# TP 1 – RPC

Gabriel Frey[1]

## 1 Appel distant en RPC

### 1.1 Rappels

Le but des RPC (Remote Procedure Call) est de permettre à un programme sur une machine d'appeller des fonctions situées sur une machine distante. Pour cela, il faut définir et enregistrer les fonctions sur ladite machine (appellée serveur) et les invoquer depuis un programme situé sur la machine appellante (le client).

#### 1.1.1 Serveur : enregistrement d'une fonction

Après avoir été définie, une fonction doit être enregistrée sur le serveur par :

```
int registerrpc (u_long no_pg, u_long no_vers, u_long no_proc,
        void * (* fonction) ( ), xdrproc_t xdr_param, xdrproc_t xdr_result) ;
```

**Paramètres**

1. *no_pg* : numéro du programme où enregistrer la fonction ;
2. *no_vers* : numéro de version ;
3. *no_proc* : numéro de procédure à donner à la fonction ;
4. *fonction* : pointeur sur la fonction à enregistrer ;
5. *xdr_param* : fonction de décodage des paramètres ;
6. *xdr_result* : fonction d'encodage du résultat.

Les trois premiers paramètres identifient la fonction de façon unique.

**Retour**

— 0 : en cas de **succès** ;
— $-1$ : en cas d'**erreur** et envoi d'un message d'erreur sur *stderr*.

**Remarques**

— Chaque fonction doit être enregistrée **individuellement**.
— Une procédure RPC ne peut avoir **qu'un seul paramètre** ! Utiliser un pointeur sur une structure si nécessaire.
— Lors du premier enregistrement de fonction, le numéro de programme est réservé et le procédure 0 est créé automatiquement (par convention, cette fonction ne prendra pas de paramètres et ne renvoie rien, elle sert uniquement à tester si un numéro de programme particulier existe).

#### 1.1.2 Serveur : mise en attente d'appel de fonction

La fonction suivante place le programme serveur en attente de demandes :

```
void svc_run() ;
```

Ne revient jamais sauf erreur grave.

1. Sujet créé à partir de documents de Guillaume Latu et de Kenneth Vanhoey

### 1.1.3 Client : appel distant

Pour invoquer une procédure RPC donnée, il faut appeller la fonction suivante :

```
int callrpc (char * machine, u_long no_prog, u_long no_vers, u_long no_proc,
             xdrproc_t xdr_param, char *param, xdrproc_t xdr_result, char *result) ;
```

**Paramètres**

1. *machine* : nom de la machine où se trouve la fonction à exécuter ;
2. *no_prog*, *no_vers*, *no_proc* : identifie la fonction à appeler ;
3. *xdr_param* : fonction d'encodage des paramètres ;
4. *param* : pointeur sur le paramètre à passer à la procédure ;
5. *xdr_result* : fonction de décodage du résultat ;
6. *result* : pointeur sur zone réservée pour stocker le résultat de la RPC.

Après l'appel :

6. *result* : pointe sur le résultat après exécution de la fonction distante.

**Retour**

— 0 : en cas de succès ;
— *autre* : en cas d'erreur (transtypé en *clnt_stat* et si passé à la fonction *clnt_perrno*, il y a affichage de l'erreur sur *stderr*).

On parle désormais d'une procédure RPC.

**Remarques**

— *callrpc* appelle la procédure RPC correspondante toutes les 5 secondes tant qu'il n'y a pas de réponse. Au bout de 25 secondes, un *timeout* est déclenché. Ainsi, un appel à *callrpc* peut entrainer au plus 5 appels à la RPC concernée et donc 5 exécutions sur le serveur.

## 1.2 Exercices : Premier exemple d'appels RPC

Soit le programme suivant, défini sur une machine distance dans le fichier *s*erver.c :

1. *server.c* :

```
#define PROGNUM 0x20000100
#define VERSNUM 1
#define PROCNUM 1

int * proc_dist(int *n)
{
  static int res = 1 ;
  printf("serveur: variable n (debut) : %d,\n",*n) ;
  res = *n + 1 ;
  *n = *n + 1 ;
  printf("serveur: variable n (fin) : %d,\n",*n) ;
  printf("serveur: variable res : %d\n",res) ;
  return &res ;
}

int main (void)
{
  registerrpc(   /* prognum */ PROGNUM,
                 /* versnum */ VERSNUM,
                 /* procnum */ PROCNUM,
                 /* pointeur sur fonction */  proc_dist,
```

```
                /* decodage arguments */ (xdrproc_t)xdr_int,
                /* encodage retour de fonction */ (xdrproc_t)xdr_int) ;

  svc_run() ; /* le serveur est en attente de clients eventuels */
}
```

