Introduction

- Primitive de base d'interaction entre éléments logiciels
  - Appel d'une procédure/fonction, exemple :

```
...
int resultat;
resultat = calculPuissance(2, 3);
printf('' 2 à la puissance 3 = %d\n'', resultat);
...
```

  - Ici `calculPuissance` est une fonction qui est appelée localement
    - Son code est intégré dans l'exécutable compilé ou chargé dynamiquement au lancement (librairie dynamique)
- Sockets TCP & UDP
  - Communication par envoi et réception de bloc de données
  - Bien plus bas niveau qu'appel de fonction

Introduction

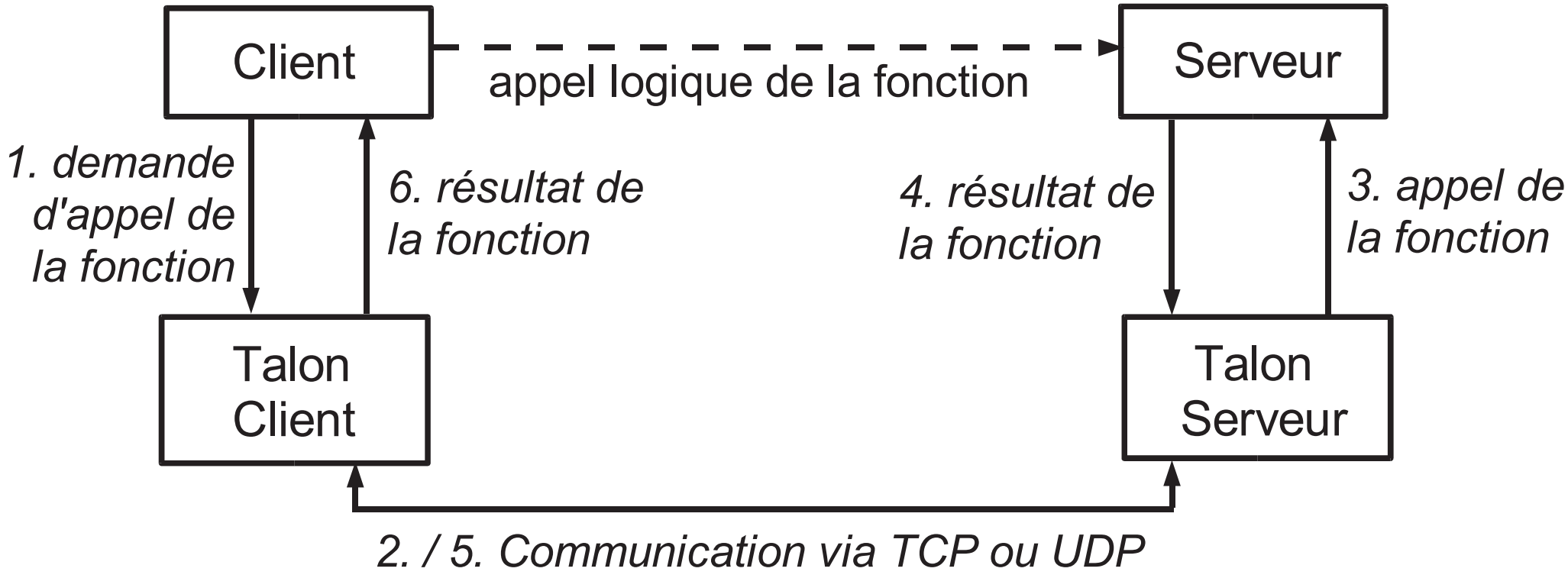
- Application distribuée client/serveur du TP 3
  - Un client veut calculer 3 à la puissance de 2
  - Le serveur est capable de faire ce calcul
  - Le client construit une requête sous la forme d'un tableau d'octets structuré envoyé au serveur via une socket
  - Le serveur décode la requête
  - Il exécute le service requis et renvoie via une socket sous forme de tableau d'octets structuré le résultat au client
  - De manière « abstraite » : le client a appelé un service de calcul sur le serveur
    - Via une mécanique dédiée de communication utilisant les sockets
  - Idée
    - Pouvoir directement appeler ce service (une fonction C) sur le serveur presque aussi facilement que si ce service était local

RPC : Remote Procedure Call

- Idée générale
  - Pouvoir appeler « presque » aussi facilement une fonction sur un élément distant que localement
- Principes
  - On différencie le coté appelant (client) du coté appelé (serveur)
    - Appelé offre la possibilité à des éléments distants d'appeler une ou plusieurs fonctions chez lui
  - Le coté client appelle localement la fonction sur un élément spécial qui relayera la demande d'appel de fonction coté serveur
  - Coté serveur, un élement spécial appellera la fonction et renverra le résultat coté client
  - Eléments spéciaux : talons (ou *stubs*)

Remote Procedure Call

- Fonctionnement général d'un appel de fonction



- Communication via sockets TCP ou UDP
  - Pour faire transiter les demandes d'appels de fonction et le résultat des appels
  - Toutes les données transitant via les sockets sont codées via XDR

