemple : réfléchissez à comment coder un entier 128 bits).

Introduction aux RPC [1/10]

- Nous venons de voir un mécanisme permettant aux programmeurs de s'affranchir des problèmes de représentation des données ;
- De même, il existe des mécanismes qui évitent aux programmeurs de s'occuper des problèmes de gestion des connexions, déconnexions, timeout, etc. Une des premières solutions proposées : RPC (proposés par SUN vers 1981/1984).

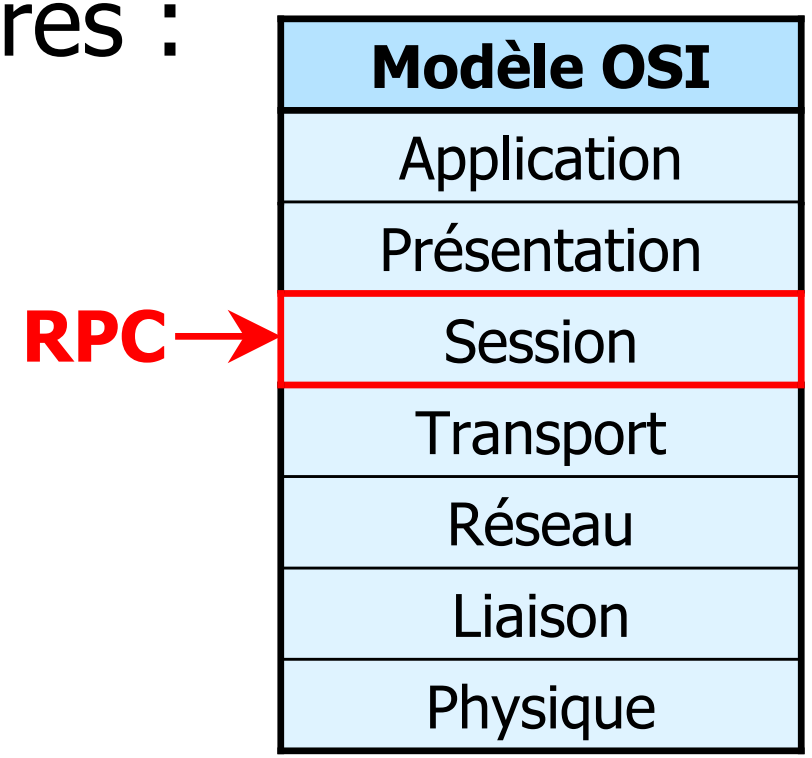
Introduction aux RPC [2/10]

- RPC = Remote Procédure Call (appel d'une procédure distante). En résumé :



Introduction aux RPC [3/10]

- ➔Le programmeur n'a plus qu'à focaliser ses efforts sur la programmation de l'application, le reste étant géré pas les couches inférieures :



Introduction aux RPC [4/10]