2. *client.c* : Soit le programme suivant, défini sur la machine locale dans le fichier *c*lient.c :

```
#define PROGNUM 0x20000100
#define VERSNUM 1
#define PROCNUM 1

int main (int argc, char **argv)
{
  int res = 0,  n=0x41424344 ;
  char *host = argv[1] ;

  if (argc != 2)
  {
          printf("Usage: %s machine_serveur\n",argv[0]) ; exit(0) ;
  }

  printf("client: variable n (debut) : %d %s,\n",n,(char *)&n) ;

  stat = callrpc(/* host */ host,
                /* prognum */ PROGNUM,
                /* versnum */ VERSNUM,
                /* procnum */ PROCNUM,
                /* encodage argument */ (xdrproc_t)xdr_int,
                /* argument */ (char *)&n,
                /* decodage retour */ (xdrproc_t)xdr_int,
                /* retour de la fonction distante */(char *)&res) ;

  if (stat != RPC_SUCCESS)
  {
    fprintf(stderr, "Echec de l'appel distant\n") ;
    clnt_perrno(stat) ;
    fprintf(stderr, "\n") ;
    return 1 ;
  }

  printf("client: variable n (fin) : %d,\n",n) ;
  printf("client: variable res : %d\n",res) ;
}
```

**Questions**

1. La compilation du programme client sera faite par la ligne suivante :

   ```
   gcc -Wall -o client client.c -lrpcsvc -lnsl
   ```

   Expliquez cette ligne.
   Compiler l'exemple se trouvant dans le répertoire *1-exempleTP_ RPC/* :
   — Utiliser la commande **makefile** pour compiler ;
   — Lancer le serveur et constater qu'il est enregistré au niveau du serveur de liaison (le numéro de programme en décimale de votre service est le *536871168*) ;
   — Dans une autre console, lancer le client (avec comme argument le serveur *localhost*) et observer les écritures sur la sortie standard des deux côtés.

2. Déterminer le résultat des affichages respectifs sur la sortie standard (astuce : le code ASCII hexadécimal $0x41$ correspond au caractère 'A' ; et l'entier $Ox41424344$ s'écrit 1094861636 en décimale) ;

3. Discutez de la valeur finale de (*n) sur le serveur et sur le client ;

4. Les affichages du programme serveur et du programme client respectivement, diffèrent-ils en fonction de l'architecture de la machine support ?

5. Sur le programme serveur, pourquoi la variable *res* est-elle déclarée *static* ?

# 2 Fonctions d'encodage/décodage personnalisées

## 2.1 Rappels

Le protocole XDR (eXternal Data Representation) permet de traiter les problèmes de non unicité des représentations de données sur différentes architectures (taille des données et ordre des octets principalement). XDR procède via l'encodage des données vers un format conventionnel unique (grand boutiste, représentation IEEE pour les flottants, ...) avant l'envoi vers une machine distante, puis par décodage depuis ce format sur ladite machine.

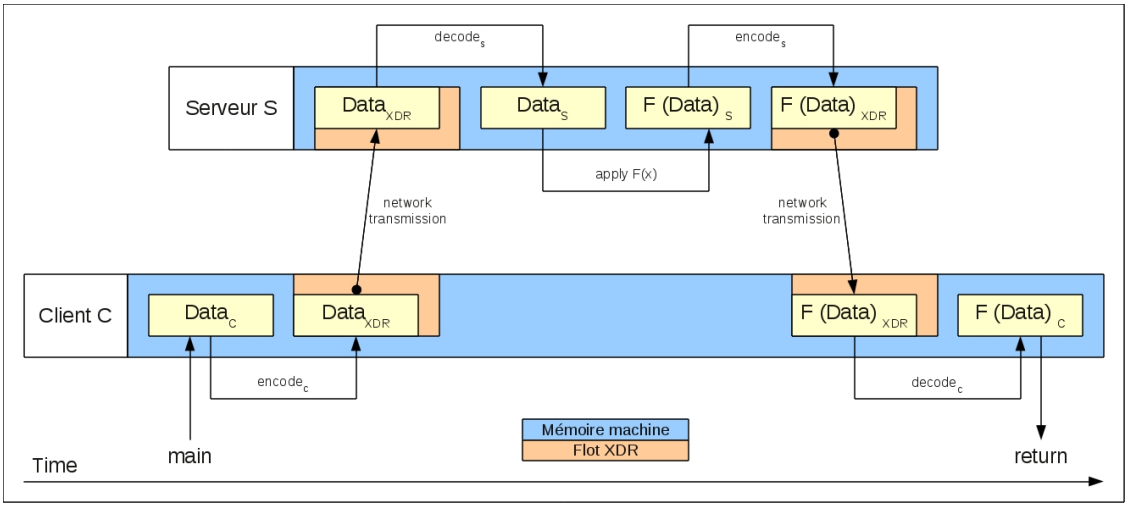


FIGURE 1 – Schéma décrivant le déroulement d'un appel RPC sur une machine distante et avec encodage XDR.

