TD 1 – RPC

Kenneth VANHOEY
https ://dpt-info.u-strasbg.fr/∼kvanhoey

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.

1.1.3 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.

   - -Wall : Warning All (Active tous les warning) ;
   - client est désigné comme l'exécutable à créer ;
   - client.c est désigné comme fichier source ;
   - -lrpcsvc : opère le lien avec la librairie de services RPC ;
   - -lnsl : opère le lien avec la librairie socket (utilisée par RPC).

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) ; Sur un PC (voir réponse à la question 4), l'affichage serait le suivant :

```
client : variable n (debut) : 1094861636 DCBA
client : variable n (fin) : 1094861636
client : variable res : 1094861637

serveur : variable n (debut) : 1094861636
serveur : variable n (fin) : 1094861637
serveur : variable res : 1094861637
```

3. Discutez de la valeur finale de (*n) sur le serveur et sur le client ;
   Ce qui pourrait surprendre ici, c'est que la variable (*n) n'ait pas été incrémentée chez le client après l'appel à la fonction. Un appel classique (en local et sans RPC) à la fonction distante aurait en effet résulté par une incrémentation de cette variable car la fonction accède à la valeur derrière le pointeur.
   En RPC cependant, la variable reçue en paramètre est d'abord copiée localement par le serveur avant d'être passée en argument à la fonction appellée. Ceci est logique vu qu'il n'y a aucune mémoire partagée entre la fonction appellante et la fonction appellée (le pointeur du serveur ne peut aucunement accéder à la case mémoire dans laquelle se situe la valeur de n chez le client).
4. Les affichages du programme serveur et du programme client respectivement, diffèrent-ils en fonction de l'architecture de la machine support ?

   Oui. En fonction du boutisme de la machine (c'est-à-dire si elle stocke les octets d'une variable entière de droite à gauche ou inversement), la ligne suivante affichera des résultats différents :

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

   – Sur un PC : « client : variable n (debut) : 1094861636 DCBA » ;
   – Sur un SPARC (SUN) : « client : variable n (debut) : 1094861636 ABCD ».
5. Sur le programme serveur, pourquoi la variable *res* est-elle déclarée *static* ?
   On ne peut déclarer la variable résultante que dans le corps de la procédure RPC. Or, l'accès doit pouvoir se faire en-dehors de celle-ci pour le renvoi du résultat au client. De plus, mis à part par le mécanisme transparent des RPC qui récupère le résultat de l'appel à la procédure, cette variable n'est accessible autrepart et ne peut donc être libérée. L'allocation dynamique permet la réutilisation de la même variable à chaque appel, ne multipliant ainsi pas les allocations mémoire. Ainsi, on privilégie une allocation statique à une allocation dynamique.

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.

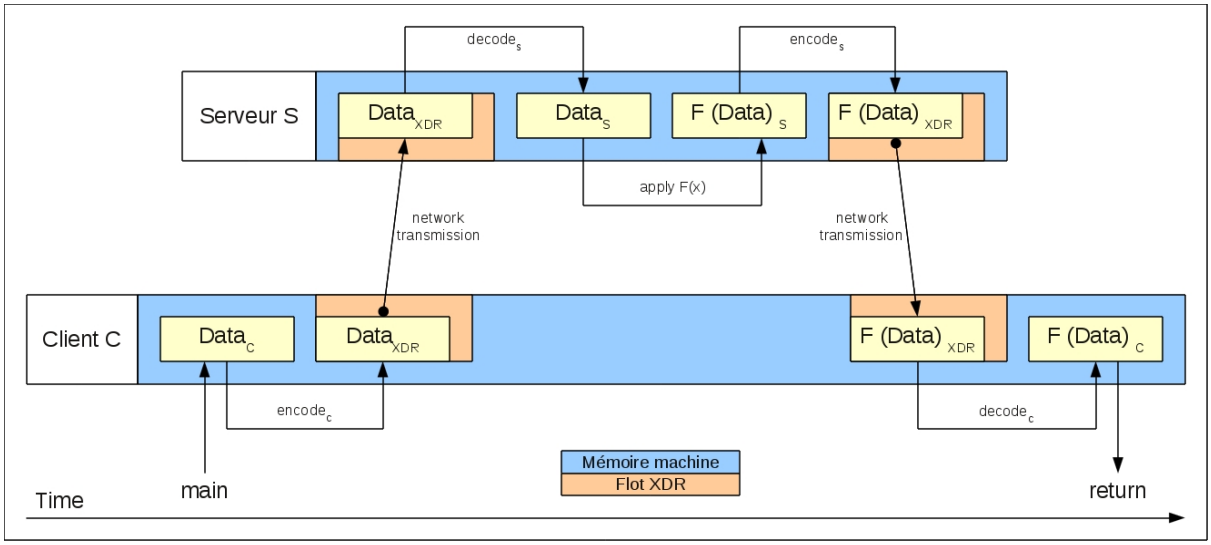

FIGURE 1 – Schéma décrivant le déroulement d'un appel RPC sur une machine distante et avec encodage XDR.

