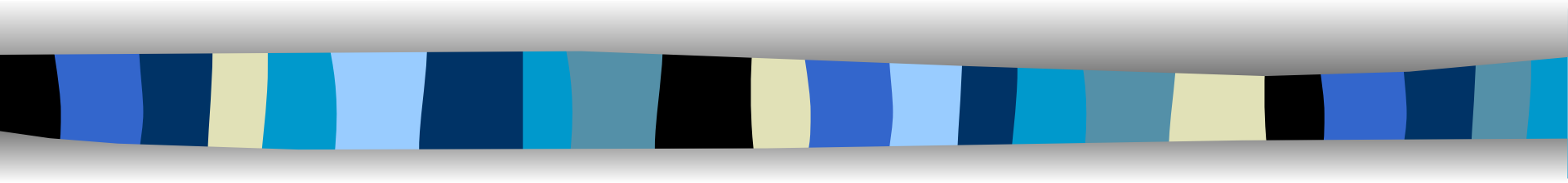
Appel de procédure à distance



Département d'Informatique
Université de Batna 2

Master 1 : ISI
Semestre : 2

Plan du chapitre

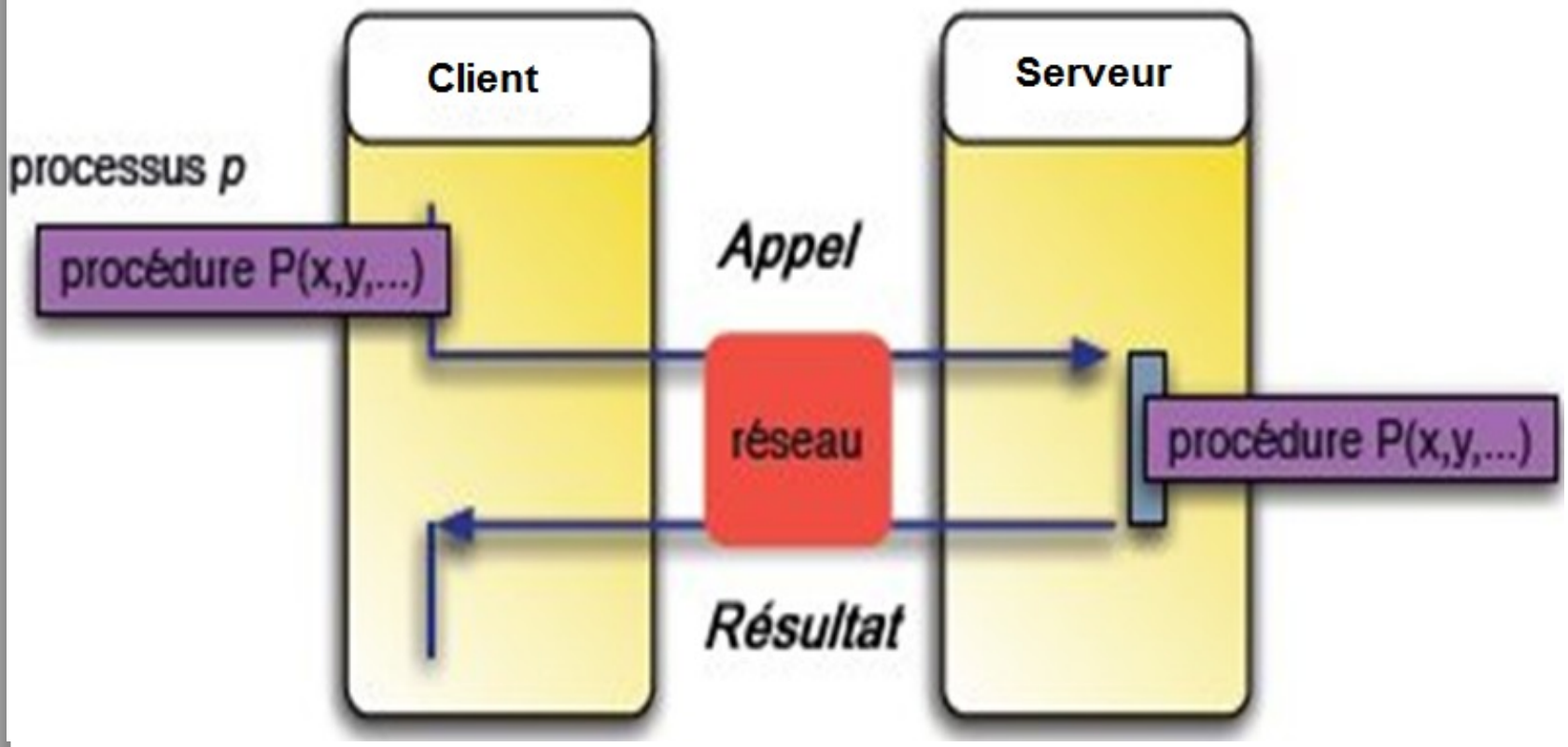
I. **Introduction**
   I. Principe général
   II. Fonctionnement détaillé
II. **Gestion du contrôle**
   I. Gestion du parallélisme chez le client
   II. Gestion du parallélisme chez le serveur
III. **RPC sous Unix**
   I. Principe général
   II. Problème d'hétérogénéité
   III. Problème de nommage
   IV. PCGEN
   V. Langage RPCL
   VI. Etapes de Mise œuvre

I. Introduction

- **RPC (Remote Procedure call) : Appel de procédure à distance qui représente l'une des exemple implémentant le modèle client/serveur.**
- **Système de distribution de services proposé par SUN au début des années 1980**
- **Mécanisme permettant du coté serveur de proposer des services qui seront appelés par des clients distants**
  - Service = une fonction C
  - Coté client : appel d'une fonction locale qui aboutira à l'exécution de la fonction requise sur le serveur