RPCGEN

- RPCGEN
  - RPC Generator
  - Utilitaire permettant de générer automatiquement le code des talons et des fonctions XDR associées aux données utilisées par les fonctions
- Principe d'utilisation
  - On décrit dans un fichier (d'extension .x)
    - Les structures de données propres aux fonctions
    - Les fonctions appelables à distance
  - RPCGEN génère ensuite un ensemble de fichiers

Exemple

- Fonctionnement de RPC et de `rpcgen` par l'exemple
  - Ensemble de fonctions de traitement de formes géométriques appelables à distance par des clients
    - ```
      int surface_rectangle(struct rectangle rect);
      struct rectangle creer_rectangle(int x1, y1, x2, y2);
      booleen inclus(struct rectangle r, struct point p);
      ```
  - Avec les structures et types suivants
    - ```
      struct point {
          int x, y;
      };
      ```
    - ```
      struct rectangle {
          struct point p1, p2;
      };
      ```
    - ```
      typedef int booleen;
      ```

Exemple : fonctions originales

- Code des 3 fonctions
  - On supposera que pour un rectangle, le point `p1` est le coin inférieur gauche et `p2` le supérieur droit
    - ```
      int surface_rectangle(struct rectangle rect) {
        return abs((rect.p1.x - rect.p2.x) *
                   (rect.p1.y - rect.p2.y));
      }
      ```
    - ```
      struct rectangle creer_rectangle(int x1, int x2,
                                       int y1, int y2) {
        struct rectangle rect;
        rect.p1.x = x1; rect.p1.y = y1;
        rect.p2.x = x2; rect.p2.y = y2;
        return rect;
      }
      ```
    - ```
      booleen inclus(struct rectangle rect, struct point p){
         return ( (p.x >= rect.p1.x) && (p.x <= rect.p2.x)
               && (p.y >= rect.p1.y) && (p.y <= rect.p2.y));
      }
      ```
      ```

Règles d'écriture du fichier .x

- Fichier décrivant les fonctions et données
  - Pour données
    - Décrit les structures presque comme en C (voir suite pour détails)
  - Pour fonction : règle fondamentale
    - Une fonction ne prend qu'UN seul paramètre
    - On doit donc définir une structure de données dédiée à la fonction si on veut passer plusieurs valeurs en paramètre
  - Définition d'un « programme » RPC
    - Programme = ensemble de fonctions/services
    - Chaque programme possède un nom avec un numéro associé
    - Un programme peut exister en plusieurs versions
      - Chaque version est identifiée par un nom et un numéro associé
      - Chaque version définit une liste de fonctions
      - Chaque fonction est identifiée par un numéro unique

Exemple : geometrie.x

- Fichier `geometrie.x`

```
struct point {
  int x;
  int y;
};

struct rectangle {
  struct point p1;
  struct point p2;
};

struct coordonnees {
  int x1;
  int x2;
  int y1;
  int y2;
};
```

```
struct param_inclus {
    struct rectange rect;
    struct point p;
};

typedef int booleen;
```

Exemple : geometrie.x

- Fichier `geometrie.x` (suite)

```
program GEOM_PROG {
  version GEOM_VERSION_1 {
    int SURFACE_RECTANGLE(rectangle) = 1;
    rectangle CREER_RECTANGLE(coordonnees) = 2;
    booleen INCLUS(param_inclus) = 3;
  } = 1;
} = 0x20000001;
```

  - Nom du programme : GEOM_PROG d'identifiant 20000001 (en hexa)
  - Nom de version : GEOM_VERSION_1 de numéro 1
  - Les 3 fonctions sont numérotées 1, 2 et 3
  - Les structures `coordonnees` et `param_inclus` ont été créées pour les fonctions nécessitant plus de 2 paramètres
  - Par convention, on écrit les noms de fonctions, programmes et versions en majuscule

Génération des fichiers

- Pour générer les fichiers avec `rpcgen`
  - `$ rpcgen geometrie.x`
- Fichiers générés à partir du .x
  - `geometrie.h`
    - Définition de toutes les structures et des signatures des opérations
  - `geometrie_xdr.c`
    - Fonctions de codage XDR des structures de données
  - `geometrie_clnt.c` et `geometrie_svc.c`
    - Talons cotés client et serveur
- Avec option `-a` de `rpcgen`
  - Génére fichier `geometrie_server.c`
    - Squelette pour écriture des fonctions du programme

Exemple : fichier geometrie.h

```
typedef int booleen;

struct point {
  int x;
  int y;
};
typedef struct point point;
bool_t xdr_point(XDR*, point*);

struct rectangle {
  struct point p1;
  struct point p2;
};
typedef struct rectangle rectangle;
bool_t xdr_rectangle(XDR*, rectangle*);

struct coordonnees {
  int x1;
  int x2;
  int y1;
  int y2;
};
typedef struct coordonnees coordonnees;
bool_t xdr_coordonnees(XDR*, coordonnees*);
```