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. Définissez l'affichage à l'exécution ;

```
Entier lu : -1001
Reel lu : 3.140000
les pointeurs sur les chaines : 0 <adresse non nulle 1>
Chaine lue : Bingo !
Chaine lue : Re Bingo !
les pointeurs sur les chaines : <adresse non nulle 2> <adresse non nulle 1>
```

2. Définissez l'affichage à l'exécution lorsque l'on remplace la définition de $TAILLE$ par $\#define\ TAILLE\ 16$;

```
Erreur d'encodage de la chaine 0
Erreur d'encodage de la chaine 1
Entier lu : -1001
Reel lu : 3.140000
les pointeurs sur les chaines : 0 <adresse non nulle 1>
Erreur decodage chaine 0
Erreur decodage chaine 1
les pointeurs sur les chaines : <adresse non nulle 2> <adresse non nulle 1>
```

Justification Le buffer est de taille 16 octets. On y écrit successivement :
- Un entier (4 octets)
- Un réel à virgule flottante (4 octets)
- Une chaîne de caractères de taille 7+caractère de fin de chaîne (8 octets) + son entête de longueur (un entier : 4 octets).

Au total, on a écrit donc 20 octets dans le flot XDR, ce qui générè une erreur d'encodage de la chaine 0.

À tester : l'encodage de la chaîne 0 fonctionne avec $\#define\ TAILLE\ 20$.

Pour faire passer la chaîne 1, il faut non pas $\#define\ TAILLE\ 35\ (= 20 + 11 + 4)$, mais 36. Ceci est dû au fait que l'on encode toujours un multiple de 4 octets pour les strings (donc un string complété par des 0 si nécessaire). Voir RFC (Request For Comments) N° 1014.

Remarque Les filtres de décodage des types de base (entiers, caractères, réels), suppose que le pointeur passé en paramètre pointe sur un emplacement mémoire alloué. Pour les autres types (à allocation dynamqique), c'est le filtre qui se charge de l'allocation mémoire.

3. Vous voulez maintenant envoyer une donnée de couleur RVB (triplet de flottants correspondant aux intensités des couleurs rouge, vert et bleu) via votre mécanisme RPC. Définissez une structure adéquate et écrivez la fonction d'encodage/décodage en vous basant sur les filtres prédéfinis pour les flottants ;

```
struct colour
{
  float tab[3] ;
} ;

bool_t xdr_colour(XDR *xdrs, colour *m)
{
  return(     xdr_float(xdrs, &m->tab[0]) &&
              xdr_float(xdrs, &m->tab[1]) &&
              xdr_float(xdrs, &m->tab[2]) ;
}
```

4. Idem. pour des données de taille N (N-uplet).
   Vous devrez définir les éléments suivants :
   - La structure N-uplet ;
   - Un malloc pour allouer une zone mémoire afin d'accueillir cette structure (ainsi qu'une fonction libérant la mémoire) ;
   - La fonction d'encodage/décodage XDR.
     Astuce : dans la fonction d'encodage/décodage, il est possible d'accéder au type (encodage/décodage ou libération mémoire) du flot XDR via son paramètre x_op.

```
typedef struct
{
  int n ;
  float *tab ;
} dNuplet ;
```

```
dNuplet * malloc_Nuplet(int n)
{
  dNuplet *dN ;
  dN = (dNuplet *) malloc (sizeof(dNuplet)) ;

  dN->n = n ;
  dN->tab = (float*) malloc (n*sizeof(float)) ;

  fprintf(stderr,"zones allouees %x %x\n",(unsigned)dN, (unsigned)dN->tab) ;

  return (dN) ;
}

void freep(void *pointeur)
{
  fprintf(stderr,"zone desallouee %x\n",(unsigned)pointeur) ;
  free(pointeur) ;
}

bool_t xdr_Nuplet(XDR *xdrs, dNuplet **Nuplet)
{
  int i, j, n ;
  dNuplet *dN ;
  float *c ;

  switch (xdrs->x_op)
  {
    case XDR_ENCODE : fprintf(stderr,"xdr_Nuplet  : ENCODE(%d)\n",xdrs->x_op) ; break ;
    case XDR_DECODE : fprintf(stderr,"xdr_Nuplet  : DECODE(%d)\n",xdrs->x_op) ; break ;
    case XDR_FREE : fprintf(stderr,"xdr_Nuplet  : FREE(%d)\n",xdrs->x_op) ; break ;
    default : fprintf(stderr,"xdr_Nuplet  : default(%d)\n",xdrs->x_op) ; break ;
  }

  if (xdrs->x_op == XDR_ENCODE)
  {
    dN = *Nuplet ;
    n = dN->n ;
    if (!xdr_int(xdrs, &n))
      return FALSE ;
  }
  else if (xdrs->x_op == XDR_DECODE)
  {
    if (!xdr_int(xdrs, &n))
       return FALSE ;
    dNuplet = malloc_Nuplet(n) ;
    *Nuplet = dN ;
  }
  else if (xdrs->x_op == XDR_FREE)
  {
    if (dNuplet != NULL)
    {
      free(dNuplet) ;
    }
  }
```

```
  c = dN->tab ;
  for (i = 0; i < n; ++i)
  {
    if (!xdr_float(xdrs, c))
      return FALSE ;
    ++c ;
  }

  return TRUE ;
}
```

Remarque En-dehors de l'aspect pédagogique, la fonction d'encodage/décodage d'un tableau de taille N n'a pas beaucoup d'intérêt car la fonction xdr_array fournit ce service.

Remarque Un appel à un flot dont le type est XDR_FREE est fait automatiquement par le serveur lorsqu'il n'a plus besoin des variables XDR allouées. Ceci permet de libérer les variables que les filtres XDR ont alloués.