I. Introduction

- **Principe général**

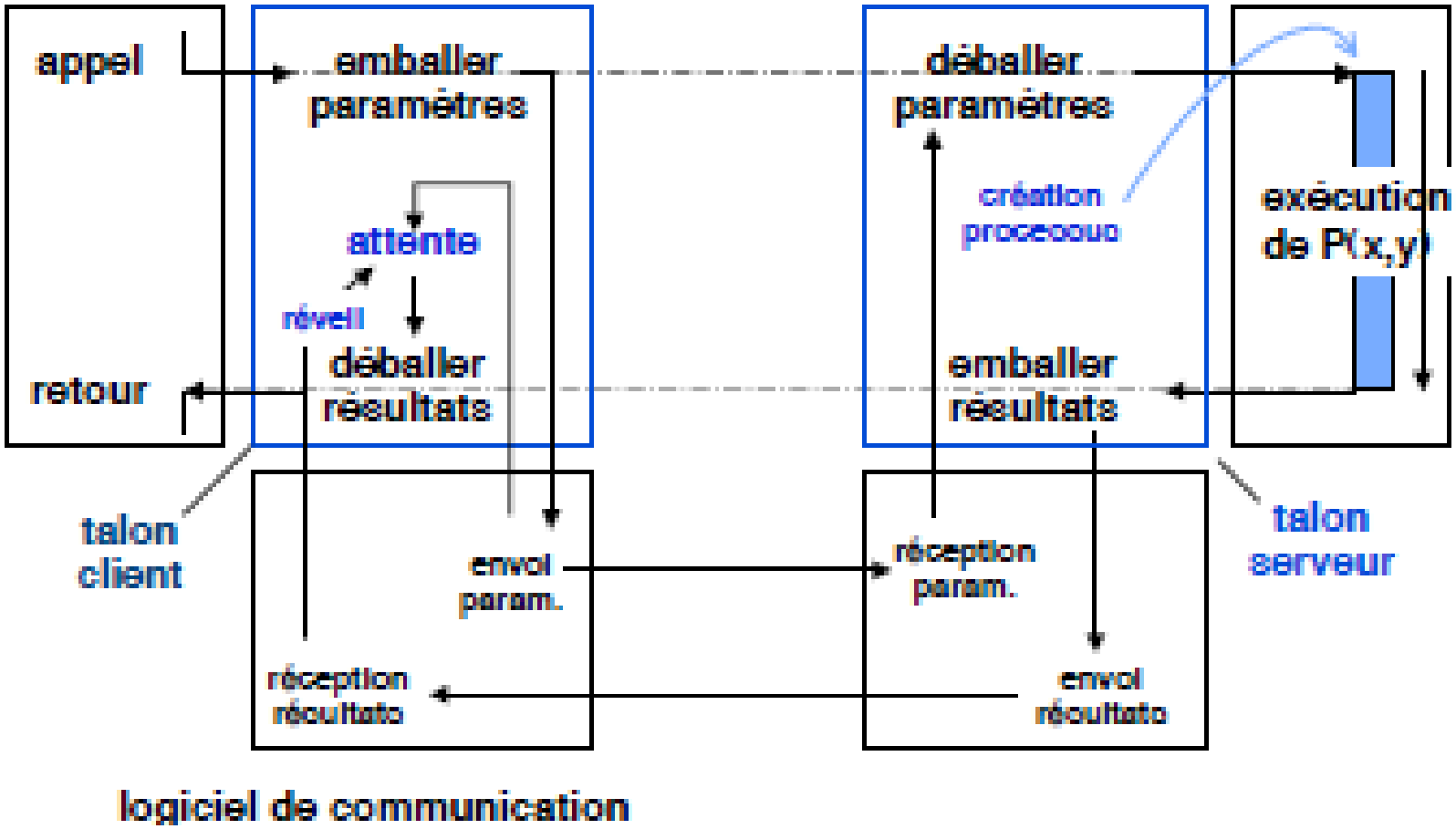


I. Introduction

- **Implantations**
  - les implantations de RPC traditionnelles
    - **SUN ONC/RPC (ONC, Open Network Computing)**
  - les implantations de RPC dans les systèmes d'objets répartis
    - SUN Java RMI (Remote Methode Invocation)
    - OMG CORBA (Object Management Group)
  - les implantation de RPC dans les systèmes de composants
    - Web Service et protocole SOAP *(Simple Object Access Protocol)*

I. Introduction

- **Fonctionnement détaillé**



I. Introduction

- **Fonctionnement détaillé**

Souche Client : (talon) stub

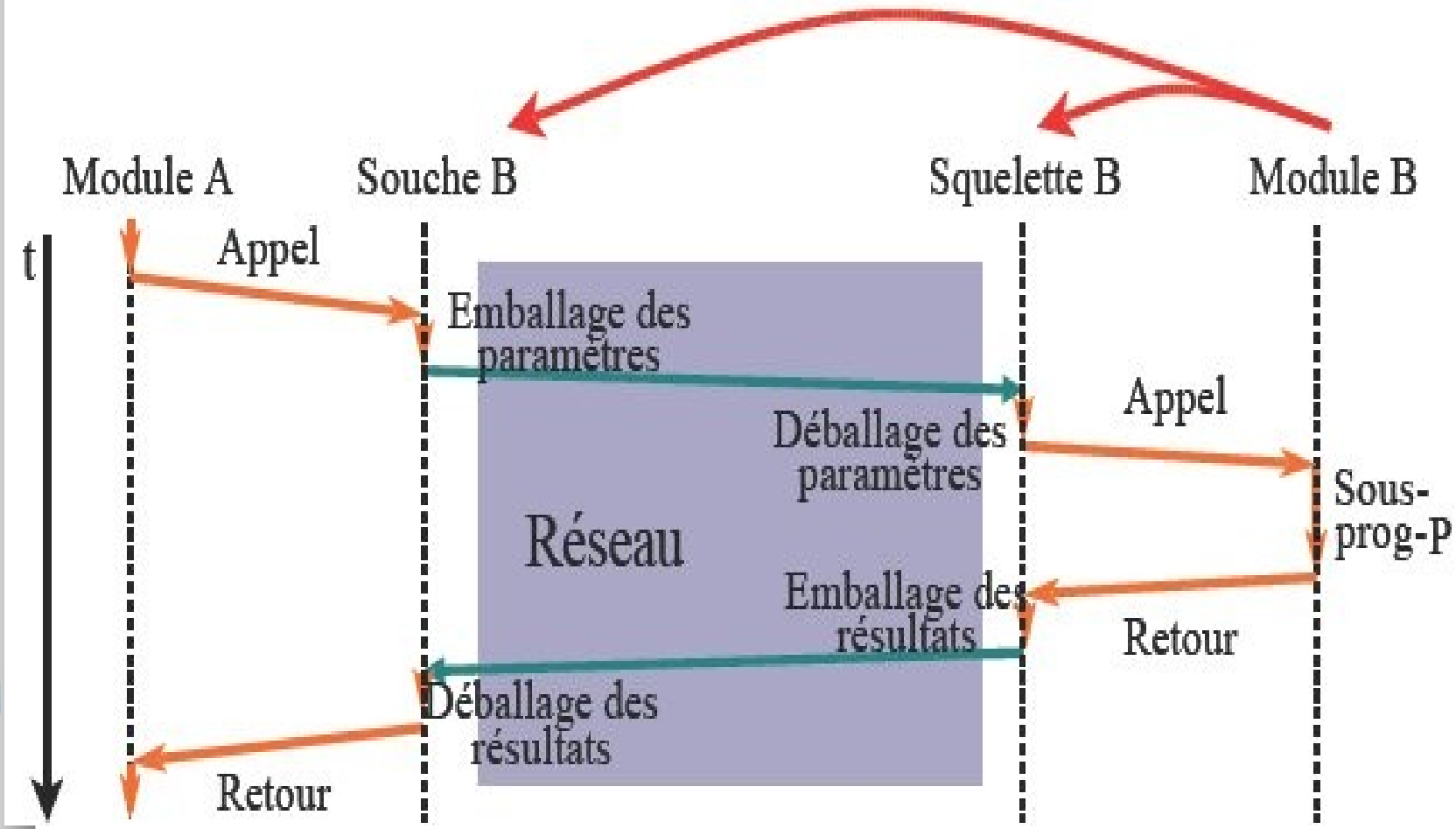
- *C'est la procédure d'interface du client.*
- *Reçoit l'appel en mode local*
- *Le transforme en appel distant en envoyant un message.*
- *Reçoit le message contenant les résultats après l'exécution*
- *Retourne les résultats à l'appelant.*

Souche Serveur: skeleton

- *C'est la procédure d'interface du serveur.*
- *Reçoit l'appel sous forme de message.*
- *Fait réaliser l'exécution sur le serveur*
- *Transmet les résultats par message*

I. Introduction

- **Fonctionnement détaillé**

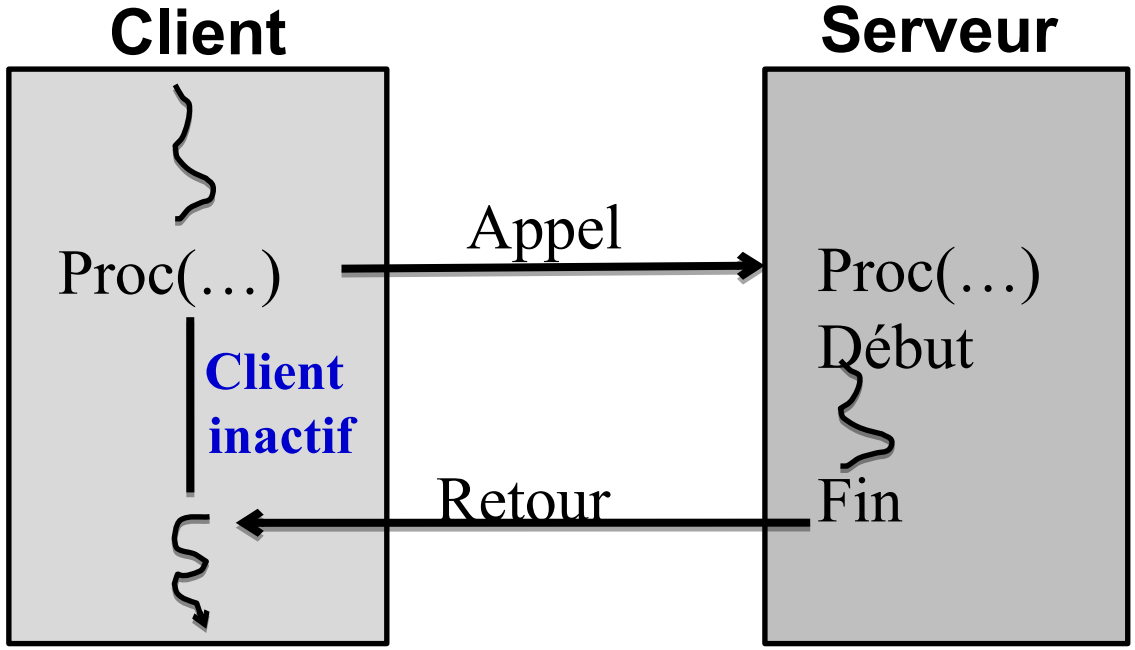


II. Gestion du contrôle

- **Parallélisme chez le Client**
  - RPC en Mode Synchrone
    - Synchrone Inactif
    - Synchrone Actif
  - RPC en mode Asynchrone
- **Parallélisme chez le Serveur**
  - Exécution séquentielle des appels
  - Exécution parallèle des appels

II. Gestion du contrôle/Parallélisme chez le Client

- **RPC en mode Synchrone inactif**
  - L'exécution du client est suspendue tant que la réponse du serveur n'est pas revenue ou qu'une condition d'exception n'a pas entraîné un traitement spécifique.

