PROGRAMMATION RESEAU

SOCKETS

TLI

RPC

HUE Hervé

Année Spéciale Informatique 1993-1994

Un grand merci à Serge ROUVEYROL
pour tout ce qu'il m'a apporté, y compris les cafés.

Je remercie aussi COMER, TANENBAUM, STEVENS...

TABLE DES MATIERES

0. INTRODUCTION

Mais au fait, qu'est-ce qu'un réseau ?

D'une façon générale, un réseau identifie un ensemble de voies et de supports reliant entre eux plusieurs points à des fins de communication. Les réseaux sont à la base des communications inter-systèmes.

Et pourquoi un modèle en couches ?

Etant donné un problème à résoudre, tel un échange de fichiers entre deux ordinateurs connectés par un réseau, il est judicieux de le diviser et de résoudre chaque morceau indépendamment. La solution globale sera la réunion de ces diverses parties. Une telle façon de procéder conduit à une solution meilleure et plus extensible que la résolution du problème en un seul bloc. Il est aussi possible qu'une partie du problème puisse servir dans un autre contexte.
Dans le domaine des réseaux, ceci est appelé le "modèle en couches" (layering). Le problème des communications est divisé en parties (couches) et chacune fournit une fonction particulière. Notons que cette méthode demande des interfaces entre couches parfaitement définies.

Et naquit le modèle OSI

Le modèle de référence OSI (Open System Interconnection) divise les fonctions de communication réseau en sept couches, comme indiqué dans le tableau 1.

Tableau 1 : le modèle en couche OSI

NOM	No
Application	7
Présentation	6
Session	5
Transport	4
Réseau	3
Liaison	2
Physique	1

Nous donnons ici un descriptif sommaire des couches du modèle OSI :

Couche physique

C'est la couche matériel du modèle. Elle est constituée des connecteurs, du medium de transmission (cable ethernet, fibre optique...), des multiplexeurs.

Couche liaison

Cette couche effectue le transmission et la reception des données. Du coté transmission, le logiciel ethernet (par exemple), rassemble les données en paquets de taille convenable, et les encapsule (rajoute des informations telles que l'adresse physique du receveur...). elle transmet les paquets et les retransmet si nécessaire.

Du coté réception, le matériel ethernet reconnait les paquets comportant son adresse et les récupère. Le logiciel enlève l'enveloppe et rassemble les données. Il devrait auusi détecter les erreurs de transmission.

Couche réseau

Cette couche fait le routage des paquets (construit l'itinéraire à travers le réseau), transformant l'adresse logique en adresse physique.

Couche transport

Controle le flux sur le réseau.

Couche session

Organise des sessions de communication fiables entre des processus (reprise sur erreur...).

Couche présentation

Réalise la conversion entre la représentation des données sur la machine locale et le format indépendant des machines pour le transport sur le réseau.

Couche application

Dans cette couche résident les services et programmes du niveau utilisateur. Ce sont par exemple telnet, ftp (programmes) et NFS,DNS (services).

Mais, OSI n'est qu'un modèle !

Plusieurs familles de protocoles existent, basées sur ce modèle ou antérieures à celui-ci. Parmi celles-ci, on peut en remarquer quelques unes :

- SNA d'IBM (System Network Architecture)
- XNS de Xerox (Xerox Networking Systems)
- TCP/IP (les protocoles Internet)

Cette dernière famille trouve ses origines dans les années 60, descendante d'un projet du Département de Défense des Etats-Unis (DoD). Plusieurs raisons ont conduit à son succès, la rendant maintenant extrêmement répandue :

- son indépendance vis à vis des constructeurs de matériels (contrairement à SNA ou XNS)
- son implantation sur un large éventail de matériels.

TCP/IP

La famille de protocoles TCP/IP, antérieure au modèle OSI n'entre pas en correspondance exacte avec lui, même si l'analogie est évidente. Par rapport au modèle décrit ci-dessus, TCP/IP est formé de quatre couches comme le montre le tableau 2.

Tableau 2 : les couches de TCP/IP

NOM	No de couche OSI
Application	5 à 7
Transport	4
Réseau	3
Interface matérielle	1 et 2

Nous nous intéressons aux deux couches intermédiaires.

couche réseau

Dans TCP/IP, c'est le Protocole Internet (IP) qui réalise cette couche. Il gère l'acheminement des données pour les couches supérieures. IP est dit sans connexion et non fiable : les paquets sont considérés comme indépendants, IP ne peut donc faire aucune connexion entre eux. IP ne garantit pas non plus la livraison des paquets, ni l'ordre dans lequel ceux-ci arrivent, il est non fiable.

couche transport

TCP/IP contient deux principaux protocoles pour la couche transport :

- TCP : Transmission Control Protocol
- UDP : User Datagram Protocol

Notre but n'étant pas de détailler ces différents protocoles, nous ne donnerons que les informations nécessaires pour comprendre leur fonctionnement. La bibliographie contient des références qui seront utiles à ceux qui désirent en savoir plus.

En juillet 1992, 700000 ordinateurs étaient connectés entre eux, à travers le monde, grâce à Internet. Ce chiffre doublant tous les 10 mois, nous en serions donc à près de 3000000 de ces machines reliées entre elles par ce réseau (à vérifier).

Parallèlement à cette croissance, un changement important a eu lieu ces dernières années dans l'utilisation d'Internet (et donc de TCP/IP): avant cette période, les utilisateurs se servaient surtout des services de base (transfert de fichiers, courrier électronique,...). Maintenant, les applications personnelles se développent offrant de nouveaux services au dessus de TCP/IP, services basés sur l'architecture client-serveur (WWW, Archie, Wais...). Les applications réseau privées ne se comptent plus.

Du point de vue d'une application, TCP/IP ne fournit que des outils pour le transfert des données. Par exemple, TCP/IP permet à un programmeur d'établir une communication entre deux applications et de transférer des données entre elles.
Bien que TCP/IP spécifie les détails du transport des données, il n'impose rien sur la manière qu'ont les applications d'interagir, ni sur la manière dont un programmeur doit organiser son application distribuée.
En pratique, une méthode d'organisation domine, appelée architecture client-serveur.

L'architecture client-serveur

Une application qui a l'initiative d'une communication est appelée un CLIENT. Les utilisateurs lancent généralement un programme client lorsqu'ils utilisent un service du réseau. La plupart des logiciels clients sont des programmes «classiques»: chaque fois qu'une application client s'exécute, elle contacte un serveur, envoie une requête et attend la réponse. Quand celle-ci arrive, le client continue son déroulement. Souvent, les clients sont plus faciles à écrire que les serveurs et ne nécessitent pas de privilèges particuliers pour fonctionner.

Un SERVEUR est un programme qui attend une requête d'un client. Lorsqu'il en reçoit une (d'un client quelconque), il fait le nécessaire (ce que le client lui a demandé et ce pour quoi il est fait) et retourne le résultat. Les serveurs devant souvent accéder à des ressources ou des données (comme un serveur ftp doit avoir accès à des fichiers), ils nécessitent souvent des privilèges.

Les programmes utilisant des ressources réseau ne rentrent pas toujours exactement dans une de ces deux catégories. Un serveur peut avoir besoin d'un service sur le réseau et donc agir en client.

Les serveurs orientés connexion et sans connexion

Quand un programmeur conçoit un logiciel client-serveur, il doit choisir entre deux modes d'interaction: orienté connexion ou sans connexion. Ces deux styles correspondent aux deux principaux protocoles de la couche transport (couche 4) de TCP/IP:

-TCP orienté connexion
-UDP sans connexion

La distinction entre ces deux modes est très importante car elle détermine le niveau de fiabilité fournit par le système.

TCP fournit toute la fiabilité nécessaire pour communiquer à travers le s réseaux. Il vérifie automatiquement les données qui arrivent et retransmet les segments erronés. Il vérifie que les segments arrivent dans le bon ordre et élimine les duplications éventuelles. Il fournit également un contrôle de flux pour éviter que l'émetteur ne transmette les données plus vite que le destinataire ne peut les digérer.

Par contre, les clients et serveurs qui utilisent UDP ne fournissent aucune garantie sur la fiabilité de la transmission. L'application doit donc prendre les mesures appropriées pour détecter et corriger les erreurs.

La fiabilité d'UDP dépend beaucoup des couches inférieures (comme IP). Dans un environnement local, les erreurs sont rares, mais dans les communications longue distance...Dans les tests d'un client-serveur, qui sont souvent réalisés en local, il est nécessaire de tenir compte du fait que cette application sera peut-être utilisée lors de communications longue distance.

Pour réaliser la partie communication d'un client-serveur plusieurs solutions sont possibles. Nous allons en exposer trois : les sockets, les TLI et les RPC, cette dernière solution allant être particulièrement développée par la suite.

Les sockets

Les sockets sont apparues sous UNIX en 1981 dans BSD 4.2, elles forment un composant de base de la communication entre processus (processus situés sur une même machine ou sur deux machines distinctes).
Les sockets fournissent un accès aux protocoles de transport (couche 4 du modèle OSI).Une socket est une extrémité de connexion (socket : prise électrique femelle) à laquelle on peut associer un nom. Elle peut être de différents types et être associée à plusieurs processus.
La description et l'utilisation des sockets seront détaillées au chapitre 1.

Les TLI

L'Interface de la couche Transport (TLI : Transport Layer Interface) est un ensemble de fonctions permettant aux applications réseau l'utilisant d'obtenir une certaine indépendance vis à vis du protocole de transport utilisé.
La description et l'utilisation des TLI seront détaillées au chapitre 2.

Les RPC

Les RPC (Remote Procedure Calls) forment un ensemble d'outils de programmation permettant une utilisation plus simple et plus puissante du modèle client-serveur que les sockets et TLI. Ils furent développés par SUN pour NFS.
Ils permettent de distribuer facilement certaines parties d'une application sur des machines différentes tout en gardant le mécanisme d'appel de procédure. Les RPC seront détaillés aux chapitre 3 et suivants et forment la partie principale de ce rapport.

Mais, que choisir ?

A travers ces trois possibilités d'implémentation du modèle client-serveur, nous verrons les avantages et inconvénients de chacune. Des exemples permettront de se faire une idée de l'utilisation de ces solutions, leur étude permettra à chacun de se former un début d'opinion.

Les exemples de ce rapport ont tous été testés et sont disponibles par ftp sur imag.fr dans le répertoire `/pub/DOC.UNIX/RPC` sous le nom `rpc_exemples.tar.gz`. Une structure arborescente est construite et contient des répertoires se rapportant à chaque exemple. De plus, chaque exemple est accompagné d'un fichier LISEZ_MOI qui présente un moyen de compiler et d'utiliser les exemples. La version PostScript de ce rapport se trouve au même endroit dans `rapport_rpc.ps.gz`.

1.LES SOCKETS

L'analogie avec les fichiers

Une socket est un point de communication par lequel un processus peut émettre ou recevoir des informations. A l'intérieur d'un processus, une socket sera identifiée par un descripteur de même nature que ceux identifiant les fichiers. Ceci permet, par exemple, la redirection des entrées-sorties standards vers des sockets. De même, tout processus hérite des descripteurs de son père.

Une application qui veut gérer une communication réseau peut appeler la fonction `socket()` pour créer un descripteur de socket. L'extrémité de connexion est alors créée, il reste à spécifier ces caractéristiques.

Le descripteur de socket rassemble en fait un ensemble de pointeurs vers des structures internes comme le montre la figure 1.1.

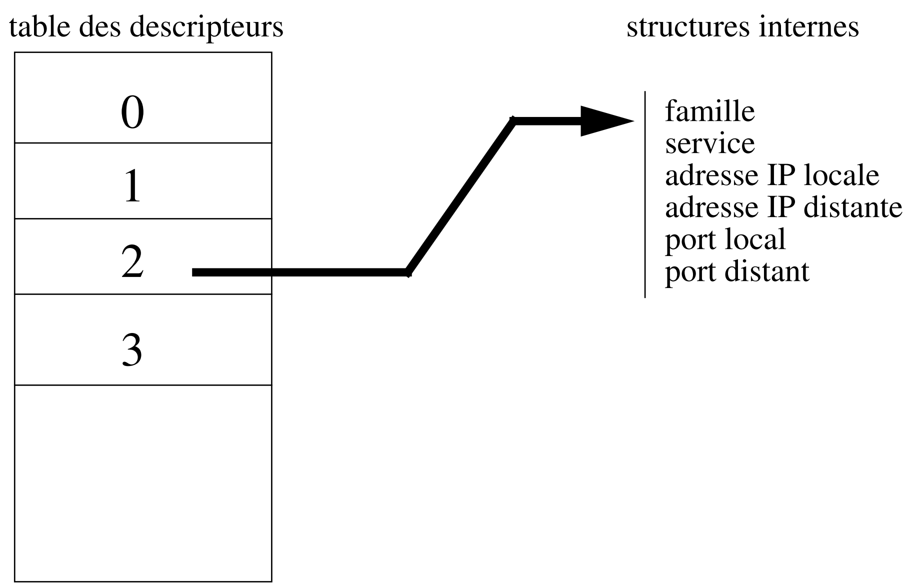

figure 1.1 : les descripteurs pointent sur des structures internes

Certains champs de ces structures ne sont pas initialisés par l'appel à `socket()`, il faut donc les remplir. Une fois la socket créée, elle peut se comporter de deux manières :

- attendre une connexion extérieure, socket passive
- générer une connexion (appel à un serveur), socket active.

L'adresse d'une socket

Bien que d'utilisations différentes, ces deux sortes de sockets sont créées de manière unique. Après la création, il est donc nécessaire de spécifier les caractéristiques de l'utilisation envisagée. En particulier, il faut associer aux sockets une adresse sur la machine hôte. Cette extrémité de connexion (l'adresse) est composée de deux champs :

- une adresse IP
- un numéro de port.

Si cette socket doit être active, il faudra bien sùr préciser l'adresse de la machine distante.

Ceci est valable pour TCP/IP, d'autres familles de protocoles pouvant définir une adresse différemment. On doit donc définir la famille d'adresses pour connaitre la représentation utilisée pour une adresse. Pour TCP/IP, ce sera AF_INET.

(rem : on essayera de faire la distinction entre famille de protocoles et familles d'adresses).

La structure d'une adresse de socket sous TCP/IP est la suivante :

```
struct sockaddr_in {
    u_short sin_family;/* famille d'adresses */
    u_short sin_port; /* numéro de port */
    u_long sin_addr; /* adresse IP */
    char sin_zero[8]; /* inutilisé */
};
```

Bien d'autres familles d'adresses existent, parmi lesquelles AF_SNA, AF_APPLETALK ...

Les principaux appels système utilisés pour les sockets

Nous allons effectuer un rapide survol des prinipales fonctions pour pouvoir introduire des exemples (bien plus intéressants qu'un long discours).

- `socket()`

Comme vu précédemment, permet la création d'une socket et renvoie un descripteur qui lui est associé. Les arguments comportent la famille de protocole (PF_INET pour TCP/IP) et le type de service demandé (TCP ou UDP).

- `connect()`

Permet à un client d'envoyer une requête de connexion à un serveur. Les arguments comprennent l'adresse du serveur (adresse IP et numéro de port du socket où contacter le serveur).

- `write()`

Permet l'envoi de données à travers le réseau. Elle est identique à la fonction liée aux entrées-sorties «classiques» et comporte les mêmes arguments (descripteur de socket, adresse du tampon où sont stockées les données et taille de ce tampon).

- `read()`

Permet la récéption de données. Elle se comporte de la même façon que la fonction liée aux entrées-sorties «classiques» et prend les mêmes arguments (descripteur de socket, adresse du tampon où stocker les caractères reçus, et taille de ce tampon). Si le buffer d'entrée est vide, la fonction est bloquante. Elle renvoie le nombre de caractères lus (toujours inférieur ou égal à la taille donnée pour le buffer).

Remarque : l'analogie avec les écritures et lectures sur fichiers est renforcée par le fait que les descripteurs de sockets sont stockés avec les descripteurs de fichiers, ce qui implique de plus qu'une socket et un fichier ne peuvent pas avoir leurs descripteurs identiques pour un même processus.

- `close()`

On s'y attendait un peu, cette fonction permet de fermer un descripteur de socket. La communication sera fermée réellement s'il n'y a plus de processus utilisant cette connexion (compteur de références).

- `bind()`

Permet de relier un descripteur de socket à une adresse. Une socket qui n'était accessible que par son descripteur le deviendra par son adresse.

- `listen()`

Permet à un serveur de rendre une socket passive (en attente de la requête de connexion d'un client). Pour ne pas perdre trop de ces requêtes, on peut associer une queue aux requêtes en attente.

- `accept()`

Permet au serveur d'extraire une requête de connexion de la file d'attente, et crée une nouvelle socket pour chaque connexion.

- `recv()` et `recvfrom()`

Permettent la réception de données en récupérant l'adresse de l'envoyeur.

- `snd()` et `sendto()`

Permettent l'envoi de données en précisant l'adresse du destinataire ou pas si la socket a été connectée.

Le problème du format des entiers

Suivant les machines, la représentation interne des entiers peut différer (octets le plus significatif en premier ou en dernier). Pour permettre un dialogue entre des machines diverses

sans ce heurter à ce problème, TCP/IP spécifie une représentation standard des entiers sur le réseau sous le nom de «network byte order». Pour l'anecdote, l'octet le plus significatif est placé en premier.

On dispose donc de routines permettant la conversion entre le format interne de l'hôte et le format réseau. Il est nécessaire à la portabilité d'une application de toujours utiliser ces routines de conversions même si votre machine a la même représentation des entiers que TCP/IP.