Pour cela, XDR nécessite d'avoir une plage mémoire (ou un espace disque) à disposition sur chaque machine afin de pouvoir y écrire et lire des données dans son format. Ces espaces sont appelés flots XDR. La structure de déscription de flot comprend entre autres les élements suivants :
— Un tampon (mémoire) ;
— Un curseur ;
— Un flag permettant de savoir s'il s'agit d'une zone d'encodage ou de décodage.

### 2.1.1 Création et utilisation d'un flot XDR

1. Déclaration des **structures de déscription de flots** $XDR$ (un pour l'encodage, un second pour le décodage) :

```
XDR xdr_encode, xdr_decode ;
```

2. **Création des flots** mémoire XDR :

```
void xdrmem_create(XDR* xdrs, char* addr, u_int size, enum xdr_op type_flot)
```

où $type\_flot$ peut prendre les valeurs $XDR\_ENCODE$, $XDR\_DECODE$ ou $XDR\_FREE$. Exemples :

```
xdrmem_create(&xdr_encode, tab, TAILLE, XDR_ENCODE) ; // Memoire d'encodage
xdrmem_create(&xdr_decode, tab, TAILLE, XDR_DECODE) ; // Memoire de decodage
```

3. **Déstruction d'un flot** mémoire XDR :

```
void xdr_destroy(XDR* xdrs)
```

### 2.1.2 Encodage/décodage dans un flot XDR

Les fonctions d'encodage/décodage, aussi appellés $filtres$, permettent d'encoder et décoder les types de base. Ce sont elles qui opèrent la conversion entre la représentation locale des données et celle d'XDR et vice-versa

```
bool_t (*xdrproc_type) (XDR* xdrs, void* val) ;
```

où le type $xdrproc\_type$ désigne de manière générique le pointeur sur les fonctions XDR pour encoder ou décoder les données.

Le tableau suivant liste les filtres prédéfinis pour quelques types de base :

| Type | Filtre | Type XDR |
|---|---|---|
| $char$ | $xdr\_char(XDR*, char*)$ | $int$ |
| $short$ | $xdr\_short(XDR*, short*)$ | $int$ |
| $u\_short$ | $xdr\_u\_short(XDR*, u\_short*)$ | $u\_int$ |
| $int$ | $xdr\_int(XDR*, int*)$ | $int$ |
| $u\_int$ | $xdr\_u\_int(XDR*, u\_int*)$ | $u\_int$ |
| $long$ | $xdr\_long(XDR*, long*)$ | $long$ |
| $u\_long$ | $xdr\_u\_long(XDR*, u\_long*)$ | $u\_long$ |
| $float$ | $xdr\_float(XDR*, float*)$ | $float$ |
| $double$ | $xdr\_double(XDR*, double*)$ | $double$ |
| $void$ | $xdr\_void(XDR*, void*)$ | $void$ |
| $enum$ | $xdr\_enum(XDR*, enum*)$ | $int$ |

Ces fonctions renvoient $True$ si tout se passe bien et $False$ en cas d'échec.

Exemple :

```
bool_t xdr_int(&xdrs, &entier) ;
```

Lorsque vous aurez à transmettre des données entre client et serveur qui sont une composition de types élémentaires, vous écrirez votre propre fonction d'encodage/décodage. Celle-ci aura généralement le prototype suivant :

```
bool_t mafonction (XDR *, (mastructure *) pointeur)
```