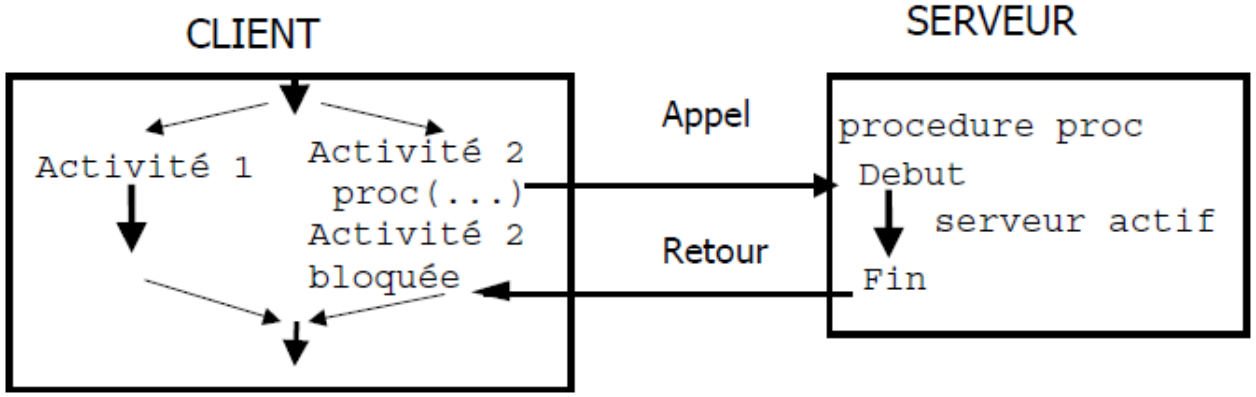


II. Gestion du contrôle/Parallélisme chez le Client

- **RPC en mode Synchrone Inactif**
  - **Avantage**
    - le flot de contrôle est le même que dans l'appel en mode centralisé
  - **Inconvénient**
    - le client reste inactif pendant l'appel.

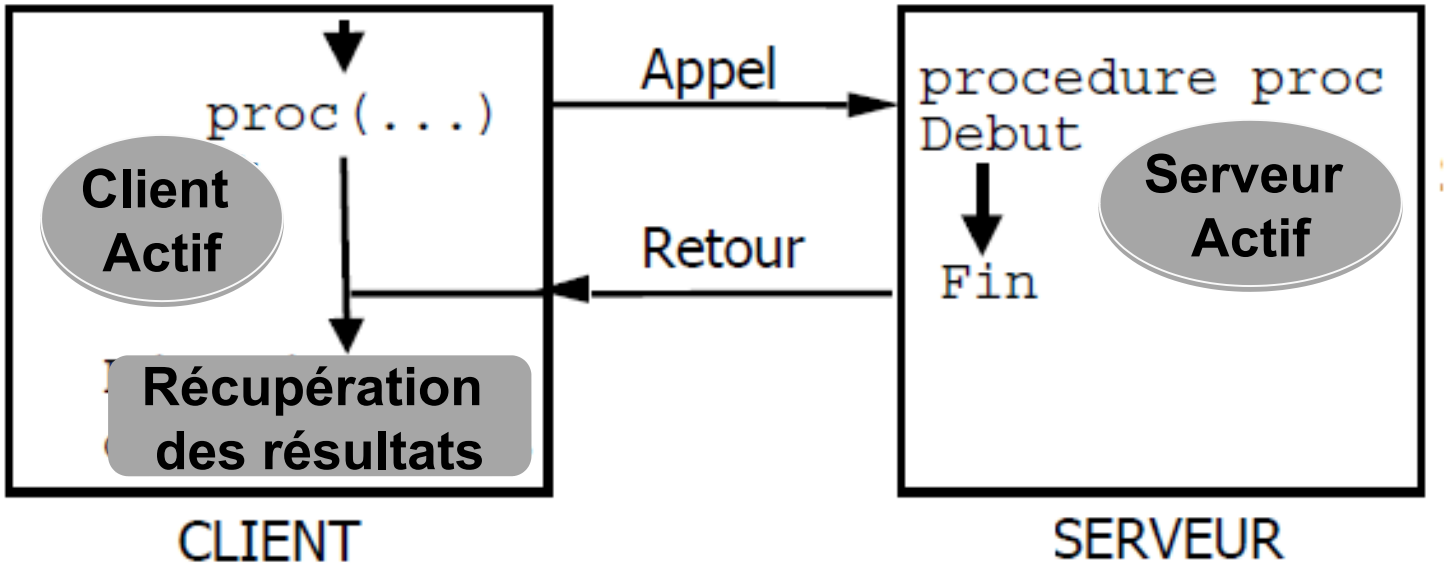
II. Gestion du contrôle/Parallélisme chez le Client

- **RPC en mode Synchrone actif**
  - Création de (au moins) deux activités (‘ex: threads’) sur le site client
    - L'une occupe le site appelant par un travail à faire.
    - L'autre gère l’appel en mode synchrone en restant inactif (bloquée): Le fonctionnement est exactement celui d'un appel habituel.



II. Gestion du contrôle/Parallélisme chez le Client

- **RPC en mode Asynchrone**
  - Le client poursuit son exécution **immédiatement** après l'émission du message porteur de l'appel.
  - La procédure distante **s'exécute en parallèle** avec la poursuite du client.
  - Le client doit **récupérer les résultats** quand il en a besoin (primitive spéciale).

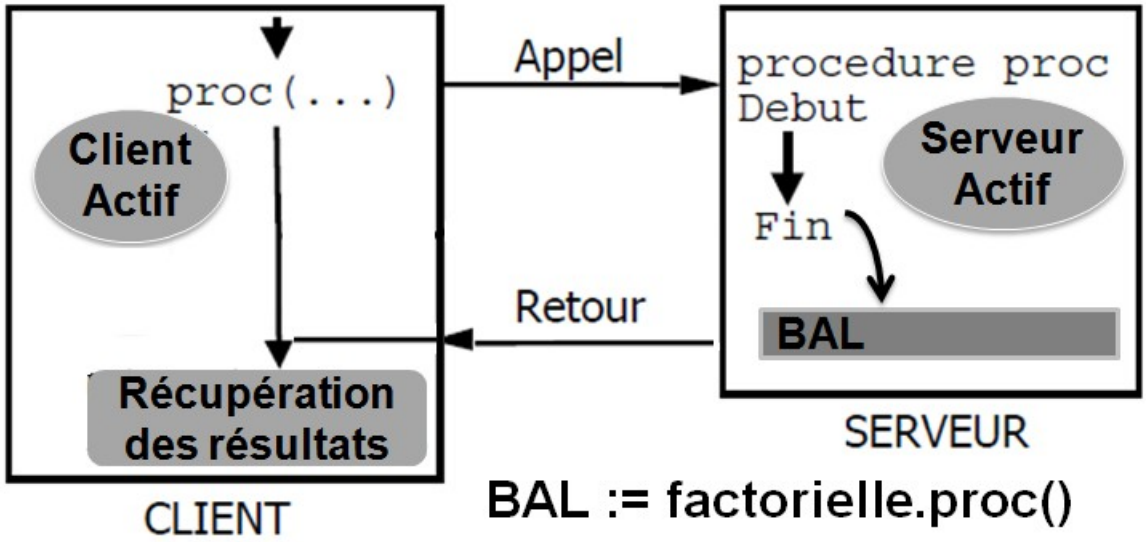


II. Gestion du contrôle/Parallélisme chez le Client

- **RPC en mode Asynchrone : Récupération des résultats**
  - Notion de **futur :** une structure de donnée (un objet) permettant de récupérer des résultats.
    - Futurs explicites
    - Futurs implicites

II. Gestion du contrôle/Parallélisme chez le Client

- **Invocation asynchrone à futurs explicites**
  - Les structures de données sont définies par le client avant l'appel (le serveur les connaît et y dépose les résultats).
  - Exemple: En ACT++ une boite à lettre définie par le client sert de moyen de communication pour les paramètres résultats.