Un exemple de ces routines est `htons()` qui convertis un short int (s) du format hôte (h) au (to) format réseau (n comme network).

Une communication orientée connexion

On peut modéliser cette communication comme représenté figure 1.2.

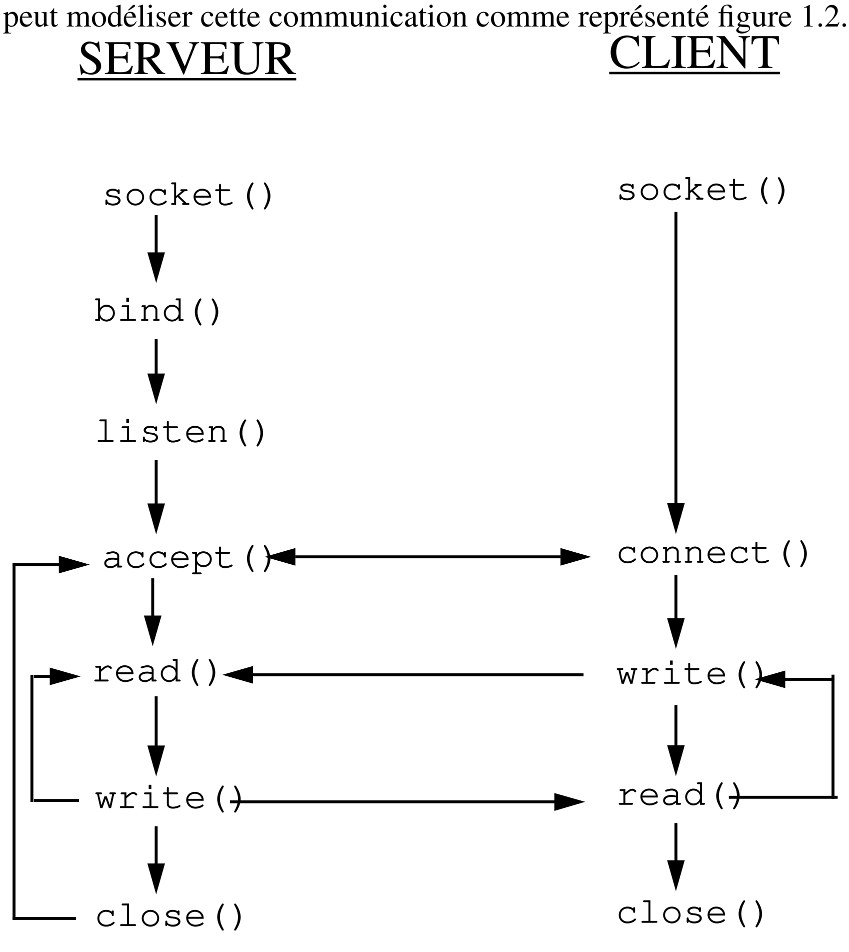

figure 1.2 : une communication orientée connexion

Pour le serveur, `accept()` crée une nouvelle socket qui est reliée au client qui a demandé la connexion. C'est cette socket que l'on ferme lorsque la connexion avec le client est terminée.

`shutdown()` permet de fermer une socket de manière plus douce que `close()` en fermant une socket en ecriture ou en lecture (ou les deux). Un processus peut donc continuer de recevoir s'il a fermer une socket en ecriture.

Une communication sans connexion

De la même façon que précédemment, on modélise cette communication comme représenté figure 1.3.

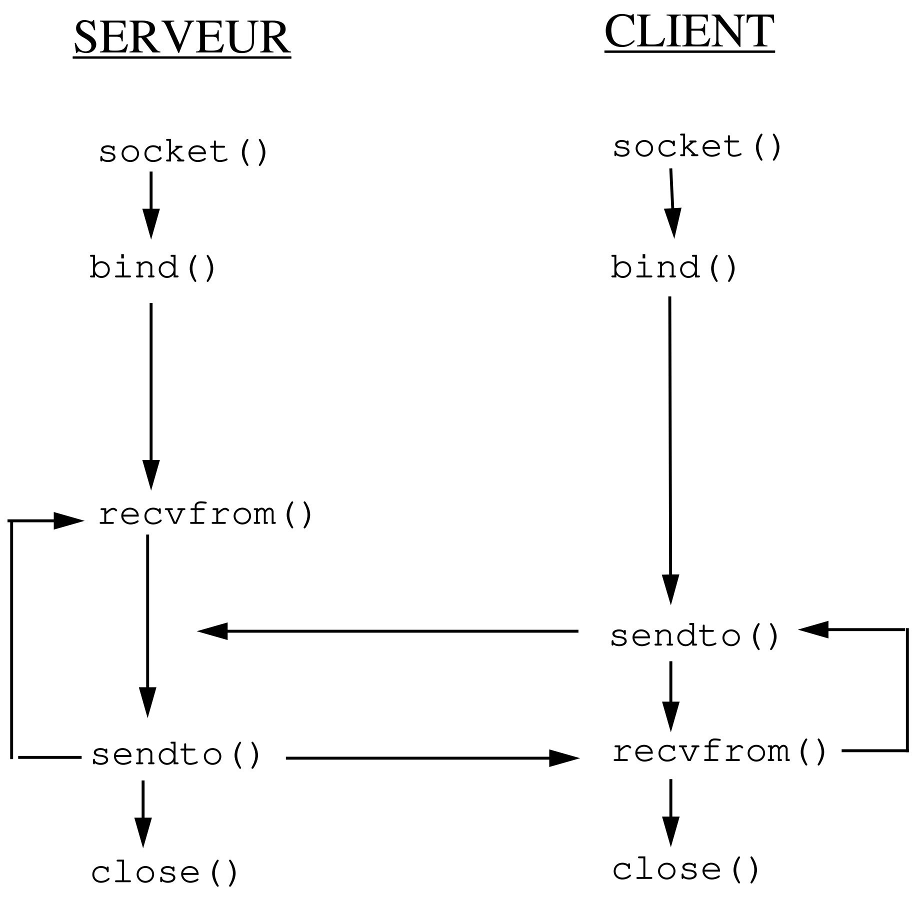

figure 1.3: une communication orientée connexion

Les sockets datagramme peuvent aussi utiliser `connect()` pour associer une socket à une destination précise. Dans ce cas, on peut utiliser `send()` à la place de `sendto()`.

De même, si l'adresse de l'envoyeur d'un datagramme n'intéresse pas un processus, il peut utiliser `recv()` à la place de `recvfrom()`.

Les routines de manipulation d'adresses

Il existe de nombreuses routines permettant de manipuler les noms de machines, de réseaux, de services. Nous ne les citons qu'à titre indicatif, même si elles sont souvent nécessaires pour les connexions.

- noms de machines : une structure `hostent` relie le nom d'une machine à son adresse. Cette structure est manipulée grâce aux routines `gethostbyname()`, `gethostbyaddr()`.
- noms de réseaux : une structure `netent` relie le nom d'un réseau à son numéro. On peut manipuler cette structure grâce aux routines `getnetbyname()`, `getnetbyaddr()` et `getnetent()`.
- noms de protocoles : la structure se nomme `protoent` et les routines de manipulation `getprotobyname()`, `getprotobynumber()` et `getprotoent()`.
- noms de services : la structure se nomme `servent` et la routine utile se nomme `getservbyname()`.

Programmation avancée des sockets

Dans certaines applications certaines capacités supplémentaires des sockets pourront être utiles. Encore une fois, notre but n'est pas d'être exhaustif, mais de donner au lecteur une idée des possibilités des sockets. Nous ne ferons donc que mentionner ces possibilités.

Dans l'interface socket, il existe la notion de données urgentes (out-of-band data), qui permet d'envoyer un message indépendamment des autres. Cette facilité est, par exemple employée par rlogin et rsh pour propager des signaux entre les processus clients et serveurs.

Certaines applications nécessitent des sockets non-bloquantes. Les requêtes qui ne peuvent être honorées immédiatement, ne sont pas executées, et un code d'erreur est retourné.

Dans les applications temps-réel, une communication asynchrone est souvent nécessaire. Les sockets peuvent donc être rendues asynchrones.

On peut aussi gérer les sockets sur interruptions, le signal `SIGIO` signifiant une fin de transfert de données sur une socket.

Il existe, de plus, des fonctions telles que `getsockopt()` et `setsockopt()` qui permettent d'obtenir des informations sur une socket particulière ou d'en modifier le comportement

Des exemples

Les pages suivantes montrent un exemple de programme client-serveur utilisant les sockets. Deux versions de ce programme ont été écrites : la première en mode orienté connexion et la seconde en mode sans connexion.

Ce programme permet de lister les répertoires se trouvant sur la machine hôte du serveur depuis une machine distante hôte du client.

La commande s'écrit :

`rls <hôte du serveur> <No port> <chemin accès répertoire>`

exemple : `rls ensisun.imag.fr /users2`

Ce programme se retrouvera dans les chapitres suivants écrit d'une autre manière. Une constante est le fichier `read_dir.c` qui est commun à toutes les versions de `rls`.

Le fichier read_dir.c

```
/* read_dir.c : liste un directory
       Ce fichier est commun a tous les exemples
       concernant le listing de directory a distance
*/

#include <sys/types.h>
#include <dirent.h>
#include <stdio.h>

/* En entree, dir contient le chemin d'accès du repertoire
   En sortie, dir contient le listing du contenu du repertoire
*/

read_dir(dir)
char *dir;
{
       DIR *dirp;
       struct dirent *d;

       /* ouvrir le directory */

       dirp=opendir(dir);
       if(dirp==NULL) return NULL;

       /* recopie des fichiers dans le buffer dir */

       dir[0]=NULL;
       while(d=readdir(dirp)) sprintf(dir,"%s%s\n",dir,d->d_name);

       /* retourner le résultat */

       closedir(dirp);
       return((int)dir);

}
```

Le client version TCP

```
/* Partie client du programme de listing de répertoires
       sur machine distante.
       Version socket-tcp
```

```
*/


#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>


#define BUFSIZE 8192



main(argc,argv)
int argc;
char *argv[];
{
int sock;
struct sockaddr_in serveur,client;
struct hostent *hp,*gethostbyname();
char buf[BUFSIZE];
int rval;

/* création d'une socket en mode stream */

sock=socket(AF_INET,SOCK_STREAM,0);
if(sock==-1) {
       perror("ouverture de la socket");
       exit(1);
}

/* connecter la socket grâce au nom de l'hôte passé en argument */

serveur.sin_family=AF_INET;
hp=gethostbyname(argv[1]);

/* gethostbyname retourne une structure qui contient l'adresse réseau de
l'hôte specifié */


if(hp==(struct hostent *)0) {
       fprintf(stderr,"%s: hôte inconnu\n",argv[1]);
          exit(2);
}

/* sous UNIX BSD, memcpy() s'appelle bcopy() */

memcpy((char *) &serveur.sin_addr,(char *)hp->h_addr,hp->h_length);
serveur.sin_port=htons(atoi(argv[2]));

/* connexion au serveur */

if(connect(sock,(struct sockaddr *) &serveur,sizeof(serveur))==-1) {
       perror("connection socket");
       exit(1);
}
```

```
/* écriture du chemin d'accès du répertoire sur la socket */

if(write(sock,argv[3],strlen(argv[3]))==-1)
       perror("ecriture sur socket");

/* lecture du résultat sur la socket */

do {
       if((rval=read(sock,buf,BUFSIZE))==-1)
            perror("lecture");
       if(rval==0)
            printf("\nFin de connection\n");
       else {
            buf[rval]='\0';
            printf("%s",buf);
       }
while(rval>0);

/* fermeture de la socket */

close(sock);

exit(0);

}
```

Le serveur version TCP

```
/* Partie serveur du programme de listing de répertoires
sur machine distante.Ce serveur est dit itératif, il ne traite
qu'une requête à la fois.
Version socket-tcp
*/

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define TRUE 1
#define BUFSIZE 8192

main()
{

int sock,length;
struct sockaddr_in serveur,client;
int msgsock;
char buf[1024],dir[8192];
int rval;
int client_addr_len;
```

```
/* création d'une socket */

sock=socket(AF_INET,SOCK_STREAM,0);
if(sock==-1) {
       perror("ouverture socket");
       exit(1);
}

/* lier cette socket avec l'adresse du serveur */

serveur.sin_family=AF_INET;
serveur.sin_addr.s_addr=INADDR_ANY;
serveur.sin_port=0; /* on laisse la machine choisir le port pour nous */

/* on lie la socket à une adresse */

if(bind(sock,(struct sockaddr *)&serveur,sizeof(serveur))==-1){
       perror("bind");
       exit(1);
       }
length=sizeof(serveur);
if(getsockname(sock,(struct sockaddr *)&serveur,&length)==-1) {
       perror("getsockname");
       exit(1);
}
/* affichage du numéro de port.
   Une autre solution consiste a choisir son numéro de port
   soi-même. Pour les tests, une socket restant indisponible
   apres sa fermeture pendant un certain temps, cette méthode
   nous permet d'avoir toujours un numéro de port valide.
   Un serveur en activité aura toujours le même numéro de port
   et il faudra, alors, le choisir une fois pour toutes */

printf("Serveur sur le port %d\n",ntohs(serveur.sin_port));

/* le serveur attend une connexion.
   La queue des connexions en attente a une longueur de 5 */

listen(sock,5);

do {
       client_addr_len=sizeof(client);

/* acceptation de la connexion et ouverture d'une socket
   pour cette connexion */

       msgsock=accept(sock,(struct sockaddr *) &client, &client_addr_len);
       memset(buf,0,sizeof(buf));
       if(msgsock==-1) perror("accept");

/* lecture du chemin d'accès sur la socket */

       rval=read(msgsock,buf,128);
       buf[rval]='\0';
       sprintf(dir,"%s",buf);
```

```
/* lecture du répertoire */

       read_dir(dir);

/* écriture du résultat sur la socket */

       write(msgsock,dir,strlen(dir));

/* fermeture de la socket temporaire */

       close(msgsock);
} while(TRUE);

/* Ce point n'est jamais atteint */

exit(0);
}
```

Le client version UDP

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define BUFSIZE 8192

main(argc,argv)
int argc;
char *argv[];
{

int sock;
struct sockaddr_in serveur,client;
struct hostent *hp,*gethostbyname();
char buf[BUFSIZE];
int rval;

/* création d'une socket */

sock=socket(AF_INET,SOCK_DGRAM,0);
if(sock==-1) {
       perror("ouverture de la socket");
       exit(1);
}

/* connecter la socket grâce a l'adresse passée en argument */

serveur.sin_family=AF_INET;
hp=gethostbyname(argv[1]);

/* gethostbyname retourne une structure qui contient l'adresse réseau de
```

```
l'hôte specifié */

if(hp==(struct hostent *)0) {
        fprintf(stderr,"%s: hôte inconnu\n",argv[1]);
        exit(2);
}
memcpy((char *) &serveur.sin_addr,(char *)hp->h_addr,
hp->h_length);
serveur.sin_port=htons(atoi(argv[2]));



/* envoi du message au serveur ( argv[3]=chemin d'accès du répertoire )
  Nous n'avons pas utilisé connect() donc on doit utiliser sendto()
*/

if(sendto(sock,argv[3],strlen(argv[3]),0,(struct sockaddr *) &ser-
veur,sizeof(serveur))==-1)
        perror("ecriture sur socket");

/* attente de la réception de la réponse. Nous connaissons déjà l'adresse du
serveur, nous n'avons donc pas besoin de récupérer celle-ci par recv_from(),
bien que nous aurions pu le faire */

if((rval=recv(sock,buf,BUFSIZE,0))==-1)
        perror("lecture");

if(rval==0)
        printf("\nFin de connection\n");
else {
        buf[rval]='\0';
        printf("%s",buf);
}
close(sock);

exit(0);

}
```

Dans cette version du client, nous utilisons `send_to()` en spécifiant à chaque fois l'adresse du serveur. Dans un cas comme celui-ci où le destinataire est toujours le même, on peut utiliser `connect()` et `send()`, ce qui évite de répéter l'adresse du serveur à chaque fois (ici, nous n'envoyons qu'un message au serveur ce qui diminue l'intérêt de la chose !!!).

Le serveur version UDP

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define TRUE 1
```

```
#define BUFSIZE 8192



main()
{

int sock,length;
struct sockaddr_in serveur,client;
int msgsock;


char buf[1024],dir[8192];
int rval;
int client_addr_len;

/* création d'une socket */

sock=socket(AF_INET,SOCK_DGRAM,0);

if(sock==-1) {
	perror("ouverture socket");
	exit(1);
}

/* récupération de notre adresse pour la lier à la socket */

serveur.sin_family=AF_INET;
serveur.sin_addr.s_addr=INADDR_ANY;
serveur.sin_port=0;
if(bind(sock,(struct sockaddr *)&serveur,sizeof(serveur))==-1) {
	perror("bind");
	exit(1);
}

length=sizeof(serveur);
if(getsockname(sock,(struct sockaddr *)&serveur,&length)==-1) {
	perror("getsockname");
	exit(1);
}
printf("Serveur sur le port %d\n",ntohs(serveur.sin_port));

/* boucle d'attente des datagrammes en provenance des clients */

do {
	client_addr_len=sizeof(client);
	memset(buf,0,sizeof(buf));

/* Pour pouvoir répondre au client, il est nécessaire de récupérer son
adresse dans le message que le serveur reçoit. Ici, nous devons utiliser
recv_from() */

	rval=recvfrom(sock,buf,BUFSIZE,NULL,(struct sockaddr *)
&client,&client_addr_len);
	buf[rval]='\0';
	sprintf(dir,"%s",buf);
	fprintf(stderr,"%s\n",dir);
	read_dir(dir);
```

```
/* envoi de la réponse au client */

       sendto(sock,dir,strlen(dir),0,(struct sockaddr *)
&client,client_addr_len);
} while(TRUE); /* boucle sans fin */

/* terminaison du programme, qui ne devrait jamais être atteinte */

exit(0);

}
```

Remarque au sujet de `recv_from()` : dans cette fonction, nous passons l'adresse d'une structure `sockaddr`. Celle-ci sera remplie par la fonction `recv_from()` avec l'adresse de l'expéditeur. Ceci est nécessaire pour un serveur qui ne sait pas à l'avance qui est son client (la plupart des cas : ftp ...).

