

***IUT 'A' Paul SABATIER***

**Dpt Informatique**

**S4**

***M4102C : Programmation répartie***

***TD2 : Les RPC (Remote Procedure Call)***

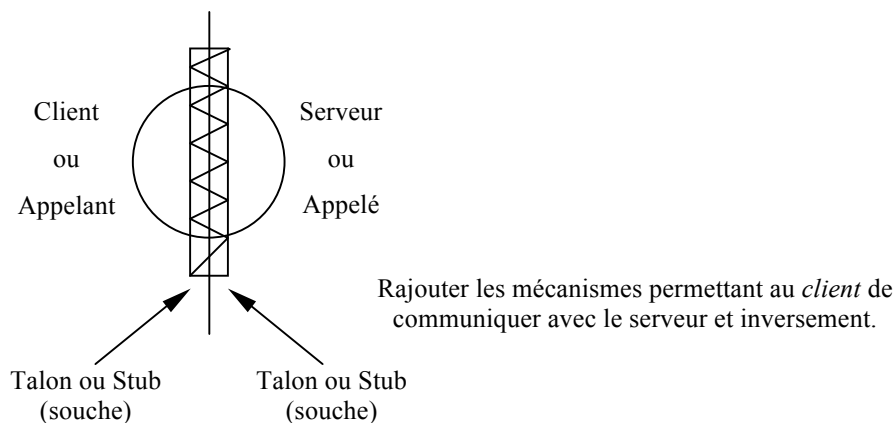
---

# UTILISATION DES RPC DE SUN

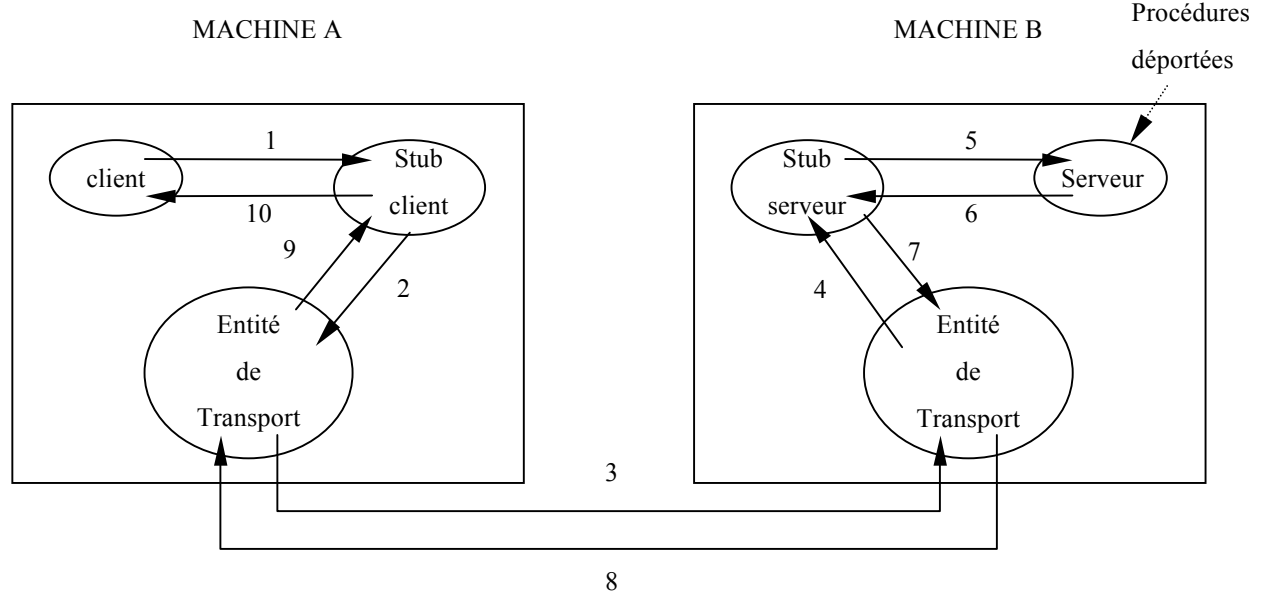
C'est un étudiant en thèse au MIT qui a écrit un compilateur : rpcgen.

Démarche : élaborer des programmes qui font appel à des procédures distantes.