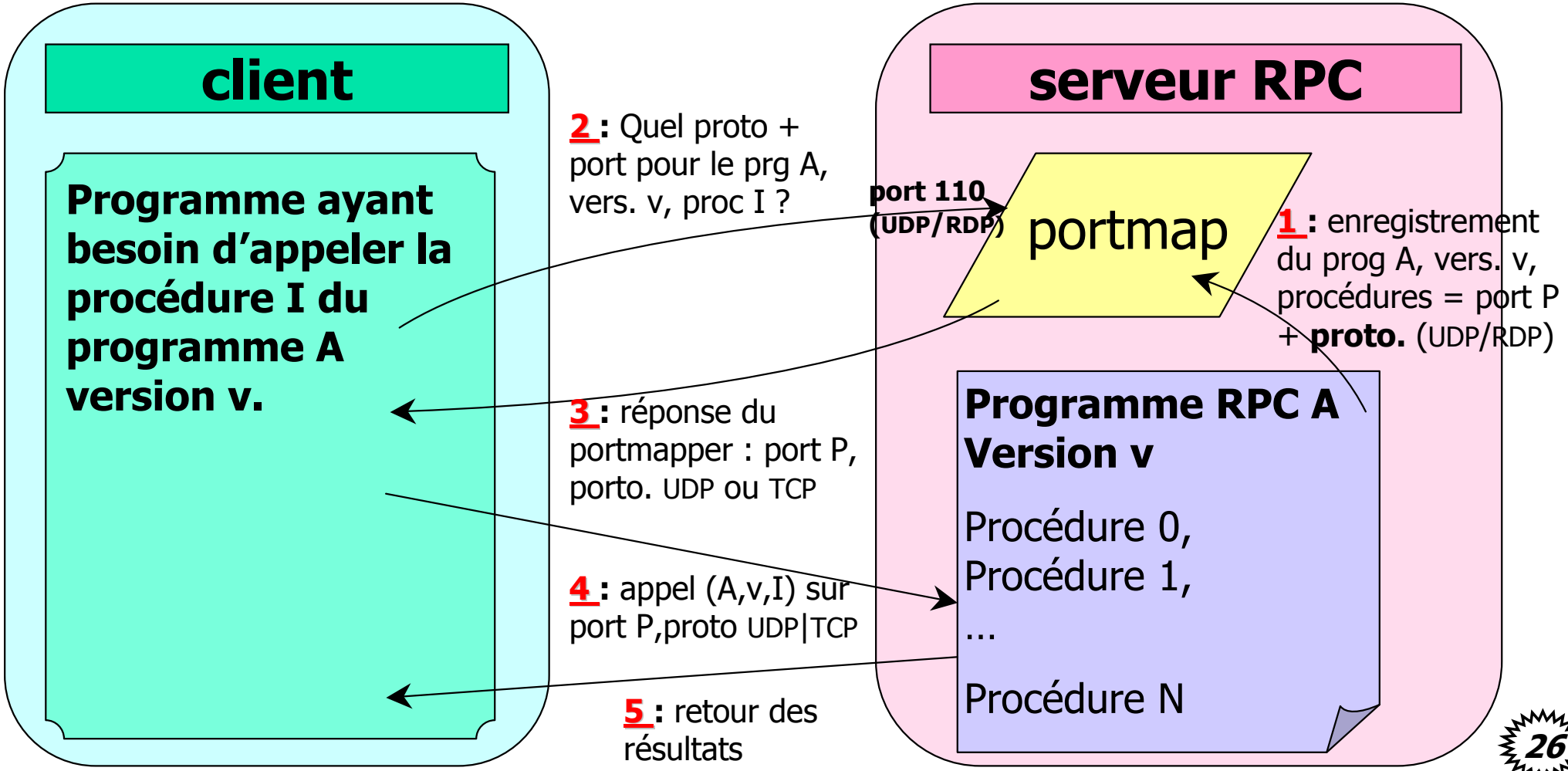
- Au juste… : RPC = RFCs n°1050 & 1831
- Principe des RPC proposés par SUN :
  - Chaque programme distant est identifié par un entier codé sur 32 bits, utilisé par l'appelant
  - Les procédures d'un programme distant sont identifiées séquentiellement par les entiers 0, 1,…N
  - Une procédure distante est identifiée par le triplet :
    (programme, version, procédure)
    avec:
    - **programme** : identifie le programme distant
    - **version** : identifie la version du programme
    - **procédure** : identifie la procédure

Introduction aux RPC [5/10]

- Pour chaque version de chaque programme, la procédure n°0 est réservée : elle permet de tester la disponibilité du service
- Le standard RPC propose un mécanisme d'authentification (en pratique, peu sécurisé)
- On peut faire des RPC sur UDP et/ou sur TCP
- Chaque programme est toujours à l'écoute d'un numéro de port donné
- Problème : comment un processus client peut-il connaître le numéro de port à interroger ?
- Solution proposée par SUN : le « *portmapper* »

Introduction aux RPC [6/10]

- Fonctionnement du portmapper :



Introduction aux RPC [7/10]

- Ex. d'application : Network File Service (NFS), inventé par SUN, utilise RPC+XDR
- Intérêts des RPC :
  - simple à utiliser : pour le programmeur, l'API est faite pour qu'un appel à une procédure distante soit aussi simple à réaliser qu'un appel à une procédure locale
  - le mécanisme du portmapper rend transparent la recherche du bon n° de port (il faut tout de même connaître l'adresse du serveur à interroger)

Introduction aux RPC [8/10]

- Les limites des RPC :
  - Orienté « programmation procédurale » : pas de notion d'objet, pas de référence à des services distants (méthodes)
  - Dépend de la localisation du serveur (le client doit connaître l'@ du serveur, pas de *« servermapper »*)
  - Pour les RPC sur UDP, la non réponse du serveur n'indique pas si la procédure a eu lieu sur le serveur (ex. de pb : si l'appel à une procédure distante met à jours un SGBD, la non réponse du serveur ne permet pas de savoir si la base de données a été modifiée).

Introduction aux RPC [9/10]

- Autres formes d'appel de fonction à distance selon le modèle client/serveur plus récents (tentant de pallier à certains inconvénients) :
  - OSF/DCE,
  - CORBA,
  - Java RMI,
  - Object RPC,
  - MS-COM/DCOM,
  - WS SOAP,
  - XML-RPC,
  - SUN J2EE-EJB,
  - OMG-CCM, ... etc. (Cf. prochains cours...)

Introduction aux RPC [10/10]

- Pour en savoir plus sur les RPC :
  - Concernant l'API C/Unix : « **`man rpc`** »
  - Commande à lancer sur une machine pour connaître les services RPC disponibles :

    `rpcinfo`

    ➔ « **`man rpcinfo`** »
- Exemple d'exercice d'entraînement : refaire le TP 2 + 3 en utilisant les RPC