2. TLI

Introduction

TLI (Transport Layer Interface) fut introduite en 1986 dans système V.4. Elle fournit une interface à la couche transport du modèle OSI, s'appuyant sur le mécanisme général d'entrées-sorties par streams.

On définit deux termes nouveaux qui reviennent souvent dans TLI:

- extrémité de transport: TLI nomme ainsi les deux processus communicants.
- fournisseur de transport: l'ensemble des routines, sur l'ordinateur hôte, qui fournissent le support de communication pour le processus utilisateur.

Il faut bien noter que TLI n'est pas un fournisseur de transport mais un interface.

TLI s'utilise de façon identique aux sockets, une différence notable se situant au niveau de l'initialisation d'une extrémité de connexion. Chaque protocole de transport, est designé par un fichier device (unité) . la fonction `t_open()` permettra d'ouvrir ce fichier et ainsi d'initialiser l'extrémité de connexion. Trois types de service sont proposés :

- T_COTS orienté connexion avec fermeture brutale (perte de données)
- T_COTS_ORD orienté connexion avec possibilité de déconnexion ordonnée (pas de perte) comme TCP
- T_CLTS non connecté, par exemple, UDP.

Les adresses sous TLI

Une des particularités de TLI par rapport aux sockets est d'autoriser une indépendance vis à vis du protocole utilisé. Par exemple, TLI n'impose donc aucune restriction sur la structure d'une adresse transport, l'interprétation en est laissée au fournisseur de transport. Cela permet d'introduire de nouveaux protocoles sans toucher au noyau de l'application.

On a donc une structure générique qui permet d'échanger les données entre les fonctions TLI et le code de l'utilisateur.

```
#include <tiuser.h>

struct netbuf {
    unsigned int maxlen; /* taille maximum de buf */
    unsigned int len; /* taille réelle de buf */
    char *buf; /* données dépendant du protocole */
};
```

Cette structure permet de passer une structure d'adresse correspondant au protocole choisi, `buf` pointera sur l'adresse de cette structure.

Les principales fonctions des TLI

- `t_open()`

Permet d'initialiser une connexion, cette fonction renvoie un descripteur de fichier.

- `t_alloc()`

Permet d'allouer dynamiquement les structures échangées entre TLI et le code utilisateur. Cette routine est associée à `t_free()` pour la libération de l'espace

- `t_bind()`

Permet d'associer une adresse à une extrémité de connexion.

- `t_connect()`

Pour une communication orientée connexion, le client envoie une requête de connexion au serveur.

- `t_listen()`

Pour une communication orientée connexion, le serveur se bloque sur `t_listen()` en attente d'une requête de connexion venant d'un client.

- `t_accept()`

Permet d'accepter une requête lorsqu'une demande de connexion est arrivée au serveur.

- `t_snd()` et `t_rcv()`

Permettent l'échange de données entre les extrémités de connexion en mode orienté connexion.

-`t_sndudata()` et `t_rcvudata()`

Permettent l'échange de données (datagrammes) entre les extrémités de connexion en mode sans connexion.

- `t_close()`

Ferme une extrémité de connexion.

- `t_look()`

Une extrémité de connexion a toujours un évènement en cours (un état). Ces évènements peuvent être retrouvés grâce à la fonction t_look(). Parmi ceux-ci, on peut trouver:

- T_DATA, des données normales ont été reçues
- T_EXDATA des données urgentes ont été reçues
- T_DISCONNECT une demande de déconnexion a été reçue...

Ceci est très utile en mode asynchrone.

Une communication orientée connexion

On peut modéliser cette communication comme représenté figure 2.1.

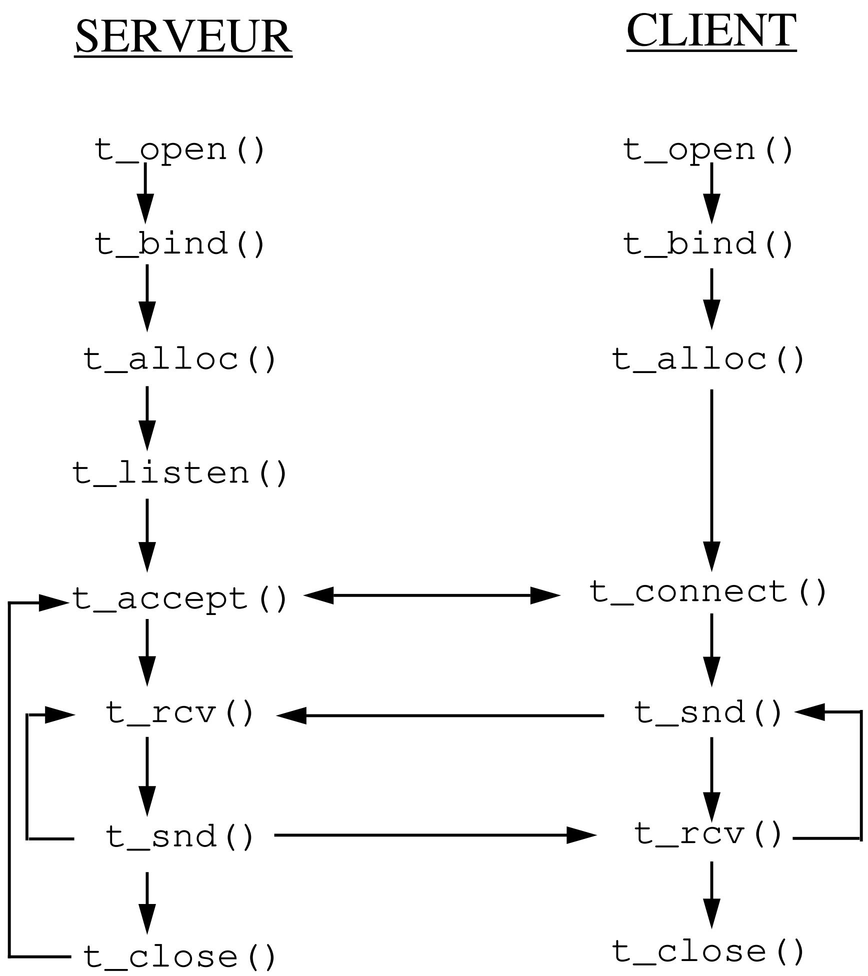

figure 2.1: une communication orientée connexion

On peut noter la ressemblance de ce schéma avec celui correspondant à une communication orientée connexion utilisant des sockets.

t_free() n'a pas été représentée, les structures allouées avec t_alloc() (une ou deux par processus) étant libérées automatiquement à la mort du processus.

Une communication sans connexion

De même, on peut modéliser une communication sans connexion comme représenté figure 2.2.

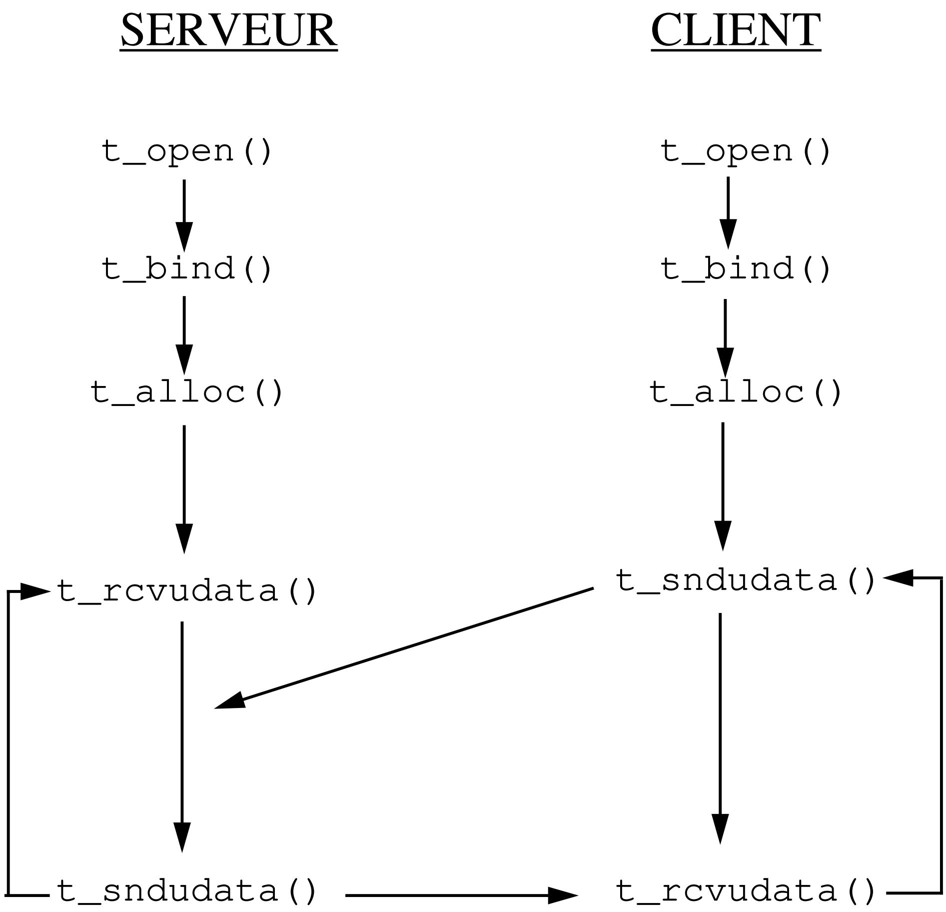

figure 2.2: une communication sans connexion

Programmation avancée

On retrouve, avec TLI, les même possibilités de programmation avancée qu'avec les sockets: données urgentes (nommées "expedited data" dans TLI), communication asynchrone,...Une possibilité intéressante à signaler est la possibilité de gérer plusieurs requêtes de clients en même temps, ce qui permet, par exemple, d'associer une priorité aux clients.

En ce qui concerne les données urgentes, il est à noter (et ceci est valable aussi pour les sockets) qu'un seul message urgent peut être en attente.

Des exemples

Donnons un exemple de communication grâce à TLI. Cette exemple est le même que celui du chapitre précédent et est orienté connexion. Il est, bien sûr, possible de faire de même sans connexion.

Le client version TCP

```
/* inet.h : les déclarations nécessaires au client :
       - le fichier device associé à TCP
       - l'adresse du serveur (ici, ensisun). Cette adresse aurait pu être
choisie dynamiquement.
*/

#include <stdio.h>
#include <fcntl.h>
#include <tiuser.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define DEV_TCP "/dev/tcp"

#define SERV_HOST_ADDR "192.33.174.34"

#define MAXLINE 255

/* tli_cl.c : partie client du programme de listing des répertoires d'une
machine distante, version TLI-tcp
*/

#include "inet.h"
#define BUFSIZE 8192

main(argc,argv)
int argc;
char *argv[];
{

int tfd;
int nbytes;
int flags=0;
char buf[BUFSIZE];

char *t_alloc();

struct t_call *callptr;
struct sockaddr_in serv_addr;

/* ouverture du device associé aux TLI protocole TCP */

if((tfd=t_open(DEV_TCP,O_RDWR,0))<0) {
       printf(stderr,"erreur t_open\n");
       exit(1);
```

```
}


/* on lie le descripteur à l'adresse (on laisse le choix de l'adresse à la
machine car on ne s'en sert pas).
*/

if(t_bind(tfd,(struct t_bind *)0,(struct t_bind *)0)<0) {
	fprintf(stderr,"erreur t_bind\n");
	exit(2);
}

/* Sous UNIX BSD, on utiliserait bzero() à la place de memset() */

memset((char *)&serv_addr,0,sizeof(serv_addr));

serv_addr.sin_family=AF_INET;
serv_addr.sin_addr.s_addr=inet_addr(SERV_HOST_ADDR);
serv_addr.sin_port=htons(atoi(argv[1]));

/* allocation de la structure t_call */

if((callptr=(struct t_call *)t_alloc(tfd,T_CALL,T_ADDR))==0) {
	fprintf(stderr,"erreur t_alloc\n");
	exit(3);
}

/* La structure addr est initialisée avec une structure sockaddr_in */

callptr->addr.maxlen=sizeof(serv_addr);
callptr->addr.len=sizeof(serv_addr);
callptr->addr.buf=(char *)&serv_addr;
callptr->opt.len=0;
callptr->udata.len=0;

/* connexion au serveur */

if(t_connect(tfd,callptr,(struct t_call *)0)<0) {
	fprintf(stderr,"erreur t_connect\n");
	exit(4);
}

/* envoi du chemin d'accès du répertoire */

if(t_snd(tfd,argv[2],strlen(argv[2]),0)==-1) {
	t_error("erreur t_snd");
	exit(1);
}

/* réception de la réponse */

while((nbytes=t_rcv(tfd,buf,1024,&flags)) !=-1) {
	if( fwrite(buf,1,nbytes,stdout)==-1) {
		fprintf(stderr,"erreur fwrite\n");
		exit(1);
	}
}
```

```
/* déconnexion : si on a reçu une demande de déconnexion, on sort, sinon on
envoie une demande de déconnexion.
*/

if((t_errno==TLOOK)&&(t_look(tfd)==T_ORDREL)) {
        if(t_rcvrel(tfd)==-1) {
                  t_error("erreur t_rcvrel");
                  exit(1);
        }
        if(t_sndrel(tfd)==-1) {
                  t_error("erreur t_sndrel");
                  exit(1);
        }
        exit(0);
}

}
```

On remarque dans cette exemple, la similitude d'utilisation de TLI avec celle des sockets, même si les fonctions ne sont pas deux à deux identiques.

Le serveur version TCP

```
/* Partie serveur du programme de listing des répertoires
   d'une machine distante, version TLI
*/

#include "inet.h"
#include <stropts.h>
#include <sys/conf.h>
#include <signal.h>

int newtfd;

/* dans cette version, on utilise quelques possibilités avancées de TLI,
notamment les signaux associés. connrelease() est appelée pour la decon-
nexion si le serveur reçoit le signal SIGPOLL (évènement en attente sur le
descripteur) */

connrelease()
{

/* t_look() permet de connaitre l'état d'un descripteur */

if(t_look(newtfd)==T_DISCONNECT) {
        fprintf(stderr,"erreur déconnexion\n");
        exit(12);
}
exit(0);
}
```

```
/* Lorsque le serveur reçoit une demande de connexion, il l'accepte, génére
un fils qui va la traiter en éxécutant run_server().
*/

run_server(listen_fd)
int listen_fd;
{
int nbytes;
FILE *logfp;
char buf[8192];
int flags;

switch(fork()) {
	/* erreur sur le fork() */
	case -1:
		perror("erreur fork");
		exit(1);
	/* code pour le fils */
	case 0:
	/* le fils ferme le descripteur principal */
		if(t_close(listen_fd)==-1) {
		     t_error("erreur t_close dans run_server fils");
		     exit(1);
		}
	/* prise en compte du signal SIGPOLL */
		signal(SIGPOLL,connrelease);
		if(ioctl(newtfd,I_SETSIG,S_INPUT)==-1) {
		     perror("erreur ioctl I_SETSIG");
		     exit(1);
		}
	/* réception du message venant du client */
		nbytes=t_rcv(newtfd,buf,1024,&flags);
		buf[nbytes]='\0';
	/* traitement */
		read_dir(buf);
	/* envoi du résultat au client */
		t_snd(newtfd,buf,strlen(buf),0);
	/* envoi de la demande de déconnexion au client */
		if(t_sndrel(newtfd)==-1) {
		     t_error("erreur t_sndrel");
		     exit(1);
		}
		pause();
	/* code pour le père */
	default:
	/* fermeture du descripteur temporaire */
		if(t_close(newtfd)==-1) {
		     t_error("erreur t_close accept_call pere");
		     exit(1);
		}
	/* retour dans la boucle d'attente des connexions */
		return;
	}
}
```

```
/* acceptation d'une nouvelle connexion, ouverture d'un nouveau descripteur
qui servira pour la connexion */

int accept_call(listen_fd,callptr,name)
int listen_fd;
struct t_call *callptr;
char *name;
{

int restfd;
extern int t_errno;


/* ouverture du descripteur */
if((restfd=t_open(name,O_RDWR,(struct t_info *)0))<0) {
       fprintf(stderr,"erreur t_open dans accept_call\n");
       exit(7);
}

/* liaison entre adresse et descripteur */
if(t_bind(restfd,(struct t_bind *)0,(struct t_bind *)0)<0) {
       fprintf(stderr,"erreur t_bind dans accept_call\n");
       exit(8);
}

/* acceptation de la connexion sur descripteur temporaire */
if(t_accept(listen_fd,restfd,callptr)<0) {
       fprintf(stderr,"erreur t_accept dans accept_call\n");
       exit(9);
}
/* renvoie le descripteur */
       return(restfd);
}


/* programme principal */

main(argc,argv)
int argc;
char *argv[];
{

int tfd,clilen,childpid;
struct sockaddr_in cli_addr,serv_addr;
struct t_bind req;
struct t_call *callptr;


/* ouverture d'une extrémité de connexion avec protocole TCP */

if((tfd=t_open(DEV_TCP,O_RDWR,(struct t_info *)0))<0) {
       fprintf(stderr,"erreur t_open\n");
       exit(1);
}
```

```
/* remplissage de la structure d'adresse */

/* la structure générique de TLI pointe sur une structure d'adresse de TCP
*/

memset((char *)&serv_addr,0,sizeof(serv_addr));

serv_addr.sin_family=AF_INET;
serv_addr.sin_addr.s_addr=htonl(INADDR_ANY);
serv_addr.sin_port=htons(atoi(argv[1]));

req.addr.maxlen=sizeof(serv_addr);
req.addr.len=sizeof(serv_addr);
req.addr.buf=(char *)&serv_addr;
req.qlen=5;

/* on lie l'adresse et le descripteur */

if(t_bind(tfd,&req,(struct t_bind *)0)<0) {
        fprintf(stderr,"erreur t_bind\n");
        exit(2);
}

/* allocation de la structure t_call */

if((callptr=(struct t_call *)t_alloc(tfd,T_CALL,T_ADDR))==NULL) {
        fprintf(stderr,"erreur t_alloc\n");
        exit(3);
}

/* boucle d'attente des connexions */

for(;;) {

/* attente d'une connexion, le serveur se bloque sur t_listen() */

        if(t_listen(tfd,callptr)<0) {
                fprintf(stderr,"erreur t_listen\n");
                exit(4);
        }

/* accepte une connexion et utilisation d'un autre
   descripteur pour cette connexion, lance l'application
*/

        if((newtfd=accept_call(tfd,callptr,DEV_TCP))<0) {
                fprintf(stderr,"erreur accept_call\n");
                exit(5);
        }
        else run_server(tfd);

}
}
```

3.INTRODUCTION AUX RPC

Principe

L'appel de procédure à distance est un modèle d'application réseau de haut niveau. Les RPC permettent d'assimiler les requêtes d'un client à des appels de procédure. Quand un appel de procédure distante est effectué, les paramètres de l'appel sont passés à la procédure distante, l'appelant (le client) attend la réponse qui est renvoyée par la procédure appelée. L'utilisation des RPC permet au programmeur d'une application distribuée de ne pas se préoccuper des détails de l'interface avec le réseau. L'indépendance des RPC vis à vis du transport, sépare complètement l'application des éléments logiques et physiques de la communication des données.