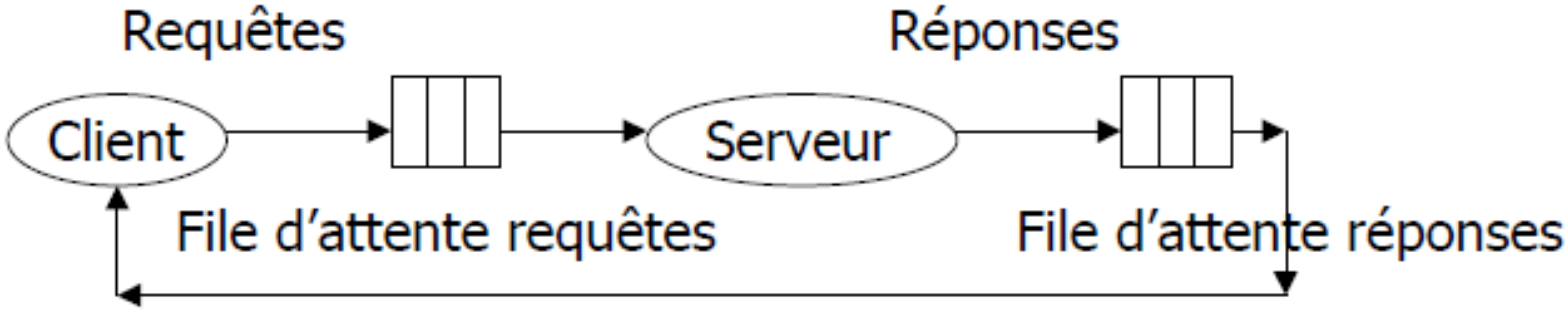


II. Gestion du contrôle/Parallélisme chez le Client

- **Invocation asynchrone à futurs implicites**
  - Les structures de données de communication pour les réponses (ex boite à lettre) sont implicitement créés par le prestataire du service RPC asynchrone.
  - La lecture de la structure de donnée résultat bloque le client s'il accède à la réponse et que celle-ci n'est pas parvenue.
  - L'usager ne s'aperçoit de rien (si le résultat est parvenu ou s 'il n'est pas parvenu).

II. Gestion du contrôle/Parallélisme chez le Serveur

- **Exécution séquentielle des appels**
  - Les requêtes d'exécution sont traitées l'une après l'autre par le serveur: exclusion mutuelle entre les traitements.
  - Si la couche transport assure la livraison en séquence et que l'on gère une file d'attente premier arrivé premier servi, on a
  - un traitement ordonné des suites d'appels.

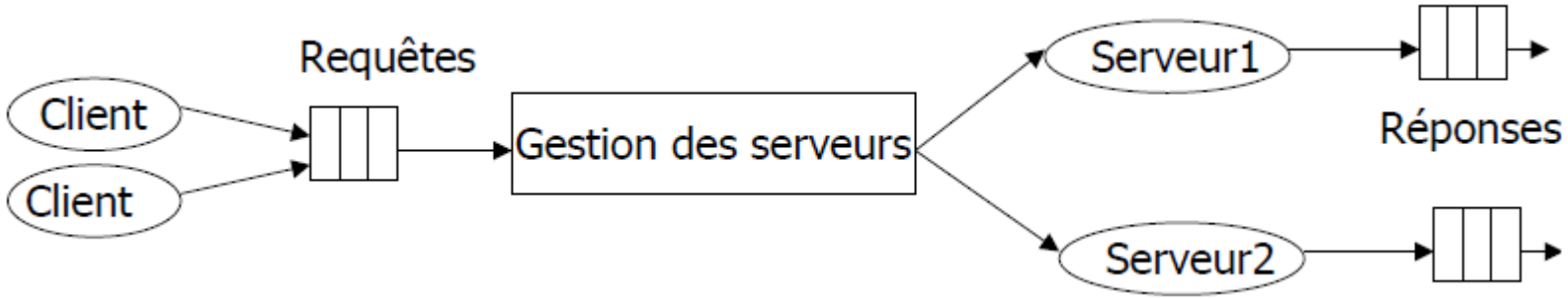


II. Gestion du contrôle/Parallélisme chez le Serveur

- **Exécution séquentielle des appels**
  - Exemple RPC SUN : traitement séquentiel des requêtes mais utilisation de UDP => requêtes non ordonnées (mais mode synchrone le client attend la fin du traitement).
  - Autres exemples: les RPC ont un mode séquentiel (ex CORBA)

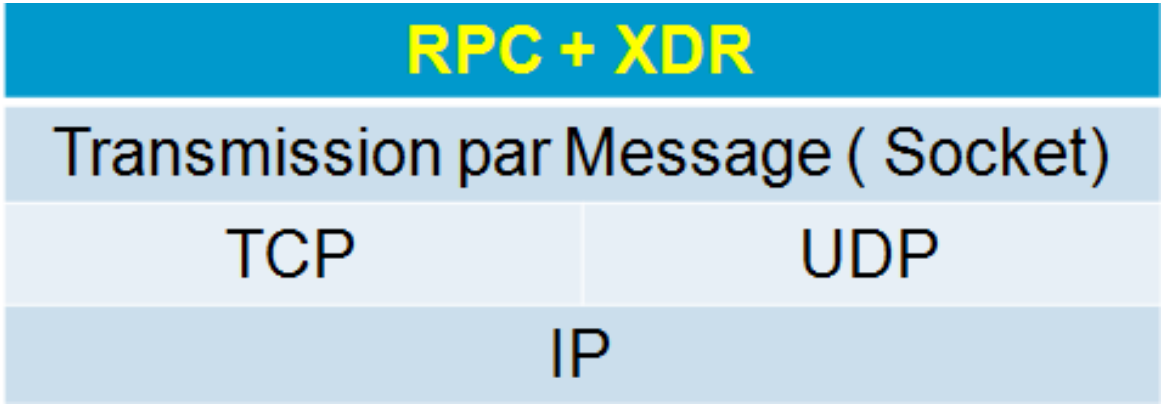
II. Gestion du contrôle/Parallélisme chez le Serveur

- **Exécution parallèle des appels**
  - Dans ce mode le serveur créé un processus ou une activité (un processus léger ou “thread”) par appel (gestion possible de pool de processus ou d ’activités).
  - Les appels sont exécutés concurremment.
  - Si les procédures manipulent des données globales rémanentes sur le site serveur, le contrôle de concurrence doit être géré.
  - Exemple : Corba Notion d’adaptateur d’objets.