Exemple : fichier geometrie.h (suite)

```
struct param_inclus {
  struct rectangle rect;
  struct point p;
};
typedef struct param_inclus param_inclus;
bool_t xdr_param_inclus(XDR*, param_inclus*);

typedef int booleen;
bool_t xdr_booleen(XDR*, booleen*);

#define GEOM_PROG 0x20000001
#define GEOM_VERSION_1 1

#define SURFACE_RECTANGLE 1
extern int *surface_rectangle_1(rectangle*, CLIENT*);
extern int *surface_rectangle_1_svc(rectangle*, svc_req*);

#define CREER_RECTANGLE 2
extern rectangle *creer_rectangle_1(coordonnees*, CLIENT*);
extern rectangle *creer_rectangle_1_svc(coordonnees*,svc_req*);

#define INCLUS 3
extern booleen *inclus_1(param_inclus*, CLIENT*);
extern booleen *inclus_1_svc(param_inclus*, svc_req*);
```

Contenu fichier .h

- Pour chaque structure définie
  - Définition de la structure équivalente en C
  - Définition d'un type du même nom correspondant à la structure
  - Signature de la méthode XDR correspondant à ce type
  - Exemple, dans .x
    - ```
      struct point {
        int x;
        int y;
      };
      ```
  - Devient dans le .h
    - ```
      struct point {
        int x;
        int y;
      };
      typedef struct point point;
      bool_t xdr_point(XDR*, point*);
      ```

Contenu fichier .h

- Pour chaque méthode
  - Dans .x
    - Pour chaque opération nommée `fonction`, de version numérotée `x`, qui prend un `type_param` en paramètre et retourne un `type_retour`
    - On aura dans le .h, signature de 2 méthodes correspondantes
    - `type_retour *fonction_x(type_param *, CLIENT *)`
      `type_retour *fonction_x_svc(type_param *, struct svc_req *)`
  - Dans les 2 cas
    - Le nom de la fonction est suffixée par la version (et svc pour la seconde)
    - Un niveau de pointeur en paramètre et retour est ajouté par rapport au .x

Contenu fichier.h

- Pour chaque méthode (suite)
  - Première version : `fonction_x`
    - Fonction qui est implémentée par le talon client
    - C'est cette fonction que la partie client appelle localement pour demander l'appel de la fonction associée coté serveur
    - Paramètre `CLIENT*`
      - Caractéristiques de la communication avec la partie serveur
  - Deuxième version `fonction_x_svc`
    - Fonction à implémenter coté serveur
      - Fonction qui est appelée par les clients
    - Paramètre `svc_req*` : caractéristiques de la requête d'appel de service et identification du client
  - Exemple
    - geometrie.x : `int SURFACE_RECTANGLE(rectangle) = 1;`
    - geometrie.h
      - `int surface_rectangle_1(rectangle *, CLIENT *);`
      - `int surface_rectangle_1_svc(rectangle *, svc_req *);`

Contenu fichier .h

- Définitions de constantes
  - Numéros de programme et de versions en associant chaque numéro au nom précisé dans .x
    - Exemple
      - ```
        #define GEOM_PROG 0x20000001
        #define GEOM_VERSION_1 1
        ```
  - Pour chaque fonction, définition d'une constante associant son nom à son numéro
    - Exemple
      - Dans .x
        - ```
          rectangle CREER_RECTANGLE(coordonnees) = 2;
          ```
      - Dans .h
        - ```
          #define CREER_RECTANGLE 2
          ```
  - Toutes ces constantes serviront à identifier le programme et les fonctions lors des appels à distance
          ```

Contenu fichier_xdr.c

- Fichier `geometrie_xdr.c`
  - Contient les méthodes de codage XDR pour chaque type de données décrit dans le .x
  - Exemple
    - Dans `geometrie.x`
      - ```
        struct point {
          int x;
          int y;
        };
        ```
    - Dans `geometrie_xdr.c`
      - ```
        bool_t xdr_point(XDR* xdrs, point* objp)  {
          if (!xdr_int(xdrs, &objp->x)) {
            return (FALSE);
          }
          if (!xdr_int(xdrs, &objp->y)) {
            return (FALSE);
          }
          return (TRUE);
        }
        ```

Implémentation des fonctions

- Cote serveur, dans un fichier à part
  - Ou dans le squelette coté serveur si on l'a généré
  - Pour chacune des fonctions du programme, on implémente le code en prenant la signature du .h
- Exemple
  - Dans .h

```
extern int *surface_rectangle_1_svc(rectangle*, CLIENT*);
extern rectangle *creer_rectangle_1_svc(coordonnees*,
                                                CLIENT*);
extern bool_t *inclus_1_svc(param_inclus*, CLIENT*);
```

  - Implémentation de ces fonctions dans fichier `geometrie_server.c`

Exemple : fichier geometrie_server.c

```
#include "geometrie.h"

int *surface_rectangle_1_svc(
                    rectangle *rect, svc_req *req) {
  static int result;
  result = (rect -> p1.x - rect -> p2.x) *
           (rect -> p1.y - rect -> p2.y);
  return &result;
}

rectangle creer_rectangle_1_svc(
                    coordonnees *coord, svc_req *req) {
  static rectangle rect;
  rect.p1.x = coord -> x1; rect.p1.y = coord -> y1;
  rect.p2.x = coord -> x2; rect.p2.y = coord -> y2;
  return &rect;
}
```