On va séparer une application en 2 : programme appelant et programme appelé. Le programme appelé installe les procédures « déportées ». Le programme appelant fait référence aux procédures comme si elles étaient locales. Les talons sont chargés de faire le lien et d'encapsuler les paramètres échangés.



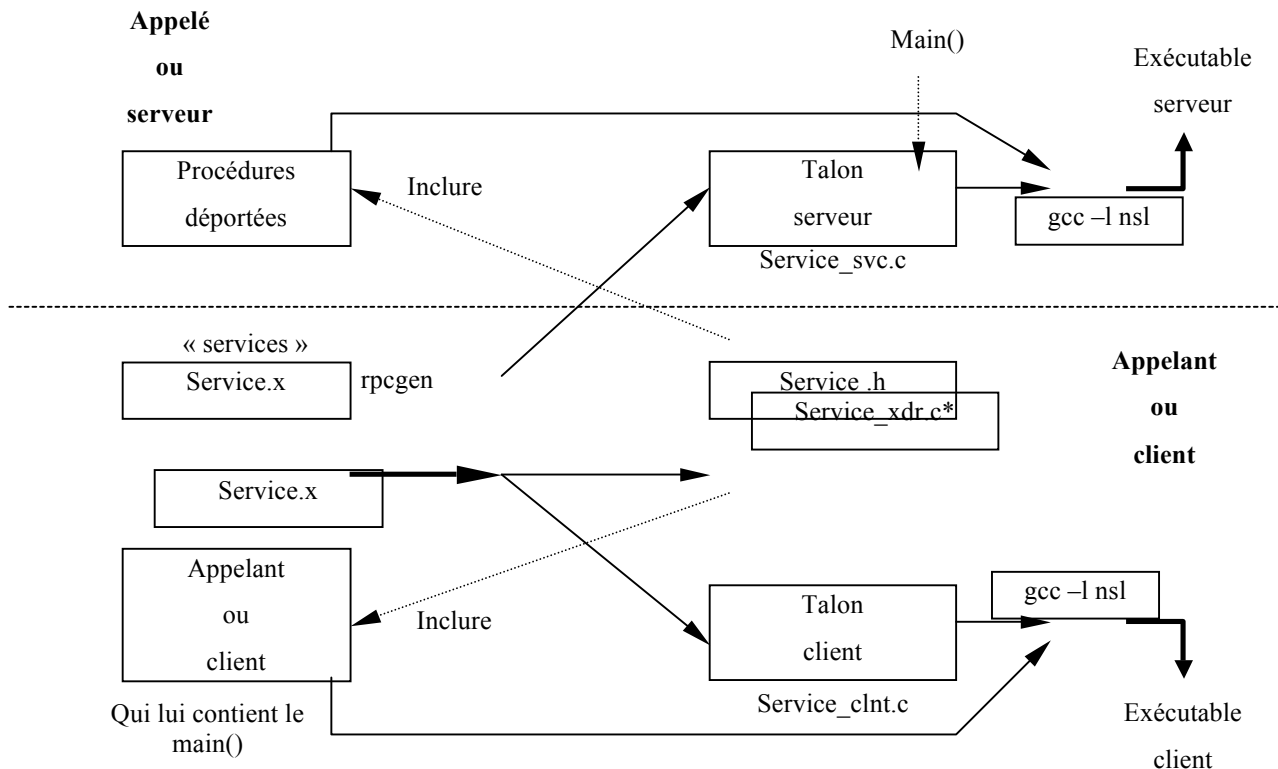
On va avoir 2 machines :



On va cacher cette complexité aux utilisateurs.

- 1 - Appel de la procédure *client-stub* avec les paramètres (permet de différencier les procédures locales).
- 2 - La procédure *stub* va construire un message avec les paramètres utilisateur : phase de marshalling.
- 3 - Transmission sur le réseaux.
- 4 - La procédure *serveur-stub* désencapsule le message.
- 5 - La procédure *serveur* appelle la fonction demandée avec les paramètres nécessaires.

On a découpé le programme en 2 morceaux :



Construire un fichier qui décrit le service.

Dans la description du service, on trouve :

- les procédures déportées,
- les paramètres.

On lui donne l'extension .x.

On passe ce fichier au compilateur rpcgen qui génère les talons.