La figure suivante montre un appel classique d'une procédure distante.

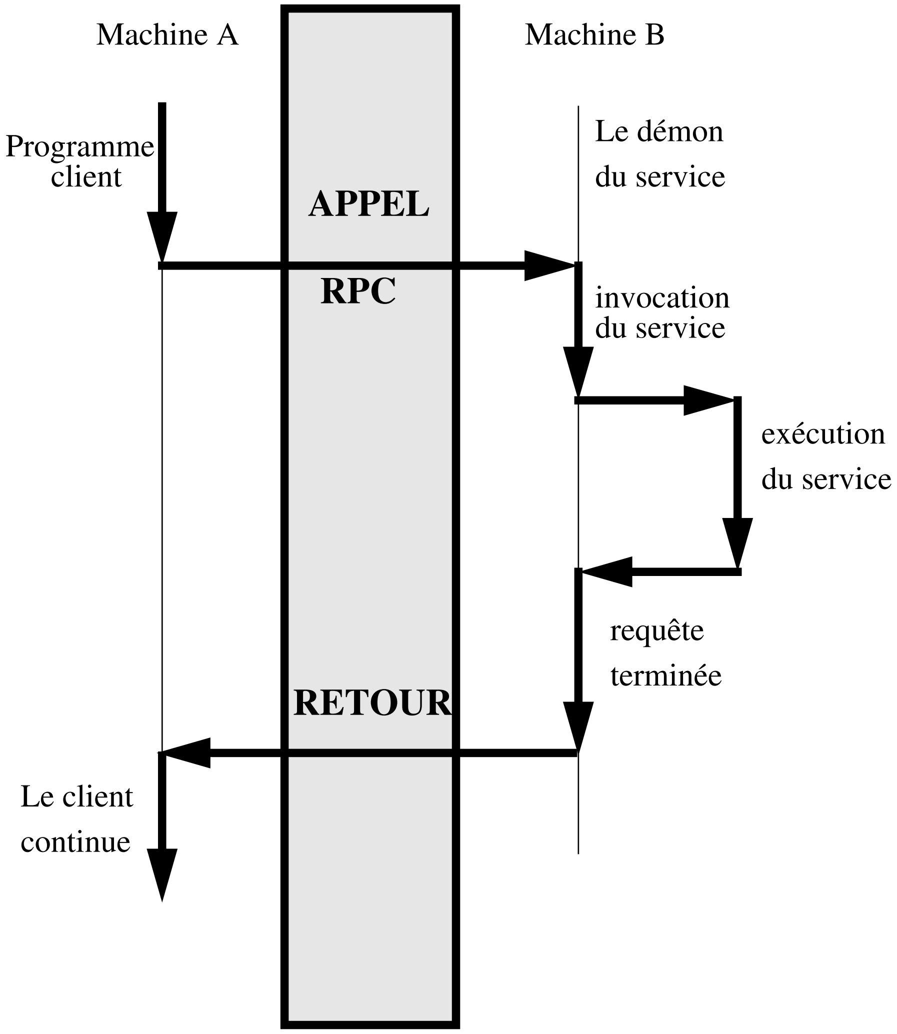

figure 3.1: un appel de procédure distante

Les RPC dans OSI

Défini après le modèle OSI, les RPC trouvent leur place naturellement dans celui-ci, dans la touche session, comme représenté dans la figure 3.2.

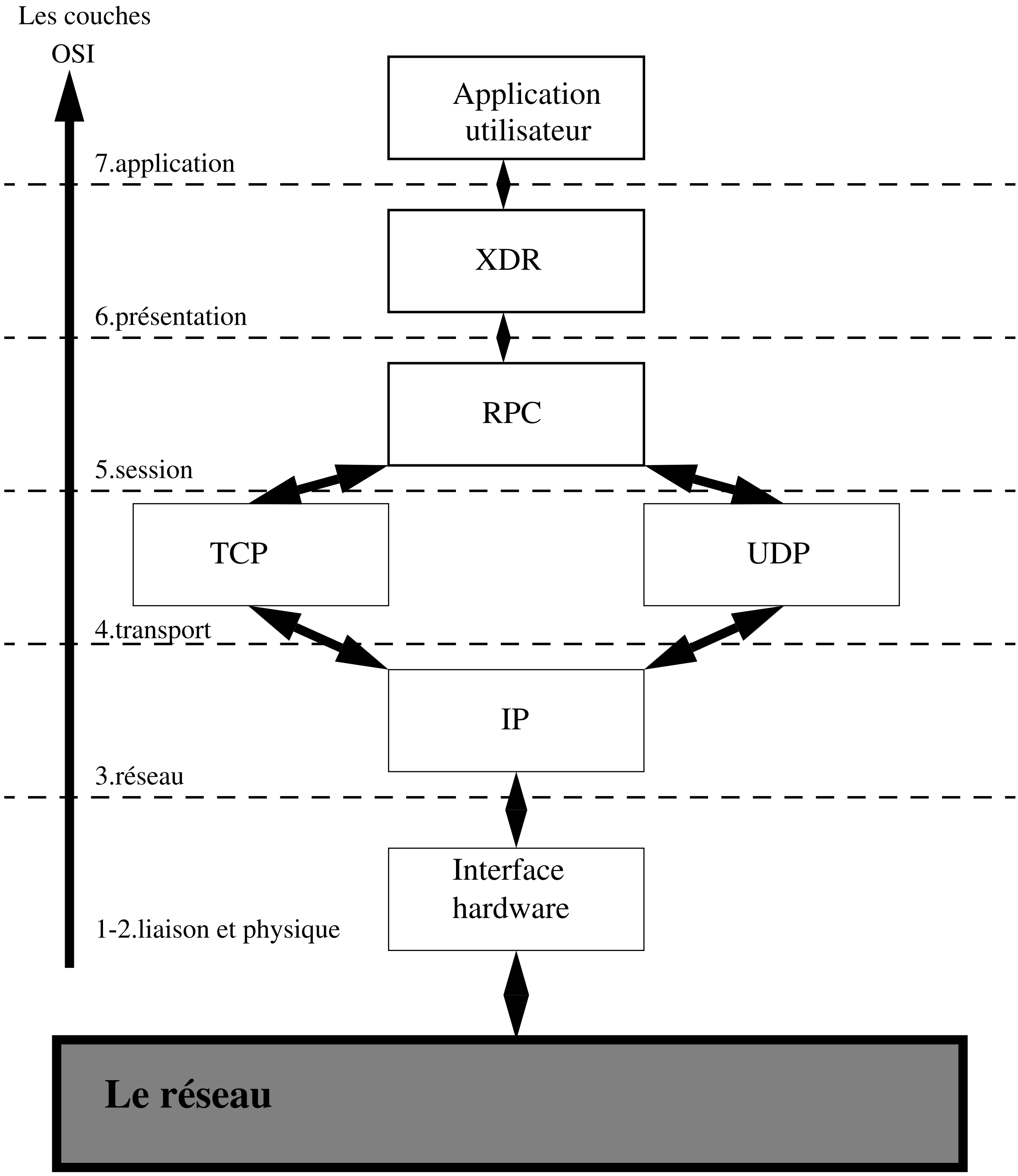

figure 3.2: les RPC dans le modèle OSI

Nous introduirons la couche présentation (XDR) ultérieurement.

Après quelques remarques générales nécessaires à la compréhension de la programmation des RPC, nous détaillerons, dans les chapitres suivants, l'utilisation des RPC.

Il est possible d'utiliser le mécanisme d'appel de procédure distante de différentes manières. Celles-ci fournissent différents degrés de facilité pour le programmeur contre-balancés par une maîtrise plus ou moins forte des mécanismes sous-tendants.

De plus, nous verrons l'utilisation de l'utilitaire `rpcgen` qui simplifie encore la programmation des RPC par la génération automatique des portions de code spécifiques aux RPC d'une application client-serveur.

Numéro de programme, de version et de procédure

Chaque procédure RPC est identifée de manière unique par un numéro de programme, un numéro de version et un numéro de procédure.
Le numéro de programme identifie un ensemble de procédures distantes en relation entre elles, chacune ayant un numéro de procédure différent. Chaque programme a aussi un numéro de version, ce qui permet de garder le même numéro de programme en cas de changements dans le service.

Des changements dans le programme (ajout de nouvelles procédures, changements dans les paramètres ou les valeurs de retour des procédures...) nécessitent un nouveau numéro de version.

Sélection du transport

La sélection du transport permet aux utilisateurs de choisir de façon dynamique le transport parmi ceux disponibles. Cette sélection utilise deux mécanismes, le fichier `/etc/netconfig` et la variable d'environnement NETPATH. `/etc/netconfig` comporte la liste des transports disponibles sur la machine et leur type. NETPATH est optionnelle et permet aux utilisateurs de sélectionner un transport ou une liste de transports parmi ceux disponibles dans le fichier `/etc/netconfig`.
Pour plus de détails sur le fichier `/etc/netconfig`, vous pouvez consulter le man `netconfig(4)`.

La variable d'environnement NETPATH est une simple liste de noms séparés par ":", par exemple: udp:tcp.

Par la mise en place de cette variable, l'utilisateur spécifie l'ordre dans lequel l'application va essayer les transports. Si cette variable n'existe pas, l'application essayera, dans l'ordre du fichier `/etc/netconfig`, tous les transports disponibles.

Bien sûr, une application peut forcer l'utilisation d'un transport particulier et ainsi ignorer NETPATH. Le choix entre ces deux méthodes (choix du transport prédéterminé ou dynamique) est à la convenance du programmeur.

Une application qui prend en compte le transport commencera par appeler:

- `setnetconfig()`
- `getnetconfig()`
- `endnetconfig()`, pour chercher, dans `/etc/netconfig`, les types de transports possibles. Ces données seront stockées dans une structure locale `netconfig` pour un usage ultérieur.

Ces mécanismes permettent un contrôle très fin sur la sélection du transport. Un utilisateur peut spécifier son transport préféré, et si elle peut, l'application l'utilisera. Sinon, elle essayera automatiquement les autres ayant les bonnes caractéristiques.

Il est à noter que cette facilité n'est présente que sur les dernières versions des RPC (SunOS 5) et est liée à l'utilisation des TLI. Dans les versions précédentes, les RPC étaient basées sur l'interface socket et ne permettaient pas l'utilisation de ce mécanisme.

Nous verrons plus tard les différences (notables) entre ces deux versions des RPC et les problèmes de compatibilités que cela entraîne.

L'utilitaire `rpcbind`

Avec la nouvelle version des RPC, on ne peut faire aucune supposition sur le port où sera localisée un service (procédure distante). Il est donc nécessaire pour un client d'avoir un moyen de trouver l'adresse du serveur. Ceci est réalisé grâce à rpcbind.

`rpcbind` dispose des fonctions suivantes :

- enregistrement d'un serveur
- suppression d'un serveur
- récupération du numéro de programme, du numéro de version et du transport utilisé pour une procédure donnée.
- réalisation d'un appel de procédure distante pour un client.

Identification des procédures

Ceci est donc réalisé par `rpcbind` pour permettre à un client de connaitre l'adresse d'une procédure. Comme `rpcbind` doit être atteint par le client, il doit se trouver sur un port bien connu (wellknow port) pour les principaux transports (TPC, UDP). Ce port est le numéro 111. C'est le seul service RPC qui doit avoir une adresse connue.

Pour trouver l'adresse d'un service, un client envoie un message au démon `rpcbind` sur la machine du serveur. Celui-ci lui renvoie l'adresse si le programme est enregistré.

Mais `rpcbind` permet aussi à un client de faire l'appel d'une procédure sans connaitre l'adresse de celle-ci. Le client envoie directement les numéros identifiant la procédure et les paramètres de celle-ci à `rpcbind` qui fera l'appel lui-même. Il renverra le résultat au client en y incluant l'adresse de la procédure pour un usage ultérieur.

L'utilitaire rpcinfo

Ce programme permet d'obtenir des informations sur les services disponibles sur une machine. Par exemple :

```
ensisun(9:18am)rpcinfo -s
   program version(s) netid(s)                         service     owner
    100000  2,3,4     udp,tcp,ticlts,ticotsord,ticots  rpcbind     superuser
    100029  2,1       ticots,ticotsord,ticlts          keyserv     superuser
    100078  4         ticots,ticotsord,ticlts          kerbd       superuser
    100087  10        udp                              admind      superuser
    100011  1         ticlts,udp                       rquotad     superuser
    100099  1         ticots,ticotsord,ticlts          -           superuser
    100002  3,2       ticlts,udp                       rusersd     superuser
```

```
    100012  1        ticlts,udp                         sprayd      superuser
    100008  1        ticlts,udp                         walld       superuser
    100024  1        ticots,ticotsord,ticlts,tcp,udp    status      superuser
    100001  4,3,2    ticlts,udp                         rstatd      superuser
    100021  2,3,1    ticots,ticotsord,ticlts,tcp,udp    nlockmgr    superuser
    100068  4,3,2    udp                                -           superuser
    100083  1        tcp                                -           superuser
    100020  2        ticots,ticotsord,ticlts,tcp,udp    llockmgr    superuser
    300215  1        udp                                -           superuser
    100003  2        udp                                nfs         superuser
    100005  2,1      ticots,ticotsord,tcp,ticlts,udp    mountd      superuser
    100026  1        ticots,ticotsord,ticlts,tcp,udp    bootparam   superuser
    300214  1        udp                                -            1106
1073773040  1        udp                                -            1106
1073759175  1        udp                                -            4013
1073765183  1        udp                                -            1020
```

4. LA PROGRAMMATION DES RPC

Les différents niveaux de l'interface RPC

Comme nous l'avons déjà signalé, RPC a différents niveaux d'interfaces, certains simples à utiliser, d'autres plus complexes mais offrant une plus grande maitrîse des mécanismes de communication.

Ces différents niveaux sont, du plus simple au plus difficile (au niveau programmation):

- l'interface simplifiée
- le niveau haut (top level)
- le niveau intermédiaire
- le niveau expert
- le niveau bas.

Nous allons ici détailler les principales fonctions de ces différents niveaux.

L'interface simplifiée

C'est le plus haut niveau de programmation des RPC, l'indépendance vis à vis du transport est totale. Les fonctions associées cachent entièrement les mécanismes de communications. On peut en décrire trois importantes:

- `rpc_call()`: cette fonction permet de faire l'appel complet à une procédure distante.
- `rpc_reg()`: cette fonction permet d'enregistrer un programme auprès de `rpcbind`.
- `svc_run()`: lance un serveur.

Avant de détailler ces fonctions, il est nécessaire d'introduire la représentation externe des données (eXternal Data Representation ou XDR).

Lors d'échanges de donnés entre différentes machines, se pose le problème de la non-unicité de la représentation interne des données. Le protocole XDR (couche présentation), défini par SUN, permet de définir une représentation standard des données échangeables entre machines.

Le problème se posait, pour les sockets, sur les entiers courts et longs (numéros de port, adresses) et imposait l'utilisation de fonctions telles que ntohl(), htonl()...Le même problème se pose pour les RPC mais avec tous les types de données (entiers, réels,...).

