IUT 'A' Paul SABATIER

Dpt Informatique

S4

M4102C : Programmation répartie

TP 5&6 : Programmation RPC

Objectif :

Mise en œuvre du mécanisme simple de RPC du système **Linux**.

Travail demandé :

Développer en langage **C**, une application répartie mettant en œuvre le mécanisme de RPC avec utilisation de XDR (eXternal Data Representation). Programmation évoluée et ajustement de paramètres.

Spécifications :

1- Ecrire un serveur de RPC comportant les procédures suivantes :

a- ls(nom_repertoire) : récupère la liste des noms des fichiers (et repertoires) du repertoire <nom_repertoire> et l'envoie au client
utiliser pour cela les fonctions :
opendir : ouvre un répertoire
readdir : lit une entrée d'un répertoire et passe à l'entrée suivante
closedir : ferme un répertoire

b- read(nom_fichier) : lit un fichier par blocs de 1024 octets
et envoie au client la liste des blocs

c- write(nom_fichier, ecraser, donnees) : crée un fichier (si ecraser = 1) ou ajoute à la fin d'un fichier les données envoyées en paramètres (les données sont organisées en liste simplement chaînée comme au 1.b)

Pour cela on utilisera les possibilités suivantes de XDR :
Definition de structures de données complexes :
- chaînes de caractères contraintes en taille

```
const MAXNOM = 255;
typedef string type_nom<MAXNOM>;
```

définit un type chaine de caractères qui peut contenir au plus 255 caractères

- 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;
};
```

- 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).

2- Ecrire un client testant les procédures du serveur précédent

Remarque : libération des données allouées

La fonction xdr_free permet de libérer automatiquement des structures de données complexes déclarées dans le fichier .x

exemple :
si on utilise la liste d'entiers définie plus haut.

```
liste ma_liste;
... /* construction/utilisation de la liste */
  xdr_free(xdr_liste, &ma_liste); /* libère toute la liste */
```

3- Modifier le timeout par défaut

Par défaut, lors d'un appel distant, l'appel de la procédure clnt_call bloque pendant 25 secondes.
Si ceci est insuffisant (lors de gros calculs par exemple), on peut modifier le délai grâce à la fonction clnt_control :

Déclarer une structure représentant le nouveau délai :

```
struct timeval delai;
```

Choisir ce delai :

```
delai.tv_sec = 60; /* nombre de secondes */
  delai.tv_usec = 0; /* nombre de microsecondes */
  clnt_control(cl, CLSET_TIMEOUT, &delai);
  /* cl est un pointeur sur la structure CLIENT allouee lors
  du clnt_create */
```

4- Sécuriser les accès au serveur

Avec le serveur défini précédemment, n'importe quel utilisateur ayant connaissance du numéro de programme, numéro de version et des numéros de procédures peut utiliser votre serveur pour lire des répertoires, des fichiers et créer/écraser des fichiers vous appartenant...

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 cryptee) (par défaut aucune identification n'est utilisee)

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

exemple :
`ls_res *ls_1(type_nom *nom_rep, struct svc_req *requete)`
on peut l'utiliser pour :
1- tester le type d'identification utilise
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;
```
Récupérer les paramètres d'identification :
```
aup = (struct authunix_parms *)requete->rq_clntcred;
```

Comparer 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).