Fichier geometrie_server.c

```
booleen inclus_1_svc(param_inclus *param, svc_req *req) {
  static booleen result;
  result = (param -> p.x >= param -> rect.p1.x) &&
           (param -> p.x <= param -> rect.p2.x) &&
           (param -> p.y >= param -> rect.p1.y) &&
           (param -> p.y <= param -> rect.p2.y));
  return &result;
```

- Notes
  - On a pas d'utilité ici à manipuler le paramètre `svc_req`
  - Les variables `result` sont déclarées en `static`
    - On doit retourner un pointeur sur une donnée qui existera toujours après l'appel de la fonction
    - Cette donnée sera retournée par copie au client

Coté client

- Pour appel d'un service `fonction` sur la partie serveur
  - Appelle simplement la fonction `fonction_x` coté client
    - Cette fonction est appelée sur le talon client qui relaie la requête coté serveur sur lequel on appellera `fonction_x_svc`
  - Avant de pouvoir appeler une fonction sur une machine distance
    - Doit identifier le programme RPC tournant sur cette machine
      - `CLIENT *clnt_create(`

```
char *machine,          nom de la machine serveur
long numero_programme,  id. du programme RPC
long numero_version,    id. de la version du programme
char *protocole);       « udp » ou « tcp »
```

      - Pour identificateurs programme et version : peut utiliser les noms associés se trouvant dans le .h
      - Retourne un identificateur de communication coté client à utiliser pour appel des fonctions ou NULL si problème
- Exemple : fichier `client.c`
  - Programme qui appelle les fonctions distantes géométriques

Exemple coté client : client.c

```
#include "geometrie.h"

int main() {
  CLIENT *client;
  rectangle *rect; point p;
  coordonnees coord; param_inclus p_inc;
  int *surface; booleen *res_inclus;

  client = clnt_create("scinfe222", GEOM_PROG,
                       GEOM_VERSION_1, "udp");
  if (client == NULL) {
      perror(" erreur creation client\n");
      exit(1);
  }
  coord.x1=12; coord.x2=20; coord.y1=10;coord.y2=15;
  p.x=14; p.y=13;
  rect = creer_rectangle_1(&coord, client);
  p_inc.rect = rect; p_inc.p = p;
  surface = calculer_surface_1(rect, client);
  res_inclus = inclus_1(&p_inc, client);

  printf(" rectangle de surface %d\n", *surface);
  if (*res_inclus) { printf(" p inclus dans rect\n");
}
```

Coté client : client.c

- Commentaires
  - Pour la connexion avec le programme RPC distant
    - Le programme s'exécute sur la machine scinfe222
    - On utilise les constantes définies dans `geometrie.h` pour identifier le programme et sa version
      - `GEOM_PROG` et `GEOM_VERSION_1`
    - On choisit une communication en UDP
  - Fonctions `creer_rectangle_1`, `calculer_surface_1` et `calculer_surface_1`
    - Vont engendrer l'appel des fonctions associés sur le serveur (sur la machine scinfe222)
    - Aucune différence avec appel d'une fonction locale
      - Transparence totale de la localisation distante du code de la fonction
- Gestion des erreurs possibles lors des appels RPC
  - Les fonctions retournent NULL
  - `clnt_perror(CLIENT*,char *msg)` : affiche l'erreur (avec `msg` avant)

Compilation parties client et serveur

- Pour les 2 parties, besoin des fonctions de codage XDR
  - `$ gcc -c geometrie_xdr.c`
- Coté client
  - `$ gcc -c geometrie_clnt.c`
  - `$ gcc -c client.c`
  - `$ gcc -o client client.o geometrie_clnt.o geometrie_xdr.o`
- Coté serveur
  - `$ gcc -c geometrie_svc.c`
  - `$ gcc -c geometrie_server.c`
  - `$ gcc -o serveur geometrie_svc.o geometrie_server.o geometrie_xdr.o`

Coté serveur

- Lancement du serveur
  - Le talon coté serveur (`geometrie_svc.c`) contient un `main` qui enregistre automatiquement le programme comme service RPC
  - Avec l'outil système `rpcinfo`, on peut connaître la liste des services RPC accessibles sur une machine
    - `$ /usr/sbin/rpcinfo -n scinfe222`

```
   program no_version protocole  no_port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100024    1   udp  32768  status
    100024    1   tcp  32770  status
    391002    2   tcp  32771  sgi_fam
 536870913    1   udp  32916
 536870913    1   tcp  38950
```

  - Notre programme est bien lancé en version 1
    - Numéro programme : $(536870913)_{10} = (20000001)_{16}$
    - Il est accesible via UDP (port 32916) ou TCP (port 38950)
  - Portmapper : démon système qui est interrogé pour récupérer la référence sur un service RPC et qui enregistre les services