Le type XDR est un flot (défini dans xdr.h), il contient toutes les informations nécessaires à sa gestion:

- nature (codage ou décodage)
- dernière opération effectuée....

Le principe de codage et décodage (on parle aussi de sérialisation et désérialisation) est d'ajouter ou d'extraire des informations dans un flot en les codant ou les décodant. A chaque type correspond une fonction qui réalise cela suivant la nature du flot. Pour les types de base, ces fonctions sont déjà implantées: xdr_void(), xdr_char(), xdr_int(), xdr_short(), xdr_float()...

De plus, il existe un type string qui possède lui aussi sa fonction de sérialisation-désérialisation xdr_string().

Toutes ces fonctions ont donc une convention sur leur nom: xdr_ suivi du nom du type. Lorsqu'il est nécessaire d'écrire des fonctions similaires pour des types particuliers, il est de bon ton de garder cette convention.

Nous pouvons à présent étudier les fonctions de l'interface simplifiée.

`rpc_call()`

Cette fonction réalise donc l'appel complet à une procédure distante. Elle prend neuf paramètres.

Le premier est le nom de la machine hôte du serveur.

Les trois suivants sont les numéros de programme, de version et de procédure de la fonction à appeler.

Les cinquième et sixième arguments sont la procédure pour encoder l'argument à passer à la procédure distante et l'argument lui-même.

Les septième et huitième arguments sont la procédure pour décoder le résultat retourné par la procédure distante et le résultat lui-même.

Le dernier est le protocole choisi (`nettype`). On pourra donc choisir, ici, un protocole ou le faire de façon dynamique (comme vu au chapitre précédent).

Donnons tout de suite un exemple de l'utilisation de cette fonction avec le client du programme `rls`, version RPC.

```
/* rls.h : fichier des déclarations pour rls */

#define DIR_SIZE 8192

#define DIRPROG ((u_long) 0x20000000)
#define DIRVERS ((u_long) 1)
#define READDIR ((u_long) 1)
```

On voit ici le premier exemple des numéros de programme, version et procédure. Les numéros de version et de procédure (égaux à un) se comprennent bien. Le numéro de programme doit être choisi avec discernement et fera l'objet d'un prochain paragraphe.

```
/* rls.c : client du programme rls utilisant les RPC */
/* il s'utilise de la manière suivante :
       rls nom-machine-hote-serveur chemin-d-acces-repertoire
*/

#include <stdio.h>
#include "/usr/ucbinclude/strings.h"
#include <rpc/rpc.h>
#include "rls.h"

/* le programme principal ne fait que récupérer les arguments de la ligne de
commande et appeler la fonction read_dir du client */

main(argc,argv)
int argc;
char *argv[];
{
char dir[DIR_SIZE];

strcpy(dir,argv[2]);
read_dir(argv[1],dir);


printf("%s\n",dir);

exit(0);
}

/* la fonction read_dir du client appelle la procédure distante */

read_dir(host,dir)
char *dir,*host;
{

extern bool_t xdr_dir();
enum clnt_stat clnt_stat;

/* rpc_call renvoie 0 si tout s'est bien passé */

clnt_stat=rpc_call(host,DIRPROG,DIRVERS,READDIR,xdr_dir,dir,
xdr_dir,dir,"visible");

if(clnt_stat!=0) clnt_perrno(clnt_stat);

}
```

On voit ici l'utilisation d'une fonction `xdr_dir()` qui code et décode le type de `dir` qui est `char *`. Cette fonction est contenue dans le fichier `rls_xdr.c` que voici.

```
/* rls_xdr.c : contient la fonction xdr_dir */

#include <rpc/rpc.h>
#include "rls.h"

bool_t xdr_dir(xdrs,objp)
XDR *xdrs;
char *objp;
{
return(xdr_string(xdrs,&objp,DIR_SIZE));
}
```

La fonction `xdr_dir()` utilise la fonction `xdr_string()` qui existe de base. Elle prend en argument une structure XDR qui permet de choisir le sens de transformation (codage, décodage) et un pointeur sur la variable à transformer.

rpc_reg()

Cette fonction permet d'enregistrer un programme (avec ses numéros) auprès de `rpcbind`. Elle prend 6 arguments.

Les trois premiers sont les numéros de programme, version et procédure de la fonction à enregistrer.

Le quatrième pointe sur une chaine contenant le nom de la fonction qui implémente la procédure distante.

Le cinquième pointe sur la fonction de transformation de l'argument de la procédure.

Le sixième pointe sur la fonction de transformation du résultat de la procédure distante.

Le dernier spécifie le type de protocole choisi.

svc_run()

`svc_run()` n'appartient pas à un niveau donné de programmation des RPC. Elle est appelée à chaque niveau et permet de lancer le serveur qui va gérer l'arrivée des appels aux procédures liées au programme.

Le serveur du programe `rls` qui correspond au client vu ci-dessus est un bon exemple de l'utilisation de ces fonctions.

```
/* rls_svc.c : côté serveur de rls */

#include <rpc/rpc.h>
#include "rls.h"

main()
{

/* déclarations des fonctions utilisées :
        - xdr_dir() contenue dans rls_xdr.c
        - read_dir() est la fonction qui effectue réellement la lecture du ré-
pertoire et qui a été vue dans les précédents chapitres.
```

```
*/
extern bool_t xdr_dir();
extern char *read_dir();

/* enregistrement de la procédure read_dir (de read_dir.c) */

rpc_reg(DIRPROG,DIRVERS,READDIR,read_dir,xdr_dir,xdr_dir);

/* lancement du serveur */

svc_run();
}
```

Cet exemple très simple (et maintenant complet) montre bien la simplicité d'utilisation des RPC de cette manière. Les détails concernant les communications sont effectivement cachés.

A titre d'information, voici la version locale de ce programme.

```
/* lls.c : listing de répertoire local */

#include <stdio.h>
#include "/usr/ucbinclude/strings.h"
#include "rls.h"


main(argc,argv)
int argc;
char *argv[];
{
char dir[DIR_SIZE];

/* on appelle la procedure locale (la fonction read_dir() de read_dir.c */

strcpy(dir, argv[1]);
read_dir(dir);

/* on sort le resultat */

printf("%s\n",dir);
}
```

Le niveau haut

La programmation au niveau haut rajoute la possibilité de choisir le type de transport désiré pour le serveur (ce qui n'était pas possible dans l'interface simplifiée). On choisit le type de transport et non le transport.

On peut observer les changements par rapport à la version précédente sur l'exemple suivant.

```
/* rls.c : partie client du programme de listing de répertoires
       version RPC niveau haut
*/
```

```
#include <stdio.h>
#include "/usr/ucbinclude/strings.h"
#include <rpc/rpc.h>
#include "rls.h" /* le même que dans les précédentes versions */

extern bool_t xdr_dir();


main(argc,argv)
int argc;
char *argv[];
{

char dir[DIR_SIZE];

/* ici nous devrons définir un temps de réponse maximum du serveur */
struct timeval time_out;

/* les variables nécessaires à la communication */
CLIENT *client;
enum clnt_stat stat;

/* le type de transport */
char *nettype;

if(argc!=3 && argc!=4){
       fprintf(stferr,"usage: %s hote path [type_transport]\n",argv[0]);
       exit(1);
}

/* on affecte nettype avec netpath ou avec la valeur passée en argument */
if(argc==3) nettype="netpath";
else nettype=argv[3];

/* on crée le client */
client = clnt_create(argv[1],DIRPROG,DIRVERS,nettype);
if(client==(CLIENT *)NULL) [
       clnt_pcreateerror("Je ne peux pas créer le client\n");
       exit(2);
}

strcpy(dir,argv[2]);


/* on fait l'appel au serveur */

time_out.tv_sec=25;
time_out.tv_usec=0;

stat=clnt_call(client,READDIR,xdr_dir,dir, xdr_dir,dir,time_out);
if(stat!=RPC_SUCCESS) {
       clnt_perror(client,"L'appel a échoué\n");
       exit(3);
}

printf("%s\n",dir);
```

```
exit(0);

}
```

A ce niveau, on gère une structure CLIENT, qui sert à effectuer les appels RPC. Le programme suivant montre la partie serveur. Ici, nous sonnes obligés de modifier la fonction read_dir(). Celle-ci, telle qu'elle est décrite ci-dessous, possède les paramètres qui seront nécessaires à toutes les procédures distantes de même type. On pourra donc construire les procédures distantes sur ce modèle.

```
/* rls_svc.c : partie serveur du logiciel de listing de répertoire
                 version RPC niveau haut
*/

#include <sys/types.h>
#include <dirent.h>
#include <rpc/rpc.h>
#include "rls.h"

extern bool_t xdr_dir();

main(argc,argv)
int argc;
char *argv[];
{

int transpnum;
char *nettype;
void read_dir();

if(argc>2) {
        fprintf(stderr,"Usage: %s [type_transport]\n",argv[0]);
        exit(1);
}
if(argc==2) nettype=argv[1];
else nettype="netpath";

/* création du service */
transpnum=svc_create(read_dir,DIRPROG,DIRVERS,nettype);

if(transpnum==0) {
        fprintf(stderr,"%s: impossible de creer le service %s\n",argv[0],net-
type);
        exit(2);
}
```

```
/* lancement du serveur */

svc_run();
}

/* read_dir.c : liste un directory. Ici, des paramètres permettent de gérer
la communication. Leurs structures ne nous intéresse pas. Il sont nécessaires
aux différents appels aux fonctions RPC, mais sont gérés, à ce niveau,
par les RPC. */


static void read_dir(rqstp,transp)
struct svc_req *rqstp;
SVCXPRT *transp;

{
DIR *dirp;
struct dirent *d;
static char dir[8192];

/* cette fonction permet de récupérer les arguments passés par le client */
svc_getargs(transp,xdr_dir,dir);

/* ouvrir le directory */

dirp=opendir(dir);


/* svc_sendreply() permet de renvoyer le résultat au client */
if(dirp==NULL)
       if(!svc_sendreply(transp,xdr_dir,NULL)) {
            svcerr_systemerr(transp);
       }


/* recopie des fichiers dans le buffer dir */

dir[0]='\0';
while(d=readdir(dirp)) sprintf(dir,"%s%s\n",dir,d->d_name);

/* retourner le resultat */

closedir(dirp);

if(!svc_sendreply(transp,xdr_dir,dir)) {
       svcerr_systemerr(transp);
}

}
```

Le niveau intermédiaire

A ce niveau, le programmeur peut choisir son transport. Il va donc utiliser les fonctions `getnetconfigent()` et `freenetconfigent()` (voir `getnetconfig(3N)`). Cela introduit plusieurs changements dans le code par rapport à l'exemple précédent :

- il faut inclure `netconfig.h` pour les structures et les déclarations liées à `getnet-config`
- il faut remplir une structure netconfig
- `client_create()` (respectivement `svc_create()`) est remplacée par `client_tp_create()` (respectivement `svc_tp_create()`) où `nettype` est remplacé par la structure `netconfig`.
- un service est designé par une structure `SVCXPRT` définie dans `rpc.h` et renvoyée par `svc_tp_create()`.

```
/* rls.c : partie client du programme de listing de répertoires
       version RPC niveau intermédiaire
*/

#include <stdio.h>
#include "/usr/ucbinclude/strings.h"
#include <netconfig.h> /* le fichier à rajouter */
#include <rpc/rpc.h>
#include "rls.h"

extern bool_t xdr_dir();

main(argc,argv)
int argc;
char *argv[];
{
char dir[DIR_SIZE];
struct netconfig *nconf;/* la structure netconfig */
char *netid; /* le nom du protocole */
struct timeval time_out;
CLIENT *client;
enum clnt_stat stat;


if(argc!=4){
       fprintf(stderr,"usage: %s hote path [netid]\n",argv[0]);
       exit(1);
}

/* récupération du nom du protocole */

netid=argv[3];

/* remplissage de la structure netconfig */

if((nconf=getnetconfigent(netid))==(struct netconfig *)NULL) {
       fprintf(stderr,"Mauvais netid : %s\n",netid);
```

```
        exit(2);
}

/* création du client */

client = clnt_tp_create(argv[1],DIRPROG,DIRVERS,nconf);
if(client==(CLIENT *)NULL) {
        clnt_pcreateerror("Je ne peux pas creer le client\n");
        exit(3);
}

/* libération de la structure netconfig */

freenetconfigent(nconf);

/* appel de la procédure distante */

strcpy(dir,argv[2]);

time_out.tv_sec=25;
time_out.tv_usec=0;

stat=clnt_call(client,READDIR,xdr_dir,dir, xdr_dir,dir,time_out);
if(stat!=RPC_SUCCESS) {
clnt_perror(client,"L'appel a echoue\n");
exit(4);
}

printf("%s\n",dir);

exit(0);

}
```

```
/* rls_svc.c : partie serveur du logiciel de listing de répertoire
                version RPC niveau haut
*/

#include <sys/types.h>
#include <dirent.h>
#include <netconfig.h> /* le fichier supplémentaire à inclure */
#include <rpc/rpc.h>
#include "rls.h"

extern bool_t xdr_dir();




main(argc,argv)
int argc;
char *argv[];
{

SVCXPRT *transp;
```

```
struct netconfig *nconf;
char *netid;
void read_dir();

if(argc!=2) {
        fprintf(stderr,"Usage: %s netid\n",argv[0]);
        exit(1);
}

/* récupération du nom du protocole */

netid=argv[1];

/* remplissage de la structure netconfig */

if((nconf=getnetconfigent(netid))==(struct netconfig *)NULL) {
        fprintf(stderr,"Mauvais netid\n");
        exit(2);
}

/* création du serveur */

if((transp=svc_tp_create(read_dir,DIRPROG,DIRVERS,nconf))==(SVCXPRT  *)NULL)
{
        fprintf(stderr,"%s: impossible de creer le service %s\n",argv[0],ne-
tid);
        exit(3);
}

/* libération de la structure netconfig */

freenetconfigent(nconf);

/* lancemant du serveur */

svc_run();
}
```

```
/* read_dir.c : liste un directory */


static void read_dir(rqstp,transp)
struct svc_req *rqstp;
SVCXPRT *transp;

{
/* identique à la version précédente */
}
```

Le niveau expert

Ce niveau permet la sélection du transport comme le niveau intermédiaire, avec un comtrôle des détails des structures CLIENT et SVCXPRT. Ici, on se rapproche des fonctions TLI (voir chapitre 2). Les routines de ce niveau qui permettent la création du client et du serveur sont `clnt_tli_create()` et `svc_tli_create()`. Pour des détails sur l'utilisation de ces routines, reportez vous au manuel UNIX (`svc_tli_create(3N)` et `clnt_tli_create(3N)`).

Le niveau bas

Les routines de ce niveau autorise un contrôle très fin des options. Toutes les fonctions des niveaux supérieurs sont basées sur celles de celui-ci.

Ces routines créent les structures de données internes, la gestion des tampons...Des exemples de ces routines sont `clnt_vc_create()` et `clnt_dg_create()`. Les routines de ce niveau sont rarement utilisées.

Le niveau «local»

En plus de ces différentes manières de programmer les RPC, il existe des fonctions très utiles au déboguage d'un client-serveur: `svc_raw_create()` et `clnt_raw_create()`.
Ces fonctions ne font pas réellement appel au réseau, elles ne servent qu'à permettre des tests d'une application où le client et le serveur sont sur la même machine. Le programme suivant est un exemple de l'utilisation de ces routines.

```
/* rls.c : utilisation des routines locales de création de client et serveur
pour l'application le listing de répertoires.
*/

#include <stdio.h>
#include "/usr/ucbinclude/strings.h"
#include <rpc/rpc.h>
#include <rpc/raw.h> /* à inclure pour les déclarations de svc_raw_create()
et clnt_raw_create() */
#include "rls.h"
#include <dirent.h>

extern bool_t xdr_dir();

main(argc,argv)
int argc;
char *argv[];
{
char dir[DIR_SIZE];

CLIENT *cl; /* la structure client */
SVCXPRT *svc; /* la structure serveur */
```

```
void read_dir(); /* déclaration de read_dir() qui joue le rôle du serveur */

struct timeval time_out = { 0,0 }; /* ce time_out est nécessaire pour
l'appelde la procédure «distante», mais n'est pas utilisée */

if(argc==2) strcpy(dir,argv[1]);
else {
fprintf(stderr,"usage: %s path\n",argv[0]);
exit(1);
}

/* création du serveur, sans parametres. Celui-ci doit toujours être créé
avant le client. */

svc=svc_raw_create();
if(svc==(SVCXPRT *)NULL) {
fprintf(stderr,"Impossible de creer le serveur\n");
exit(2);
}

/* enregistrement du serveur. La structure netconfig étant nulle, le serveur
ne sera pas réellement enregistrée par rpcbind. */

svc_reg(svc,DIRPROG,DIRVERS,read_dir,(struct netconfig *)NULL);

/* création du client */

cl=clnt_raw_create(DIRPROG,DIRVERS);

if(cl==(CLIENT *)NULL) {
fprintf(stderr,"Impossible de creer le client \n");
exit(4);
}

/* appel de la procédure distante (read_dir()) */

if(clnt_call(cl,DIRVERS,xdr_dir,dir,xdr_dir,dir,time_out)!=RPC_SUCCESS) {
clnt_perror(cl,"raw");
exit(5);
}

printf("resultat: %s\n",dir);

exit(0);

}

/* read_dir() joue le rôle du serveur. Elle est identique à la fonction
read_dir() du niveau intermédiaire */

static void read_dir(rqstp,transp)
struct svc_req *rqstp;
SVCXPRT *transp;
{
DIR *dirp;
struct dirent *d;
static char dir[8192];
```

```
svc_getargs(transp,xdr_dir,dir);

/* ouvrir le directory */

dirp=opendir(dir);
if(dirp==NULL)
if(!svc_sendreply(transp,xdr_dir,NULL)) {
svcerr_systemerr(transp);
}


/* recopie des fichiers dans le buffer dir */

dir[0]=›\0›;
while(d=readdir(dirp)) sprintf(dir,"%s%s\n",dir,d->d_name);

/* retourner le resultat */

closedir(dirp);

if(!svc_sendreply(transp,xdr_dir,dir)) {
svcerr_systemerr(transp);
}

}
```

La programmation avancée des RPC

En plus de ces différents niveaux de programmation, les RPC offrent diverses possibilités supplémentaires qui peuvent être utiles dans certaines applications:

- une alternative à `svc_run()` pour les applications effectuant un travail durant l'attente de connexions.
- une possibilité de broadcast.
- une possibilité d'effectuer des appels et de continuer sans attendre une réponse.
- une possibilité d'identification du client auprès du serveur.

