

# TP Réseau Application Client serveur RPC

---

Pierre Champion

## Introduction

Dans ce TP le but est de mettre en place une application RPC. Ce protocole est un protocole réseau qui est utilisé dans le modèle client-serveur pour assurer la communication entre le client et le serveur.

## Le protocole RCP

### Définition

Le protocole **RCP** ( **R**emote **P**rocedure **C**all ) permet d'invoquer des procédures sur un serveur (ordinateur distant). Son gros avantage est que le programmeur n'a pas à se charger de la partie purement connexion (exemple : Sockets), le protocole se charge de tout.

### Familiarisation

Dans un premier temps nous avons repris l'exemple du cours et nous l'avons mis en oeuvre. Cette application permet :

- D'additionner deux nombres
- De multiplier deux nombres

Pour comprendre le déroulement d'une application client-serveur voici les points importants :

- Un fichier .x qui définit notre application **RPC** contenant :
  - La structure des messages employés
  - Le prototype des procédures qui pourront être invoquées par le client et leur numéro (unique)
  - Le numéro de notre programme
  - Le numéro de version de notre programme

Le numéro de programme doit être choisi entre `0x20000000` à `0x3FFFFFFF` , cet intervalle est réservé pour l'utilisateur.

Dans le cadre du programme du cours les données envoyées au serveur sont 2 entiers et nous recevons en retour du serveur un entier (le résultat)

```

/*
Fichier add.x
Ce fichier decrit l'application client-serveur RPC
*/
struct intpair{ /* Structure de donnees client --> serveur */
    i nt a;
    i nt b;
};
program MATHPROG
{
    version MATHVERS
    {
        i nt MATHPROC_ADD(intpair) = 1; /*Procedure 1 : addition*/
        i nt MATHPROC_MULT(intpair) = 2; /*Procedure 2 : multiplication*/
    } = 1; /* Numero de version */
} = 0x20000001; /* Numero de programme */

```

Une fois le fichier `add.x` rempli et à l'aide de la commande `rcpgen add.x -a` pour traiter le fichier `.x`. L'option `-a` permet de générer des codes sources d'exemples. Les fichiers suivants sont alors générés :

```

Makefile.add add_client.c add_clnt.c add.h
add_server.c add_svc.c add.x add_xdr.c

```

```

#include "maths.h"
/* Procedure d'addition */
int * mathproc_add_1_svc(intpair *argp, struct svc_req *rqstp) {
    static int result;
    /* La fonction renvoie l'addition des deux entiers */
    result = argp->a + argp->b;
    return &result;
}

```

La procedure d'addition a été écrite dans le fichier `add_server.c` (générer pas `rcpgen`) Pour faire fonctionnée cette application avec un client on utilisera la routine `clnt_create`, à partir de l'exemple de `rpcgen`. La routine prend 4 arguments :

- le nom de l'hôte
- le programme en question
- la version du programme
- le type de protocole (UDP ou TCP)

Le fichier `add_client.c` est donc le suivant :

```
/*
 * Programme cotée client
 */

#include "maths.h"

int main(int argc, char *argv[]){
    char *host;
    CLIENT * cl;
    intpair paire_entiers;
    int * resultat;
    if (argc < 4) {
        printf ("usage: %s <server_host> <entier a> <entier b>\n", argv[0]);
        exit (1);
    }

    host = argv[1];
    cl = clnt_create(host, MATHPROG, MATHVERS, "udp"); /* Creation du client */
    if (cl == NULL) {
        clnt_pcreateerror(argv[1]);
        exit(1);
    }

    paire_entiers.a = atoi(argv[2]); /* Initialisation des champs de */
    paire_entiers.b = atoi(argv[3]); /* la structure paire_entiers*/
    resultat = mathproc_add_1(&paire_entiers, cl); /* procedure 1 */
    if (resultat == NULL) {
        clnt_perror(cl, "add");
        exit(1);
    }

    printf("--> L'addition retourne: %d\n", *resultat);
    resultat = mathproc_mult_1(&paire_entiers, cl); /* procedure 2 */
    if (resultat == NULL) {
        clnt_perror(cl, "mult");
        exit(1);
    }
    printf("--> La mutipltcation retourne : %d\n", *resultat);
    exit (0);
}
```

Il ne nous reste alors plus qu'à recompilier l'application en utilisant le `makefile` généré par rpcgen  
`make -f Makefile.add` . Le test du bon fonctionnement de l'application nous montre que tout marche bien :

```
$ ./add_server
$ ./add_client 127.0.0.1 42 50
--> L'addition retourne : 92
--> La mutipltcation retourne : 2100
```

# Filtrage

`rpcgen` se charge de la partie client-serveur, nous pouvons donc nous concentrer sur le filtrage d'image.

## Filtre moyeneur

Le niveau de gris du pixel central est remplacé par la moyenne des niveaux de gris des pixels environnants.

## Filtre médian

Le filtre médian est un filtre numérique non linéaire, souvent utilisé pour la réduction de bruit.

## Réalisation

Le contenu du fichier `image.x` est le suivant :

```
struct image_t {
    opaque bytes<>;
    int height;
    int width;
};

struct image_filtre_t {
    image_t image;
    int hpad;
    int wpad;
};

program IMAGEPROG {
    version IMAGEVERSION {
        image_t moyenne(image_filtre_t) = 1;
        image_t mediane(image_filtre_t) = 2;
        image_t min(image_filtre_t) = 3;
        image_t max(image_filtre_t) = 4;
    } = 1;
} = 0x200000002;
```

Le code de la fonction `xdr_image.c` produite par `rpcgen` permet de s'assurer que la taille d'un élément n'a pas changer entre l'envoi et la réception

# Conclusion

Les **RPC** permettent un gain de temps énorme, ils permettent de réaliser une application client-serveur sans connaître les *sockets*. Cependant ce n'est pas compatible entre les différents systèmes d'exploitation comme Windows et Unix.