RPC Language

- RPC Language
  - Langage des fichiers .x
  - Langage de type IDL : Interface Definition Language
    - Interface : ensemble des opérations qu'offre un élément logiciel
- Compléments sur la définition des éléments dans les structures de données
  - On peut utiliser des pointeurs, des énumérations et définir des types avec typedef
    - Utilisation et syntaxe comme en C
  - Pour définir des constantes
    - `const NAME = val;`
    - Exemple
      - Dans .x : `const DOUZAINE = 12;`
      - Traduit dans .h en : `#define DOUZAINE 12`

RPC Language

- Tableau
  - Tableaux de taille fixe et connue : comme en C, avec [ ]
    - Exemple : `int data[12];`
  - Tableaux de taille variable
    - Utilise les < > au lieu des [ ]
      - Peut préciser la taille maximale du tableau entre les < >
    - Exemple
      - `int data< >` taille quelconque
        `double valeurs<10>` tableau au plus de 10 double
    - Traduction en C
      - Définition d'un tableau de taille variable : structure de 2 éléments
        - Le pointeur correspond au tableau
        - La taille du tableau
      -
```
struct {
   u_int data_len;
   int *data_val;
} data;
```
```
struct {
   u_int valeurs_len;
   double *valeurs_val;
} valeurs;
```

RPC Language

- Chaînes de caractères
  - Utilise le type `string`
  - Taille de la chaîne : quelconque ou taille maximale
  - Utilise aussi la notation en < >
  - Exemple
    - `string nom<20>;` chaine de taille d'au plus 20 caractères
      `string message<>;` chaine de taille quelconque
  - Traduit dans le .h en `char*` tous les deux
    - `char *nom;`
      `char *message;`
- Données opaques
  - Utilise le type opaque
  - Exemple
    - `opaque id_client[128];`

RPC Language

- Unions
  - Syntaxe d'utilisation
    - ```
      union nom_union switch(type discrim) {
        case value : ... ;
        case value : ... ;
        ...
        default : ...;
      };
      ```
  - Exemple
    - Selon le code d'erreur d'un calcul, on veut stocker le résultat normal (un flottant) ou le code de l'erreur (un entier)
    - ```
      union res_calcul switch(int errno) {
      case 0:
          float res;
      default:
          int error;
      };
      ```

RPC Language

- Exemple union (suite)
  - Traduit dans le .h en :

```
struct res_calcul {
  int errno;
  union {
    float res;
    int error;
  } res_calcul_u;
};
typedef struct res_calcul res_calcul;
```

- Notes sur les numéros de programmes RPC
  - 4 plages de valeurs, en hexa
    - 0000 0000 à 1FFF FFFF : gérés par Sun
    - 2000 0000 à 3FFF FFFF : programmes utilisateurs
    - 4000 0000 à 5FFF FFFF : transient
    - 6000 0000 à FFFF FFFF : réservé, non utilisé

RPC : détails de fonctionnement

Relations client / serveur

- Démon « portmap » (ou « portmapper ») coté serveur
  - Sert à enregistrer les programmes/services RPC tournant sur la machine serveur
  - Reçoit les demandes d'identification de programmes de la part de clients
  - Utilise le port 111
- Fonctionnement général
  1. Le programme RPC s'enregistre localement auprès de son pormap, en précisant
     - Son numéro de programme, sa version et ses fonctions
  2. Le client voulant appeler une fonction de ce programme interroge le portmap pour récupérer le numéro de port UDP ou TCP
  3. Le client communique avec ce port via des sockets pour demander l'appel de la fonction
  4. La fonction est appelée sur le programme
  5. Le client reçoit en retour le résultat de la fonction

Temporisations

- Deux niveaux de temporisation coté client pour la gestion des appels des fonctions distantes
  - Timeout
    - Temps maximum que l'on attend après la première tentative d'appel avant de considérer le programme distant comme injoignable
    - Par défaut : 25 secondes
  - Retry
    - Temps que l'on attend avant de relancer l'appel distant de la fonction si on a pas reçu de réponse
    - Par défaut : 5 secondes
- Notes
  - Délais concernant l'appels des fonctions
    - Donc une fois que la connexion avec le portmap a eu lieu et a réussi
  - Temporisation « retry » uniquement en UDP car en TCP pas de possiblité de perte demande d'appel une fois connexion établie
  - Coté serveur, une même fonction peut donc être appelée plusieurs fois pour un seul appel coté client

Couches RPC

- Peut programmer les RPC à 2 niveaux
  - Couche haute
    - Plus simple : 3 fonctions principales
    - Mais moins de possibilité de gestion des communications
      - Uniquement en UDP, pas de gestion des temporisations ...
  - Couche basse
    - Plus complexe mais permet une gestion plus précise des communications
- Code généré par RPCGEN
  - Utilise la couche basse