L'alternative à svc_run()

Un serveur qui doit effectuer des opérations durant l'attente de connexions a deux possibilités :

- si l'opération doit se faire à date fixe, il peut armer SIGALARM. Sur réception du signal, il effectue ses traitement et retourne à `svc_run()`.
- au lieu d'appeler `svc_run()`, il est possible d'accéder à la file des requêtes par `svc_get_reqset()`.

Les RPC BroadCast

Un message peut être envoyé à tous les démons `rpcbind` d'un réseau. Tous les démons ayant le service concerné enregistré, feront appel à lui. Le client attend alors plusieurs réponses.

Ceci ne peut se faire qu'avec un protocole en mode sans connexion. L'exemple suivant montre l'utilisation des RPC broadcast. Il permet de connaitre les adresses des machines où est enregistré un service. Par exemple, pour nfs version 2, on lancera :

```
bcast nfs 2
```

```
/* bcast.c : exemple d'utilisation des RPC broadcast. */


#include <stdio.h>
#include <rpc/rpc.h>
#include <rpc/rpcent.h>
#include <netdir.h>


/* la fonction bcast_proc est appelée chaque fois que l'on reçoit une réponse
d'une machine.
  Elle affiche l'adresse de la machine connaissant le service.
*/

bool_t bcast_proc(res,t_addr,nconf)
void *res;
struct t_bind *t_addr;
struct netconfig *nconf;
{
register struct hostent *hp;
char *uaddr;

uaddr=taddr2uaddr(nconf,&t_addr->addr);
if(uaddr==(char *)NULL) {
	printf("reponse : inconnu\n");
}
else {
	printf("reponse : %s\n",uaddr);
	free(uaddr);
}
return(FALSE);
}


main(argc,argv)
int argc;
char *argv[];
{
enum clnt_stat rpc_stat;
u_long prognum, versnum;
struct rpcent *re;

if(argc!=3) {
	fprintf(stderr,"usage: %s prognum versnum\n",argv[0]);
```

```
        exit(1);
}

/* si le deuxième argument de la commande est un numéro de service, on le
prend, si c'est un nom, on recherche le numéro associé. */

if(isdigit(*argv[1])) prognum=atoi(argv[1]);
else {
re=getrpcbyname(argv[1]);
if(!re) {
fprintf(stderr,"Service RPC inconnu\n");
exit(2);
}
prognum=re->r_number;
}

versnum=atoi(argv[2]);

/* appel de rpc_broadcast. Ici, on n'a pas d'argument a passé ( d'où (char
*)NULL et xdr_void()).
On passe en argument à rpc_broadcast un pointeur sur la fonction à appeler
lorsqu'une réponse arrive.
*/

rpc_stat=rpc_broadcast(prognum,versnum,NULLPROC,xdr_void,
(char *)NULL,xdr_void,(char *)NULL,bcast_proc,NULL);

/* si le temps imparti pour l'arrivée des réponses est dépassé, on sort */

if(rpc_stat!=RPC_SUCCESS && rpc_stat!=RPC_TIMEDOUT) {
fprintf(stderr,"Broadcast rate: %s\n",clnt_sperrno(rpc_stat));
exit(3);
}

exit(0);

}
```

Le batch

Dans le principe de base des RPC, un client envoie un message à un serveur et attend une réponse. Le client est donc bloqué tant que le serveur n'a pas fini de traiter sa requête. Si le client n'a pas besoin d'un réponse du serveur, ce mode de fonctionnement devient rapidement inefficace.

Le batch permet donc à un client de placer une suite de requêtes dans une file et de continuer son éxecution. Ceci implique que:

- le serveur ne doit pas répondre aux messages intermédiaires
- on utilise un protocole fiable (TCP)
- on ne se bloque pas sur l'envoi (timeout=0)

Le fournisseur de transport est libre de gérer cette file de messages comme il le désire, par exemple en envoyant plusieurs messages en un seul write(), pour diminuer le nombre de communications.

Identification du client

En fonction du service délivré, il peut être nécessaire de procéder à une identification du client avant de satisfaire sa requête. Ceci peut être réalisé de différentes manières avec les RPC. Il en existe cinq de base:

- AUTH_NONE: identification par défaut, qui est en fait une absence d'dentification.
- AUTH_SYS: ressemblante au système d'identification des processus (anciennement AUTH_SYS)
- AUTH_SHORT: version simplifiée de la précédente
- AUTH_DES: basée sur les techniques de cryptage DES
- AUTH_KERB: identification de la forme KERBEROS.

L'identification sous RPC est un système ouvert, c'est à dire que le programmeur peut définir lui-même son mode d'identification.

Dans la structure `CLIENT` existe un membre `cl_auth`. `client_create()` lui associe la valeur renvoyée par `authnone_create()` (identification par défaut).

Si vous désirez changer ce style d'identification, vous devez d'abord le détruire par `auth_destroy(clnt->cl_auth)`.Lorsque le serveur reçoit une requête, celle-ci contient une structure `opaque_auth` définie comme suit:

```
struct opaque_auth {
    enum_t oa_flavor; * style d'identification */
    caddr_t oa_base;
    u_int oa_length;
};
```

Nous allons détailler l'identification AUTH_SYS pour voir l'utilisation de ce qui précède.

Après `auth_destroy(clnt->cl_auth)`, on déclare le nouveau mode d'identification par `clnt->cl_auth=authsys_create_default()`. Chaque appel RPC contiendra alors une structure `authsys_parms` telle que la suivante:

```
struct authsys_parms {
    u_long aup_time; /* date de création de
                        l'identification */
    char *aup_machname; /* nom de la machine
                        hôte du client */
    uid_t aup_uid; /* uid effectif du client */
    gid_t aup_gid; /* gid courant du client */
    .
    .
    .
};
```

Du coté serveur, la vérification de l'identité du client pourrrait se faire suivant le modèle ci-après.

```
struct svc_req *rqstp;

if(rqstp->rq_cred.oaflavor==AUTH_SYS) {
     uid=sys_cred->aup_uid; /* récupération de l'uid du client */
     /* ici, on vérifie les droits du client */

}
else
{
     /* erreur, mauvais mode d'identification */
}
```

Différences entre RPC SUNOS 4.1 et RPC SUNOS 5.2

La différence majeure entre ces deux versions est l'utilisation des TLI à la place des sockets. Ceci permet une certaine indépendance vis à vis du transport qui n'était pas possible avec les sockets.

Dans l'ancienne version, la liste des services était maintenue comme une liste de numéros de port où se situaient les programmes RPC. Avec rpcbind, la version du SUNOS 5.2 s'est affranchie des numéros de port fixes. L'adresse d'un service est enregistrée dans un format universel.

Tous les transports identifiés dans le fichier /etc/netconfig son maintenant utilisables, alors qu'avant seuls TCP et UDP l'étaient.

Rpcgen a été beaucoup modifié avec l'ajout de nouvelles possibilités:

- les procédures distantes suportent pluseurs arguments alors qu'il fallait en faire une structure sur la verson précédente (voir chapitre 5)
- des fichiers contenant l'ossature d'un client et d'un serveur sont générés.
- les arguments des procédures distantes peuvent être passés par valeurs.

Les fonctionnalités des RPC, qui étaient incluses dans la librairie `libc`, sont maintenant dans la librairie `libnsl` (nécessaire à l'édition de liens).

5.RPCGEN

Un exemple d'utilisation

Le logiciel rpcgen simplifie la programmation des rpc par la génération automatique de certaines parties du code nécéssaire.

Ce code est généré en suivant les spécifications décidées par le programmeur et écrites par lui en RPCL (Remote procedure Call Language). Ce langage, similaire à C (surtout dans la dernière version des RPC), permet d'indiquer de façon précise les caractéristiques des procédurres distantes. Ces spécifications sont incluses dans un fichier ayant le suffixe x.

exemple :

pour `rls`, on écrira un fichier `rls.x` qui aura la forme suivante

```
program DIRPROG {
     version DIRVERS {
          string read_dir(string)=1;
     }=1;
}=0x20000001;
```

Cette déclaration permet de donner les numéros relatifs au programme, version et procédure, ainsi que l'interface de la procédure distante.

Par `rpcgen -a rls.x`, on génère les fichiers suivants:

- `rls.h` qui contient les déclarations nécessaires au client et au serveur.
- `rls_clnt.c` où sont rassemblées les routines de communication du client
- `rls_client.c` est une ossature pour le programme client
- `rls_xdr.c` où sont rassemblées toutes les routines de conversion des types au format réseau et vice-versa
- `rls_svc.c` contient les routines de communication du serveur
- `rls_server.c` est une ossature pour les procédures distantes
- `makefile`.