Ex :

<b>service.x</b>	®	<b>service.h</b>	qui devra être inclus dans :
			- les procédures déportées,
			- le <i>main()</i> du <i>client</i> .
	®	<b>service_clnt.c</b>	
	®	<b>service_svc.c</b>	
	®	<b>service_xdr.c</b>	lorsque des types construits sont utilisés

`gcc procédures déportées + service_svc.c + service_xdr.c -lnsl`    ⤷    exécutable *serveur*.

`gcc main() du client + service_clnt.c + service_xdr.c -lnsl`    ⤷    exécutable *client*.

Protocole XDR : eXternal Data Representation ⤷ représentation des données sous la forme T,L,V.

Les procédures générées par rpcgen sont :

- le squelette *client* contenant la procédure « *Client-Stub* » (quelque chose qui permet de faire le lien),
- le squelette *serveur* contenant la procédure « *Serveur-Stub* »,
- les routines de filtre XDR pour les paramètres et les résultats,
- un fichier d'en-tête,
- facultativement, des moyens d'autorisations (le serveur vérifie si le client qui s'adresse à lui est autorisé ou pas).

#### Démarche :

DEFINIR LE CONTRAT DE SERVICE DANS LE FICHIER « *service.x* »

/\* Description du « service » \*/

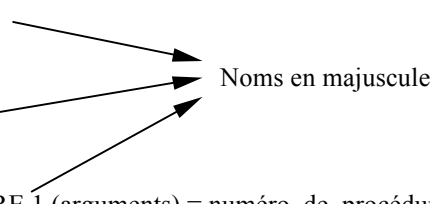
/\* Avoir les caractéristiques d'identification du *service distant* (programme, version(s), procédures déportées) écrit en un langage : RPCL (RPC Language) \*/

/\* **service.x** \*/

```

program NOM_PROGRAMME
{
    version NOM_VERSION
    {
        résultat PROCEDURE.1 (arguments) = numéro_de_procedure ;
        résultat PROCEDURE.2 (arguments) = numéro_de_procedure ;
        .....
    } = numéro_de_version ;
} = numéro_de_service ;

```



Noms en majuscule

\* numéro\_de\_procedure et numéro\_de\_version : on y met le numéro qu'on veut.

\* numéro\_de\_service sur 32 bits (4 octets) :

- de 0x 0000 0000 à 0x 1fff ffff : réservé à SUN
- de 0x 2000 0000 à 0x 3fff ffff : réservé à l'utilisateur
- de 0x 4000 0000 à 0x 5fff ffff : temporaires
- de 0x 6000 0000 à 0x 7fff ffff : RESERVE

#### Exemple :

/\* **ex01.x** : protocole d'impression de message distant (par exemple, envoi d'un message sur un terminal distant)\*/

```

program MESSAGE_PROG
{
    version MESSAGE_VERS
    {
        int PRINT_MESSAGE (string) = 1 ;
    } = 1 ;
} = 0x 20000001 ;