## 2.2 Exercices

Soit le code suivant :

```
#define LIRE 0
#define ECRIRE 1
#define TAILLE 256
#define LONGCHAINE 20

int main (int argc, char *argv[])
{
  XDR xdr_encode, xdr_decode ;
  char tab[TAILLE] ;
```

```
  int entier = -1001 ; float reel = 3.14 ;
  char *chaine0 = "Bingo !" ; char *chaine1 = "Re Bingo !" ;
  char *ptr0 = NULL ;  char *ptr1 = NULL ;

  /* Creation des flots XDR --------------- */
  xdrmem_create(&xdr_encode, tab, TAILLE, XDR_ENCODE) ;
  xdrmem_create(&xdr_decode, tab, TAILLE, XDR_DECODE) ;

  /* Encodage dans un flot XDR ------------ */
  if (!xdr_int(&xdr_encode, &entier))
         fprintf(stdout,"Erreur d'encodage de l'entier\n") ;
  if (!xdr_float(&xdr_encode, &reel))
         fprintf(stdout,"Erreur d'encodage du reel\n") ;

  if (!xdr_string(&xdr_encode, &chaine0, LONGCHAINE))
         fprintf(stdout,"Erreur d'encodage de la chaine 0\n") ;
  if (!xdr_string(&xdr_encode, &chaine1, LONGCHAINE))
         fprintf(stdout,"Erreur d'encodage de la chaine 1\n") ;

  /* Decodage du flot XDR ------------ */
  entier = 0 ; reel = 0 ;
  if (!xdr_int(&xdr_decode, &entier))
        fprintf(stdout,"Erreur de decodage de l'entier\n") ;
  else
        fprintf(stdout,"Entier lu : %d\n",entier) ;
  if (!xdr_float(&xdr_decode, &reel))
        fprintf(stdout,"Erreur de decodage du reel\n") ;
  else
        fprintf(stdout,"Reel lu : %f\n",reel) ;

  ptr1 = malloc(LONGCHAINE*sizeof(char)) ;

  fprintf(stdout,"les pointeurs sur les chaines : %x %x\n",ptr0,ptr1) ;
  if (!xdr_string(&xdr_decode, &ptr0, LONGCHAINE))
       fprintf(stdout,"Erreur decodage chaine 0\n") ;
  else
       fprintf(stdout,"Chaine lue : %s\n",ptr0) ;
  if (!xdr_string(&xdr_decode, &ptr1, LONGCHAINE))
       fprintf(stdout,"Erreur decodage chaine 1\n") ;
  else
       fprintf(stdout,"Chaine lue : %s\n",ptr1) ;

  fprintf(stdout,"les pointeurs sur les chaines : %x %x\n",ptr0,ptr1) ;

  xdr_free((xdrproc_t)xdr_string, (char*)&ptr0) ; // libère ce qui a été alloué
  xdr_free((xdrproc_t)xdr_string, (char*)&ptr1) ; // dans xdr_string (et met à NULL)

  free(ptr0) ; // Libère le pointeur alloué
  free(ptr1) ; // aucun effet car déjà à NULL

  xdr_destroy(&xdr_encode) ; // détruit le flot XDR
  xdr_destroy(&xdr_decode) ;

  return(0) ;
}
```

1. Compiler et tester l'exercice sur l'encodage/décodage XDR. Le fichier source et le makefile se trouvent dans le répertoire *2-exempleTP_XDR/*.
2. Définissez l'affichage à l'exécution ;
3. Définissez l'affichage à l'exécution lorsque l'on remplace la définition de $TAILLE$ par $\#define\, TAILLE\, 16$ ;

# 3 Division entière : RPC et filtres XDR

Définir un appel RPC qui permette de calculer, de façon distante, la division entière d'un nombre par un autre (en récupérant le quotient et le reste). Un squelette de programme est fourni dans le répertoire *3-divisionEntiere/*. Dans tous les fichiers (4 au total), il y a des « ? ? ? » à remplacer par le code correct.

Pour tester votre application distante, vous allez vous positionner sur deux machines : *turing* et *osr-etudiant.unistra.fr* (il y a un montage automatique des fichiers de l'un sur l'autre : les modifications sur l'un se répercutent immédiatement sur l'autre par l'intermédiaire du répertoire DocumentsLinux). Vous développerez votre code sur *turing* et exécuterez l'un de vos deux exécutables sur *osr-etudiant.unistra.fr*. Pour cela, gardez une console ouverte en permanence sur *osr-etudiant.unistra.fr* grâce à la commande *ssh osr-etudiant.unistra.fr*.

# 4 Exercice complet : RPC et XDR

Pour cet exercice, chaque sous-section se fera dans un répertoire différent, respectivement *4-Matrices22/* et *4-MatricesNN/*.

## 4.1 Calcul de matrices $2 \times 2$

Implantez un serveur de calcul sur des matrices $2 \times 2$ à coefficients réels et offrant comme services :
— la multiplication de deux matrices ;
— l'addition de deux matrices.
Le client offrira la possibilité d'appeler le service de multiplication ou d'addition, au choix.

## 4.2 Calcul de matrices $N \times N$

Reprenez la section précédente considérant que les matrices sont carrées et de taille $N \times N$. Mettez en évidence qu'il existe un time-out (au niveau du client) et une retransmission lorsque l'appel de fonction distant prend un peu de temps.

Déterminez la taille d'envoi maximal d'un bloc RPC, sachant qu'il s'agit toujours d'un multiple de 1024 octets, avec un minimum de 8Ko. Quelle est alors la taille $N_{max}$ maximale pour l'envoi d'un couple de matrices $N \times N$ ?

**Remarque** La commande **rpcgen** permet de générer des filtres XDR et de profils de fonctions automatiquement à partir d'une déscription générique. Si vous envisagez de faire votre projet en RPC, pensez à regarder cet outil.