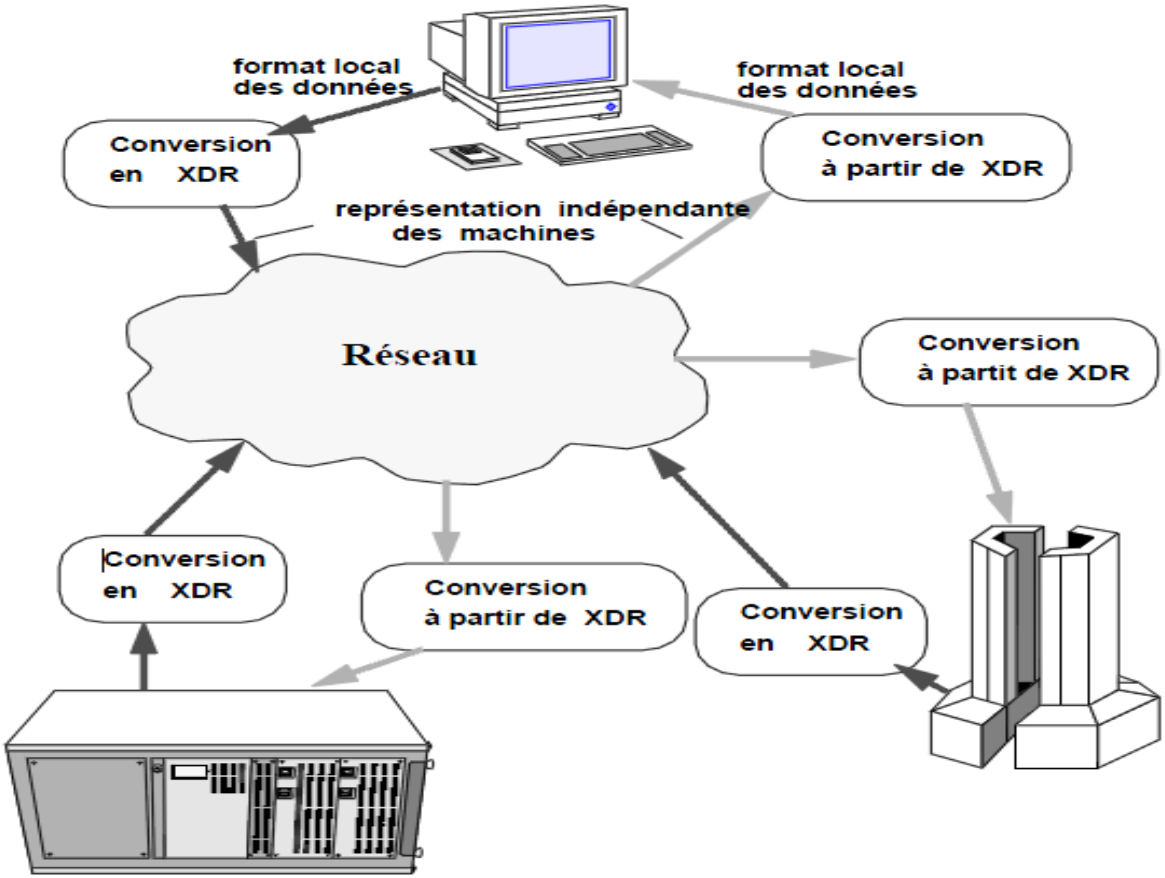
III. RPC sous UNIX/ Principe général

- **Principe général**
  - Open source
  - Implantés au-dessus de la couche transport,
  - ils sont notamment utilisés par NFS (Network File System : Système de Fichier en Réseau) et NIS (Network Information Service : Service d'Information Réseau).
  - Utilise le protocole XDR (*eXternal Data Representation*) pour le transport des données



III. RPC sous UNIX / Heterogénéité

- **Problème d'hétérogénéité (solution XDR)**
  - Trouver un codage commun de tous les types de données indépendant de leurs plateformes.



III. RPC sous UNIX / XDR

- **XDR**
  - Toutes les données échangées, pour réaliser les RPCs, sont codées en **XDR**
  - Librairie XDR (types de données XDR + primitives de codage/décodage pour chaque type)
  - **XDR** est un protocole de **sérialisation** indépendant du client et du serveur
  - la **sérialisation** (de l'anglais *serialization*) est un processus visant à coder l'état d'une information qui est en mémoire sous la forme d'une **suite d'informations plus petites** (dites *atomiques* le plus souvent des octets).
  - Cette **suite** pourra par exemple être utilisée pour la sauvegarde ou le transport sur le réseau.

III. RPC sous UNIX / XDR

- **XDR**
  - L'activité symétrique, visant à décoder cette **suite** pour créer une copie conforme de l'information d'origine, s'appelle la **désérialisation**.
- **Fonctionnement de XDR**
  - La talon client sérialise en **XDR** les arguments, envoie la requête au serveur, récupère le résultat et le déserialise .
  - le talon serveur déserialise les arguments, exécute le code demande, sérialise en XDR le résultat et l'envoie au client.

III. RPC sous UNIX / XDR

- **Filtres XDR**
  - Un filtre a trois fonctionnalités. Il permet l'encodage, le décodage mais également la libération de la mémoire qui a pu être allouée par le filtre.
  - Tous les filtres renvoie TRUE en cas de réussite et FALSE en cas d'échec!

Filtre	Type
char	xdr_char()
short	xdr_short()
……	……
float	xdr_float()
double	xdr_double()
void	xdr_void()
enum	xdr_enum()

III. RPC sous UNIX / XDR

- **Filtre XDR**
  - Les deux arguments d'un filtre sont de même type en encodage et en décodage.

    boolt xdr_*type* (XDR*pt_xdr,type*pt);
  - Paramètres :
    1. pt_xdr : pointeur sur le flot XDR en encodage ou décodage
    2. pt : pointeur sur zone contenant valeur à encoder ou zone réservée pour recevoir la valeur décodée
  - Retour :
    1. TRUE : en cas de succès
    2. FALSE : en cas d'échec

III. RPC sous UNIX / Nommage

- **Problème**
  - Comment un client fait -il pour trouver le serveur ?
    - Solution statique : écrit son adresse dans son code
    - Problème : solution rigide (et si le serveur change d'adresse !!!)
  - Ce problème est baptisé problème de **Nommage**
- **Solution robuste : nommage dynamique (dynamic Binding)**
  - Gestionnaire de noms : intermédiaire entre le client et le serveur
  - Portmapper : processus daemon s'exécutant sur le serveur

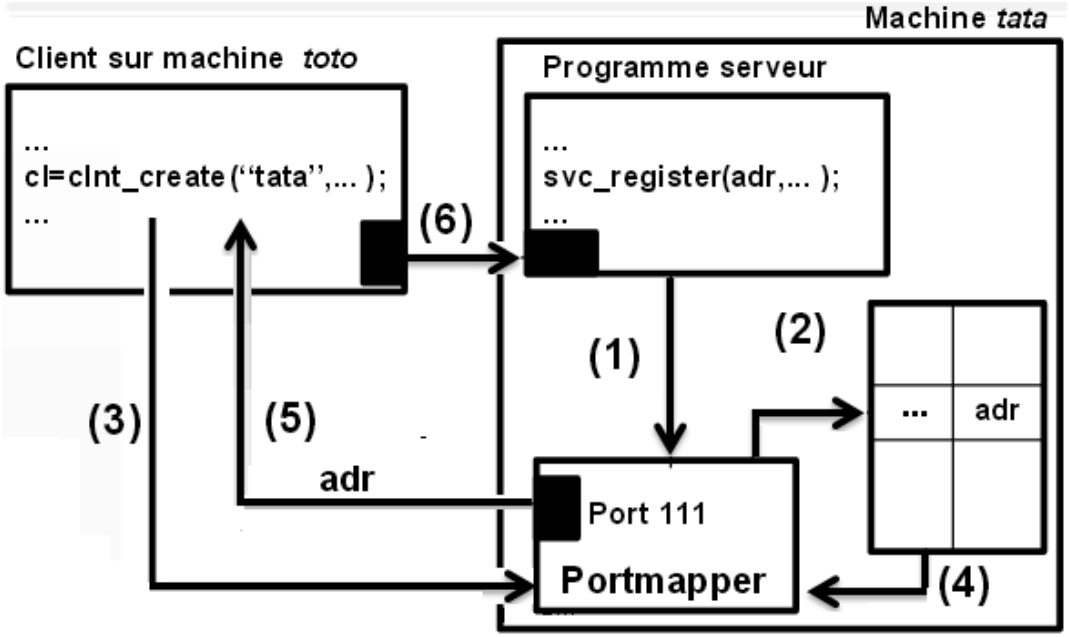
III. RPC sous UNIX / Nommage

- **Problème de nommage (portmap)**
  - Utilisation d'un serveur qui centralise les processus qui fournissent des services RPC : le **portmap**
    - Lorsqu'un programme veut proposer des services, il s'enregistre auprès du **portmap** en donnant un numéro de programme, un numéro de version et un numéro par service proposé.
  - **protmap** permet de rediriger un client vers le numéro de port hébergeant le service .
    - est un logiciel daemon sous Unix/Linux qui convertit les numéros de programmes RPC en numéros de port logiciel
  - Le **portmap** est sur un numéro de port réservé : 111 : Les clients n'ont besoin de connaître que ce seul numéro de port/service.

Logiciel daemon un processus ou un ensemble de processus qui s'exécute en arrière-plan plutôt que sous le contrôle direct d'un utilisateur.