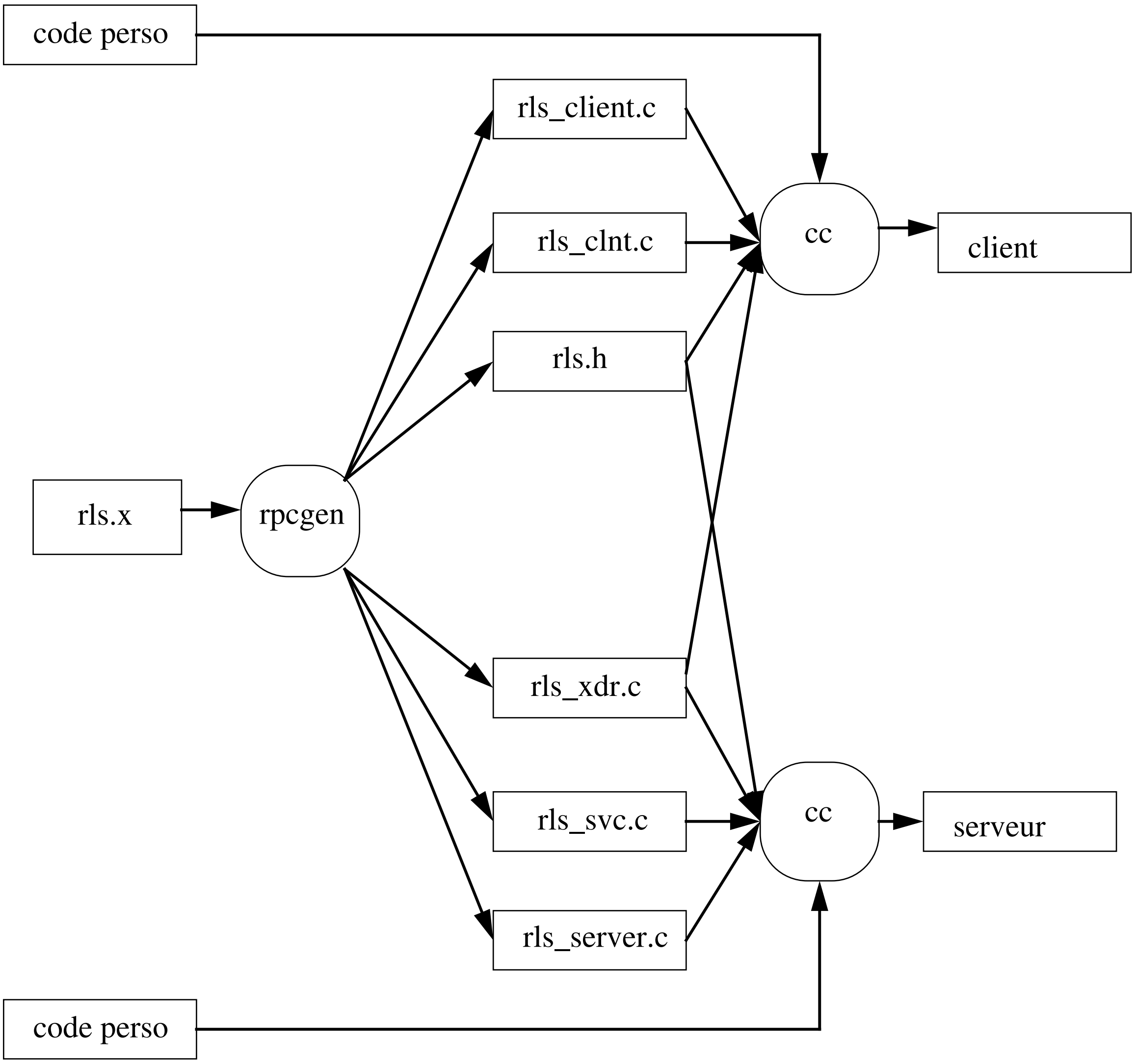

figure 5.1 : génération d'un client-serveur avec rpcgen

Par exemple, pour `rls.x`, on génère les fichiers suivants

```
/*
 * rls.h généré par rpcgen.
*/

#ifndef _RLS_H_RPCGEN
#define _RLS_H_RPCGEN

#include <rpc/rpc.h>

#define DIRPROG ((unsigned long)(0x20000001))
#define DIRVERS ((unsigned long)(1))
#define read_dir ((unsigned long)(1))
extern  char ** read_dir_1();
extern int dirprog_1_freeresult();

#endif /* !_RLS_H_RPCGEN */
```

```
/*
* rls_clnt.c généré par rpcgen. Contient la routine read_dir_1
* qui est l'appel à la procédure distante.
 */

#include «rls.h»

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

char **
read_dir_1(argp, clnt)
char **argp;
CLIENT *clnt;
{
static char *clnt_res;

memset((char *)&clnt_res, 0, sizeof (clnt_res));
if (clnt_call(clnt, read_dir,(xdrproc_t) xdr_wrapstring, (caddr_t) argp,
       (xdrproc_t)xdr_wrapstring,(caddr_t)&clnt_res,TIMEOUT)!=RPC_SUCCESS)
{
       return (NULL);
}
return (&clnt_res);
}
```

```
/*
 * rls_client.c généré par rpcgen. Contient un exemple d'appel à
* read_dir_1 (dans rls_clnt.c). Ce fichier doit être utilisé comme un
* exemple.
*/

#include "rls.h"


void
dirprog_1(host)
char *host;
{
CLIENT *clnt;
char * *result_1;
char * read_dir_1_arg;

#ifndef DEBUG
clnt = clnt_create(host, DIRPROG, DIRVERS, "netpath");
if (clnt == (CLIENT *) NULL) {
       clnt_pcreateerror(host);
       exit(1);
}
#endif /* DEBUG */


/* C'EST ICI QUE SE FAIT L'APPEL A LA PROCEDURE DISTANTE.
lE PROGRAMME UTILISATEUR DEVRA EFFECTUER DES APPELS SIMILAIRES
*/
result_1 = read_dir_1(&read_dir_1_arg, clnt);
if (result_1 == (char **) NULL) {
       clnt_perror(clnt, "call failed");
}
#ifndef DEBUG
clnt_destroy(clnt);
#endif /* DEBUG */
}


main(argc, argv)
int argc;
char *argv[];
{
char *host;

if (argc < 2) {
       printf("usage:  %s server_host\n", argv[0]);
       exit(1);
}
host = argv[1];
dirprog_1(host);
}
```

```
/*
 * rls_svc.c généré par rpcgen. Contient les routines de communication
* du serveur.
* Ce fichier ne devrait pas être édité.
*/

#include "rls.h"
#include <stdio.h>
#include <stdlib.h> /* getenv, exit */
#include <signal.h>
#include <sys/types.h>
#include <memory.h>
#include <stropts.h>
#include <netconfig.h>
#include <sys/resource.h> /* rlimit */
#include <syslog.h>

#ifdef DEBUG
#define RPC_SVC_FG
#endif

#define _RPCSVC_CLOSEDOWN 120
static int _rpcpmstart; /* Started by a port monitor ? */
/* States a server can be in wrt request */

#define _IDLE 0
#define _SERVED 1
#define _SERVING 2

static int _rpcsvcstate = _IDLE; /* Set when a request is serviced */

static
void _msgout(msg)
char *msg;
{
#ifdef RPC_SVC_FG
if (_rpcpmstart)
       syslog(LOG_ERR, msg);
else
       (void) fprintf(stderr, "%s\n", msg);
#else
syslog(LOG_ERR, msg);
#endif
}

static void
closedown(sig)
int sig;
{
if (_rpcsvcstate == _IDLE) {
       extern fd_set svc_fdset;
       static int size;
       int i, openfd;
       struct t_info tinfo;
```

```
        if (!t_getinfo(0, &tinfo) && (tinfo.servtype == T_CLTS))
             exit(0);


        if (size == 0) {
             struct rlimit rl;

             rl.rlim_max = 0;
             getrlimit(RLIMIT_NOFILE, &rl);
             if ((size = rl.rlim_max) == 0) {
                  return;
             }
        }
        for (i = 0, openfd = 0; i < size && openfd < 2; i++)
             if (FD_ISSET(i, &svc_fdset))
                  openfd++;
             if (openfd <= 1)
                  exit(0);
}
if (_rpcsvcstate == _SERVED)
        _rpcsvcstate = _IDLE;

(void) signal(SIGALRM, (void(*)()) closedown);
(void) alarm(_RPCSVC_CLOSEDOWN/2);
}


static void
dirprog_1(rqstp, transp)
struct svc_req *rqstp;
register SVCXPRT *transp;
{
union {
        char *read_dir_1_arg;
} argument;
char *result;
bool_t (*xdr_argument)(), (*xdr_result)();
char *(*local)();

_rpcsvcstate = _SERVING;
switch (rqstp->rq_proc) {
        case NULLPROC:
             (void) svc_sendreply(transp, xdr_void,
             (char *)NULL);
             _rpcsvcstate = _SERVED;
             return;

        case read_dir:
             xdr_argument = xdr_wrapstring;
             xdr_result = xdr_wrapstring;
             local = (char *(*)()) read_dir_1;
             break;
```

```
        default:
             svcerr_noproc(transp);
             _rpcsvcstate = _SERVED;
             return;
}

(void) memset((char *)&argument, 0, sizeof (argument));
if (!svc_getargs(transp, xdr_argument, &argument)) {
        svcerr_decode(transp);
        _rpcsvcstate = _SERVED;
        return;
}
result = (*local)(&argument, rqstp);
if (result != NULL && !svc_sendreply(transp, xdr_result, result)) {
        svcerr_systemerr(transp);
}
if (!svc_freeargs(transp, xdr_argument, &argument)) {
        _msgout("unable to free arguments");
        exit(1);
}
_rpcsvcstate = _SERVED;
return;
}

main()
{
pid_t pid;
int i;
char mname[FMNAMESZ + 1];

if (!ioctl(0, I_LOOK, mname) &&
(!strcmp(mname, "sockmod") || !strcmp(mname, "timod"))) {
        char *netid;
        struct netconfig *nconf = NULL;
        SVCXPRT *transp;
        int pmclose;

        _rpcpmstart = 1;
        openlog("rls", LOG_PID, LOG_DAEMON);

        if ((netid = getenv("NLSPROVIDER")) == NULL) {
             /* started from inetd */
             pmclose = 1;
        } else {
             if ((nconf = getnetconfigent(netid)) == NULL)
                  _msgout("cannot get transport info");

             pmclose = (t_getstate(0) != T_DATAXFER);
        }
        if (strcmp(mname, "sockmod") == 0) {
             if (ioctl(0, I_POP, 0) || ioctl(0, I_PUSH, "timod")) {
                  _msgout("could not get the right module");
                  exit(1);
             }
        }
```

```
        if ((transp = svc_tli_create(0, nconf, NULL, 0, 0)) == NULL) {
             _msgout("cannot create server handle");
             exit(1);
        }
        if (nconf)
             freenetconfigent(nconf);
        if (!svc_reg(transp, DIRPROG, DIRVERS, dirprog_1, 0)) {
             _msgout("unable to register (DIRPROG, DIRVERS).");
             exit(1);
        }
        if (pmclose) {
             (void) signal(SIGALRM, (void(*)()) closedown);
             (void) alarm(_RPCSVC_CLOSEDOWN/2);
        }
        svc_run();
        exit(1);
        /* NOTREACHED */
} else {

        #ifndef RPC_SVC_FG
        int size;
        struct rlimit rl;
        pid = fork();
        if (pid < 0) {
             perror("cannot fork");
             exit(1);
        }
        if (pid)
             exit(0);
        rl.rlim_max = 0;
        getrlimit(RLIMIT_NOFILE, &rl);
        if ((size = rl.rlim_max) == 0)
             exit(1);
        for (i = 0; i < size; i++)
             (void) close(i);
        i = open("/dev/console", 2);
        (void) dup2(i, 1);
        (void) dup2(i, 2);
        setsid();
        openlog("rls", LOG_PID, LOG_DAEMON);
        #endif
}
if (!svc_create(dirprog_1, DIRPROG, DIRVERS, "netpath")) {
        _msgout("unable to create (DIRPROG, DIRVERS) for netpath.");
        exit(1);
}

svc_run();
_msgout("svc_run returned");
exit(1);
/* NOTREACHED */
}
```

Ce fichier contient tout le code nécessaire à gérer la communication, les erreurs, etc...Si vous utilisez `rpcgen`, c'est avant tout pour ne pas vous occuper de cette partie dont l'essentiel ne variera pas d'une application à l'autre. Le fichier `rls_server.c` qui suit est beaucoup plus intéressant.

```
/*
* rls_client.c généré par rpcgen, il constitue un guide pour la programmation
* du serveur
*/

#include «rls.h»

char **
read_dir_1(argp, rqstp)
char **argp;
struct svc_req *rqstp;
{
static char * result;

/*
* Insérez votre code ici.
*/

return (&result);
}
```

Dans la fonction `read_dir_1()` ci-dessus, il ne vous reste qu'à mettre votre code pour réaliser la lecture du répertoire et à mettre le résultat dans `result`.

Les options de rpcgen

rpcgen dispose de plusieurs directives préprocesseur telles que RPC_HDR, RPC_XDR,...qui permettent de personnaliser les fichiers créés par rpcgen. Ceci est complété par la possibilité de passer des lignes du fichier '.x' directement au fichiers générés en commençant les lignes par '%'.

Pour pouvoir déclarer les procédures distantes avec plusieurs arguments, on utilisera l'option -N de rpcgen. Sans celle-ci, les arguments, s'ils sont plusieurs, devront être intégrés à une structure.

Avec l'option -a, on peut donc généré tous les fichiers nécessaires, mais on peut choisir de n'en générer que certains:

- -Sc ossature client
- -Ss ossature serveur
- -Sm le Makefile

Diverses autres options existent telles que la possibilité de générer du C ANSI (-C), ou d'utiliser les RPC basées sur les sockets (-b).

6. DES EXEMPLES DE CLIENTS-SERVEURS

Le serveur de sémaphores

Le but de ce programme est de permettre la synchronisation de processus situés sur des machines différentes. On bati donc un serveur qui offrira à ses clients un interface similaire à celle des sémaphores. Les processus distants pourront partager des sémaphores en utilisant les primitives suivantes:

- int s_creer() pour créer un sémaphore
- int s_attente() qui est l'implantation du wait()
- int s_libérer() implantation du signal()
- int s_detruire() pour détruire un sémaphore
- int s_compte() pour avoir la valeur d'un sémaphore
- int s_fixval() pour changer la valeur d'un sémaphore.

Le serveur crée un tableau de sémaphores UNIX et des structures qui permettent de se rappeler quels sont les processus bloqués sur les sémaphores. La complication vient du fait que c'est un fils du serveur qui va se bloquer sur le sémaphore et non réellemenr le client.

Toutes ces primitives font appel à des procédures distantes, définies dans le fichier suivant.

```
/* fichier semaphore.x. Définition des procédures distantes pour rpcgen */

typedef int sem;
%#define NB_SEMA 20

program SEMAPROG {
       version SEMAVERS {
            sem screer(int) = 1;
            int sattente(sem) =2;
            int sliberer(sem) = 3;
            int sdetruire(sem) = 4;
            int scompte(sem) = 5;
            int sfixval(sem,int) = 6;
       } = 1;
} = 0x20000199;
```

On obtient un fichier header intéressant à regarder:

```
/* semap.h
 * Please do not edit this file.
 * It was generated using rpcgen.
 */
```

```
#ifndef _SEMAP_H_RPCGEN
#define _SEMAP_H_RPCGEN

#include <rpc/rpc.h>

char *host;

typedef int sem;


/* les trois arguments de s_attente() sont placés dans une structure.
  Dans les versions précédentes de rpcgen, ceci devait être fait par le
  programmeur dans le fichier .x
  Ceci est fait pour toutes les procédures ayant plusieurs arguments.
  On définit de plus un nouveau type pour chaque structure (question de
rapidité d'écriture).
*/

struct sattente_1_argument {
       sem arg1;
       int arg2;
       int arg3;
};

typedef struct sattente_1_argument sattente_1_argument;

struct sfixval_1_argument {
       sem arg1;
       int arg2;
};
typedef struct sfixval_1_argument sfixval_1_argument;

/* Déclaration des fonctions, le '_1' à la fin des noms des procédures
signifie qu'elles appartiennent à la version 1 du programme
*/

#define SEMAPROG ((unsigned long)(0x20000199))
#define SEMAVERS ((unsigned long)(1))
#define screer ((unsigned long)(1))
extern  sem * screer_1();
#define sattente ((unsigned long)(2))
extern  int * sattente_1();
#define sliberer ((unsigned long)(3))
extern  int * sliberer_1();
#define sdetruire ((unsigned long)(4))
extern  int * sdetruire_1();
#define scompte ((unsigned long)(5))
extern  int * scompte_1();
#define sfixval ((unsigned long)(6))
extern  int * sfixval_1();
extern int semaprog_1_freeresult();


/* the xdr functions */
extern bool_t xdr_sem();
extern bool_t xdr_sfixval_1_argument();
extern bool_t xdr_sattente_1_argument();
```

```
#endif /* !_SEMAP_H_RPCGEN */
```

Dans ce fichier, trois routines xdr sont déclarées et définies dans le fichier semap_xdr.c.

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "semap.h"

bool_t
xdr_sattente_1_argument(xdrs,objp)
register XDR *xdrs;
sattente_1_argument *objp;
{
if(!xdr_sem(xdrs,&objp->arg1)) return(FALSE);
if(!xdr_int(xdrs,&objp->arg2)) return(FALSE);
if(!xdr_int(xdrs,&objp->arg3)) return(FALSE);

return(TRUE);
}

bool_t
xdr_sem(xdrs, objp)
register XDR *xdrs;
sem *objp;
{

register long *buf;

if (!xdr_int(xdrs, objp)) return (FALSE);
return (TRUE);
}
#define NB_SEMA 5

bool_t
xdr_sfixval_1_argument(xdrs, objp)
register XDR *xdrs;
sfixval_1_argument *objp;
{

if (!xdr_sem(xdrs, &objp->arg1)) return (FALSE);
if (!xdr_int(xdrs, &objp->arg2)) return (FALSE);
return (TRUE);
}
```

Les deux fichiers suivants représentent les parties communication du client et du serveur. On doit les modifier pour que le serveur initialise les sémaphores et les structures associées lorsqu'il est lancé.

```
/* semap_clnt.c
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "semap.h"

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 2500, 0 };

sem *
screer_1(arg1, clnt)
int arg1;
CLIENT *clnt;
{
static sem clnt_res;

memset((char *)&clnt_res, 0, sizeof (clnt_res));
if (clnt_call(clnt, screer,(xdrproc_t) xdr_int, (caddr_t) &arg1,
       (xdrproc_t) xdr_sem, (caddr_t) &clnt_res,TIMEOUT) != RPC_SUCCESS) {
            return (NULL);
}
return (&clnt_res);
}

int *
sattente_1(arg1,arg2,arg3, clnt)
sem arg1;
int arg2;
int arg3;
CLIENT *clnt;
{
sattente_1_argument arg;
static int clnt_res;

arg.arg1=arg1;
arg.arg2=arg2;
arg.arg3=arg3;

memset((char *)&clnt_res, 0, sizeof (clnt_res));
if (clnt_call(clnt, sattente,(xdrproc_t) xdr_sattente_1_argument,
       (caddr_t)&arg,(xdrproc_t)xdr_int,(caddr_t)&clnt_res,TIMEOUT)!=
       RPC_SUCCESS) {
            return (NULL);
}
return (&clnt_res);
}
```

```
int *
sliberer_1(arg1, clnt)
sem arg1;
CLIENT *clnt;
{
static int clnt_res;

memset((char *)&clnt_res, 0, sizeof (clnt_res));
if (clnt_call(clnt, sliberer,(xdrproc_t) xdr_sem, (caddr_t) &arg1,
(xdrproc_t) xdr_int, (caddr_t) &clnt_res,TIMEOUT) != RPC_SUCCESS) {
       return (NULL);
}
return (&clnt_res);
}

int *
sdetruire_1(arg1, clnt)
sem arg1;
CLIENT *clnt;
{
static int clnt_res;

memset((char *)&clnt_res, 0, sizeof (clnt_res));
if (clnt_call(clnt, sdetruire,(xdrproc_t) xdr_sem, (caddr_t) &arg1,
(xdrproc_t) xdr_int, (caddr_t) &clnt_res,TIMEOUT) != RPC_SUCCESS) {
       return (NULL);
}
return (&clnt_res);
}

int *
scompte_1(arg1, clnt)
sem arg1;
CLIENT *clnt;
{
static int clnt_res;

memset((char *)&clnt_res, 0, sizeof (clnt_res));
if (clnt_call(clnt, scompte,(xdrproc_t) xdr_sem, (caddr_t) &arg1,
(xdrproc_t) xdr_int, (caddr_t) &clnt_res,TIMEOUT) != RPC_SUCCESS) {
       return (NULL);
}
return (&clnt_res);
}

int *
sfixval_1(arg1, arg2, clnt)
sem arg1;
int arg2;
CLIENT *clnt;
{
sfixval_1_argument arg;
static int clnt_res;

memset((char *)&clnt_res, 0, sizeof (clnt_res));
arg.arg1 = arg1;
arg.arg2 = arg2;
if(clnt_call(clnt,sfixval,(xdrproc_t)xdr_sfixval_1_argument,(caddr_t) &arg,
```

```
(xdrproc_t) xdr_int, (caddr_t) &clnt_res,TIMEOUT) != RPC_SUCCESS) {
       return (NULL);
}
return (&clnt_res);
}


/* semap_svc.c
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "semap_svc.h"
#include <stdio.h>
#include <stdlib.h> /* getenv, exit */
#include <signal.h>
#include <sys/types.h>
#include <memory.h>
#include <stropts.h>
#include <netconfig.h>
#include <sys/resource.h> /* rlimit */
#include <syslog.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#ifdef DEBUG
#define RPC_SVC_FG
#endif

#define _RPCSVC_CLOSEDOWN 12000
#define NB_SEMA 5
static int _rpcpmstart; /* Started by a port monitor ? */
/* States a server can be in wrt request */

#define _IDLE 0
#define _SERVED 1
#define _SERVING 2

static int _rpcsvcstate = _IDLE; /* Set when a request is serviced */
static struct sembuf op[1]={ 0,99,SEM_UNDO};
static
void _msgout(msg)
char *msg;
{
#ifdef RPC_SVC_FG
if (_rpcpmstart)
       syslog(LOG_ERR, msg);
else
       (void) fprintf(stderr, "%s\n", msg);
#else
       syslog(LOG_ERR, msg);
#endif
}

static void
closedown(sig)
int sig;
{
```

```
if (_rpcsvcstate == _IDLE) {
        extern fd_set svc_fdset;
        static int size;
        int i, openfd;
        struct t_info tinfo;

        if (!t_getinfo(0, &tinfo) && (tinfo.servtype == T_CLTS))
             exit(0);
        if (size == 0) {
             struct rlimit rl;

             rl.rlim_max = 0;
             getrlimit(RLIMIT_NOFILE, &rl);
             if ((size = rl.rlim_max) == 0) {
                  return;
             }
        }
        for (i = 0, openfd = 0; i < size && openfd < 2; i++)
             if (FD_ISSET(i, &svc_fdset)) openfd++;
        if (openfd <= 1) exit(0);
}
if (_rpcsvcstate == _SERVED) _rpcsvcstate = _IDLE;

(void) signal(SIGALRM, (void(*)()) closedown);
(void) alarm(_RPCSVC_CLOSEDOWN/2);
}

static sem *
_screer_1(argp, rqstp)
int  *argp;
struct svc_req *rqstp;
{
return (screer_1(*argp, rqstp));
}

static int *
_sattente_1(argp, rqstp)
sattente_1_argument  *argp;
struct svc_req *rqstp;
{
return (sattente_1(argp->arg1,argp->arg2,argp->arg3, rqstp));
}

static int *
_sliberer_1(argp, rqstp)
sem  *argp;
struct svc_req *rqstp;
{
return (sliberer_1(*argp, rqstp));
}

static int *
_sdetruire_1(argp, rqstp)
sem  *argp;
struct svc_req *rqstp;
{
return (sdetruire_1(*argp, rqstp));
}
```

```
static int *
_scompte_1(argp, rqstp)
sem  *argp;
struct svc_req *rqstp;
{
return (scompte_1(*argp, rqstp));
}

static int *
_sfixval_1(argp, rqstp)
sfixval_1_argument *argp;
struct svc_req *rqstp;
{
return (sfixval_1(argp->arg1, argp->arg2, rqstp));
}

static void
semaprog_1(rqstp, transp)
struct svc_req *rqstp;
register SVCXPRT *transp;
{
union {
int screer_1_arg;
sattente_1_argument sattente_1_arg;
sem sliberer_1_arg;
sem sdetruire_1_arg;
sem scompte_1_arg;
sfixval_1_argument sfixval_1_arg;
} argument;
char *result;
bool_t (*xdr_argument)(), (*xdr_result)();
char *(*local)();

att=0;
_rpcsvcstate = _SERVING;
switch (rqstp->rq_proc) {
       case NULLPROC:
            (void) svc_sendreply(transp, xdr_void,
            (char *)NULL);
            _rpcsvcstate = _SERVED;
            return;

       case screer:
            xdr_argument = xdr_int;
            xdr_result = xdr_sem;
            local = (char *(*)()) _screer_1;
            break;

       case sattente:
            xdr_argument = xdr_sattente_1_argument;
            xdr_result = xdr_int;
            local = (char *(*)()) _sattente_1;
            break;

       case sliberer:
            xdr_argument = xdr_sem;
            xdr_result = xdr_int;
```