```

*rpcgen* génère : « *service.h* »

```
#define MESSAGE_PROG ((u_long) 0x 2000 0001)
#define MESSAGE_VERS ((u_long) 1)
#define PRINT_MESSAGE ((u_long) 1)
/* référence à la fonction déportée devant être appelée dans le programme client*/
extern int * print_message_1(char **, CLIENT *); /* pas de passage par valeur car pas de mémoire commune */
/* référence à la fonction déportée devant être appelée dans le programme serveur et qui devra être définie dans
service_proc.c */
extern int * print_message_1_svc(char **, struct svc_req *);

/* appellant.c */
/* Programme client */
#include <rpc/rpc.h>
#include "service.h"
main(int argc, char *argv[])
{
    CLIENT *cl ;    /* Pour un appel de procédure distante, le client et le serveur doivent être en phase */
                   /* Structure permettant d'établir une poignée entre le client et le serveur */
    int *result ;    /* Résultat de la procédure distante */
    /* Etablir la poignée avec le serveur */
    cl = clnt_create (HOST, PROGRAMME, VERSION, protocole de transport) ;
                   /* HOST : chaîne de caractères contenant le nom ou l'adresse de la machine où se trouve le serveur */
                   /* PROGRAMME, VERSION : on peut récupérer les constantes générées dans « service.h » */
                   /* Protocole de transport : « tcp » par exemple */
    if (cl == NULL)
    {
        /* Connexion impossible */
        clnt_pcreateerror (HOST) ;
        exit (1) ;
    }
    /* Appel à la procédure distante */
    result = procedure.1_1 (adresse des arguments, cl) ;
                                /* Nom de la procédure en minuscule */
                                /* cl : pour pouvoir faire référence à la poignée */
                                /* donc au serveur qui répond aux spécifications */
                                /* de programme et de version précisées dans cl */
    if (result == NULL)
    {
        /* Erreur pendant l'appel */
        clnt_perror (cl, HOST) ;    /* Un message est généré */
        exit (1) ;
    }
}
```

```

}

/* La procédure distante a été appelée correctement */
if (*result == 0)
{
    /* Le traitement n'a pas eu lieu */
    /* MESSAGE D'ERREUR */
    .....
    exit (1) ;
}
/* Tout s'est bien déroulé */
.....
exit (0) ;      /* Pour dire que ça s'est bien passé */
}

/* appelé.c */
/* Fichier des procédures déportées */
#include <rpc/rpc.h>
#include "service.h"

/* Remarque 1 : toute procédure distante accepte en entrée un pointeur sur les arguments qu'elle aurait eu en local */
                P indirection lors de l'appel

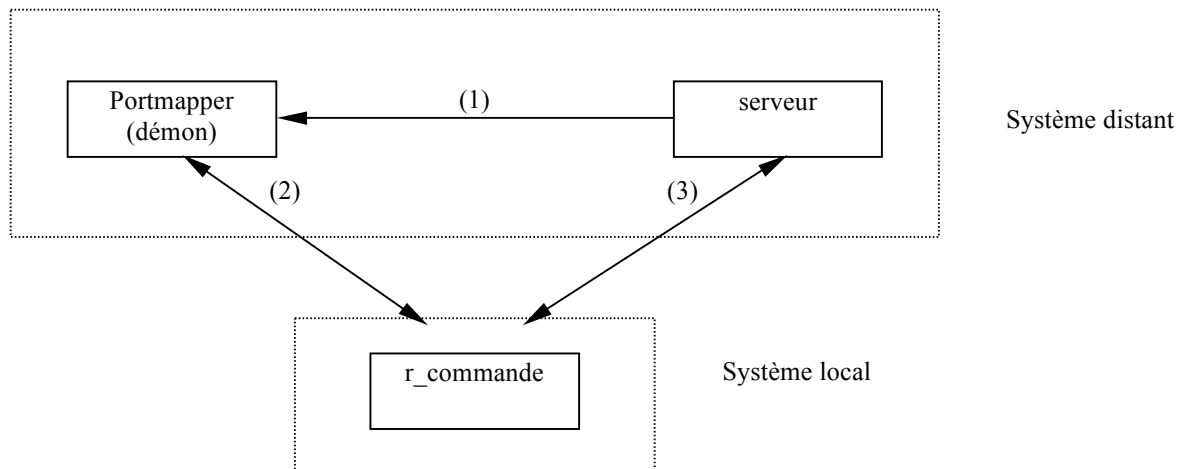
/* Remarque 2 : toute procédure distante retourne un pointeur sur les résultats qu'elle aurait retourné en local */
/* Remarque 3 : _numéro_de_version_svc est rajouté au nom local de la procédure */
int * procedure.1_1 _svc(pointeurs sur les arguments, struct svc_req *rqstp)
{
    static int result ;      /* result doit exister au delà de la fonction car on retourne son adresse */
                             /* En static, il est dans la zone de donnée et on peut y référer après */

    /* traitement */
    .....
    return (&result) ;
}

```

Ici, on a caché toute la lourdeur des SOCKETS.

## Mise en place du mécanisme



(1) : Enregistrement du serveur au niveau du portmapper par `svc_register()` @ stub serveur

(2) : *Client* : utilisation de `clnt_create` : on s'appuie sur le protocole qu'on veut : TPC, UDP, ...

(3) : Fonctions déportées : `printmessage_1()`

## Passage de paramètres

On a un argument et un résultat qui sont permis.

↳ Si on veut plusieurs arguments ou plusieurs résultats, il faut utiliser les structures C.