III. RPC sous UNIX / Nommage

- En utilisant le **portmap**, nous devons suivre les étapes suivante pour récupérer le N°_port d'un service donné :
  - (1) , (2) pmap_set() ; enregistre un service
  - (3), (4) et (5) pmap_getport() : retourne le numéro de port associé au service
  - (6) pmap_rmtcall() : appel d'une procédure distante

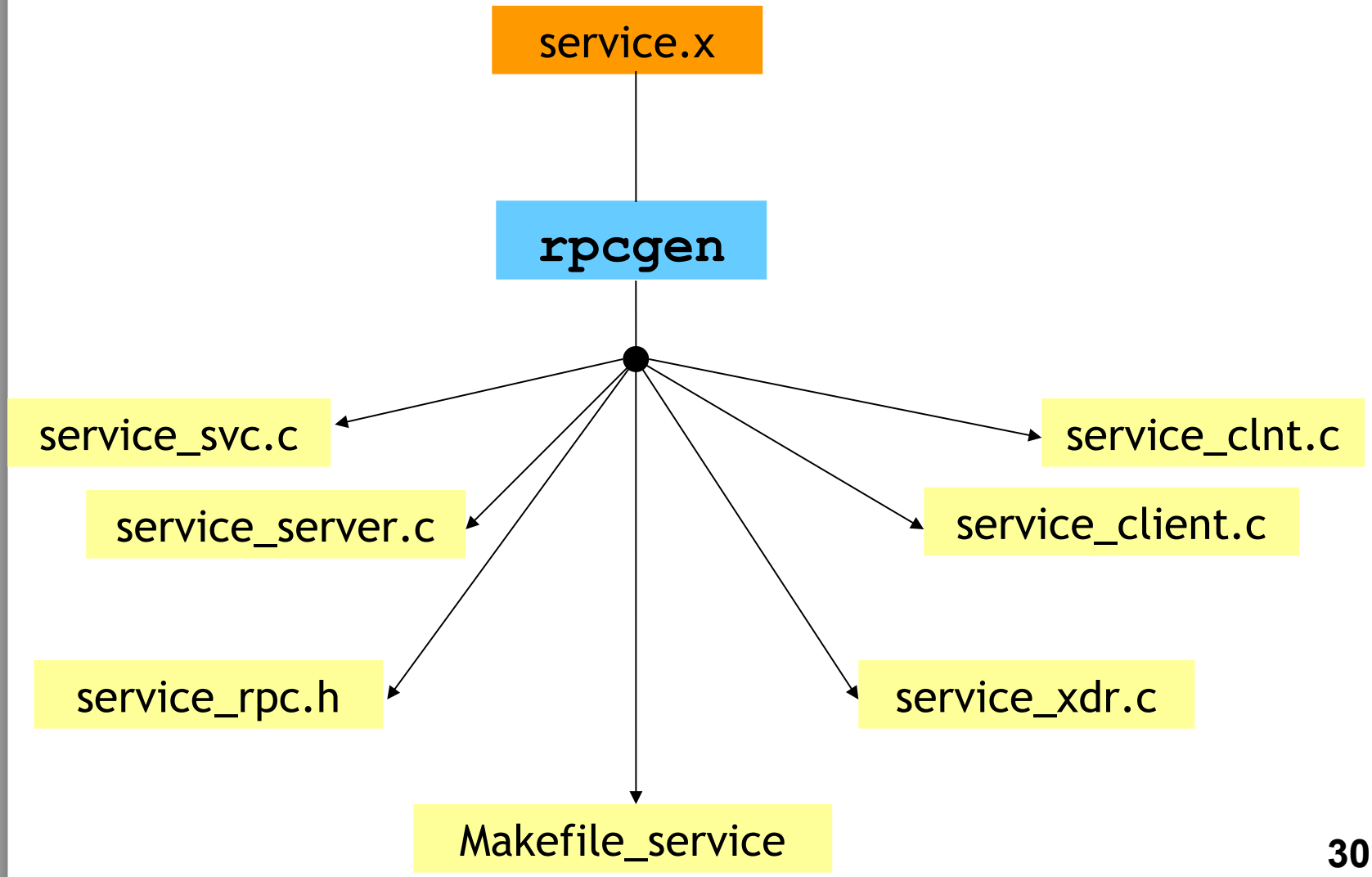


III. RPC sous UNIX / rpcgen

- **Outil pratique RPCGEN**
  - rpcgen permet d'obtenir des programmes répartis à partir de leur spécification.
  - On décrit les données et les programmes grâce à un langage voisin du C dans un fichier suffixé « .x » (prog.x).
    - **RPCL** (RPC Language)
  - On traite ce fichier par le générateur rpcgen.

III. RPC sous UNIX / rpcgen

- **Outil pratique RPCGEN**



III. RPC sous UNIX / rpcgen

- **utilisation de RPCGEN**
  - rpcgen **–a** service.x
  - les fichiers Makefile_service, service_xdr.c, service_rpc.h, service_svc.c, service_clnt.c, service_server.c, service_client.c sont générés
  - il ne reste plus qu'à implémenter les procédures dans service_server.c et le programme client (service_client.c)

III. RPC sous UNIX / RPCL

- **Pareillement qu'il existe un langage de description des types XDR, il existe un langage de description des procédures qui opèrent sur ces types.**
- **Ce langage est s'appel RPCL(Remote Procedure Call Language)**
- **C'est un Langage décrivant les services**
- **Extension habituel de ce fichier : .x**

III. RPC sous UNIX / RPCL

- **un exemple d'énumération RPCL :**

  enum-definition: "enum" enum-ident "{" enum-value-list "}"

  enum-value-list: enum-value "," enum-value-list

  enum-value: enum-value-ident

  enum-value-ident "=" value

```
enum{
    LUNDI = 0,  MARDI = 1, MERCREDI = 2, JEUDI = 3,
    VENDREDI = 4, SAMEDI = 5, DIMANCHE = 6
  };
```

- **Un exemple de constante RPCL :**

  const-definition:

  "const" const-ident "=" integer

```
const JOURS_PAR_SEMAINE = 7;
```

III. RPC sous UNIX / RPCL

- **Déclaration des types**

  typedef-definition: "typedef" declaration

  declaration: simple-declaration fixed-array-declaration variable-array-declaration pointer-declaration

  simple-declaration: type-ident variable-ident

  fixed-array-declaration: type-ident variable-ident "["value"]"

  variable-array-declaration: type-ident variable-ident "<"value">"

  type-ident variable-ident "< >"

  pointer-declaration: type-ident "*"variable-ident

- **Déclaration des structures**

  struct-definition: "struct" struct-ident "{"declaration-list "}"

  declaration-list: declaration "; " declaration ";" declaration-list

III. RPC sous UNIX / RPCL

- **Déclaration des unions**

  union-definition: "union" union-ident "switch" "("simple declaration")" "{" case-list "}"

  case-list:

  "case" value ":" declaration ";"

  "case" value ":" declaration ";" case-list

  "default" ":" declaration ";"

- **Les *unions* constituent un autre type de structure. Elles sont déclarées avec le mot clé *union*, qui a la même syntaxe que *struct*.**
- **La différence entre les structures et les unions est que les différents champs d'une union occupent le même espace mémoire.**
- **On ne peut donc, à tout instant, n'utiliser qu'un des champs de l'union.**
- **Exemple. Déclaration d'une union**
  - union entier_ou_reel { int entier; float reel; }; union entier_ou_reel x;
  - *x* peut prendre l'aspect soit d'un entier, soit d'un réel. Par exemple :
  - x.entier=2; affecte la valeur *2* à *x.entier*, ce qui détruit *x.reel*.
  - Si, à présent, on fait : x.reel=6.546; la valeur de *x.entier* est perdue, car le réel *6.546* a été stocké au même emplacement mémoire que l'entier *x.entier*.
- **Les unions, contrairement aux structures, sont assez peu utilisées, sauf en programmation système où l'on doit pouvoir interpréter des données de différentes manières selon le contexte. Dans ce cas, on aura avantage à utiliser des unions de structures anonymes et à accéder aux champs des structures, chaque structure permettant de manipuler les données selon une de leurs interprétations possibles.**

III. RPC sous UNIX / RPCL

- **Déclaration d'un programme**

  program-definition: "**program**" program-ident "{"
  version-list "}" "=" value;

  version-list: version "; " version ";" version-list

  version:
  "**version**" version-ident "{" procedure-list "}" "=" value;
  procedure-list: procedure "; "procedure ";" procedure-list
  procedure: type-ident procedure-ident "(" type-ident ")" "="
  value;
  procedure: type-ident procedure-ident "(" type-ident-list ")" "="
  value;
  type-ident-list: type-ident type-ident "," type-ident-list

III. RPC sous UNIX / RPCL

- **Types de base**
  - char, short, int, unsigned int, foat, double, bool, string

III. RPC sous UNIX / Mise en œuvre

- **Forme générale du contrat**

```
/* Définitions de types utilisateur */
...
program «nomprog» {
version «nomversion1» {
«typeres1» PROC1(«param1») = 1;
...
«typeresn» PROCn(«paramn») = n;
} = 1;
...
version «nomversionm» {
...
} = m;
} = «numéro_du_programme»;
```

III. RPC sous UNIX / Mise en œuvre

- **Méthodologie de développement**
  - **Etape 1 :**
    - Ecrire le contrat prog.x dans le langage RPCL
    - Compiler le contrat avec RPCGEN
    - Fichiers générés sont :
      - prog.h : déclarations des constantes et types utilisés dans le code généré pour le client et le serveur
      - prog_xdr.c : procédures XDR utilisés par le client et le serveur pour encoder/décoder les arguments,
      - prog_clnt.c : procédure stub côté client
      - prog_svc.c : procédure stub côté serveur