Couche haute : primitives

- Coté serveur
  - `registerrpc` : enregistrement sur le portmap d'une fonction RPC du programme
    - Chaque fonction doit être enregistrée une par une
  - `pmap_unset` : désenregistrement du programme complet
  - `svc_run` : se mettre en attente d'appels de fonction
- Coté client
  - `callrpc` : demande d'appel d'une fonction d'un programme distant
- Caractéristiques couche haute
  - Fonctionne uniquement en UDP
  - Pas de paramétrage des temporisations
    - 25s de timeout et 5s de retry systématiquement

Couche haute : coté serveur

- Enregistrement d'une fonction auprès du portmap
  - `int registerrpc(`

`u_long num_prog,`	numéro du programme
`u_long version,`	numéro de version
`u_long num_fct,`	numéro de la fonction
`void *(*fonction)(),`	nom de la fonction
`xdrproc_t xdr_param,`	fonction xdr codage paramètre
`xdrproc_t xdr_result)`	fonction xdr codage résultat

  - Retourne
    - 0 si tout s'est bien passé
    - -1 en cas de problème avec affichage message d'erreur sur sortie d'erreur standard
  - Fonctions XDR
    - Les paramètres et le résultat seront systématiquement encodés/décodés : doit préciser avec quelles fonctions le faire

Couche haute : coté serveur

- Attente d'appels de fonctions de la part de clients
  - `svc_run()`
  - Ne retourne jamais sauf erreur
- Désenregistrement d'un programme complet
  - `void pmap_unset(`
    `u_long num_prog,` numéro du programme
    `u_long version)` numéro de version
  - Si l'exécutable implémentant les fonctions et s'étant enregistré auprès du portmap est planté ou stoppé
    - Le portmap continue de référencer le programme RPC
    - Nécessité donc d'explicitement le désenregistrer

Couche haute : coté client

- Appel d'une fonction d'un programme RPC distant
  - `int callrpc (char *host,` — nom de la machine distante
    `u_long num_prog,` — numéro du programme
    `u_long version,` — numéro de la version
    `u_long num_fct,` — numéro de la fonction
    `xdrproc_t xdr_param,` — fonction xdr codage paramètre
    `void *param,` — paramètre de la fonction
    `xdrproc_t xdr_result,` — fonction xdr codage résultat
    `void *result)` — contiendra le résultat
    - Retourne
      - 0 en cas de succès de l'appel de la fonction distante
      - Une valeur de type `enum clnt_stat` en cas d'erreur
        - Voir fichier <rpc/clnt.h> pour détail des erreurs
        - Erreur affichable avec `clnt_perrno`

Couche haute : exemple

- Même exemple de gestion de rectangle
  - Définitions communes : rectangle.h
    -
```
#define RECT_PROG 0x20000001
#define RECT_V_1 1
#define SURFACE_RECTANGLE 1
#define CREER_RECTANGLE 2
#define INCLUS 3

struct rectangle {
    struct point p1;
    struct point p2;
};
typedef struct rectangle rectangle;

extern bool_t xdr_rectangle (XDR *, rectangle*);
...
```

Couche haute : exemple

- Coté serveur

```
#include "rectangle.h"

int *surface_rectangle(rectangle *rect){
     static int result;
     result = abs((rect -> p1.x - rect -> p2.x) *
                  (rect -> p1.y - rect -> p2.y));
     return &result;
}

int main() {
     // désenregistrement éventuel du programme
     pmap_unset(RECT_PROG, RECT_V_1);
     // enregistrement de la fonction
     if (registerrpc(RECT_PROG, RECT_V_1, SURFACE_RECTANGLE,
           surface_rectangle, xdr_rectangle, xdr_int) == -1){
          printf("erreur surface_rectangle\n");
          exit(1);
     }
     // attente d'appels distants de clients
     svc_run();
     printf("erreur attente appels !\n");
}
```