↳ Dans le « .x » on aura :

```
struct arguments
{
.....
}
struct result
{
.....
}
program P
{
    version V
    {
        struct result *f(struct arguments arg1) ;
        .....
    }
}
```

## **Mise en œuvre des exceptions**

Quand on utilise UDP, on a un mécanisme de time out et de retransmission (si nécessaire).

Erreur générée après un nombre déterminé de tentatives.

## **Sémantique des appels**

A chaque requête est associé un identifiant sur 32 bits appelé xid.

Permet de pouvoir identifier la requête et donc la réponse associée : ne pas exécuter la même fonction plusieurs fois.

Traite le problème des requêtes non idempotentes.

## **Rôle de XDR (eXternal Data Representation)**

XDR est une librairie qui définit des procédures permettant de transmettre des types de données simples en faisant abstraction des différences d'architectures matérielles (ordre des octets dans un mot de 32 bits représentant un entier long par exemple). Ces procédures sont utilisées pour coder les paramètres des procédures distantes et décoder leurs résultats.

XDR permet aussi l'utilisation de structures de données complexes en paramètres (et en retour) des procédures distantes. Elle permet de manipuler :

- des tableaux de taille fixe :

```
typedef int type_tab1[20];
```

- des tableaux contraints en taille

```
typedef int type_tab2<20>;
```

définit un int\* pouvant représenter des tableaux d'au plus 20 caractères.

- des chaînes de caractères contraintes en taille. Par exemple :

```
const MAXNOM = 255;
```

```
typedef string type_nom<MAXNOM>;
```

définit un type chaîne de caractère qui peut contenir au plus 255 caractères.

- des structures simples :

```
struct intervalle {  
    int borne_min;  
    int borne_max;  
};
```

- des unions avec discriminant

```
union resultat switch (int erreur) {  
    case 0:  
        /* si il n'y a pas d'erreur on renvoie une liste */  
        liste    une_liste;  
    default:  
        /* sinon on ne renvoie rien d'autre que l'erreur */  
        void;  
};
```

définit une union représentant une liste si la valeur de l'entier erreur est égale à 0 et rien si elle est différente de 0.



- des structures récursives (exemple liste chaînée)

```
typedef struct cellule *liste;
struct cellule {
    int         donnee;
    liste      cellule_suivante;
};
```

définit une liste simplement chaînée d'entiers qui sera envoyée automatiquement (on la construit, puis on passe en paramètre d'une fonction stub le début de la liste et XDR envoie toute la liste au serveur ; de façon symétrique, on construit une liste, puis on retourne le début de la liste et XDR envoie toute la liste au client).

## Sécurisation des accès au serveur de RPC

Avec un serveur généré par **rpcgen**, n'importe quel utilisateur ayant connaissance du numéro de programme, numéro de version et des numéros de procédures peut utiliser le serveur.

Pour éviter cela, les RPC gèrent un mécanisme d'autorisation qui permet de savoir quel est l'utilisateur d'un service donné (identification).

Ceci est réalisé en deux étapes :

- dans le client rajouter après le **clnt\_create** :

```
cl->cl_auth = authunix_create_default();
```

qui permet de sélectionner une identification UNIX simple (non cryptée) (par défaut aucune identification n'est utilisée)

- dans le serveur :

pour chaque procédure distante, un deuxième paramètre peut être déclaré pour obtenir des informations sur la requête :

exemple :

```
int * procedure.1_1 (pointeurs sur les arguments, struct svc_req *requete)
```

on peut l'utiliser pour :

1- tester le type d'identification utilisé

tester "requete->rq\_cred.oa\_flavor" qui peut être égal à AUTH\_NULL si aucune identification n'est utilisée ou à AUTH\_UNIX si une identification UNIX simple est utilisée

2- tester l'identificateur de l'utilisateur client

```
déclarer :
struct authunix_parms *aup;
```

recupérer les paramètres d'identification :

```
aup = (struct authunix_parms *)requete->rq_clntcred;
```

comparer par exemple l'uid du client : "aup->aup\_uid" avec l'uid  
du serveur obtenu par getuid()

3- tester d'autres informations : groupe ou nom de machine

REMARQUE : ce type d'identification n'est pas très difficile à outrepasser (en se faisant passer pour un autre utilisateur).  
Il existe d'autres mécanismes plus complexes (non présentés en TP).