III. RPC sous UNIX / Mise en œuvre

- **Méthodologie de développement**
  - **Etape 2 : Ecriture des codes du serveur et client**
    - Ecrire le client (client.c) et le serveur (serveur.c)
    - serveur.c : implémentation de l'ensemble des procédures
  - **Etape 3 : Compilation**
    - Sur le site client, construire l'exécutable client (programme : fichier_client.c)

      gcc -o client fichier_client.c fichier_clnt.c
    - Sur le site serveur, construire l'exécutable serveur (programme : fichier_serveur.c)

      gcc -o server fichier_server.c annuaire_svc.c

III. RPC sous UNIX / Mise en œuvre

- **Méthodologie de développement**
  - **Etape 4 : Exécution**
    - Sur le site serveur, lancer l'exécutable serveur

      server &
    - Sur le site client, lancer l'exécutable client

      client

III. RPC sous UNIX / Mise en œuvre

- **Exemple**
  - Pour comprendre le fonctionnement des RPC, nous allons donc écrire, à titre d'exemple, un programme simple mais néanmoins complet .
  - Il y a 02 programmes:
    - un programme client
    - un programme serveur.
  - La fonction distante
    - prendra 02 nombres en paramètres et
    - renverra leur somme ainsi qu'un code d'erreur indiquant s'il y a eu un overflow ou non.

III. RPC sous UNIX / Mise en œuvre

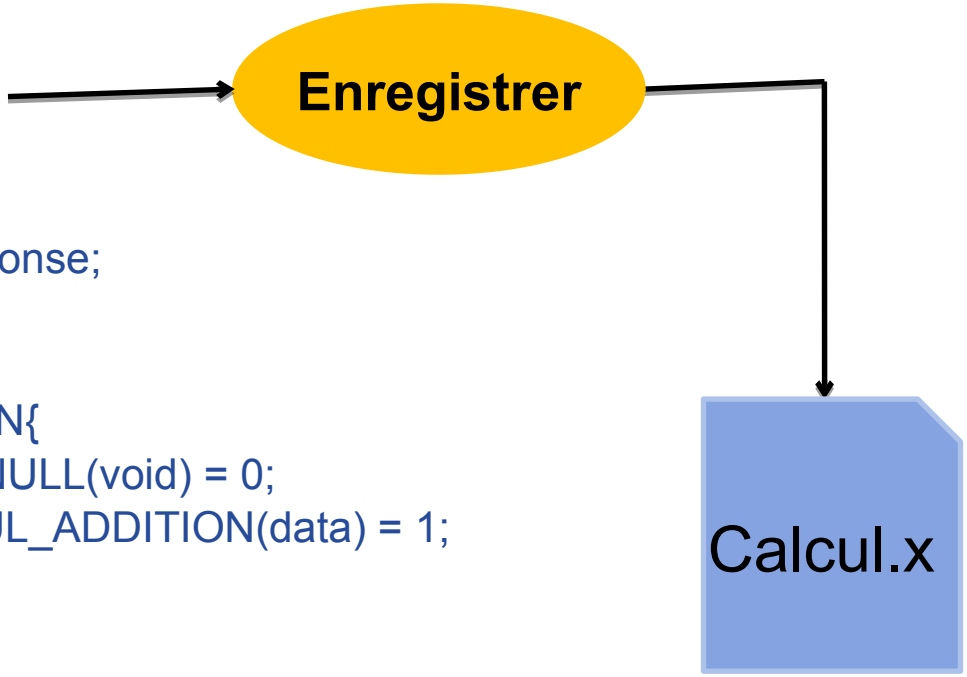
- **Etape de développement**
  - Définition de l'interface, écrite en utilisant l'IDL(Interface Definition Language)

```
struct data {
        unsigned int arg1;
        unsigned int arg2; };
typedef struct data data;

struct reponse {
        unsigned int somme;
        int errno; };
typedef struct reponse reponse;

program CALCUL{
        version VERSION_UN{
                void CALCUL_NULL(void) = 0;
                reponse CALCUL_ADDITION(data) = 1;
        } = 1;
} = 0x20000001;
```

III. RPC sous UNIX / Mise en œuvre

- **Etape de développement**
  - Définition de l'interface, écrite en utilisant l'IDL(Interface Definition Language)

```
struct data {
        unsigned int arg1;
        unsigned int arg2; };
typedef struct data data;

struct reponse {
        unsigned int somme;
        int errno; };
typedef struct reponse reponse;

program CALCUL{
        version VERSION_UN{
                void CALCUL_NULL(void) = 0;
                reponse CALCUL_ADDITION(data) = 1;
        } = 1;
} = 0x20000001;
```



III. RPC sous UNIX / Mise en œuvre

- **Etape de développement**
  - Définition de [illegible]DL(Interface Definition La[illegible]

Le programme s'appelle **CALCUL** et dans sa version VERSION_UN contient deux procédures:
CALCUL_NULL est toujours requise pour tester si le système marche (ping)
CALCUL_ADDITION.

```
struct data {
        unsigned [illegible]
        unsigned [illegible]
typedef struct da[illegible]

struct reponse {
        unsigned int somm[illegible]
        int errno; };
typedef struct repo[illegible]e reponse;

program CALCUL{
        version VERSION_UN{
                void CALCUL_NULL(void) = 0;
                reponse CALCUL_ADDITION(data) = 1;
        } = 1;
} = 0x20000001;
```

III. RPC sous UNIX / Mise en œuvre

- **Etape de développement**
  - Définition de l'interface, écrite en utilisant l'IDL(Interface Definition Language)

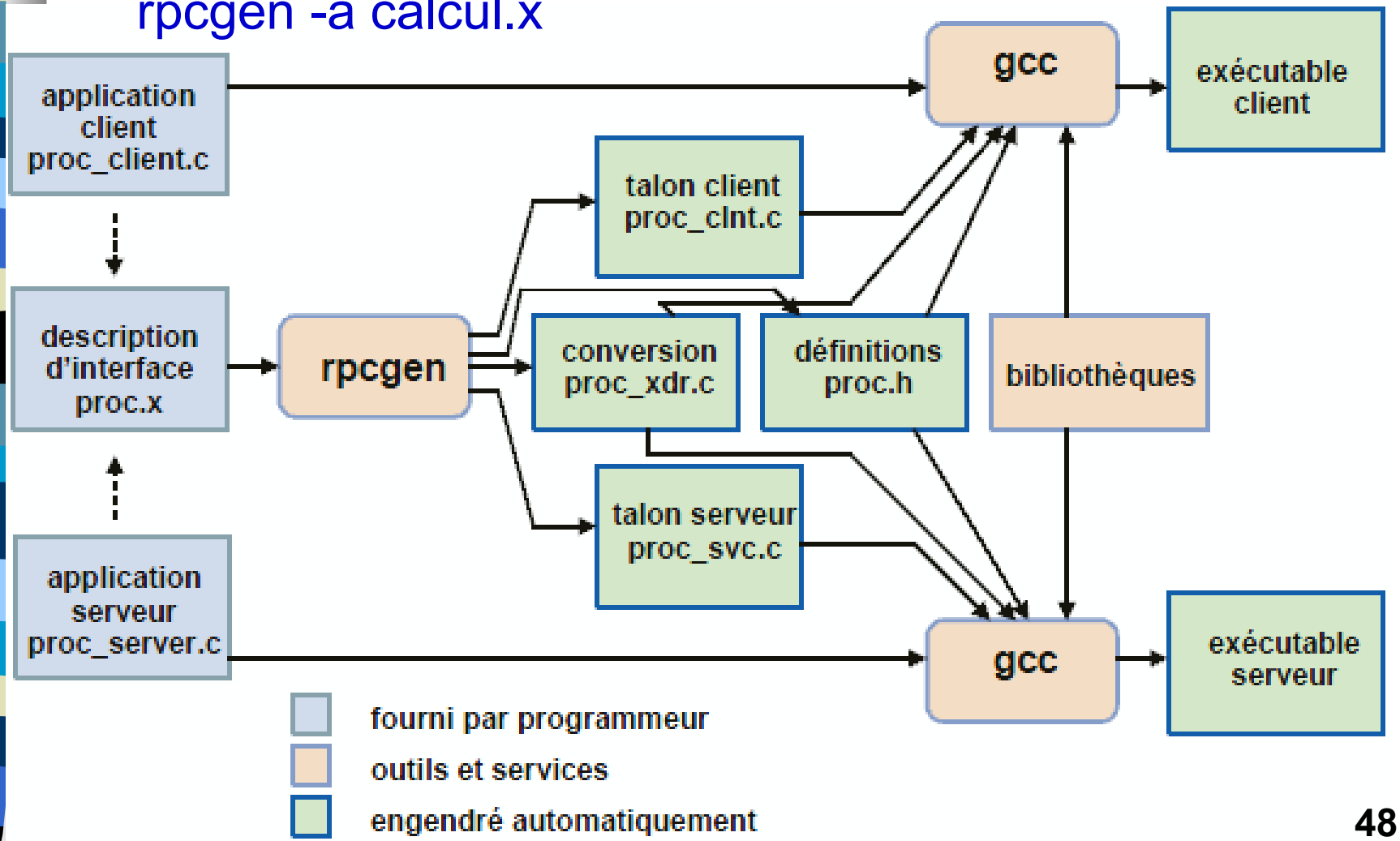
```
struct data {
        unsigned int arg1;
        unsigned int arg2; };
typedef struct data data;

struct reponse {
        unsigned int somme;
        int errno; };
typedef struct reponse rep

program CALCUL{
        version VERSION_UN{
                void CALCUL
                reponse CALCUL_ADDITION(data) = 1;
        } = 1;
} = 0x2000000
```