```
            local = (char *(*)()) _sliberer_1;
            break;

        case sdetruire:
            xdr_argument = xdr_sem;
            xdr_result = xdr_int;
            local = (char *(*)()) _sdetruire_1;
            break;

        case scompte:
            xdr_argument = xdr_sem;
            xdr_result = xdr_int;
            local = (char *(*)()) _scompte_1;
            break;

        case sfixval:
            xdr_argument = xdr_sfixval_1_argument;
            xdr_result = xdr_int;
            local = (char *(*)()) _sfixval_1;
            break;

        default:
            svcerr_noproc(transp);
            _rpcsvcstate = _SERVED;
            return;
}
(void) memset((char *)&argument, 0, sizeof (argument));
if (!svc_getargs(transp, xdr_argument, &argument)) {
        svcerr_decode(transp);
        _rpcsvcstate = _SERVED;
        return;
}
if(att==0) {
        result = (*local)(&argument, rqstp);
        if (result != NULL && !svc_sendreply(transp, xdr_result, result)) {
            svcerr_systemerr(transp);
        }

        if (!svc_freeargs(transp, xdr_argument, &argument)) {
            _msgout("unable to free arguments");
            exit(1);
        }
}
if(att==1) exit(0);
_rpcsvcstate = _SERVED;
return;
}

main()
{
pid_t pid;
int i;
char mname[FMNAMESZ + 1];

signal(SIGCLD,SIG_IGN);
if((id=semget(CLE,NB_SEMA,0777|IPC_CREAT))<0) {
        printf("erreur creation semaphores\n");
        exit(1);
```

```
}

printf("id= %d\n",id);



if (!ioctl(0, I_LOOK, mname) &&
(!strcmp(mname, "sockmod") || !strcmp(mname, "timod"))) {
        char *netid;
        struct netconfig *nconf = NULL;
        SVCXPRT *transp;
        int pmclose;

        _rpcpmstart = 1;
        openlog("semap", LOG_PID, LOG_DAEMON);

        if ((netid = getenv(«NLSPROVIDER»)) == NULL) {
             /* started from inetd */
             pmclose = 1;
        } else {
             if ((nconf = getnetconfigent(netid)) == NULL)
                  _msgout("cannot get transport info");

             pmclose = (t_getstate(0) != T_DATAXFER);
        }
        if (strcmp(mname, "sockmod") == 0) {
             if (ioctl(0, I_POP, 0) || ioctl(0, I_PUSH, "timod")) {
                  _msgout("could not get the right module");
                  exit(1);
             }
        }
        if ((transp = svc_tli_create(0, nconf, NULL, 0, 0)) == NULL) {
             _msgout("cannot create server handle");
             exit(1);
        }
        if (nconf) freenetconfigent(nconf);
        if (!svc_reg(transp, SEMAPROG, SEMAVERS, semaprog_1, 0)) {
             _msgout("unable to register (SEMAPROG, SEMAVERS).");
             exit(1);
        }
        if (pmclose) {
             (void) signal(SIGALRM, (void(*)()) closedown);
             (void) alarm(_RPCSVC_CLOSEDOWN/2);
        }
        svc_run();
        exit(1);
        /* NOTREACHED */
}else {
        #ifndef RPC_SVC_FG
        int size;
        struct rlimit rl;
        pid = fork();
        if (pid < 0) {
             perror("cannot fork");
             exit(1);
        }
        if (pid) exit(0);
```

```
        rl.rlim_max = 0;
        getrlimit(RLIMIT_NOFILE, &rl);
        if ((size = rl.rlim_max) == 0) exit(1);
        for (i = 0; i < size; i++) (void) close(i);
        i = open("/dev/console", 2);
        (void) dup2(i, 1);
        (void) dup2(i, 2);
        setsid();
        openlog("semap", LOG_PID, LOG_DAEMON);
        #endif
}
if (!svc_create(semaprog_1, SEMAPROG, SEMAVERS, "netpath")) {
        _msgout("unable to create (SEMAPROG, SEMAVERS) for netpath.");
        exit(1);
}

svc_run();
_msgout("svc_run returned");
exit(1);
/* NOT REACHED */
}
```

La partie suivante est celle qui comporte le code écrit par le programmeur. Elle contient le code des procédures distantes et l'interface coté client qui permet de les appeler.

Le serveur gère un tableau de sémaphores UNIX, qu'il va utiliser pour satisfaire les demandes des clients.

Lorsqu'un client demande la création d'un sémaphore, le serveur en réserve un dans son tableau de sémaphore et retourne son numéro au client. Celui-ci joue le rôle d'identificateur qui sera nécessaire pour toute opération sur ce sémaphore.

Le `wait()` et le `signal()` sont réalisés de la manière suivante: quand un client fait `wait()` sur un sémaphore, le serveur crée un processus fils qui décrémente le sémaphore. Si ce processus se bloque, le client sera ui aussi bloqué en attente de la réponse du serveur. Lorsqu'un client fait `signal()` sur un sémaphore, le serveur incrémente le sémaphore, ce qui débloque un processus fils du serveur en attente sur ce sémaphore (s'il y en a) et donc le client associé.

```
/* semap_svc.h: reprend les grandes lignes de semap.h utilisé pour le
client. On y a rajouté les structures nécessaires à la gestion des sémapho-
res et de processus client en attente.
*/

#ifndef _SEMAP_H_RPCGEN
#define _SEMAP_H_RPCGEN

#include <rpc/rpc.h>

typedef int sem;
#define NB_SEMA 5
int id;
int att;

struct pp {
int p;
```

```
int u;
};
typedef struct pp pp;




struct sema {
int utilise;
int valeur;
int nb_procb;
pp procb[30];
};
typedef struct sema sema;

sema tab[NB_SEMA];

#define CLE ((key_t) 12345L)

struct sattente_1_argument {
sem arg1;
int arg2;
int arg3;
};
typedef struct sattente_1_argument sattente_1_argument;

struct sfixval_1_argument {
sem arg1;
int arg2;
};
typedef struct sfixval_1_argument sfixval_1_argument;

#define SEMAPROG ((unsigned long)(0x20000199))
#define SEMAVERS ((unsigned long)(1))
#define screer ((unsigned long)(1))
extern  sem * screer_1();
#define sattente ((unsigned long)(2))
extern  int * sattente_1();
#define sliberer ((unsigned long)(3))
extern  int * sliberer_1();
#define sdetruire ((unsigned long)(4))
extern  int * sdetruire_1();
#define scompte ((unsigned long)(5))
extern  int * scompte_1();
#define sfixval ((unsigned long)(6))
extern  int * sfixval_1();
extern int semaprog_1_freeresult();

/* the xdr functions */
extern bool_t xdr_sem();
extern bool_t xdr_sfixval_1_argument();
extern bool_t xdr_sattente_1_argument();

#endif /* !_SEMAP_H_RPCGEN */
```

```
/* semap_server.c */

#include "semap_svc.h"
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define NB_SEMA 5
extern int errno;

/* création d'un sémaphore, vu du client */

sem *
screer_1(arg1, rqstp)
int arg1;
struct svc_req *rqstp;
{
static sem  result;
int i;
union semun {
       int val;
       struct semid_ds *buf;
       ushort *array;
} semctl_arg;

/* on recherche un sémaphore libre */

i=0;
while(tab[i].utilise==1 && i<NB_SEMA) i++;

/* il n'y en a plus */

if(i==NB_SEMA) result=-1;

/* il en reste */
else {
       tab[i].utilise=1; /* on le marque utilisé */
       tab[i].valeur=arg1; /* on l'initialise */
       semctl_arg.val=arg1;
       semctl(id,i,SETVAL,semctl_arg); /* on initialise le vrai sémaphore */
       result=i; /* on retourne son numéro */
}
return (&result);
}

/* primitive wait() */

int *
sattente_1(arg1,arg2,arg3, rqstp)
sem arg1;
int arg2;
int arg3;
```

```
struct svc_req *rqstp;
{
int i,deja;
static int  result;
struct sembuf op[1]={0,99,0}; /* opération wait() sur un sémaphore UNIX */
result=0;
if(tab[arg1].utilise==1) {

/* on regarde si le processus n'est pas déjà en attente */
	deja=0;
	for(i=0;i<tab[arg1].nb_procb;i++) {
	if(tab[arg1].procb[i].p==arg2 && tab[arg1].procb[i].u==arg3) deja=1;
	}
	if(tab[arg1].valeur>0) {
	     tab[arg1].valeur--;
	     result=0;
	     deja=1;
	     op[0].sem_num=arg1;
	     op[0].sem_op=-1;
	     semop(id,&op[0],1); /* wait() sur sémaphore */
	     return(&result);
	}
	if(deja==0) {
	     tab[arg1].valeur--;
	     tab[arg1].nb_procb++;
	     tab[arg1].procb[tab[arg1].nb_procb-1].p=arg2; /* on met le
client en attente */
	     tab[arg1].procb[tab[arg1].nb_procb-1].u=arg3;
	     if(fork()==0) {
	          att=1;
	          op[0].sem_num=arg1;
	          op[0].sem_op=-1;
	          semop(id,&op[0],1);
	          return (&result);
	     }
	else return(NULL);
}
else {
	return(NULL);
}
}
else {
	result=-1;
	return(&result);
}
}

/* primitive signal() */

int *
sliberer_1(arg1, rqstp)
sem arg1;
struct svc_req *rqstp;
{
int ncount;
union semun {
	int val;
	struct semid_ds *buf;
```

```
        ushort *array;
} arg;
static int  result;
struct sembuf op[1] = { 0,99,0}; /* opération signal() UNIX */


/* si un processus était en attente, on l'enlève de la liste des processus
en attente
*/

ncount=semctl(id,arg1,GETNCNT,arg);
if(ncount>0) {
        tab[arg1].nb_procb--;
}
op[0].sem_num=arg1;
op[0].sem_op=1;
semop(id,&op[0],1);
tab[arg1].valeur++;
result=tab[arg1].procb[tab[arg1].nb_procb].p;
return (&result);
}

/* libération d'un sémaphore */

int *
sdetruire_1(arg1, rqstp)
sem arg1;
struct svc_req *rqstp;
{
static int  result;

if(tab[arg1].utilise==0) result=-1;
else {
        tab[arg1].utilise=0;
        result=0;
}
return (&result);
}

/* récupération de la valeur d'un sémaphore */

int *
scompte_1(arg1, rqstp)
sem arg1;
struct svc_req *rqstp;
{
static int  result;
union semun {
        int val;
        struct semid_ds *buf;
        ushort *array;
} semctl_arg;


result=semctl(id,(int)arg1,GETVAL,semctl_arg);

return (&result);
}
```

```
/* changement de la valeur d'un sémaphore */

int *
sfixval_1(arg1, arg2, rqstp)
sem arg1;
int arg2;
struct svc_req *rqstp;
{
static int  result;
union semun {
       int val;
       struct semid_ds *buf;
       u_short *array;
} semctl_arg;
semctl_arg.val=arg2;
semctl(id,arg1,SETVAL,semctl_arg);
result=0;
tab[arg1].valeur=arg2;

return (&result);
}


/* Ce fichier contient les routines client associees au fonctions du serveur


- int s_creer(int val) : permet la creation d›un semaphore
       val : valeur initiale du semaphore
       valeur de retour : identificateur du semaphore
- int s_attente(int id) : fonction wait
       id : identificateur du semaphore
- int s_liberer(int id) : fonction signal
       id : identificateur du semaphore
- int s_detruire(int id) : detruire un semaphore
       id : identificateur du semaphore
- int s_compte(int id) : retourne la valeur du semaphore
       id : identificateur du semaphore
       valeur de retour : valeur du semaphore
- int s_fixval(int id, int val) : permat de fixer la valeur d'un semaphore
       id : identificateur du semaphore
       val : valeur a donner au semaphore

Toutes ces fonctions retourne -1 en cas d'erreur.

*/

#include "semap.h"

int s_creer(val)
int val;
{
```

```
CLIENT *clnt;
int *result;

/* Ouverture d›une connexion avec le serveur */

clnt = clnt_create(host, SEMAPROG, SEMAVERS, "netpath");
if (clnt == (CLIENT *) NULL) {
       *result=-1;
       return(*result);
}

/* appel de la procedure distante screer_1 */

result = screer_1(val, clnt);

if (result == (sem *) NULL) {
       *result=-1;
}

/* fermeture de la connexion */

clnt_destroy(clnt);

return(*result);

}

int s_attente(id)
int id;
{
CLIENT *clnt;
int *result;

clnt = clnt_create(host, SEMAPROG, SEMAVERS, "netpath");
if (clnt == (CLIENT *) NULL) {
       *result=-1;
       return(*result);
}

result = sattente_1(id,getpid(),getuid(), clnt);

if (result == (int *) NULL) {
       *result=-1;
}

clnt_destroy(clnt);

return(*result);

}

int s_liberer(id)
int id;
{
CLIENT *clnt;
int *result;

clnt = clnt_create(host, SEMAPROG, SEMAVERS, "netpath");
```

```
if (clnt == (CLIENT *) NULL) {
        *result=-1;
        return(*result);
}

result = sliberer_1(id, clnt);
if (result == (int *) NULL) {
        *result=-1;
}

clnt_destroy(clnt);
return(*result);


}

int s_detruire(id)
int id;
{
CLIENT *clnt;
int *result;

clnt = clnt_create(host, SEMAPROG, SEMAVERS, "netpath");
if (clnt == (CLIENT *) NULL) {
        *result=-1;
        return(*result);
}

result = sdetruire_1(id, clnt);
if (result == (int *) NULL) {
        *result=-1;
}

clnt_destroy(clnt);
return(*result);

}


int s_compte(id)
int id;
{

CLIENT *clnt;
int *result;

clnt = clnt_create(host, SEMAPROG, SEMAVERS, "netpath");
if (clnt == (CLIENT *) NULL) {
        *result=-1;
        return(*result);
}

result = scompte_1(id, clnt);
if (result == (int *) NULL) {
        *result=-1;
}

clnt_destroy(clnt);
```

```
return(*result);

}



int s_fixval(id,val)
int id;
int val;
{

CLIENT *clnt;
int *result;

clnt = clnt_create(host, SEMAPROG, SEMAVERS, "netpath");
if (clnt == (CLIENT *) NULL) {
       *result=-1;
       return(*result);
}

result = sfixval_1(id, val, clnt);
if (result == (int *) NULL) {
       *result=-1;
}
clnt_destroy(clnt);
return(*result);

}
```

Voici un exemple d'utilisation des primitives:

```
/*

       exemple d'utlisation du serveur de sémaphores

*/

#include "semap.h"

main(argc, argv)
int argc;
char *argv[];
{
int sem;

if (argc < 2) {
       printf("usage:  %s server_host\n", argv[0]);
       exit(1);
}
host = argv[1];

printf("Je cree un semaphore de valeur 1\n");

if((sem=s_creer(1))==-1) exit(1);

printf("Je regarde sa valeur : %d\n",s_compte(sem));

printf("je fais s_attente sur le semaphore %d\n",sem);
```

```
s_attente(sem);

printf("Je regarde la valeur du semaphore : %d\n",s_compte(sem));

if(fork()==0) {
       printf("je suis le fils, je fais sleep(10)\n");
       sleep(10);
       printf("Je suis le fils, je me reveille et fais signal sur le sema-
phore\n");
       s_liberer(sem);
       printf("val sem: %d\n",s_compte(sem));

       exit(0);
}
else {
       printf("Je suis le pere, je me bloque\n");
       s_attente(sem);
       printf("Je suis le pere, je suis libere\n");

}

s_detruire(sem);

}
```

L'échiquier distribué

Cet exemple peut être consulté dans l'arborescence des exemples.

L'idée est de réaliser un serveur qui permettent à des joueurs sur différentes machines de partager un échiquier et donc de pouvoir s'affronter.

L'ossature a été générée par rpcgen grâce au fichier echec.x suivant:

```
/* echec.x : le jeu d'echecs en client-serveur permettant a 2 joueurs
  de s'affronter a distance.
*/


program ECHECPROG {
       version ECHECVERS {
            int connexion(string,string,string)=1;
            string recoit_coup(string,string)=2;
            int envoit_coup(string,string,string)=3;
            int deconnexion(string,string)=4;
       }=1;
}=0x20000076;
```

Dans le programme fournis, la déconnexion n'est pas implantée.

BIBLIOGRAPHIE

BLOOMER, John [1992], Power Programming with RPC (O'Reilly & Associates).

COMER D.E. et STEVENS D.L. [1993], Internetworking with TCP/IP Vol 3 : Client-Server Programming and Applications (PRENTICE HALL).

COMER Douglas [1992], TCP/IP : architecture, protocoles, applications (InterEditions).

DELAMARRE Gérard [1989], Dictionnaire des réseaux (collection TRANSPAC).

RIFFLET Jean-Marie [1991], La communication sous UNIX (McGRAW-HILL).

STEVENS, W.R. [1990], UNIX Network Programming (PRENTICE HALL).

TANENBAUM Andrew [1989], Réseaux : architectures, protocoles, applications (InterEditions).