Couche haute : exemple

- Coté client
  - ```
    #include "rectangle.h"

    int main() {
         rectangle rect;
         point p1, p2;
         int result, surface;

         p1.x = 12; p1.y = 10;
         p2.x = 20; p2.y = 30;
         rect.p1 = p1; rect.p2 = p2;

         // appel de la fonction sur la machine scinfe122
         result = callrpc("scinfe122", RECT_PROG, RECT_V_1,
             SURFACE_RECTANGLE, xdr_rectangle, &rect,
             xdr_int, &surface);

         if (result !=0) {
              clnt_perrno((enum clnt_stat)result);
              exit(1);
         }
         printf("surface = %d\", surface);
    }
    ```
    ```

Couche basse : primitives

- Coté serveur
  - Type `SVCXPRT` : caractéristiques serveur et client appelant une fonction
    - `SVCXPRT *svcudp_create() & *svctcp_create()`
      - Création objet `SCVXPRT` avec gestion de la socket utilisée
    - void `svc_destroy()` : destruction d'un objet `SCVXPRT`
  - Type `svc_req` : identification d'une requête d'appels de fonction de la part d'un client
  - `svc_register` & `svc_unregister` : enregistrement et désenregistrement d'un programme auprès du portmap
  - Ensemble de macros pour décoder les requêtes d'appels : `svc_getargs`, `svc_freeargs`, `svcerr_decode` ...
  - Renvoi d'un résultat au client : `svc_sendreply()`
  - Attente requête de clients : `svc_run()`

Couche basse : primitives

- Coté client
  - Type `CLIENT` : caractéristiques de la liaison client / serveur
  - Création d'un objet client : `clnt_create`
    - Variantes pour plus de précisions sur la communication entre client et serveur : `clntudp_create` & `clnttcp_create`
  - `clnt_destroy` : destruction objet client
  - `clnt_control` : modification des caractéristiques du client
  - `clnt_call` : appel d'une fonction sur le programme RPC distant

Couche basse : coté serveur

- Type SVCXPRT : caractéristiques programme RPC
  - Identification de la socket et du port utilisé par le programme, champs gérant l'appels des fonctions ...
- Création objet SVCXPRT
  - `SVCXPRT *svcudp_create(int sock)`
    - Création pour utilisation d'UDP
    - `sock` : identificateur de la socket à utiliser
    - Pour créer et utiliser une socket quelconque : `RPC_ANYSOCK`
  - `SVCXPRT *svctcp_create(int sock, int taille_send, taille_receive)`
    - Idem mais pour TCP
    - `taille_send` et `taille_receive` : tailles des buffers d'émission et de réception pour gérer les enregistrements XDR
      - Valeur 0 : tailles par défaut (4000 octets)

Couche basse : coté serveur

- Destruction objet SVCXPRT
  - `void svc_destroy(SVCXPRT *ptr)`
- Enregistrement d'un programme RPC au niveau du portmap
  - `svc_register`
  - Principe couche haute
    - On enregistre une à une chaque fonction du programme
  - Principe couche basse
    - On enregistre le programme et on précise une fonction qui sera appelée à chaque appel distant d'une fonction de ce programme
      - Fonction « dispatch » : c'est elle qui décode le numéro de la fonction appelée et appelle la fonction associée

Couche basse : coté serveur

- Enregistrement d'un programme
  - `bool_t svc_register(`
    `SVCXPRT *svc,` — le programme à enregistrer
    `u_long num_prog,` — numéro du programme
    `u_long version,` — version du programme
    `void (*dispatch)(),` — fonction de dispatch
    `u_long proto);` — pour UDP ou TCP
  - `proto` : précise pour quelle couche on enregistre le programme
    - Constantes `IPPROTO_TCP` ou `IPPROTO_UDP`
  - Retourne vrai si l'enregistrement s'est bien passé, faux sinon
    - Signature de la fonction de dispatch
      - `void dispatch(struct svc_req *req, SVCXPRT *svc)`
      - `req` : identification de la requête d'appel de fonction
      - `svc` : identification du programme et caractéristiques du client
- Désenregistrement d'un programme
  - `void svc_unregister(u_long num_prog, u_long version)`

Couche basse : coté serveur

- Fonction de dispatch : fonctionnement
  - Récupérer le numéro de la fonction à appeler dans la structure `svc_req`
  - Décoder le paramètre via le contenu de la structure `SVCXPRT`
  - Appeler la fonction avec son paramètre
  - Renvoyer le résultat au client via la fonction `svc_sendreply`
    -
```
bool_t svc_sendreply(
        SVCXPRT *svc,
        xdrptoc_t xdr_result,
        void *result)
```
      - `svc` : structure passée à l'appel de la fonction de dispatch, pour identifier le client et sa requête
      - `xdr_result` : fonction XDR de codage du type du résultat
      - `result` : le résultat à envoyer au client
      - Retourne vrai si l'envoi au client a réussi, faux sinon

Couche basse : serveur

- Exemple fonction dispatch, avec exemple précédent

```
void dispatch_rect(struct svc_req *req, SVCXPRT *svc){
  switch (req -> rq_proc) {
     // fonction code 0
     case NULLPROC:
       svc_sendreply (svc, (xdrproc_t)xdr_void, NULL);
       return;