Le programme possède un numéro unique (0x20000001) dans le monde

III. RPC sous UNIX / Mise en œuvre

- **Traitement du fichier CALCUL.x par l'utilitaire**

rpcgen -a calcul.x



III. RPC sous UNIX / Mise en œuvre

- **Le format XDR est là pour géré l'hétérogénéité**
  - les types que nous avons définis nécessitent un filtre XDR, c'est le rôle des fonctions définies dans le fichier calcul_xdr.c, à compiler puis à lier avec le client et le serveur.
  - La compilation peut être faite maintenant (gcc -c calcul_xdr.c) et produit un calcul_xdr.o.
- **Compilation des tallons client et serveur**
  - gcc -c calcul_clnt.c
  - gcc -c calcul_svc.c
- **écrivons maintenant la fonction distante qui effectue réellement le travail.**
  - le squelette de la fonction est dans le fichier calcul_server.c

III. RPC sous UNIX / Mise en œuvre

- **calcul_serveur.c**

```
/* This is sample code generated by rpcgen.
* These are only templates and you can use them
* as a guideline for developing your own functions. */
#include "calcul.h"
void * calcul_null_1_svc(void *argp, struct svc_req *rqstp)
{ static char* result;
/*
* insert server code here
*/
return((void*) &result);}
reponse *calcul_addition_1_svc(data *argp, struct svc_req *rqstp)
{ static reponse result;
/*  insert server code here */
return(&result); }
```

III. RPC sous UNIX / Mise en œuvre

- **après modifications, on aurra calcul_server.c**

```
#include "calcul.h"
void * calcul_null_1_svc(void *argp, struct svc_req *rqstp) {
        static char* result;
        /* Ne rien faire */
        return((void*) &result); }
reponse *calcul_addition_1_svc(data *argp, struct svc_req *rqstp) {
        static reponse result;
        unsigned int max;
        result.errno = 0; /* Pas d'erreur */
        /* Prend le max */
        max = argp->arg1 > argp->arg2 ? argp->arg1 : argp->arg2;
        /* On additionne */  result.somme = argp->arg1 + argp->arg2;
        /* Overflow ? */ if ( result.somme < max ) { result.errno = 1; }
        return(&result);
}
```

III. RPC sous UNIX / Mise en œuvre

- **Compilation du fichier squelette du serveur**
  - gcc -c calcul_server.c
- **Édition de lien des fichiers objets générés par le compilateur c.**
  - gcc -o server calcul_svc.o calcul_server.o calcul_xdr.o
- **A ce stade, nous avons terminé la partie serveur de notre application. On peut alors démarrer le serveur :**
  - > ./server & (lancement du serveur)

III. RPC sous UNIX / Mise en œuvre

- **On peut utiliser rpcinfo pour vérifier qu'il tourne:**
  - > rpcinfo –p permet de connaître la liste des programmes RPC actuellement enregistrés sur la machine.
  - > rpcinfo -p

```
   program  vers    proto    port
   100000       2    tcp  111      rpcbind
   100000   2   udp      111      rpcbind
   100005   1   udp      673      mountd
   100005   2   udp      673      mountd
   100005   1   tcp  676      mountd
   100005   2   tcp  676      mountd
   100003   2   udp      2049     nfs
   100003   2   tcp  2049     nfs
   536870913    1   udp      897
   536870913    1   tcp  899
```

  - Les numéros de programmes sont affichés en décimal .
    - 536870913 est l'équivalent décimal de 0x20000001

III. RPC sous UNIX / Mise en œuvre

- **rpcinfo –u permet de tester programme indiqué en appelant sa procédure 0**
  - > rpcinfo -u localhost 536870913

    program 536870913 version 1 ready and waiting

III. RPC sous UNIX / Mise en œuvre

- **Modifications du squelette client**

```
#include "calcul.h"
Void calcul_1( char* host ) {
        CLIENT *clnt;
        void *result_1;
        char* calcul_null_1_arg;
        reponse *result_2;
        data calcul_addition_1_arg;
        clnt = clnt_create(host, CALCUL, VERSION_UN, "udp");
        if (clnt == NULL) { clnt_pcreateerror(host); exit(1); }
        result_1 = calcul_null_1((void*)&calcul_null_1_arg, clnt);
        if (result_1 == NULL) { clnt_perror(clnt, "call failed:"); }
        result_2 = calcul_addition_1(&calcul_addition_1_arg, clnt);
        if (result_2 == NULL) { clnt_perror(clnt, "call failed:"); }
        clnt_destroy( clnt );
}
```

III. RPC sous UNIX / Mise en œuvre

- **Modifications du squelette client**

```
main( int argc, char* argv[] ) {
        char *host;
        if(argc < 2) {
                printf("usage: %s server_host\n", argv[0]);
                exit(1);
        }
        host = argv[1];
        calcul_1( host );
}
```

III. RPC sous UNIX / Mise en œuvre

- **Pour faire un vrai programme, il nous faut donner des valeurs aux paramètres et il faut utiliser effectivement les résultats des appels distants (par exemple en les affichant à l'écran).**

III. RPC sous UNIX / Mise en œuvre

```
#include <limits.h>
#include "calcul.h"
CLIENT *clnt;
Void test_addition (uint param1, uint param2) {
reponse *resultat;     data parametre;
/* 1. Preparer les arguments */
parametre.arg1 = param1;   parametre.arg2 = param2;
printf("Appel CALCUL_ADDITION avec parametres: %u et %u \n",
     parametre.arg1,parametre.arg2);
/* 2. Appel de la fonction distante */
       resultat = calcul_addition_1 (&parametre, clnt);
        if (resultat == (reponse *) NULL) {
            clnt_perror (clnt, "call failed"); /* permet d'afficher l'erreur*/
            clnt_destroy (clnt); /*Libération du descripteur RPC*/
           exit(EXIT_FAILURE);
         }
         else if ( resultat->errno == 0 ) {
             printf("Le resultat de l'addition est: %u \n\n",resultat->somme);}
     else { printf("overflow \n\n"); }
}
```

III. RPC sous UNIX / Mise en œuvre

```
Int main (int argc, char *argv[]) {
        char *host;
        if (argc < 2) {
              printf ("usage: %s server_host\n", argv[0]);
              exit (1);
        }
        host = argv[1];
        /*Création d'un descripteur RPC*/
        clnt = clnt_create (host, CALCUL, VERSION_UN, "udp");
        if (clnt == NULL) {
               clnt_pcreateerror (host);
               exit (1);
        }
        test_addition ( UINT_MAX - 15, 10 );/* UINT_MAX valeur max d'un
            entier non signé*/
        test_addition ( UINT_MAX, 10 );
        /*Libération du descripteur RPC*/
        clnt_destroy (clnt);
        exit(EXIT_SUCCESS);
}
```

III. RPC sous UNIX / Mise en œuvre

Compilation du code source client

> gcc -c calcul_client.c

Edition de liens des codes objets client

> gcc -o client calcul_client.o calcul_clnt.o calcul_xdr.o

Lancement du programme exécutable client

> ./client localhost

Appel de la fonction CALCUL_ADDITION avec les parametres: 4294967280 et 10

Le resultat de l'addition est: 4294967290

Appel de la fonction CALCUL_ADDITION avec les parametres: 4294967295 et 10

La fonction distante ne peut faire l'addition a cause d'un overflow

>

Fin du Chapitre « RPC et RPC sous Unix»

Merci de votre attention