     case SURFACE_RECTANGLE:
       rectangle rect;
       if (svc_getargs(svc, xdr_rectangle, &rect)==FALSE){
          svcerr_decode(svc);
          return;
       }
       if (svc_sendreply(svc, xdr_int,
                         surface_rectangle(&rect))==FALSE)
          svcerr_decode(svc);
       return;
     ...
     default: svcerr_noproc(svc);
       return;
}
```

Couche basse : serveur

- Fonction dispatch : explications
  - Tout programme RPC doit avoir une fonction numérotée 0 ne faisant rien et répondant une réponse vide au client
    - Sorte de « ping » pour tester que le programme fonctionne
    - On renvoie la valeur NULL « codée » par `xdr_void`
  - Pour chaque fonction
    - `svc_getargs` : macro pour décoder l'argument via la fonction XDR associée
    - `svcerr_decode` : si erreur de décodage, on doit renvoyer une erreur au client précisant cela
  - Cas par défaut du switch
    - On a pas trouvé la fonction voulue par le client : on l'en informe via `svcerr_noproc`

Couche basse : serveur

- Fonctions d'erreurs
  - Pour préciser au client qu'une erreur particulière a eu lieu
  - `svcerr_noproc` : numéro de fonction invalide
  - `svcerr_decode` : erreur de codage/décodage XDR
  - `svcerr_noprog` : numéro de programme invalide
  - `svcerr_progvers` : numéro de version invalide
  - `svcerr_auth` : erreur d'authentification
  - `svcerr_systemerr` : erreur système/autre
- Récupérer informations sur localisation du client
  - `struct sockaddr_in *svc_getcaller(SVCXPRT *svc)`

Couche basse : client

- Type CLIENT
  - Contient données d'authentification et de gestion des appels distants
- Création d'un objet client
  - Version générique : `clnt_create` (voir transparent 23)
  - Versions spécifiques : une pour UDP, une pour TCP
    -
```
CLIENT *clnt_udpcreate(
        struct sockaddr_in *adr,
        long numero_programme,
        long numero_version,
        struct timeval retry,
        int *sock);
```
adresse programme distant
id. du programme RPC
id. de la version du programme
temps de relance de la requête
id. de socket à utiliser pour comm. avec serveur ou RPC_ANYSOCK
  - Définition d'un intervalle de temps
    -
```
struct timeval {
        int tv_sec;          secondes
        int tv_usec;         microsecondes
};
```

Couche basse : client

- Création objet CLIENT, version spécifique TCP
  - `CLIENT *clnt_tcpcreate(`

```
        struct sockaddr_in *adr,   adresse programme distant
        long numero_programme,     id. du programme RPC
        long numero_version,       id. de la version du programme
        int *sock,                 id. de socket à utiliser pour comm.
                                   avec serveur ou RPC_ANYSOCK
        u_int taille_send,         taille buffer émission pour flots XDR
        u_int taille_receive)      taille buffer réception pour flots XDR
```

- Destruction d'un object CLIENT
  - `void clnt_destroy(CLIENT *client)`
- Modification/récupération des caractéristiques de la communication client/serveur
  - `bool_t clnt_control(`

```
        CLIENT *client,    objet CLIENT à configurer ou interroger
        int requete,       type de la requête
        void *info);       données lues ou à lire pour modification
```

Couche basse : client

- Caractéristiques communication client/serveur avec `clnt_control`
  - Pour temporisations, valeurs de `requete`
    - Timeout : CLSET_TIMEOUT & CLGET_TIMEOUT
    - Retry : CLSET_RETRY_TIMEOUT & CLGET_RETRY_TIMEOUT
      - Note : uniquement utilisable en UDP
    - Champ `info` : de type `struct timeval*`
      - En écriture pour les « GET »
      - En lecture pour les « SET »
  - Récupérer l'adresse du programme RPC distant
    - `requete` : CLGET_SERVER_ADDR
    - `info` : de type `struct sockaddr_in*`

Couche basse : client

- Appel d'une fonction sur le programme distant
  - `enum clnt_stat clnt_call(`

`CLIENT *client,`	objet client initialisé
`u_long num_fonction,`	numéro de la fonction à appeler
`xdrproc_t xdr_param,`	fonction XDR codage paramètre
`void *param,`	paramètre de la fonction
`xdrproc_t xdr_result,`	fonction XDR codage résultat
`void *result,`	contiendra le résultat
`struct timeval timeout)`	timeout global

  - Valeur retournée
    - `RPC_SUCCESS` : l'appel s'est bien déroulé
    - Autre valeur : précise le type d'erreur rencontrée
  - Note
    - L'identification du programme et de sa version est dans l'objet CLIENT

Couche basse : exemple, serveur

```
int main (int argc, char *arg[]) {
  SVCXPRT *svc;

  pmap_unset (RECT_PROG, RECT_V_1);

  svc = svcudp_create(RPC_ANYSOCK);
  if (svc == NULL) {
    printf ("erreur création svc\n");
    exit(1);
  }
  if (!svc_register(svc, RECT_PROG, RECT_V_1,
                    dispatch_rect, IPPROTO_UDP)) {
    printf ("erreur enregistrement\n");
    exit(1);
  }
  svc_run();
  printf("erreur svc_run\n");
  exit(1);
}
```

Couche basse : exemple, client

```
int main(int argc, char *argv[]) {
  CLIENT *client;
  rectangle rect;
  int surface;
  struct timeval TO = { 25, 0 };
  enum clnt_stat result;

  rect.p1.x = 12; rect.p1.y = 10;
  rect.p2.x = 20; rect.p2.y = 30;

  client = clnt_create("scinfe122", RECT_PROG,RECT_V_1, "udp");

  if (client == NULL) {
    printf(" erreur creation client\n");
    exit(1);
  }

  result = clnt_call(client, SURFACE_RECTANGLE,
              (xdrproc_t)xdr_rectangle, &rect,
              (xdrproc_t)xdr_int, &surface, TO);
  if (result !=  RPC_SUCCESS) {
     clnt_perrno(client, "erreur appel fonction "); exit(1);
  }
}
```